CS 4342: Class 5

Jacob Whitehill

Exercise

- Machine learning models can differ in several ways:
    1. How are they trained?
    2. What parameters do they have?
    3. How do they operate after they are trained?
- Answer questions 1, 2, and 3 above for:
    - The model in homework 1
    - Linear regression

Iterative solution to linear regression

Linear regression

- Linear regression is one of the few ML algorithms that has an analytical solution:

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

- **Analytical solution**: there is a closed formula for the answer.

Linear regression

- Alternatively, linear regression can be solved numerically using gradient descent.

- **Numerical solution**: need to iterate (according to some algorithm) many times to *approximate* the optimal value.

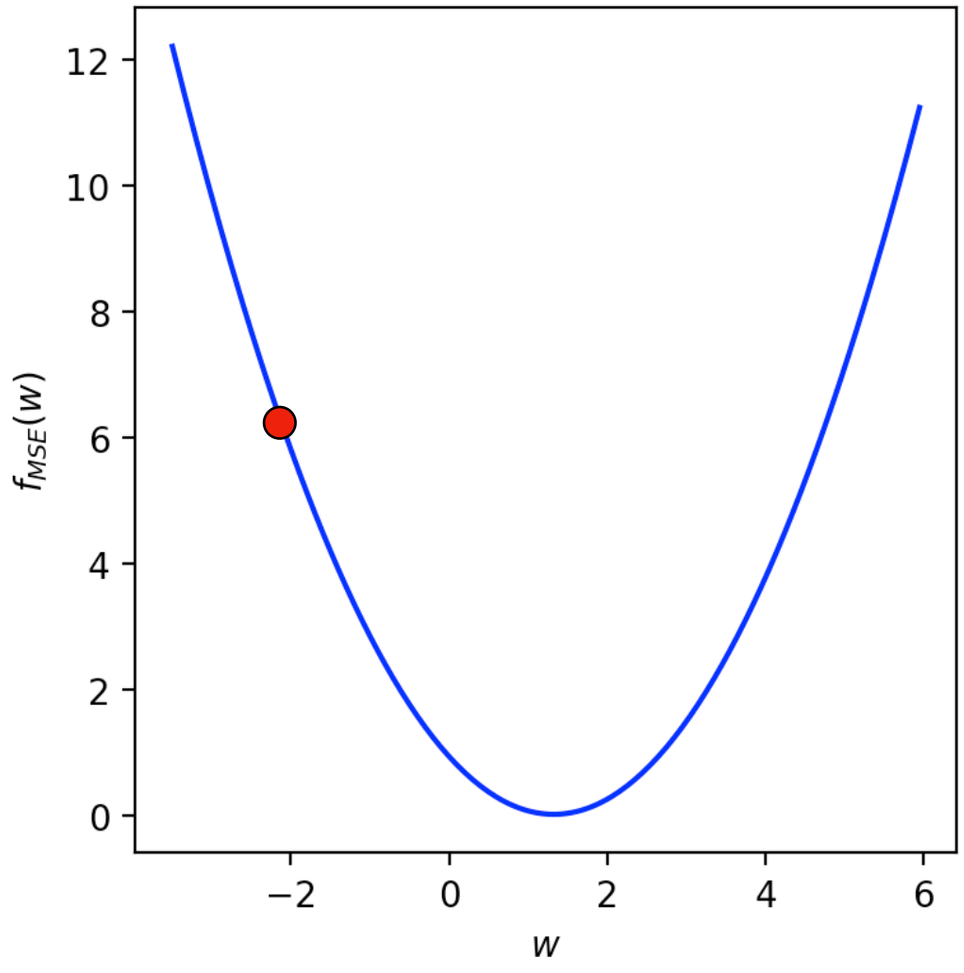
- Gradient descent is more laborious to code than the one-shot solution, but it generalizes to a wide variety of ML models.

Gradient descent

- Gradient descent is a **hill climbing algorithm** that uses the gradient (aka slope) to decide which way to “move” **w** to reduce the objective function (e.g., f_{MSE}).

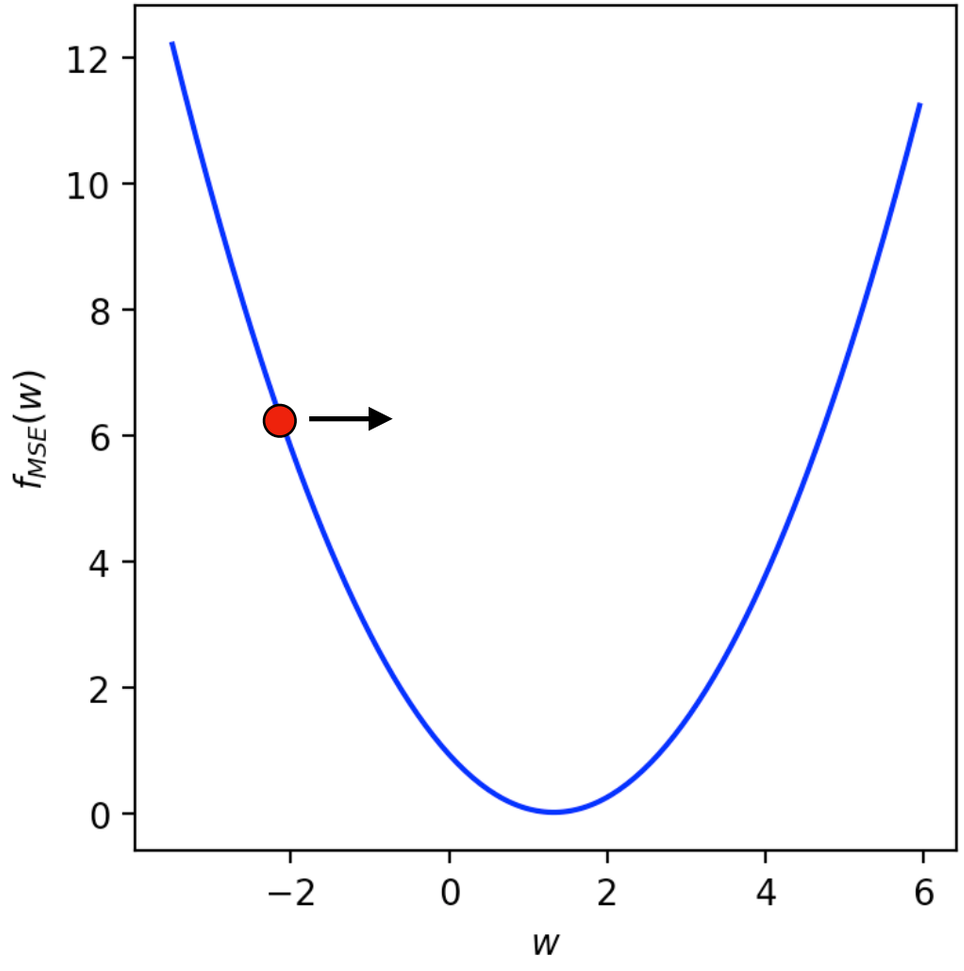
Gradient descent

- Suppose we just guess an initial value for **w** (e.g., -2.1).
- How can we make it better — increase it or decrease it?



Gradient descent

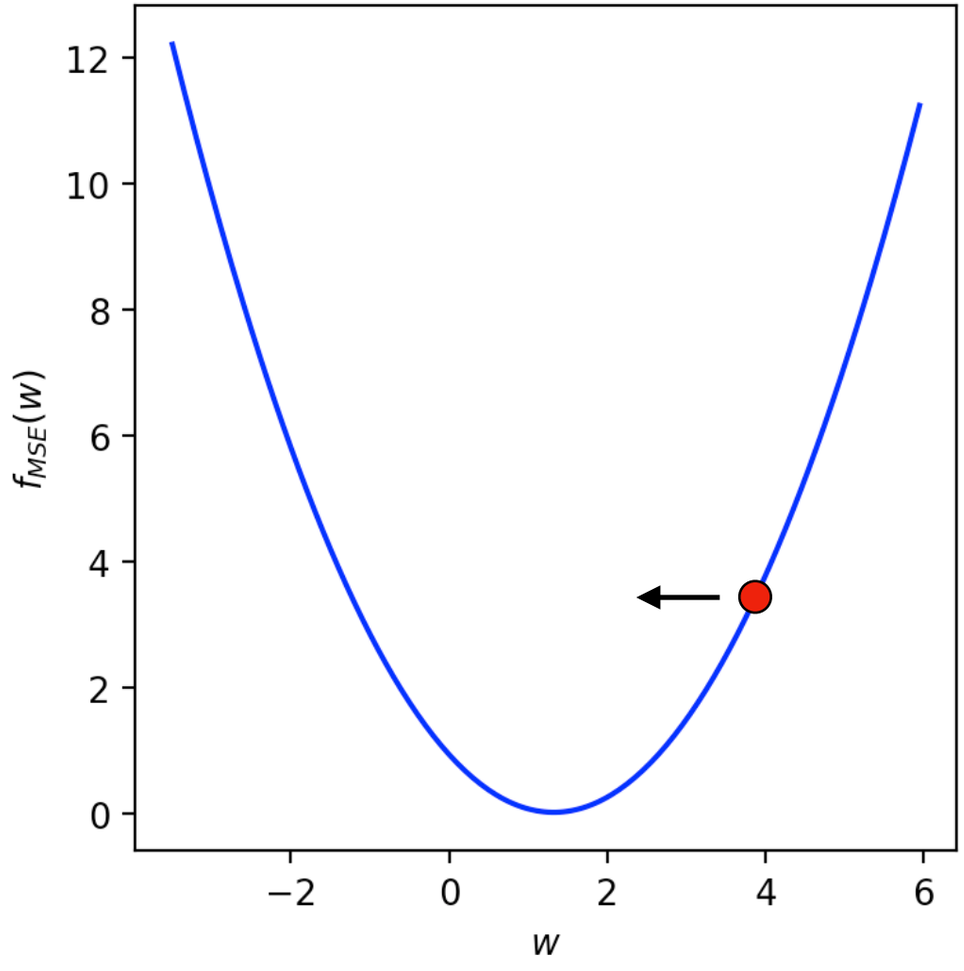
- Suppose we just guess an initial value for **w** (e.g., -2.1).
- How can we make it better — **increase** it or decrease it?
  - What does the **slope** of f_{MSE} tell us to do?



The slope at f_{MSE}(-2.1) is *negative*, i.e., we can *decrease* our cost by *increasing* w.

Gradient descent

- Or maybe our initial guess for **w** was 3.9.
- How can we make it better — increase it or **decrease** it?
  - What does the **slope** of f_{MSE} tell us to do?



The slope at $f_{\text{MSE}}(3.9)$ is *positive*, i.e., we can *decrease* our cost by *decreasing* w.

Gradient descent

- How do we know the slope? Compute the **gradient** of f_{MSE} w.r.t. **w**:

$$
\begin{aligned}
\nabla_{\mathbf{w}} f_{\text{MSE}}(\mathbf{y}, \hat{\mathbf{y}}; \mathbf{w}) &= \nabla_{\mathbf{w}} \left[\frac{1}{2n} \sum_{i=1}^{n} \left(\mathbf{x}^{(i)\top} \mathbf{w} - y^{(i)} \right)^2 \right] \\
&= \frac{1}{2n} \sum_{i=1}^{n} \nabla_{\mathbf{w}} \left[\left(\mathbf{x}^{(i)\top} \mathbf{w} - y^{(i)} \right)^2 \right] \\
&= \frac{1}{n} \sum_{i=1}^{n} \mathbf{x}^{(i)} \left(\mathbf{x}^{(i)\top} \mathbf{w} - y^{(i)} \right) \\
&= \frac{1}{n} \mathbf{X} \left(\mathbf{X}^{\top} \mathbf{w} - \mathbf{y} \right)
\end{aligned}
$$

Gradient descent

- How do we know the slope? Compute the **gradient** of f_{MSE} w.r.t. **w**:

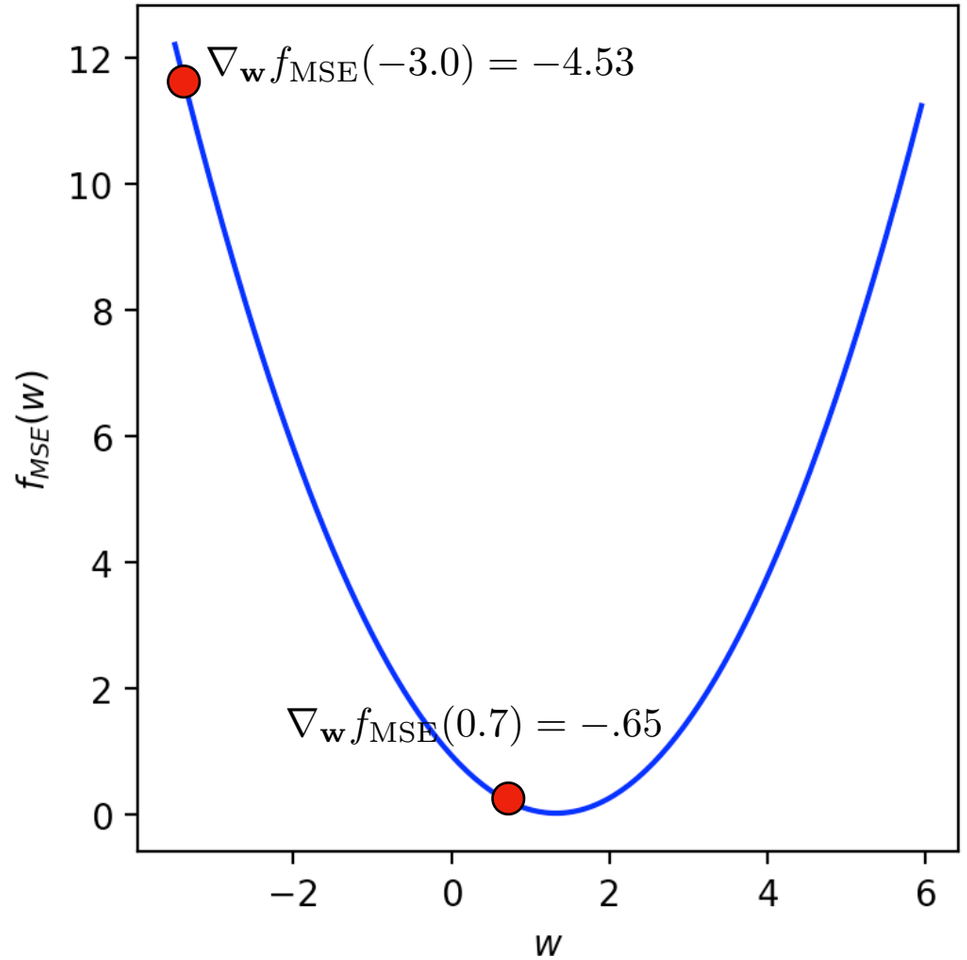
$$\begin{aligned}
\nabla_{\mathbf{w}} f_{\text{MSE}}(\mathbf{y}, \hat{\mathbf{y}}; \mathbf{w}) &= \nabla_{\mathbf{w}} \left[\frac{1}{2n} \sum_{i=1}^{n} \left(\mathbf{x}^{(i)\top} \mathbf{w} - y^{(i)} \right)^2 \right] \\
&= \frac{1}{2n} \sum_{i=1}^{n} \nabla_{\mathbf{w}} \left[\left(\mathbf{x}^{(i)\top} \mathbf{w} - y^{(i)} \right)^2 \right] \\
&= \frac{1}{n} \sum_{i=1}^{n} \mathbf{x}^{(i)} \left(\mathbf{x}^{(i)\top} \mathbf{w} - y^{(i)} \right) \\
&= \frac{1}{n} \mathbf{X} \left(\mathbf{X}^{\top} \mathbf{w} - \mathbf{y} \right)
\end{aligned}$$

- Then plug in the current value of **w**.
  (Note that **X** and **y** are computed from the data and are constant.)

Gradient descent

- How *far* do we "move" left or right?

  - Notice that, in the graph below, the **magnitude** of the slope (aka gradient) gives an indication of how far we need to go to reach the optimal **w**.



Gradient descent algorithm

- Set **w** to random values; call this initial choice $\mathbf{w}^{(0)}$.

  Python: `w = 0.01 * np.random.randn(M)  # Just an example!`

Gradient descent algorithm

- Set **w** to random values; call this initial choice $\mathbf{w}^{(0)}$.
- Compute the gradient: $\nabla_{\mathbf{w}} f(\mathbf{w}^{(0)})$

Gradient descent algorithm

- Set **w** to random values; call this initial choice $\mathbf{w}^{(0)}$.
- Compute the gradient: $\nabla_{\mathbf{w}} f(\mathbf{w}^{(0)})$
- Update **w** by moving opposite the gradient, multiplied by a **step size** ε. $\mathbf{w}^{(1)} \leftarrow \mathbf{w}^{(0)} - \epsilon \nabla_{\mathbf{w}} f(\mathbf{w}^{(0)})$

Gradient descent algorithm

- Set **w** to random values; call this initial choice $\mathbf{w}^{(0)}$.
- Compute the gradient: $\nabla_{\mathbf{w}} f(\mathbf{w}^{(0)})$
- Update **w** by moving opposite the gradient, multiplied by a **step size** ε. $\mathbf{w}^{(1)} \leftarrow \mathbf{w}^{(0)} - \epsilon \nabla_{\mathbf{w}} f(\mathbf{w}^{(0)})$
- Repeat…

$$\mathbf{w}^{(2)} \leftarrow \mathbf{w}^{(1)} - \epsilon \nabla_{\mathbf{w}} f(\mathbf{w}^{(1)})$$

$$\mathbf{w}^{(3)} \leftarrow \mathbf{w}^{(2)} - \epsilon \nabla_{\mathbf{w}} f(\mathbf{w}^{(2)})$$

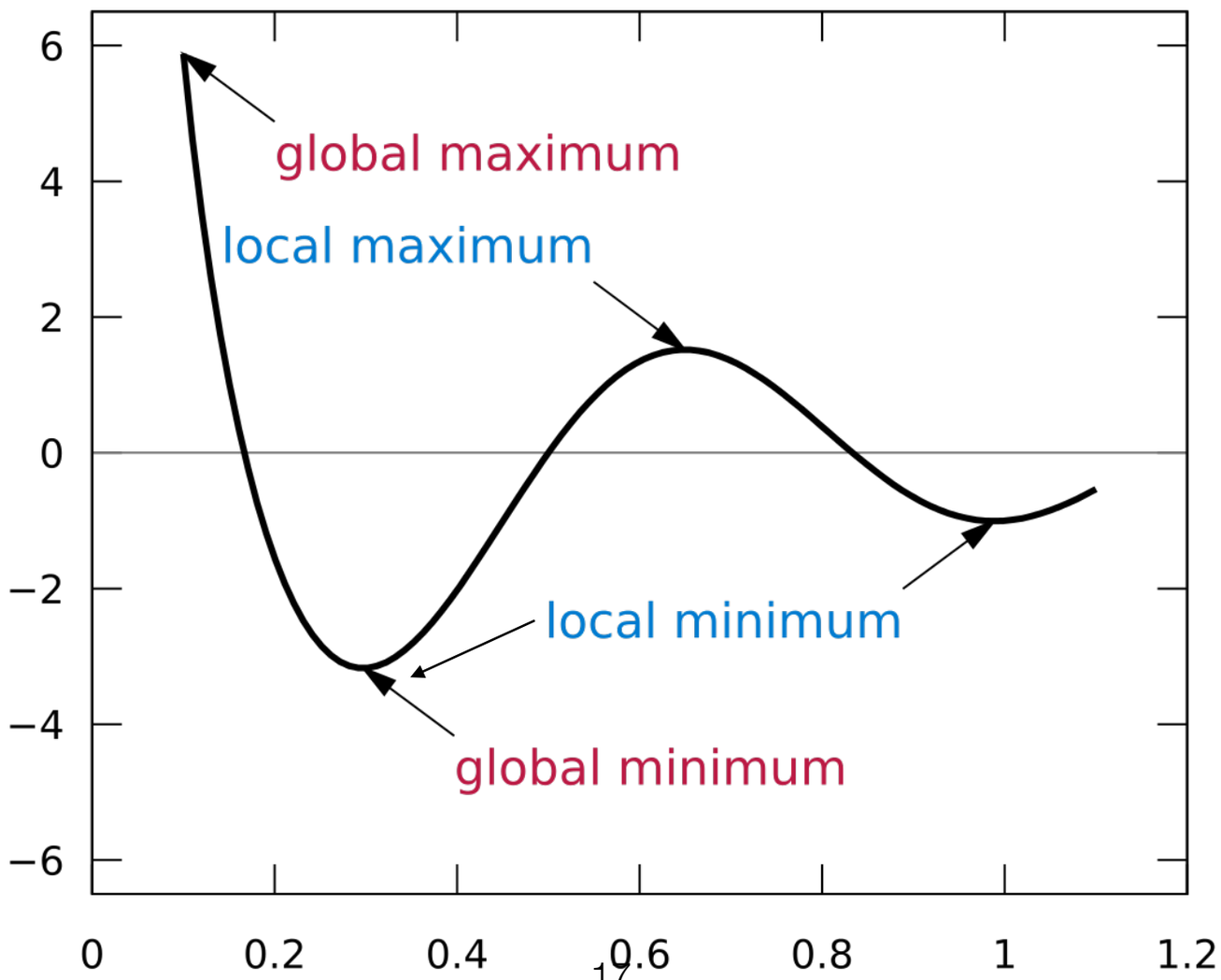
$$\ldots$$

$$\mathbf{w}^{(t)} \leftarrow \mathbf{w}^{(t-1)} - \epsilon \nabla_{\mathbf{w}} f(\mathbf{w}^{(t-1)})$$

Python: `w = w - EPS * gradient(w, X, y)`

Convergence

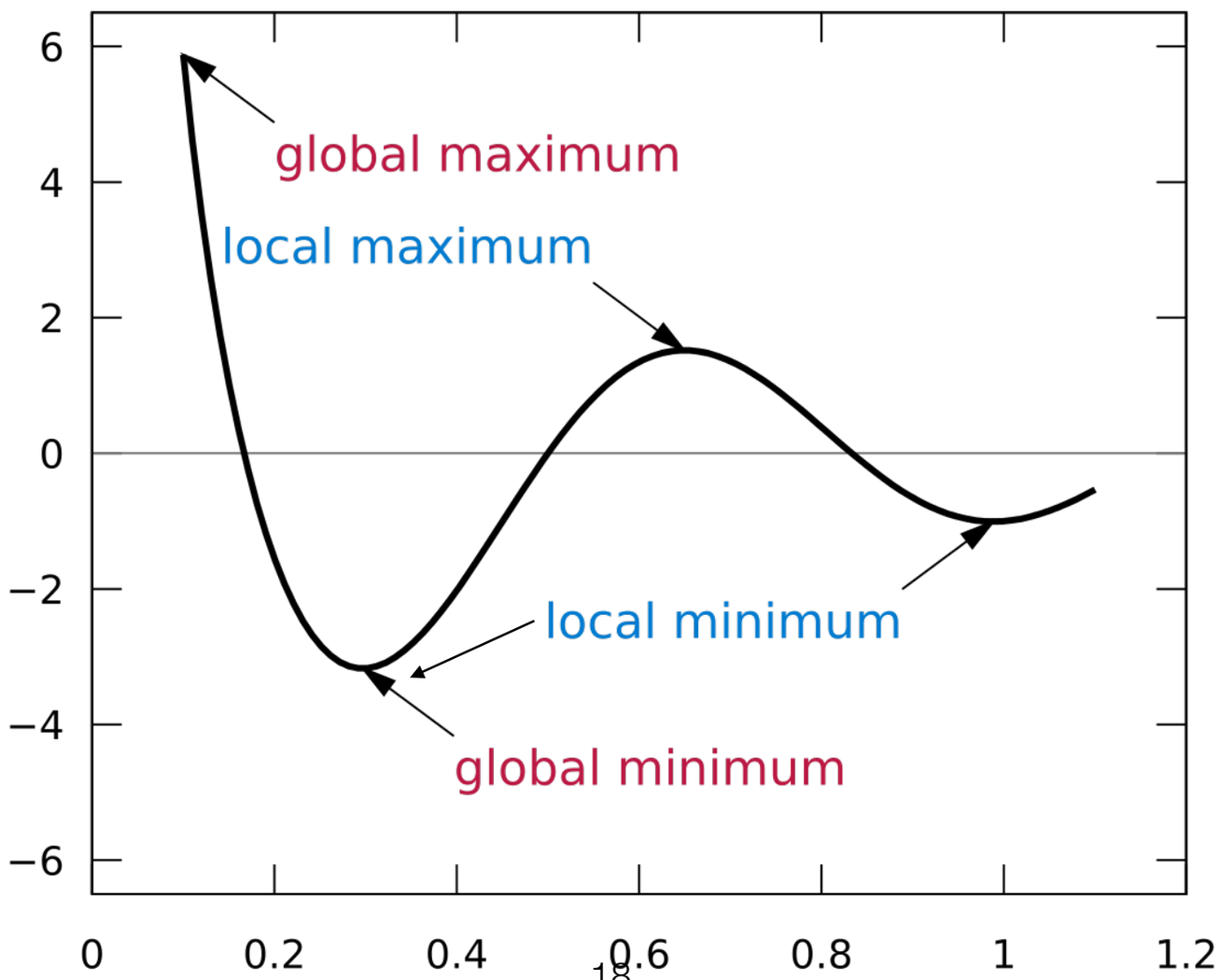
- In general, gradient descent is useful for finding a local minimum of *f*:
  - **Local minimum:** gradient is 0; second derivative is positive.



https://en.wikipedia.org/wiki/Maxima_and_minima

Convergence

- In general, gradient descent is useful for finding a local minimum of *f*:
  - **Global minimum** is the smallest value of the function *f*.



https://en.wikipedia.org/wiki/Maxima_and_minima

Convergence

- For the special case of linear regression (and a few other ML models), gradient descent will (for appropriate ε) converge to the *global* minimum of f_{MSE}.

Convergence

- For the special case of linear regression (and a few other ML models), gradient descent will (for appropriate ε) converge to the *global* minimum of f_{MSE}.
- What does “appropriate ε” mean (intuitively)?
  - Big enough to make progress (from random starting point) to local minimum.
  - Small enough not to “jump around” too much.
    - Show demo.

Convergence

- In practice:
  - Choose some ε so that the cost f_{MSE} declines *smoothly*.
  - If it's too slow, try increasing.
  - If it jumps around, try decreasing.

Gradient descent algorithm

- How many iterations to run?
- Several alternative strategies:
  - Train for a fixed number of iterations T.

Gradient descent algorithm

- How many iterations to run?
- Several alternative strategies:
  - Train for a fixed number of iterations T.
  - Train until the difference in training cost diminishes below a threshold δ:

$$|f(\mathbf{w}^{(t-1)}) - f(\mathbf{w}^{(t)})| < \delta$$

Gradient descent algorithm

- How many iterations to run?
- Several alternative strategies:
  - Train for a fixed number of iterations T.
  - Train until the difference in training cost diminishes below a threshold δ:

$$|f(\mathbf{w}^{(t-1)}) - f(\mathbf{w}^{(t)})| < \delta$$

  - Train until the loss/accuracy on a validation set (coming soon) no longer improves.

Gradient descent demos

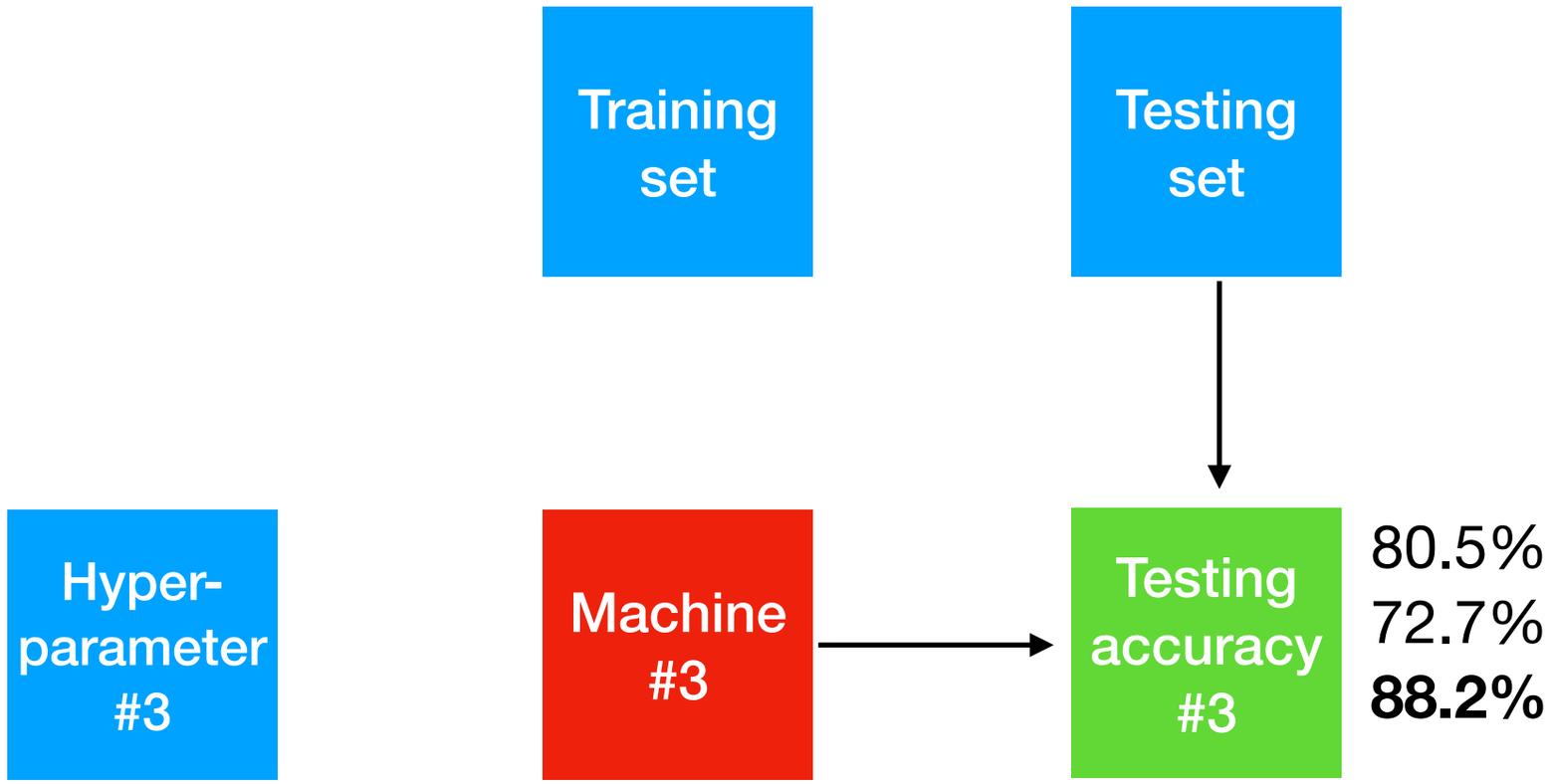
- 1-d
- 2-d

Hyperparameters

- T and δ are called **hyperparameters** — they are not part of the machine itself, but rather affect how the machine is *trained*.
- The choice of hyperparameters can make a big impact on performance; recall the discussion of “implicit cheating” from Lecture 3…

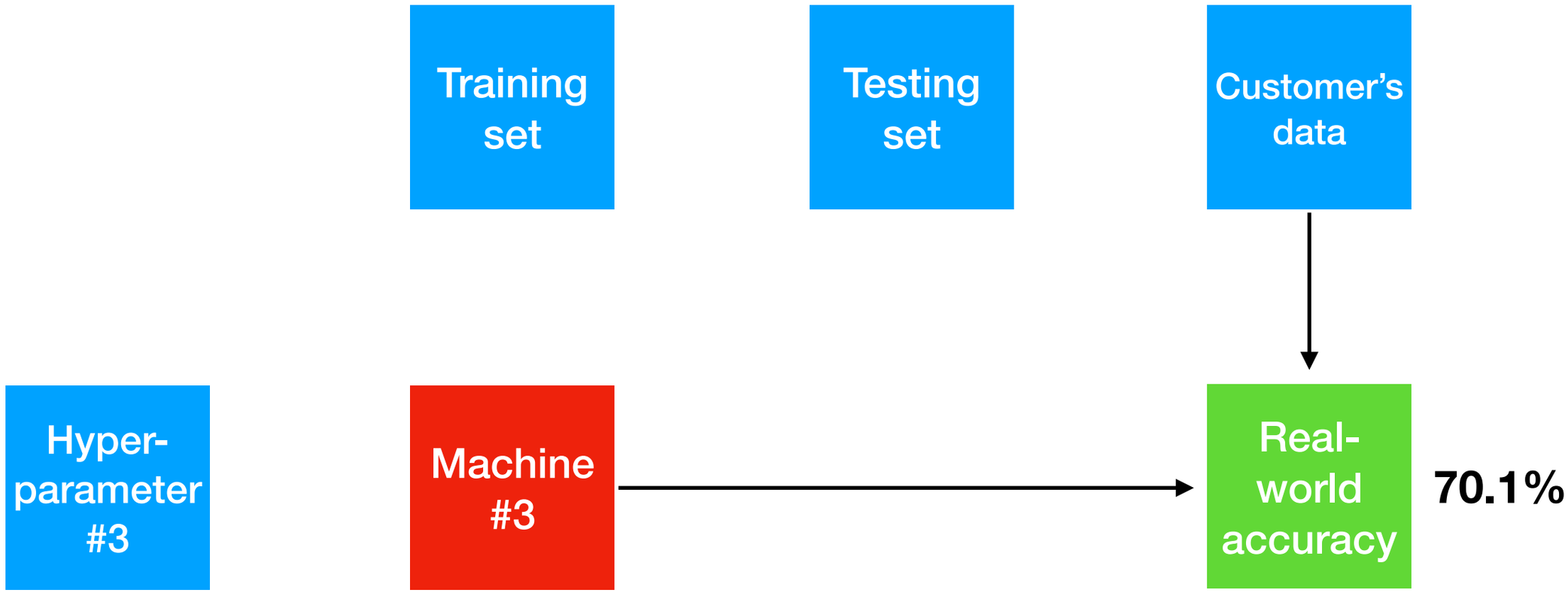
Implicit cheating

- Much better! Let's keep machine #3 and sell it on Amazon!



Implicit cheating

- Oops — the real-world accuracy was much less than what we estimated on the test set!



Hyperparameter optimization

- The proper way to optimize hyper parameters is to use a separate validation set.

- I.e., from our combined dataset that we collect, we now have:

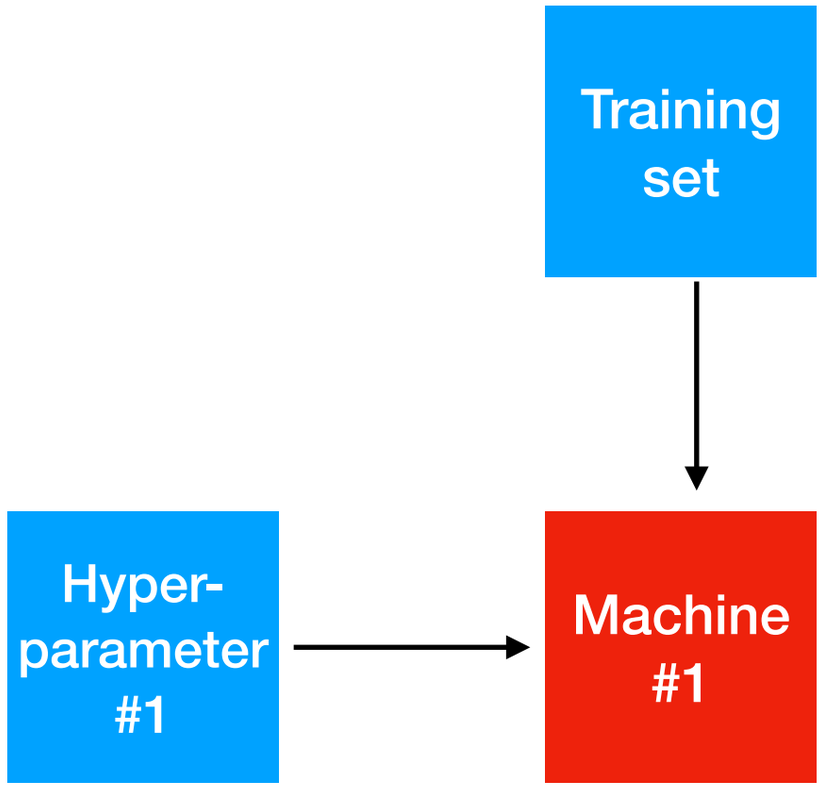
  - **Training data**: used to estimate best model parameters (e.g., **w**)

  - **Validation data**: used to estimate best hyperparameters (e.g., *T*)

  - **Testing data**: used to characterize performance of the final trained machine.

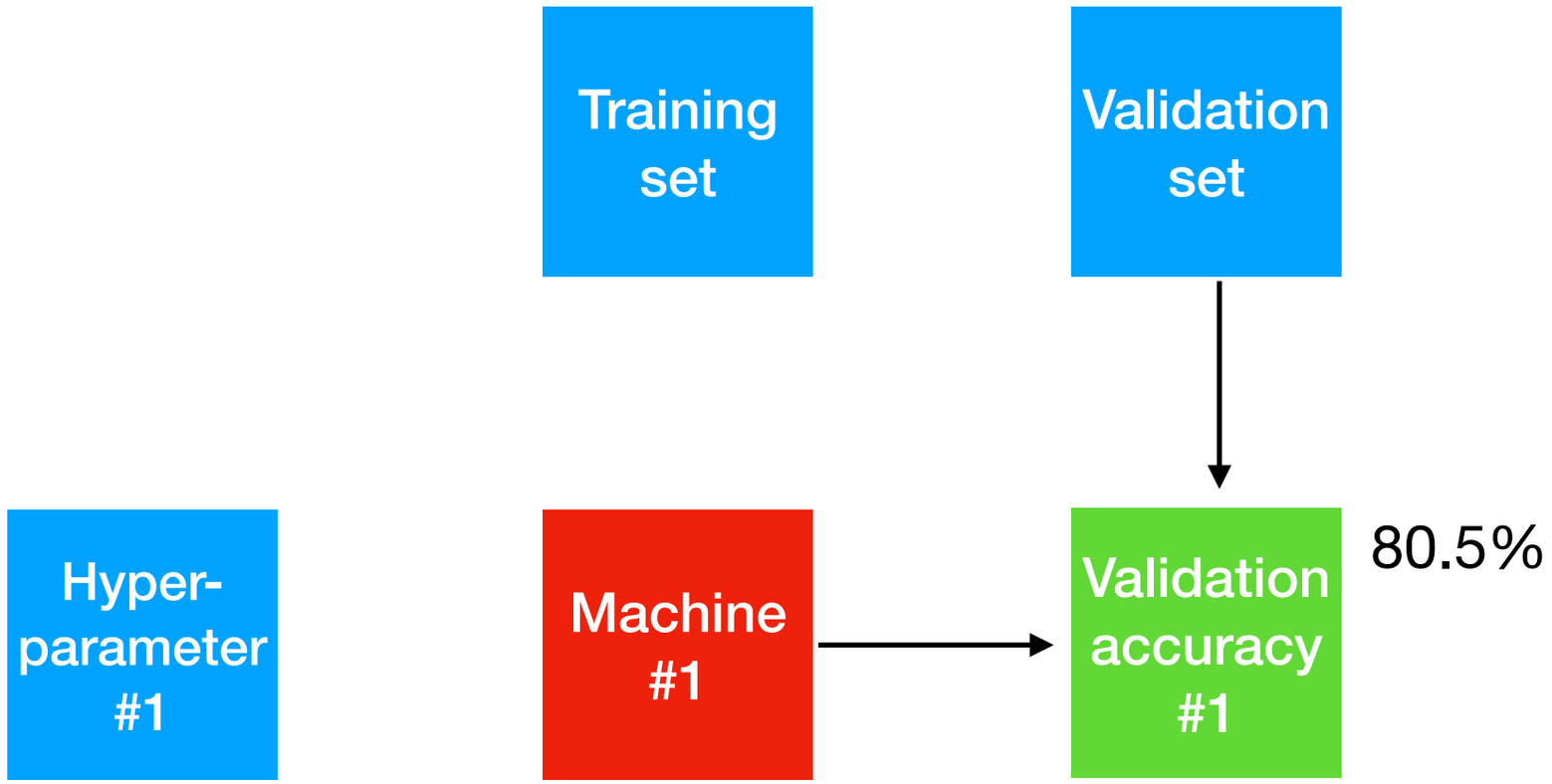
Proper hyperparameter optimization

- Select hyperparameters; use them to train a machine on the training data.



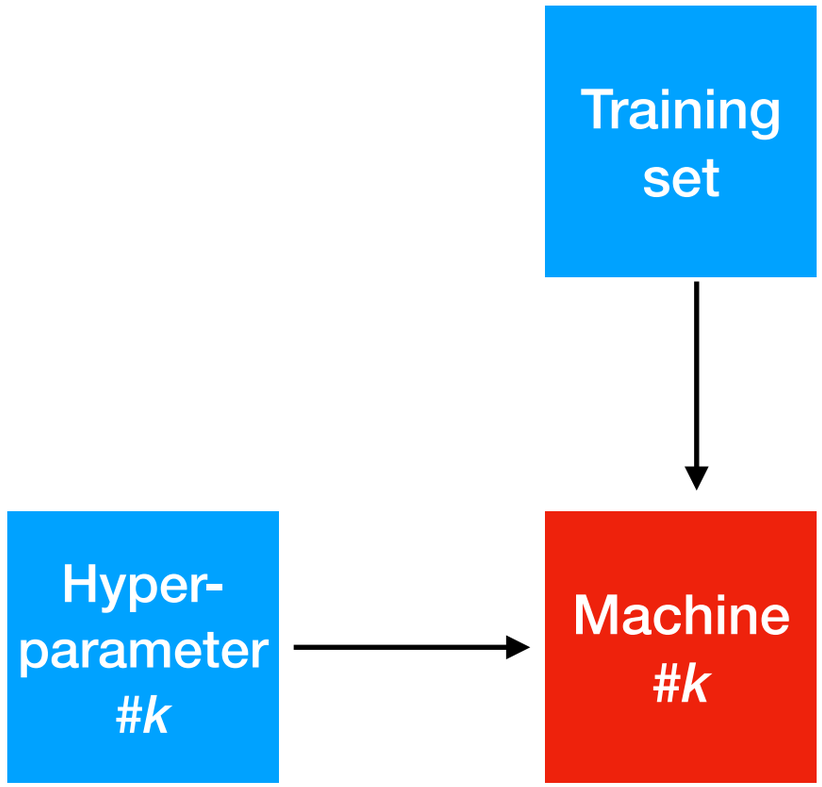
Proper hyperparameter optimization

- Estimate performance on the validation set.



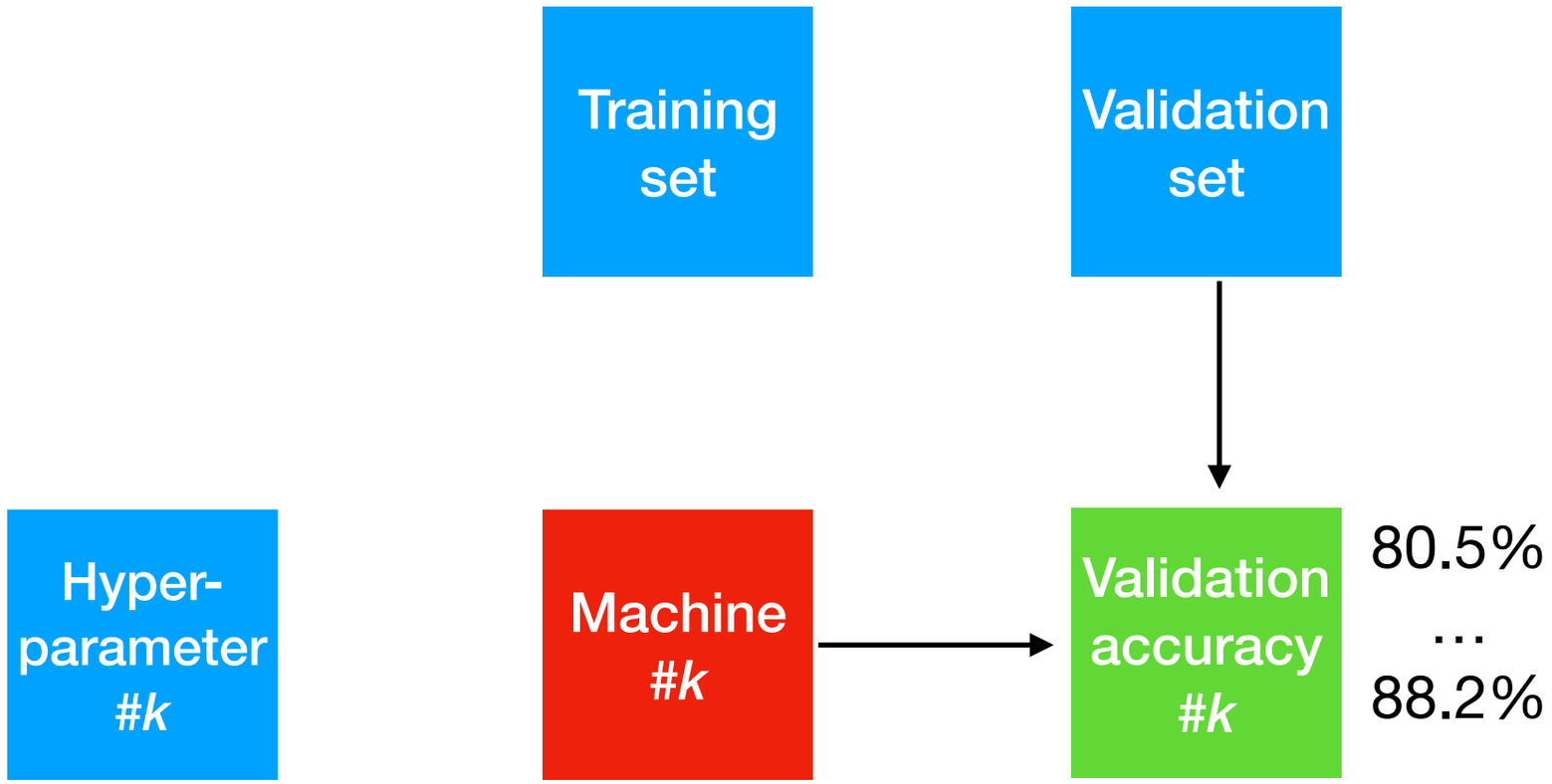
Proper hyperparameter optimization

- Repeat for every hyperparameter value under consideration.



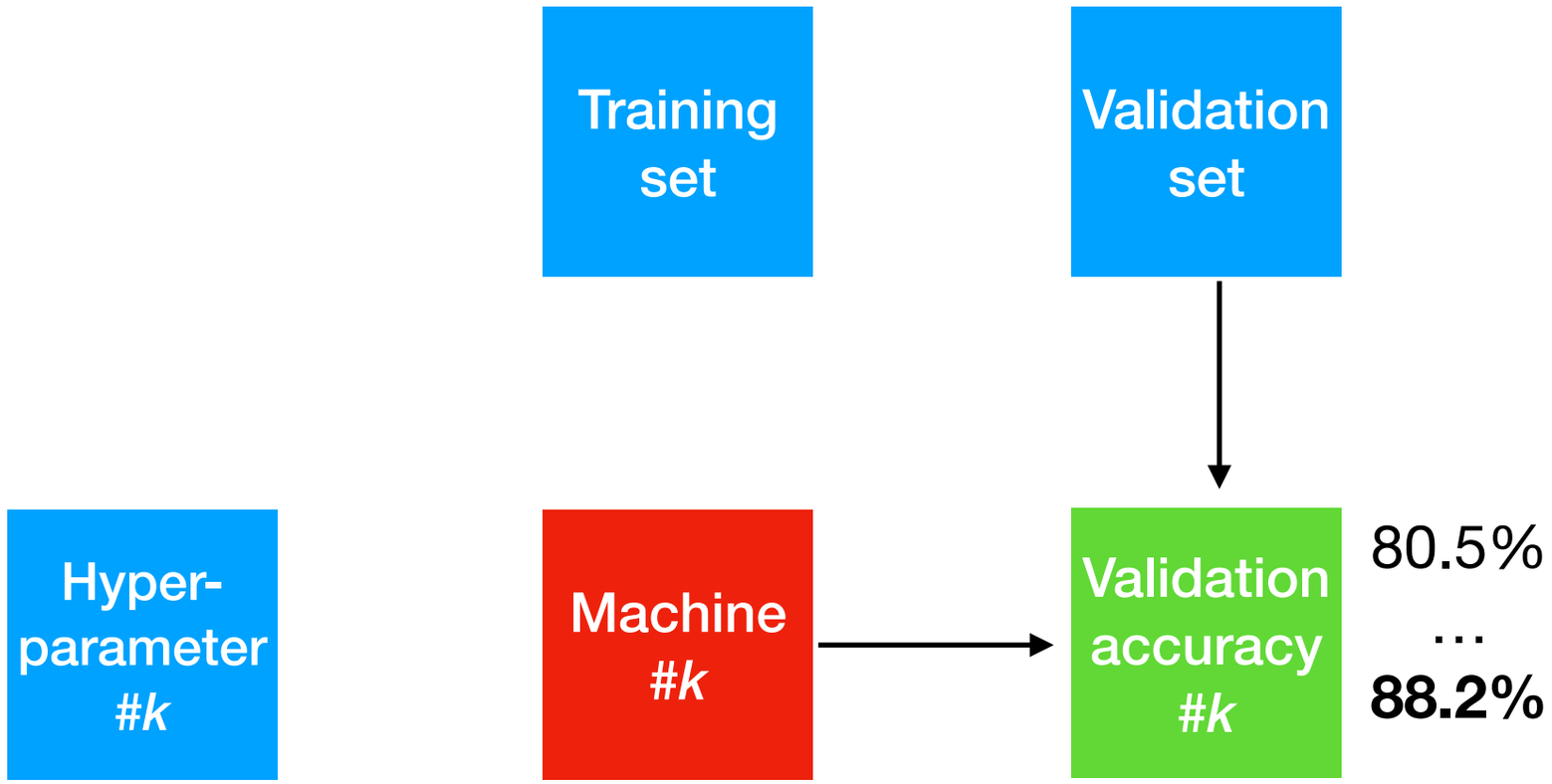
Proper hyperparameter optimization

- Repeat for every hyperparameter value under consideration.



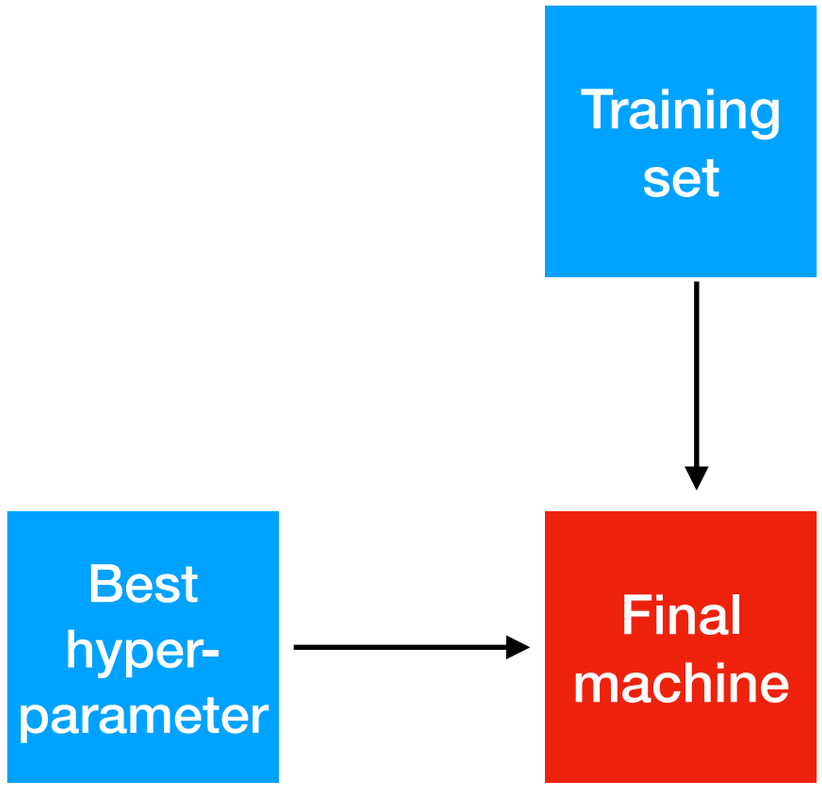
Proper hyperparameter optimization

- Freeze the hyperparameter setting that gave the best performance on the validation set.



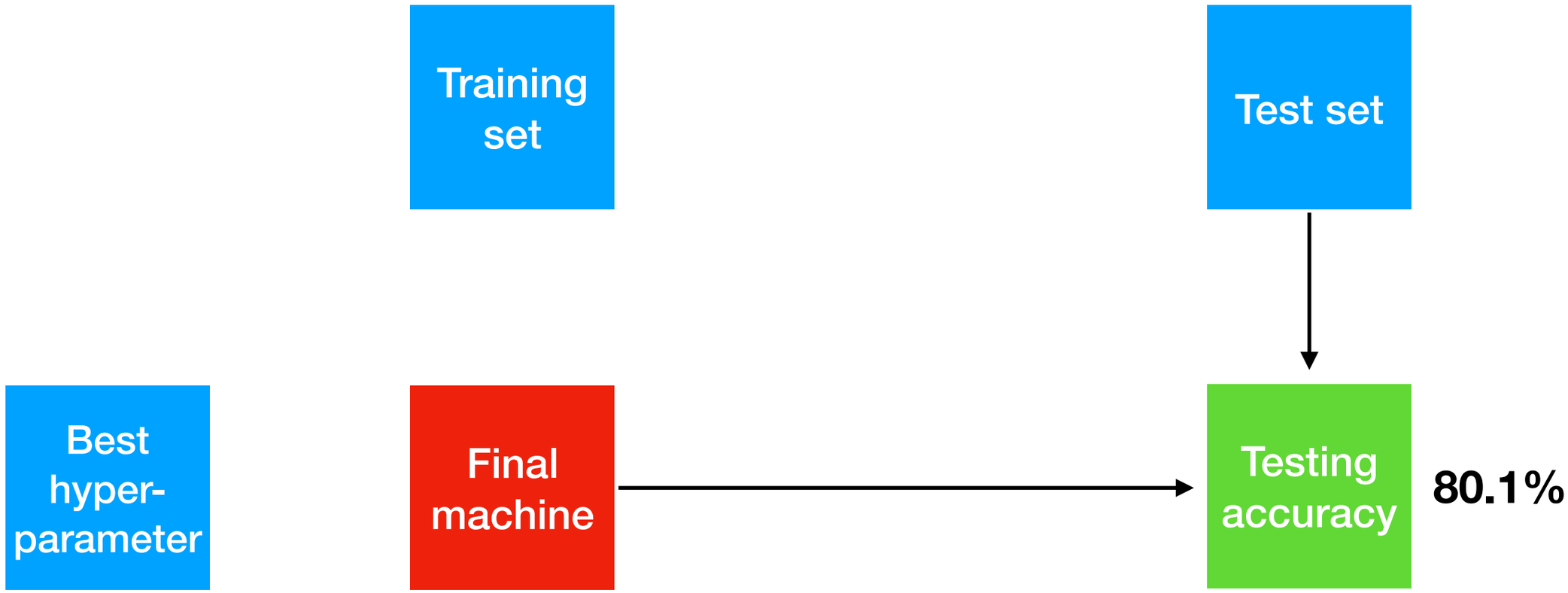
Proper hyperparameter optimization

- Use that hyperparameter to train one more machine.



Proper hyperparameter optimization

- Measure the machine's performance on the test set.



Polynomial regression

Linear regression

- Linear regression is efficient to optimize and very useful, but is limited in its expressiveness.
- Unsurprisingly, it can only model “linear” (technically *affine*) relationships:

$$\hat{y} = \mathbf{x}^\top \mathbf{w} + b$$

- In 1-d:

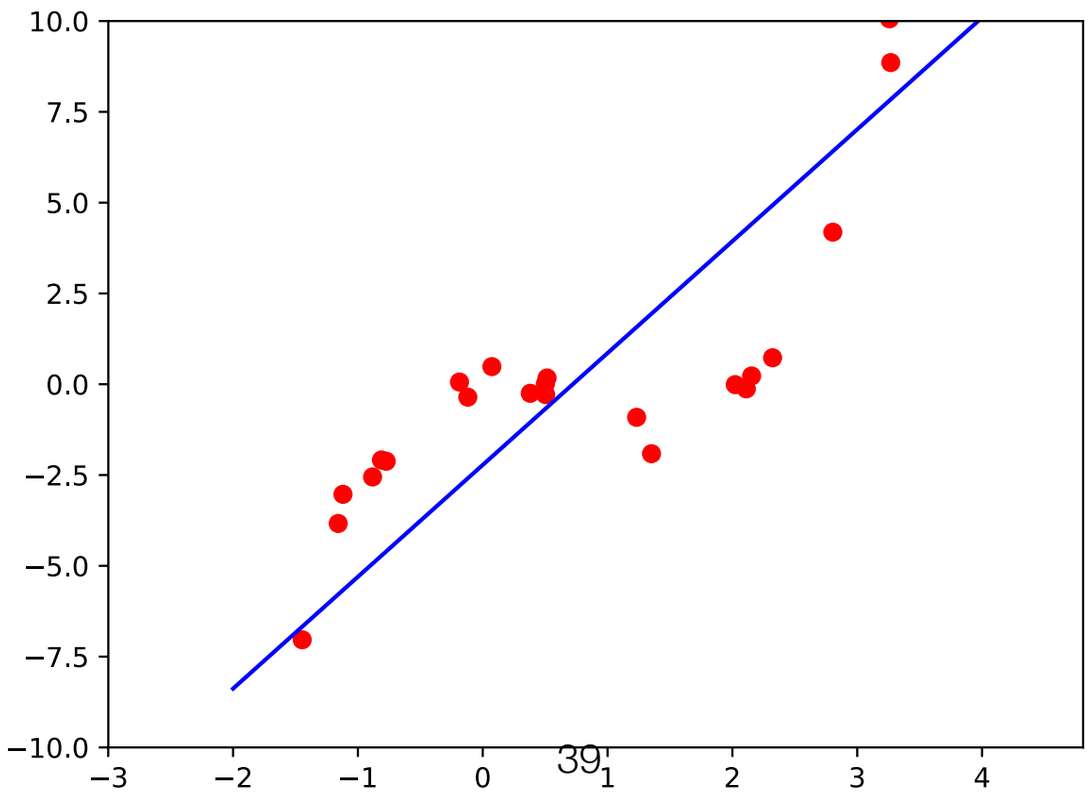
$$\hat{y} = wx + b$$

Linear regression

- But sometimes the target values *y* have a non-linear relationship with the input **x**.

- Linear regression may not do a good job then.

- Example: $y = 0.2x - 1.8x^2 + 0.8x^3 + \text{noise}$



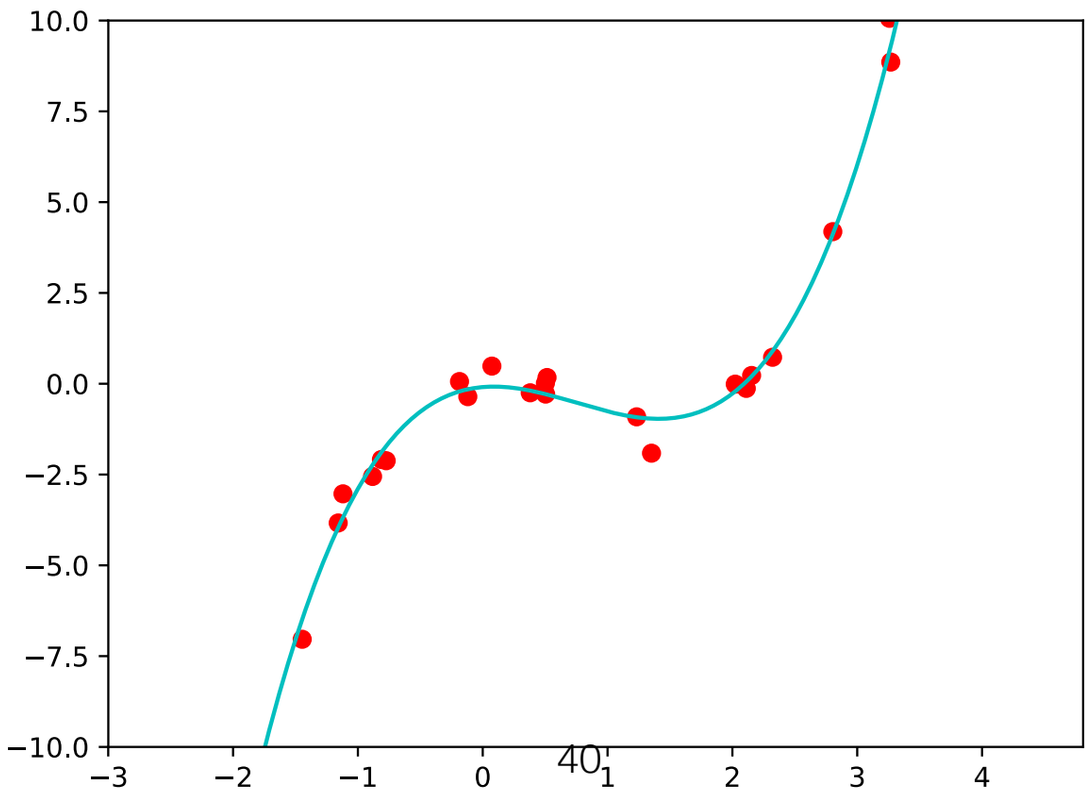
Not a great fit.

Polynomial regression

- If the labels *y* are a polynomial function of the inputs *x*, why not enable the model to express polynomial relationships?

- In 1-d, we can build a *cubic* regression model as follows:

$$\begin{aligned}\hat{y} &= w_1 x^1 + w_2 x^2 + w_3 x^3 + b \\ &= w_0 x^0 + w_1 x^1 + w_2 x^2 + w_3 x^3\end{aligned}$$



Much better fit!

Polynomial regression

- How do we train the weights of the polynomial regression (for 1-d inputs)?
  - Pretend that each power of *x* is a *separate feature*.
  - From the scalar input *x*, form the vector
    $\mathbf{x} = [\, x^0, x^1, x^2, x^3 \,]^T$

Polynomial regression

- How do we train the weights of the polynomial regression (for 1-d inputs)?
  - Pretend that each power of x is a *separate feature*.
  - From the scalar input x, form the vector $\mathbf{x} = [\, x^0, x^1, x^2, x^3 \,]^T$
- Example:
  - Suppose the raw input $x = -1.5$.
  - Then $x^0 = 1$, $x^1 = -1.5$, $x^2 = 2.25$, and $x^3 = -3.375$. Hence, $\mathbf{x} = [\, 1, -1.5, 2.25, -3.375 \,]^T$.

Polynomial regression

- Now, notice that:

$$\hat{y} = w_0 x^0 + w_1 x^1 + w_2 x^2 + w_3 x^3$$

Polynomial regression

- Now, notice that:

$$\begin{aligned}\hat{y} &= w_0 x^0 + w_1 x^1 + w_2 x^2 + w_3 x^3 \\ &= \begin{bmatrix} x^0 & x^1 & x^2 & x^3 \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ w_3 \end{bmatrix}\end{aligned}$$

Polynomial regression

- Now, notice that:

$$\begin{aligned}
\hat{y} &= w_0 x^0 + w_1 x^1 + w_2 x^2 + w_3 x^3 \\
&= \begin{bmatrix} x^0 & x^1 & x^2 & x^3 \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ w_3 \end{bmatrix} \\
&= \mathbf{x}^\top \mathbf{w}
\end{aligned}$$

When we "pre-compute" each power of *x*, we convert the polynomial regression back into a linear regression model.

Polynomial regression

- We can convert each input *x* into a feature vector **x**, and create a design matrix **X**, and compute the optimal **w** as before…

Polynomial regression

- Suppose we have *n*=3 training examples whose values are -1.5, -1, and 3.25.

i=1	*i*=2	*i*=3
-1.5	**-1**	**3.25**

Polynomial regression

- Suppose we have *n*=3 training examples whose values are -1.5, -1, and 3.25.
- Then for each (scalar) *x* we build a vector **x** consisting of $[\, x^0, x^1, x^2, x^3 \,]^T$:

	i=1	*i*=2	*i*=3
d=0	1	1	1
d=1	**-1.5**	**-1**	**3.25**
d=2	2.25	1	10.5625
d=3	-3.375	-1	4.32812

Polynomial regression

- The matrix of all our examples (as column vectors) constitutes the design matrix **X**, as usual.

$$\mathbf{X} = \begin{bmatrix} 1 & 1 & 1 \\ -1.5 & -1 & 3.25 \\ 2.25 & 1 & 10.5625 \\ -3.375 & -1 & 4.32812 \end{bmatrix}$$

Polynomial regression

- The matrix of all our examples (as column vectors) constitutes the design matrix **X**, as usual.

- We can now find the optimal polynomial regression coefficients by computing:

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

- …just like with linear regression.

$$\mathbf{X} = \begin{bmatrix} 1 & 1 & 1 \\ -1.5 & -1 & 3.25 \\ 2.25 & 1 & 10.5625 \\ -3.375 & -1 & 4.32812 \end{bmatrix}$$

Overfitting and regularization

Overfitting

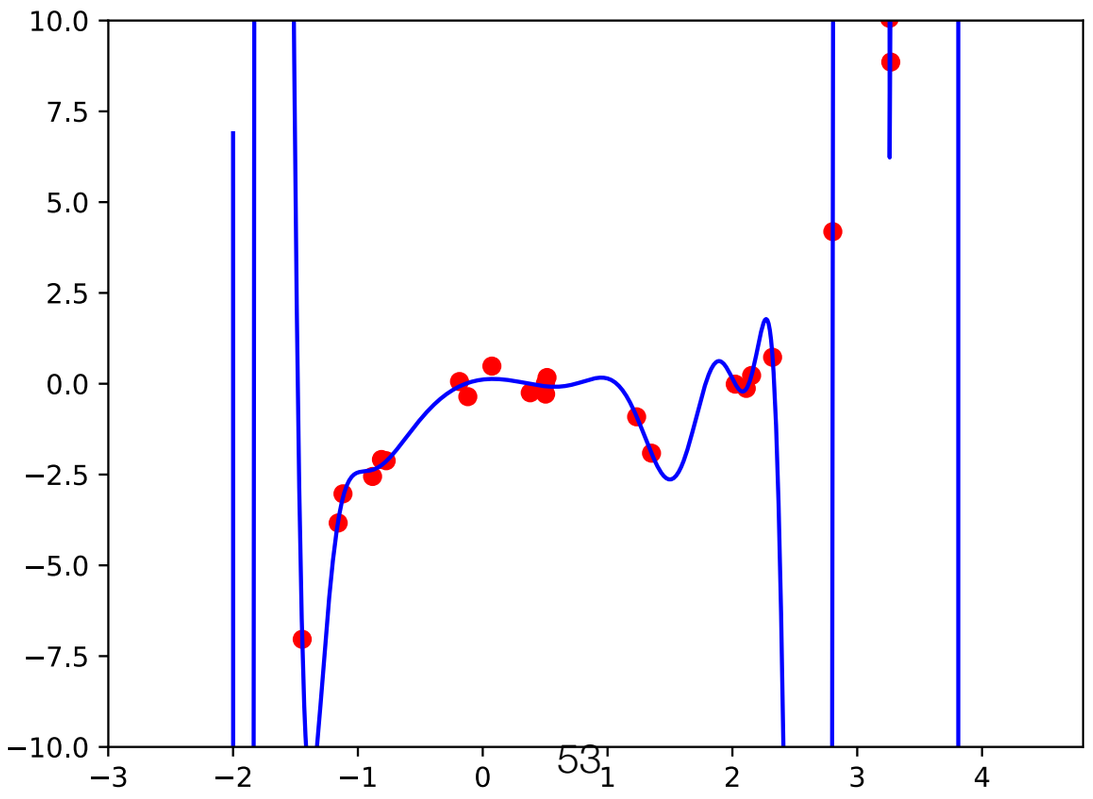
- If polynomial regression with degree 3 worked well, why not increase the degree even higher?

- Let’s try with degree 25…

Overfitting

- If polynomial regression with degree 3 worked well, why not increase the degree even higher?

- Let's try with degree 25…



We nailed almost every point exactly!… but maybe this is overkill?

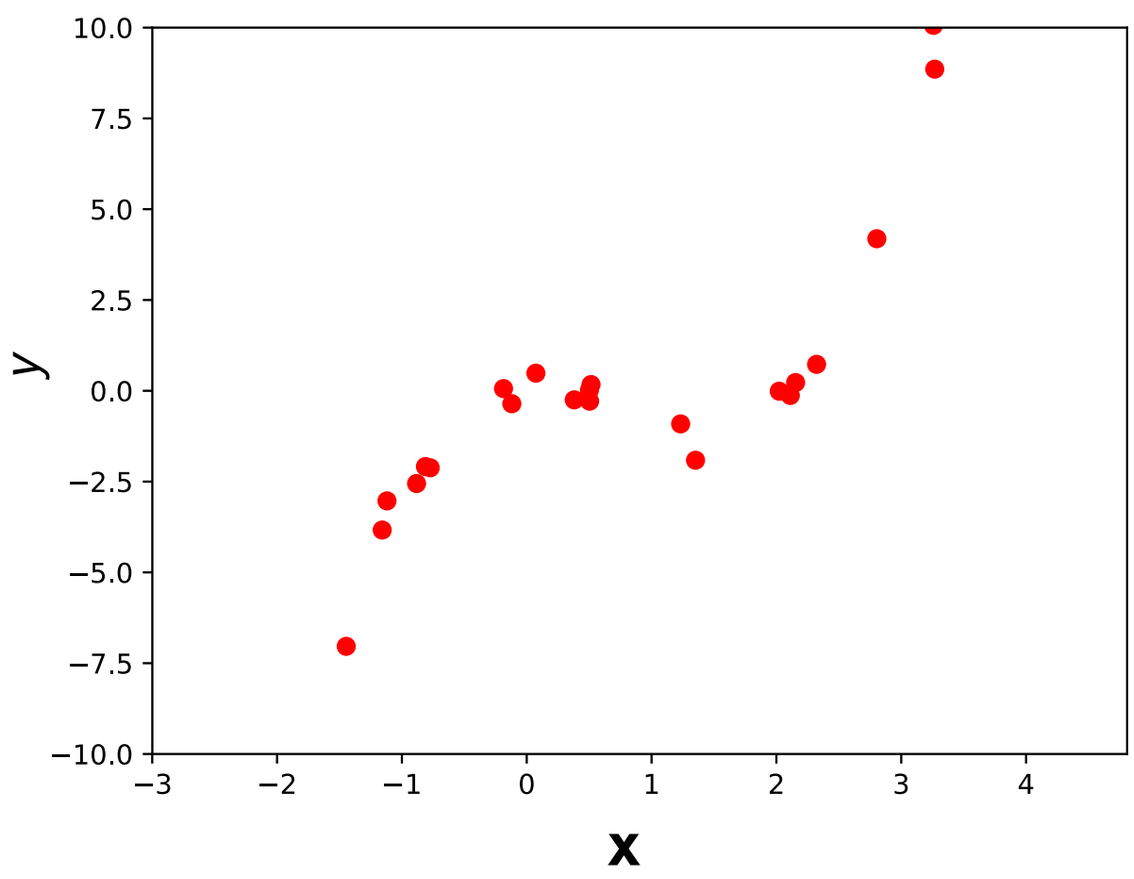
Overfitting

- Why is this bad? Recall that **overfitting** means that training error is low, but testing error is high.

- **Testing error** represents how well we expect our machine to perform on data we have **not seen before**.

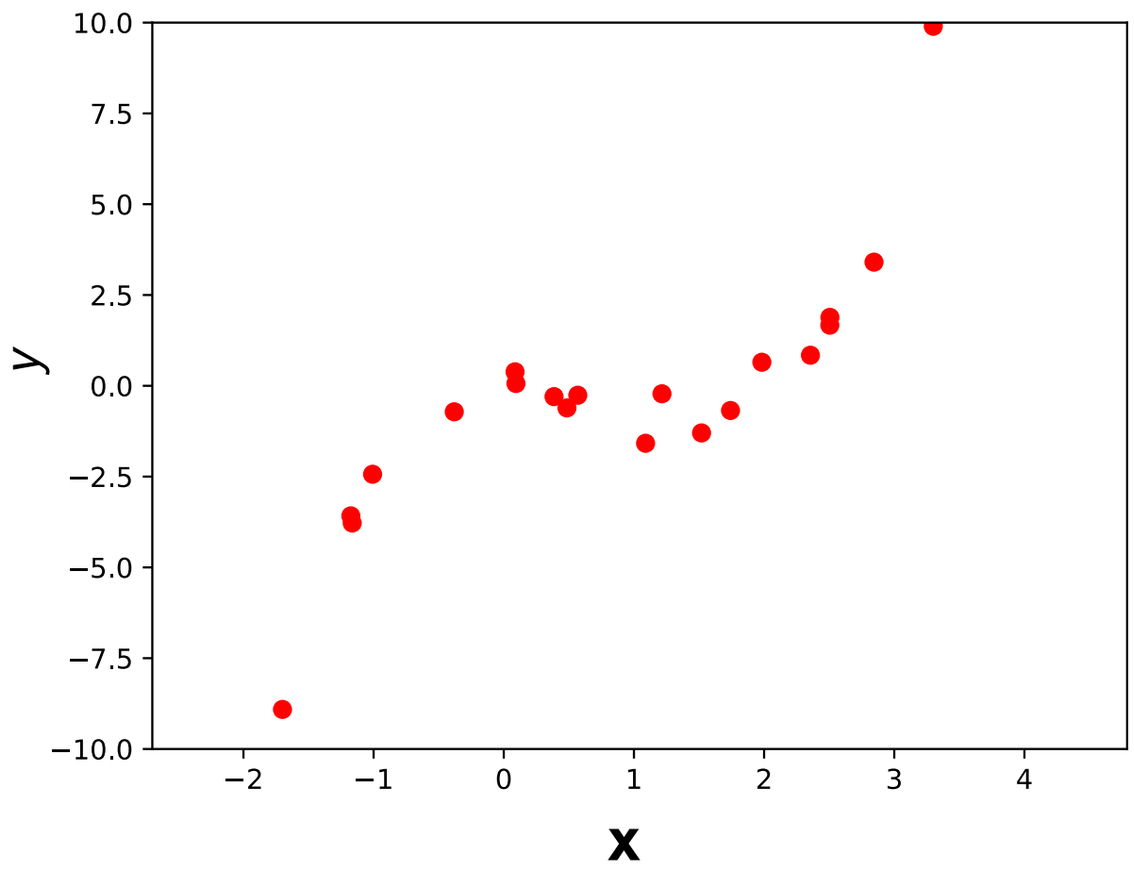
Overfitting

- Suppose this scatter plot represents the *training* set:



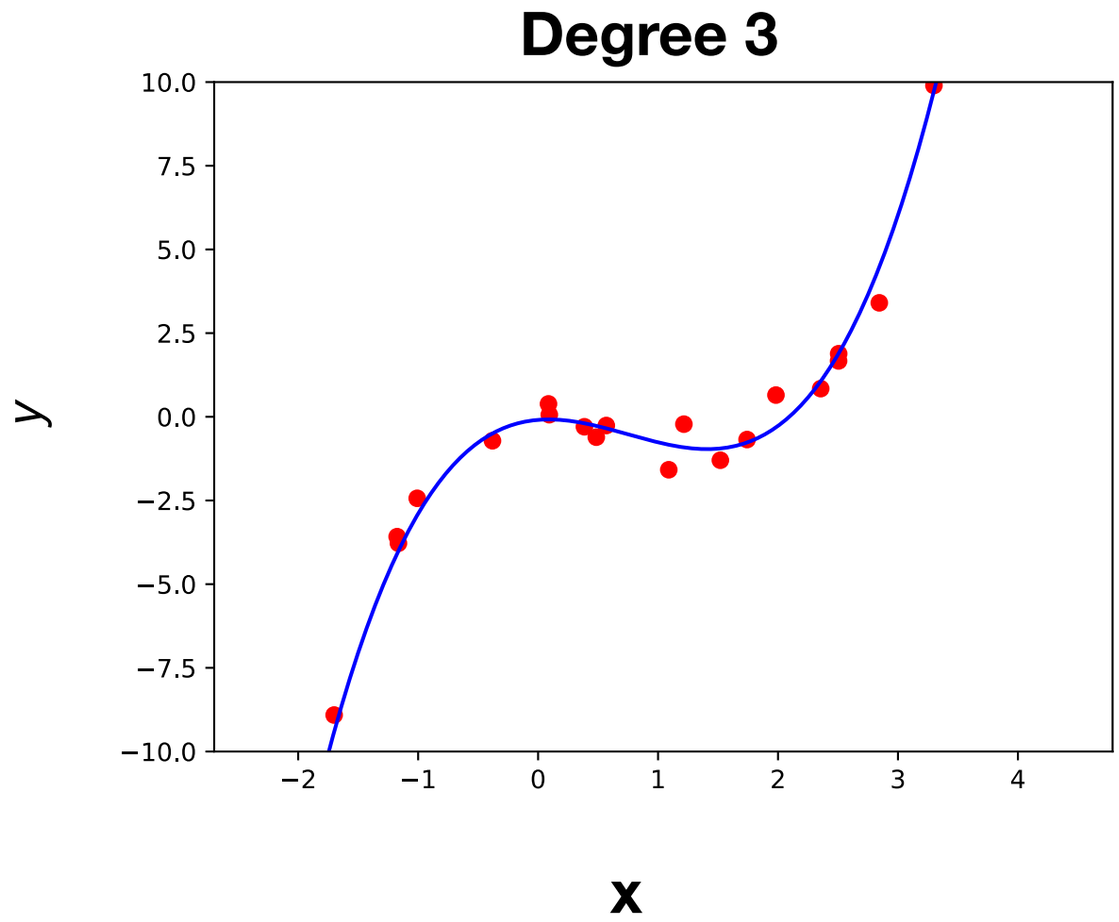
Overfitting

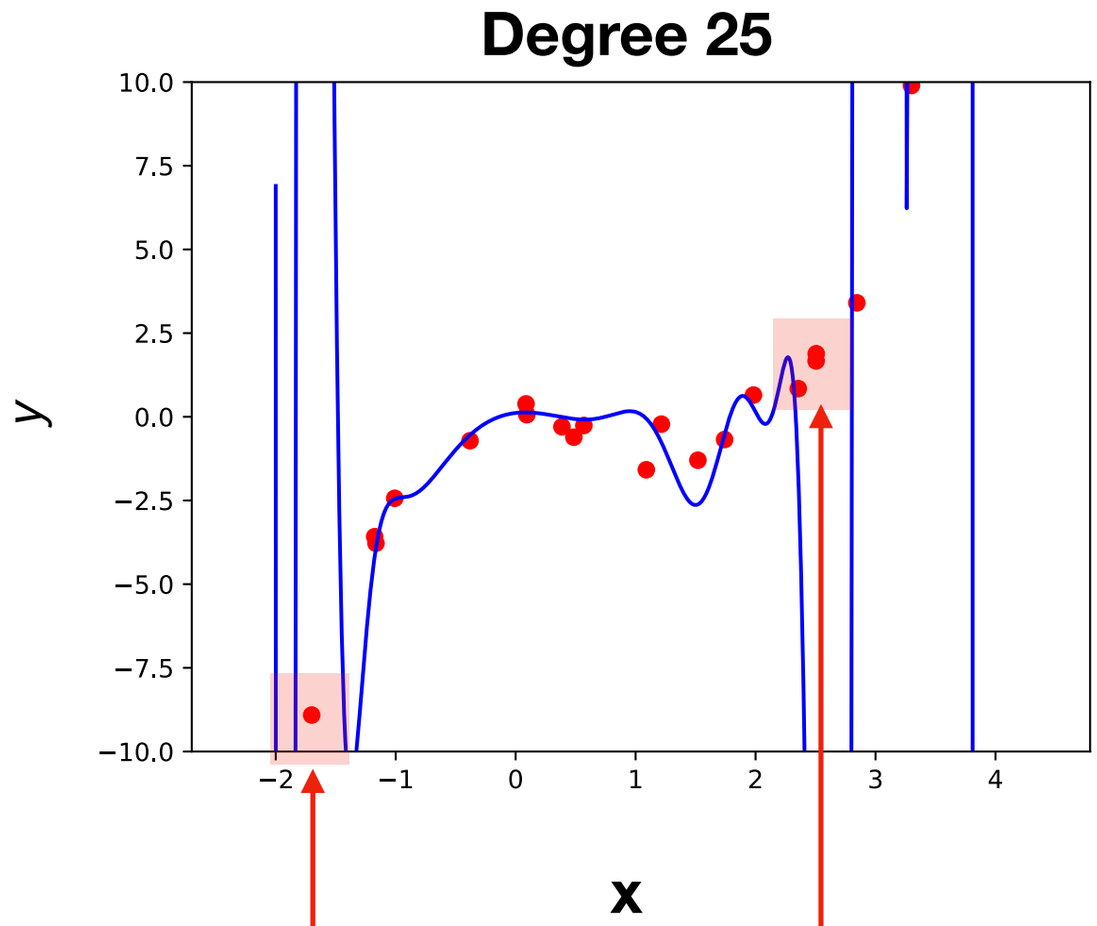
- Suppose this scatter plot represents the *testing* set:



Overfitting

- Here are the machine's predictions on the testing set using poly. regression, with either degree 3 or degree 25:





For these data points, the predictions are very inaccurate, which makes f_{MSE} large.

Overfitting

- Why is this bad? Recall that **overfitting** means that training error is low, but testing error is high.

- **Testing error** represents how well we expect our machine to perform on data we have **not seen before**.

Preventing overfitting

- How to prevent this? Two strategies:
  - Keep the *degree d* of the polynomial modest.

Preventing overfitting

- How to prevent this? Two strategies:

- Keep the *degree d* of the polynomial modest.

- Keep the *weight* associated with each term modest.

$$\hat{y} = w_0 x^0 + w_1 x^1 + w_2 x^2 + \ldots + w_d x^d$$

Random polynomials

- Let's generate some polynomials by randomly selecting each w_i in:

$$\hat{y} = w_0 x^0 + w_1 x^1 + w_2 x^2 + \ldots + w_d x^d$$

- Compute the average squared coefficient as:

$$\mu = \frac{1}{d+1} \sum_{i=0}^{d} w_i^2$$

- We will generate random polynomials for different degrees d and different coefficient magnitudes μ.

Random polynomials

$$\mu = \frac{1}{d+1}\sum_{i=0}^{d} w_i^2$$

- Examples:

$$d = 2: \quad \hat{y} = 4 + 2x - 2x^2 \qquad \mu = \quad \textbf{?}$$
$$d = 4: \quad \hat{y} = x^2 + 0.5x^3 - 2x^4 \qquad \mu = \quad \textbf{?}$$

Random polynomials

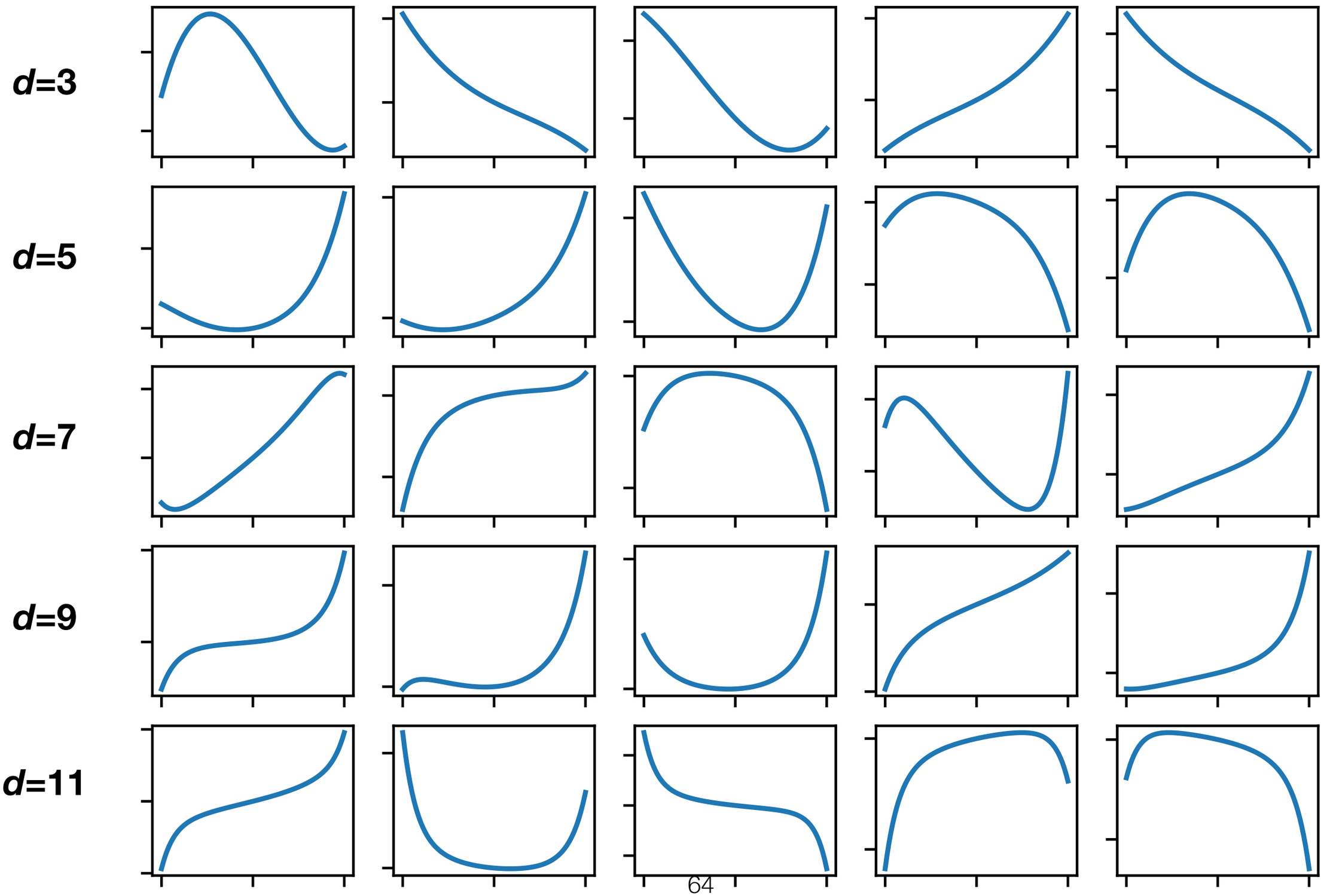
$$\mu = \frac{1}{d+1} \sum_{i=0}^{d} w_i^2$$

- Examples:

$$\begin{aligned} d = 2: \quad & \hat{y} = 4 + 2x - 2x^2 && \mu = 24/3 = 8 \\ d = 4: \quad & \hat{y} = x^2 + 0.5x^3 - 2x^4 && \mu = 5.25/5 = 1.05 \end{aligned}$$

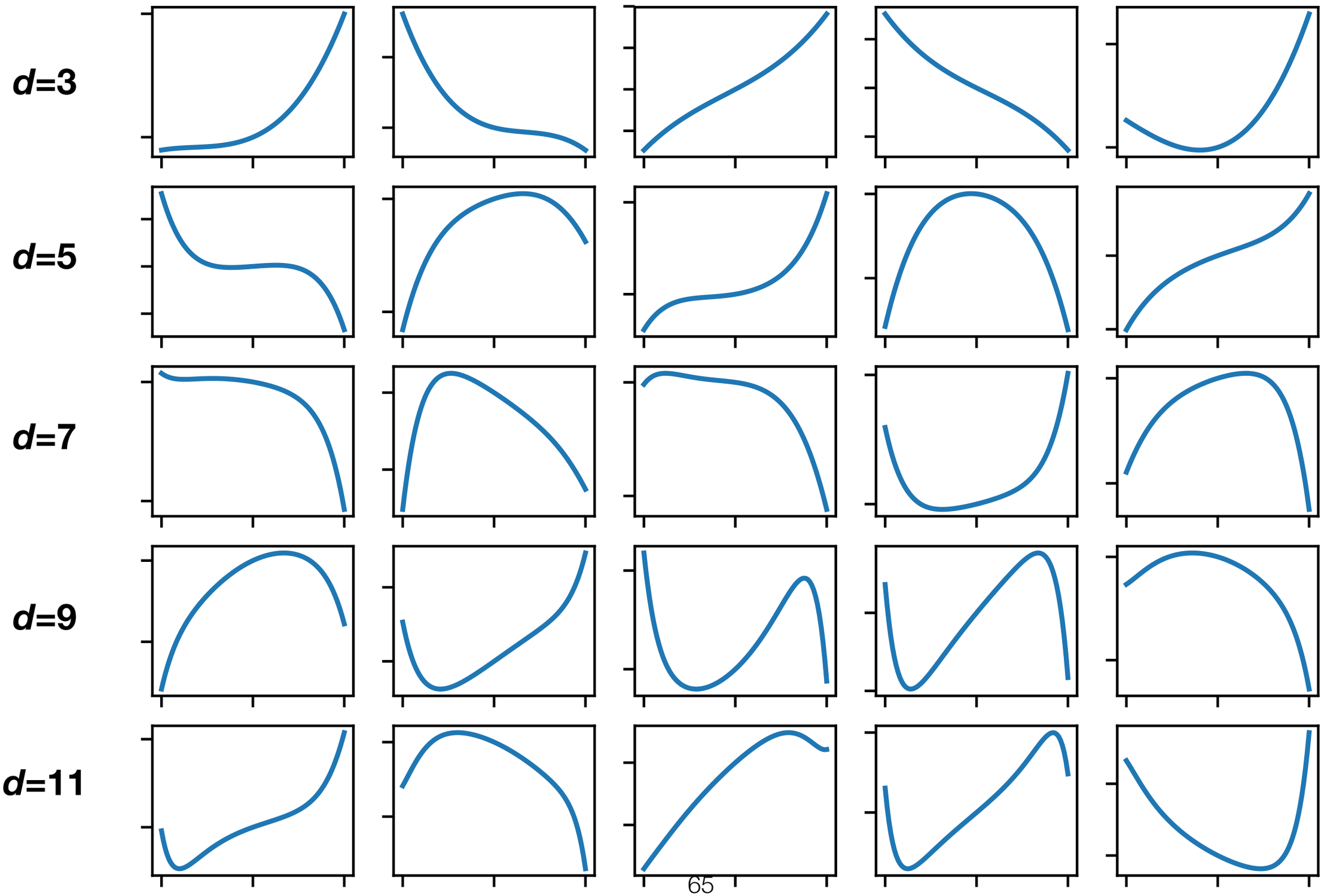
Random polynomials

μ=7.25e-07



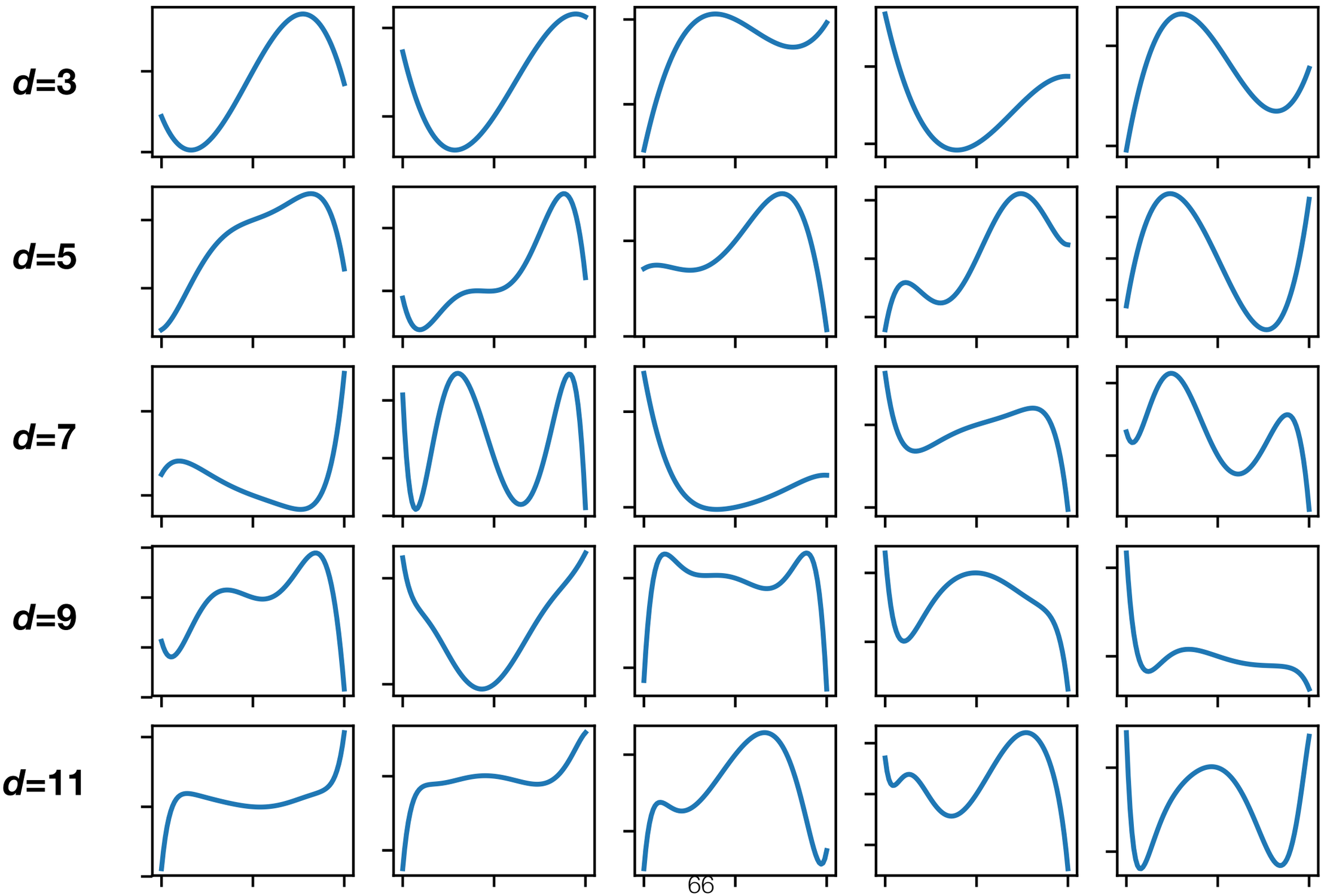
Random polynomials

μ=0.00063



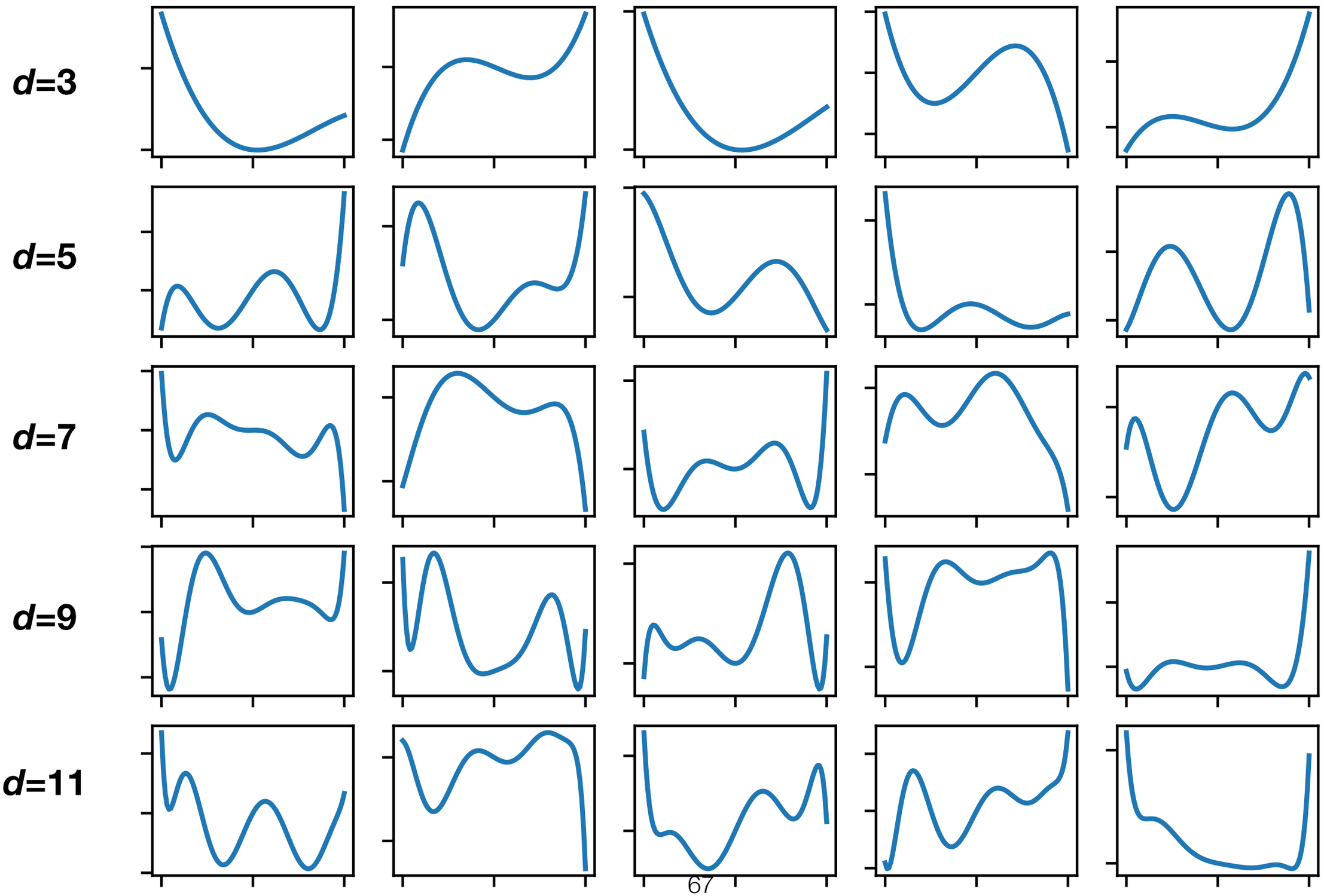
Random polynomials

μ=0.058



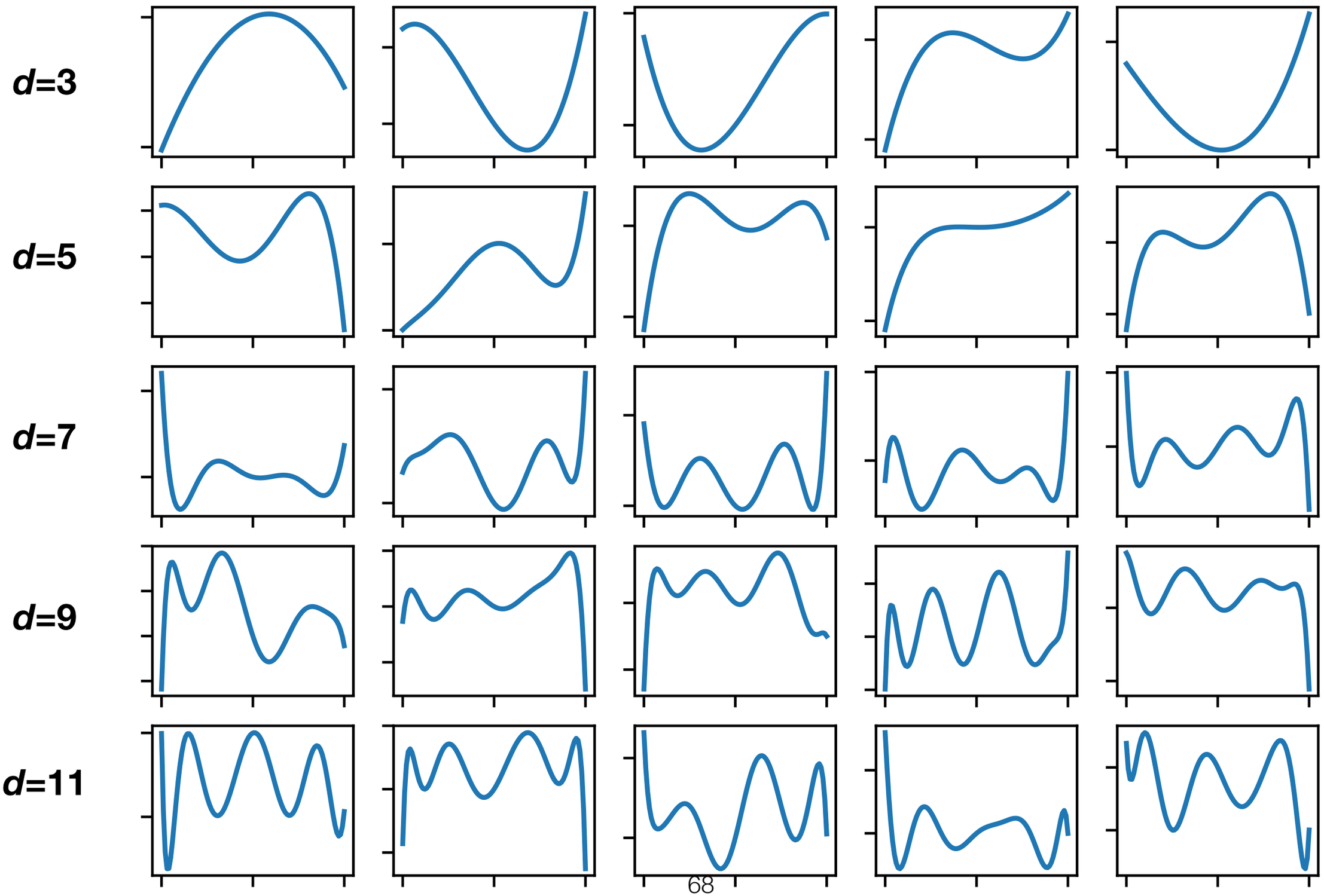
Random polynomials

μ=3.31



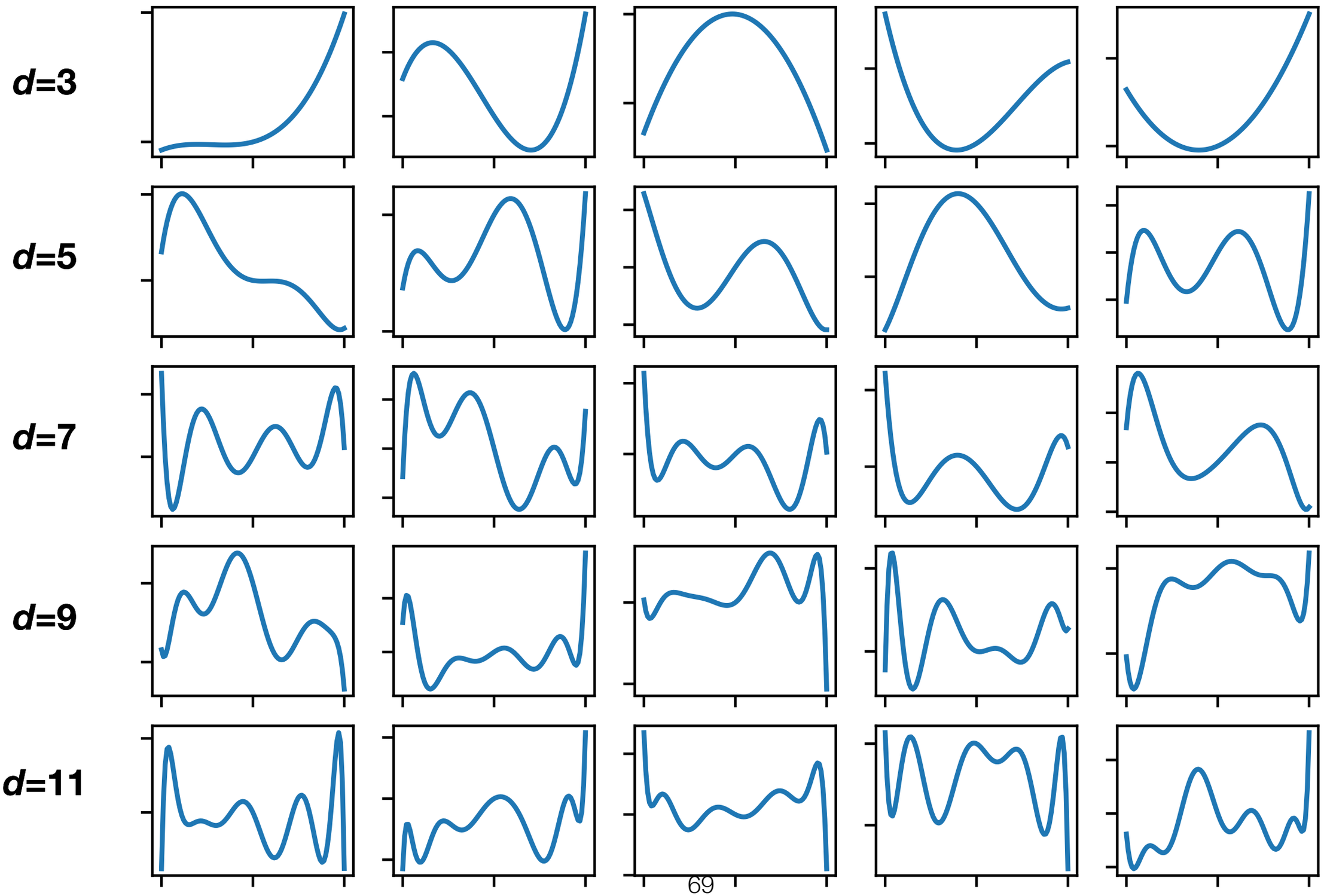
Random polynomials

μ=90.9



Random polynomials

μ=537.8



Regularization

- The larger the coefficients (weights) **w** are allowed to be, the more the polynomial regressor can overfit.

- If we “encourage” the weights to be small, we can reduce overfitting.

- This is a form of **regularization** — any practice designed to improve the machine’s ability to **generalize** to new data.

Regularization

- One of the simplest and oldest regularization techniques is to *penalize* large weights in the cost function.
- We can define an extra cost:

$$\sum_{i=1}^{m} w_i^2 = \mathbf{w}^\top \mathbf{w}$$

- This is called a ***L_2* regularization term**.

Regularization

- The "unregularized" f_{MSE} is:

$$f_{\text{MSE}}(\mathbf{w}) = \frac{1}{2n} \sum^{n}_{i=1} (y^{(i)} - \hat{y}^{(i)})^2$$

Regularization

- The "unregularized" f_{MSE} is:

$$f_{\text{MSE}}(\mathbf{w}) = \frac{1}{2n} \sum^{n}_{i=1} (y^{(i)} - \hat{y}^{(i)})^2$$

- The ***L*$_2$-regularized** f_{MSE} becomes:

$$f_{\text{MSE}}(\mathbf{w}) = \frac{1}{2n} \sum^{n}_{i=1} (y^{(i)} - \hat{y}^{(i)})^2 + \frac{\alpha}{2n} \mathbf{w}^\top \mathbf{w}$$

- Here, α (typically a scalar) is the **regularization strength**.