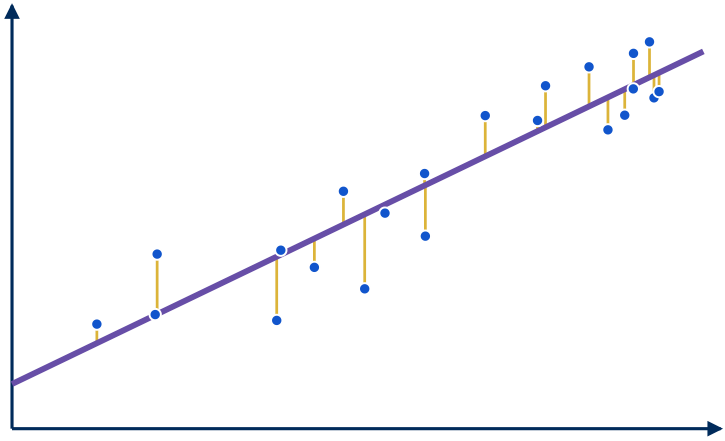


UNIT 02 · CHAPTER 5

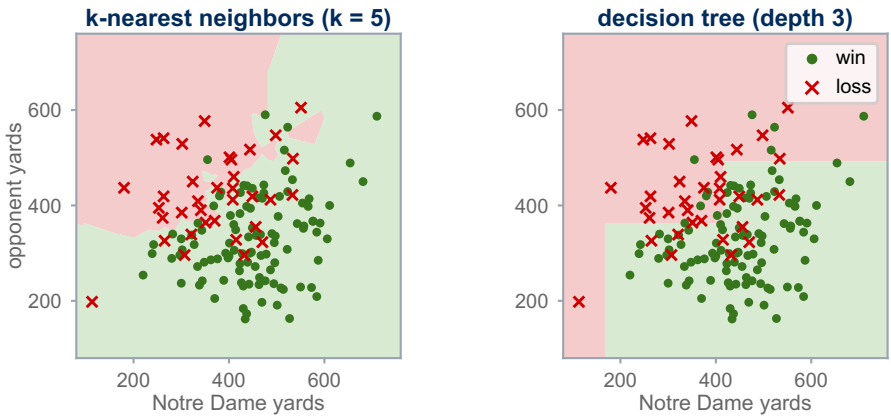
Linear Regression

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

Classification Tasks

Given our dataset of Notre Dame football games, we've been trying to predict a simple outcome based on the data: **"Did we win?"**

This is an example of a **classification task** because the model's output will be one of two discrete labels: **Yes (1)** or **No (0)**. The two models we've looked at thus far, **k-Nearest Neighbours (kNNs)** and **Decision Trees**, were built for classification.



As we can see above, **kNN** draws a wiggly boundary around the losses (0s), while the **Decision Tree** carves the plane into rectangles. They both achieve about the same accuracy (about **82%** of the time on unseen games).

What our models can't predict, however, is how many points Notre Dame will score.

Regression Tasks

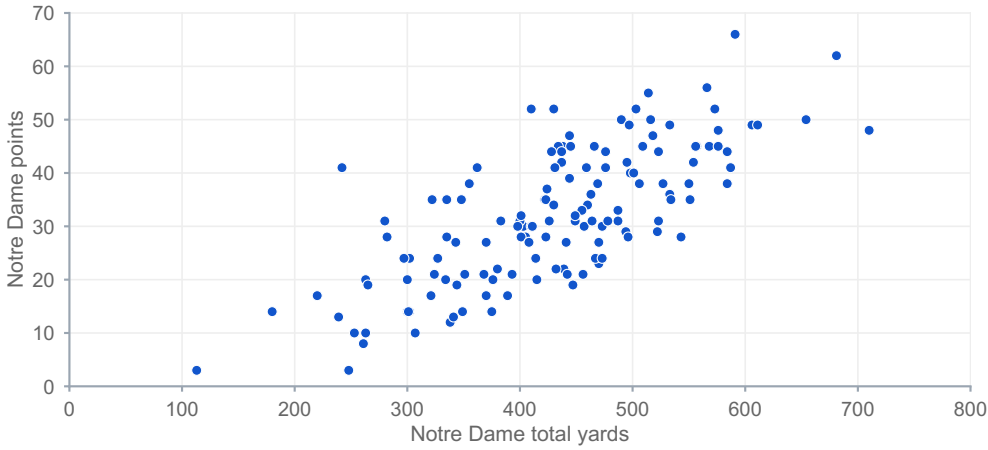
Predicting Notre Dame's score is a **fundamentally different type of question** to ask. Our output will no longer be a discrete label but a continuous value. Notre Dame's score could be any whole number from `0` up (there are enough of them that we treat the score as continuous). We call a task with a continuous output a **Regression Task**. Other examples of a regression task could include **predicting the temperature**, or the **stock price of a stock on the next day**.

Regression is another form of **supervised learning**, because every game in our data comes with the ground-truth answer (the final score), so we can always check how far off our prediction was.

A new task calls for a new model, and the first regression model we will consider is called a **Linear Regression**.

The Simplest Model: A Line

Generally speaking, teams that gain more yards tend to score more points. The plot below shows the number of yards Notre Dame gained in each of our **141** games against the number of points it scored.



The simplest model that we could fit to that plot is a straight line with the equation $\hat{y} = mx$. Our input to the model is x (total yards), and the only thing the model gets to choose is m, the slope. Given those, we can predict $\hat{y}$ (points scored).

Since regression is supervised learning, we can compare our prediction against the ground truth value y in order to see how far off our prediction was. We call the difference between the true value (y) and the predicted value ($\hat{y}$) the **residual**: $r = y - \hat{y}$

Linear Regression

Unfortunately, because linear regression is so old, **it has a lot of vocab specific to it** that has accrued and changed over time, depending on the field using it. This can make it tricky to recognize a linear regression in different contexts.

[5.1] a line with one feature

$$\hat{y} = w_1 x_1 + w_0$$

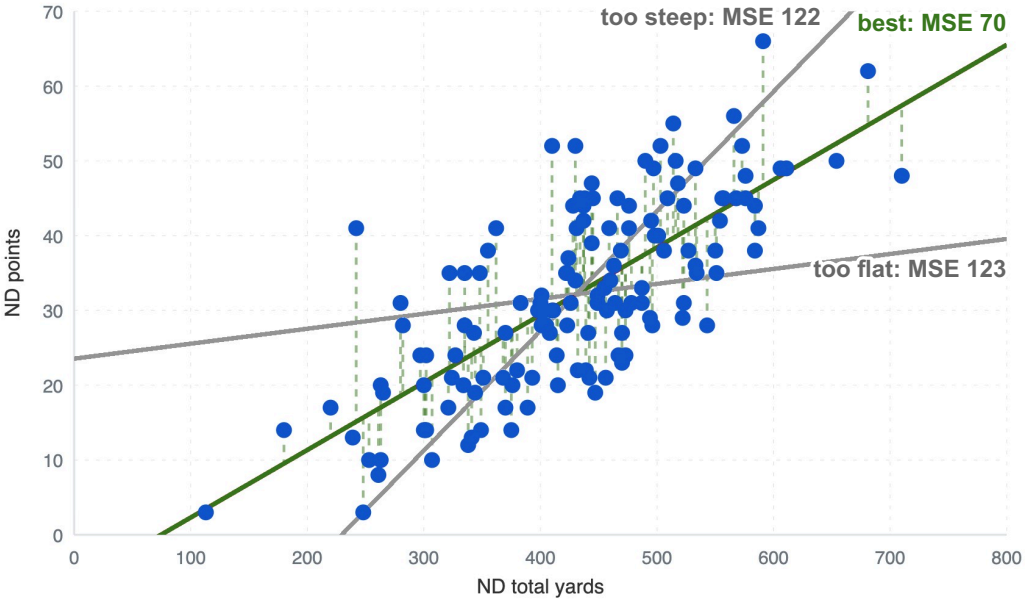
Note: You have probably seen a line written as $y = mx + b$. We call the slope w_1 and the intercept w_0, and w is the letter we'll use for weights in every model from here on out.

- $\hat{y}$: the prediction (predicted target)
- w_1, w_0: slope and intercept
- x_1: the input feature

We call it	You might also see
weights (w_1, w_2, ...)	coefficients, parameters, β_1 (statistics and economics)
intercept (w_0)	bias, constant, β_0, b
features (x_1, x_2, ...)	inputs, predictors, regressors, covariates, independent variables
target (y)	response, outcome, label, dependent variable
prediction ($\hat{y}$)	fitted value, estimate

Finding Our Line

How do we actually find the line that "fits" the data the best?



If you were to try and draw a line that "best fits" this data, you'd likely get pretty close to the green line that goes through the middle of the group of points. Out of the three lines drawn above, it's intuitive which of the three best fits the points.

Like most of machine learning, the harder question is **how do we get a computer to have that same intuition that we have naturally**.

Which Line Is Best?

Using our residuals from earlier ($y_i - \hat{y}_i$), we could imagine calculating them for every game, for each of the three lines, and adding them up. This would give us a **total amount of error for each line**, and the best line would be the one with the least error. The equation that gives us the total error is typically called a **cost, or a loss, function**.

Unfortunately, just adding up the residuals doesn't work. A game we over-predicted by `10` points and a game we under-predicted by `10` points would cancel out, and a truly terrible line could end up with a total error of `0`. Squaring each residual first makes every miss, positive or negative, count against us. It also means **big misses are punished much more than small ones** (being off by `20` points costs `400`, being off by `2` costs only `4`).

Cost and Loss Functions

For linear regression we often use **Sum of Squared Error (SSE)** or **Mean Squared Error (MSE)** as our cost function.

sum of squared errors

$$[5.2] \quad \mathrm{SSE}(\mathbf{w}) = \sum_i (y_i - \hat{y}_i)^2$$

mean squared error

$$[5.3] \quad \mathrm{MSE}(\mathbf{w}) = \frac{1}{n} \sum_i (y_i - \hat{y}_i)^2$$

MSE is just **SSE** normalized by the number of games, so **both pick the same line**. Normalizing means the error doesn't grow just because we have more games in our dataset. The three lines on the last page have **MSEs** of **70** (green), **122** (too steep) and **123** (too flat), so the loss agrees with our intuition.

Why Squares, Again

Maximum likelihood estimation (MLE) is a method of estimating the parameters of an assumed probability distribution, given some observed data.

Here's the assumption: every game's score is our line plus some random noise, and the noise follows a bell curve (a Gaussian) centered on zero with spread σ. Then we can ask a new question: which weights make the scores we actually saw the most likely?

the assumption

$$y_i = \hat{y}_i + e_i, \qquad e_i \sim \mathcal{N}(0, \sigma^2)$$

how likely one game is

$$p(y_i \mid w) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-(y_i - \hat{y}_i)^2 / 2\sigma^2}$$

↓ multiply over every game, then take the log

the log-likelihood

$$\log p(y \mid w) = -\frac{n}{2} \log(2\pi\sigma^2) - \frac{SSE(w)}{2\sigma^2} \quad [5.4]$$

The first term doesn't depend on w at all. So **maximizing the likelihood is minimizing the SSE**, and least squares has been doing maximum likelihood this whole time. That's the better reason for squaring: it's what you get if the noise is a bell curve (for football, one with a spread of about `8.4` points).

MLE trusts the data completely, so it fits *these* games as closely as it can, noise included.

Optimization Problems

Once we have a cost function that can measure the overall error of any given line, we could say that our goal is to **select the set of weights (or to optimize our weights) to produce a line with the lowest possible error**. Broadly speaking, this class of problems is called **optimization problems**, and these underlie almost all of machine learning.

Loss as a Function of the Weights

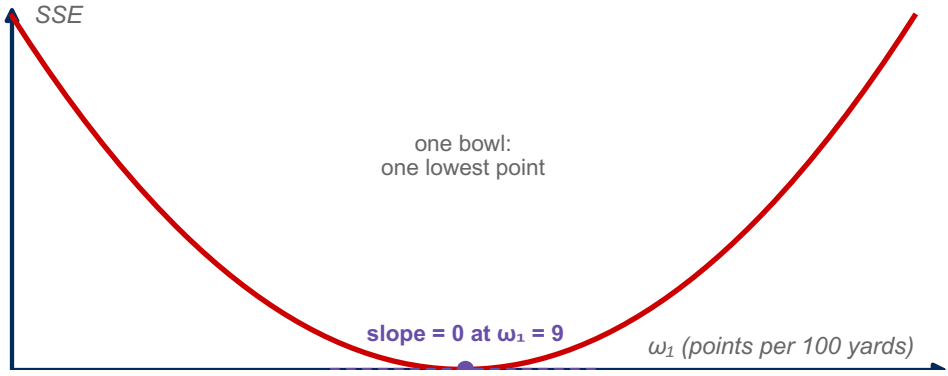
If the dataset stays fixed, the only thing that can change the loss is the weights, so **the loss function is really a function of the weights**: pick a w_1 and w_0, get one number back. This means that we could plot our loss function with respect to the model's parameters.

To see what that function looks like, let's shrink the problem to five games (small enough to do by hand later on), with yards rounded to the nearest hundred so x_1 is in hundreds of yards. We'll also hold the intercept still at $w_0 = -5.2$, so only the slope moves.

game	yards	x_1	y (points)
2019 Michigan	180	2	14
2012 Michigan State	300	3	20
2013 Purdue	400	4	31
2019 Boston College	501	5	40
2017 Temple	606	6	49

Visualizing Loss Functions with Respect to the Weights

We can plot the loss function with respect to ω_1:



Roughly speaking, our plot is a bowl. A ω_1 value that's too small misses every game low, too large misses them high, resulting in the SSE climbing on both sides.

SSE is a sum of squares of things that are linear in ω_1, meaning that our plot is always a bowl with exactly one lowest point. This is important, because the value of ω_1 at the lowest point of the bowl, is the value of ω_1 that will give the lowest loss, and **the value of ω_1 that gives us the best fitting line!**

Therefore, we could say that our optimization problem is trying to find the value of ω_1 at this lowest point.

Finding the Bottom

So given a loss function, our goal is to minimize it. We do this by finding the value of w_1 at the bottom of the bowl. But when would this loss function be minimized? If you think carefully, you may also notice that the bottom of this bowl is the **only place on our plot where the slope of the curve is 0**. Therefore, we can find the value of w_1 that corresponds to the slope of our loss function (or the gradient), being zero.

So we can take the derivative of the SSE, **set it equal to zero and solve**. With the intercept held at $w_0 = -5.2$ there's only one weight (w_1) to solve for:

[5.5]

the loss, one weight moving

$$\text{SSE}(w_1) = \sum_i (y_i - w_1 x_i - w_0)^2$$

↓ take the derivative (chain rule) and set it to 0

$$-2 \sum_i x_i \, (y_i - w_1 x_i - w_0) = 0$$

↓ split the sum and solve for w_1

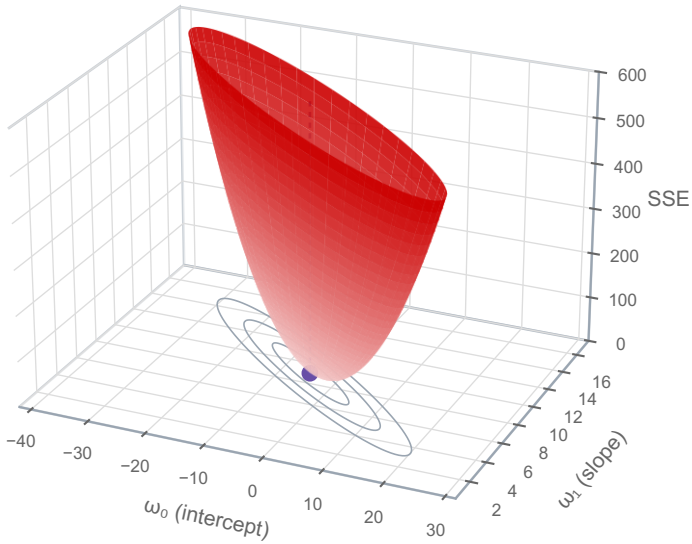
five games: 810 / 90

$$w_1 = \frac{\sum_i x_i (y_i - w_0)}{\sum_i x_i^2} = 9.0$$

That's exactly where the bowl bottoms out, 9 points per 100 yards.

Two Weights at Once

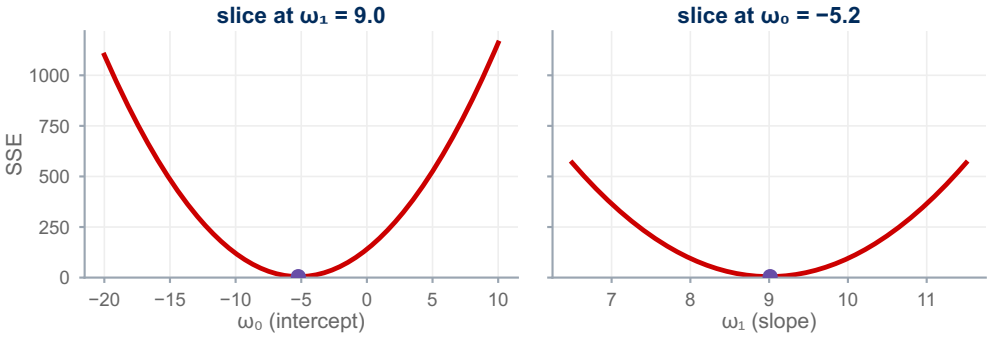
Of course most of the time we don't actually know the intercept. With both weights free the bowl becomes a bowl in two directions, and the bottom is where both derivatives are zero at once. Pictured below is the SSE of our five games over both weights.



Notice the bowl is long and thin. Raising the intercept while lowering the slope barely changes the loss, because with yards between 2 and 6 hundred the two moves mostly cancel.

Slicing the Bowl

Slicing through the lowest point along each weight gives two ordinary one-weight bowls. On the right is the bowl from before (the intercept held at $w_0 = -5.2$), and on the left the slope is held at $w_1 = 9.0$ while the intercept moves. Each slice bottoms out at the same best weights.



That's two equations to solve together, and with ten features it would be eleven. Writing each one out by hand gets old fast.

Matrix Notation for Linear Regressions

We can instead represent our entire problem as matrices and matrix operations. Put a $\mathbf{1}$ at the front of every game, $\mathbf{x}_i = [1, \mathbf{x}_{i1}]$, and put the weights in one row, $\boldsymbol{\omega} = [\boldsymbol{\omega}_0, \boldsymbol{\omega}_1]$. A prediction is then one dot product, and stacking the games as the rows of X predicts all of them at once:

[5.6] every prediction at once

$$\hat{\mathbf{y}} = \mathbf{X}\,\boldsymbol{\omega}^{\mathsf{T}}$$

Rewriting our problem using matrix notation allows us to **solve for all weights at the same time!**

Note: Most textbooks (and the sklearn documentation) write $\boldsymbol{\omega}$ as a column instead of a row, so predictions are $\hat{\mathbf{y}} = \mathbf{X}\boldsymbol{\omega}$ and the solution is $\boldsymbol{\omega} = (\mathbf{X}^{\mathsf{T}}\mathbf{X})^{-1}\mathbf{X}^{\mathsf{T}}\mathbf{y}$. It's the same math with the weights stood on end. We keep $\boldsymbol{\omega}$ as a row so one game's prediction, $\mathbf{x}_i\,\boldsymbol{\omega}^{\mathsf{T}}$, reads in the same order as our equation for a line.

Ordinary Least Squares

We can again **solve for the gradient equaling 0**, but let's use matrix notation this time:

SSE for every sample at once

$$\mathcal{L}(\mathbf{w}) = \| \mathbf{y} - \mathbf{X}\mathbf{w}^\top \|^2$$

↓ take the gradient and set it to 0

$$-2\,\mathbf{X}^\top(\mathbf{y} - \mathbf{X}\mathbf{w}^\top) = \mathbf{0}$$

↓ multiply out

the normal equations

$$\mathbf{X}^\top\mathbf{X}\,\mathbf{w}^\top = \mathbf{X}^\top\mathbf{y}$$

↓ multiply both sides by $(\mathbf{X}^\top\mathbf{X})^{-1}$

ordinary least squares [5.7]

$$\mathbf{w}^\top = (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{y}$$

Ordinary Least Squares, Continued

The middle line is just the two derivatives from the one-weight page, stacked. The row of `1`s in X^T gives the derivative for w_0, the row of yards gives the one for w_1. Everything after that is solving two equations at once.

This is called **Ordinary Least Squares (OLS)**, and it gives us **the exact best value for every weight in our model**, straight from the training data (X and y).

Worked OLS Example

Back to our five games, with yards rounded to the nearest hundred (x_1 is in hundreds of yards).

Give X a **column of 1s** for the intercept, then follow the OLS recipe.

the data, written on its side (transposed) to save room

$$X^T = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 2 & 3 & 4 & 5 & 6 \end{bmatrix} \qquad y^T = \begin{bmatrix} 14 & 20 & 31 & 40 & 49 \end{bmatrix}$$

two small products

$$X^T X = \begin{bmatrix} 5 & 20 \\ 20 & 90 \end{bmatrix} \qquad X^T y = \begin{bmatrix} 154 \\ 706 \end{bmatrix}$$

2 × 2 inverse: swap, negate, divide by $5 \cdot 90 - 20 \cdot 20 = 50$

$$(X^T X)^{-1} = \frac{1}{50} \begin{bmatrix} 90 & -20 \\ -20 & 5 \end{bmatrix}$$

the weights

[5.8]
$$\omega^T = \frac{1}{50} \begin{bmatrix} 90 \cdot 154 - 20 \cdot 706 \\ -20 \cdot 154 + 5 \cdot 706 \end{bmatrix} = \begin{bmatrix} -5.2 \\ 9.0 \end{bmatrix}$$

So $\hat{y} = 9.0\, x_1 - 5.2$: about 9 points per 100 yards.

What the Line Learned

If we use all 141 games, OLS gives us $\hat{y} = 9.0\,x_1 - 6.7$, the same slope (these five games happen to sit close to the full-data line). The intercept is another story. It claims a team that gains no yards at all scores −6.7 points. Our fewest yards in any game was 113, so the line was never asked to be right anywhere near zero.

Using a model outside the data it was fit on is called **extrapolation**, and it's **where lines say silly things**.

OLS in sklearn

```python
import pandas as pd
from sklearn.linear_model import LinearRegression

games = pd.read_csv("nd_football.csv")
X = games[["nd_total_yards"]] / 100   # hundreds
y = games["nd_points"]

model = LinearRegression().fit(X, y)
print(model.coef_.round(2), model.intercept_.round(2))
print(round(model.score(X, y), 2))    # R²
```

OUTPUT

```
[9.03] -6.73
0.55
```

The last number is **R²**, the share of the game-to-game variation in points that our line explains. An `R²` of `1` would be a perfect fit, and `0` would be no better than always guessing the average score. Yards alone explain a bit over half of it.

Multiple Linear Regression

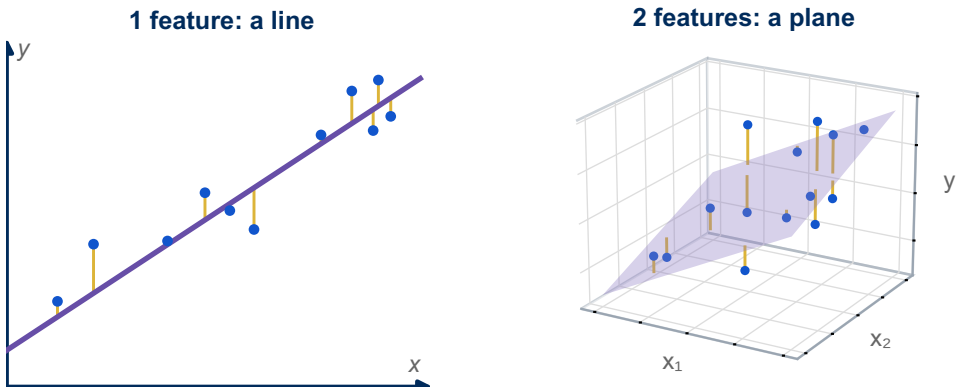
Yards gained of course isn't the whole story of a football game. It's rare that we have datasets with only a single feature, and even more rare that we'd ever want to use only a single feature.

We can expand our equation of a line to include any number of features we might have:

$$\hat{y} = w_0 + w_1 x_1 + w_2 x_2 + \ldots + w_n x_n$$

When we perform a linear regression on a dataset with more than one feature, we call this a multiple linear regression.

Hyperplanes



With one feature the model is a line, with two it's a **plane** and with more than two it's a **hyperplane**.

Yards and Turnovers

Give the model Notre Dame's turnovers as a second feature, x_2. Nothing about the method changes, X just grows a third column and the same **OLS** formula hands back three weights:

two features, fit to 141 games

$$\hat{y} = 8.8\,x_1 - 2.7\,x_2 - 2.3$$

Multiple Linear Regressions

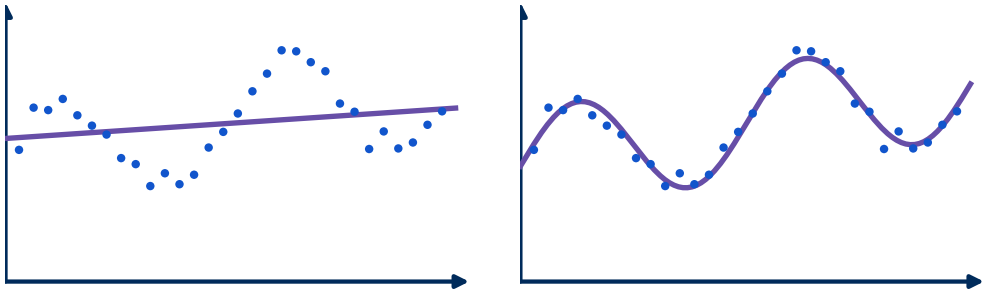
Each weight now answers a "holding everything else still" question. If yards stay the same, **every turnover costs Notre Dame about `2.7` points**. The yards weight barely moved (`9.0` alone, `8.8` here), so turnovers are telling the model something yards didn't. The typical error drops too, the **MSE** goes from `70` to `60`.

This is the payoff for all the matrix notation. One feature or a hundred, it's the same equation:

$$\mathbf{w}^T = (X^T X)^{-1} X^T \mathbf{y}$$

Representing Curves

Not every dataset is best represented with a straight line. How could you use a **linear regression** to model a sine wave?



A linear regression is only linear in its weights, not in its inputs. We could hand it new features made from the old ones, like x^2 or $\sin(x)$, and it would still fit them with the same equation:

[5.9] a straight line in new features

$$\hat{y} = w_0 + w_1 x + w_2 \sin(1.25x)$$

Building new features out of old ones like this is called **feature engineering**.

Polynomial Regression

The most common recipe for feature engineering is using powers of the input (often called **polynomial regression**): x, x^2, x^3 and so on. The highest power is called the model's **degree**. A `degree-2` model can bend once, while a `degree-8` model can bend seven times. While bending is a powerful technique (please refer to Avatar), it can also lead to something called **overfitting**.

Overfitting

Let's fit the `13` games from `2012` using three different models: `degree-1`, `degree-4`, and `degree-8`.

Model	MSE on 2012
line (degree 1)	39
degree 4	35
degree 8	15

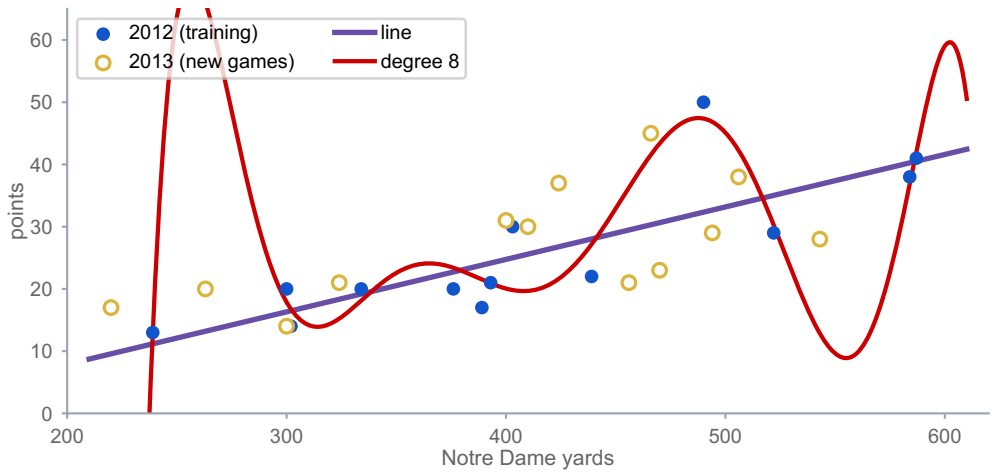
The `degree-8` model has a lovely **MSE** of `15` when fit on the games from `2012`. Now with these three models, let's try and predict the results of the games in `2013`.

Overfitting

Model	MSE on 2012	MSE on 2013
line (degree 1)	39	56
degree 4	35	65
degree 8	15	**9,553**

Our `degree-8` model does terribly with an **MSE** of **9,553**, despite having the lowest **MSE** on the training data (games from `2012`)!

The `degree-8` curve misses our `2013` data by about **98 points a game**. With `9` parameters, but only `13` games in `2012`, it can bend through almost every point, so it learns the `2012` games almost perfectly instead of the underlying pattern. That's **overfitting**. The `degree-8` model's biggest weight is about `1.9` million. An abnormally high weight is usually a good sign that **overfitting** is taking place.

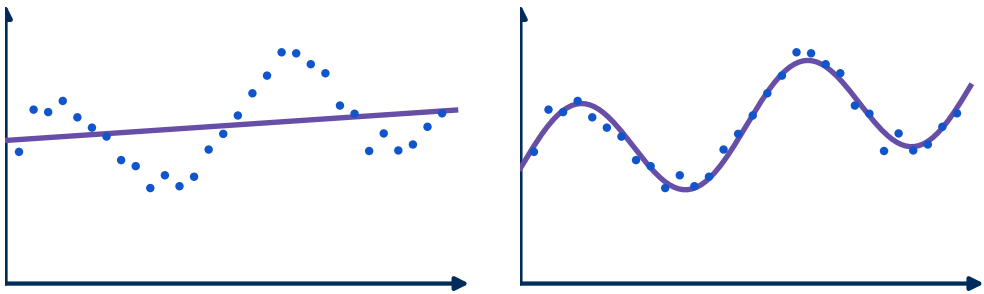


Variance

When a model is overfit, it will do really well on its training data, but very poorly on unseen data. We would say that the model has a **high variance**. Its performance changes a lot when applied to different datasets (`2012` games and `2013` games) that are drawn from the same underlying data distribution (Notre Dame football games).

Underfitting

The inverse of **overfitting** is **underfitting**. This is when your model is "too weak" to capture all of the underlying patterns in the data.



The straight line through the sine wave here is underfit. No choice of slope and intercept can correctly follow the curve, and giving it more data won't help.

Bias

Similarly to our `degree-8` model having a high variance, we would say that our linear model has a **high bias**. By picking a linear model to try and fit our sine wave, we introduced a **bias** into the problem. We made the assumption that a linear model would be strong enough to capture the pattern (and we were wrong).

Bias is how wrong a model is because it's too rigid or simple, like the line through the wave. **Variance** is how much performance changes when the data changes, the `degree-8` model would look completely different fit to `2014` instead of `2012`.

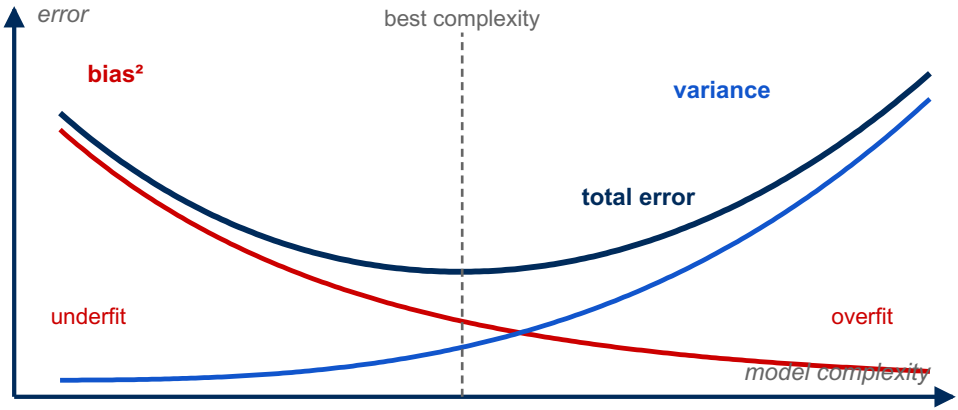
We can move between **bias** and **variance** by increasing and decreasing the complexity of our model. Our linear model had only two weights: w_0 and w_1 but our `degree-8` model had `9` weights:

degree 8: nine weights

$$\hat{y} = w_0 + w_1 x + w_2 x^2 + w_3 x^3 + w_4 x^4 + w_5 x^5 + w_6 x^6 + w_7 x^7 + w_8 x^8$$

Adding complexity **decreases bias** but **increases variance**. Removing complexity **decreases variance** but **increases bias**. We call this the bias-variance tradeoff.

The Bias-Variance Tradeoff



The **bias–variance tradeoff** describes the relationship between a model's complexity, the accuracy of its predictions (measured as overall error), and how well it can make predictions on unseen data.

Assessing our model's **bias** and **variance** requires data that the model didn't see during the fitting/training process.

Checking on Games We Haven't Seen

The standard way to do that is to hold a subset of our data back. We split the games into a **training set**, which the model gets to fit, and a **test set**, which it doesn't see until we're done. For football the natural split is by time: train on the `103` games from `2012` to `2019` and test on the `38` from `2020` to `2022`. Doing so yields the results below:

Features	How many	Training MSE	Test MSE
total yards	1	64	89
yards and turnovers	2	52	**82**
seven box-score stats	7	47	89
prior seven, plus their squares and products (polynomial expansion)	35	33	107
every product of up to three of them	119	0	1,443

The **MSE** on our training data decreases every time we add features. The **MSE** on our testing data decreases once but then climbs. The model with `119` features can pass exactly through all `103` training games but misses the next three seasons by about `38` points a game. **The model with the best training MSE is the worst model in the table.** We could say that the model with `119` features (the most complex model) is overfit to the training data (**low bias** but **high variance**).

Issues with Inverses

While making a model more complex via **polynomial regression** can be a powerful technique (we get a training **MSE** of `0` using `119` features on the prior page), using a **polynomial regression** also means that we **increase the number of features**, which can quickly run us into trouble.

Recall that calculating **OLS** relies on computing the inverse of X^TX. Taking the inverse of this can be **extremely computationally expensive** and **sometimes just literally impossible**. With `119` features, X^TX is a $<119, 119>$ matrix, and inverting it takes on the order of 119^3 steps. Imagine if we had a million features! Worse, if one feature can be built by combining other features, **X^TX has no inverse at all**. The `119`-feature model from the last page is one of these: it has more features than training games (`103`), so the formula as written couldn't have fit it.

Our football data walks straight into this. Total yards is just pass yards plus rush yards, so if we hand the model all three, one column is the sum of the other two. Any amount of weight on total yards can be swapped for the same weight on pass and rush yards, the predictions don't change, and **OLS** can't pick a best answer, because infinitely many combinations of weights tie! We will address overfitting and the inverse issue in the next chapter.