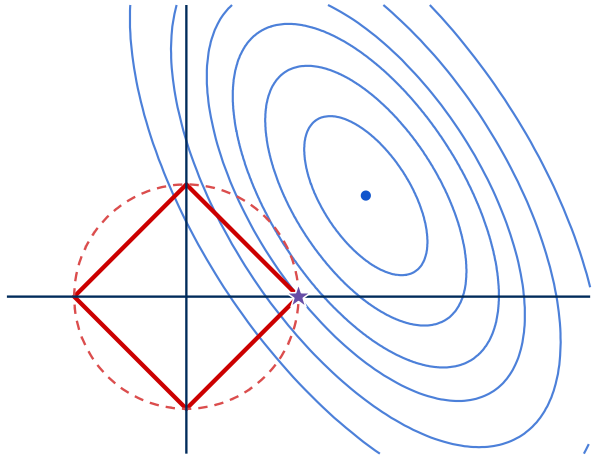


UNIT 02 · CHAPTER 6

# Optimization and Regularization

CSE 30124 · Introduction to Artificial Intelligence
William Theisen · University of Notre Dame

# Problems with OLS

Last chapter we found the best line through Notre Dame's games by writing down a loss (the **SSE**), picturing it as a bowl over the weights, and solving for the bottom of the bowl. That gave us **Ordinary Least Squares**, $\mathbf{w}^T = (X^T X)^{-1} X^T \mathbf{y}$, and a line worth about `9` points per `100` yards.

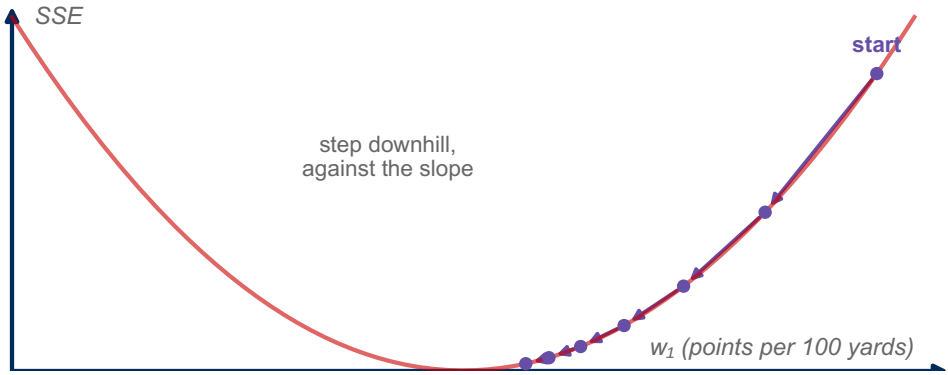
Unfortunately, the chapter ended on two problems. The first is the inverse. With `119` features $X^T X$ is a `119` by `119` matrix, and inverting it takes on the order of $119^3$ steps. Worse, when one feature can be built from the others (total yards is just pass yards plus rush yards) the inverse **doesn't exist at all**.

The second is **overfitting**. Our `degree-8` curve fit the `13` games from `2012` with an **MSE** of `15`, then missed the `2013` games with an **MSE** of `9,553`.

The first problem is about how we find our weights. The second is about which weights we might want in the first place.

# An iterative method

Instead of jumping straight to the bottom of the bowl, we can **walk downhill** a little at a time.




[6.1] one step of gradient descent

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \frac{\partial \mathcal{L}}{\partial \mathbf{w}}$$

Here $\mathbf{w}$ is the whole list of weights, $\partial\mathcal{L}/\partial\mathbf{w}$ is the same gradient we set to zero last chapter, and $\alpha$ is the **learning rate**, how big a step we take each time. Rather than solving for where the gradient is zero, we keep stepping against it until it is. This is called **gradient descent**, and notice that **there's no inverse anywhere**. Each step costs a couple of matrix multiplications, so a million features is slow but no longer impossible.

# The Learning Rate

Starting from weights of `0` on our five games, here's how close each learning rate gets after `100` steps. The bottom of the bowl ($w_0 = -5.2$, $w_1 = 9.0$) has an **SSE** of `4.8`.

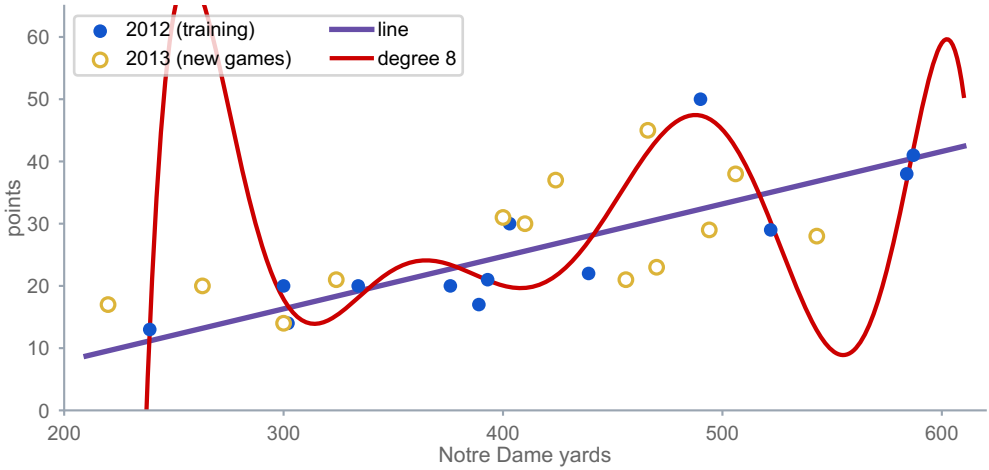
| $\alpha$ | $w_0$ | $w_1$ | SSE |
|---|---|---|---|
| 0.001 | 0.98 | 7.62 | 26.0 |
| 0.005 | −1.16 | 8.10 | 13.9 |
| 0.01 | −2.83 | 8.47 | 7.9 |

A bigger step gets there faster, until it doesn't. At $\alpha = 0.02$ each step jumps clean over the bottom of the bowl and lands higher up the other side, and every step after that makes it worse (by step `1,000` every weight is `nan`). For these five games anything above `0.0106` blows up. **Picking the learning rate is most of the work** in getting gradient descent to behave.

Unfortunately, gradient descent only fixes the cost. It walks to the bottom of the same bowl **OLS** solved for, so it inherits whatever is wrong with that bottom. Give it total, pass and rush yards and the bowl stretches into a long, nearly flat valley (total yards is the sum of the other two in all but one game). The bottom of that valley is at weights of `-64`, `+72` and `+74` points per `100` yards, and gradient descent would find the same weights, just very slowly.

# Overfitting

Those weights should look familiar. Last chapter's `degree-8` curve fit the `13` games from 2012 almost perfectly and then fell apart on 2013, and its weights were the same kind of thing, enormous and cancelling each other out.




**Linear regressions** are **prone to overfitting** because nothing in the **SSE** cares how big the weights get. If a weight of `1.9` million shaves a little more error off the training games, **OLS** will happily use it.

# Large Weights

Here are the biggest weights (leaving out the intercept) for each of last chapter's polynomials, fit to the 2012 games and tested on 2013:

| Model | Biggest weight | MSE on 2012 | MSE on 2013 |
|---|---|---|---|
| line (degree 1) | 8.4 | 39 | 56 |
| degree 2 | 5.9 | 39 | 56 |
| degree 4 | 243 | 35 | 65 |
| degree 8 | 1.9 million | 15 | 9,553 |

The weights and the 2013 error climb together. To bend through every 2012 game the `degree-8` curve needs each weight to almost exactly cancel the ones next to it, and that balance only holds where the 2012 games were. Our one 2013 game with fewer yards than any 2012 game (220) gets a prediction of `-330` points.

A model that needs **enormous, finely balanced weights** is almost always fitting the noise. So the obvious fix is to make big weights cost us something.

# Regularization: $L_1$ and $L_2$

To help **prevent overfitting** when using **linear regressions**, we can perform a technique called **regularization**.

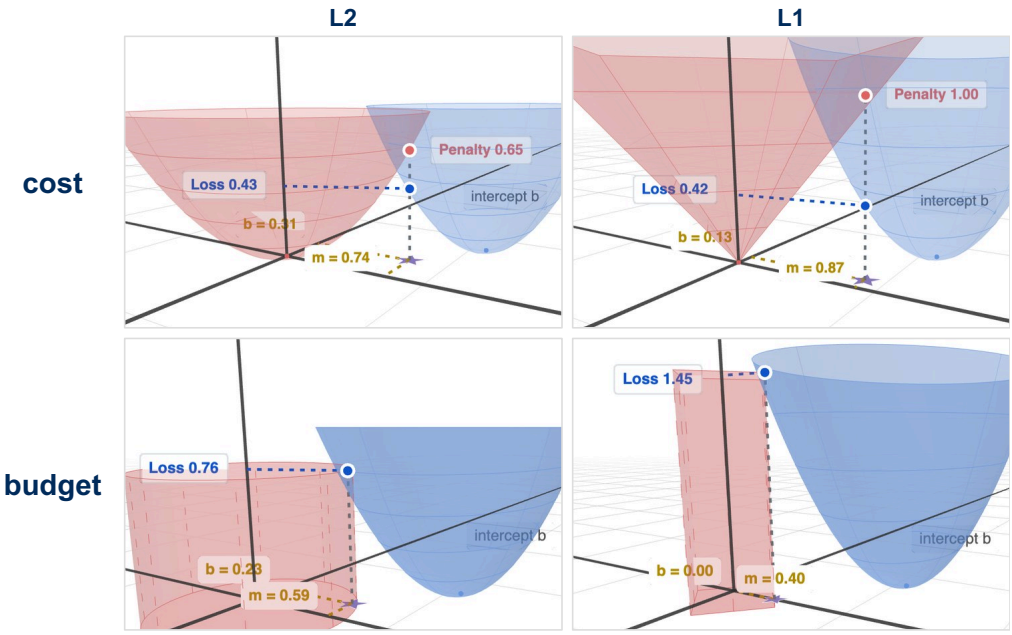
The goal of **regularization** is to **encourage models to learn the broader patterns** within the data rather than memorizing it.

**Regularization** does this by adding a second term to the loss that measures how big the weights are. There are two common ways to measure it, adding up the squared weights (called **L2**, or **ridge**) or adding up their absolute values (**L1**, or **LASSO**).

# Formalizations of regularization

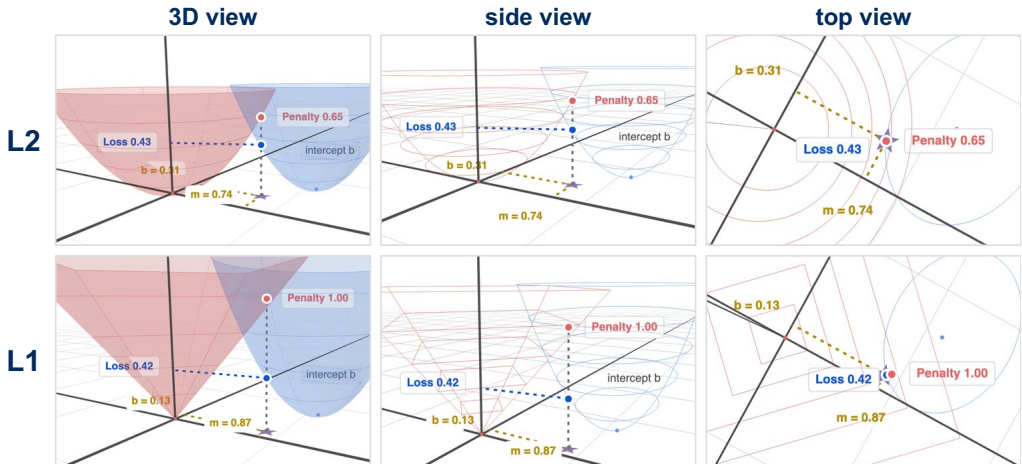
There are two ways to formalize regularization:

- As a cost (penalty) placed on the **weights**
- As a budget (constraint) for the **weights**



# Regularization as a cost

When we formalize **regularization** as a **cost**, we modify the **loss function** such that increasing the weights incurs a "cost".




We use an argument called $\lambda$ to determine how heavily the **cost** is weighed. A larger $\lambda$ increases the **cost**.

L2 as a cost

$$[6.2] \quad \mathcal{L}(w) = \text{SSE}(w) + \lambda \sum_j w_j^2$$

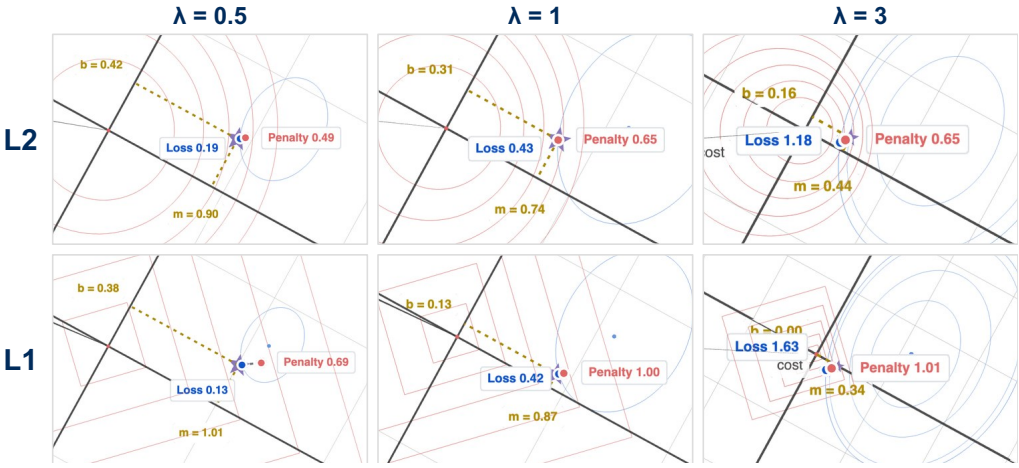
L1 as a cost

$$[6.3] \quad \mathcal{L}(w) = \text{SSE}(w) + \lambda \sum_j |w_j|$$

With $\lambda = 0$ both are just the SSE again, so OLS is the special case where we don't care at all how big the weights get.

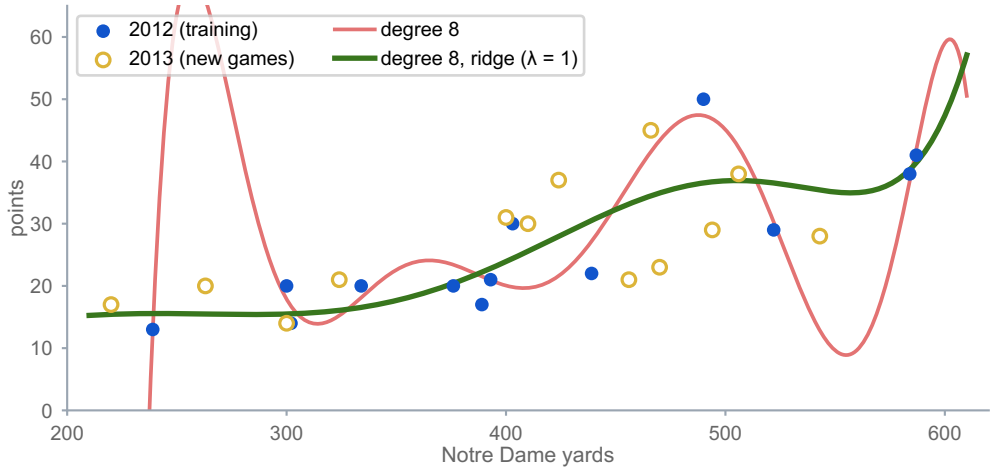
# Regularization as a cost, increasing λ

A larger λ increases the **cost** of our weights, typically resulting in the optimum being pulled closer to the origin.



# Ridge on the 2012 Season

Here is last chapter's `degree-8` polynomial fit to the 2012 games twice, once with plain **OLS** and once with ridge at $\lambda = 1$ (the intercept left out of the penalty):




| λ | Biggest weight | MSE on 2012 | MSE on 2013 |
|---|---|---|---|
| 0 (plain OLS) | 1.9 million | 15 | 9,553 |
| 1 | 2.3 | 32 | 55 |
| 100 | 0.13 | 34 | 57 |

The penalty costs us a little on 2012 and saves us almost everything on 2013. At $\lambda = 1$ the `degree-8` curve even does slightly better on 2013 than the plain line did (`55` against `56`), with nine weights instead of two. Push $\lambda$ up to `100` and the curve flattens back toward a line, so **$\lambda$ is a dial between variance and bias**.

# Ridge and the Inverse

Ridge also fixes our first problem, almost by accident. Take the gradient of the ridge loss, set it to zero like we did for **OLS**, and the only change is a $\lambda$ added down the diagonal:

[6.4] ridge, solved in one step

$$w^T = (X^T X + \lambda I)^{-1} X^T y$$

Adding any positive $\lambda$ to the diagonal **guarantees the inverse exists**, so the flat valley turns back into a bowl with one bottom. In practice we usually leave the intercept $w_0$ out of the penalty (there's no reason to push the average score toward `0`), so the top-left entry of `I` becomes a `0`.

Back to total, pass and rush yards from the learning rate page. Plain OLS gave us weights of `-64`, `+72` and `+74`. Ridge with $\lambda = 1$ gives **+5.9** on total yards, **+2.0** on pass yards and **+3.9** on rush yards. The predictions barely change (`100` passing yards is still worth about `8` points and `100` rushing yards about `10`), but now every weight is positive and none of them are absurd.

# Regularization as a budget

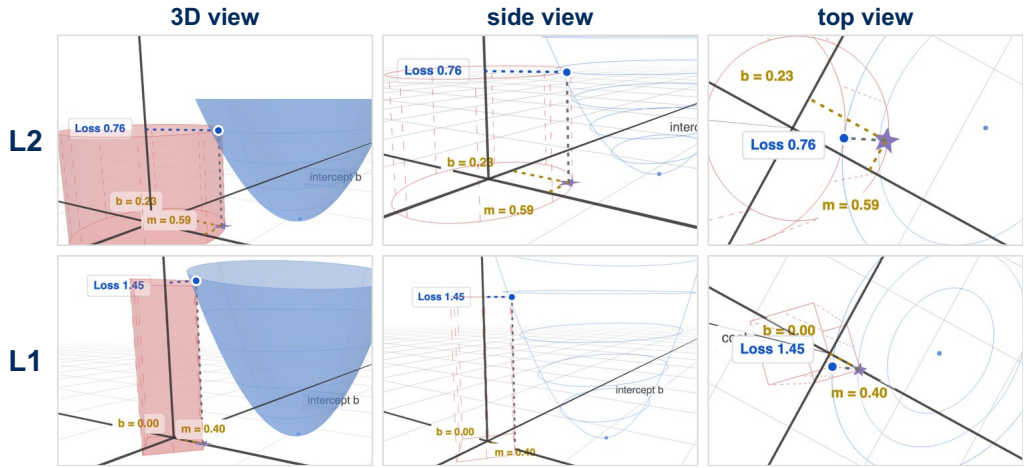
When we formalize **regularization** as a **budget**, we constrain our **loss function** such that the weights chosen must remain within "budget".

L2 as a budget

$$\min_{w} \; SSE(w) \quad \text{subject to} \quad \sum_{j} w_j^2 \leq t$$

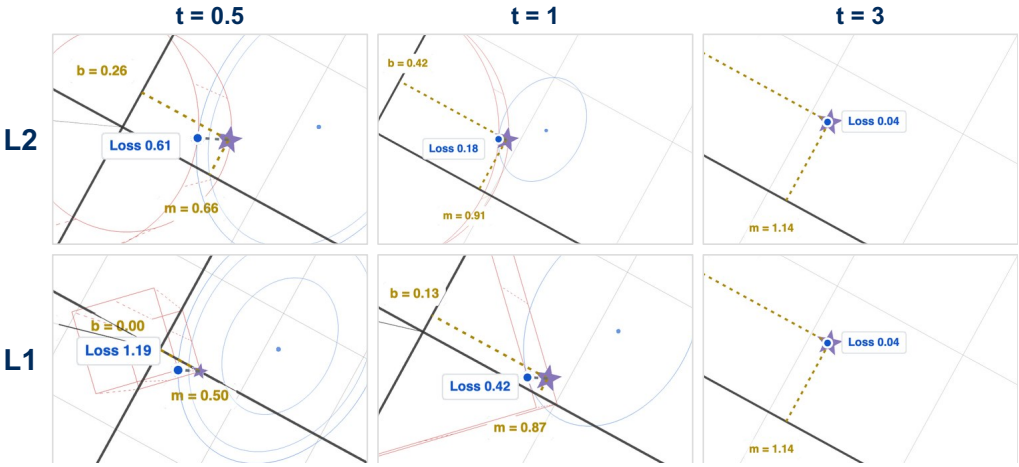
L1 as a budget

$$\min_{w} \; SSE(w) \quad \text{subject to} \quad \sum_{j} |w_j| \leq t$$



# Regularization as a budget, increasing $t$

A larger $t$ increases the **budget** for our weights, typically resulting in the optimum being closer to the unconstrained optimum.



# Sparsity

Look at the L1 row of the cost pictures. At λ = `3` the star lands exactly on one of the axes, so one of the two weights is exactly `0`. A model with weights of exactly `0` is called **sparse**. The budget pictures show why it happens: the L1 budget is a diamond, and the loss bowl almost always touches a diamond first at one of its corners, which sit right on the axes. The L2 budget is a circle with no corners, so its star gets pulled in close to an axis but never quite lands on it.

The cost and the budget aren't really two different ideas, they're the same problem written two ways. For every budget $t$ there's some **λ** that puts the star in exactly the same spot (proving it takes Lagrange multipliers). **A big λ is a small budget**, and a small **λ** is a big one.

# Feature Selection

A weight of exactly `0` means that feature is out of the model entirely, so L1 is doing **feature selection** for us for free. Let's give it something to select. Here are seven things from Notre Dame's box score: total yards, first downs, turnovers, penalties, third-down conversions, time of possession and number of plays.

Plain **OLS** uses all seven and explains `64%` of the variation in points. **LASSO** with a moderate $\lambda$ keeps exactly two of them, **total yards** (about `7.9` points per `100` yards) and **turnovers** (about `-1.9` points each), and still explains `61%`. The other five weights are exactly `0`. Most of what a box score tells you about points, it turns out, is how far you moved the ball and how often you gave it away.

One practical detail: before using LASSO we rescale every feature to the same spread. Otherwise a feature measured in big units (yards) and one measured in small units (turnovers) get penalized very differently for the same amount of usefulness.

# Ridge in sklearn

sklearn has both. One annoying detail: sklearn calls $\lambda$ `alpha`, which has nothing to do with the learning rate from gradient descent.

```
import pandas as pd
from sklearn.linear_model import LinearRegression, Ridge

games = pd.read_csv("nd_football.csv")
cols = ["nd_total_yards", "nd_pass_yards",
        "nd_rush_yards"]
X = games[cols] / 100
y = games["nd_points"]

ols = LinearRegression().fit(X, y)
ridge = Ridge(alpha=1).fit(X, y)   # alpha is λ here
print(ols.coef_.round(1))
print(ridge.coef_.round(1))
```

OUTPUT

```
[-64.4  72.4  74.2]
[5.9 2.  3.9]
```

Same predictions as **OLS**, sensible weights, exactly as on the inverse page.

# LASSO in sklearn

**LASSO** needs one extra step first, rescaling every feature to the same spread:

```
import pandas as pd
from sklearn.linear_model import Lasso
from sklearn.preprocessing import StandardScaler

games = pd.read_csv("nd_football.csv")
stats = ["nd_total_yards", "nd_first_downs",
         "nd_turnovers", "nd_penalties",
         "nd_third_down_conv", "nd_possession_min",
         "nd_plays"]
X = StandardScaler().fit_transform(games[stats])
y = games["nd_points"]

lasso = Lasso(alpha=1).fit(X, y)
for name, w in zip(stats, lasso.coef_):
    print(f"{name:20s} {w + 0:6.2f}")
```

OUTPUT

```
nd_total_yards       8.16
nd_first_downs       0.00
nd_turnovers        -2.29
nd_penalties         0.00
nd_third_down_conv   0.00
nd_possession_min    0.00
nd_plays             0.00
```

These **LASSO** weights are per standard deviation (that's what `StandardScaler` did), so compare them to each other rather than to the **9** points per **100** yards from before.

# Maximum Likelihood, Revisited

Last chapter we saw that least squares is maximum likelihood in disguise. **Regularization** fits into the same picture, with one extra term.

The setup is an assumption about the noise. Every game's score is the line plus some noise, and the noise is Gaussian with spread σ: $y_i = \hat{y}_i + \varepsilon_i$ with $\varepsilon_i \sim N(0, \sigma^2)$. **MLE** picks the weights that make the scores we saw most likely, and taking the negative log turns that product of bell curves into a sum of squares:

[6.5]

$$\hat{w}_{MLE} = \arg\max_{w} \; p(y \mid w)$$

$$\downarrow \text{take } -\log$$

$$= \arg\min_{w} \; \frac{SSE(w)}{2\sigma^2} = \arg\min_{w} \; SSE(w)$$

# Adding a prior

We could also say which weights we expected before seeing any games (a **prior**, $p(\omega)$), and ask for the most likely weights given the data. By Bayes' rule that means multiplying the prior in. If we believe each weight is probably near $0$, a Gaussian prior $\omega_j \sim N(0, \tau^2)$, its negative log is one more sum of squares:

$$\hat{\omega}_{MAP} = \arg\max_{\omega} \; p(y \mid \omega)\, p(\omega)$$

$$\downarrow \text{ take } -\log$$

$$= \arg\min_{\omega} \; \frac{SSE(\omega)}{2\sigma^2} + \frac{1}{2\tau^2} \sum_j \omega_j{}^2$$

$$\downarrow \text{ multiply by } 2\sigma^2$$

$$[6.6] \qquad = \arg\min_{\omega} \; SSE(\omega) + \frac{\sigma^2}{\tau^2} \sum_j \omega_j{}^2$$

That's the ridge loss [6.2] with $\lambda = \sigma^2/\tau^2$. So **$\lambda$ is how noisy we think the data is, divided by how big we think the weights are**. Our yards-and-turnovers line misses by about $7.7$ points a game ($\sigma^2 \approx 60$), so if we expected weights of about $\pm 5$ ($\tau = 5$) we'd get $\lambda \approx 2.4$. A flat prior ($\tau \to \infty$) sends $\lambda$ to $0$ and **MAP** back to **MLE**. A Laplace prior, $p(\omega_j) \propto e^{-|\omega_j|/b}$, gives $\lambda \Sigma |\omega_j|$ instead, which is LASSO.

# MAP estimation

**L1** and **L2 regularization** are a form of **MAP Estimation**.

**MAP** stands for Maximum A Posteriori which is a **Bayesian** approach for **applying a prior to our weights**.

So both kinds of regularization are a prior in disguise. Ridge is a belief that the weights are probably small, **LASSO** a sharper belief that piles up right at `0`, and $\lambda$ is how strongly we believe it compared to the data. **Regularization is really just us telling the model what we expected ahead of time**.