Evaluation and Model Selection

Machine Learning 1 — Lecture 8

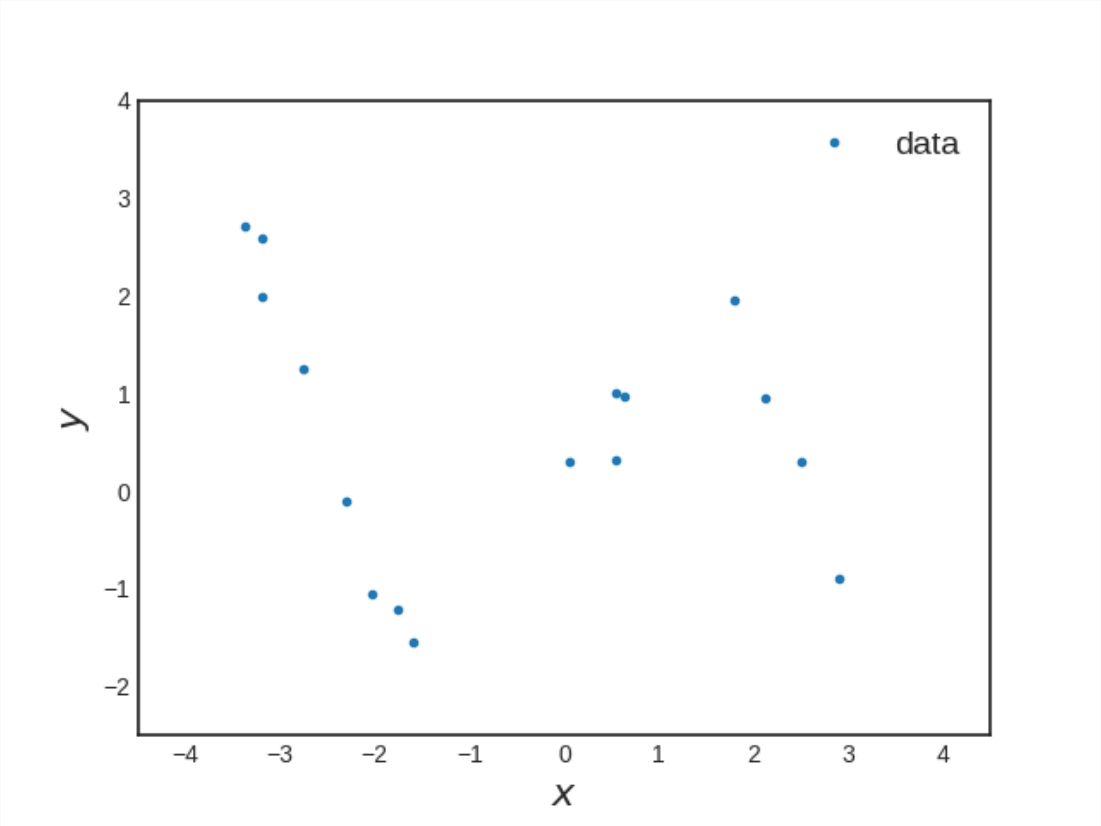
9th May 2023

Robert Peharz
Institute of Theoretical Computer Science
Graz University of Technology

Linear Regression Problem

Example

In your new job as data analyst, you get a simple regression problem, where you want to predict y from x. You already have some data ($N = 16$) which looks like this:

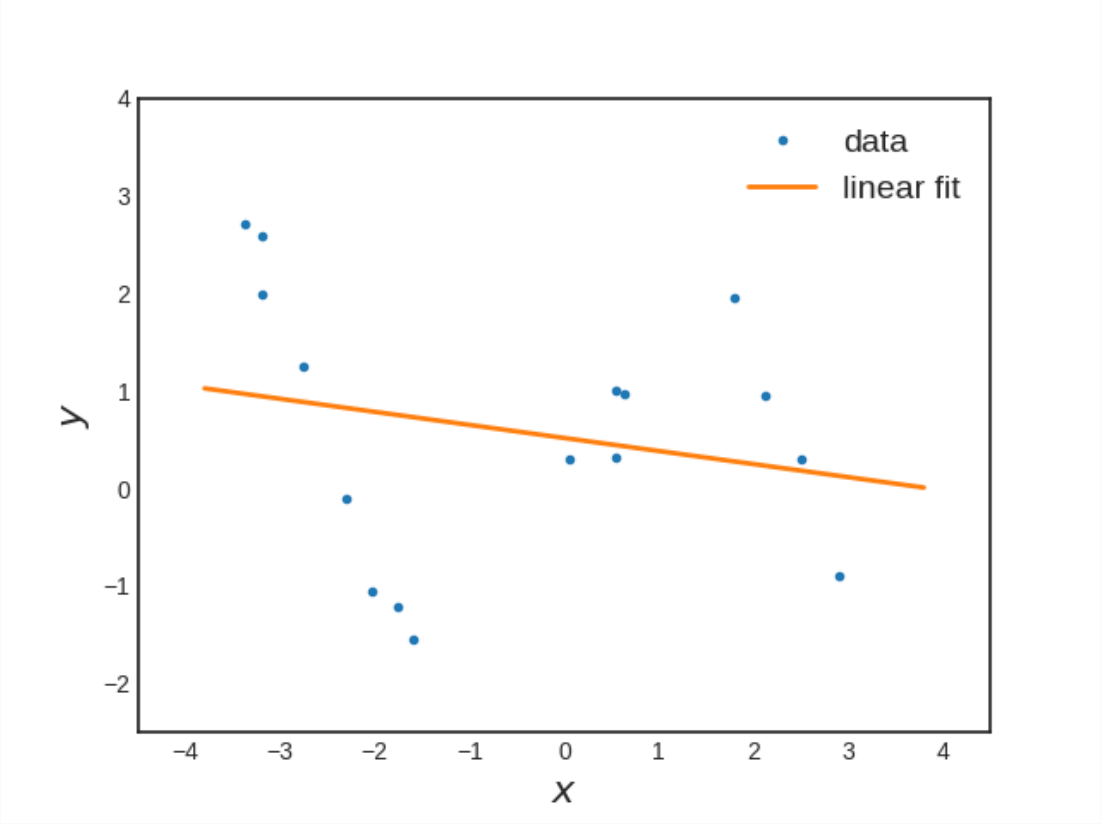


Linear Regression Problem cont'd

Example

You consult your ML1 notes and recall **linear regression**. You construct the design matrix $\boldsymbol{X}$ (you haven't forgotten the trick with the "dummy feature" $x_0 \equiv 1$ for the bias ;)) and the target vector $\boldsymbol{y}$, and you compute the **least squares solution** $\boldsymbol{\theta}^* = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y}$.

You get a pretty boring fit:



Linear regression delivers a linear model, but the data looks non-linear...

Then you recall that you can make linear regression **non-linear** by including **non-linear features** in your design matrix:

$$\mathbf{\Phi} = \begin{pmatrix} 1 & \phi_1(x^{(1)}) & \phi_2(x^{(1)}) & \dots \\ 1 & \phi_1(x^{(2)}) & \phi_2(x^{(2)}) & \dots \\ \vdots & \vdots & \vdots & \ddots \\ 1 & \phi_1(x^{(N)}) & \phi_2(x^{(N)}) & \dots \end{pmatrix}$$

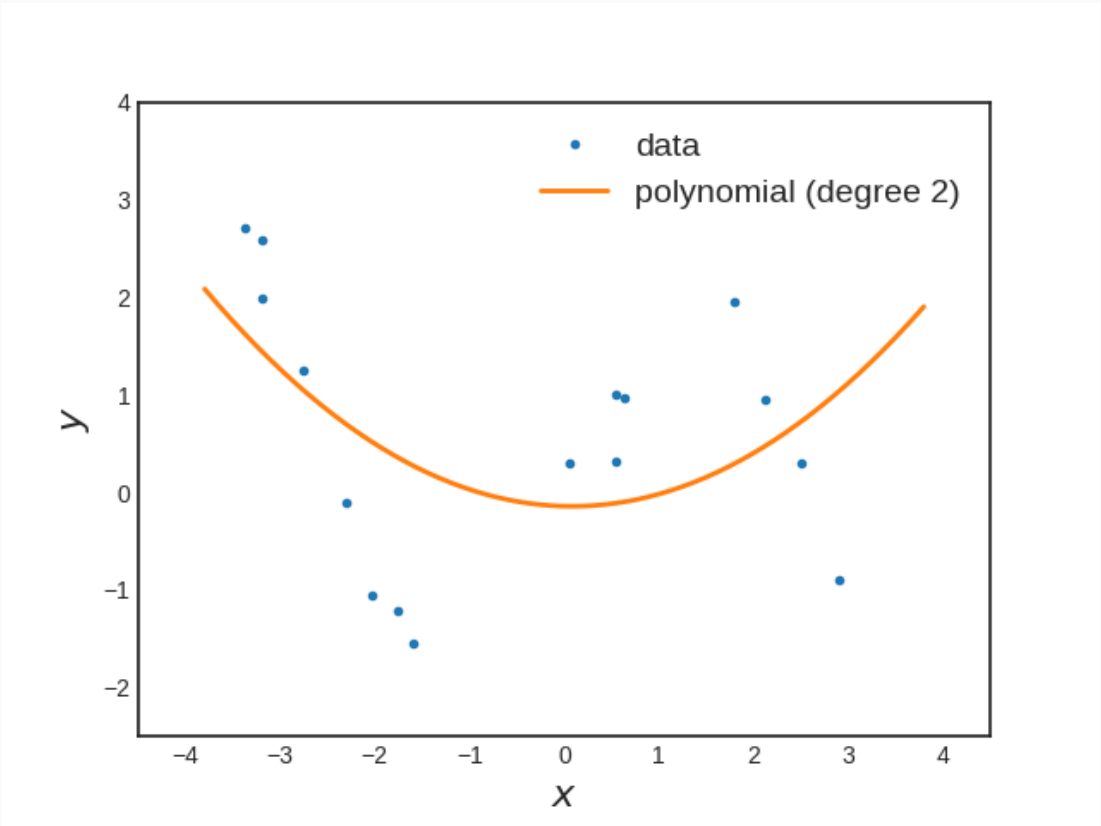
You decide to use **polynomials**, i.e. $\phi_k(x) := x^k$, so that your model becomes

$$f(x) = \theta_0 + \theta_1 \cdot x + \theta_2 \cdot x^2 + \theta_3 \cdot x^3 + \dots + \theta_K \cdot x^K$$

The **polynomial degree** K is a **hyper-parameter** of your model. The larger K, the more flexible the model.

You try $K = 2$ (**quadratic fit**). Your are happy because including non-linear functions is really no big deal to implement: you simply construct a new design matrix $\mathbf{\Phi}$ and solve: $\boldsymbol{\theta}^* = (\mathbf{\Phi}^T\mathbf{\Phi})^{-1}\mathbf{\Phi}^T\boldsymbol{y}$.

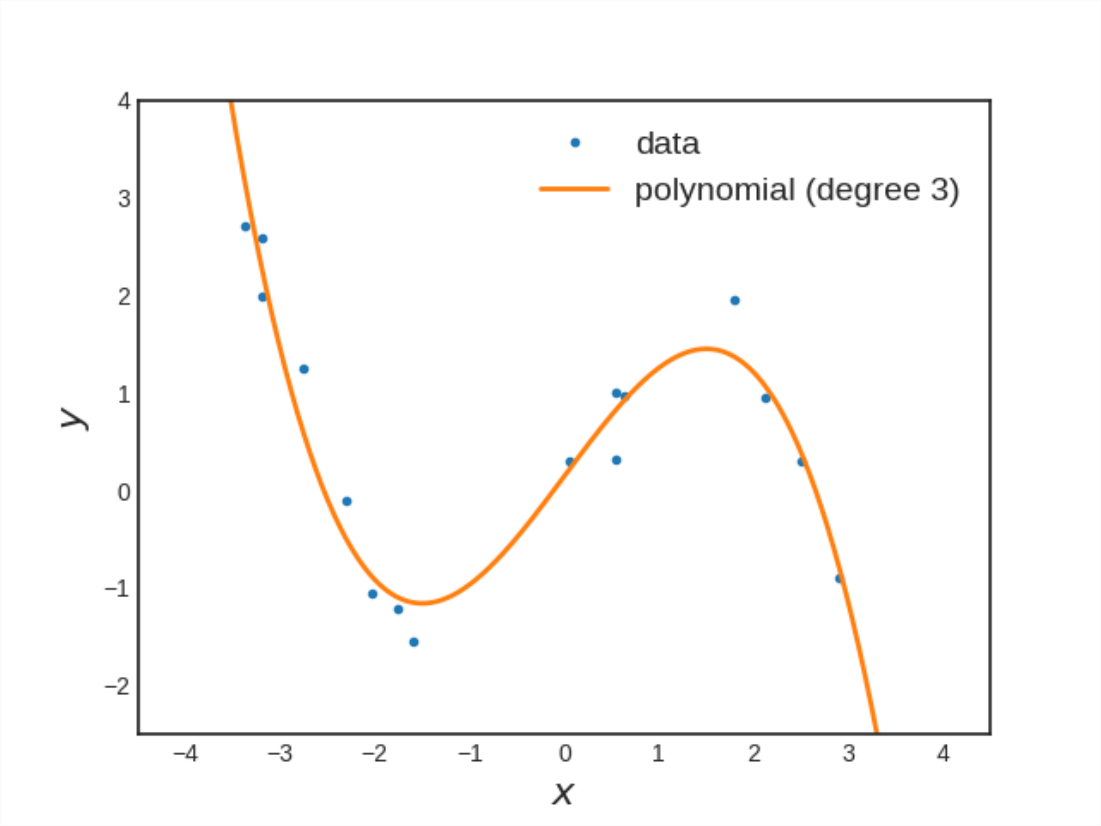
This fit looks better, but perhaps still not perfect:



Linear Regression Problem cont'd

Example

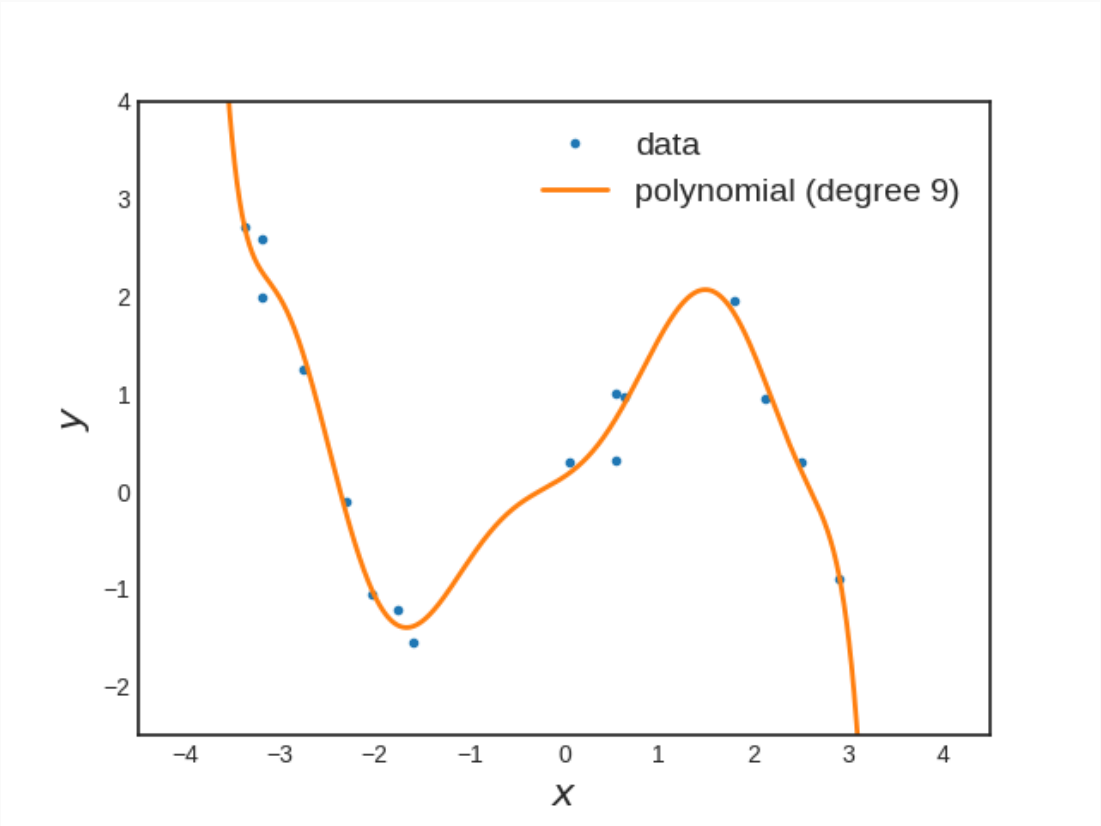
You continue with $K = 3$ (**cubic fit**). This looks pretty neat:



Encouraged by getting better and better results with larger and larger K, you decide to go further...

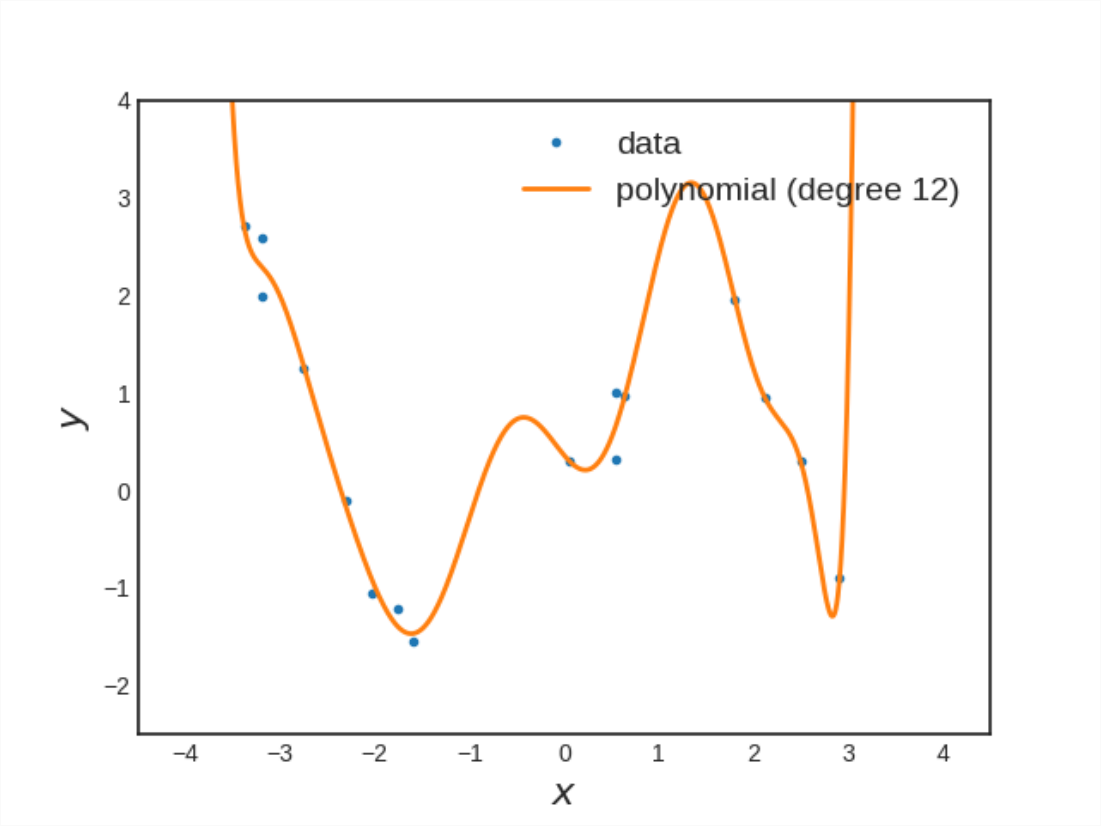
. . . So, why not go until $K = 9$?

The result seems slightly strange — while the data points are approximated better, the function looks a little jiggy:



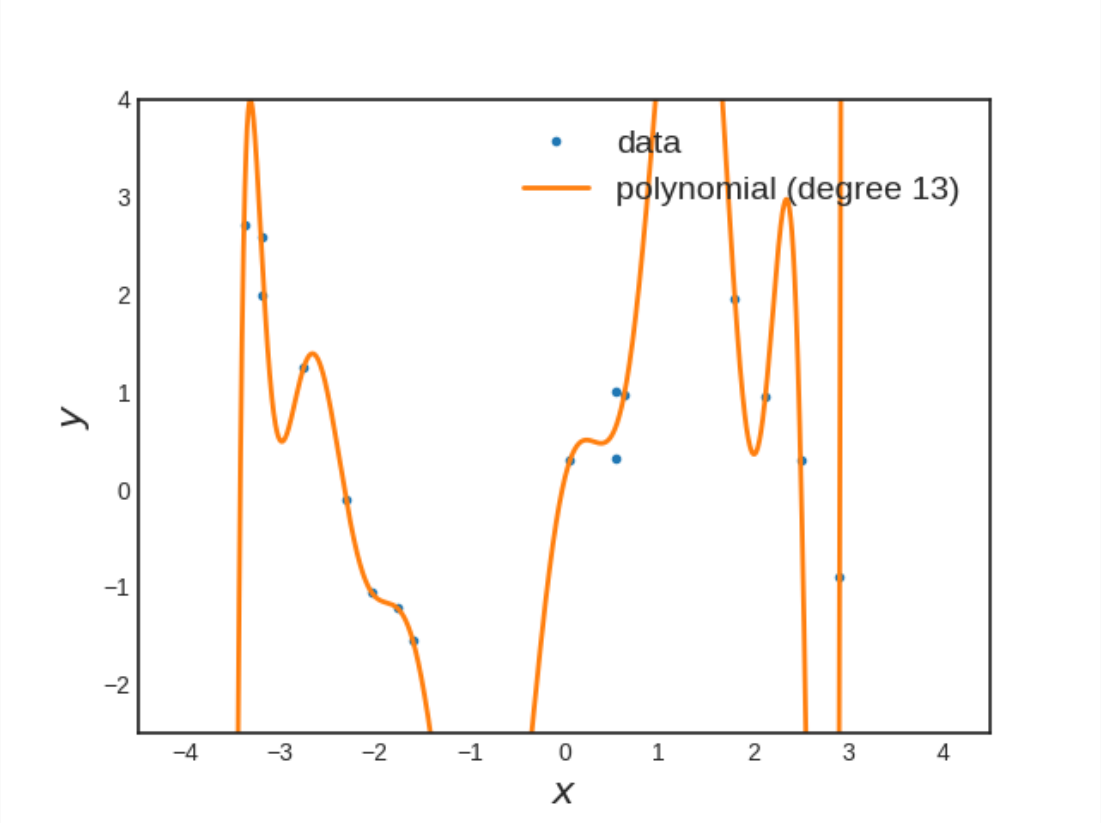
Maybe a higher degree K can fix this?

Using $K = 12$ delivers an even stranger fit. The data is approximated even better, but the function behaves strangely in between data points:

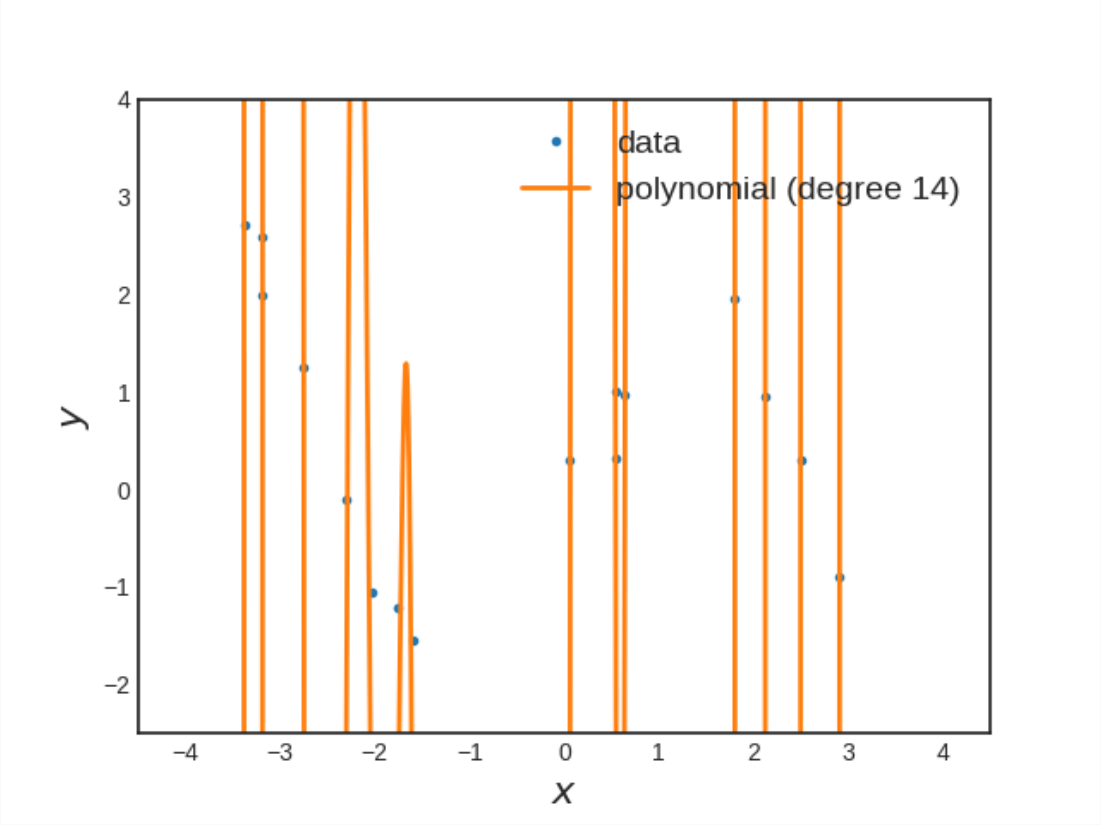


Surely the degree K is still too low!

Using $K = 13$ delivers a pretty strange function...



. . . and the result for $K = 14$ can only be described as erratic:



What did just happen?

What did just happen?

We fell prey to a phenomenon called **overfitting**.

Overfitting Explained

In machine learning, we always want to find some underlying "true" (assumed) **pattern** or **principle** underlying the data, e.g.

- the "true" regression function

$$y = f(x) + \epsilon$$

- the "true" classification concept

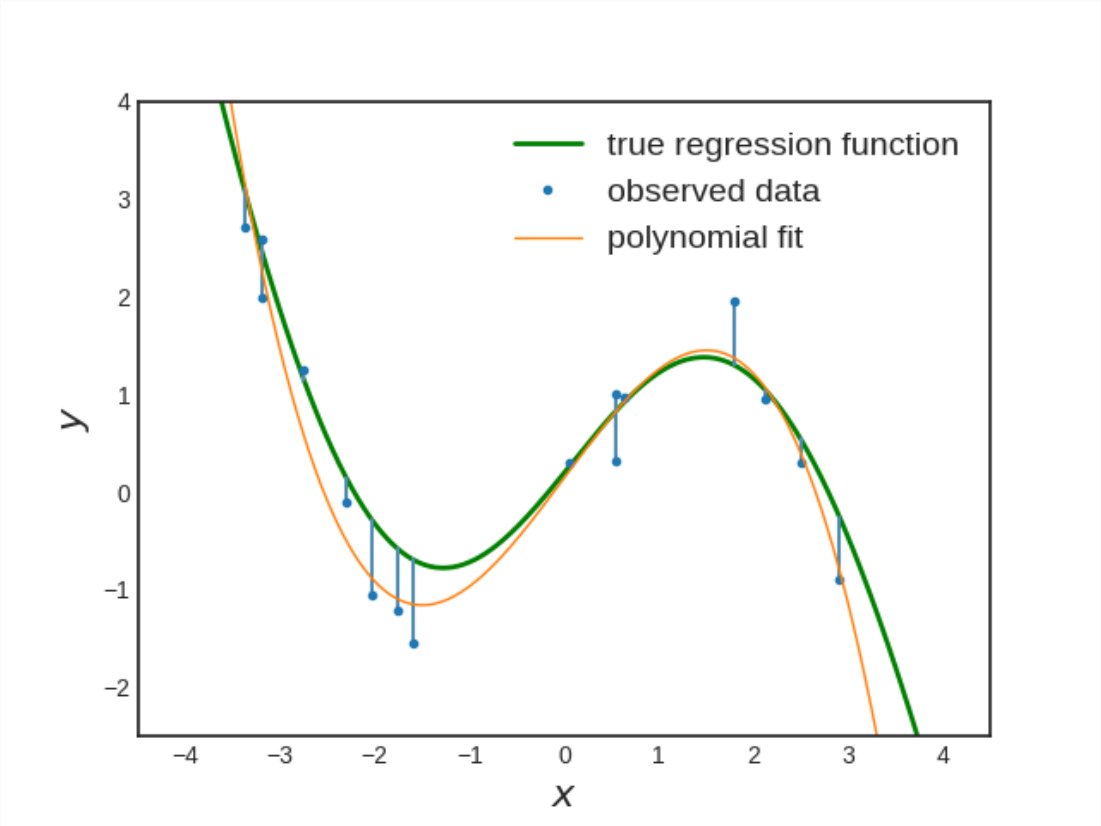
$$y = f(x)$$

- the "true" linear sub-space (e.g. in PCA)

$$\boldsymbol{Z} = \boldsymbol{X}B$$

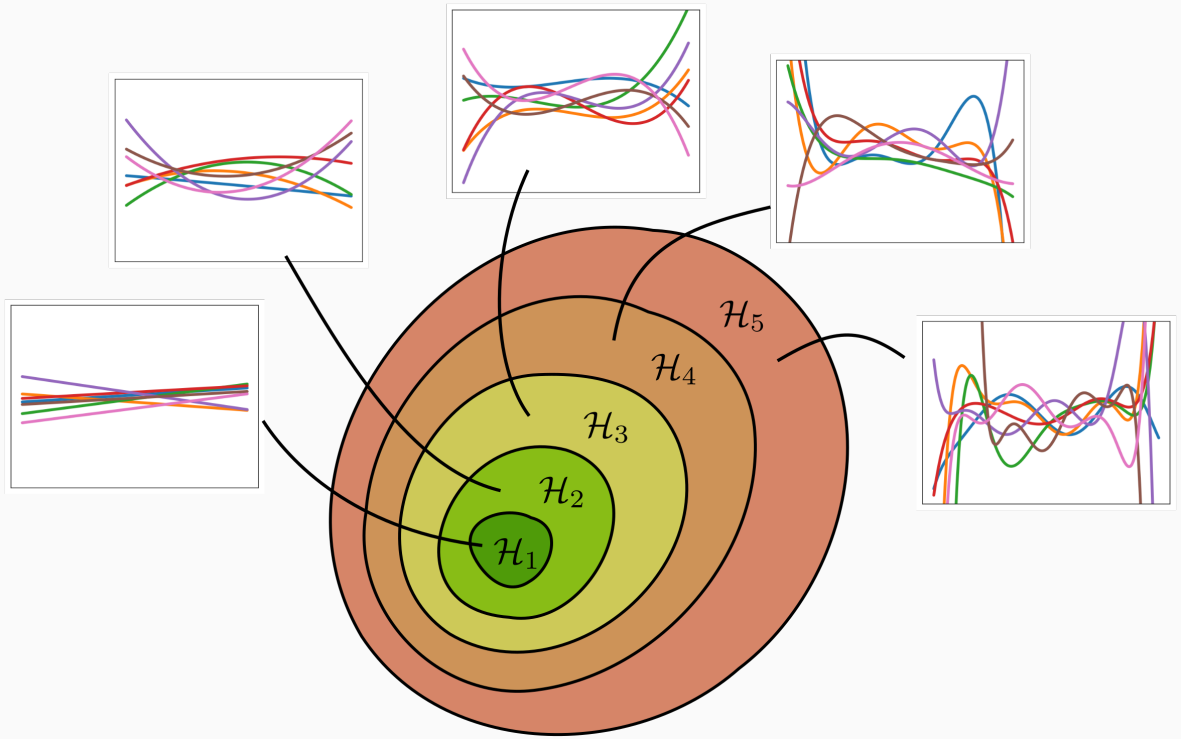
Overfitting Explained cont'd

However, we **never** observe the "true" pattern or principle, always just an imperfect and noisy version of it: **the training data** $\mathcal{D}$.
We use the training data to guide our search for a **model** (**hypothesis**), in the hope that it comes close to the ground truth.



Overfitting Explained cont'd

Recall that the model is picked from some model class $\mathcal{H}$, e.g., all polynomials of degree K. When using a **too large (powerful) model class** $\mathcal{H}$, we run risk to select a model which **fits the data** $\mathcal{D}$ **well, but not the true underlying concept.** We say "the model **overfits** to the data and does not **generalize** well."



Describing Overfitting

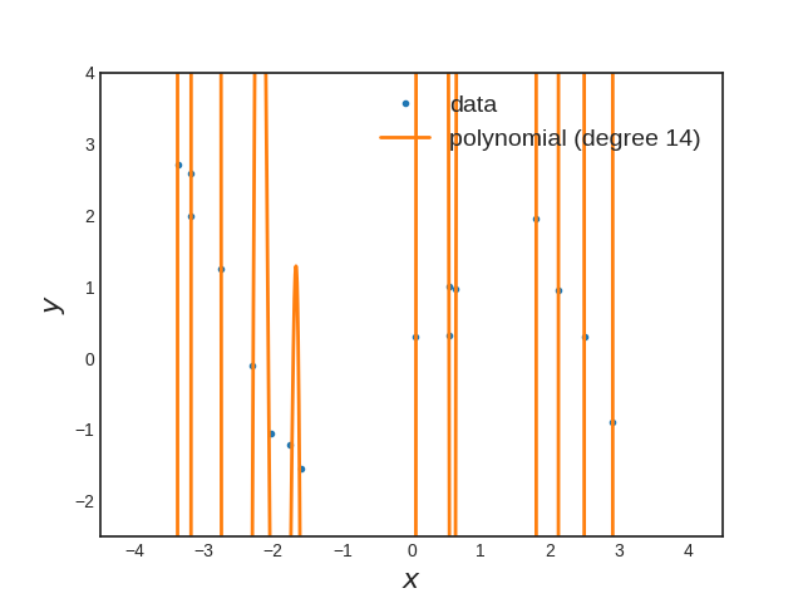
Overfitting can be understood from multiple angles:

- **Memorization, learning by heart:** The model is powerful enough to memorize the training set, but does not **generalize well** to the "true" (or actually, *intended*) concept. Compare this with a student who learns 7083 examples of 'addition' by heart, but "does not get" the concept of addition.
- **Too flexible model/Too large model class:** the hypothesis class contains too many models. Many are compatible with the available data (achieve low loss), yet they are very different.
- **Too little data:** A large model class is good, in principle, since it more likely contains the "true concept." But we simply have too little data (information) to reliably identify the "right" model (parameter).

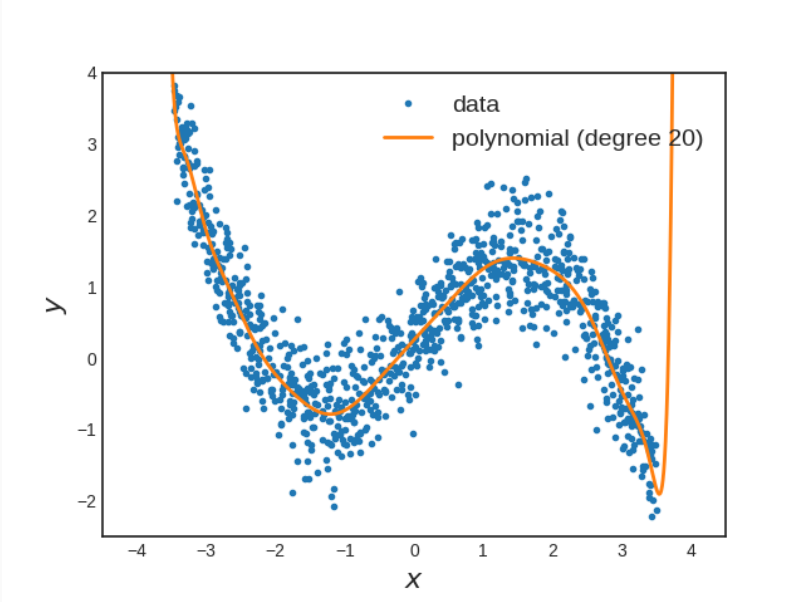
Getting more data...

Example

With $N = 1000$ data points even fitting a degree 20 polynomial works well (at least where data is provided):



$N = 16, K = 14$



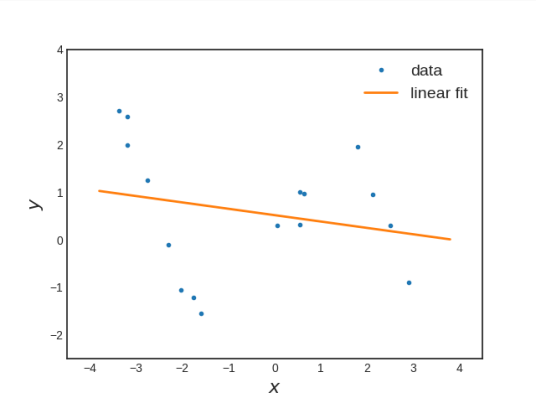
$N = 1000, K = 20$

Indeed, with sufficient data, overfitting does not occur. However, we **rarely** have sufficient data.

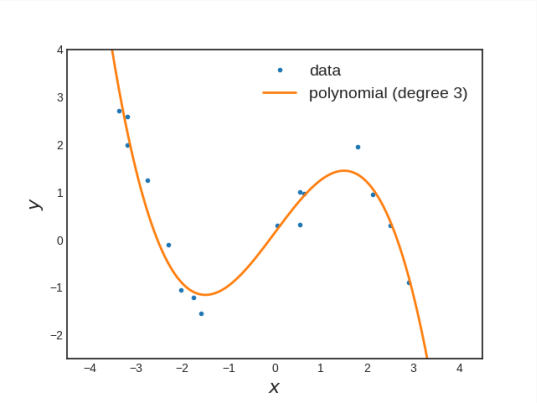
Underfitting

The converse effect of overfitting is **underfitting** — using a too weak model class, such that the model does fit neither the training data nor the true underlying concept well.

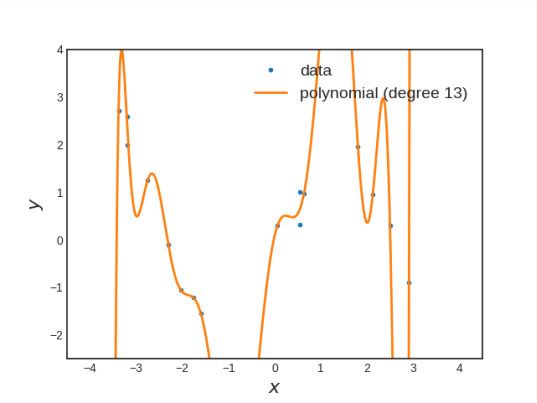
Underfitting



Just right



Overfitting



Underfitting is usually not much of a problem, since it is rather easy to increase the model power. Overfitting requires more sophisticated strategies.

In the following we will discuss

- **model evaluation**: how to specify and recognize overfitting (how good is our model really?)
- **model selection**: how to select the right model to prevent overfitting (what is the right model?)

Model Evaluation

Where does data (the samples) actually come from?

How do we describe the generation of data mathematically?

Data Generating Distribution

Generally, data in statistics and machine learning is assumed to come from some **unknown** **data-generating distribution** $p_{true}(\boldsymbol{x}, y)$. It encapsulates all uncertainty about how training inputs $\boldsymbol{x}$ and targets y are generated.

Very little might be known about about p_{true} and it might be a highly complicated distribution – but we assume it exists and it generates our data.

We often assume that the samples are produced **independently and identically distributed (i.i.d.)** according to p_{true}.

$$\boldsymbol{x}^{(i)}, y^{(i)} \sim p_{true} \qquad \text{independently for each } i$$

Let ℓ be some suitable **sample-wise loss function**, for example:

- squared loss (for regression, see Lecture 4)

$$\ell(\hat{y}, y) = (\hat{y} - y)^2$$

- cross-entropy (for classification, see Lecture 5):

$$\ell(\pi, y) = -\left[\mathbb{1}(y = 1) \cdot \log \pi + \mathbb{1}(y = -1) \cdot \log(1 - \pi)\right]$$

The **ideal goal** in machine learning is to minimize the **true loss**,

$$\mathcal{L}_{true}(\boldsymbol{\theta}) = \mathbb{E}_{\boldsymbol{x}, y \sim p_{true}} \left[\ell\left(f_{\boldsymbol{\theta}}(\boldsymbol{x}), y\right)\right]$$

which is the **expected loss** under p_{true}. $\mathcal{L}_{true}(\boldsymbol{\theta})$ is the expected loss our model will suffer in the "real world."

True Loss vs. Empirical Loss

In practice, however, we have only a **finite** dataset $\mathcal{D} = \left\{\left(\boldsymbol{x}^{(i)}, y^{(i)}\right)\right\}_{i=1}^{N}$ and optimize the **empirical loss**:

$$\mathcal{L}_{empirical}(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N} \ell\left(f_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}), y^{(i)}\right)$$

The empirical loss is an N-sample **Monte Carlo** estimator of the true loss. It converges to the true loss for an **infinite dataset**:

$$\lim_{N\to\infty} \mathcal{L}_{empirical}(\boldsymbol{\theta}) = \mathcal{L}_{true}(\boldsymbol{\theta}) = \mathbb{E}_{\boldsymbol{x},y\sim p^*}\left[\ell\left(\hat{y}, y\right)\right]$$

Thus, no overfitting for $N \to \infty$!

True Loss vs. Empirical Loss

- When $N < \infty$, $\mathcal{L}_{empirical}$ and $\mathcal{L}_{true}$ might differ substantially
- For **arbitrary** $\boldsymbol{\theta}$ (e.g. at initialization), $\mathcal{L}_{empirical}(\boldsymbol{\theta})$ is an **unbiased estimate** of $\mathcal{L}_{true}(\boldsymbol{\theta})$
- However, learning means that we aim to find

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \ \mathcal{L}_{empirical}(\boldsymbol{\theta})$$

- Thus, after optimization, $\mathcal{L}_{empirical}(\boldsymbol{\theta}^*)$ is an overly "optimistic" estimate of $\mathcal{L}_{true}(\boldsymbol{\theta}^*)$.
- How can we detect overfitting?
- How can we measure (estimate) $\mathcal{L}_{true}$?

We need two datasets:

- a **training set** $\mathcal{D}_{train}$ which we use to **learn the model**
- a **test set** $\mathcal{D}_{test}$ which we use to **evaluate/test the model**

We need two datasets:

- a **training set** $\mathcal{D}_{train}$ which we use to **learn the model**
- a **test set** $\mathcal{D}_{test}$ which we use to **evaluate/test the model**

However, we only have **only one** dataset. . .

N samples

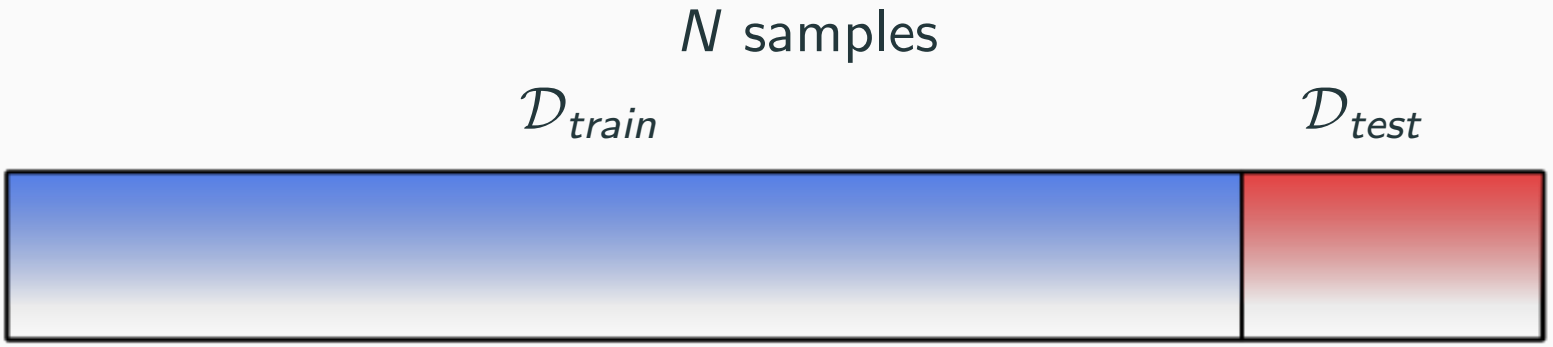
We need two datasets:

- a **training set** $\mathcal{D}_{train}$ which we use to **learn the model**
- a **test set** $\mathcal{D}_{test}$ which we use to **evaluate/test the model**

However, we only have **only one** dataset...

Thus, we **randomly split** it into a training set and a test set:

1. randomly shuffle the N samples
2. split it into N_{train} and N_{test} samples (commonly $10\% - 50\%$)



Training Loss and Test Loss

Definition

Let $\mathcal{D}_{train} = \left\{ \left(\boldsymbol{x}_{train}^{(i)}, y_{train}^{(i)} \right) \right\}_{i=1}^{N_{train}}$ and $\mathcal{D}_{test} = \left\{ \left(\boldsymbol{x}_{test}^{(i)}, y_{test}^{(i)} \right) \right\}_{i=1}^{N_{test}}$ be a training and a test set.

- Then the **training loss** for model parameter $\boldsymbol{\theta}$ is defined as

$$\mathcal{L}_{train}(\boldsymbol{\theta}) = \frac{1}{N_{train}} \sum_{i=1}^{N_{train}} \ell \left(f_{\boldsymbol{\theta}} \left(\boldsymbol{x}_{train}^{(i)} \right), y_{train}^{(i)} \right)$$

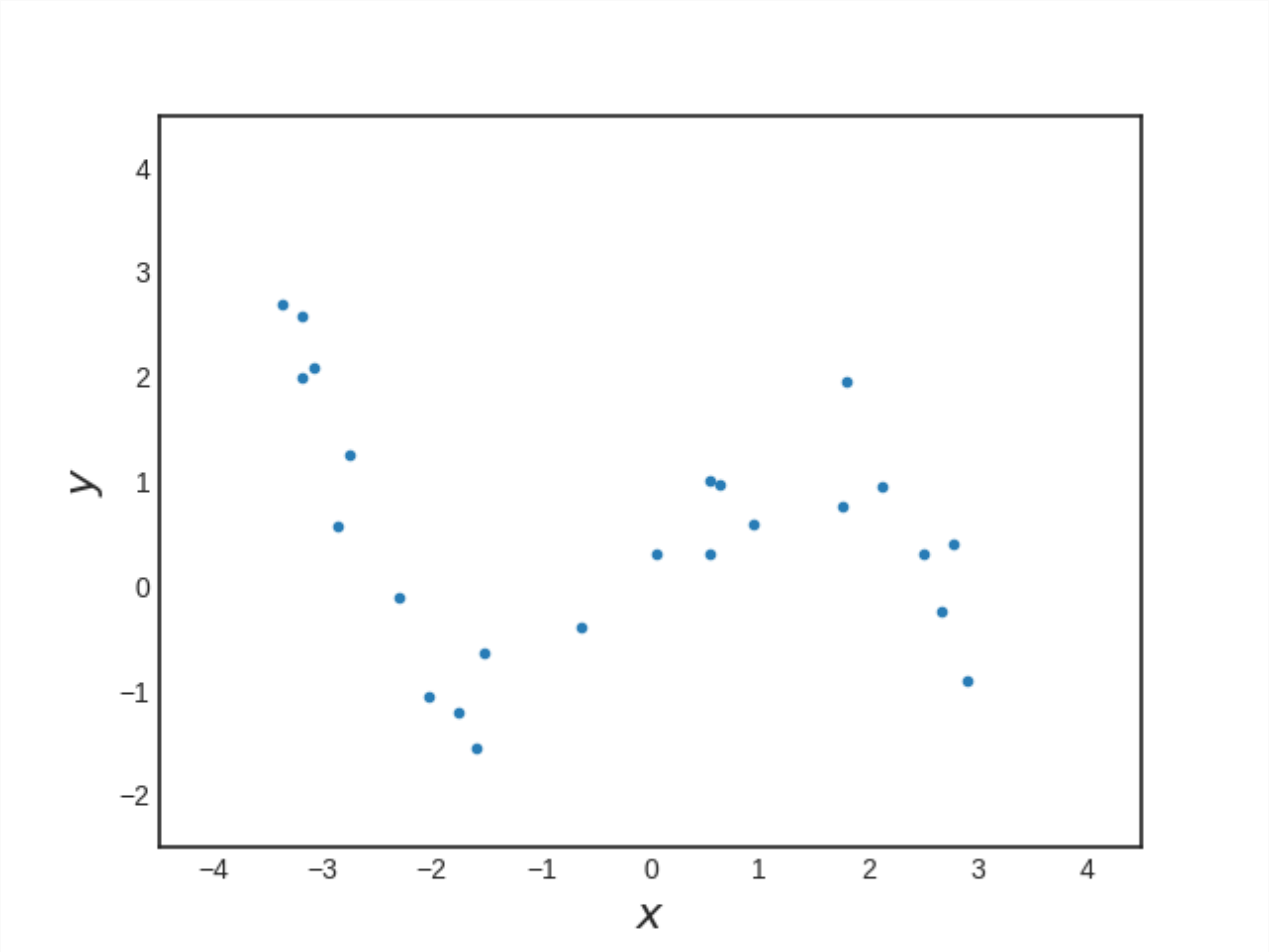
  During training, we aim to minimize $\mathcal{L}_{train}$, which will be an overly optimistic estimate of the true loss.

- Similarly, the **test loss** is defined as

$$\mathcal{L}_{test}(\boldsymbol{\theta}) = \frac{1}{N_{test}} \sum_{i=1}^{N_{test}} \ell \left(f_{\boldsymbol{\theta}} \left(\boldsymbol{x}_{test}^{(i)} \right), y_{test}^{(i)} \right)$$

  $\mathcal{L}_{test}$ serves as a noisy (but unbiased) estimate of $\mathcal{L}_{true}$. Thus, $\mathcal{L}_{test}$ measures the actual performance of our model. **The test set is "taboo" during training!**

Consider again a regression problem, with 24 data samples.

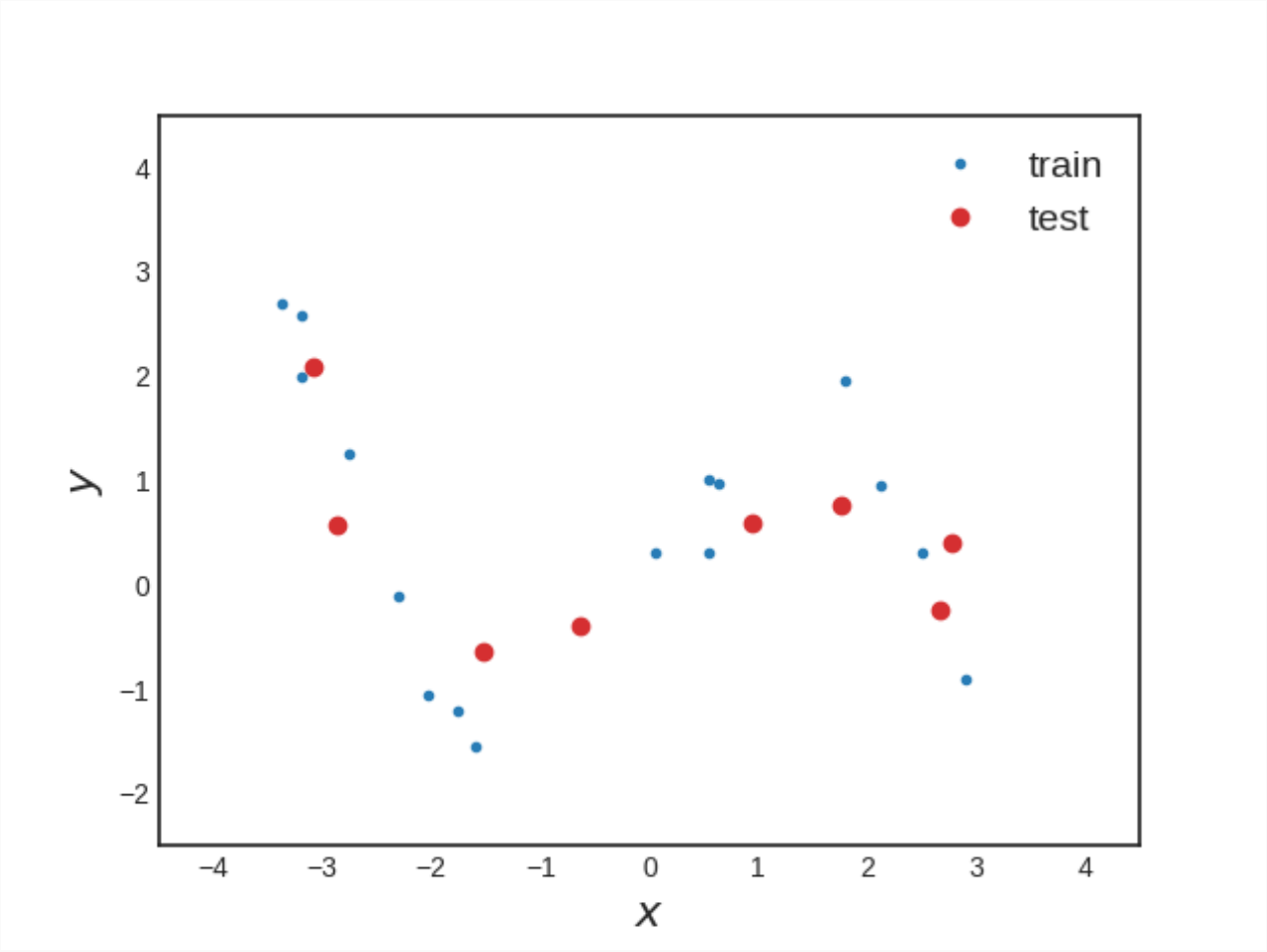


Test Loss for Regression

Example

Consider again a regression problem, with 24 data samples.

Assume a split into 16 training samples and 8 test samples:

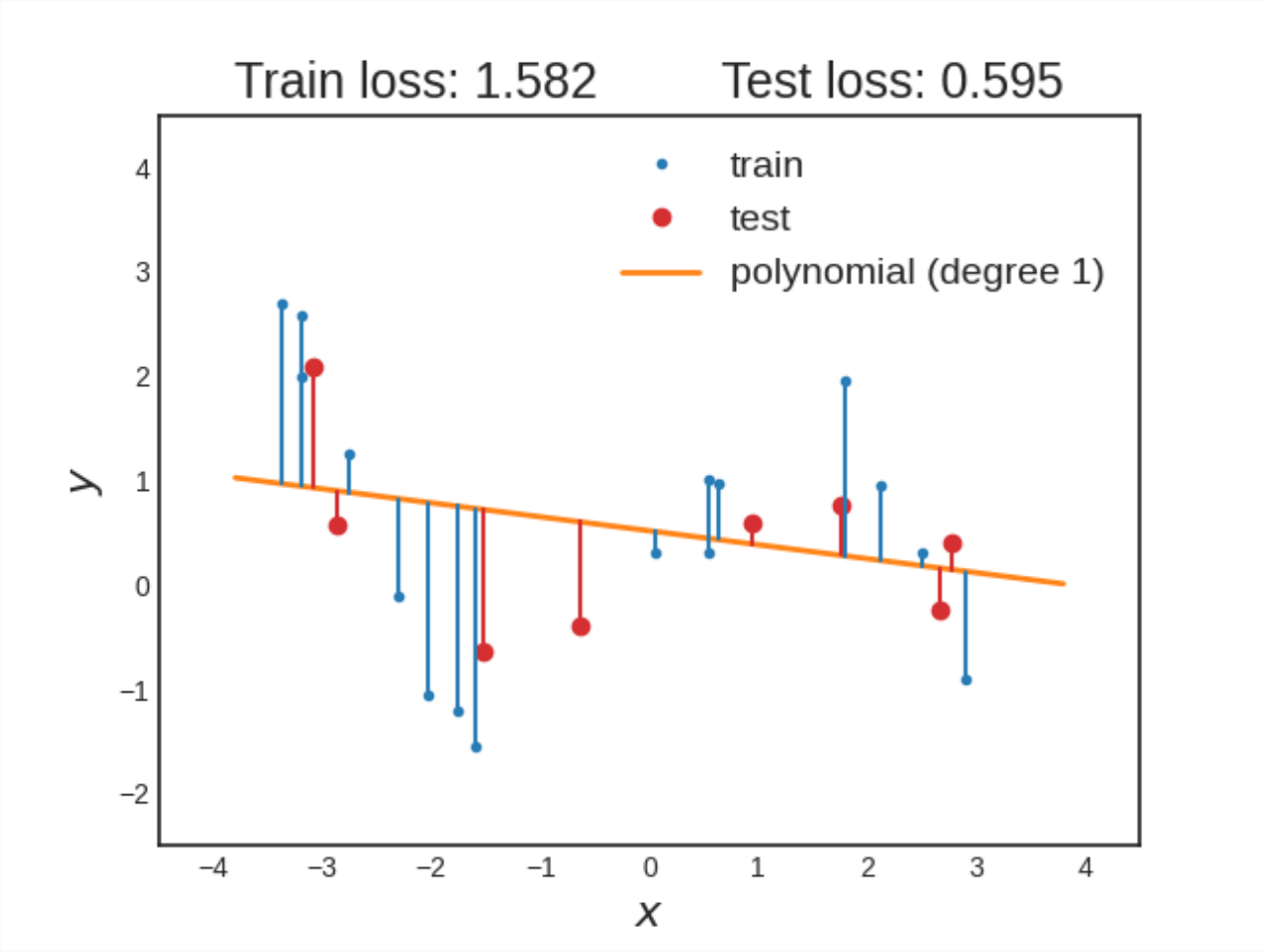


Test Loss for Regression cont'd

Example

Next, we fit polynomials of varying degree K to the training set.

We compute both the least-squares training loss and the test loss:

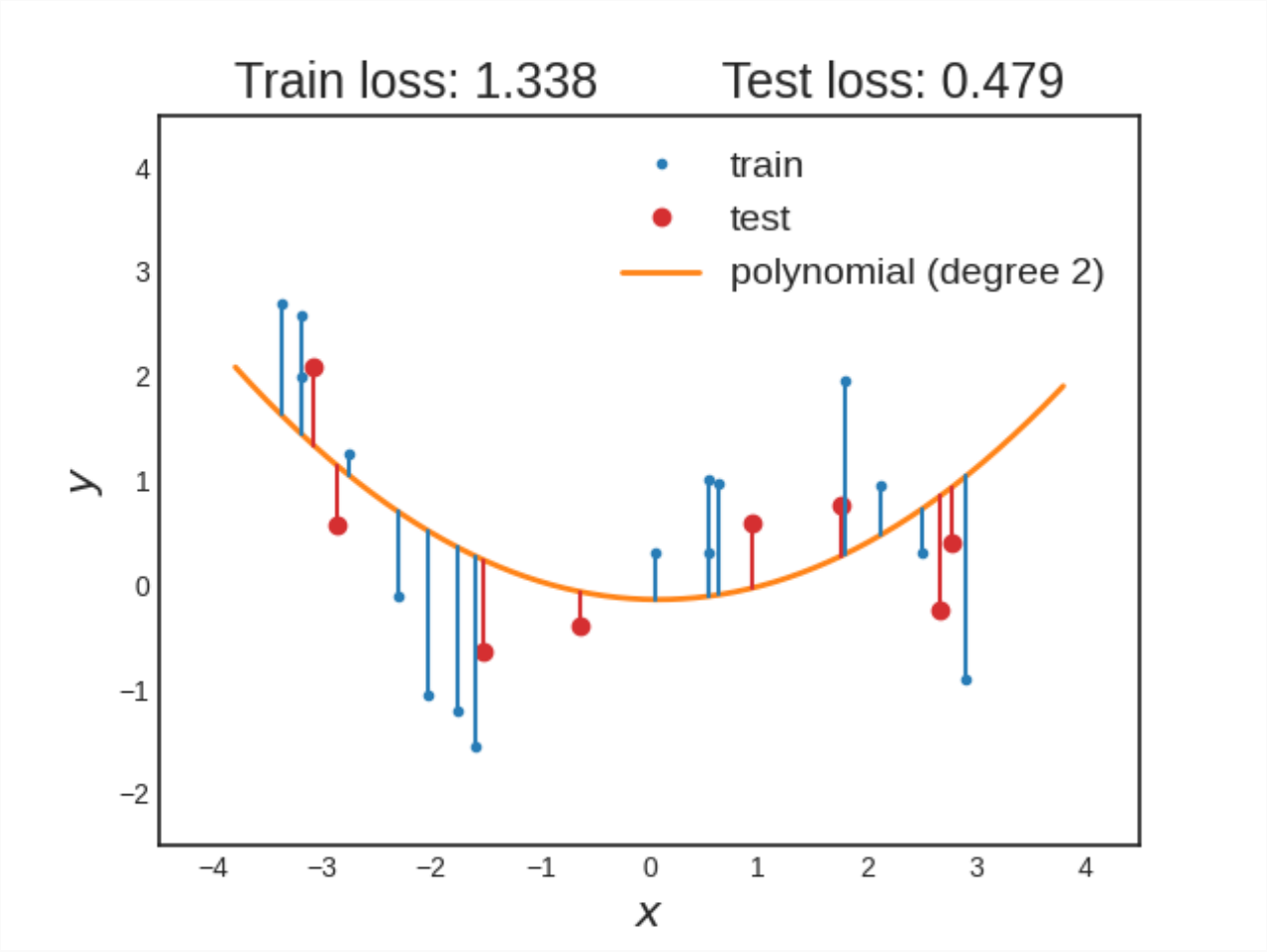


Test Loss for Regression cont'd

Example

Next, we fit polynomials of varying degree K to the training set.

We compute both the least-squares training loss and the test loss:

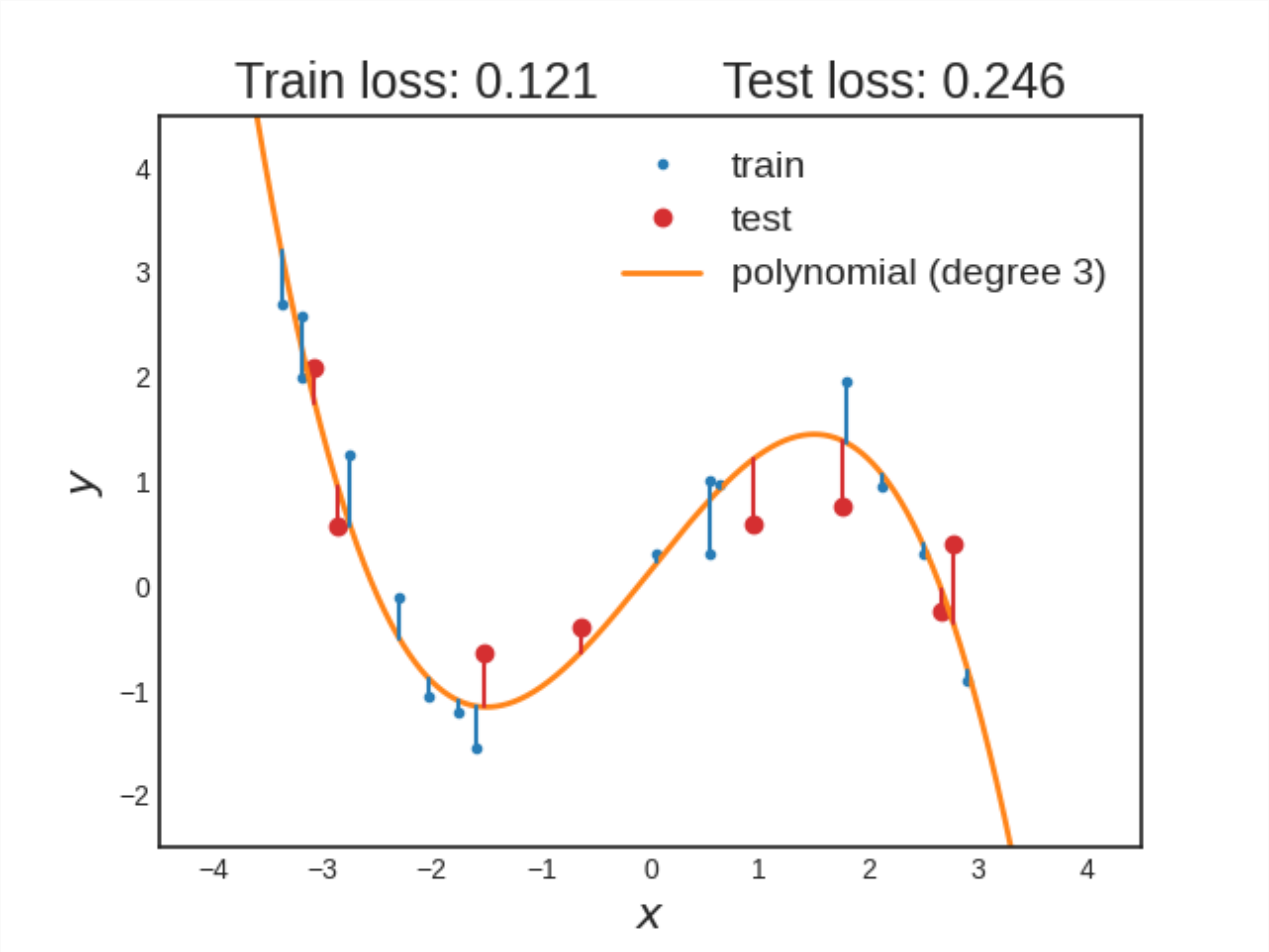


Test Loss for Regression cont'd

Example

Next, we fit polynomials of varying degree K to the training set.

We compute both the least-squares training loss and the test loss:

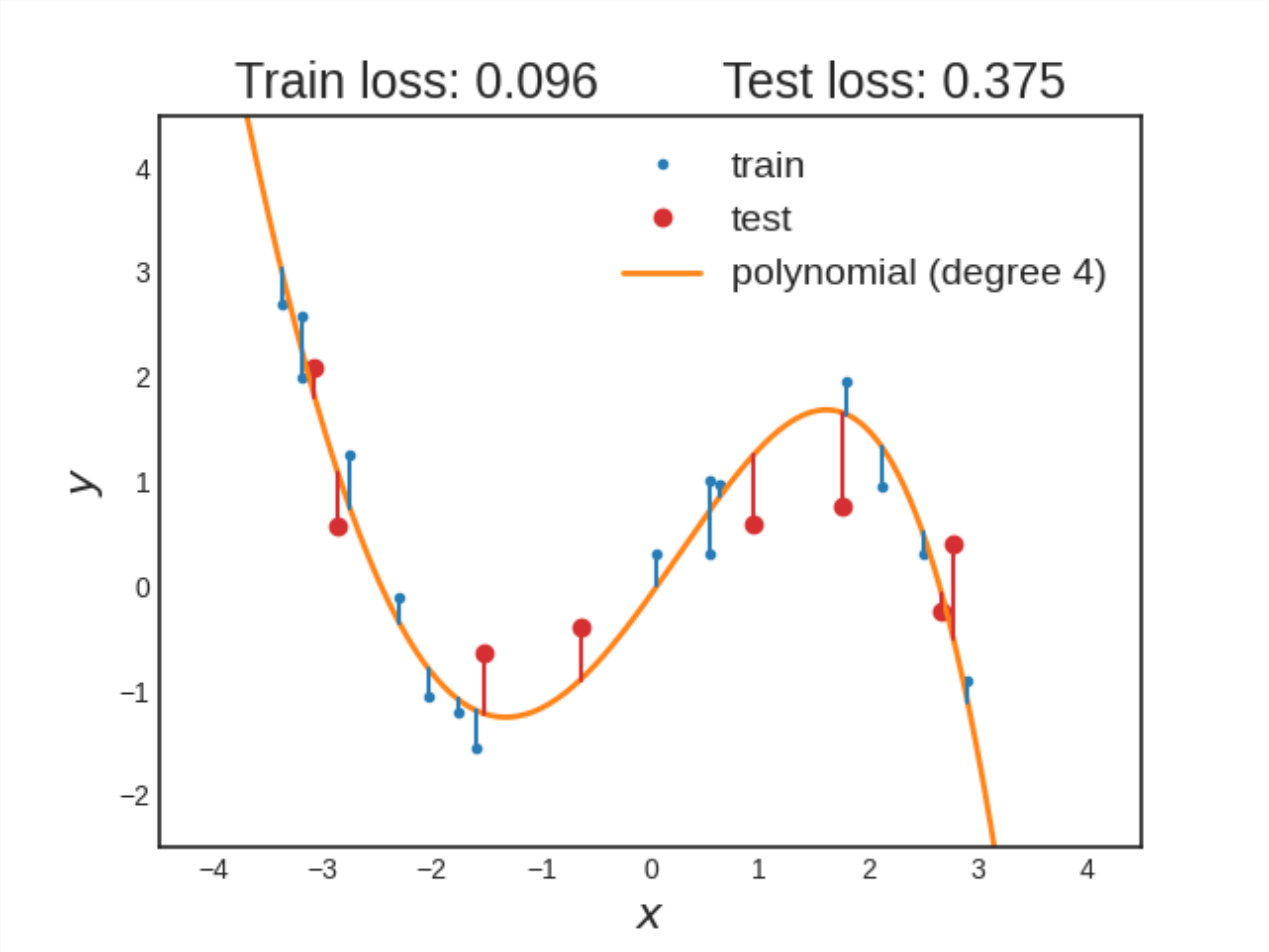


Test Loss for Regression cont'd

Example

Next, we fit polynomials of varying degree K to the training set.

We compute both the least-squares training loss and the test loss:

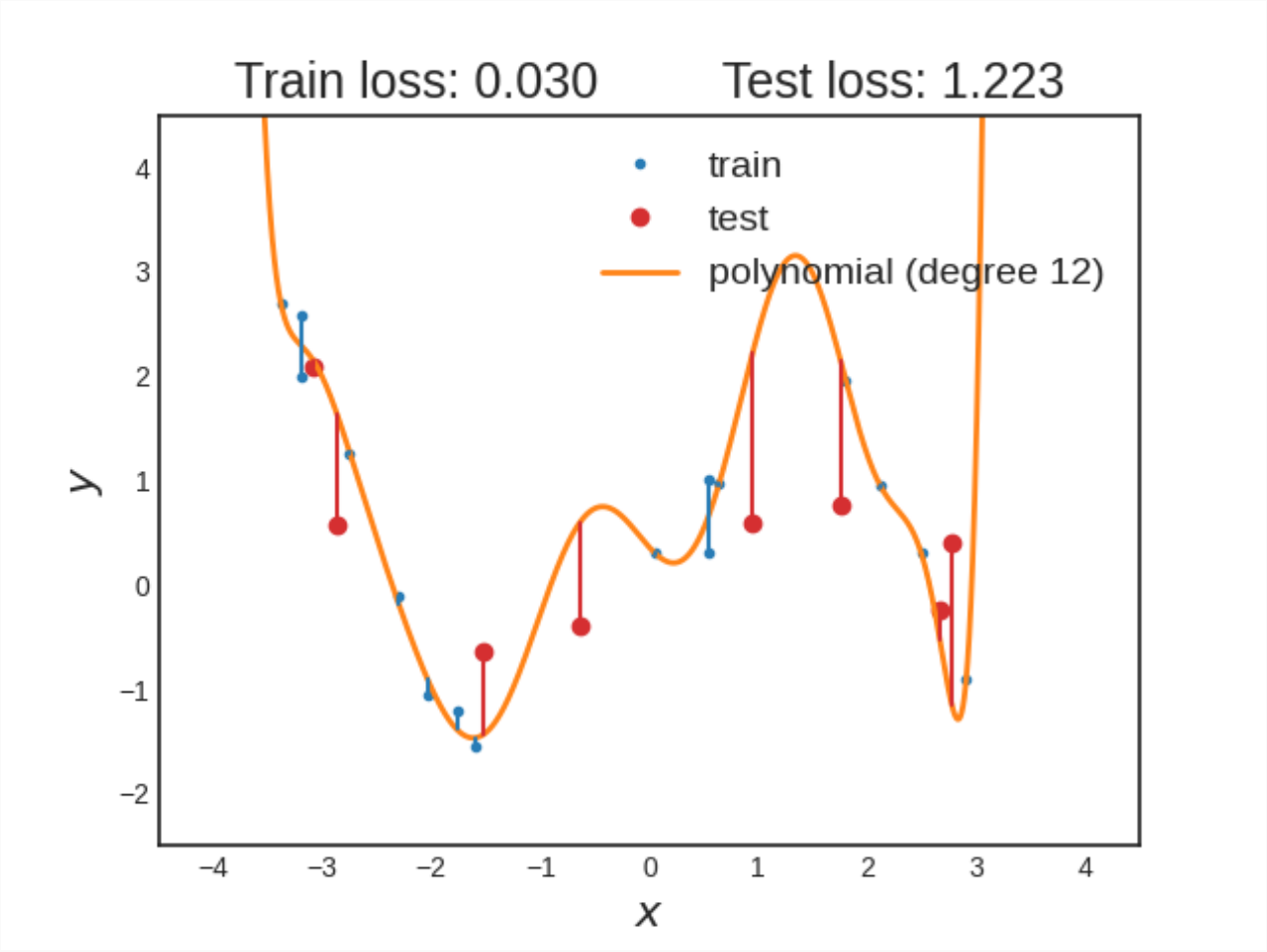


Test Loss for Regression cont'd

Example

Next, we fit polynomials of varying degree K to the training set.

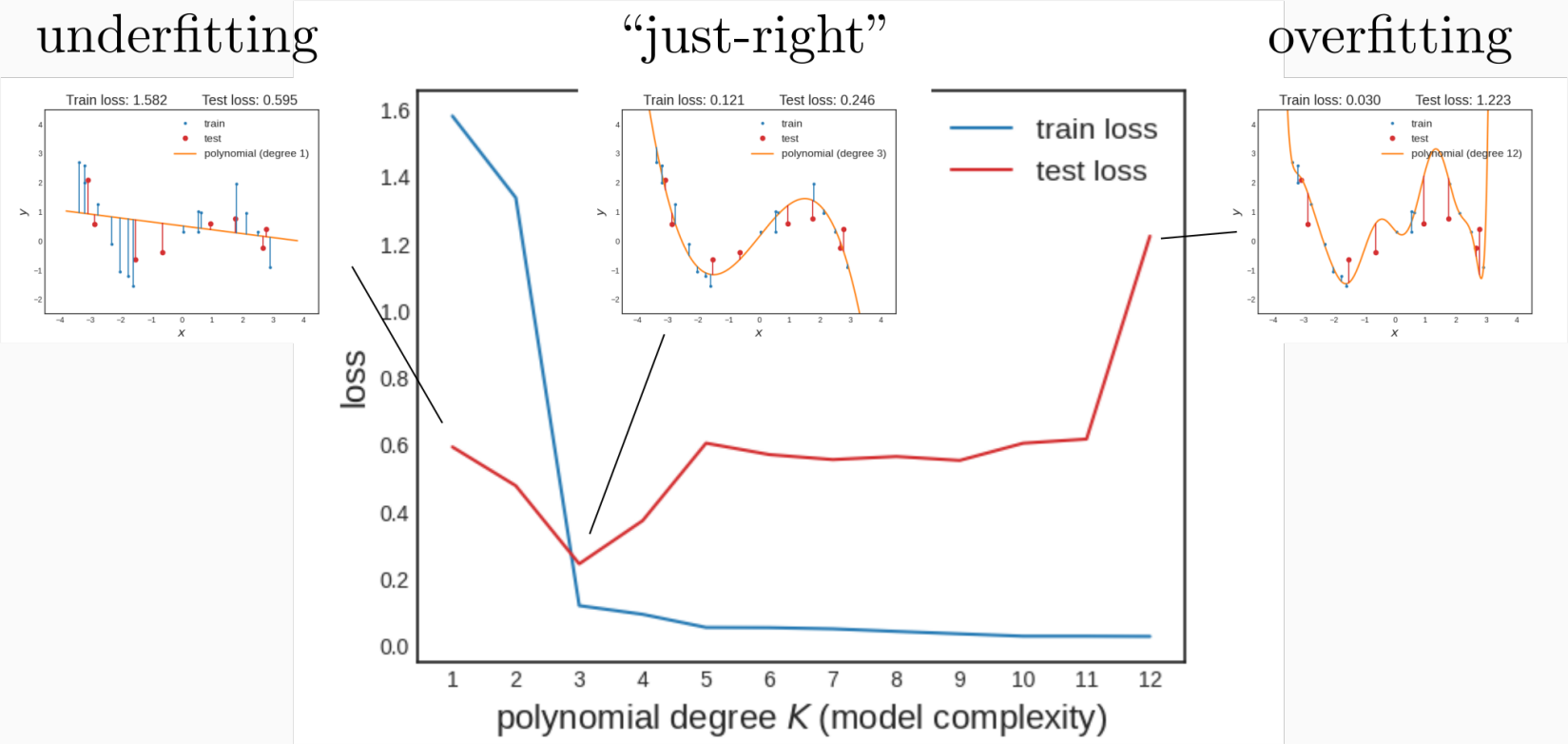
We compute both the least-squares training loss and the test loss:



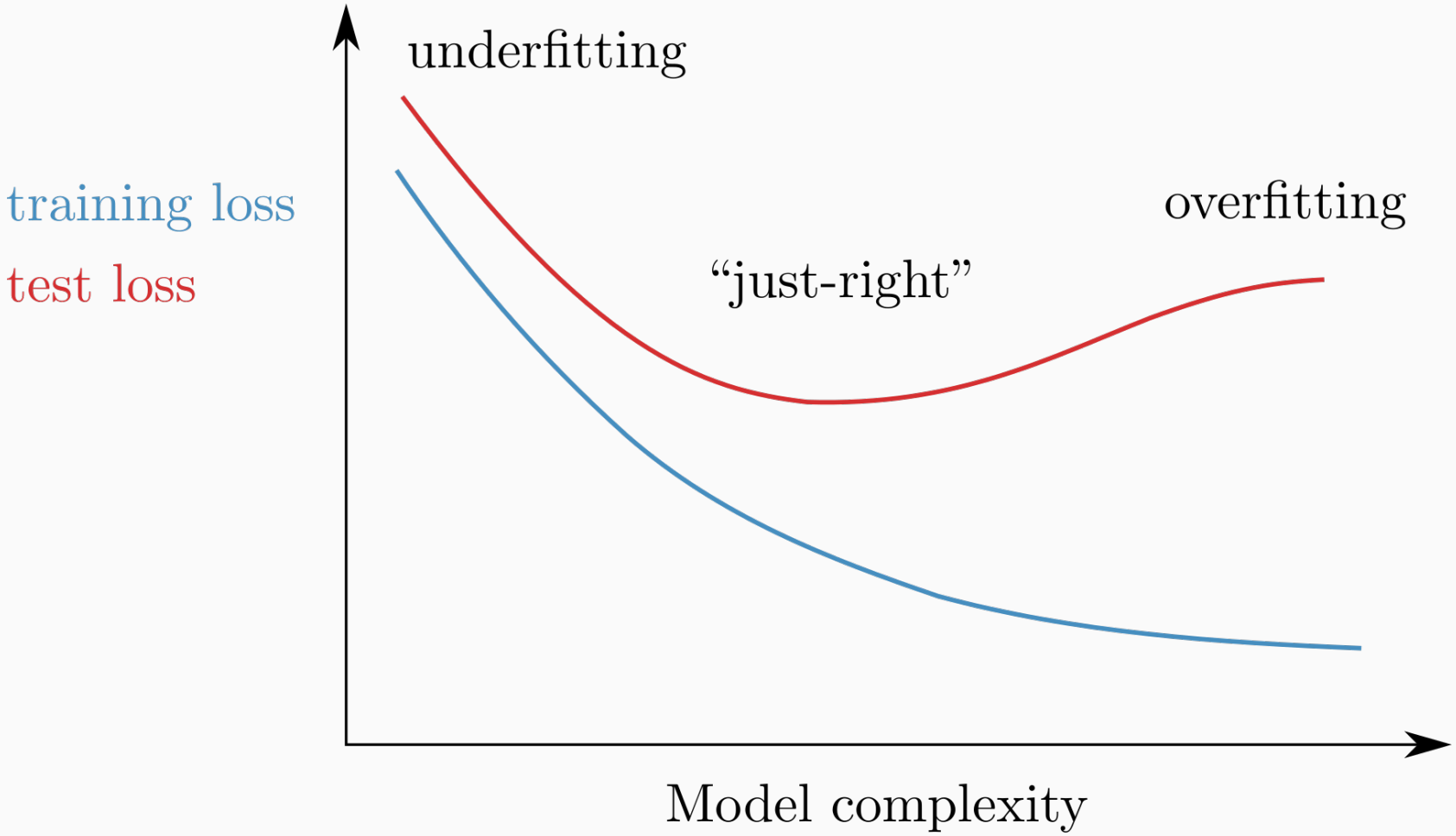
Test Loss for Regression cont'd

Example

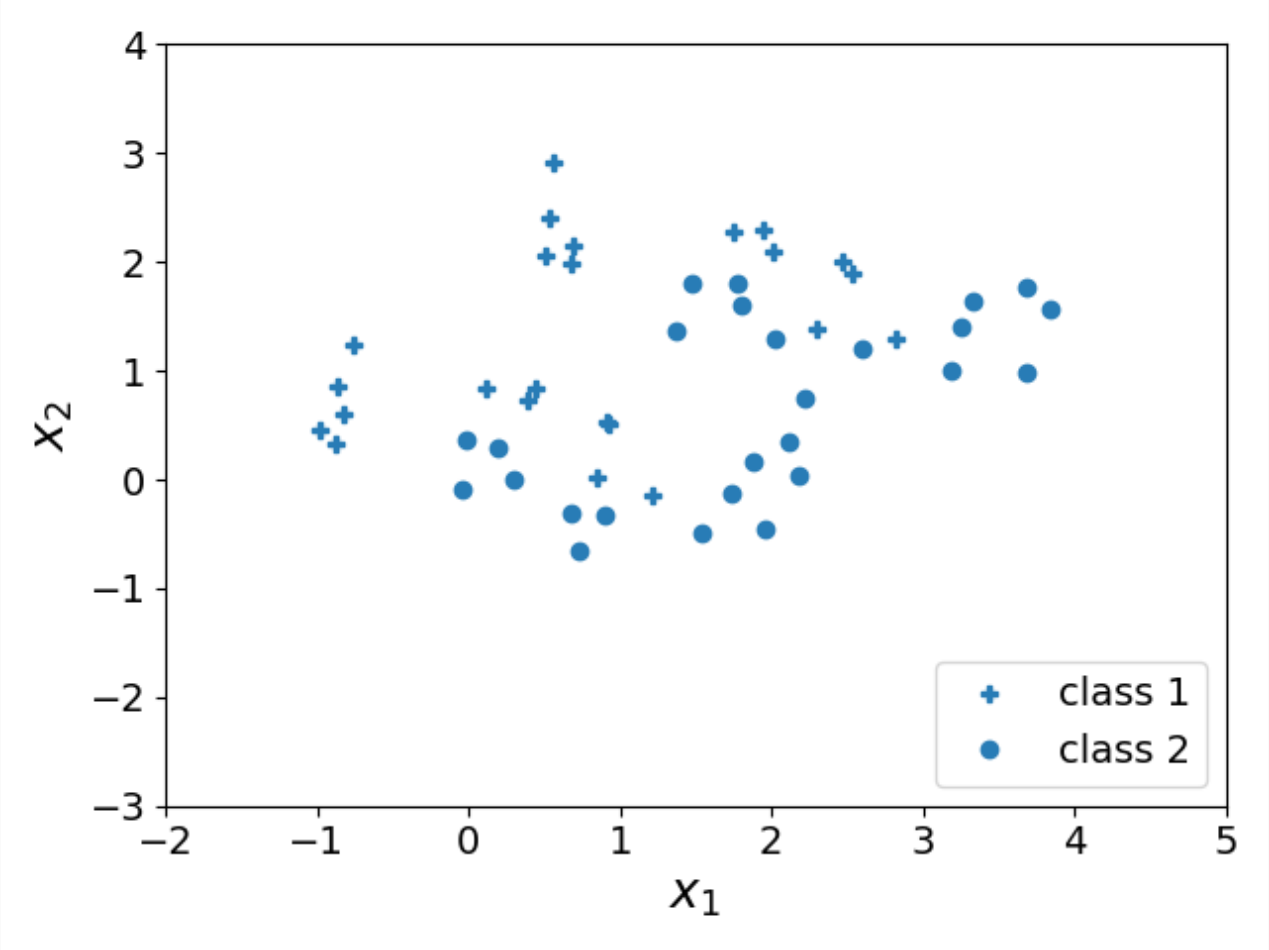
We can plot now the training loss and the test loss over the polynomial degree K (**model complexity**):



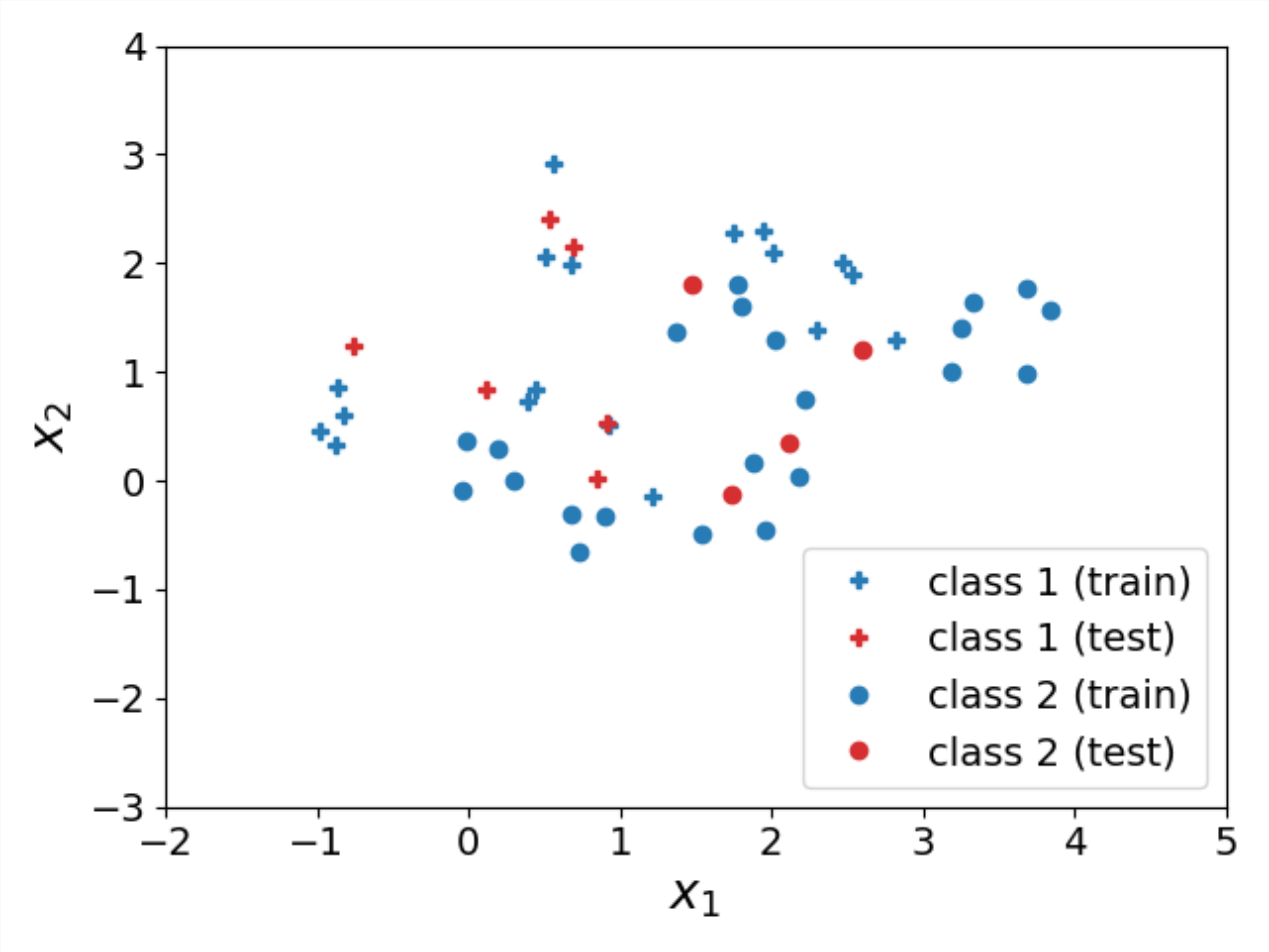
The Stereotypical Picture of Overfitting



A binary classification problem with $D = 2$ and $N = 150$:



We randomly split the data into training (80%) and test set (20%):



We train a logistic regression classifier (optimizing cross-entropy), using all polynomials up to degree K as features:

$$
\begin{aligned}
K = 0: &\quad \phi_0(\boldsymbol{x}) = 1 \\
K = 1: &\quad \phi_1(\boldsymbol{x}) = x_1, \quad \phi_2(\boldsymbol{x}) = x_2 \\
K = 2: &\quad \phi_3(\boldsymbol{x}) = x_1 x_2, \quad \phi_4(\boldsymbol{x}) = x_1^2, \quad \phi_5(\boldsymbol{x}) = x_2^2 \\
K = 3: &\quad \phi_6(\boldsymbol{x}) = x_1^2 x_2, \quad \phi_7(\boldsymbol{x}) = x_1 x_2^2, \quad \phi_8(\boldsymbol{x}) = x_1^3, \quad \phi_9(\boldsymbol{x}) = x_2^3 \\
\ldots &\quad \ldots
\end{aligned}
$$

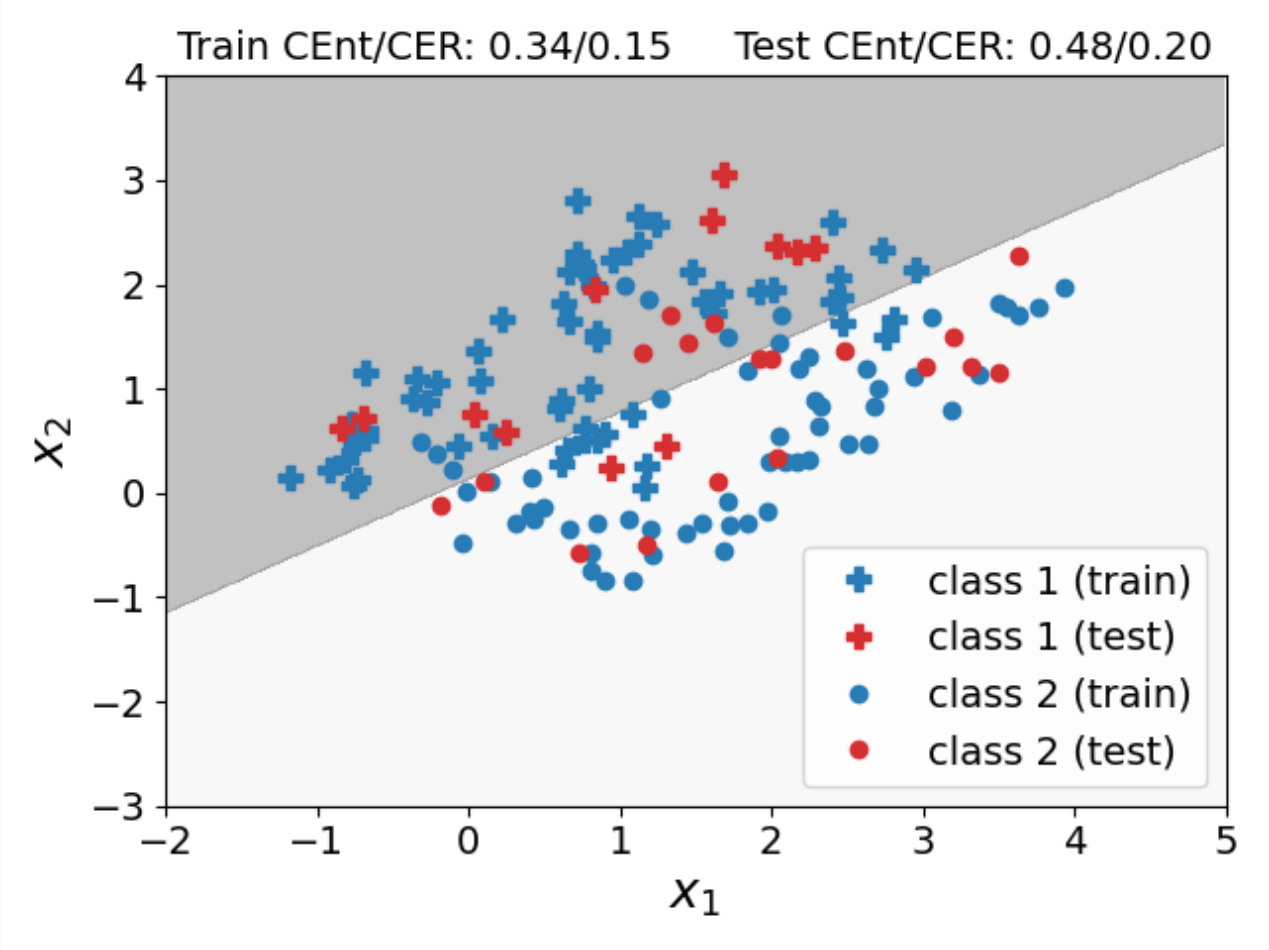
We optimize the cross-entropy loss (CEnt) on the training set using gradient descent. We also record the cross entropy on the test set.

Additionally, we record the **classification error rate** on both the training and test set:

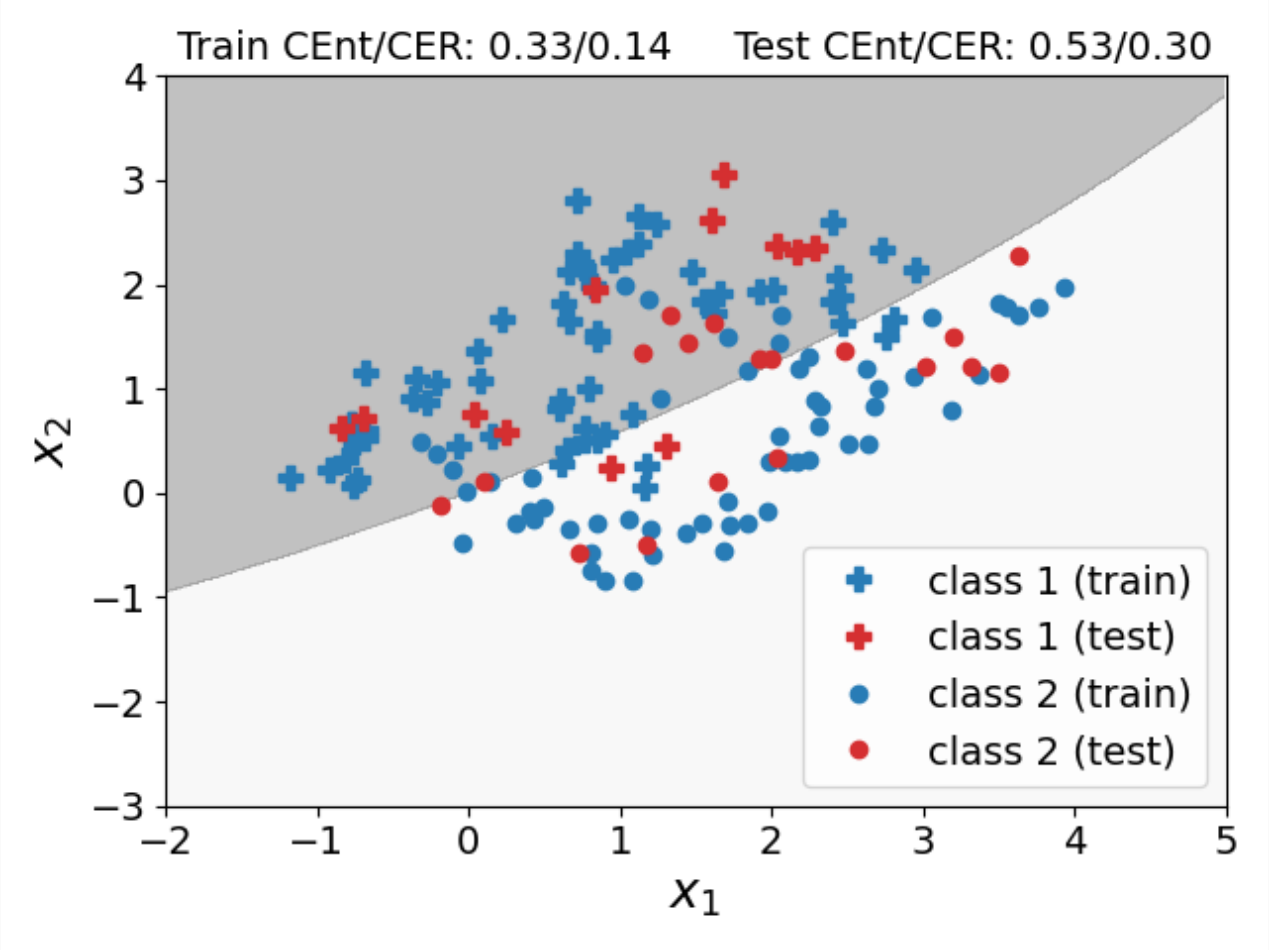
$$CER_{train} = \frac{1}{N_{train}} \sum_{i=1}^{N_{train}} \mathbb{1}\left[\hat{y}_{train}^{(i)} \neq y_{train}^{(i)}\right]$$

$$CER_{test} = \frac{1}{N_{test}} \sum_{i=1}^{N_{test}} \mathbb{1}\left[\hat{y}_{test}^{(i)} \neq y_{test}^{(i)}\right]$$

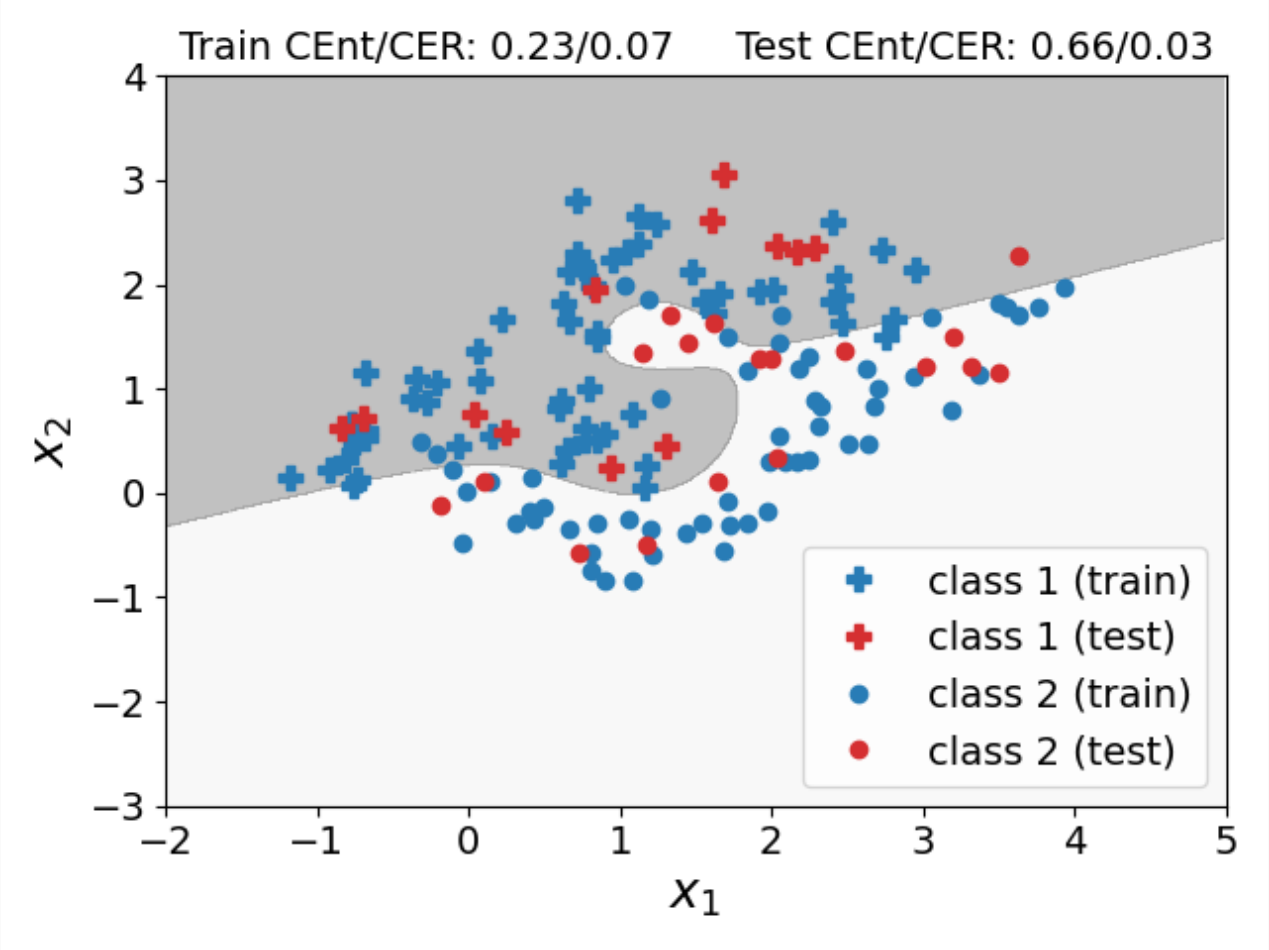
Polynomial degree $K = 1$:



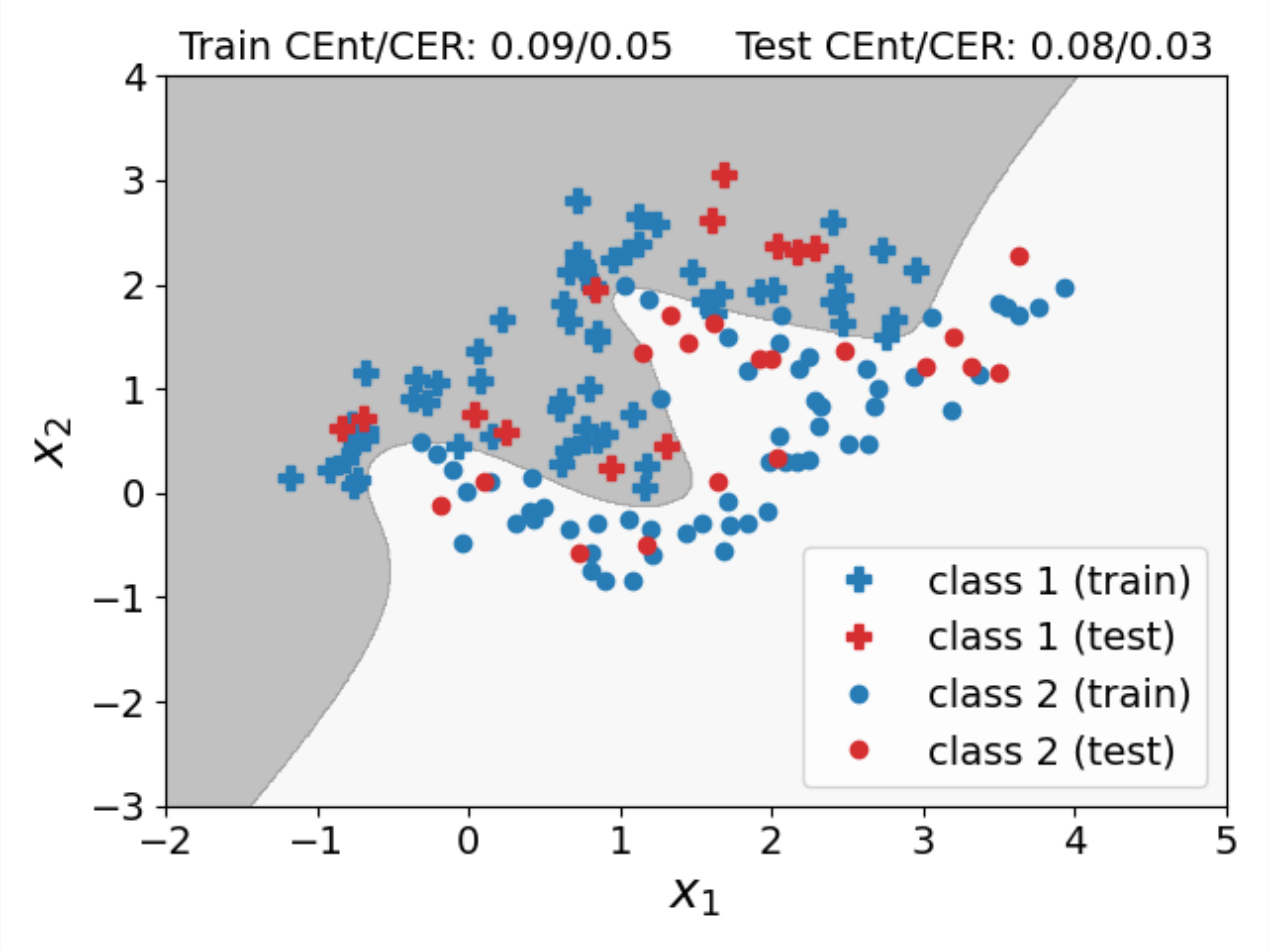
Polynomial degree $K = 2$:



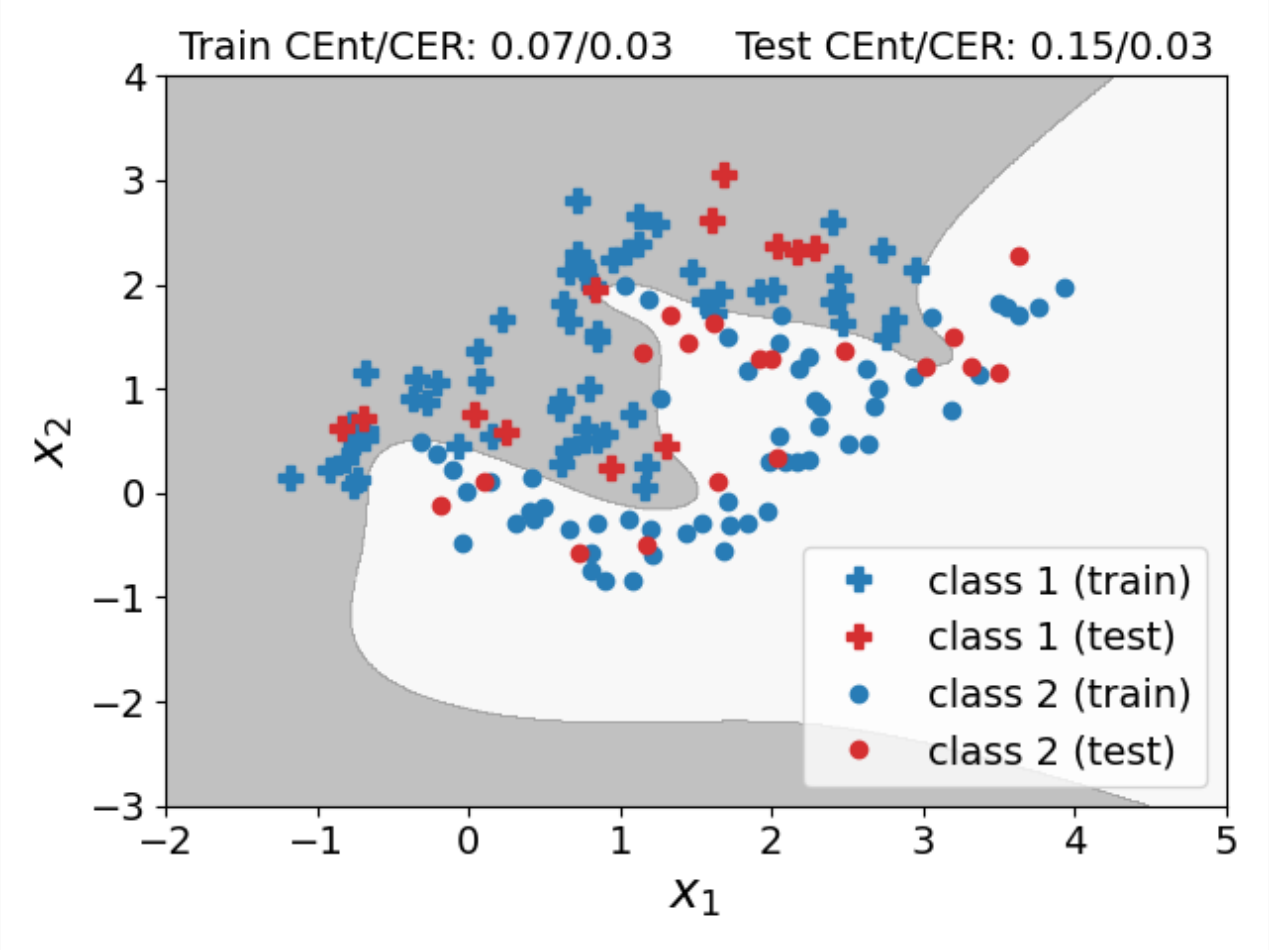
Polynomial degree $K = 3$:



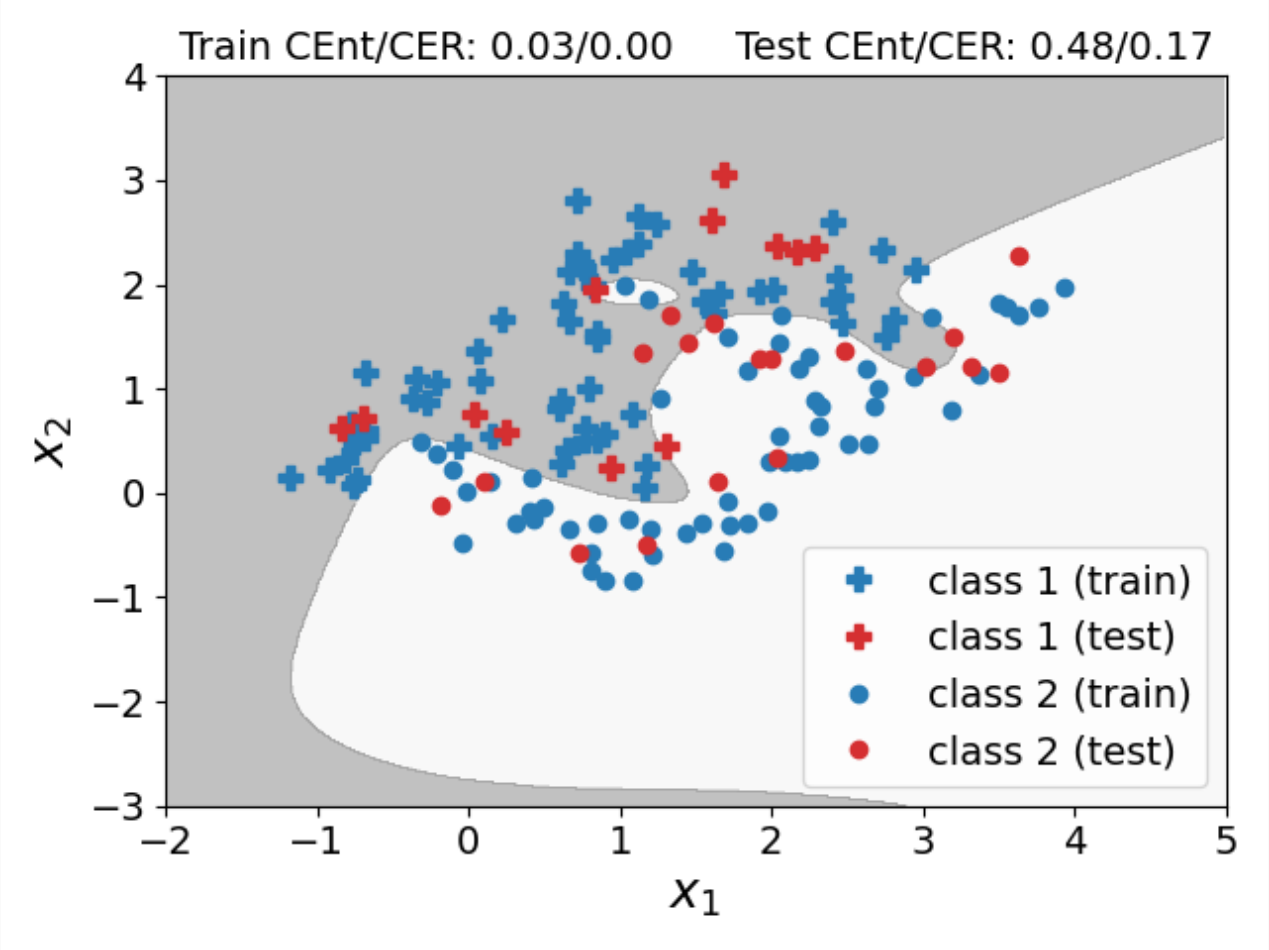
Polynomial degree $K = 5$:



Polynomial degree $K = 6$:



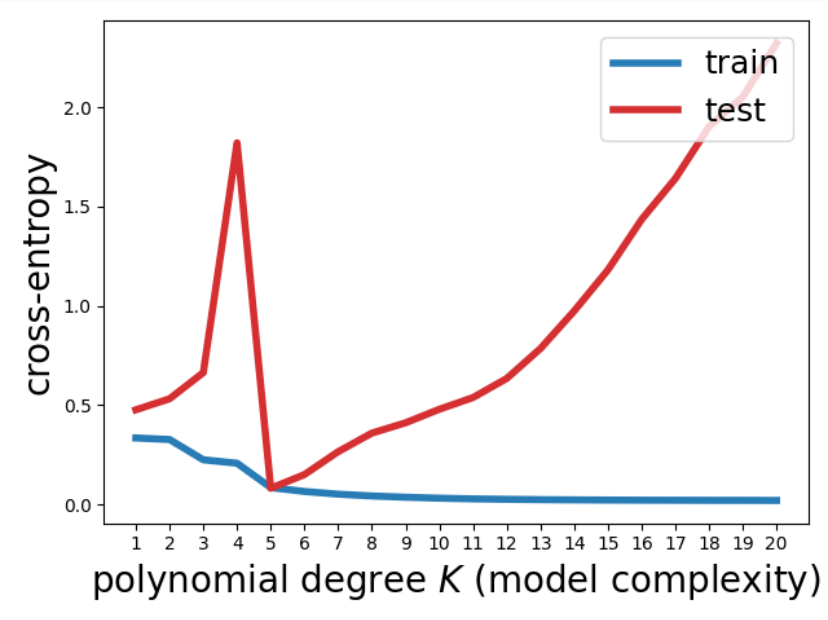
Polynomial degree $K = 10$:

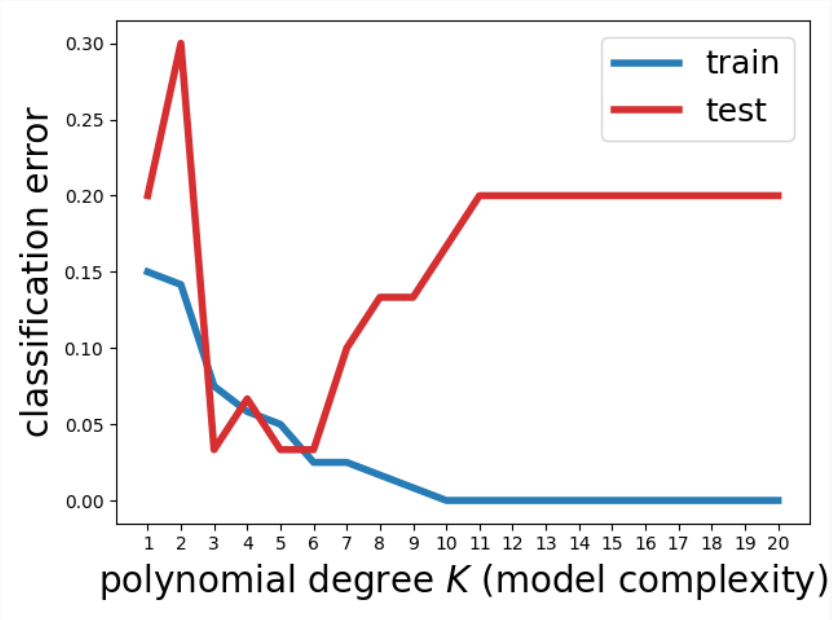


Test Loss for Classification cont'd

Example

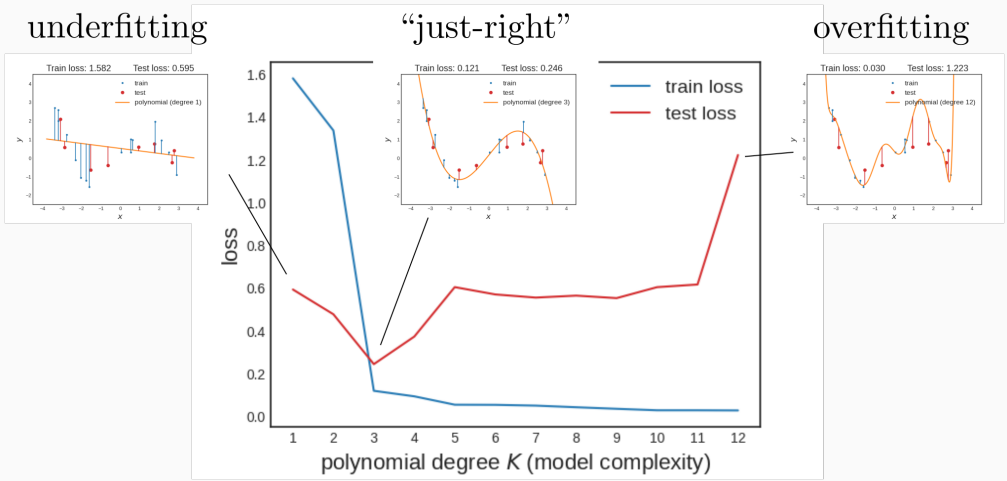
Again the typical picture of overfitting:





Model Selection

In your job as data analyst, you have now successfully evaluated your regression model, and you find that polynomial degree $K = 3$ has the smallest test error.



You deploy the model to your customer and report that it has an average squared prediction error of about 0.25.

Your customer is really happy with the model, but reports that they observed a **slightly larger test error of** 0.27 in their applications.

A "Big" Regression Problem

Example

Your next project is much more challenging:

- You have high-dimensional inputs and $N = 25,000$ samples
- By inspecting the data you note that the regression function must be "pretty complicated"
- So, you use a flexible model with 12 **different hyper-parameters**
- You select $10,000$ promising configurations of hyper-parameters, i.e. $10,000$ model candidates

A "Big" Regression Problem

Example

- You split the data into 90% training data and 10% test data
- You train each of your $10,000$ hyper-parameter configurations – this takes two weeks on your company cluster
- You evaluate the test error for each configuration and the best model achieves an mean-squared test error of 0.53
- You deploy the model and report 0.53 test error

One week later, your customer calls you. They are pretty upset.

They tell you that they have measured a mean-squared error of 1.62 in their application, which is significantly larger than your reported 0.53!

What happened?

What happened?

Again, we fell prey to overfitting.

What happened?

Again, we fell prey to overfitting.

This time, we **overfitted on the test set.**

Selecting Models, Setting Hyper-Parameters

- Most ML models have several hyper-parameters:
  - polynomial degree in polynomial regression
  - number of layers and units in neural networks (to be discussed)
  - trade-off parameter in support vector machines (to be discussed)
  - thresholds in decision trees (to be discussed)
- Let ψ denote a hyper-parameter configuration for our learning algorithm of choice
- Each hyper-parameter configuration ψ effectively corresponds to a different model class $\mathcal{H}(\psi)$
- Let $\mathbf{\Psi} = \{\psi_1, \psi_2, \ldots, \psi_J\}$ be all considered configurations
- Let $\mathcal{L}_{test}(\psi_j)$ be the achieved test error for ψ_j

Selecting Models, Setting Hyper-Parameters cont'd

When selecting a model according to minimal test error, we actually optimizing

$$\min_{\psi \in \boldsymbol{\psi}} \mathcal{L}_{test}(\psi)$$

Thus, we are actually **learning** ψ on the test set!

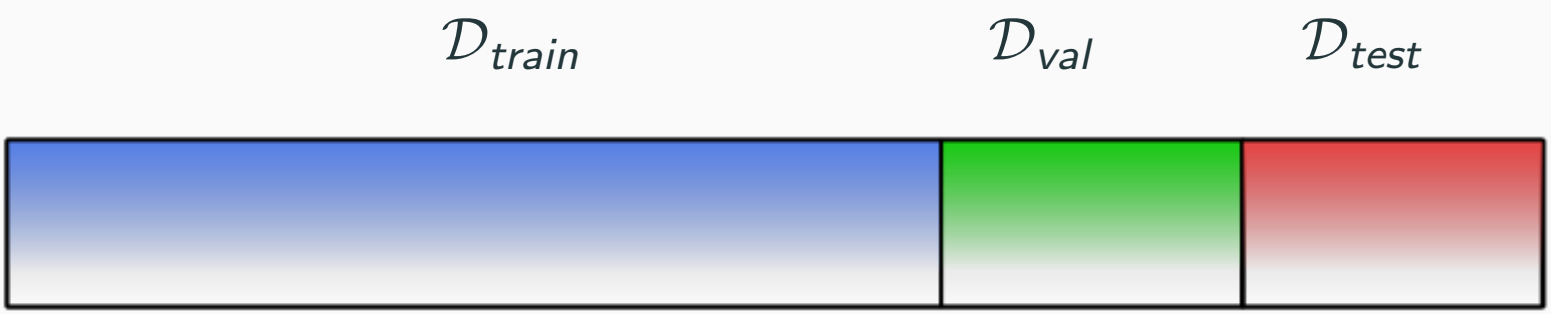
Model selection $\triangleq$ "second order learning"

However, recall that the **test set is taboo!** Using the test set for model selection will (again) yield an overly optimistic estimate of the true loss.

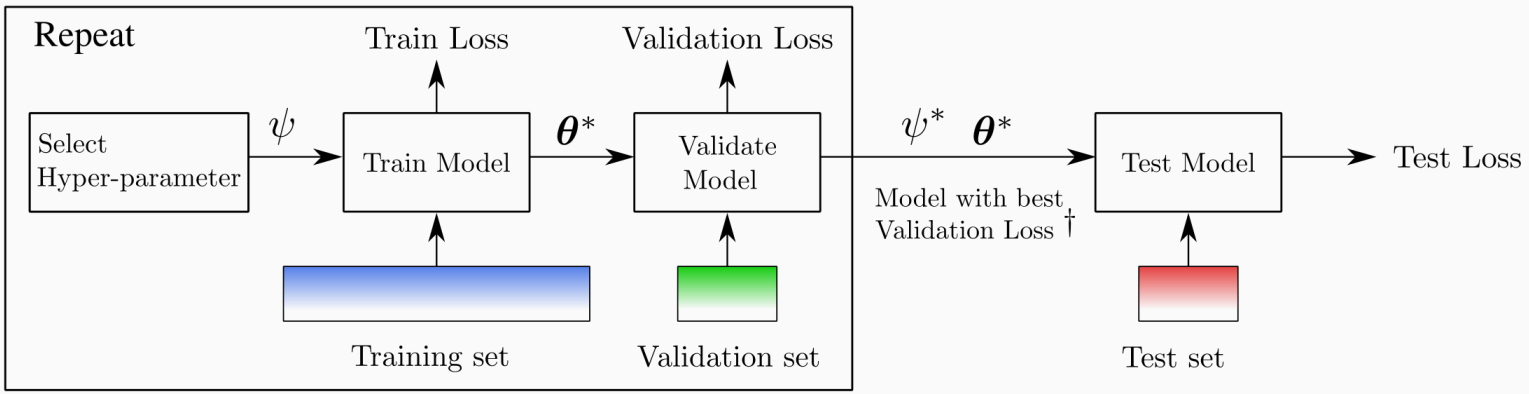
How can we prevent this?

Really, we need **three** datasets:

- a **training set** $\mathcal{D}_{train}$ which we use to **learn the model** by minimizing $\mathcal{L}_{train}(\boldsymbol{\theta})$
- a **validation set** $\mathcal{D}_{val}$ which we use to do **model selection** by selecting the hyper-parameter ψ with minimal $\mathcal{L}_{val}(\psi)$
- a **test set** $\mathcal{D}_{test}$ which we use to **evaluate the model** by computing $\mathcal{L}_{test}$

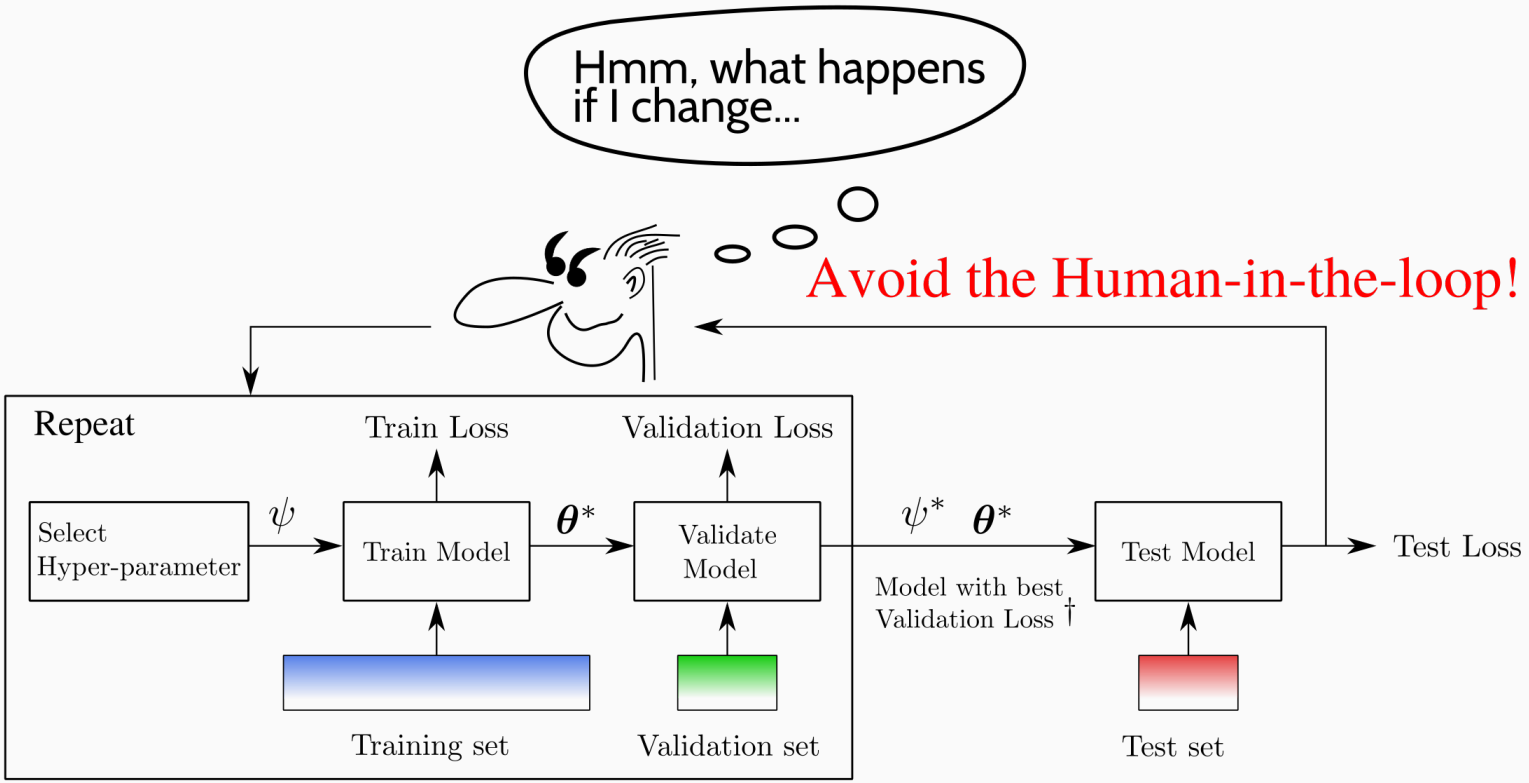


Standard Machine Learning Workflow



† optionally, retrain with selected hyper-paramter on combined training and validation set

Standard Machine Learning Workflow



† optionally, retrain with selected hyper-paramter on combined training and validation set

For a proper machine learning workflow, we need a **training set**, **validation set** and **test set**. Each should be as large as possible:

- training set should be large, in order to learn the models well
- validation set should be large, in order to reliably select the best model
- test set should be large, in order to estimate the true performance well

What can we do, if there are very few samples?

Cross-Validation

Assume we have already reserved samples for a test set. The remaining samples are split into a training set and a validation set.

Reserving samples for the validation set is called the **holdout method**. The data is split **once** into training and validation set.

In **cross-validation**, we split the data **multiple times** into training and validation set. The validation results for each split are averaged.

K-fold cross-validation is the most widely used method for cross-validation.

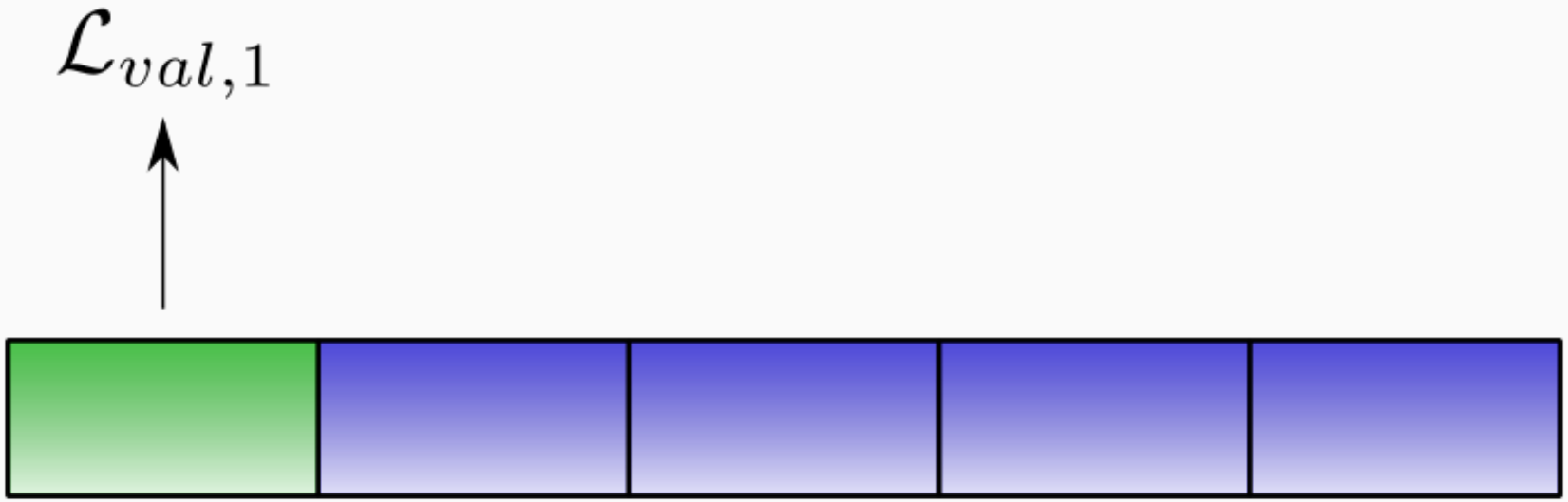
- Randomly split the data in K parts (folds)
- For each hyper-parameter configuration ψ:
  - For $k = 1 \dots K$:
    - use the k^{th} fold as validation set
    - train model on all folds, except the k^{th} fold
    - evaluate on k^{th} fold, yielding $\mathcal{L}_{val,k}$
  - Let $\mathcal{L}_{val}(\psi) = \frac{1}{K} \sum_{k=1}^{K} \mathcal{L}_{val,k}$
- Select the hyper-parameter ψ^* with lowest $\mathcal{L}_{val}(\psi)$
- Re-train on all folds with ψ^*

For each hyper-parameter configuration ψ:

For each hyper-parameter configuration ψ:

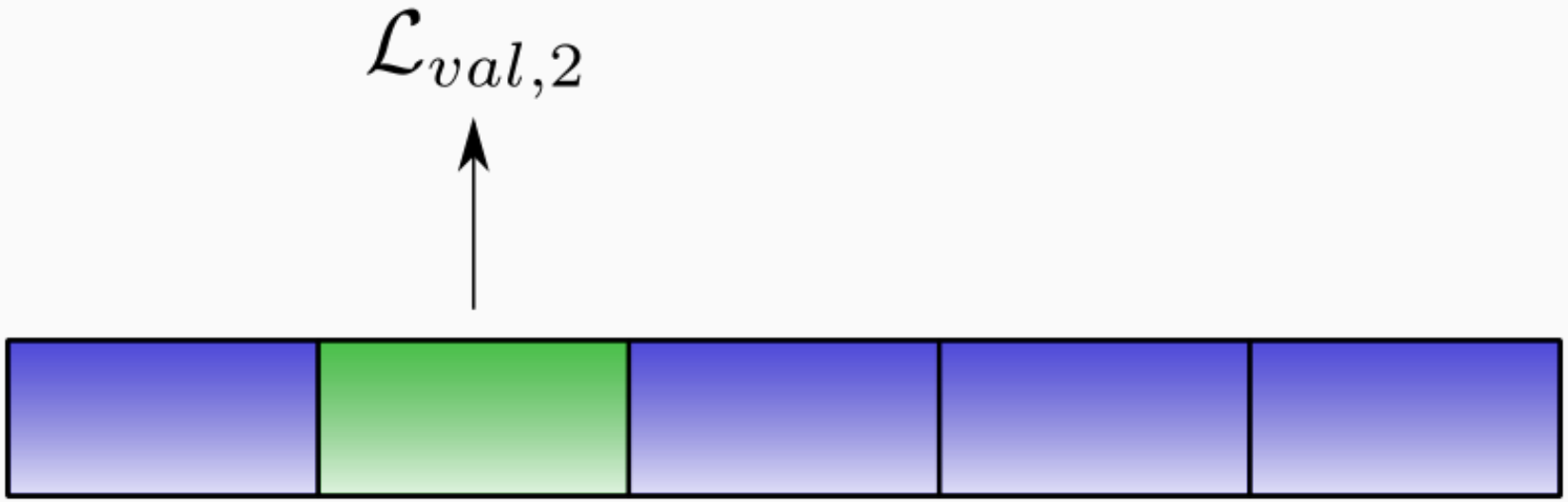
Split into $K = 5$ folds.

For each hyper-parameter configuration ψ:



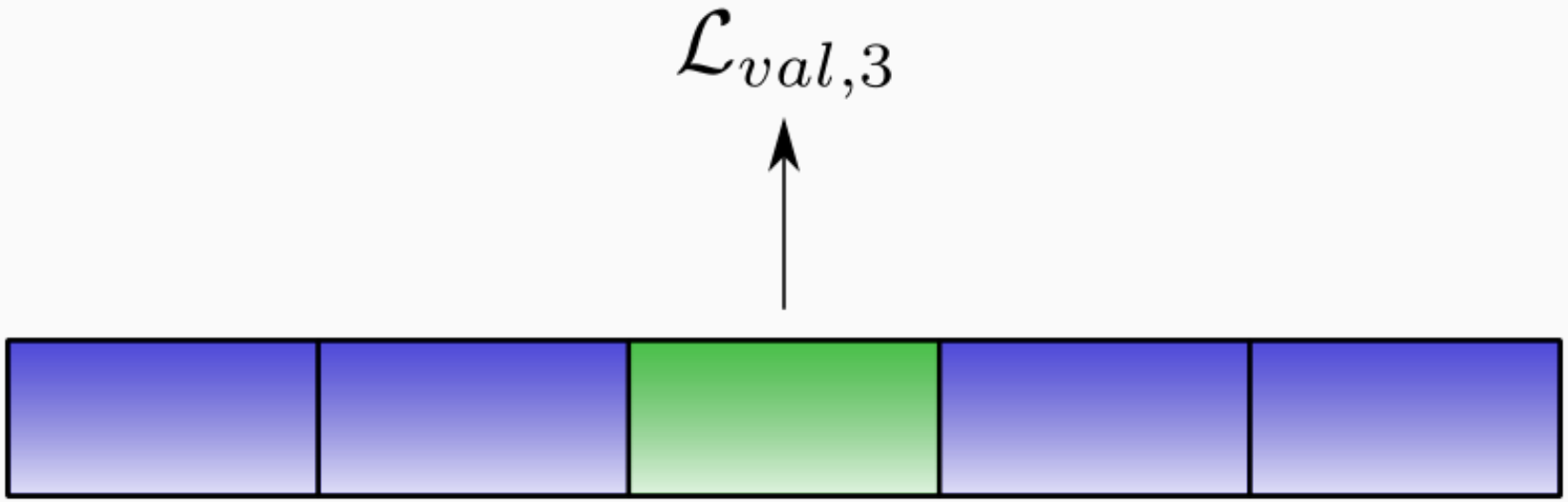
Train in turn on all except the k^{th} fold and compute $\mathcal{L}_{val,k}$.

For each hyper-parameter configuration ψ:



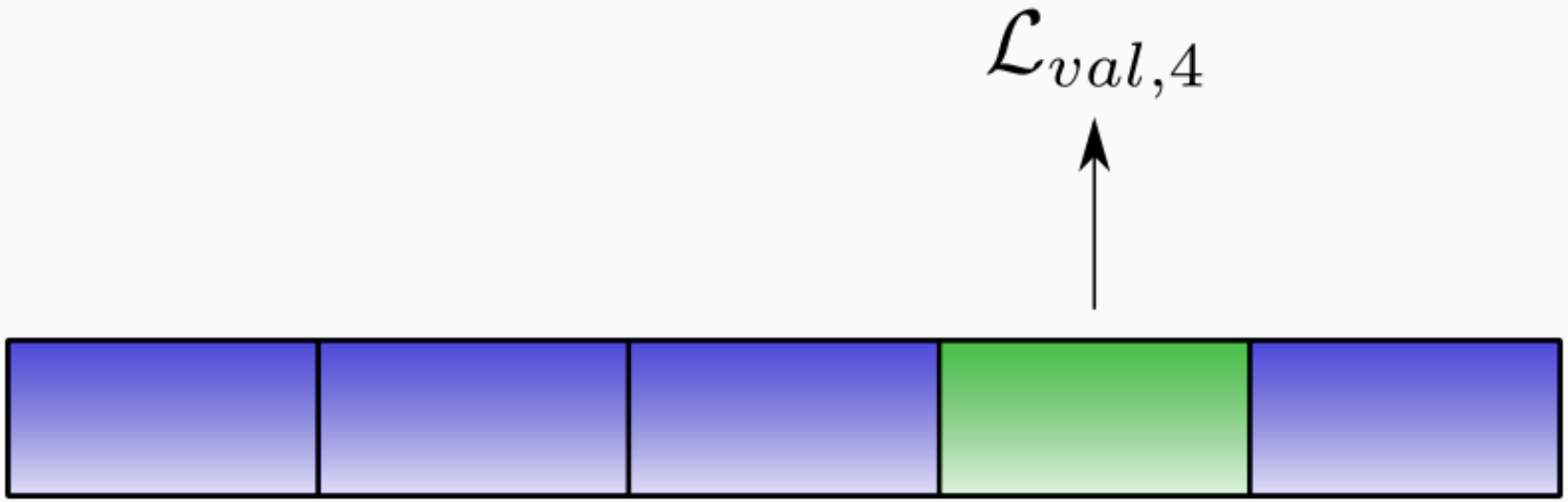
Train in turn on all except the k^{th} fold and compute $\mathcal{L}_{val,k}$.

For each hyper-parameter configuration ψ:



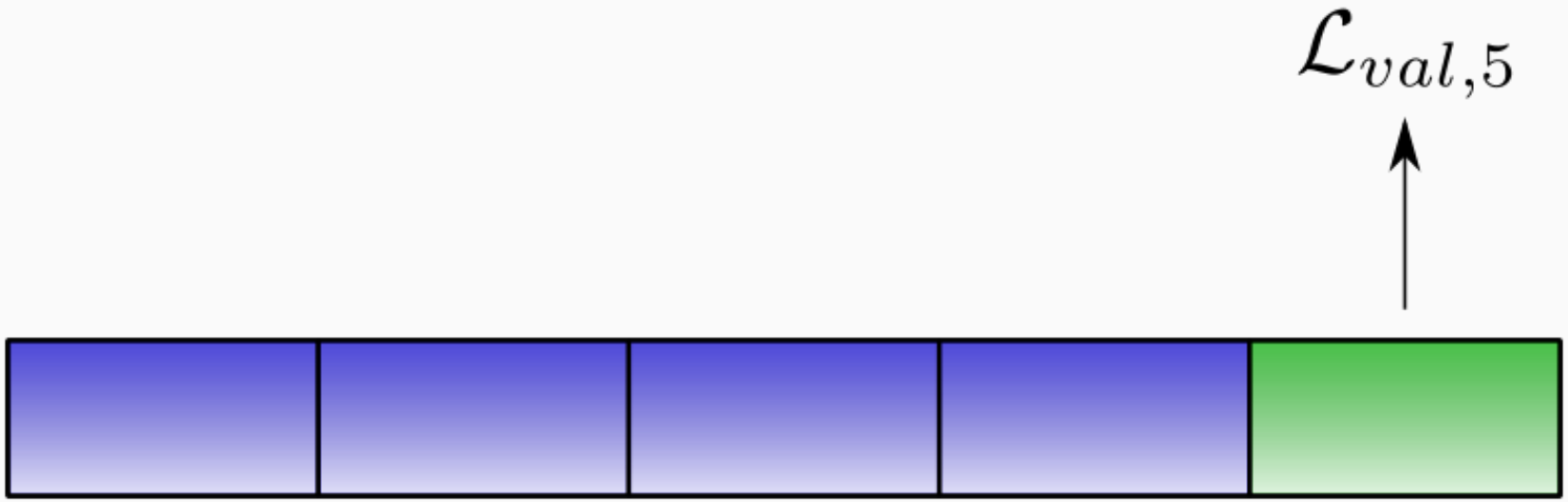
Train in turn on all except the k^{th} fold and compute $\mathcal{L}_{val,k}$.

For each hyper-parameter configuration ψ:



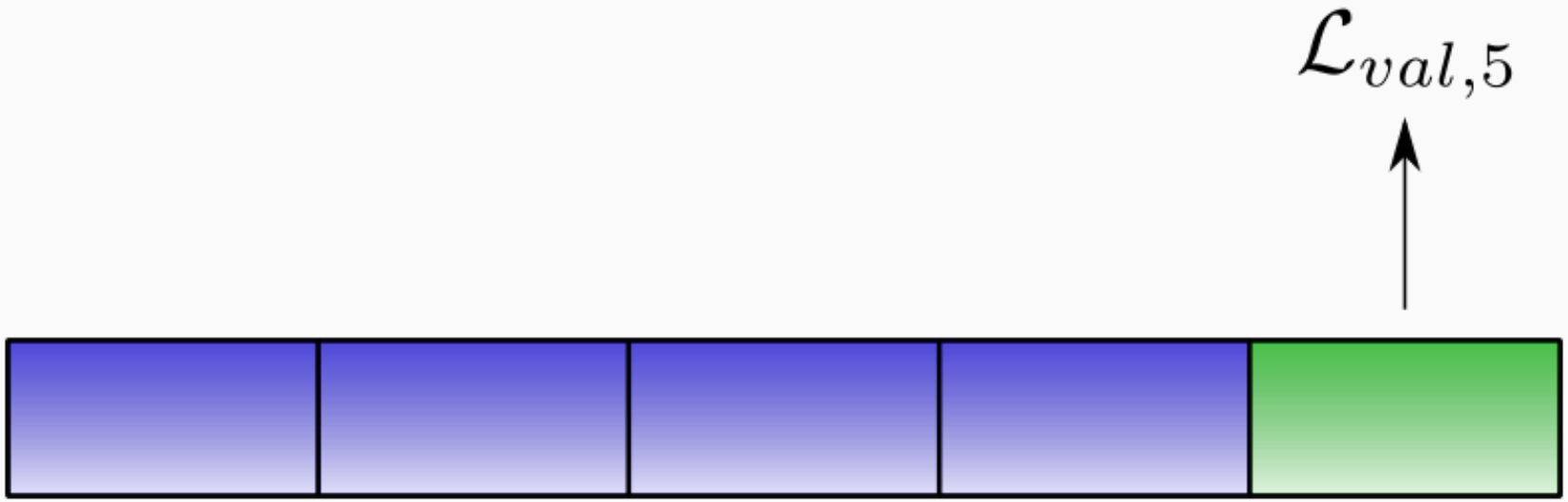
Train in turn on all except the k^{th} fold and compute $\mathcal{L}_{val,k}$.

For each hyper-parameter configuration ψ:



Train in turn on all except the k^{th} fold and compute $\mathcal{L}_{val,k}$.

For each hyper-parameter configuration ψ:



$\mathcal{L}_{val}(\psi) = \frac{1}{K} \sum_{k=1}^{K} \mathcal{L}_{val,k}$

K-fold Cross-Validation Vs. Holdout Method

- K-fold cross-validation is K times more expensive:
  - The holdout method **trains each model just once**
  - K-fold cross-validation **trains each model K times**
- Holdout uses just a part of the available data for the validation loss, while K-fold uses all available data.
- Thus, the validation loss measured with the holdout method is **statistically less accurate** than for K-fold cross-validation
- However, this might not matter, if the validation set is large enough
- **If dataset is (relatively) large and training is expensive $\Rightarrow$ Holdout method**
- **If dataset is small and training is (relatively) cheap $\Rightarrow$ K-fold cross-validation**

More Cross-Validation Methods

- **Leave-one-out cross-validation**: extreme case of K-fold with $K = N$, i.e. we treat each **single** sample in turn as validation set.
- **Repeated random sub-sampling**: instead of fixing K folds, we draw K times a random subset as validation set.

Summary

- **Overfitting**
  - Model class too powerful
  - Too little data for chosen model class
  - Memorizing the training data, not **generalizing** well
- Standard ML workflow: split data into
  - **training set**
  - **validation set**
  - **test set**
- training set to train model parameters $\boldsymbol{\theta}$
- validation set to select model (hyper-parameters) ψ
- test set to estimate true performance (true loss)
- **Cross-validation** if we have very little data