Lecture 3: Supervised Learning Methods

KI-Workshop
(HFT Stuttgart, 8-9 Nov 2023)

Michael Mommert
University of St. Gallen (soon-to-be HFT Stuttgart)

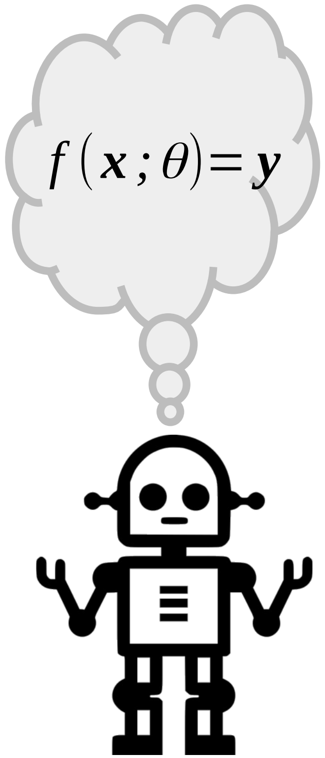
Today's lecture

Linear models

Nearest Neighbor models

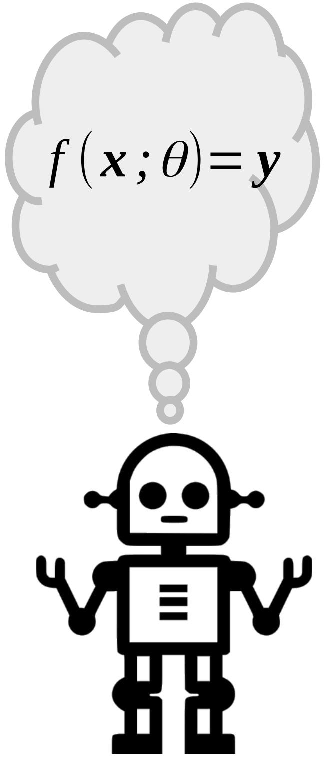
Tree-based models

Reminder: general supervised learning pipeline



Reminder: general supervised learning pipeline

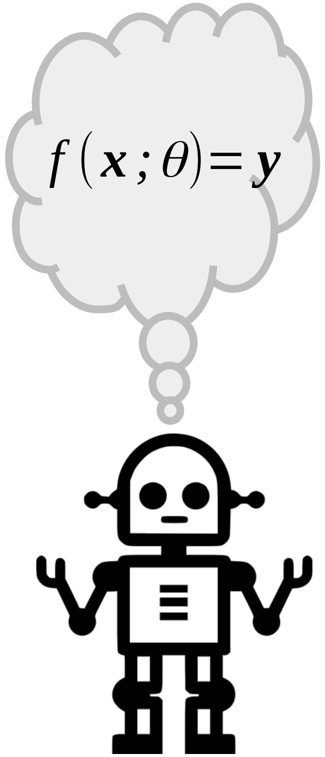
1) Feature engineering: raw data → features



Reminder: general supervised learning pipeline

1) Feature engineering: raw data → features

2) Data scaling

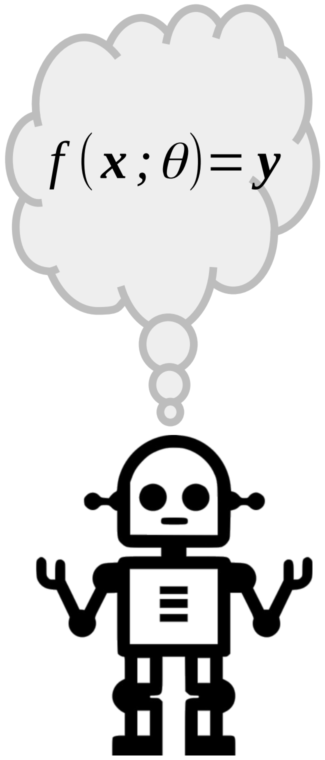


Reminder: general supervised learning pipeline

1) Feature engineering: raw data → features

2) Data scaling

3) Data splitting → training, validation, test data



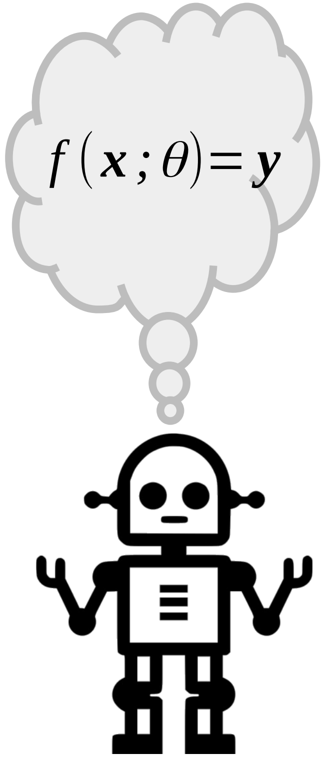
Reminder: general supervised learning pipeline

1) Feature engineering: raw data → features

2) Data scaling

3) Data splitting → training, validation, test data

4) Define hyperparameters



Reminder: general supervised learning pipeline

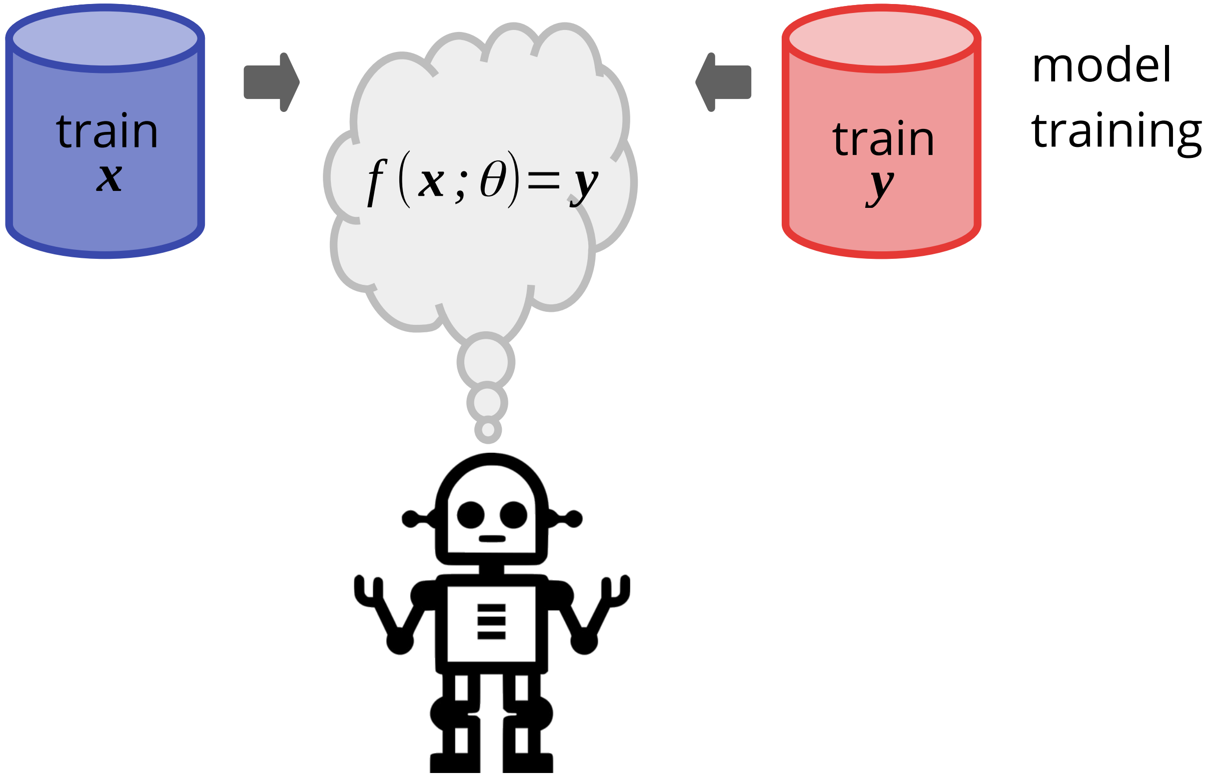
1) Feature engineering: raw data → features

2) Data scaling

3) Data splitting → training, validation, test data

4) Define hyperparameters

5) Train model on training data for fixed hyperparameters



Reminder: general supervised learning pipeline

1) Feature engineering: raw data → features

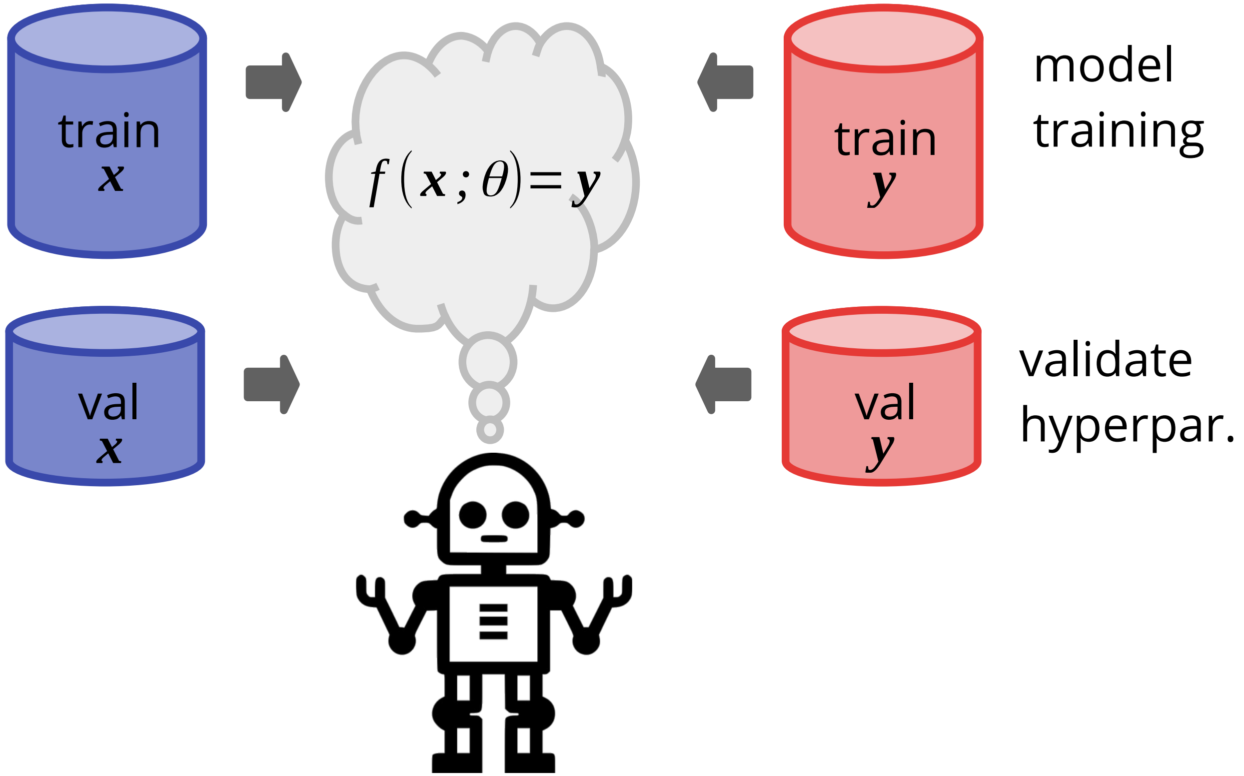
2) Data scaling

3) Data splitting → training, validation, test data

4) Define hyperparameters

5) Train model on training data for fixed hyperparameters

6) Evaluate model on validation data



Reminder: general supervised learning pipeline

1) Feature engineering: raw data → features

2) Data scaling

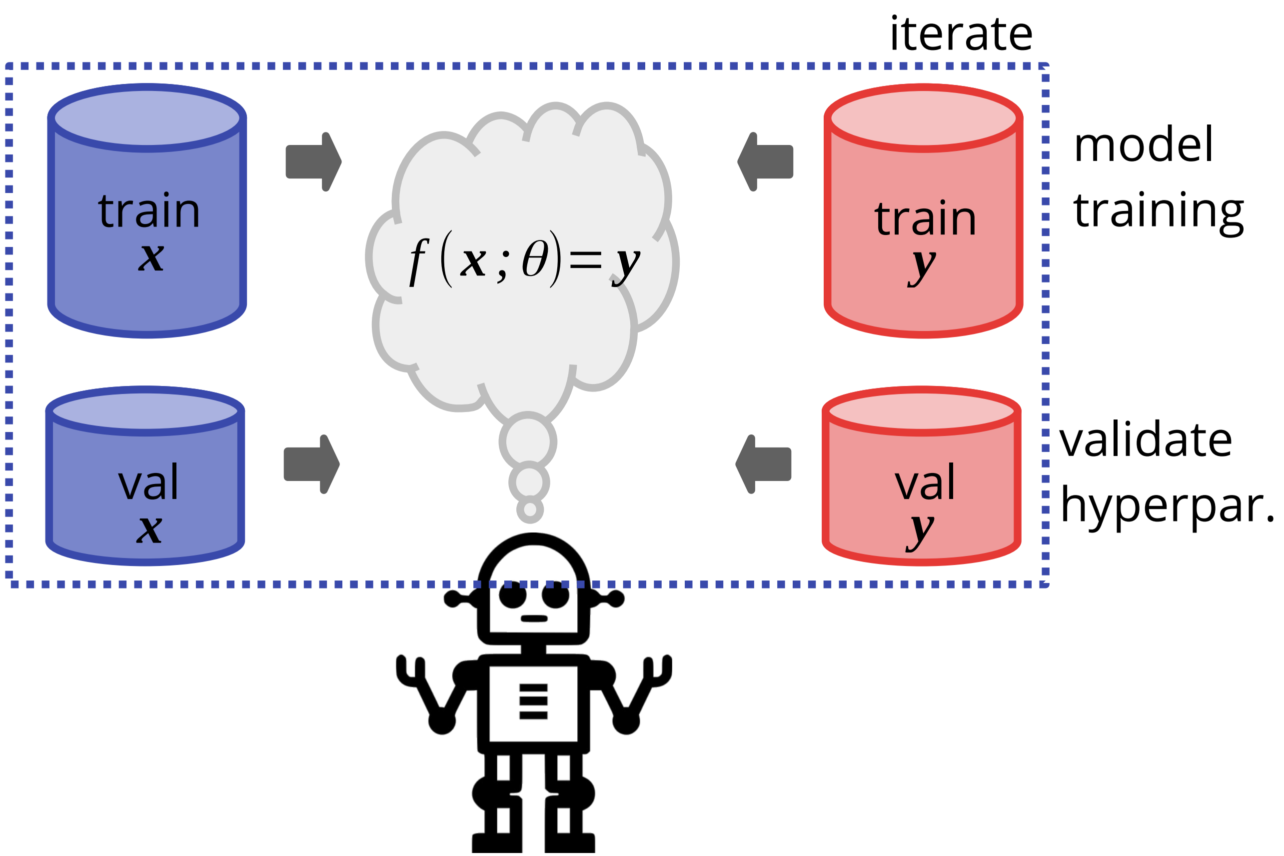
3) Data splitting → training, validation, test data

4) Define hyperparameters

5) Train model on training data for fixed hyperparameters

6) Evaluate model on validation data

7) Repeat 4) to 6) until performance on validation data maximized



Reminder: general supervised learning pipeline

1) Feature engineering: raw data → features

2) Data scaling

3) Data splitting → training, validation, test data

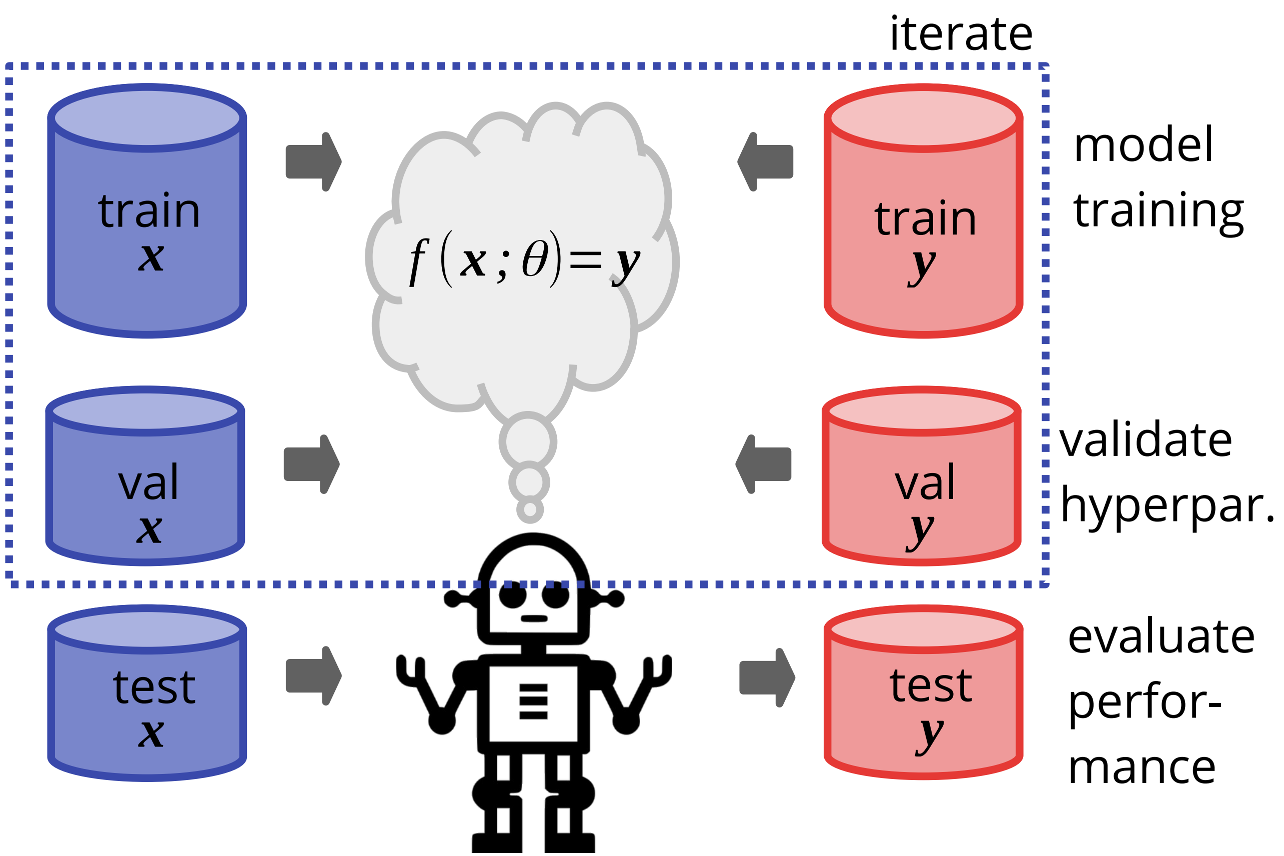
4) Define hyperparameters

5) Train model on training data for fixed hyperparameters

6) Evaluate model on validation data

7) Repeat 4) to 6) until performance on validation data maximized

8) Evaluate trained model on test data
→ report test data performance



Reminder: general supervised learning pipeline

1) Feature engineering: raw data → features

2) Data scaling

3) Data splitting → training, validation, test data

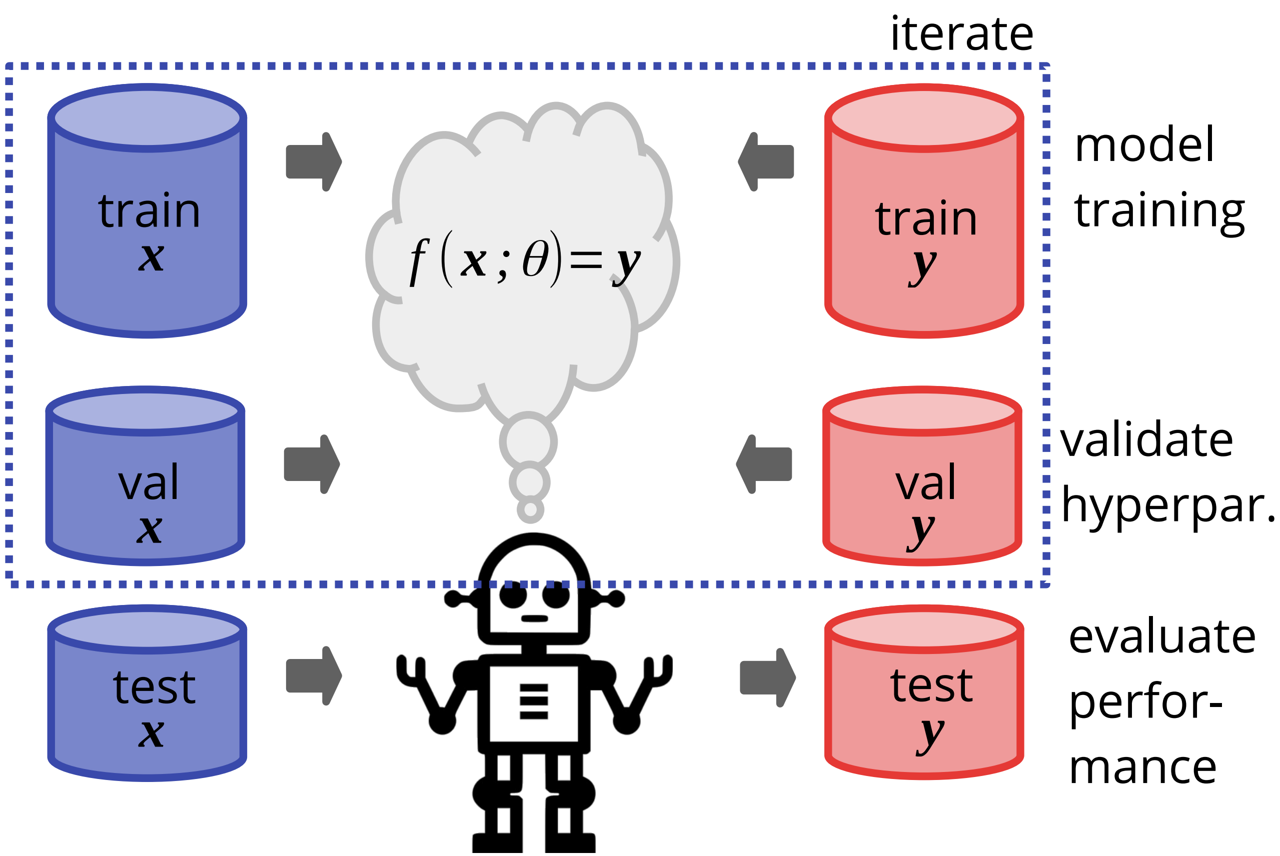
4) Define hyperparameters

5) Train model on training data for fixed hyperparameters

6) Evaluate model on validation data

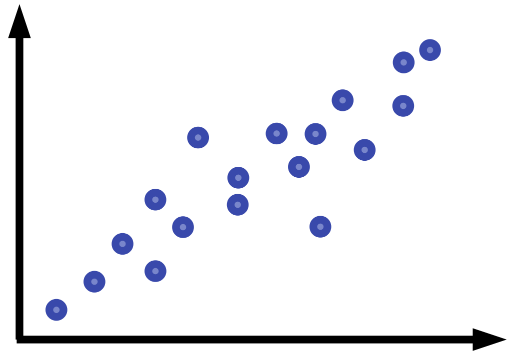
7) Repeat 4) to 6) until performance on validation data maximized

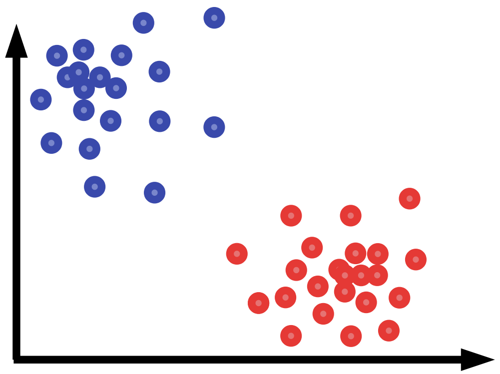
8) Evaluate trained model on test data
→ report test data performance



Goal: maximize performance while preventing overfitting!

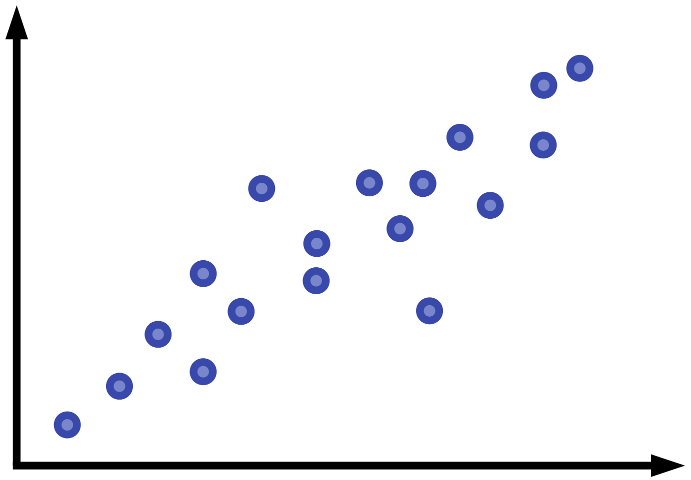
Linear models



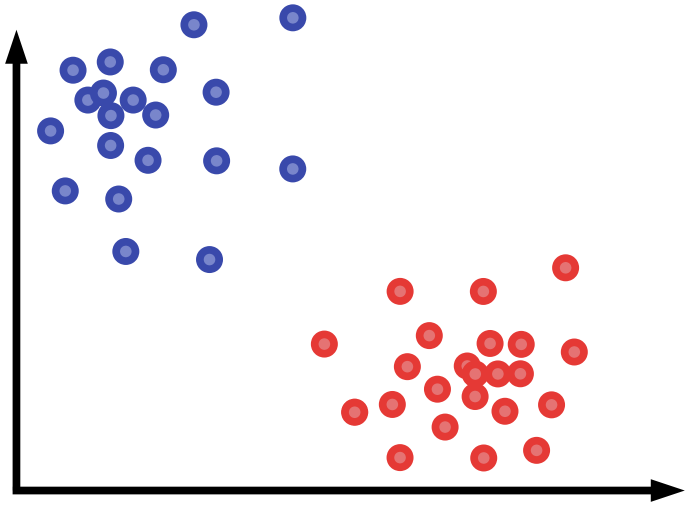


Linear models

Linear models assume **linearity** in the underlying data. They are rather simple but convey many of the concepts utilized in other, more complex models.



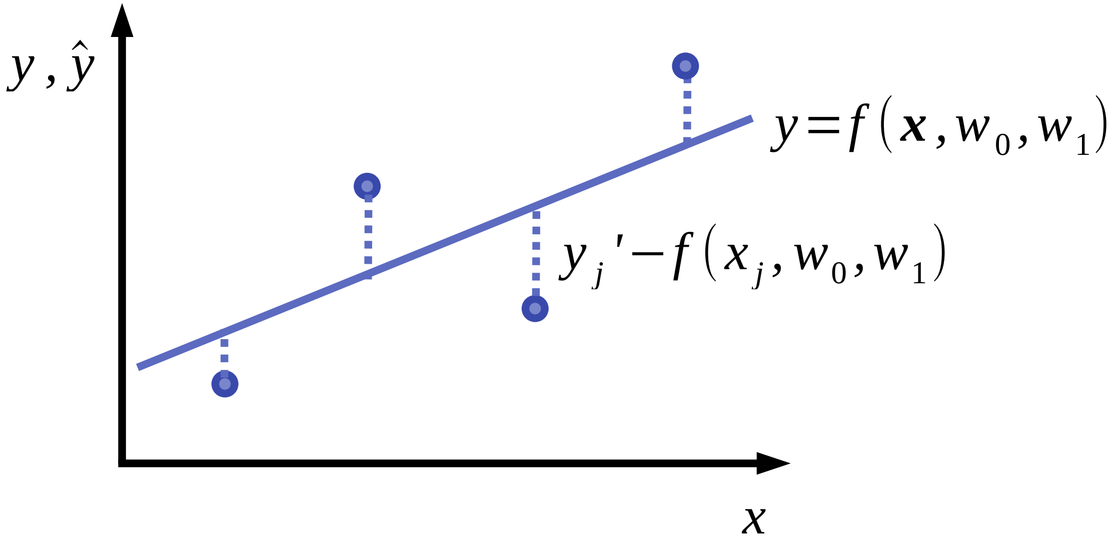
Linear regression



Linear classification

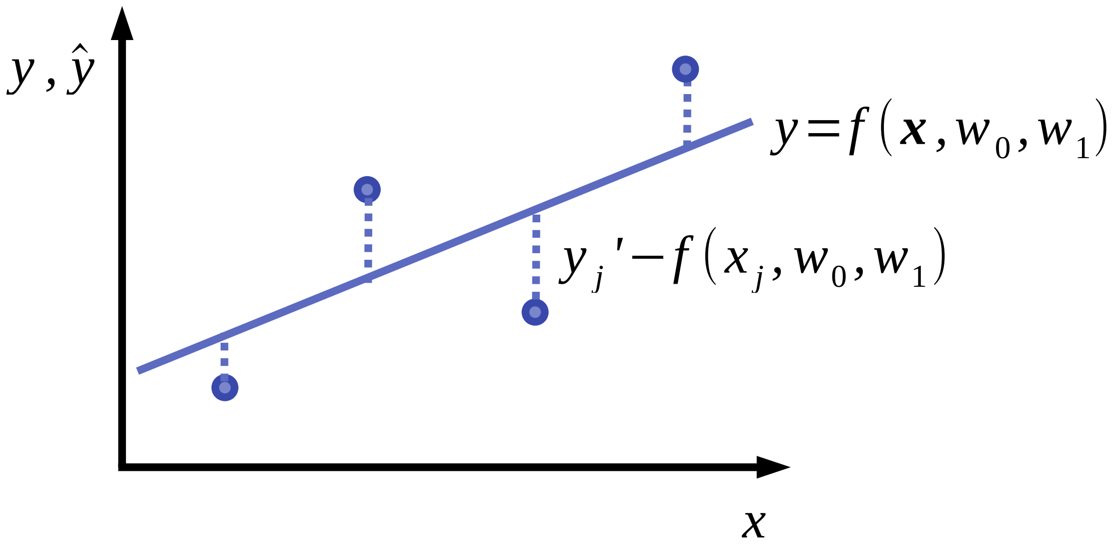
Linear regression (univariate)

Find weights w_0 and w_1 so that the linear function $f(x) = w_1 x + w_o$ with input x and output y best fits the data containing ground-truth values y'.



Linear regression (univariate)

Find weights w_0 and w_1 so that the linear function $f(x)=w_1 x+w_o$ with input x and output y best fits the data containing ground-truth values y'.

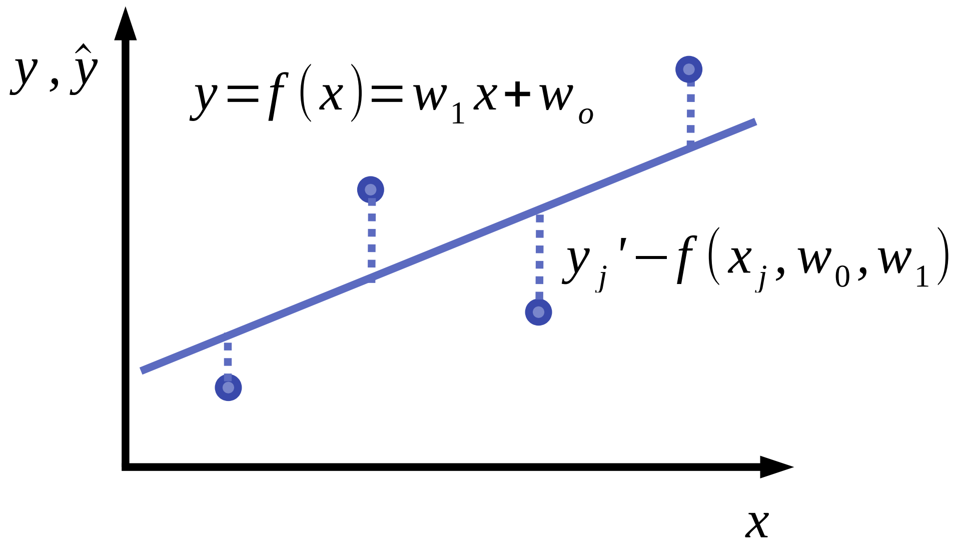


How can we learn w_0 and w_1 from data?

Linear regression (univariate) – Least squares fitting

Idea: minimize squared errors of prediction with respect to ground truth for each data point:

for data point j: $[y_j' - f(x_j, w_0, w_1)]^2$

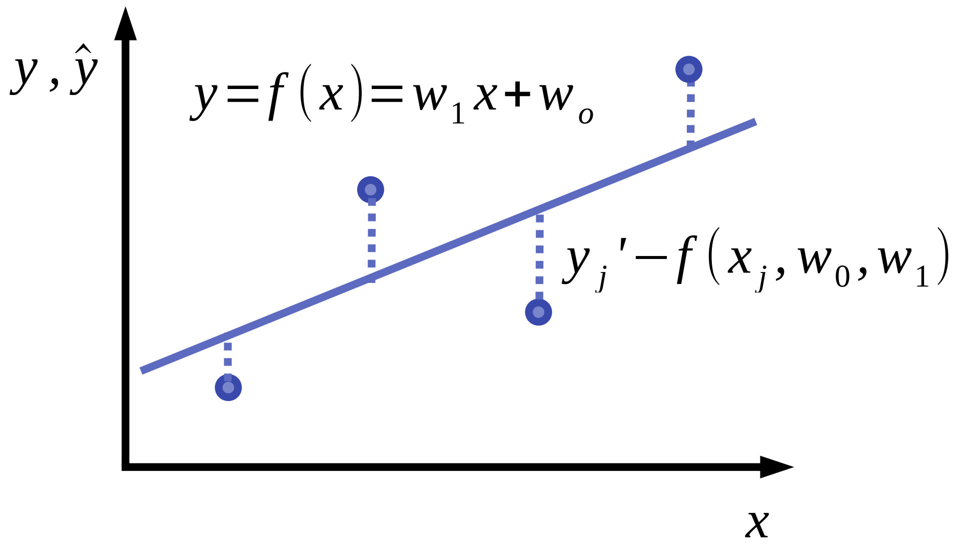


Linear regression (univariate) – Least squares fitting

Idea: minimize squared errors of prediction with respect to ground truth for each data point:

for data point j: $[y_j' - f(x_j, w_0, w_1)]^2$

We define a **Loss** (or Objective) function that is the sum of the squared errors over all data points:



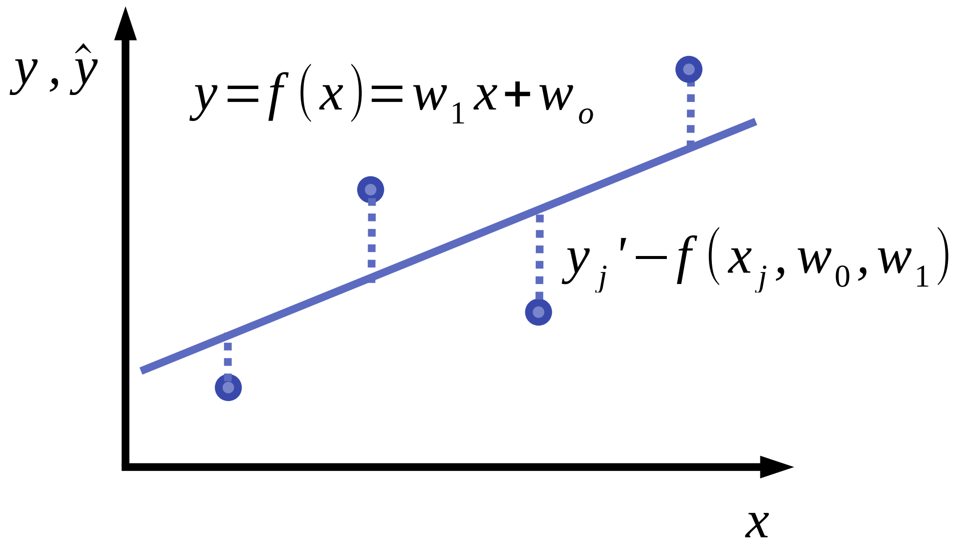
Linear regression (univariate) – Least squares fitting

Idea: minimize squared errors of prediction with respect to ground truth for each data point:

for data point j: $[y_j' - f(x_j, w_0, w_1)]^2$

We define a **Loss** (or Objective) function that is the sum of the squared errors over all data points:

$$L = \sum_j^N (y_j' - f(x_j, w_0, w_1))^2 = \sum_j^N (y_j' - (w_1 x_j + w_0))^2$$



Linear regression (univariate) – Least squares fitting

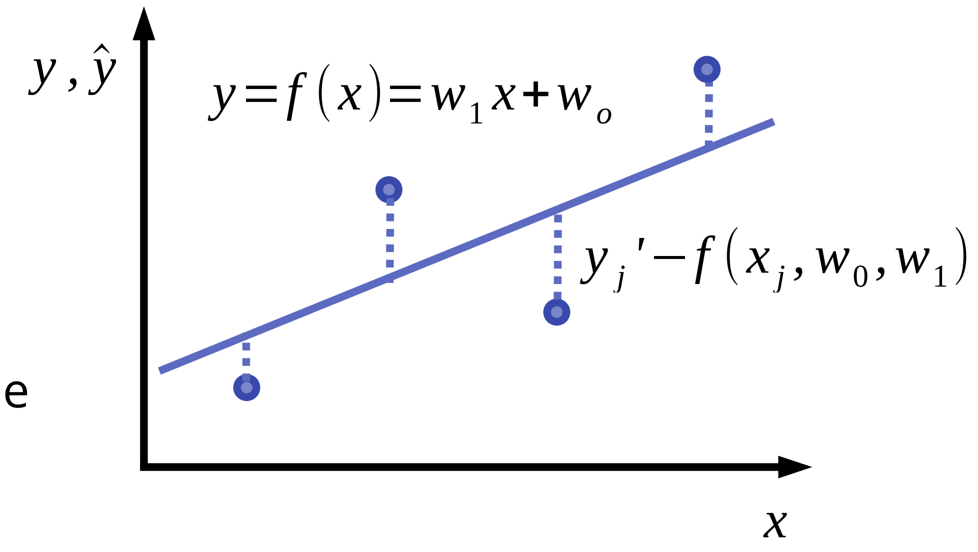
Idea: minimize squared errors of prediction with respect to ground truth for each data point:

for data point j: $[y_j' - f(x_j, w_0, w_1)]^2$

We define a **Loss** (or Objective) function that is the sum of the squared errors over all data points:

$$L = \sum_j^N (y_j' - f(x_j, w_0, w_1))^2 = \sum_j^N (y_j' - (w_1 x_j + w_0))^2$$

We can find the best-fit model parameters, by **minimizing** the Loss function with respect to those two model parameters:



Linear regression (univariate) – Least squares fitting

Idea: minimize squared errors of prediction with respect to ground truth for each data point:

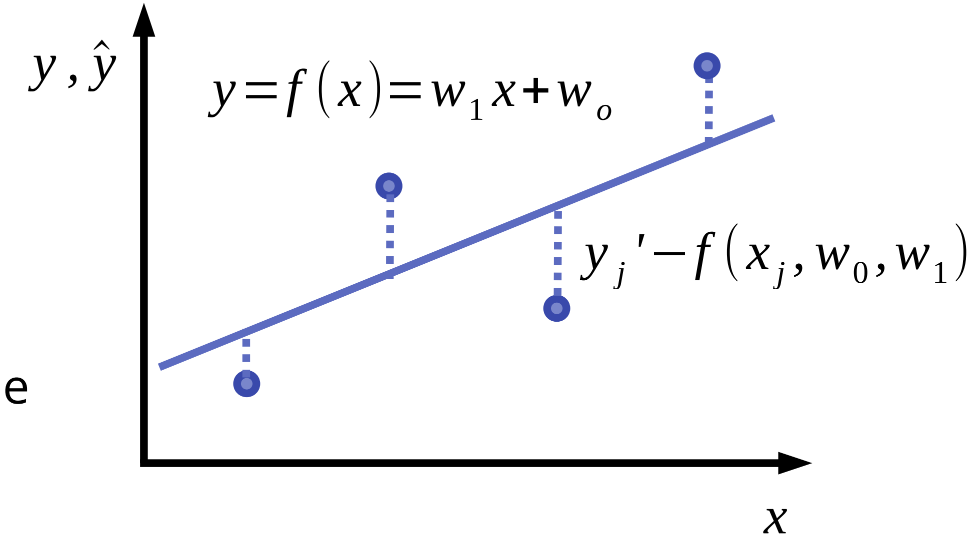
for data point j: $[y_j' - f(x_j, w_0, w_1)]^2$

We define a **Loss** (or Objective) function that is the sum of the squared errors over all data points:

$$L = \sum_j^N (y_j' - f(x_j, w_0, w_1))^2 = \sum_j^N (y_j' - (w_1 x_j + w_0))^2$$

We can find the best-fit model parameters, by **minimizing** the Loss function with respect to those two model parameters:

$$\frac{\partial L}{\partial w_0} = 0 \qquad \frac{\partial L}{\partial w_1} = 0$$



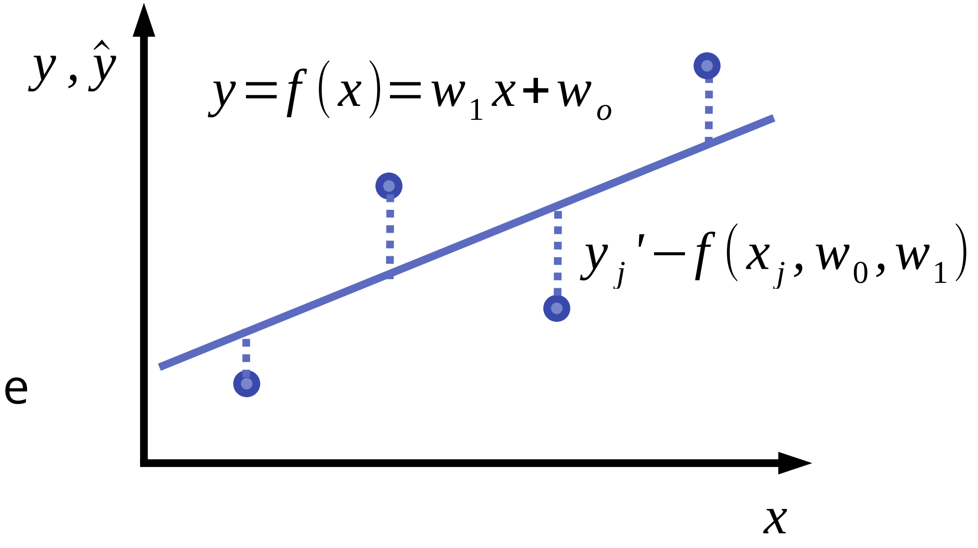
Linear regression (univariate) – Least squares fitting

Idea: minimize squared errors of prediction with respect to ground truth for each data point:

for data point j: $[y_j' - f(x_j, w_0, w_1)]^2$

We define a **Loss** (or Objective) function that is the sum of the squared errors over all data points:

$$L = \sum_j^N (y_j' - f(x_j, w_0, w_1))^2 = \sum_j^N (y_j' - (w_1 x_j + w_0))^2$$



We can find the best-fit model parameters, by **minimizing** the Loss function with respect to those two model parameters:

$$\frac{\partial L}{\partial w_0} = 0 \qquad \frac{\partial L}{\partial w_1} = 0$$

→ **closed-form expressions** for best-fit w_0 and w_1.

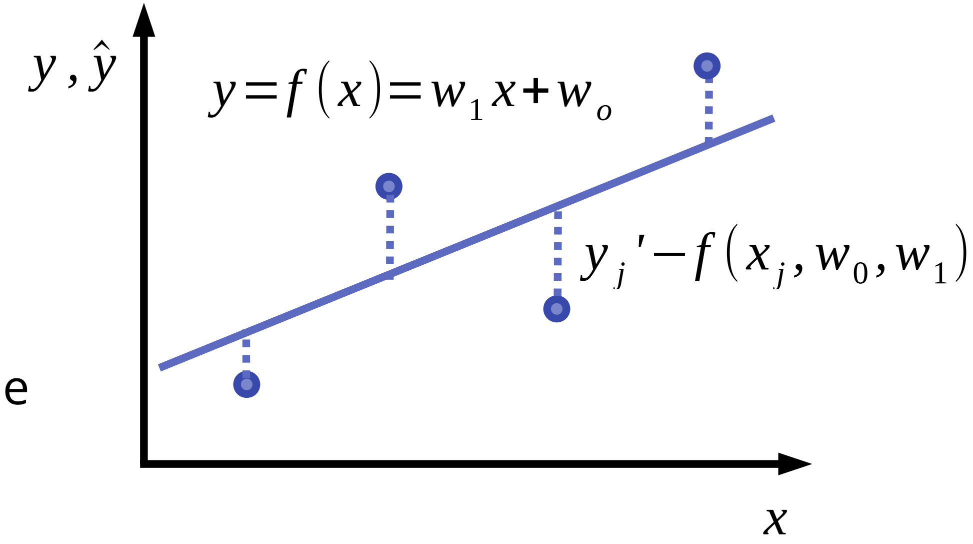
Linear regression (univariate) – Least squares fitting

Idea: minimize squared errors of prediction with respect to ground truth for each data point:

for data point j: $[y_j' - f(x_j, w_0, w_1)]^2$

We define a **Loss** (or Objective) function that is the sum of the squared errors over all data points:

$$L = \sum_j^N (y_j' - f(x_j, w_0, w_1))^2 = \sum_j^N (y_j' - (w_1 x_j + w_0))^2$$



We can find the best-fit model parameters, by **minimizing** the Loss function with respect to those two model parameters:

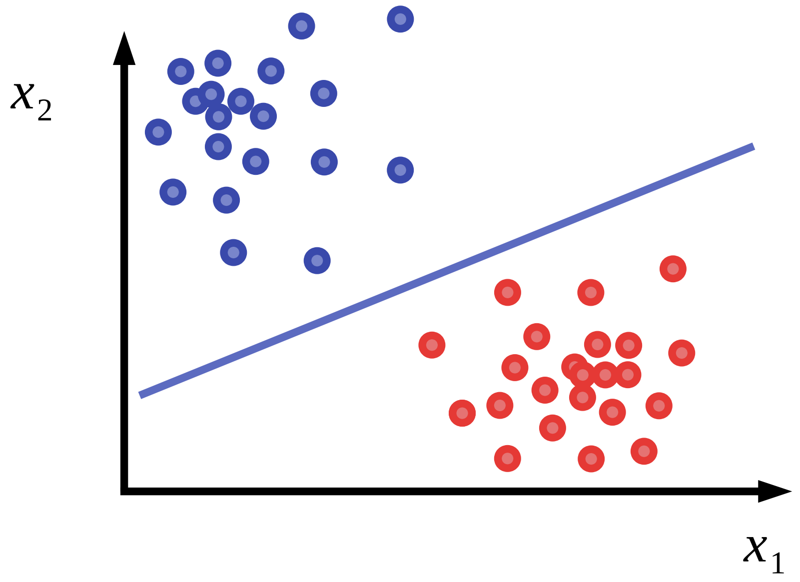
$$\frac{\partial L}{\partial w_0} = 0 \qquad \frac{\partial L}{\partial w_1} = 0$$

→ **closed-form expressions** for best-fit w_0 and w_1.

Least-squares + linear model function: the resulting minimum of the Loss function is **global**, i.e., the “learns” immediately the best-possible solution!

Linear classifier (two-dimensional case)

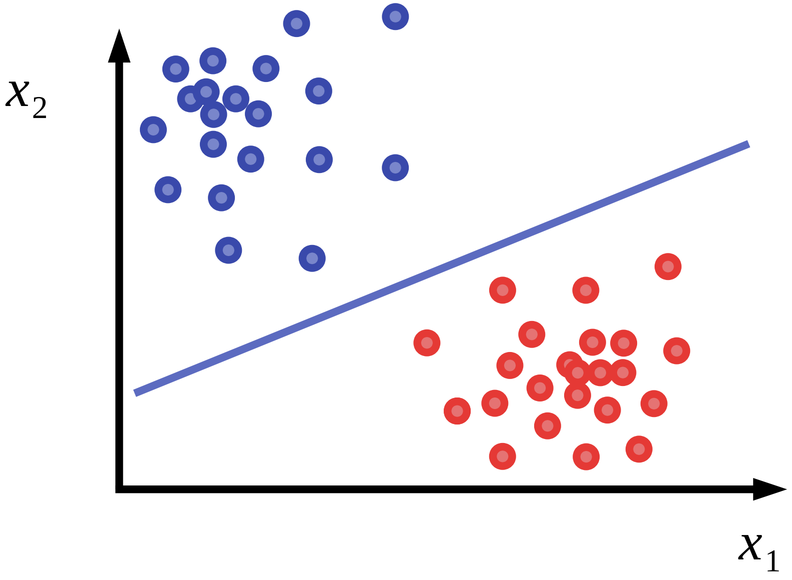
Linear functions can also be used as a classifier if the data are linearly separable.



Linear classifier (two-dimensional case)

Linear functions can also be used as a classifier if the data are linearly separable.

Define decision boundary $f(\mathbf{x},\mathbf{w}) = \mathbf{w}\,\mathbf{x} = w_0 + w_1 x_1 + w_2 x_2$ such that:

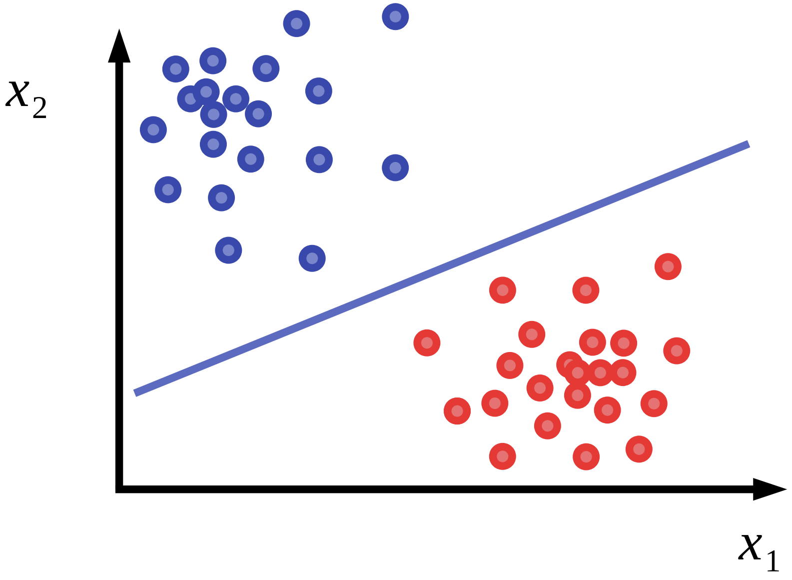


Linear classifier (two-dimensional case)

Linear functions can also be used as a classifier if the data are linearly separable.

Define decision boundary $f(\mathbf{x}, \mathbf{w}) = \mathbf{w}\,\mathbf{x} = w_0 + w_1 x_1 + w_2 x_2$ such that:

Class 1: $f(\mathbf{x}, \mathbf{w}) \geq 0$



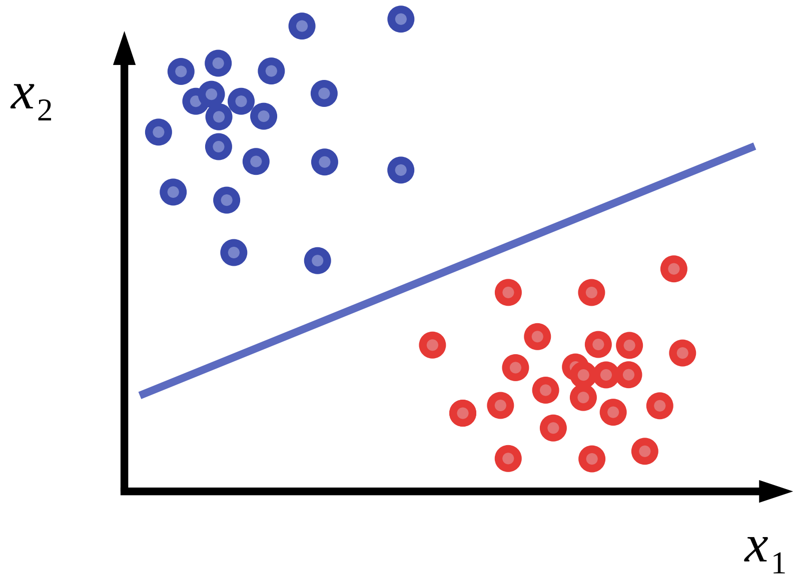
Linear classifier (two-dimensional case)

Linear functions can also be used as a classifier if the data are linearly separable.

Define decision boundary $f(\boldsymbol{x},\boldsymbol{w})=\boldsymbol{w}\,\boldsymbol{x}=w_0+w_1x_1+w_2x_2$ such that:

Class 1: $f(\boldsymbol{x},\boldsymbol{w})\geq 0$

Class 0: $f(\boldsymbol{x},\boldsymbol{w})<0$



Linear classifier (two-dimensional case)

Linear functions can also be used as a classifier if the data are linearly separable.

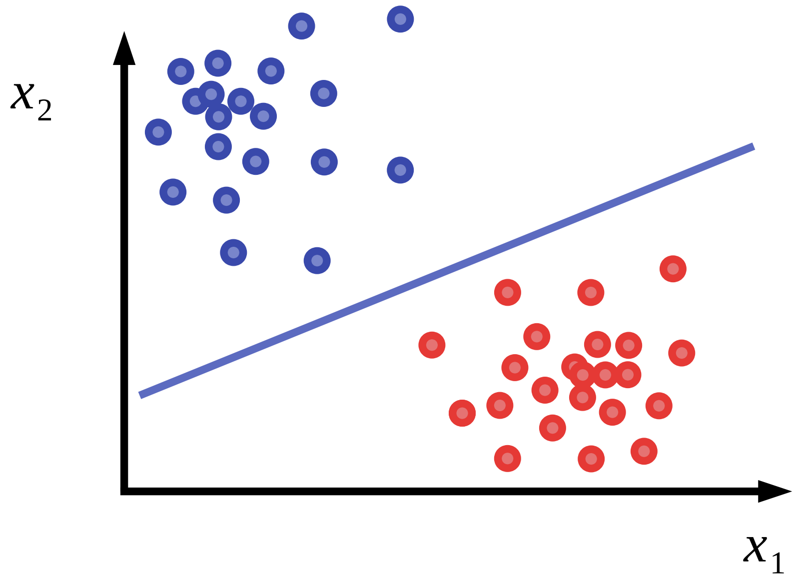
Define decision boundary $f(\boldsymbol{x},\boldsymbol{w}) = \boldsymbol{w}\,\boldsymbol{x} = w_0 + w_1 x_1 + w_2 x_2$ such that:

Class 1: $f(\boldsymbol{x},\boldsymbol{w}) \geq 0$

Class 0: $f(\boldsymbol{x},\boldsymbol{w}) < 0$

We can define class assignments through a threshold function:

$$\bar{f}(\boldsymbol{x},\boldsymbol{w}) = \begin{cases} 1 \text{ } if \text{ } f(\boldsymbol{x},\boldsymbol{w}) \geq 0 \\ 0 \text{ } if \text{ } f(\boldsymbol{x},\boldsymbol{w}) < 0 \end{cases}$$



Linear classifier (two-dimensional case)

Linear functions can also be used as a classifier if the data are linearly separable.

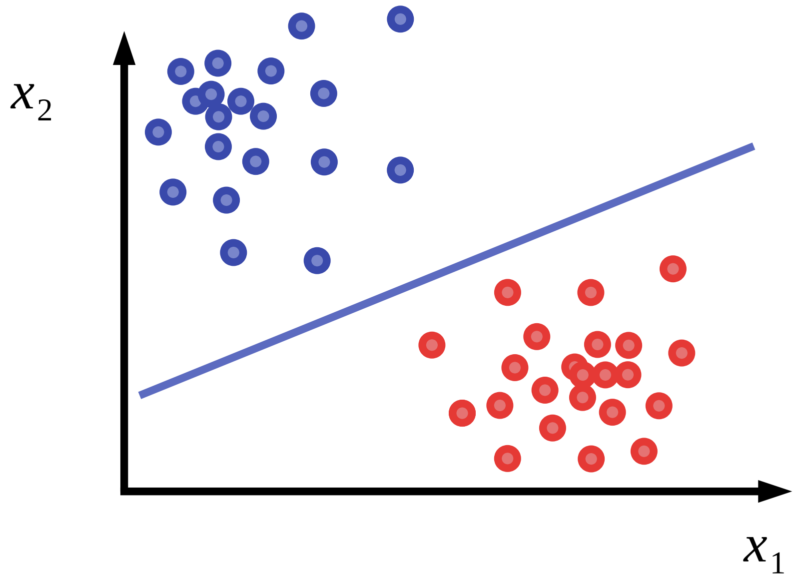
Define decision boundary $f(\boldsymbol{x},\boldsymbol{w})=\boldsymbol{w}\,\boldsymbol{x}=w_0+w_1x_1+w_2x_2$ such that:

Class 1: $f(\boldsymbol{x},\boldsymbol{w})\geq 0$

Class 0: $f(\boldsymbol{x},\boldsymbol{w})<0$

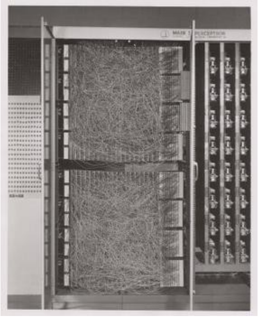
We can define class assignments through a threshold function:

$$\bar{f}(\boldsymbol{x},\boldsymbol{w})=\begin{cases}1 \text{ } if \text{ } f(\boldsymbol{x},\boldsymbol{w})\geq 0\\ 0 \text{ } if \text{ } f(\boldsymbol{x},\boldsymbol{w})<0\end{cases}$$



How can we learn $\boldsymbol{w}$ from data?

Linear classifier (two-dimensional case) – Perceptron learning rule

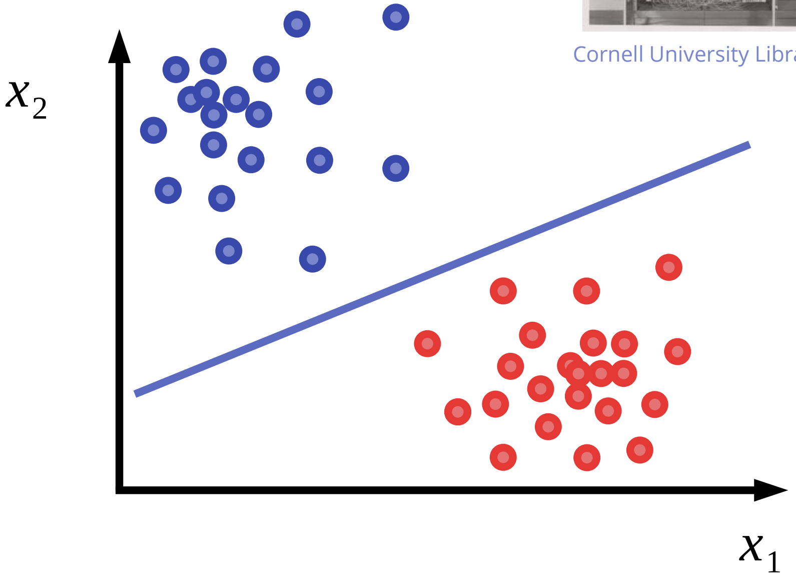


Cornell University Library

We define the following algorithm as the **Perceptron learning rule**:

We consider each data point, consisting of $\boldsymbol{x}$ and ground-truth label y' and check whether the prediction from $\bar{f}(\boldsymbol{x}, \boldsymbol{w})$ is correct, or not. If…

- $\bar{f}(\boldsymbol{x}, \boldsymbol{w}) = y$, then do nothing.
- $\bar{f}(\boldsymbol{x}, \boldsymbol{w}) = 0$ but $y' = 1$, then increase w_i if $x_i \geq 0$, or vice versa.
- $\bar{f}(\boldsymbol{x}, \boldsymbol{w}) = 1$ but $y' = 0$, then decrease w_i if $x_i \geq 0$, or vice versa.



Weights are adjusted by a step size that is called the **learning rate**. By iteratively running this algorithm over your training data multiple times, the weights can be learned so that the model performs properly. A solution is "learned" iteratively here.

Polynomial regression

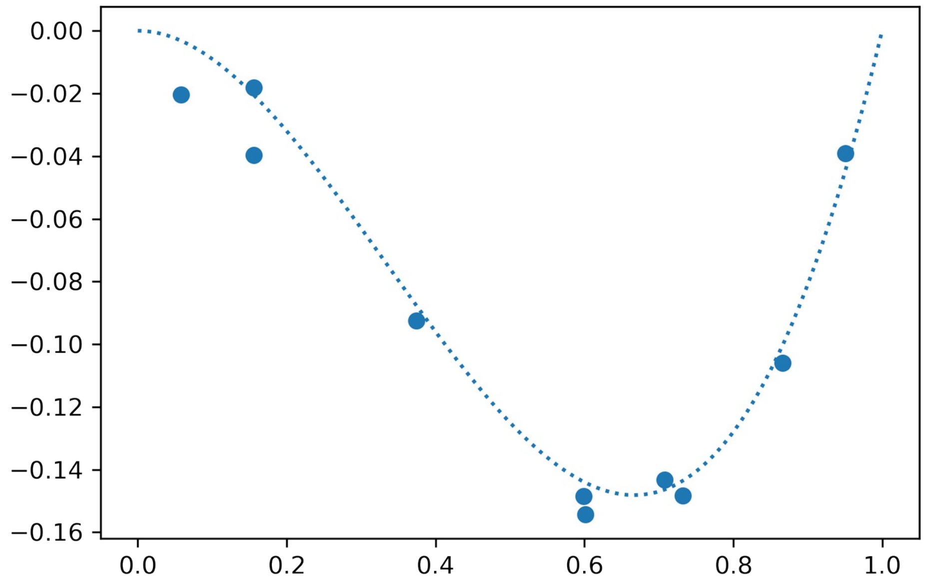
Polynomial regression

Linear models have **low predictive capacity**: can only be applied to data that are linearly distributed.

Polynomial regression

Linear models have **low predictive capacity**: can only be applied to data that are linearly distributed.

For more capacity, we can change the base function to a polynomial: $f(x) = w_0 + w_1 x + w_2 x^2 + \ldots = \sum_{i=0}^{p} w_i x^i$

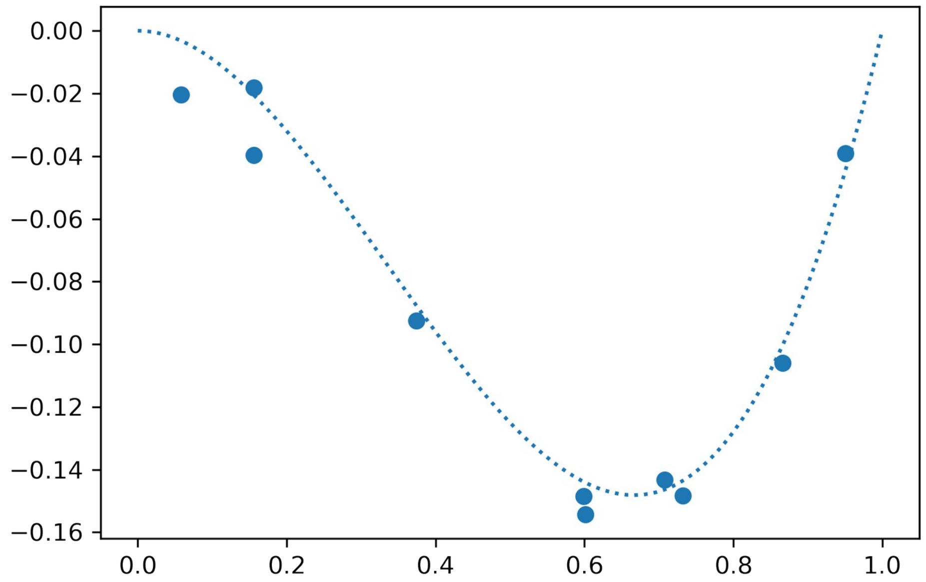


Polynomial regression

Linear models have **low predictive capacity**: can only be applied to data that are linearly distributed.

For more capacity, we can change the base function to a polynomial: $f(x) = w_0 + w_1 x + w_2 x^2 + \ldots = \sum_{i=0}^{p} w_i x^i$

The resulting regression problem is **still linear** in the weights to be found, therefore the same properties apply:



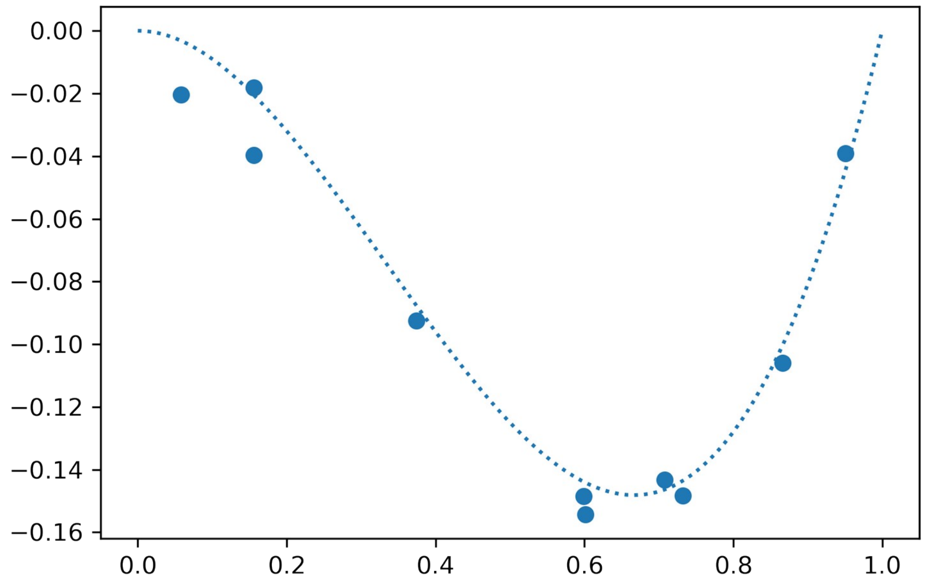
Polynomial regression

Linear models have **low predictive capacity**: can only be applied to data that are linearly distributed.

For more capacity, we can change the base function to a polynomial: $f(x) = w_0 + w_1 x + w_2 x^2 + \ldots = \sum_{i=0}^{p} w_i x^i$

The resulting regression problem is **still linear** in the weights to be found, therefore the same properties apply:

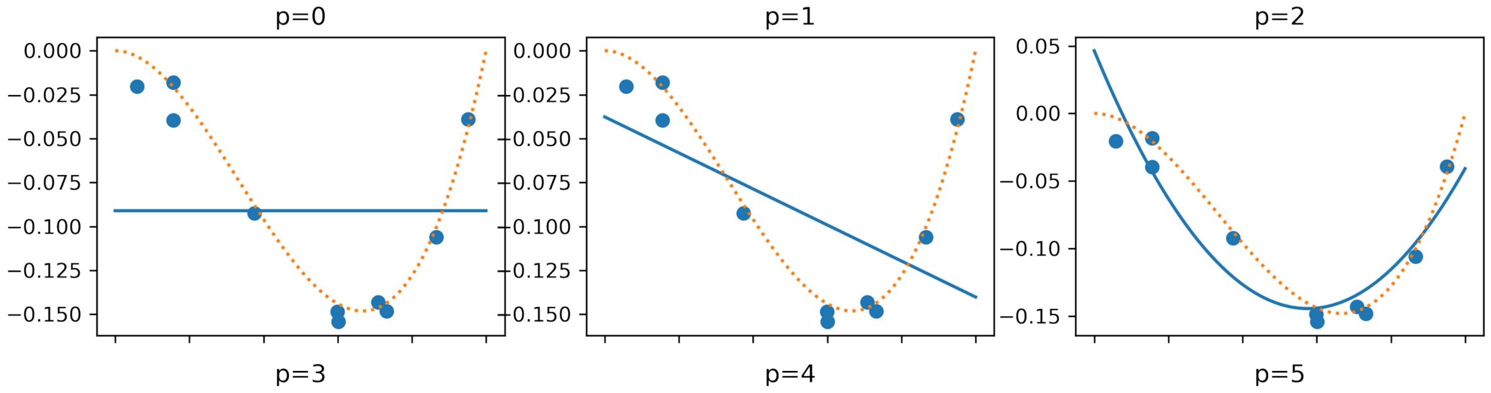
We can compute the parameters w_i to minimize the loss with a closed-form expression. This is by default the best possible solution.



Polynomial regression: an example

actual function: $f(x) = x^3 - x^2$

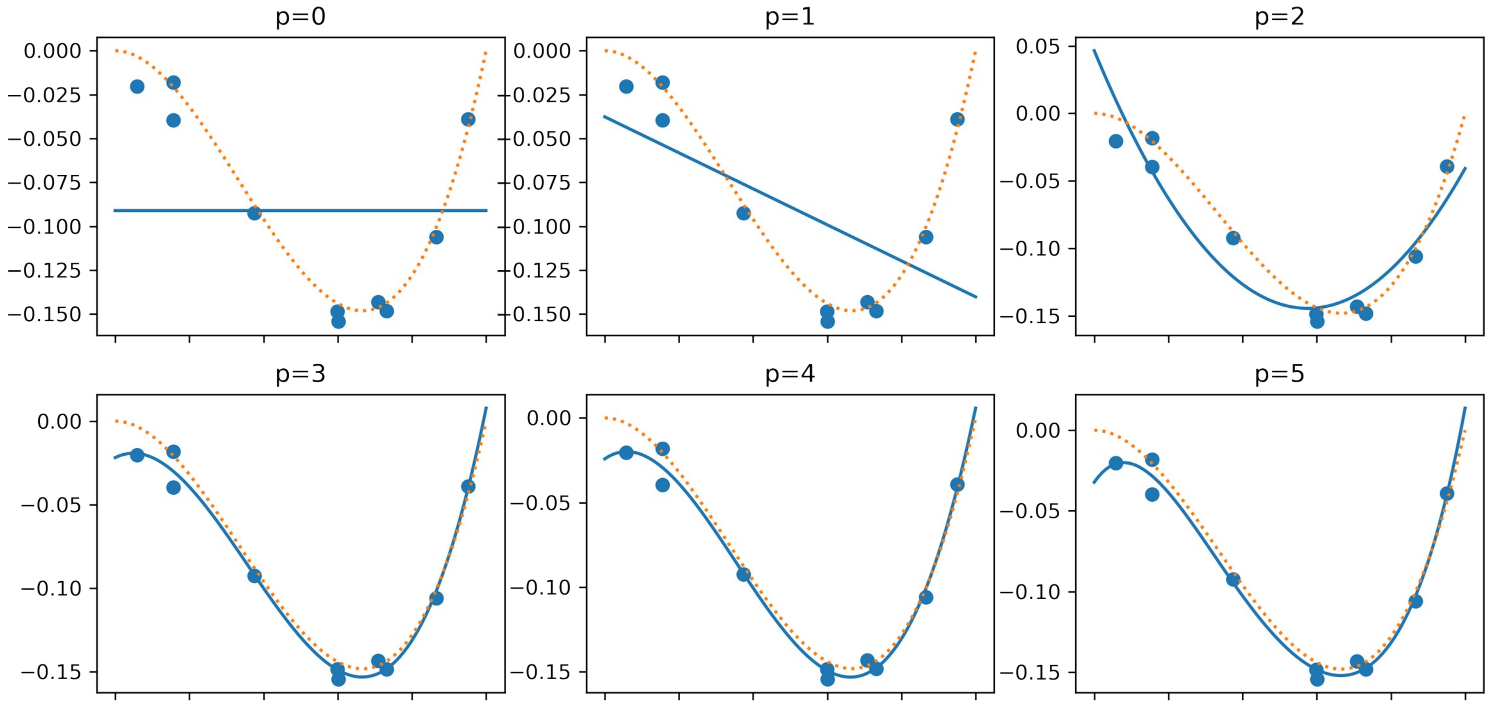
model function: $f(x) = \sum_{i=0}^{p} w_i x^i$



Polynomial regression: an example

actual function: $f(x) = x^3 - x^2$

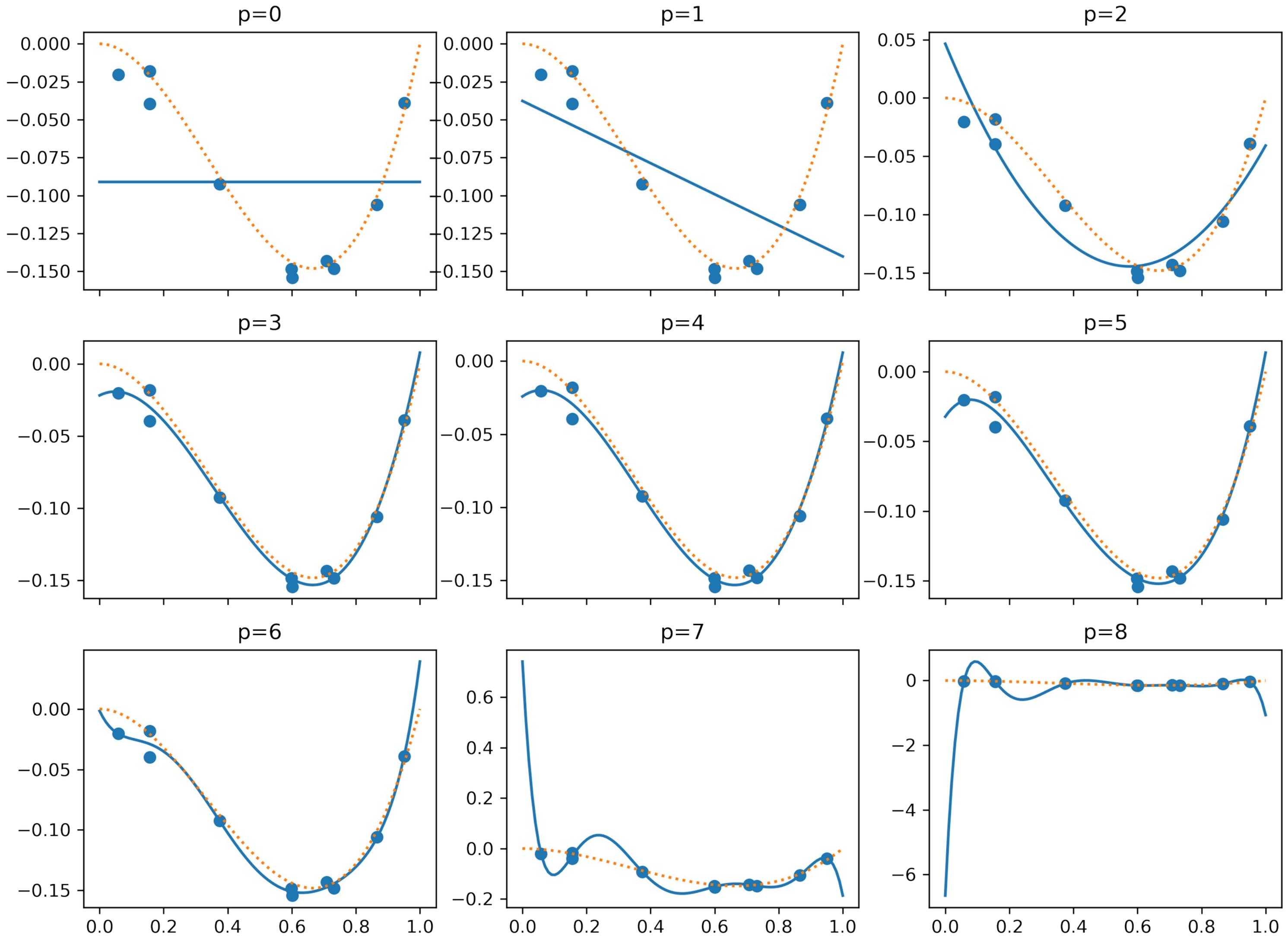
model function: $f(x) = \sum_{i=0}^{p} w_i x^i$



Polynomial regression: an example

actual function: $f(x) = x^3 - x^2$

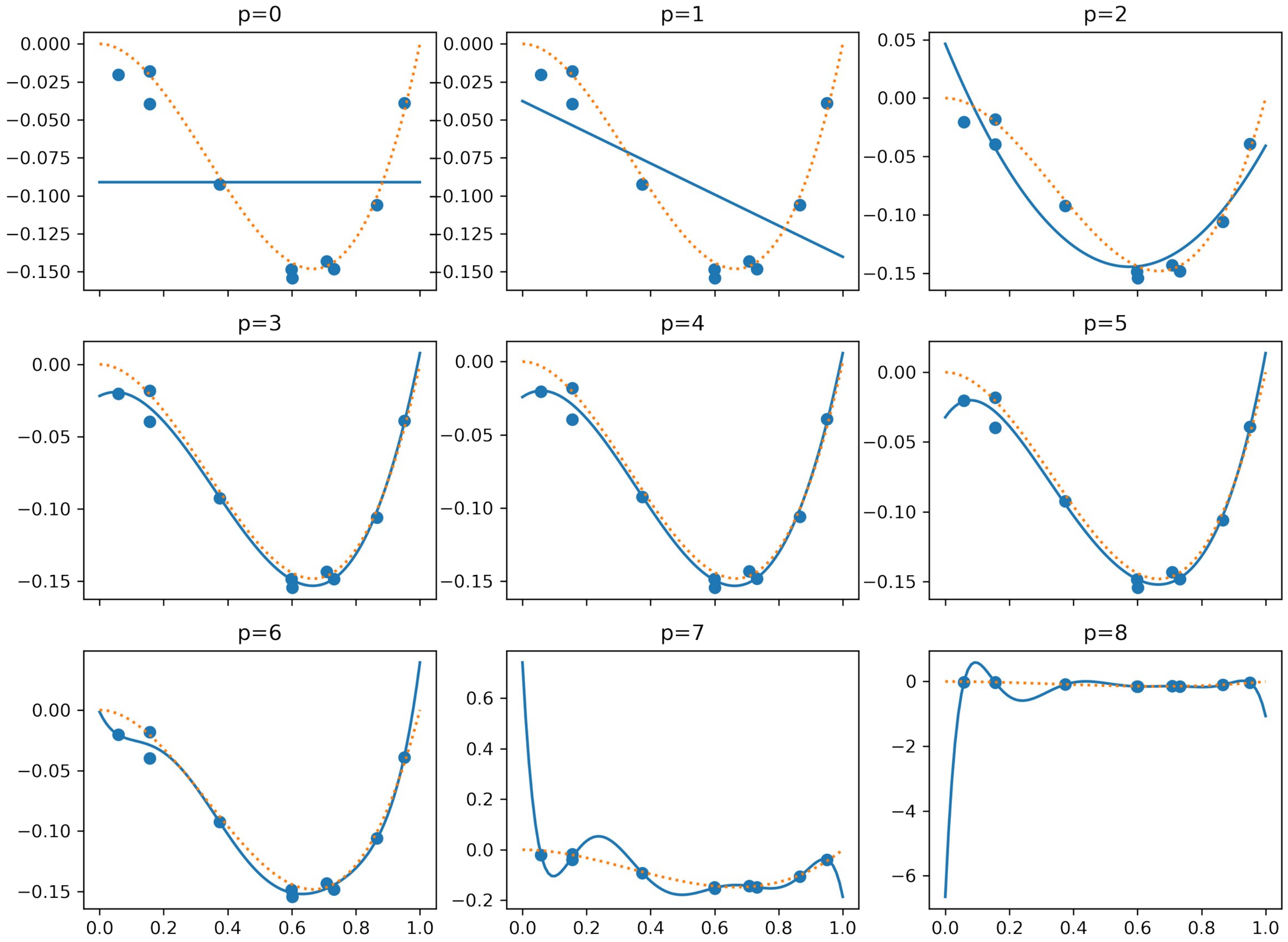
model function: $f(x) = \sum_{i=0}^{p} w_i x^i$



Polynomial regression: an example

actual function: $f(x) = x^3 - x^2$

model function: $f(x) = \sum_{i=0}^{p} w_i x^i$



Which model is the best (qualitatively)?

Interlude: Occam's Razor

With competing theories or explanations, the simpler one, for example a model with fewer parameters, is to be preferred.

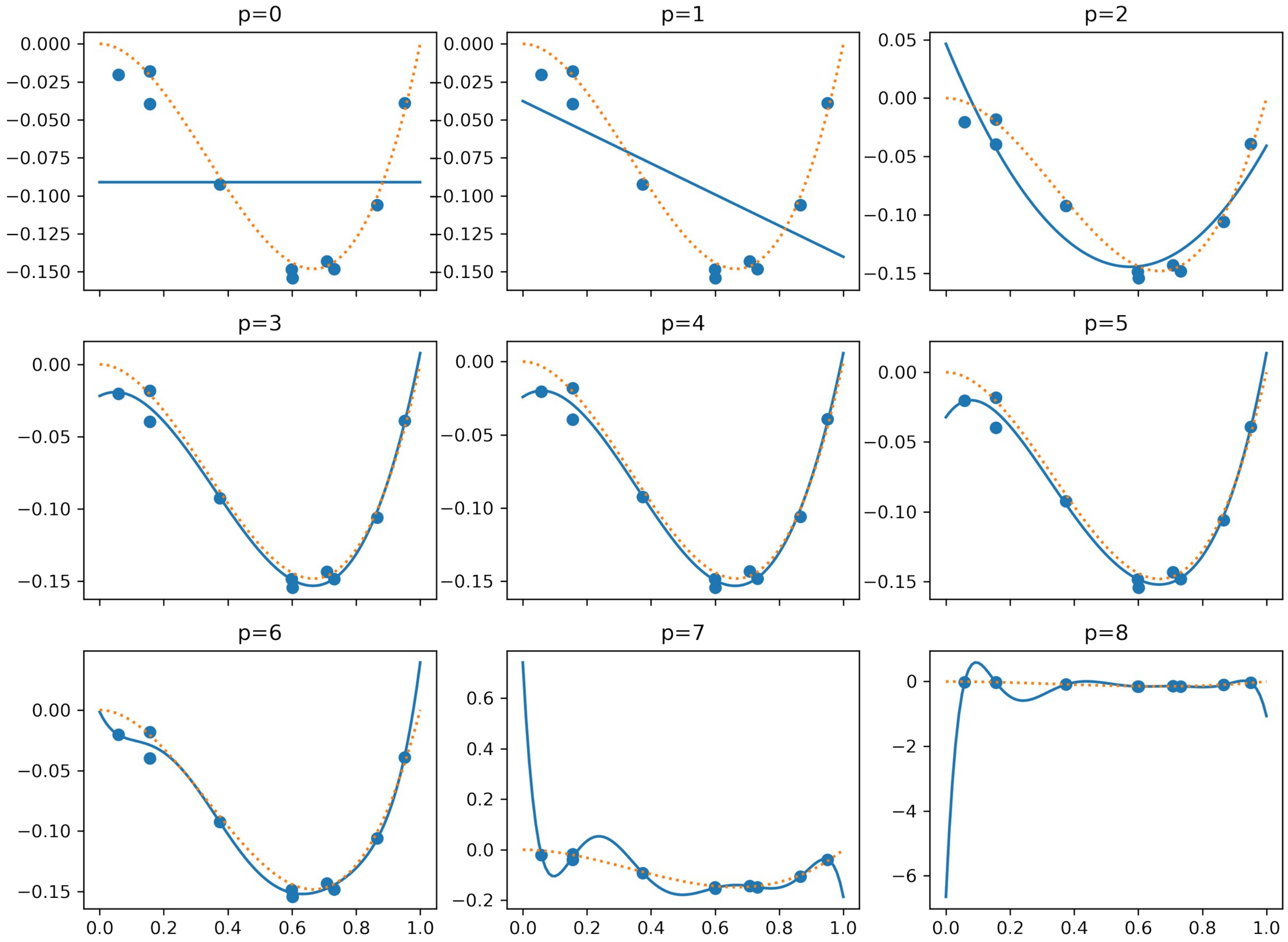
Moscarlop @ wikimedia

William of Ockham
(1287 - 1347)

Polynomial regression: an example

actual function: $f(x) = x^3 - x^2$

model function: $f(x) = \sum_{i=0}^{p} w_i x^i$



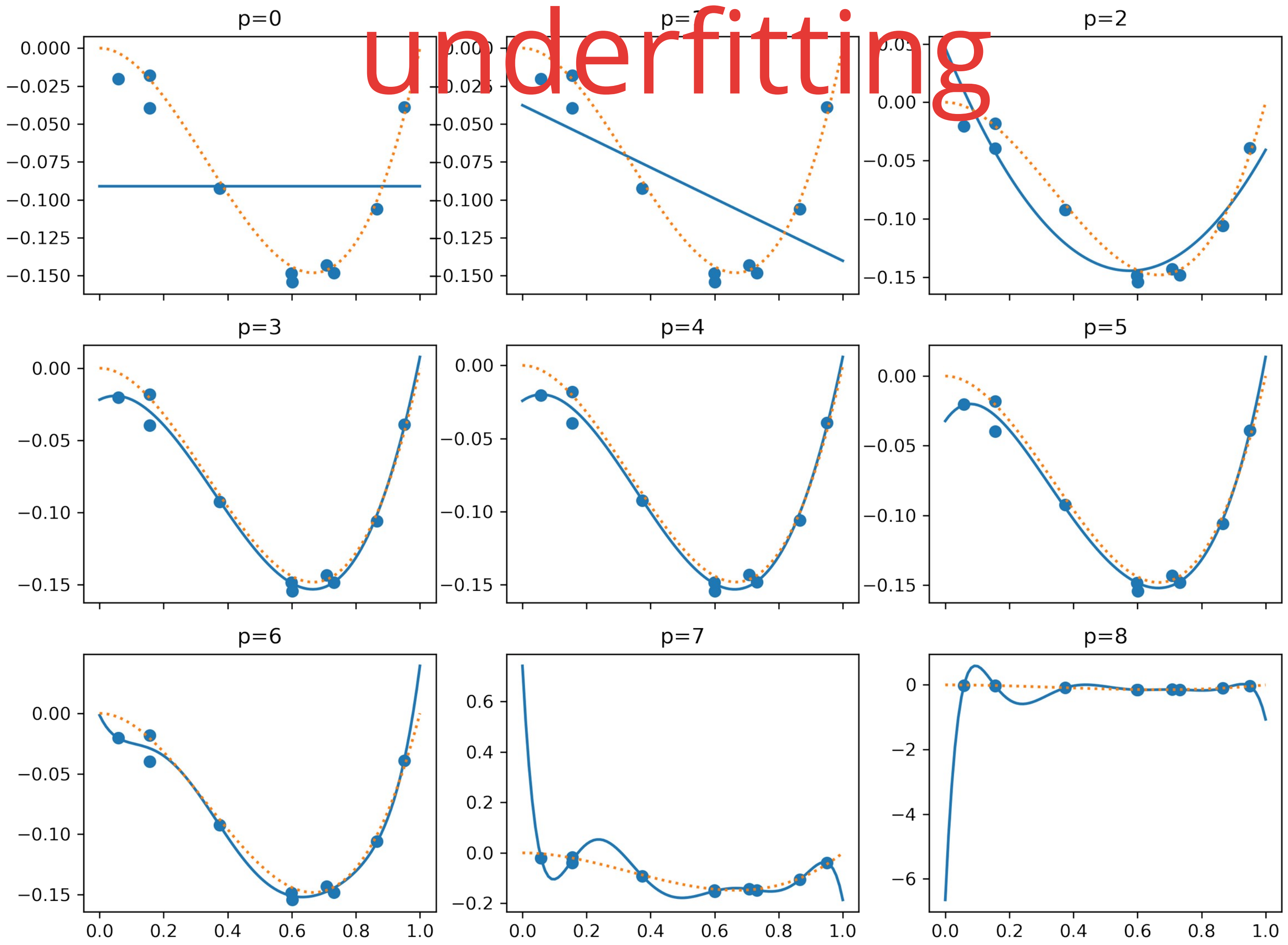
Which model is the best (qualitatively)?

Polynomial regression: an example

actual function: $f(x) = x^3 - x^2$

model function: $f(x) = \sum_{i=0}^{p} w_i x^i$

Which model is the best (qualitatively)?

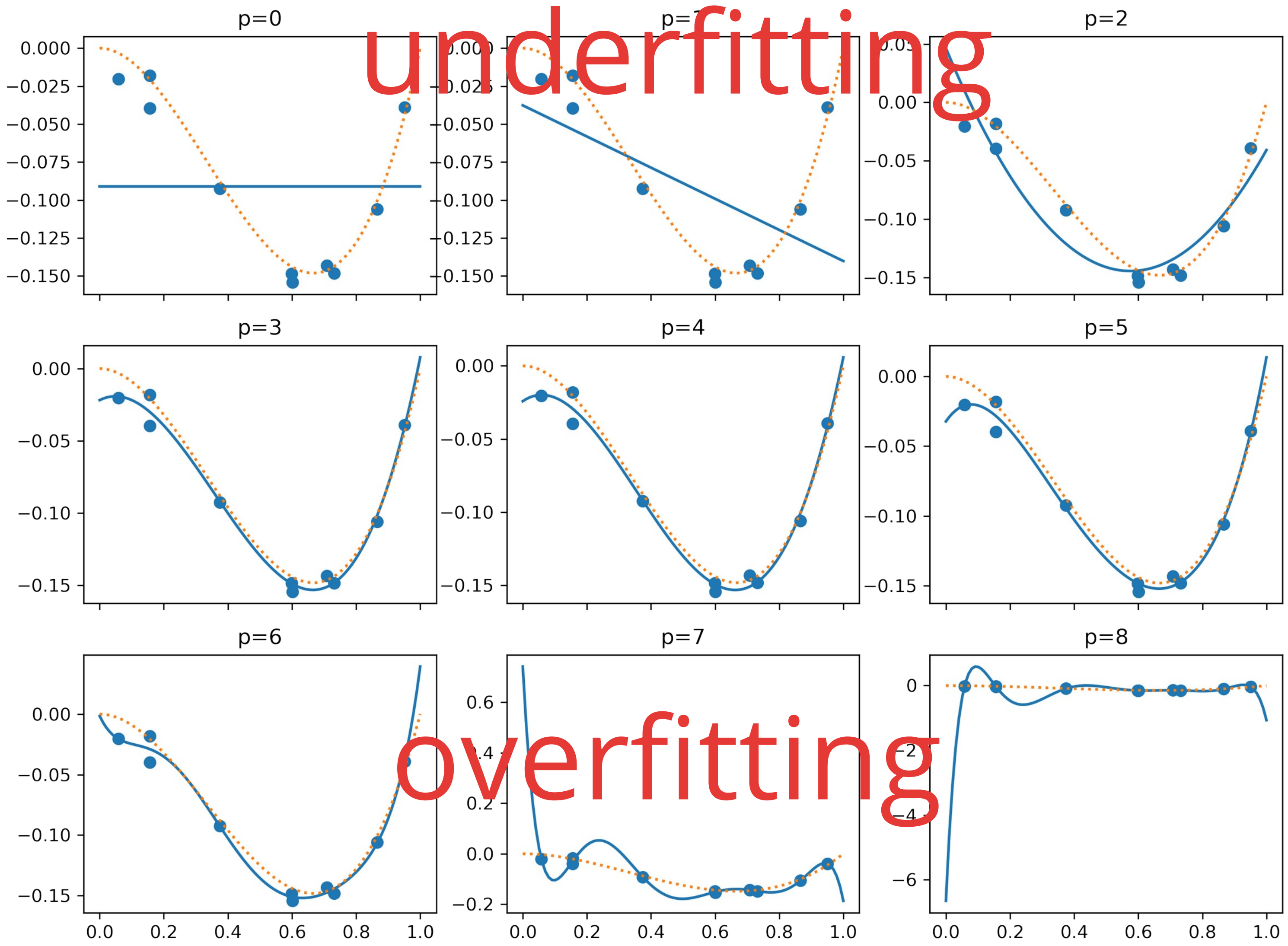


Polynomial regression: an example

actual function: $f(x) = x^3 - x^2$

model function: $f(x) = \sum_{i=0}^{p} w_i x^i$

Which model is the best (qualitatively)?



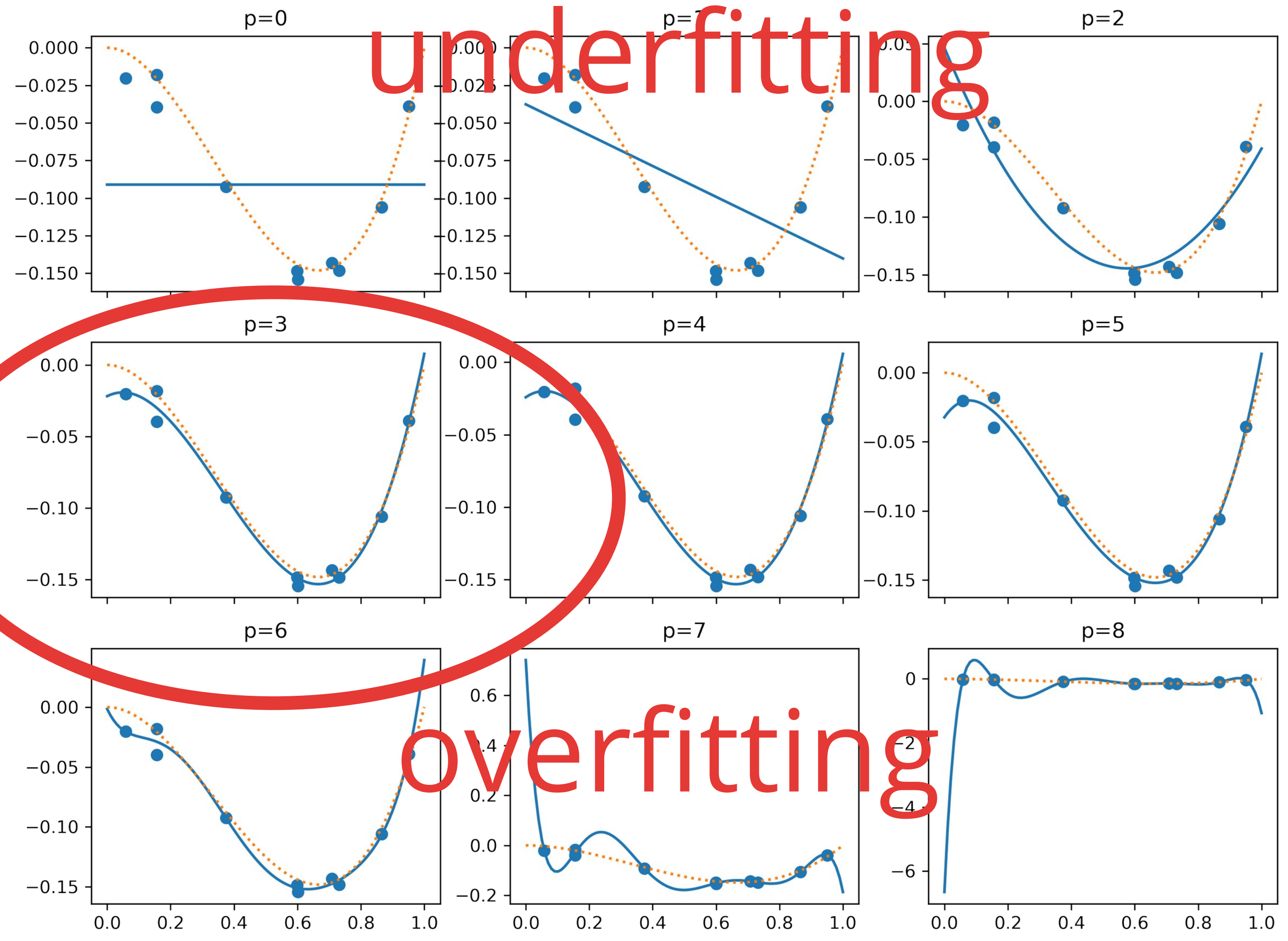
Polynomial regression: an example

actual function: $f(x) = x^3 - x^2$

model function: $f(x) = \sum_{i=0}^{p} w_i x^i$

Which model is the best (qualitatively)?

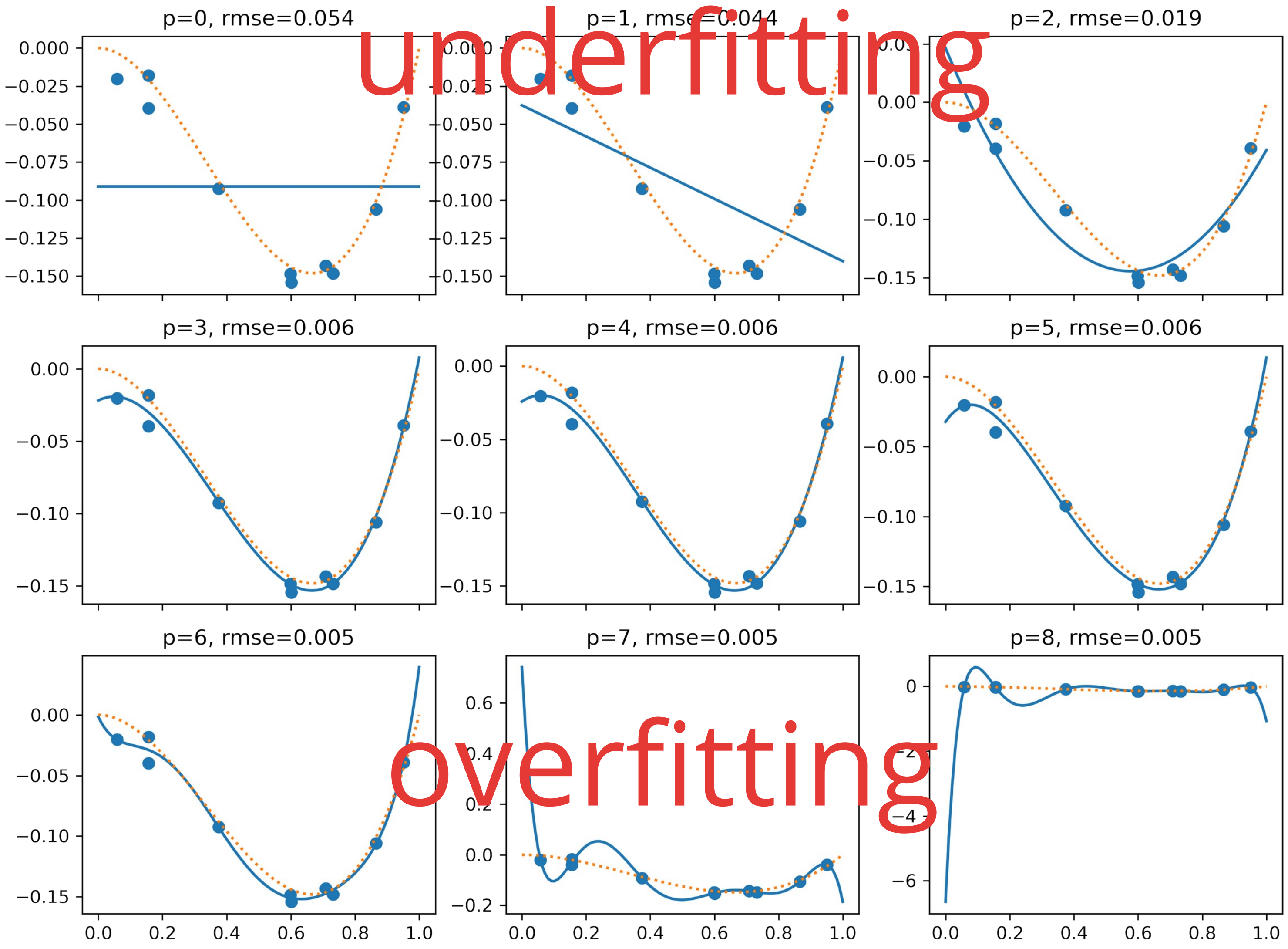
Occam's razor: $p=3$



Polynomial regression: an example

actual function: $f(x) = x^3 - x^2$

model function: $f(x) = \sum_{i=0}^{p} w_i x^i$

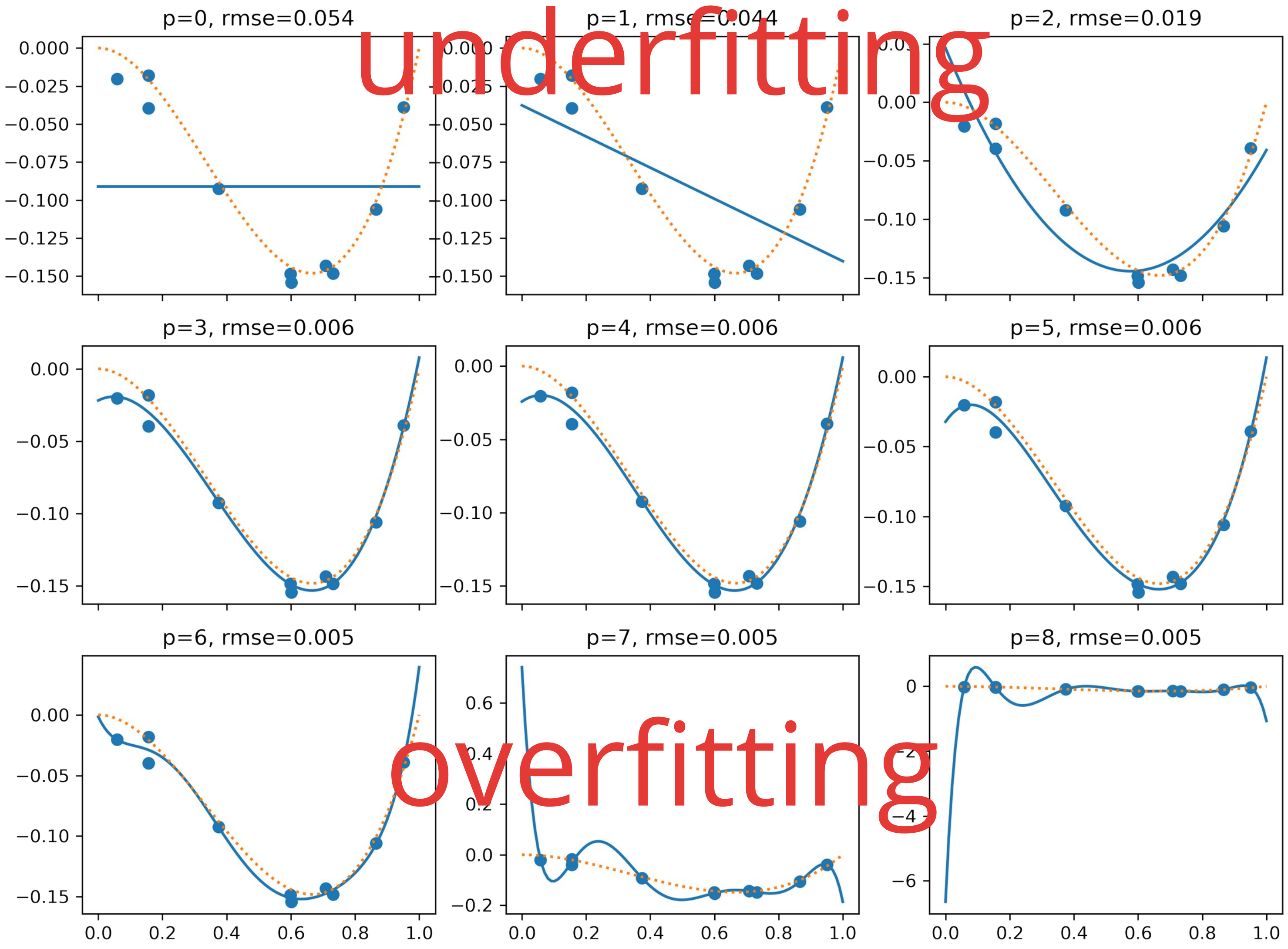


Which model is the best (quantitatively)?

Polynomial regression: an example

actual function: $f(x) = x^3 - x^2$

model function: $f(x) = \sum_{i=0}^{p} w_i x^i$



Which model is the best (quantitatively)?

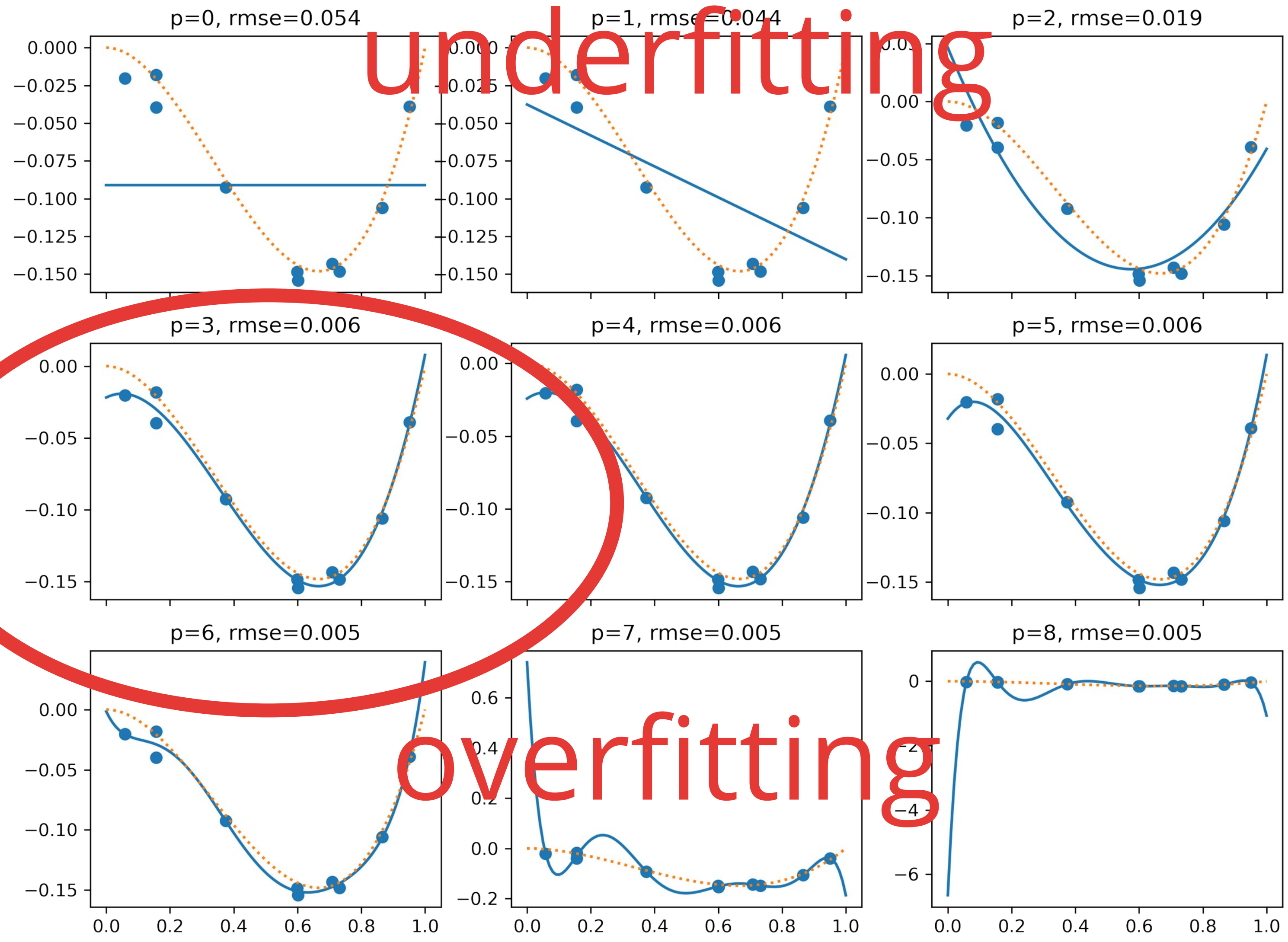
p=3-8?

p=6-8 show signs of overfitting; we **minimize capacity to regularize the model**.

Polynomial regression: an example

actual function: $f(x) = x^3 - x^2$

model function: $f(x) = \sum_{i=0}^{p} w_i x^i$



Which model is the best (quantitatively)?

p=3-8?

p=6-8 show signs of overfitting; we **minimize capacity to regularize the model**.

More regularization

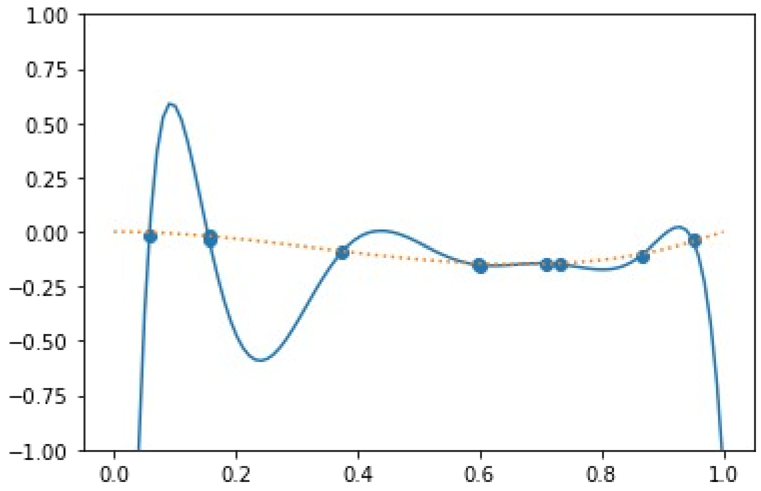
More regularization

The goal of any regression model is to minimize the loss over the data, i.e., the prediction errors.

More regularization

The goal of any regression model is to minimize the loss over the data, i.e., the prediction errors.

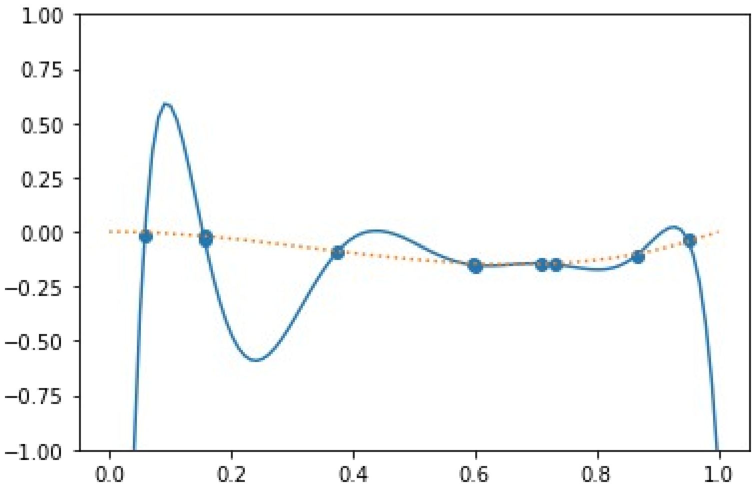
Therefore, this would be a perfectly valid result for the model:



More regularization

The goal of any regression model is to minimize the loss over the data, i.e., the prediction errors.

Therefore, this would be a perfectly valid result for the model:



The problem is that the model tries to come as close as possible to the training data to minimize the loss, inevitably **overfitting** and overshooting. We need a way to prevent the model from performing too well on the training data.

L2 regularization: Ridge regression

One way to prevent the model from “doing too well” is to **regularize the loss** based on the learned weights:

$$L'(\boldsymbol{x}, \boldsymbol{w}) = \frac{1}{N} \sum_{i}^{N} L_i(f(\boldsymbol{x}_i, \boldsymbol{w}), \boldsymbol{y}_i) + \alpha \|\boldsymbol{w}\|_2^2$$

L2 regularization: Ridge regression

One way to prevent the model from “doing too well” is to **regularize the loss** based on the learned weights:

$$L'(\boldsymbol{x},\boldsymbol{w})=\underbrace{\frac{1}{N}\sum_{i}^{N}L_i(f(\boldsymbol{x_i},\boldsymbol{w}),\boldsymbol{y_i})}_{\text{Mean loss over } N \text{ samples}}+\alpha\|\boldsymbol{w}\|_2^2$$

Mean loss over
N samples

L2 regularization: Ridge regression

One way to prevent the model from “doing too well” is to **regularize the loss** based on the learned weights:

$$L'(\boldsymbol{x},\boldsymbol{w}) = \underbrace{\frac{1}{N}\sum_{i}^{N} L_i(f(\boldsymbol{x_i},\boldsymbol{w}),\boldsymbol{y_i})}_{\text{Mean loss over } N \text{ samples}} + \underbrace{\alpha \|\boldsymbol{w}\|_2^2}_{\text{Regularization term}}$$

α : regularization parameter

$\|\boldsymbol{w}\|_2^2 = \boldsymbol{w} * \boldsymbol{w}$: L2-norm

L2 regularization: Ridge regression

One way to prevent the model from “doing too well” is to **regularize the loss** based on the learned weights:

$$L'(\mathbf{x},\mathbf{w}) = \underbrace{\frac{1}{N}\sum_{i}^{N} L_i(f(\mathbf{x}_i,\mathbf{w}),\mathbf{y}_i)}_{\text{Mean loss over } N \text{ samples}} + \underbrace{\alpha\|\mathbf{w}\|_2^2}_{\text{Regularization term}}$$

α : regularization parameter

$\|\mathbf{w}\|_2^2 = \mathbf{w} * \mathbf{w}$: L2-norm

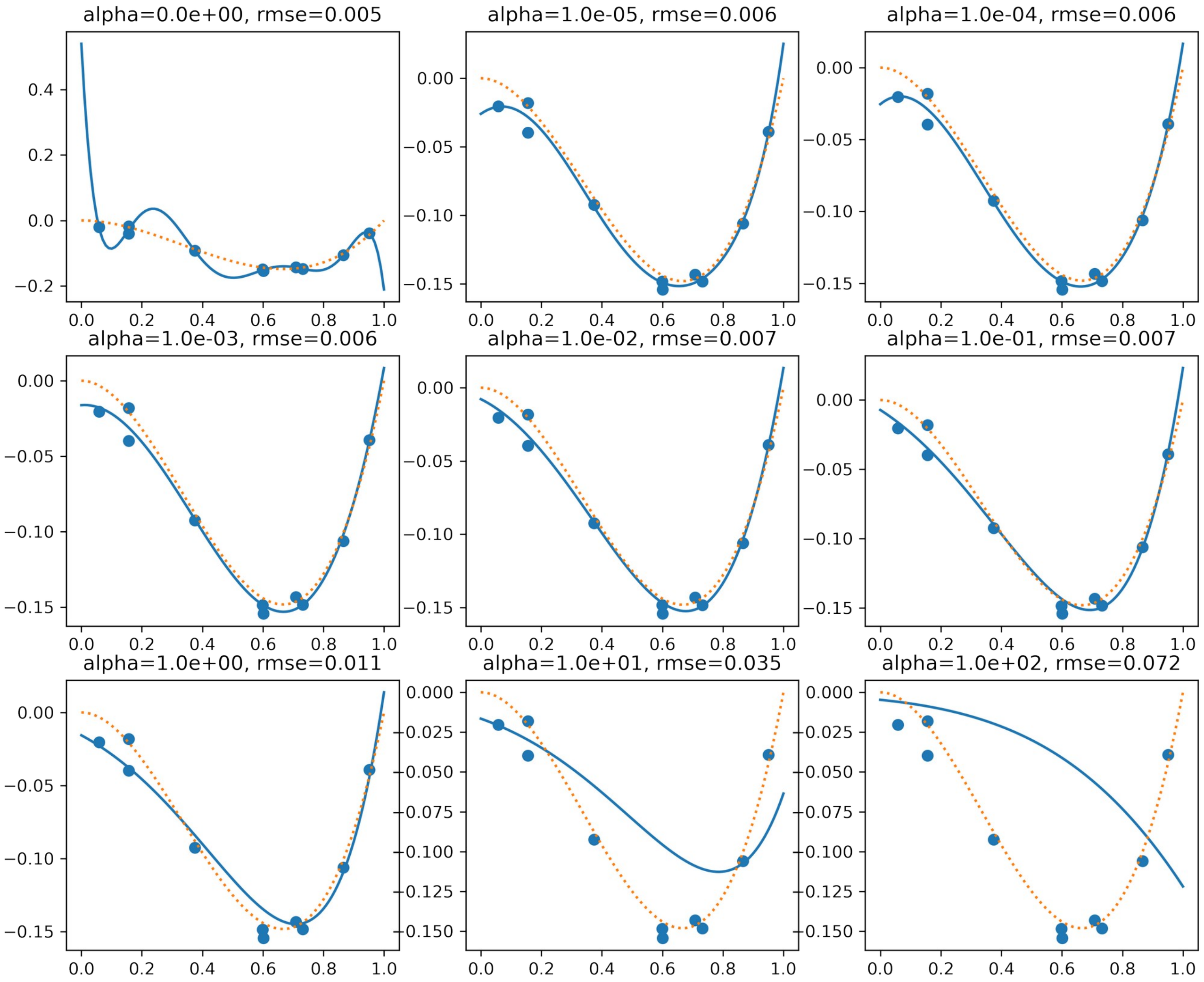
Let’s see what L2 regularization does…

L2 regularization: Ridge regression

Polynomial model with $p=8$

→ prone to overfitting

$$L'(\boldsymbol{x},\boldsymbol{w})=\frac{1}{N}\sum_i^N L_i(f(\boldsymbol{x_i},\boldsymbol{w}),\boldsymbol{y_i})+\alpha\|\boldsymbol{w}\|_2^2$$

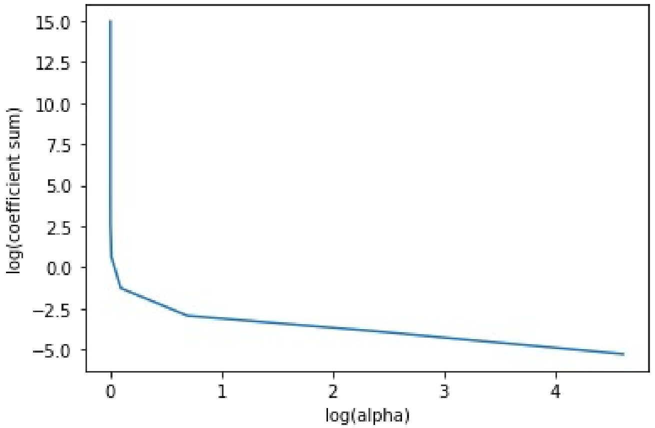


L2 regularization: Ridge regression

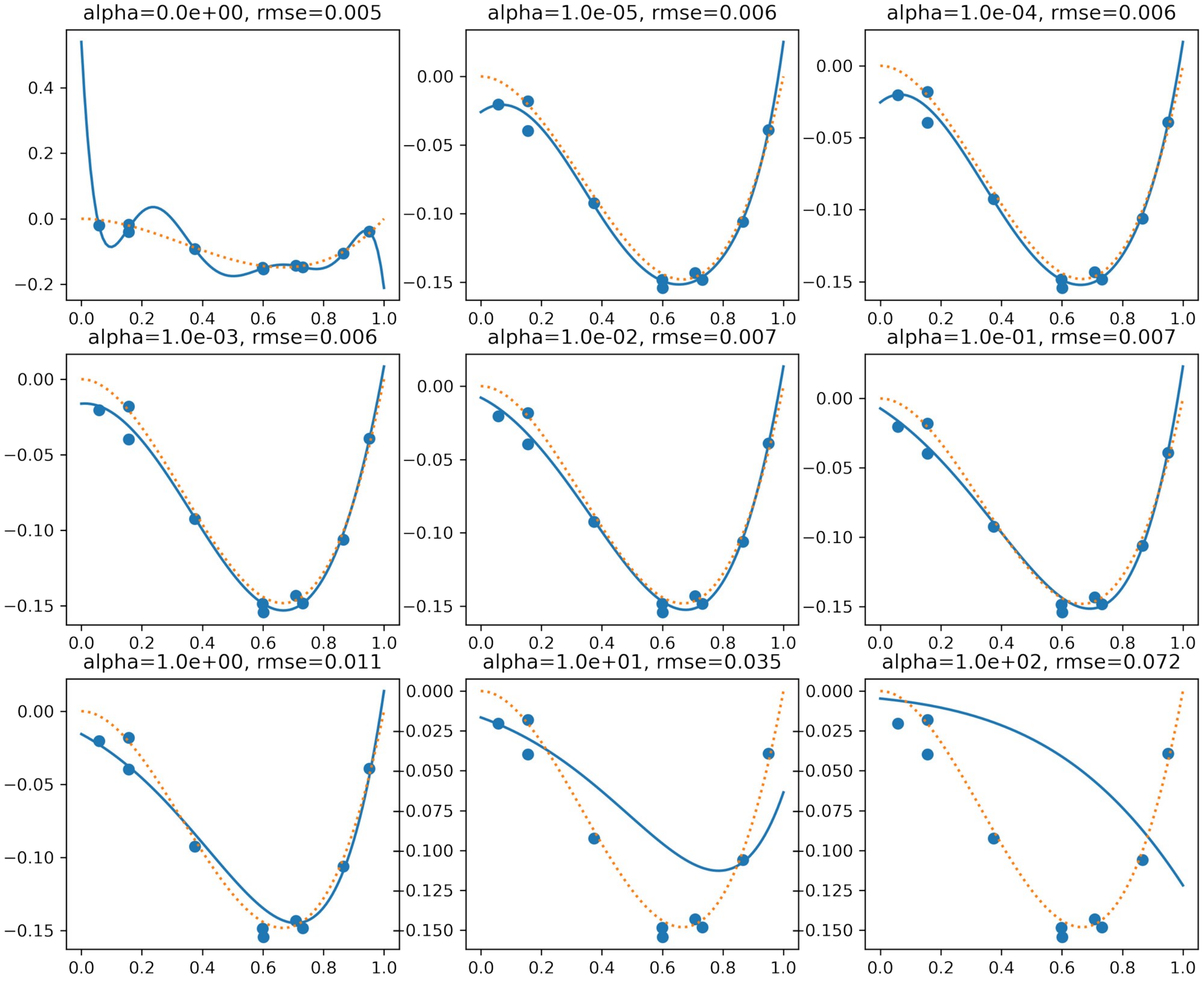
Polynomial model with $p=8$

→ prone to overfitting

$$L'(\boldsymbol{x},\boldsymbol{w})=\frac{1}{N}\sum_{i}^{N} L_i(f(\boldsymbol{x}_i,\boldsymbol{w}),\boldsymbol{y}_i)+\alpha\|\boldsymbol{w}\|_2^2$$



With increasing alpha, all coefficients w_i drop in magnitude, leading to smoother fits → **regularization**



L1 regularization: LASSO regression

We can use a different regularization term:

“LASSO”: least absolute shrinkage and selection operator

$$L'(\boldsymbol{x},\boldsymbol{w})=\frac{1}{N}\sum_{i}^{N}L_i(f(\boldsymbol{x}_i,\boldsymbol{w}),\boldsymbol{y}_i)+\alpha\|\boldsymbol{w}\|_1$$

L1 regularization: LASSO regression

We can use a different regularization term:

"LASSO": least absolute shrinkage and selection operator

$$L'(\boldsymbol{x},\boldsymbol{w}) = \underbrace{\frac{1}{N}\sum_{i}^{N} L_i(f(\boldsymbol{x_i},\boldsymbol{w}),\boldsymbol{y_i})}_{\text{Mean loss over } N \text{ samples}} + \alpha \|\boldsymbol{w}\|_1$$

L1 regularization: LASSO regression

We can use a different regularization term:

"LASSO": least absolute shrinkage and selection operator

$$L'(\boldsymbol{x},\boldsymbol{w}) = \underbrace{\frac{1}{N}\sum_i^N L_i(f(\boldsymbol{x}_i,\boldsymbol{w}),\boldsymbol{y}_i)}_{\text{Mean loss over } N \text{ samples}} + \underbrace{\alpha\|\boldsymbol{w}\|_1}_{\text{Regularization term}}$$

α : regularization parameter

$\|\boldsymbol{w}\|_1 = |\boldsymbol{w}|$: L1-norm

L1 regularization: LASSO regression

We can use a different regularization term:

"LASSO": least absolute shrinkage and selection operator

$$L'(\boldsymbol{x},\boldsymbol{w})=\underbrace{\frac{1}{N}\sum_{i}^{N}L_i(f(\boldsymbol{x}_i,\boldsymbol{w}),\boldsymbol{y}_i)}_{\text{Mean loss over } N \text{ samples}}+\underbrace{\alpha\|\boldsymbol{w}\|_1}_{\text{Regularization term}}$$

α : regularization parameter

$\|\boldsymbol{w}\|_1=|\boldsymbol{w}|$: L1-norm

Let's see what L1 regularization does...

L1 regularization: LASSO regression

Polynomial model with $p=8$

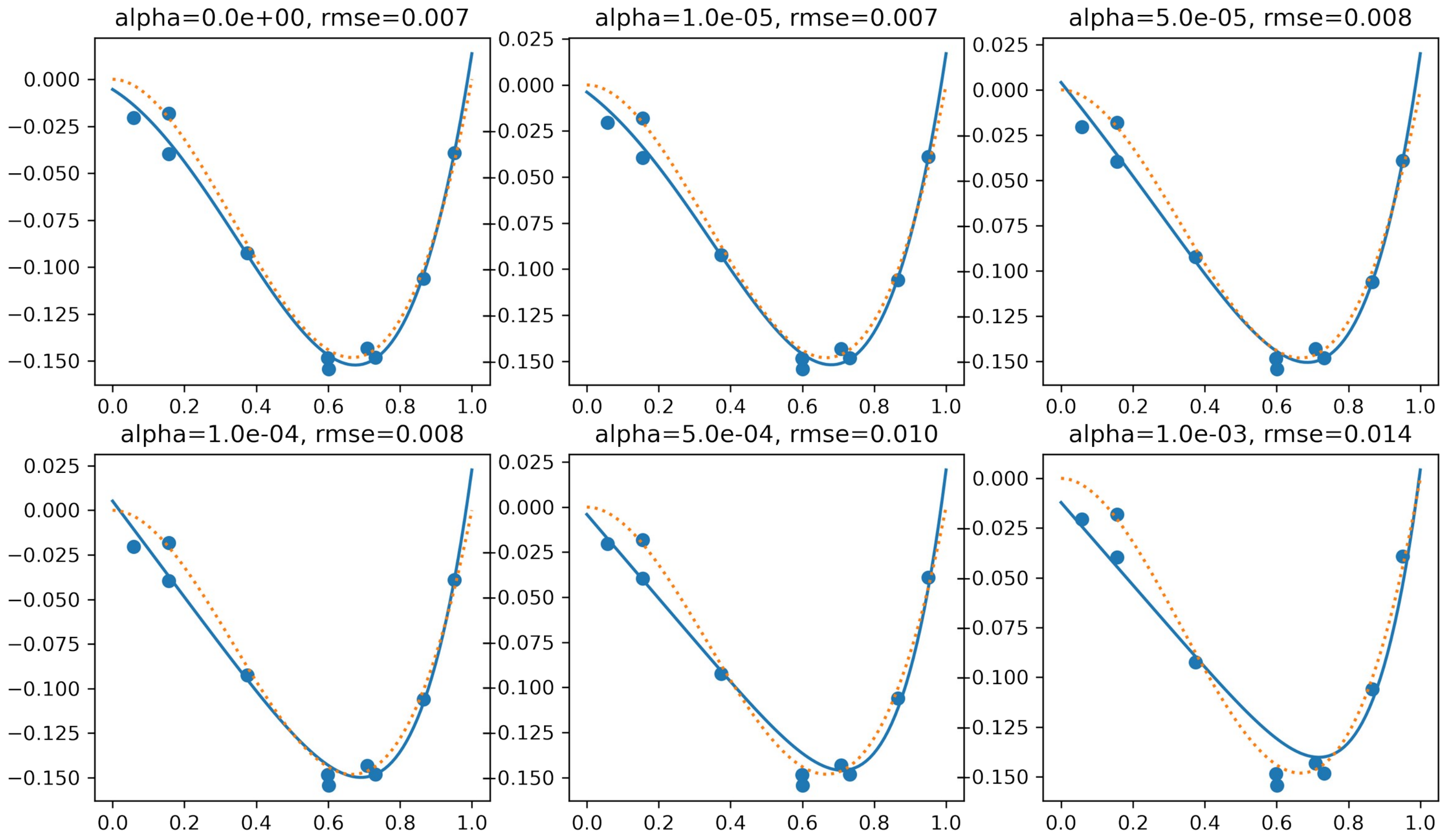
→ prone to overfitting

$$L'(\boldsymbol{x},\boldsymbol{w})=\frac{1}{N}\sum_i^N L_i(f(\boldsymbol{x_i},\boldsymbol{w}),\boldsymbol{y_i})+\alpha\|\boldsymbol{w}\|_1$$

The effects on the model seem to be similar with one significant difference:

While L2 regularization modulates all coefficients wi in the same way, L1 regularization aims to set less meaningful coefficients to zero.

L1 regularization performs **feature selection** (in this case: poly. order) .



L1 regularization: LASSO regression

Polynomial model with $p=8$

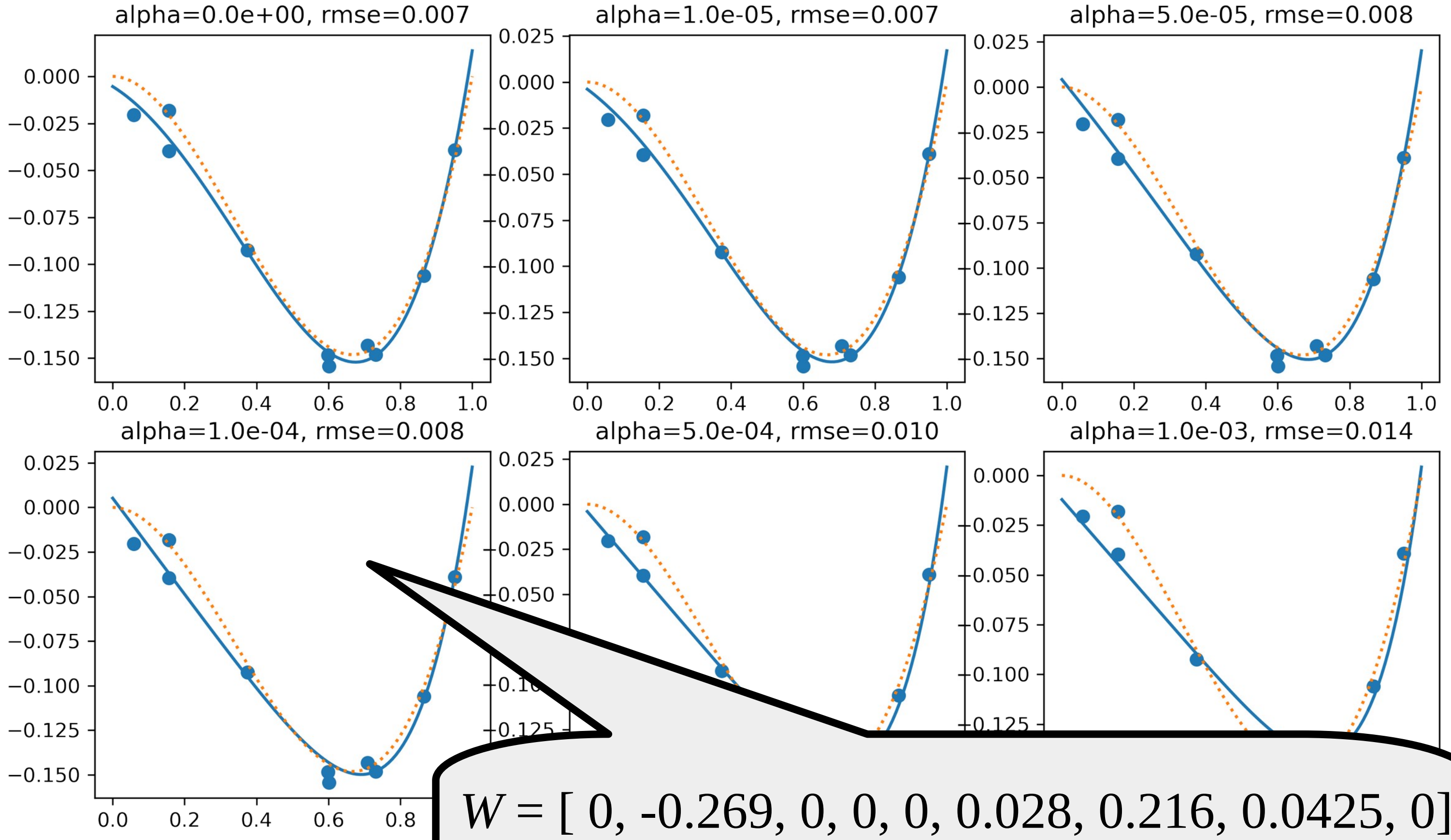
→ prone to overfitting

$$L'(\boldsymbol{x},\boldsymbol{w})=\frac{1}{N}\sum_{i}^{N}L_i(f(\boldsymbol{x_i},\boldsymbol{w}),\boldsymbol{y_i})+\alpha\|\boldsymbol{w}\|_1$$

The effects on the model seem to be similar with one significant difference:

While L2 regularization modulates all coefficients wi in the same way, L1 regularization aims to set less meaningful coefficients to zero.

L1 regularization performs **feature selection** (in this case: poly. order) .



W = [0, -0.269, 0, 0, 0, 0.028, 0.216, 0.0425, 0]

L1 regularization: LASSO regression

Polynomial model with $p=8$

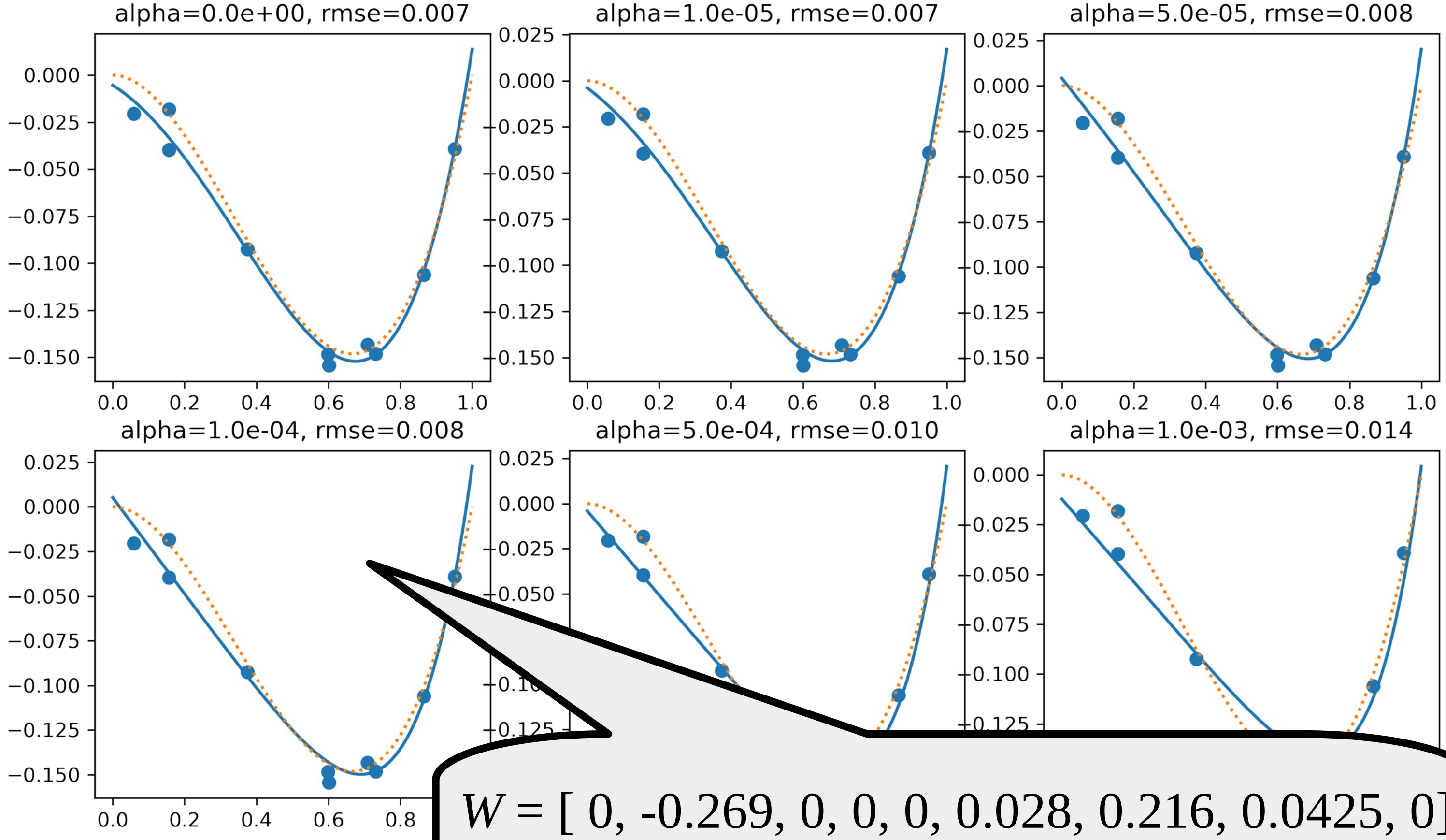
→ prone to overfitting

$$L'(\mathbf{x},\mathbf{w})=\frac{1}{N}\sum_{i}^{N} L_i(f(\mathbf{x}_i,\mathbf{w}),\mathbf{y}_i)+\alpha\|\mathbf{w}\|_1$$

The effects on the model seem to be similar with one significant difference:

While L2 regularization modulates all coefficients wi in the same way, L1 regularization aims to set less meaningful coefficients to zero.

L1 regularization performs **feature selection** (in this case: poly. order) .



W = [0, -0.269, 0, 0, 0, 0.028, 0.216, 0.0425, 0]

$$f(x)=-0.269\,x+0.028\,x^5+0.216\,x^6+0.0425\,x^7$$

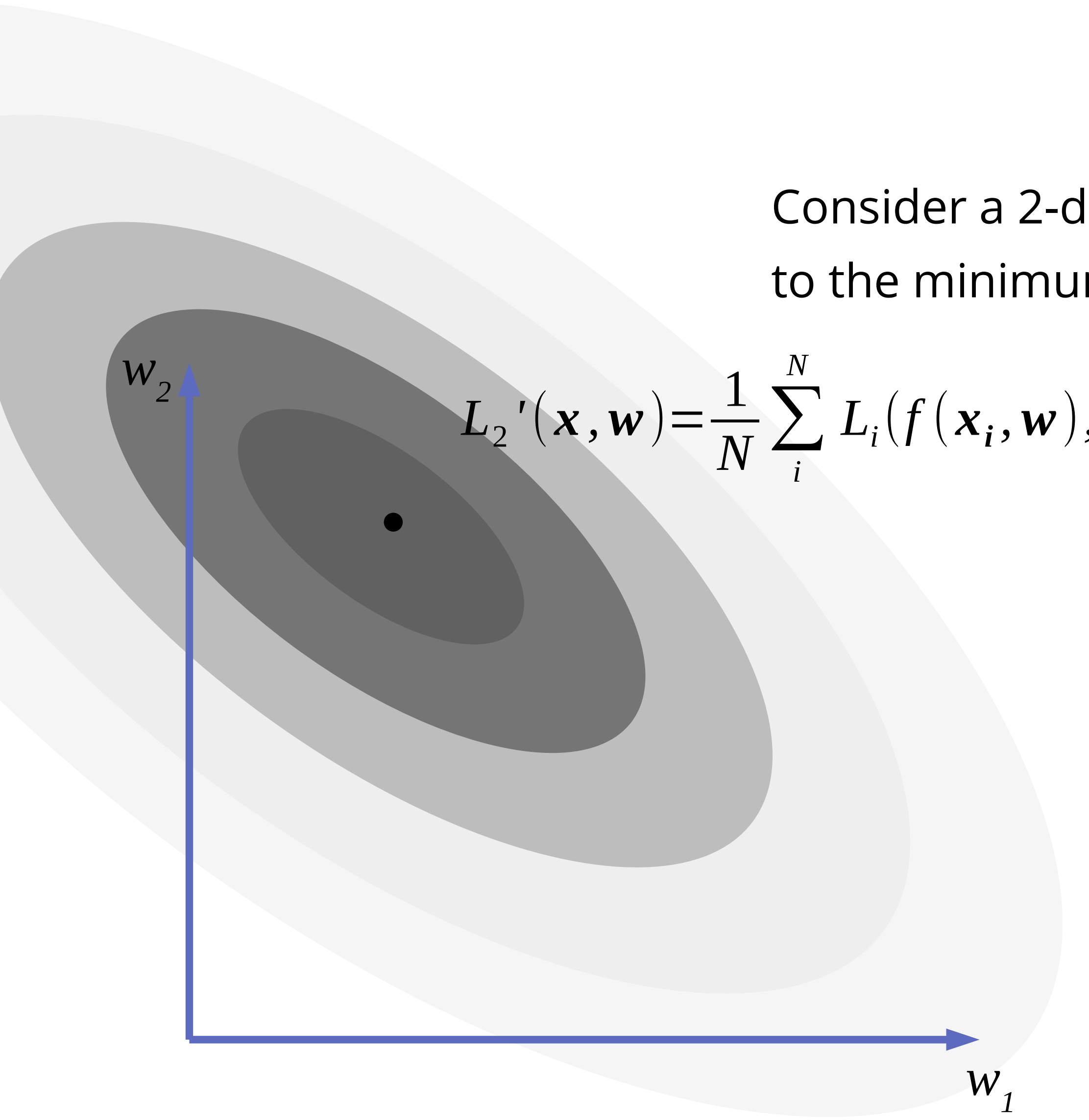
L2 vs. L1 regularization

Consider a 2-d loss space, spanned by w_1 and w_2. How can we get closest to the minimum loss with the two different regularization terms?

L2 vs. L1 regularization

Consider a 2-d loss space, spanned by w_1 and w_2. How can we get closest to the minimum loss with the two different regularization terms?

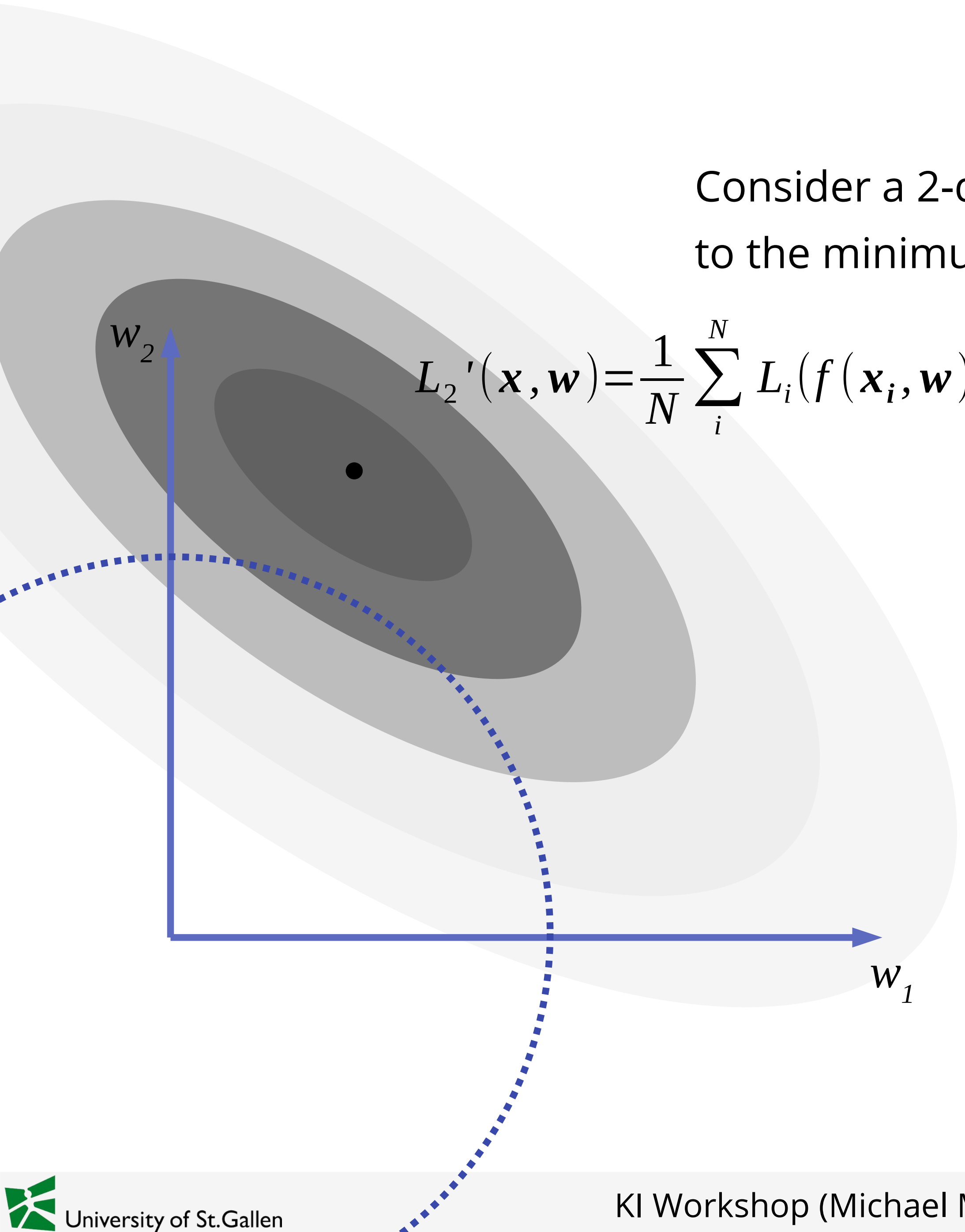
$$L_2'(\mathbf{x},\mathbf{w})=\frac{1}{N}\sum_i^N L_i(f(\mathbf{x_i},\mathbf{w}),\mathbf{y_i})+\alpha\|\mathbf{w}\|_2^2$$



L2 vs. L1 regularization

Consider a 2-d loss space, spanned by w_1 and w_2. How can we get closest to the minimum loss with the two different regularization terms?

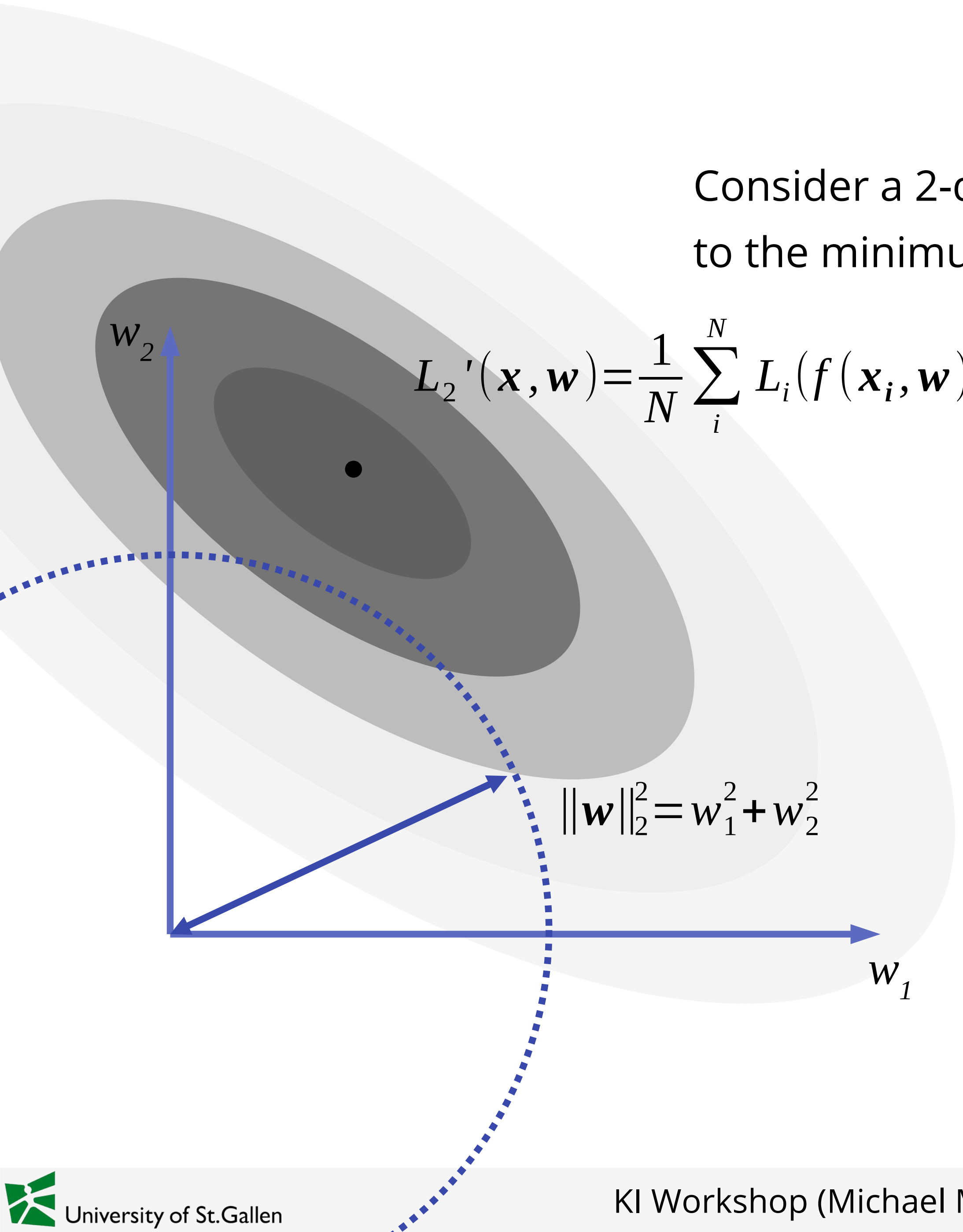
$$L_2'(\boldsymbol{x},\boldsymbol{w}) = \frac{1}{N}\sum_{i}^{N} L_i(f(\boldsymbol{x_i},\boldsymbol{w}),\boldsymbol{y_i}) + \alpha \|\boldsymbol{w}\|_2^2$$



L2 vs. L1 regularization

Consider a 2-d loss space, spanned by w_1 and w_2. How can we get closest to the minimum loss with the two different regularization terms?

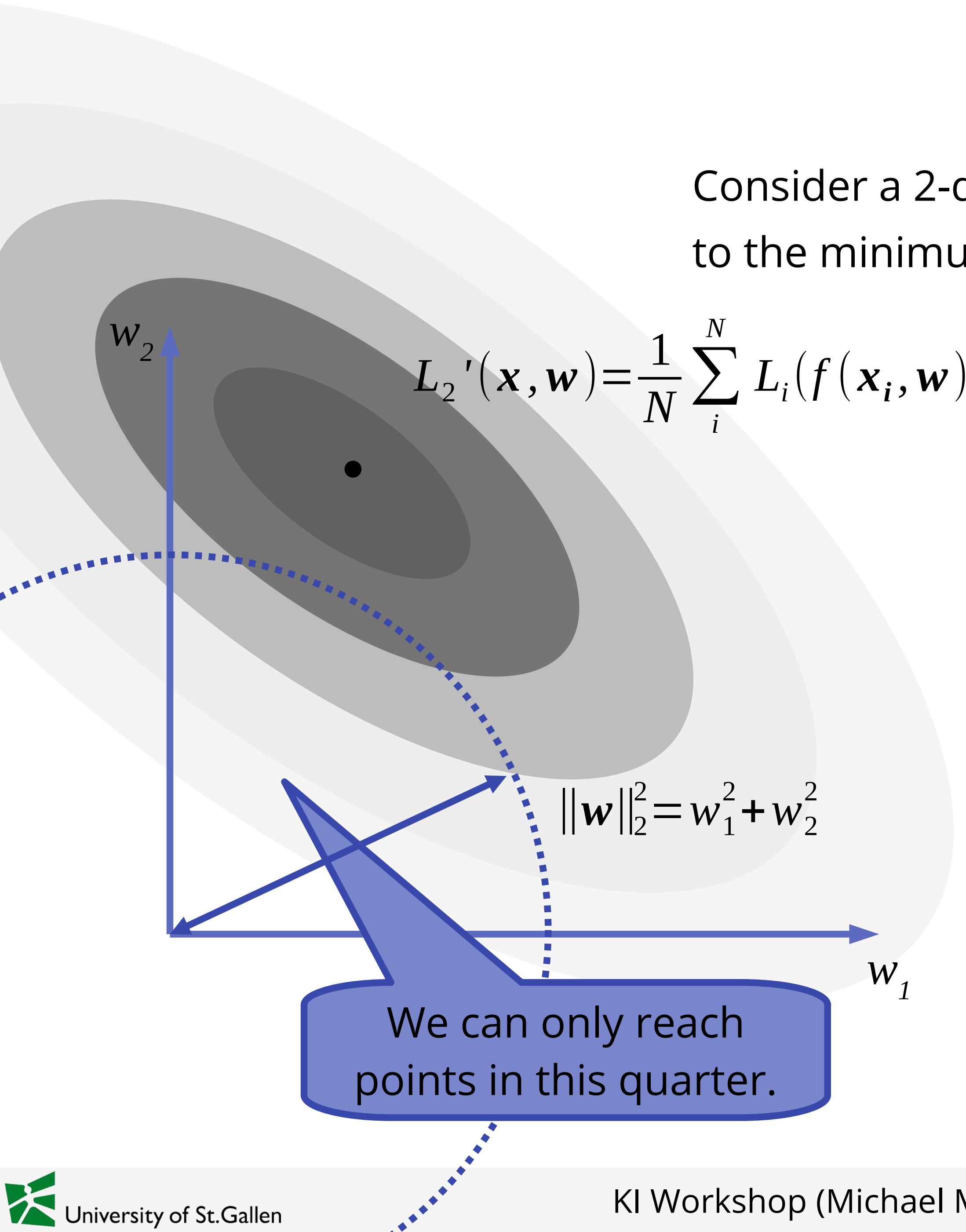
$$L_2'(\mathbf{x},\mathbf{w})=\frac{1}{N}\sum_i^N L_i(f(\mathbf{x}_i,\mathbf{w}),\mathbf{y}_i)+\alpha\|\mathbf{w}\|_2^2$$



L2 vs. L1 regularization

Consider a 2-d loss space, spanned by w_1 and w_2. How can we get closest to the minimum loss with the two different regularization terms?

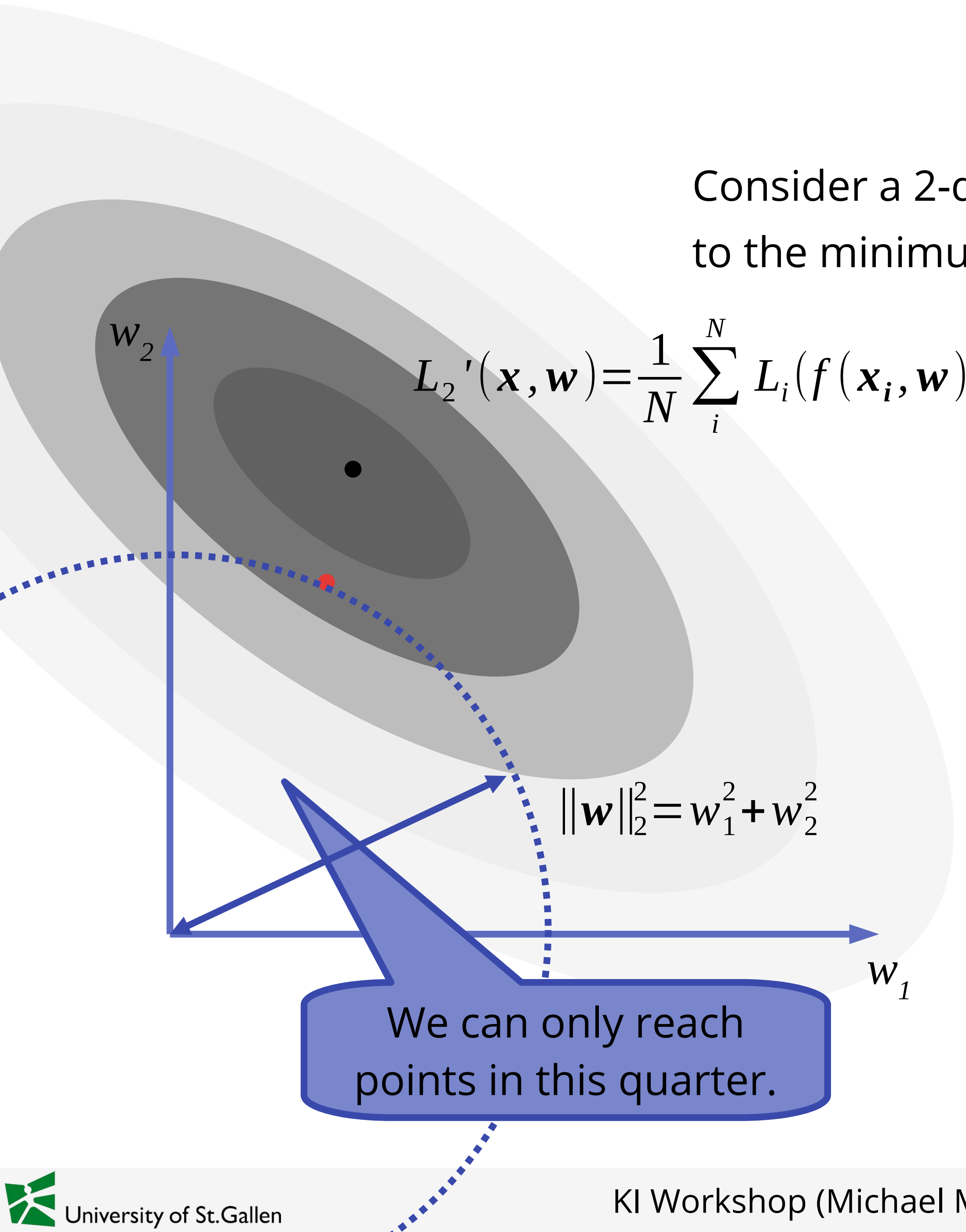
$$L_2'(\boldsymbol{x}, \boldsymbol{w}) = \frac{1}{N}\sum_i^N L_i(f(\boldsymbol{x_i}, \boldsymbol{w}), \boldsymbol{y_i}) + \alpha\|\boldsymbol{w}\|_2^2$$



L2 vs. L1 regularization

Consider a 2-d loss space, spanned by w_1 and w_2. How can we get closest to the minimum loss with the two different regularization terms?

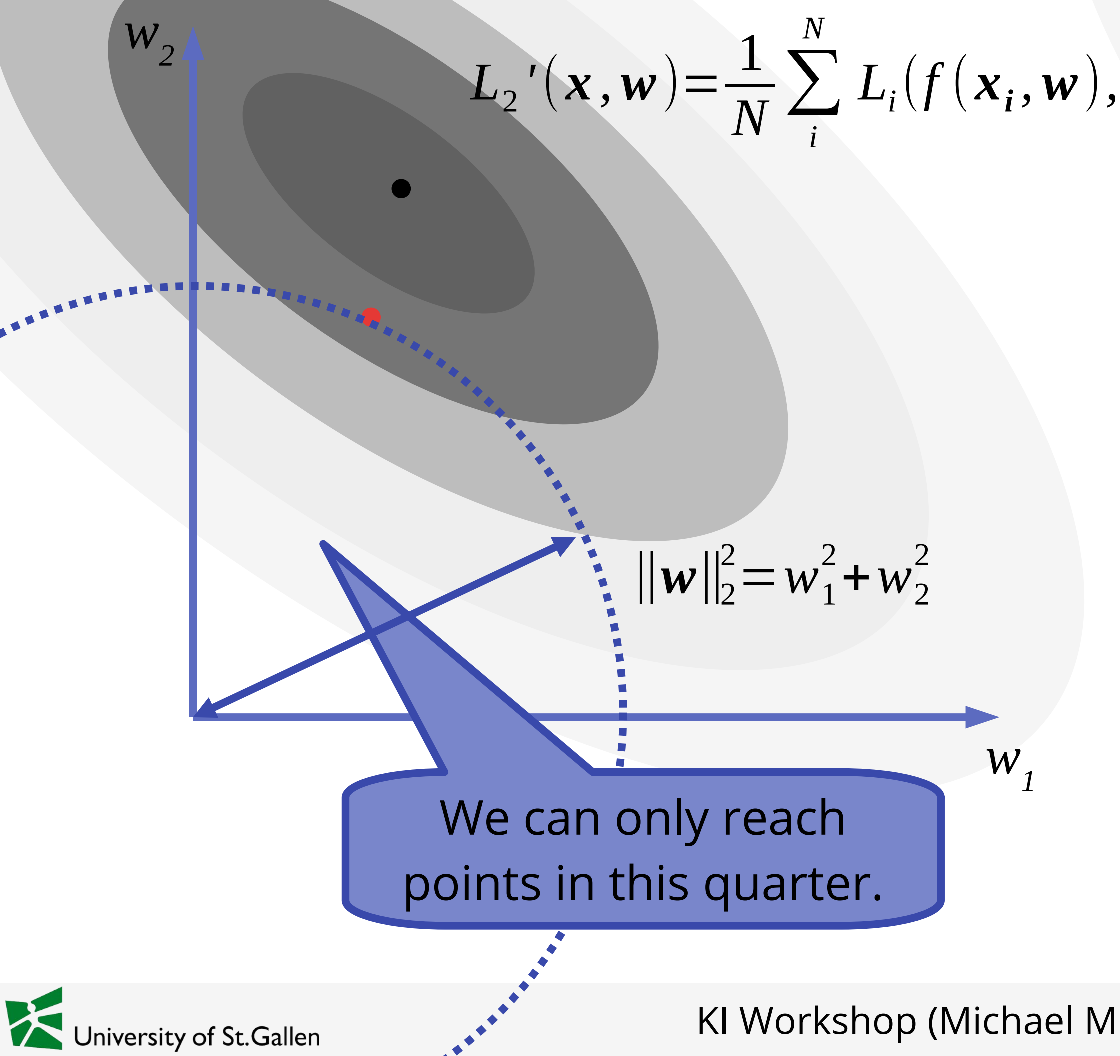
$$L_2'(\boldsymbol{x},\boldsymbol{w})=\frac{1}{N}\sum_i^N L_i(f(\boldsymbol{x_i},\boldsymbol{w}),\boldsymbol{y_i})+\alpha\|\boldsymbol{w}\|_2^2$$



L2 vs. L1 regularization

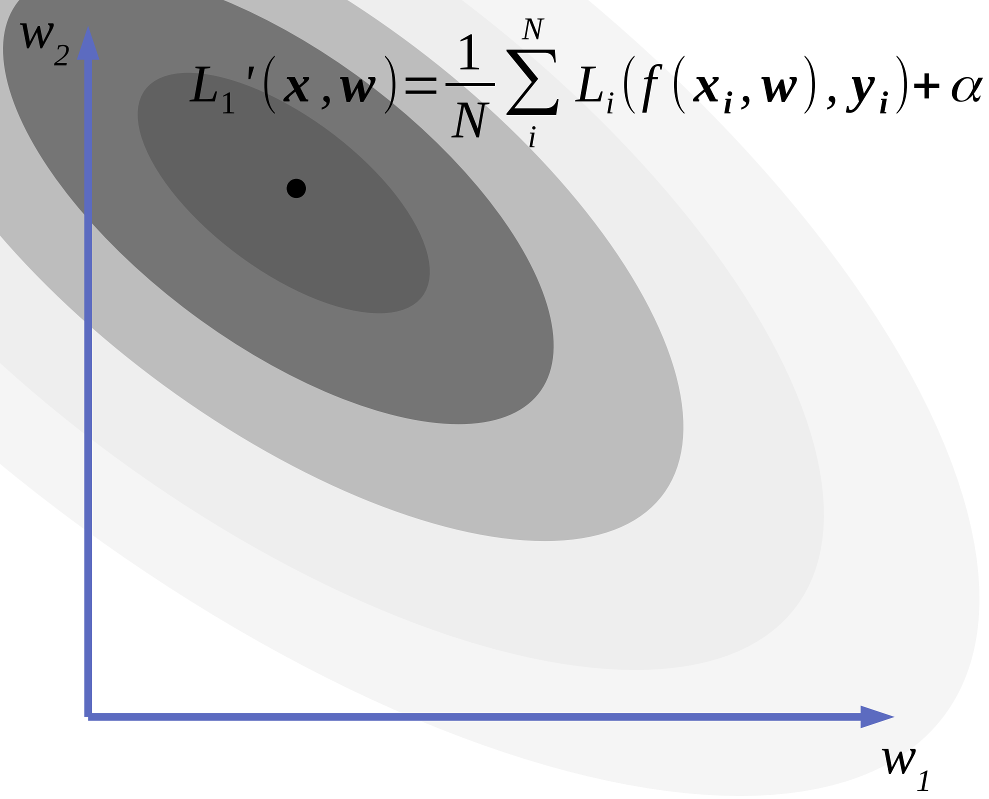
Consider a 2-d loss space, spanned by w_1 and w_2. How can we get closest to the minimum loss with the two different regularization terms?

$$L_2'(\mathbf{x},\mathbf{w})=\frac{1}{N}\sum_{i}^{N} L_i(f(\mathbf{x}_i,\mathbf{w}),\mathbf{y}_i)+\alpha\|\mathbf{w}\|_2^2$$



$$\|\mathbf{w}\|_2^2=w_1^2+w_2^2$$

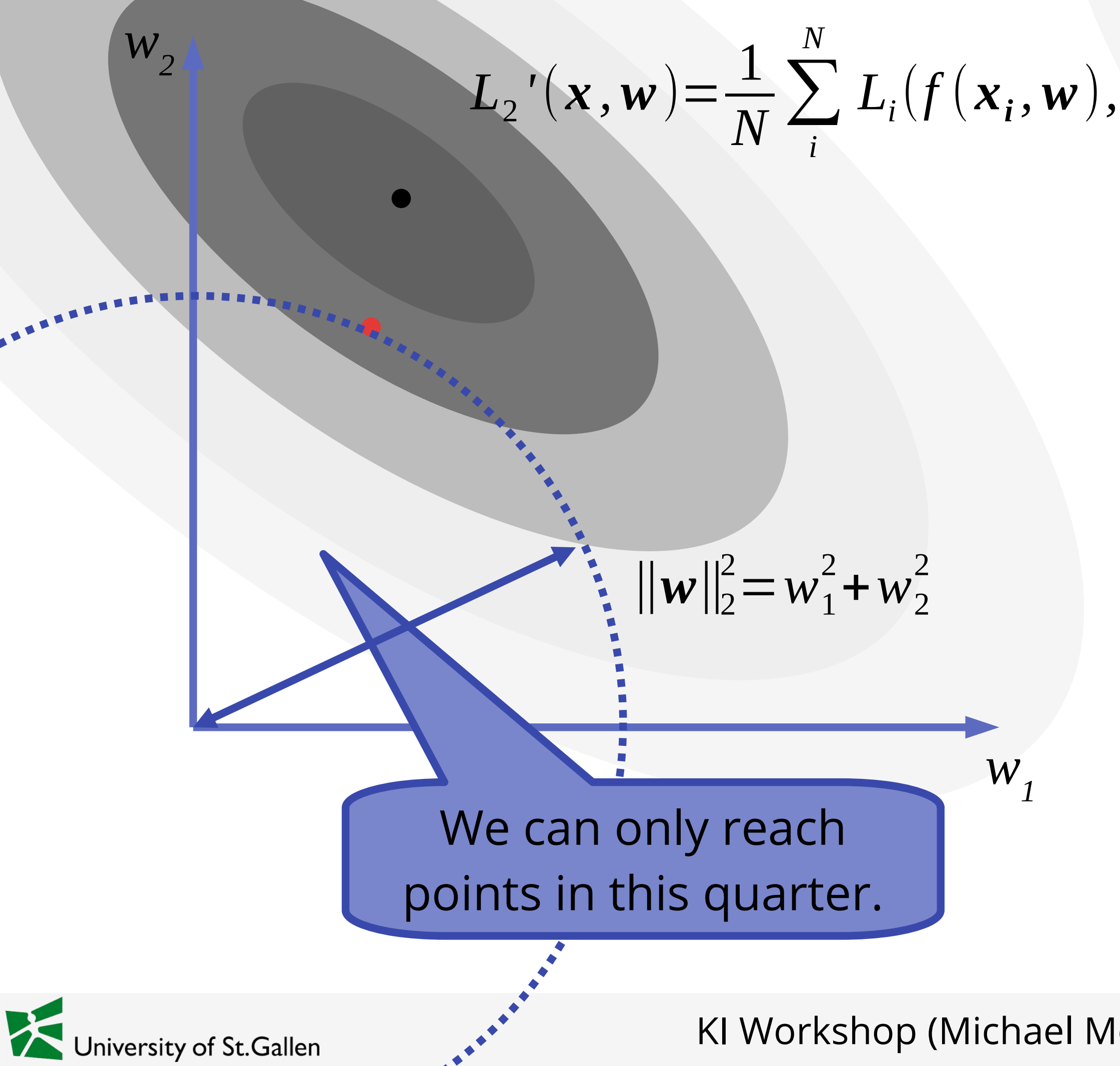
$$L_1'(\mathbf{x},\mathbf{w})=\frac{1}{N}\sum_{i}^{N} L_i(f(\mathbf{x}_i,\mathbf{w}),\mathbf{y}_i)+\alpha\|\mathbf{w}\|_1$$



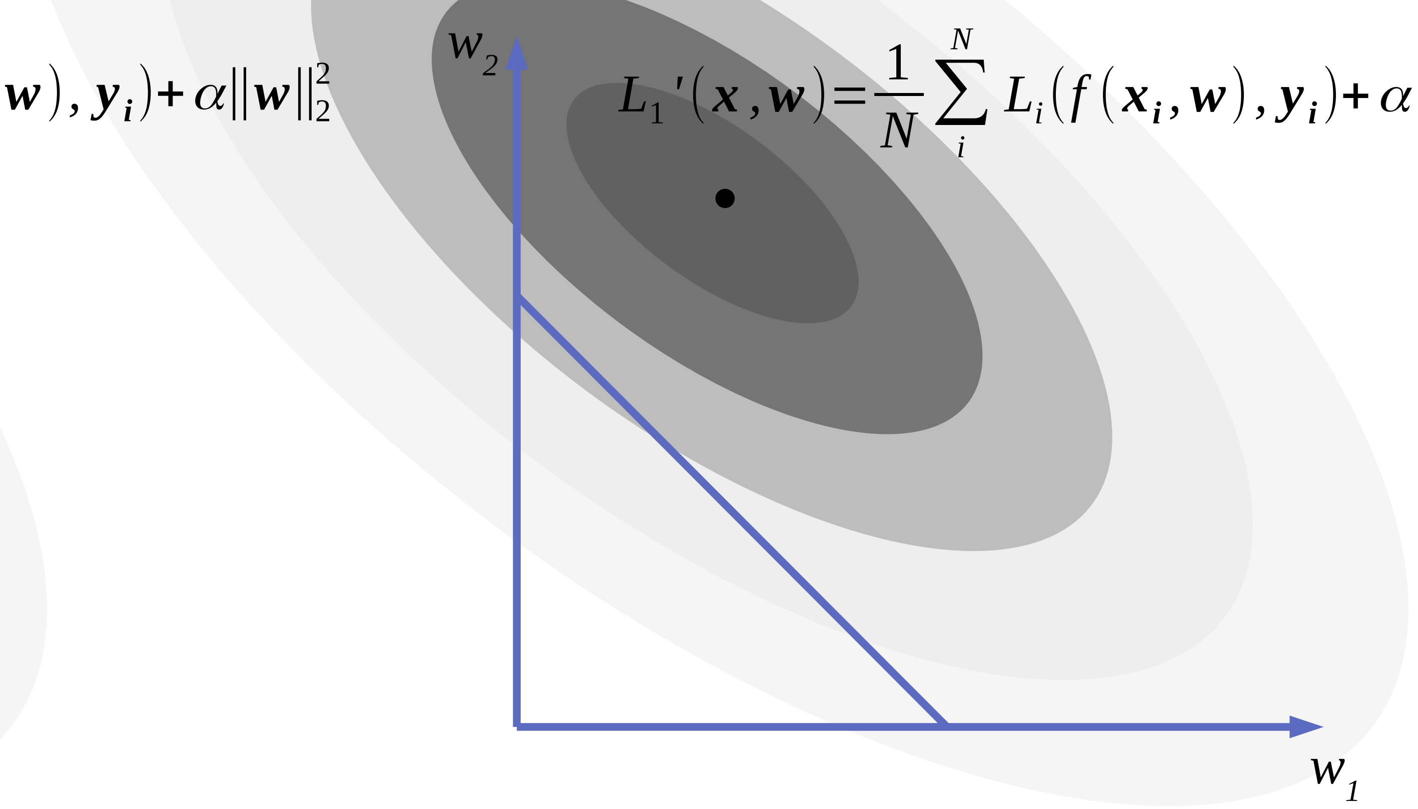
L2 vs. L1 regularization

Consider a 2-d loss space, spanned by w_1 and w_2. How can we get closest to the minimum loss with the two different regularization terms?

$$L_2'(\boldsymbol{x},\boldsymbol{w})=\frac{1}{N}\sum_i^N L_i(f(\boldsymbol{x}_i,\boldsymbol{w}),\boldsymbol{y}_i)+\alpha\|\boldsymbol{w}\|_2^2$$



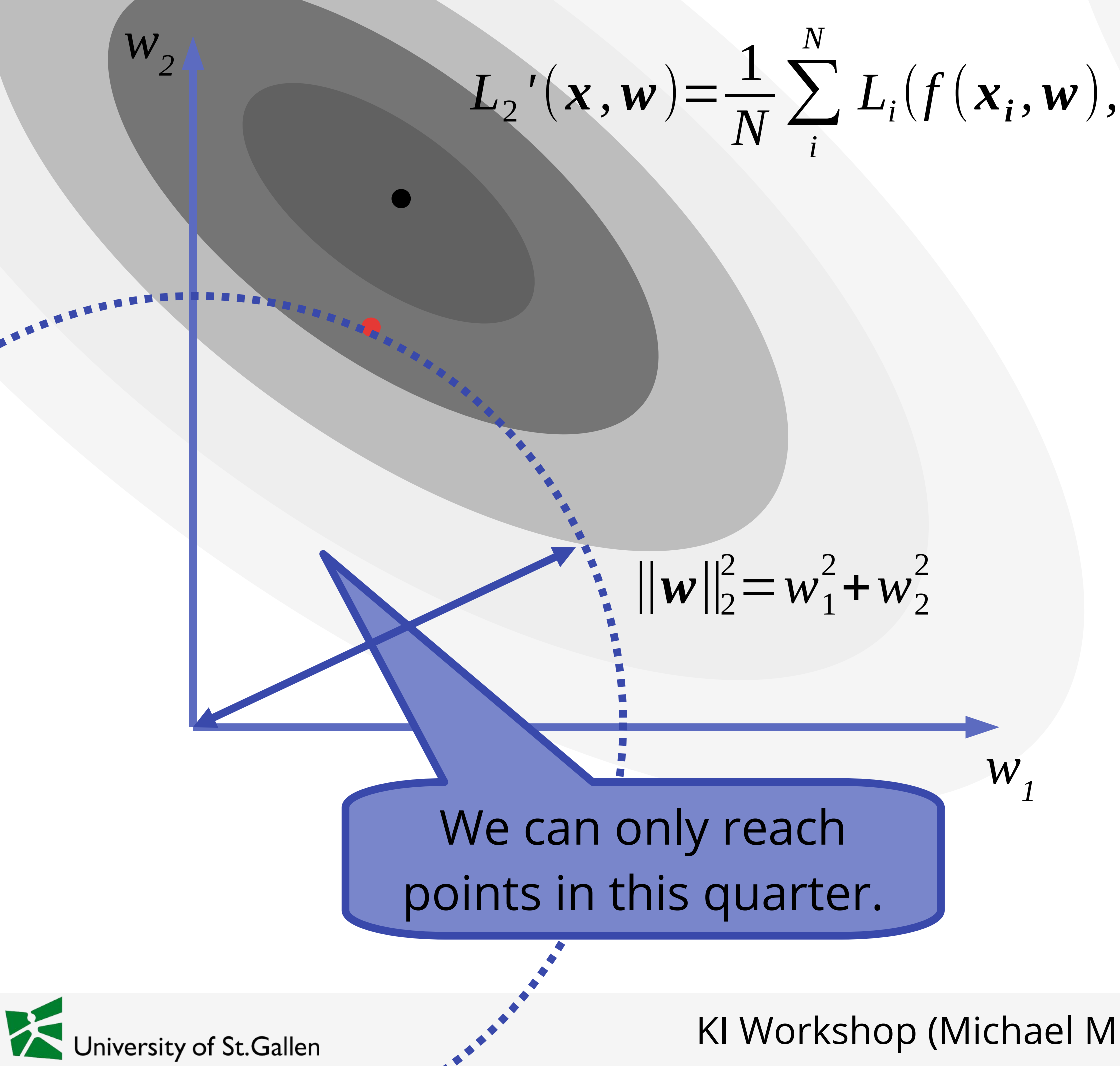
$$L_1'(\boldsymbol{x},\boldsymbol{w})=\frac{1}{N}\sum_i^N L_i(f(\boldsymbol{x}_i,\boldsymbol{w}),\boldsymbol{y}_i)+\alpha\|\boldsymbol{w}\|_1$$



L2 vs. L1 regularization

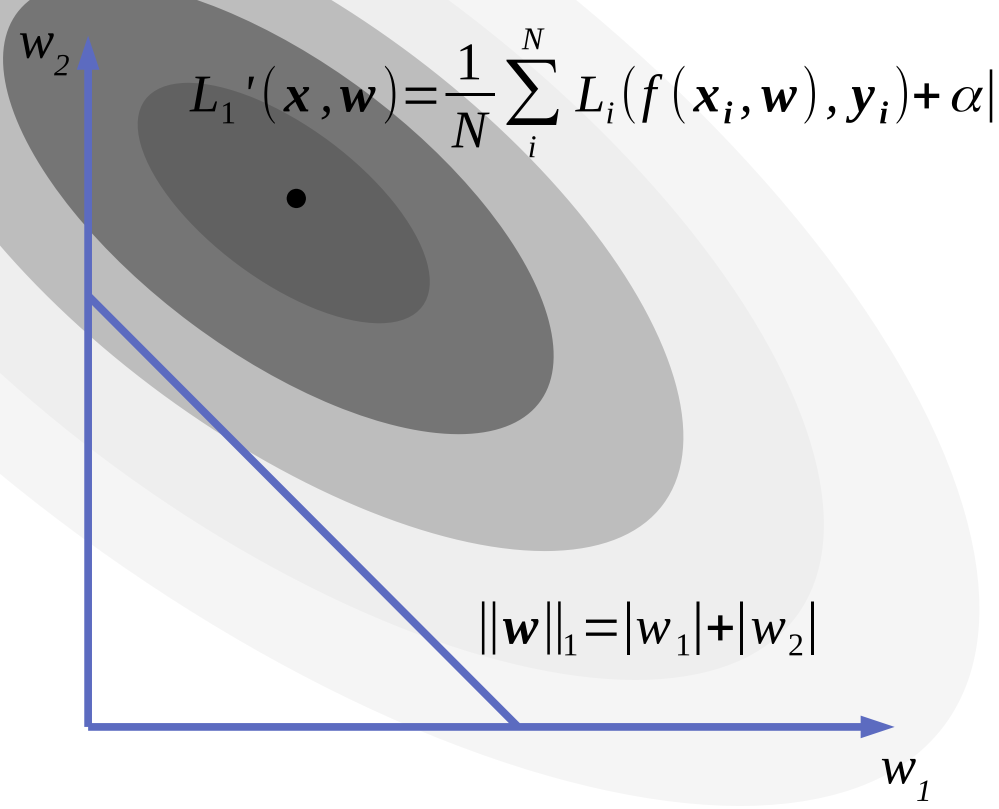
Consider a 2-d loss space, spanned by w_1 and w_2. How can we get closest to the minimum loss with the two different regularization terms?

$$L_2'(\boldsymbol{x},\boldsymbol{w})=\frac{1}{N}\sum_i^N L_i(f(\boldsymbol{x_i},\boldsymbol{w}),\boldsymbol{y_i})+\alpha\|\boldsymbol{w}\|_2^2$$



$$\|\boldsymbol{w}\|_2^2=w_1^2+w_2^2$$

$$L_1'(\boldsymbol{x},\boldsymbol{w})=\frac{1}{N}\sum_i^N L_i(f(\boldsymbol{x_i},\boldsymbol{w}),\boldsymbol{y_i})+\alpha\|\boldsymbol{w}\|_1$$

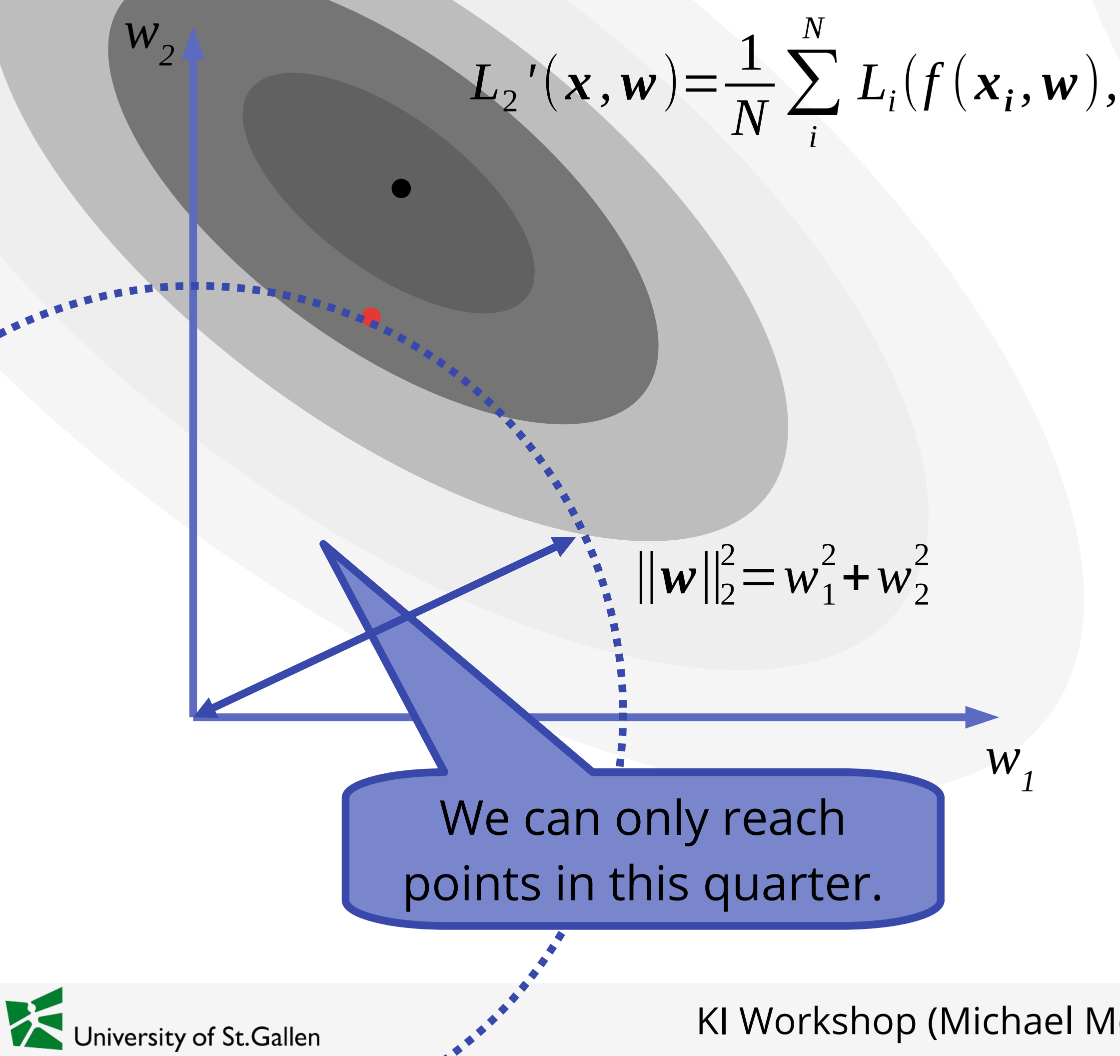


$$\|\boldsymbol{w}\|_1=|w_1|+|w_2|$$

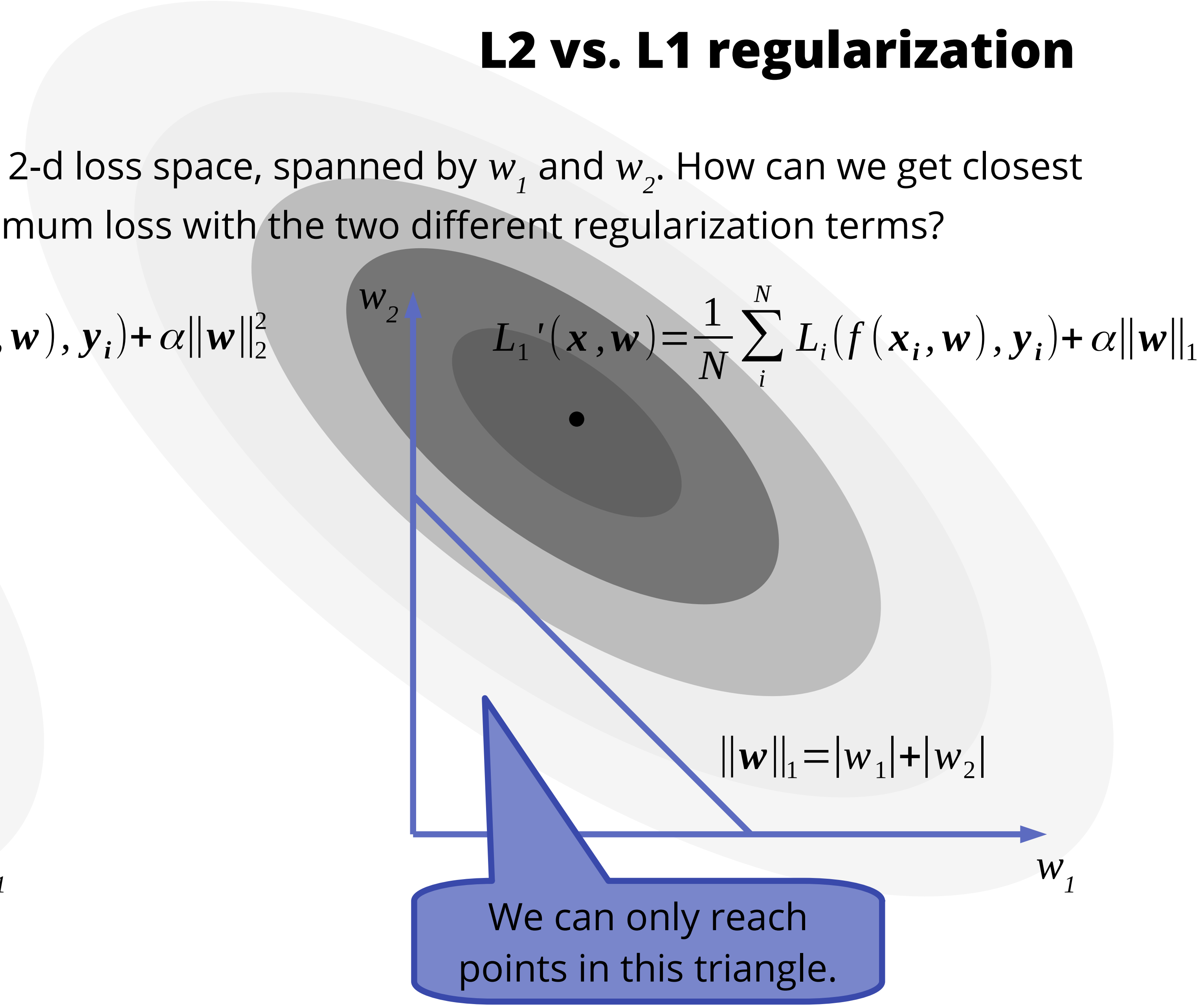
L2 vs. L1 regularization

Consider a 2-d loss space, spanned by w_1 and w_2. How can we get closest to the minimum loss with the two different regularization terms?

$$L_2'(\boldsymbol{x},\boldsymbol{w})=\frac{1}{N}\sum_i^N L_i(f(\boldsymbol{x_i},\boldsymbol{w}),\boldsymbol{y_i})+\alpha\|\boldsymbol{w}\|_2^2$$



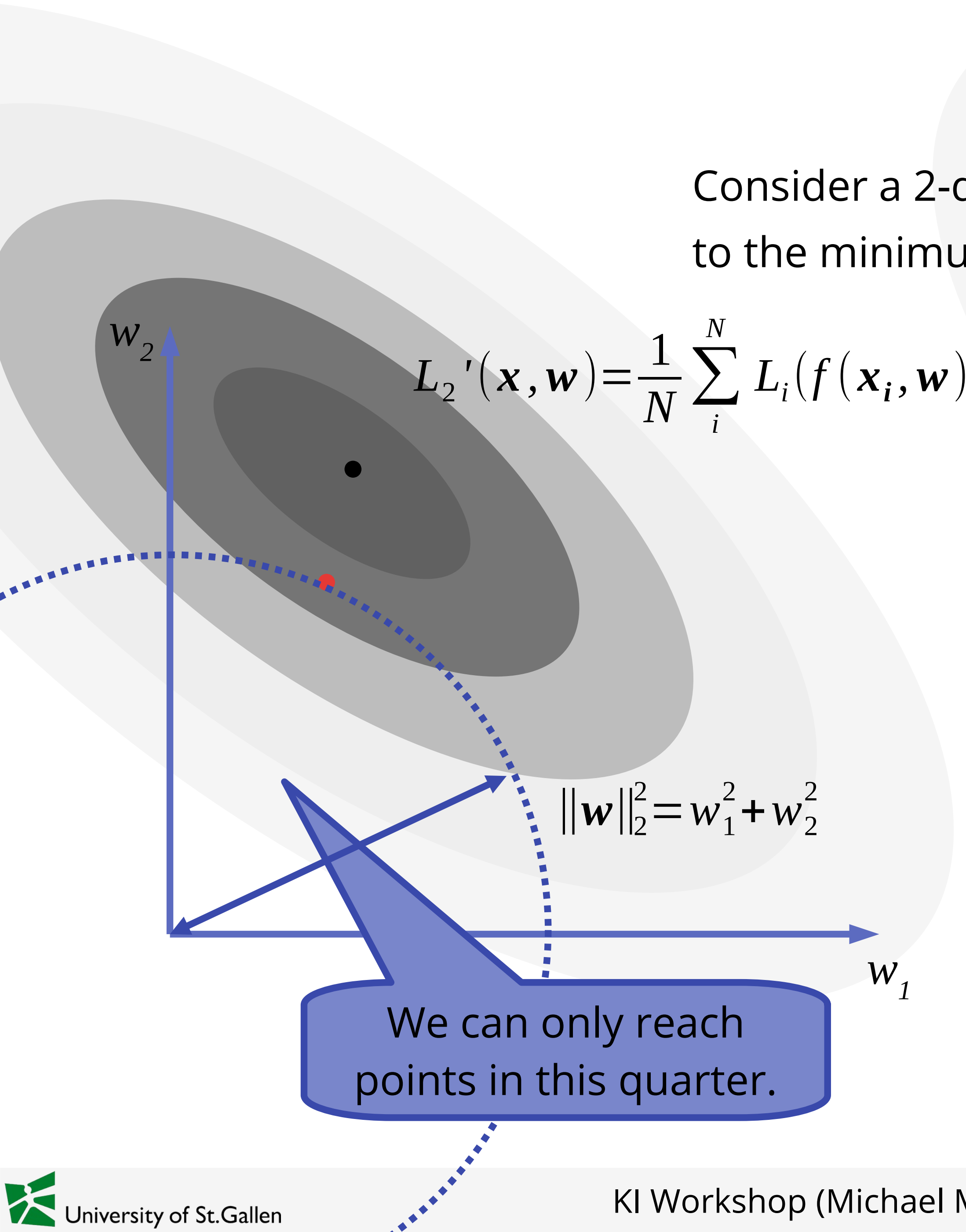
$$L_1'(\boldsymbol{x},\boldsymbol{w})=\frac{1}{N}\sum_i^N L_i(f(\boldsymbol{x_i},\boldsymbol{w}),\boldsymbol{y_i})+\alpha\|\boldsymbol{w}\|_1$$



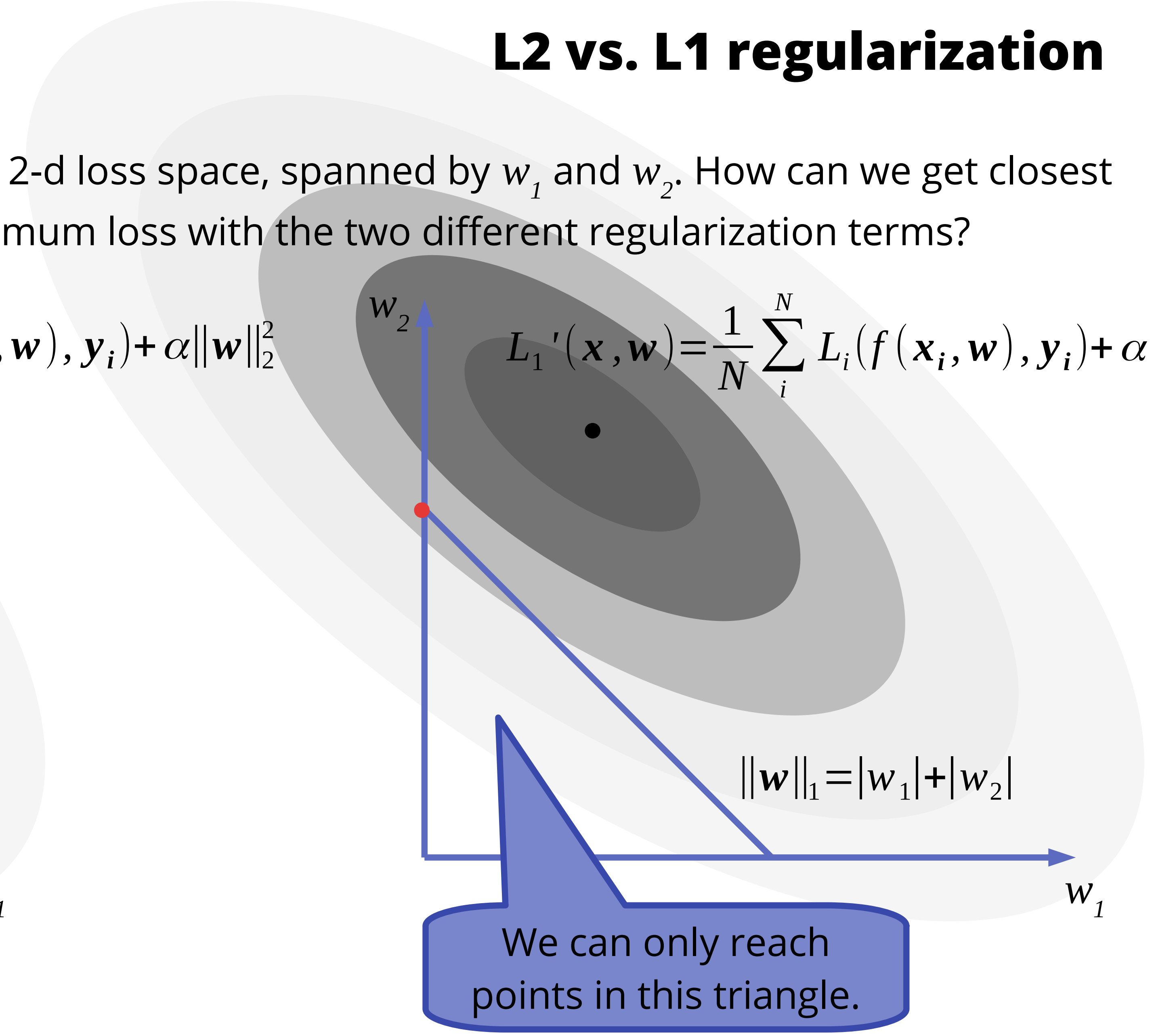
L2 vs. L1 regularization

Consider a 2-d loss space, spanned by w_1 and w_2. How can we get closest to the minimum loss with the two different regularization terms?

$$L_2'(\boldsymbol{x},\boldsymbol{w})=\frac{1}{N}\sum_i^N L_i(f(\boldsymbol{x_i},\boldsymbol{w}),\boldsymbol{y_i})+\alpha\|\boldsymbol{w}\|_2^2$$



$$L_1'(\boldsymbol{x},\boldsymbol{w})=\frac{1}{N}\sum_i^N L_i(f(\boldsymbol{x_i},\boldsymbol{w}),\boldsymbol{y_i})+\alpha\|\boldsymbol{w}\|_1$$



L2 vs. L1 regularization

L2 vs. L1 regularization

Which type of regularization should you use?

L2 vs. L1 regularization

Which type of regularization should you use?

In the end, they both do more or less the same job of regularizing your model with one significant difference:

L2 vs. L1 regularization

Which type of regularization should you use?

In the end, they both do more or less the same job of regularizing your model with one significant difference:

- L2 regularization prevents the model from overfitting by *modulating the impact of its input features in a homogeneous way*, while

L2 vs. L1 regularization

Which type of regularization should you use?

In the end, they both do more or less the same job of regularizing your model with one significant difference:

- L2 regularization prevents the model from overfitting by *modulating the impact of its input features in a homogeneous way*, while
- L1 regularization prevents the model from overfitting by *focusing on those features which seem to be most important*.

L2 vs. L1 regularization

Which type of regularization should you use?

In the end, they both do more or less the same job of regularizing your model with one significant difference:

- L2 regularization prevents the model from overfitting by *modulating the impact of its input features in a homogeneous way*, while
- L1 regularization prevents the model from overfitting by *focusing on those features which seem to be most important*.

Both regularization techniques are widely used in regression and classification tasks. They can be deployed in any machine learning model that minimizes a loss function.

Linear models – pros and cons

Linear models – pros and cons

Pros:

Linear models – pros and cons

Pros:

- Easy to understand and implement; resource efficient even for large and sparse data sets.

Linear models – pros and cons

Pros:

- Easy to understand and implement; resource efficient even for large and sparse data sets.
- Least squares method always provides best-fit result, if the data are appropriate.

Linear models – pros and cons

Pros:

- Easy to understand and implement; resource efficient even for large and sparse data sets.
- Least squares method always provides best-fit result, if the data are appropriate.
- Good interpretability due to linear nature of the model.

Linear models – pros and cons

Pros:

- Easy to understand and implement; resource efficient even for large and sparse data sets.
- Least squares method always provides best-fit result, if the data are appropriate.
- Good interpretability due to linear nature of the model.
- Easy to regularize.

Linear models – pros and cons

Pros:

- Easy to understand and implement; resource efficient even for large and sparse data sets.
- Least squares method always provides best-fit result, if the data are appropriate.
- Good interpretability due to linear nature of the model.
- Easy to regularize.

Cons:

Linear models – pros and cons

Pros:

- Easy to understand and implement; resource efficient even for large and sparse data sets.
- Least squares method always provides best-fit result, if the data are appropriate.
- Good interpretability due to linear nature of the model.
- Easy to regularize.

Cons:

- Limited flexibility: data distribution must be brought into a form that is linear (regression) or linearly separable (classification).

Linear models – pros and cons

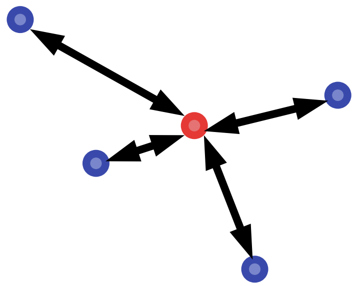
Pros:

- Easy to understand and implement; resource efficient even for large and sparse data sets.
- Least squares method always provides best-fit result, if the data are appropriate.
- Good interpretability due to linear nature of the model.
- Easy to regularize.

Cons:

- Limited flexibility: data distribution must be brought into a form that is linear (regression) or linearly separable (classification).
- Susceptible to overfitting if not combined with regularizer.

Nearest-Neighbor models



Nearest neighbor models

Nearest neighbor models are **non-parametric** and simply rely on **distances** between data points.

Nearest neighbor models

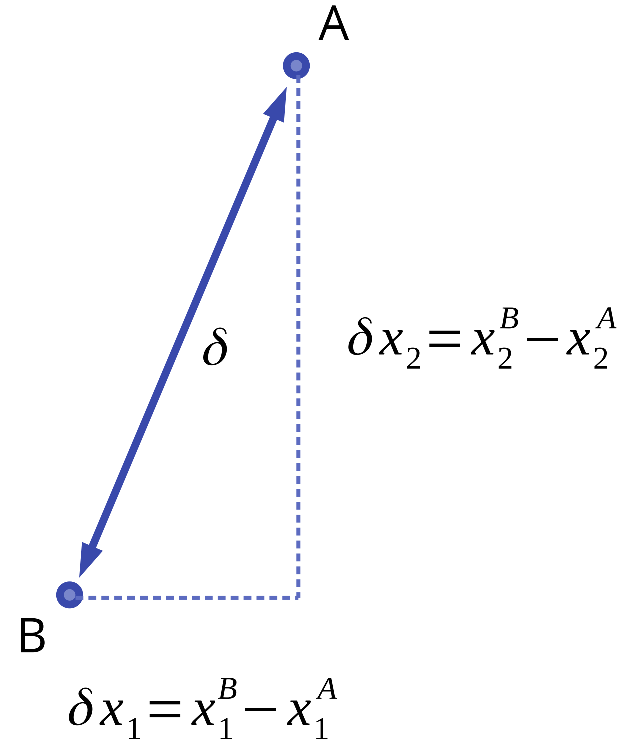
Nearest neighbor models are **non-parametric** and simply rely on **distances** between data points.

Distances can be defined as metrics. A common distance metric is the **Euclidean distance** between two data points $A=(x_1^A, x_2^A)$ and $B=(x_1^B, x_2^B)$:

Nearest neighbor models

Nearest neighbor models are **non-parametric** and simply rely on **distances** between data points.

Distances can be defined as metrics. A common distance metric is the **Euclidean distance** between two data points $A=(x_1^A, x_2^A)$ and $B=(x_1^B, x_2^B)$:

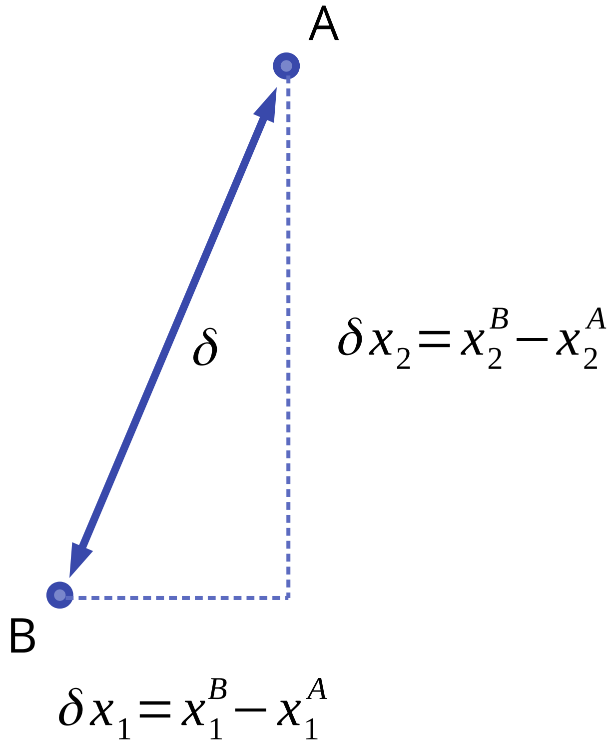


Nearest neighbor models

Nearest neighbor models are **non-parametric** and simply rely on **distances** between data points.

Distances can be defined as metrics. A common distance metric is the **Euclidean distance** between two data points $A=(x_1^A, x_2^A)$ and $B=(x_1^B, x_2^B)$:

$$\delta=\sqrt{\delta x_1^2+\delta x_2^2}=\sqrt{(x_1^B-x_1^A)^2+(x_2^B-x_2^A)^2}$$

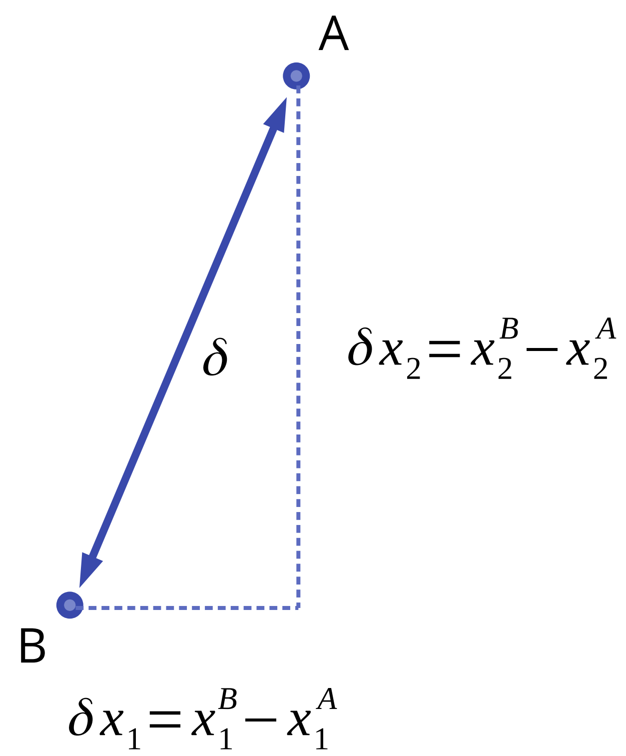


Nearest neighbor models

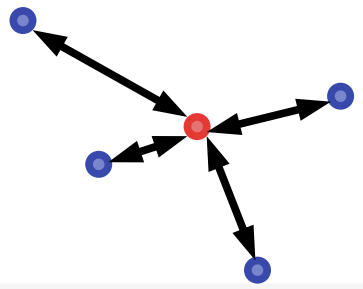
Nearest neighbor models are **non-parametric** and simply rely on **distances** between data points.

Distances can be defined as metrics. A common distance metric is the **Euclidean distance** between two data points $A=(x_1^A, x_2^A)$ and $B=(x_1^B, x_2^B)$:

$$\delta=\sqrt{\delta x_1^2+\delta x_2^2}=\sqrt{(x_1^B-x_1^A)^2+(x_2^B-x_2^A)^2}$$



Nearest neighbor methods utilize distances between datapoints for **classification** and **regression** tasks.



k-nearest neighbor classification

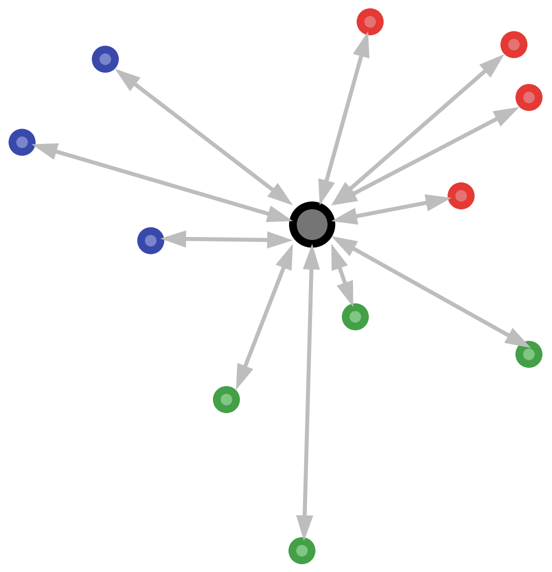
k-nearest neighbor (knn) classifiers predict class affiliation of an unseen data point based on **majority voting** of its ***k* nearest neighbors** in a seen data set with ground-truth labels.

knn models are not trained in the general sense. Instead, the distance of each unseen data point from all seen data points is calculated.

k-nearest neighbor classification

k-nearest neighbor (knn) classifiers predict class affiliation of an unseen data point based on **majority voting** of its ***k* nearest neighbors** in a seen data set with ground-truth labels.

knn models are not trained in the general sense. Instead, the distance of each unseen data point from all seen data points is calculated.

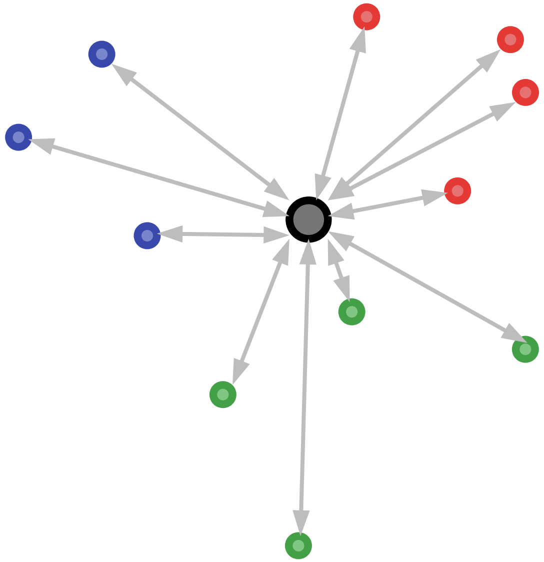


compute distances

k-nearest neighbor classification

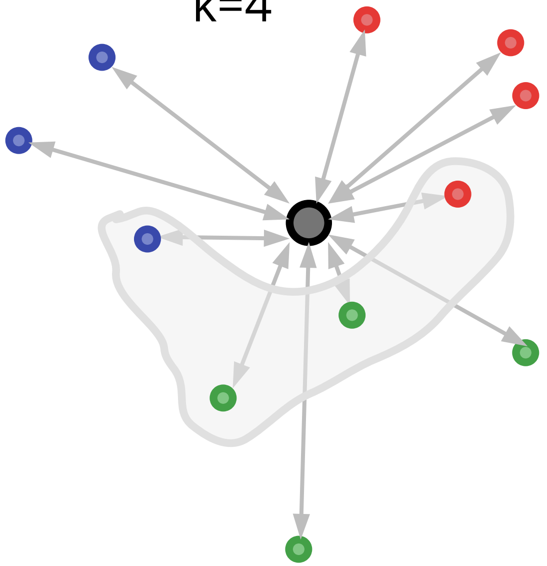
k-nearest neighbor (knn) classifiers predict class affiliation of an unseen data point based on **majority voting** of its ***k* nearest neighbors** in a seen data set with ground-truth labels.

knn models are not trained in the general sense. Instead, the distance of each unseen data point from all seen data points is calculated.



compute distances

k=4

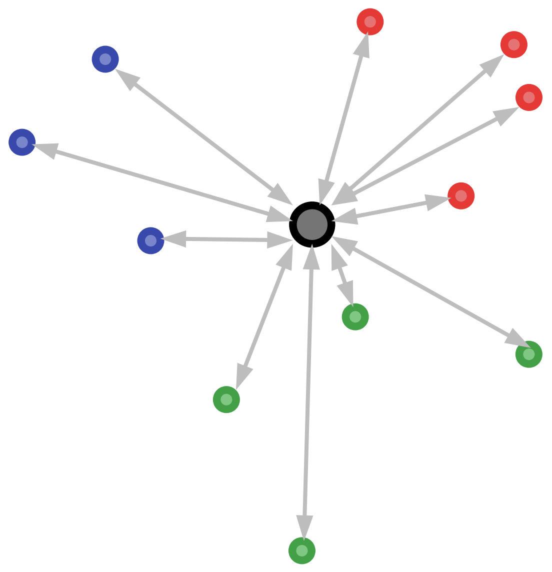


identify *k* nearest neighbors

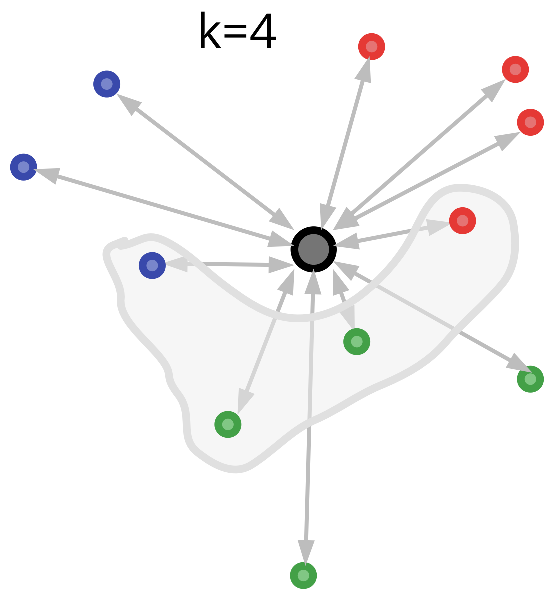
k-nearest neighbor classification

k-nearest neighbor (knn) classifiers predict class affiliation of an unseen data point based on **majority voting** of its ***k* nearest neighbors** in a seen data set with ground-truth labels.

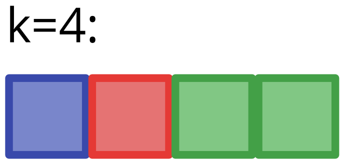
knn models are not trained in the general sense. Instead, the distance of each unseen data point from all seen data points is calculated.



compute distances



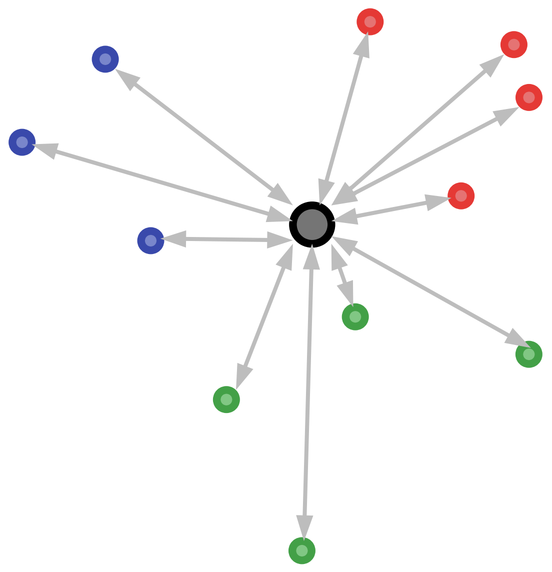
identify *k* nearest neighbors



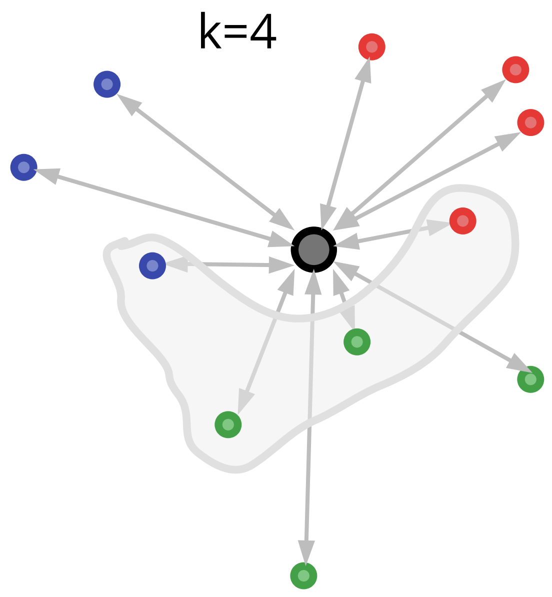
k-nearest neighbor classification

k-nearest neighbor (knn) classifiers predict class affiliation of an unseen data point based on **majority voting** of its ***k* nearest neighbors** in a seen data set with ground-truth labels.

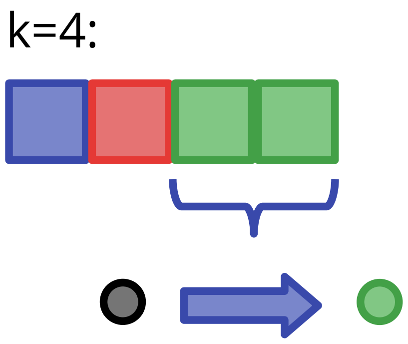
knn models are not trained in the general sense. Instead, the distance of each unseen data point from all seen data points is calculated.



compute distances

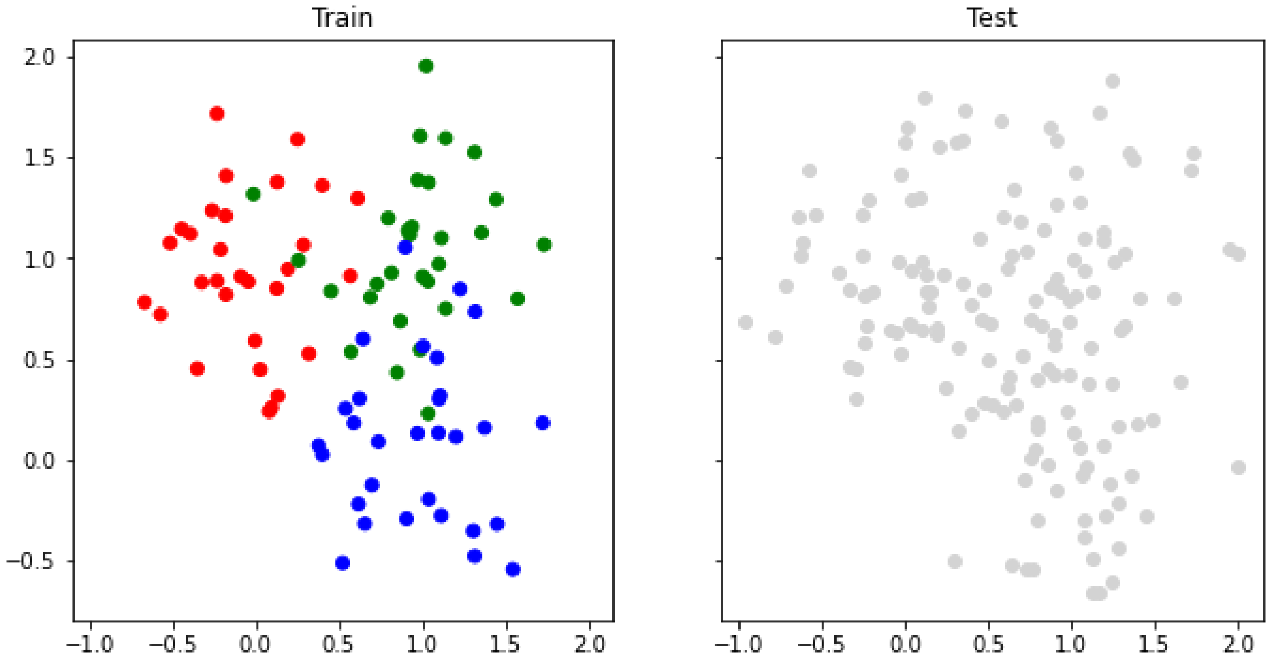


identify *k* nearest neighbors



class assignment

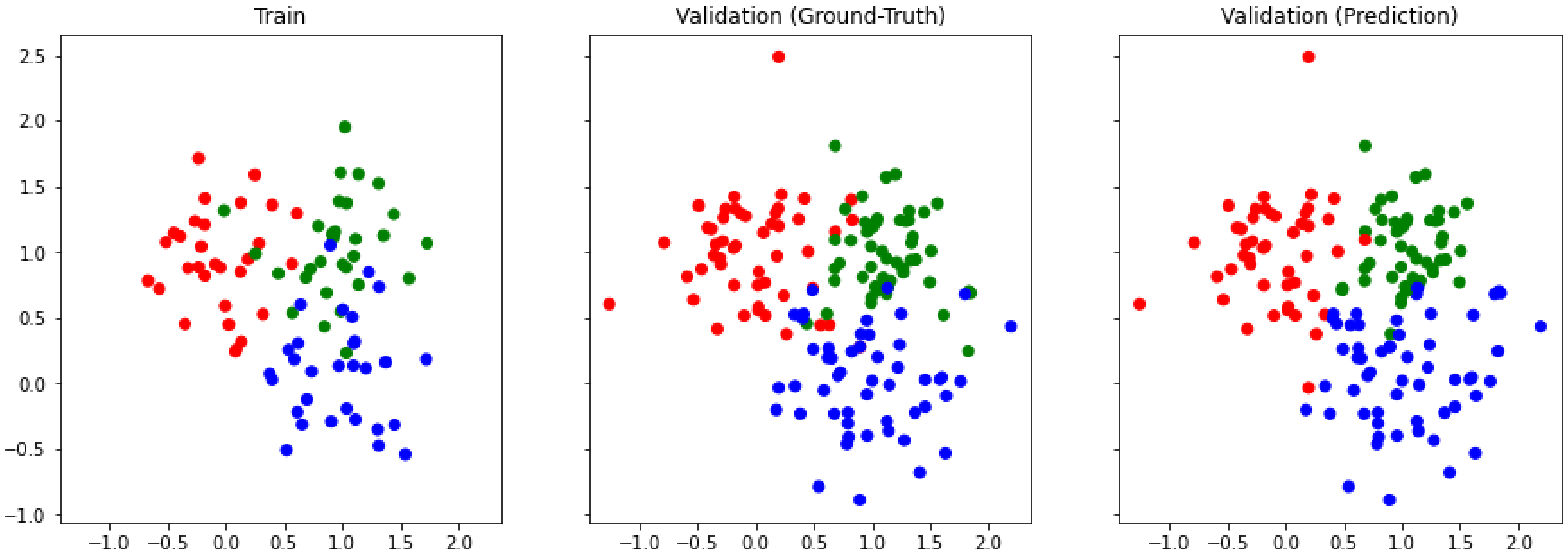
k-nearest neighbor classification - Example



3 overlapping clusters

How well can knn classify our test data set?

k-nearest neighbor classification - Example

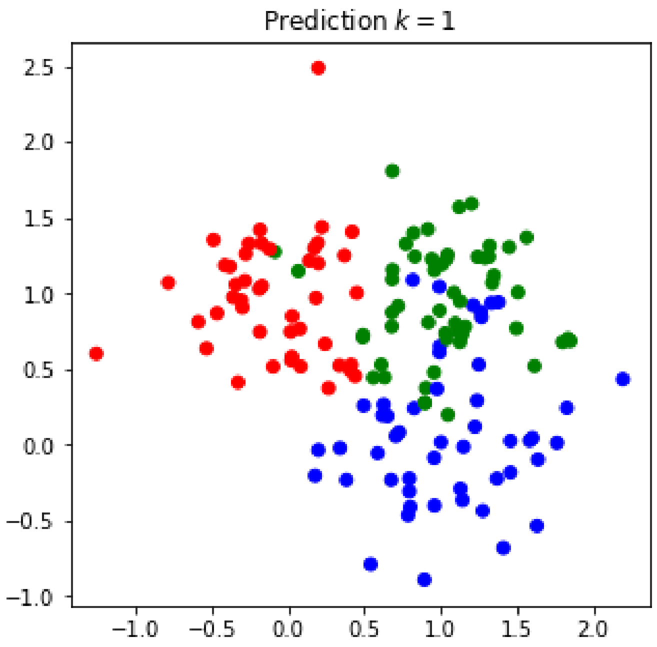


k=5: accuracy=0.867

Hyperparameter *k* has an impact on how well the model generalizes to unseen data: perform a **hyperparameter search**!

k-nearest neighbor classification - Example

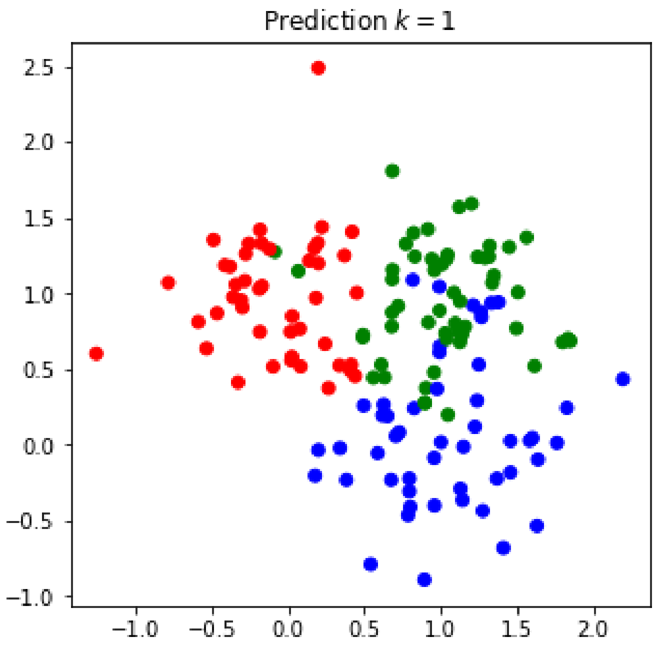
k-nearest neighbor classification - Example



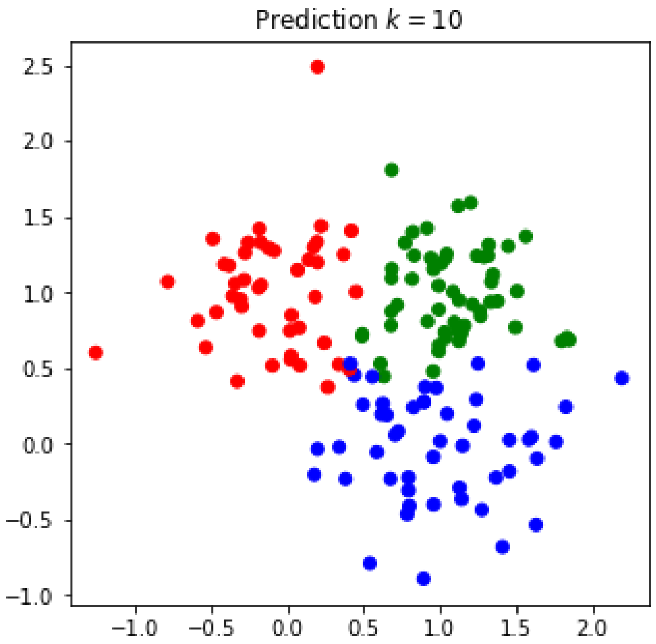
k=1

accuracy$_{val}$=0.800

k-nearest neighbor classification - Example

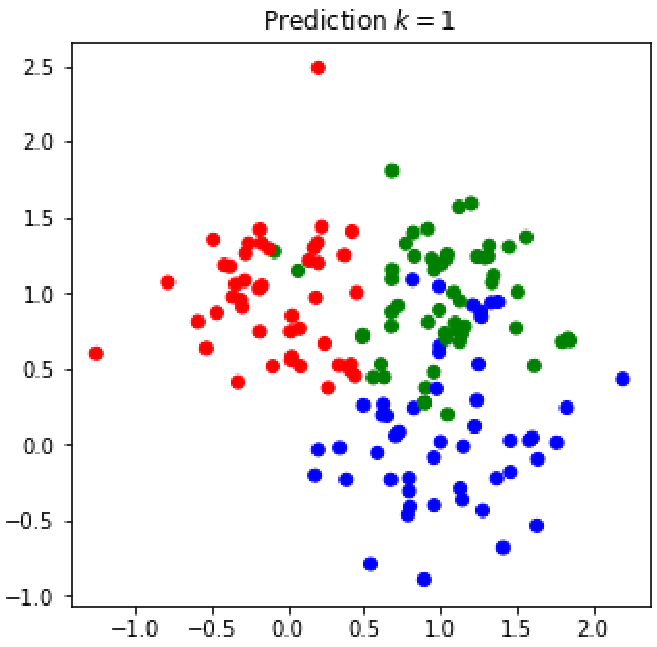


k=1
accuracy$_{val}$=0.800

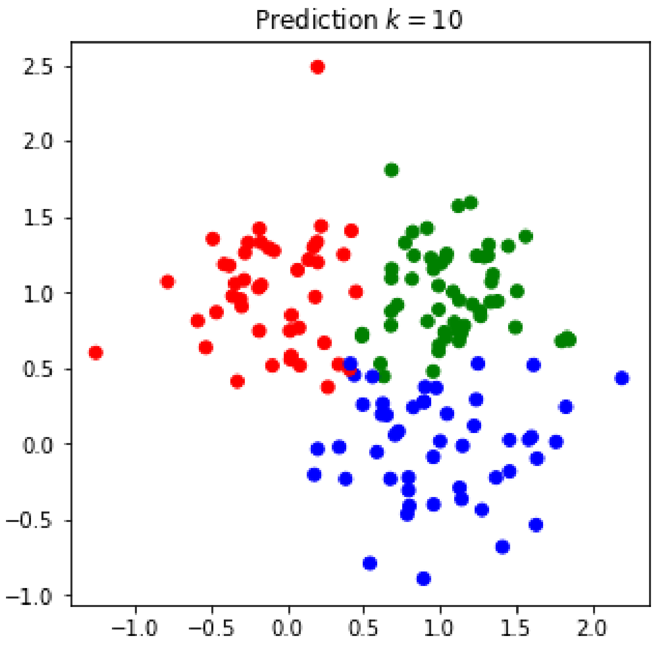


k=10
accuracy$_{val}$=0.893

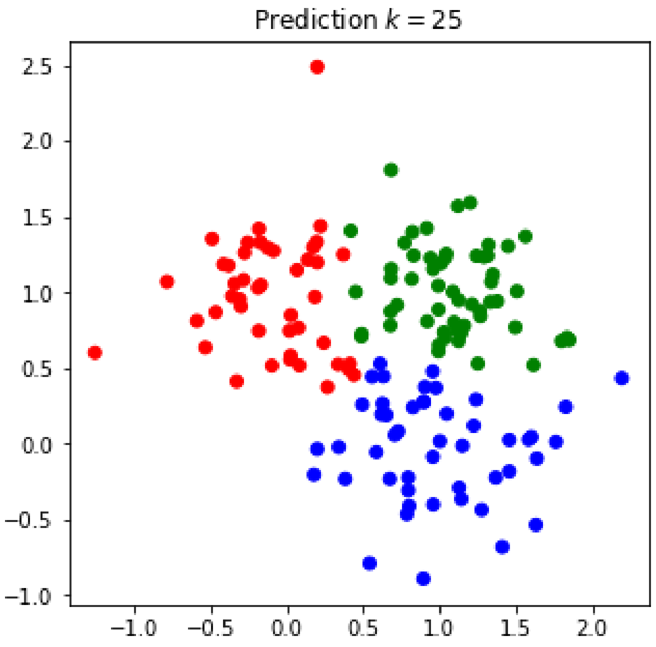
k-nearest neighbor classification - Example



k=1
accuracy$_{val}$=0.800

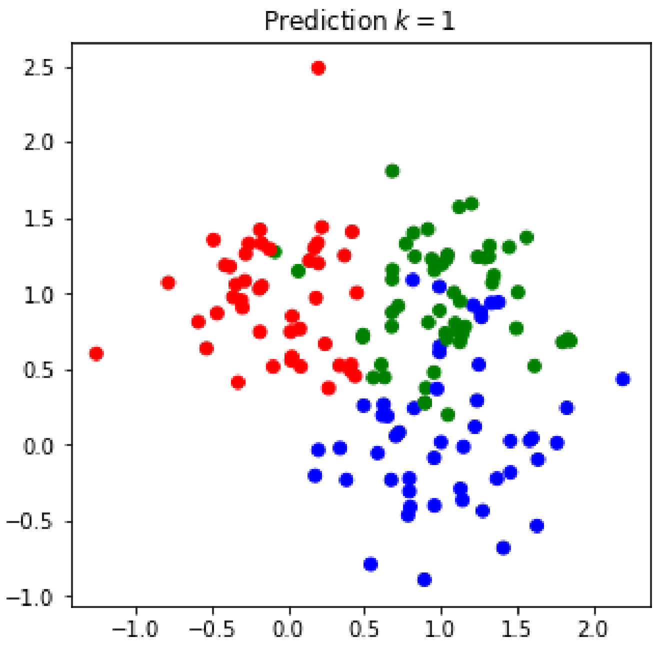


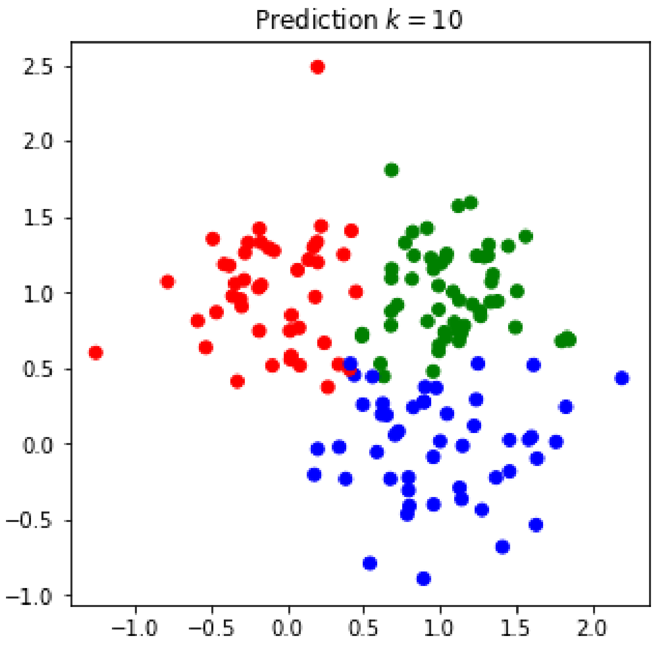
k=10
accuracy$_{val}$=0.893

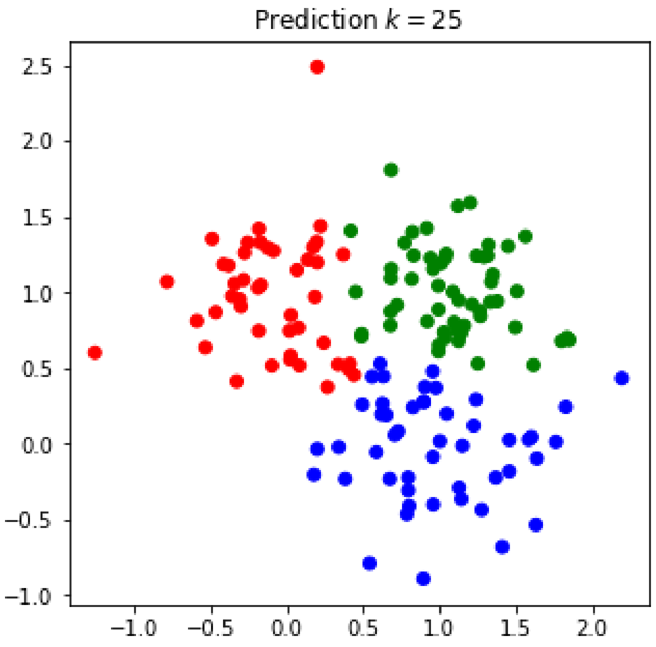


k=25
accuracy$_{val}$=0.873

k-nearest neighbor classification - Example







k=1
accuracy$_{val}$=0.800

Overfitting!

k=10
accuracy$_{val}$=0.893

accuracy$_{test}$=0.880

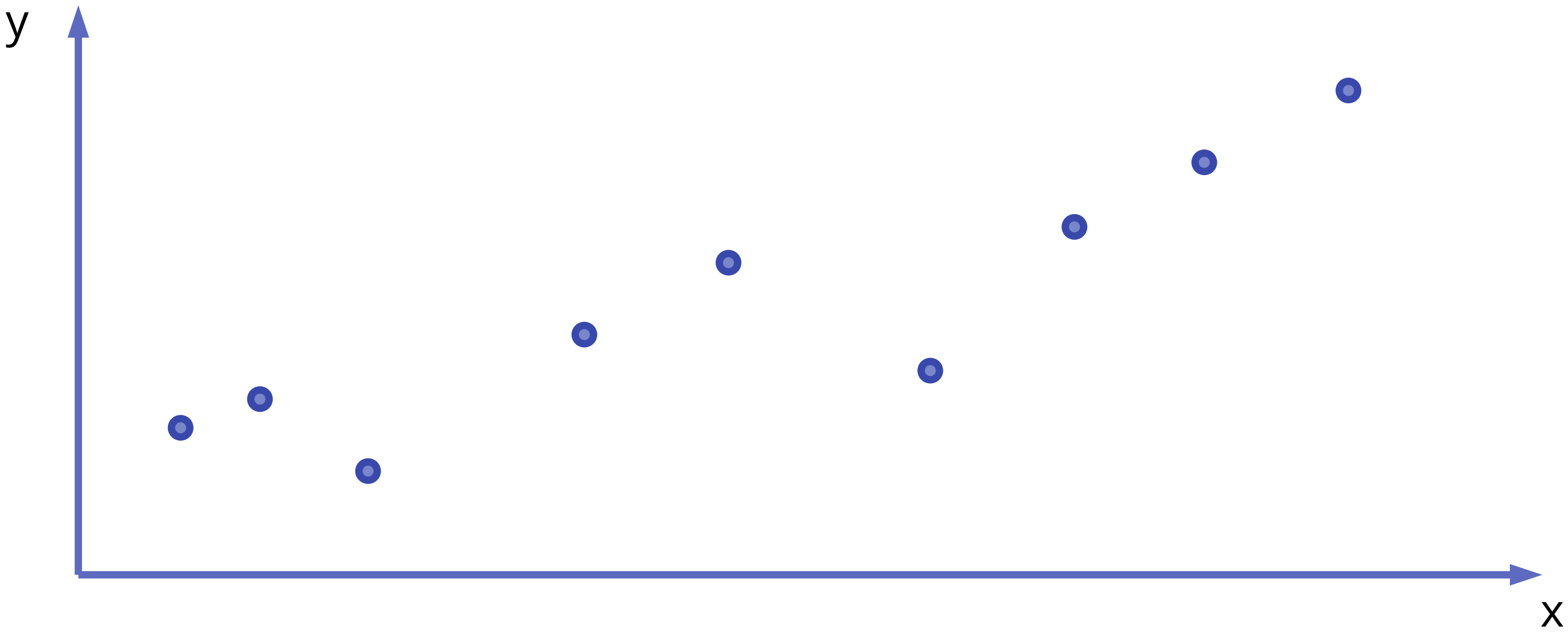
Best performance

k=25
accuracy$_{val}$=0.873

Underfitting!

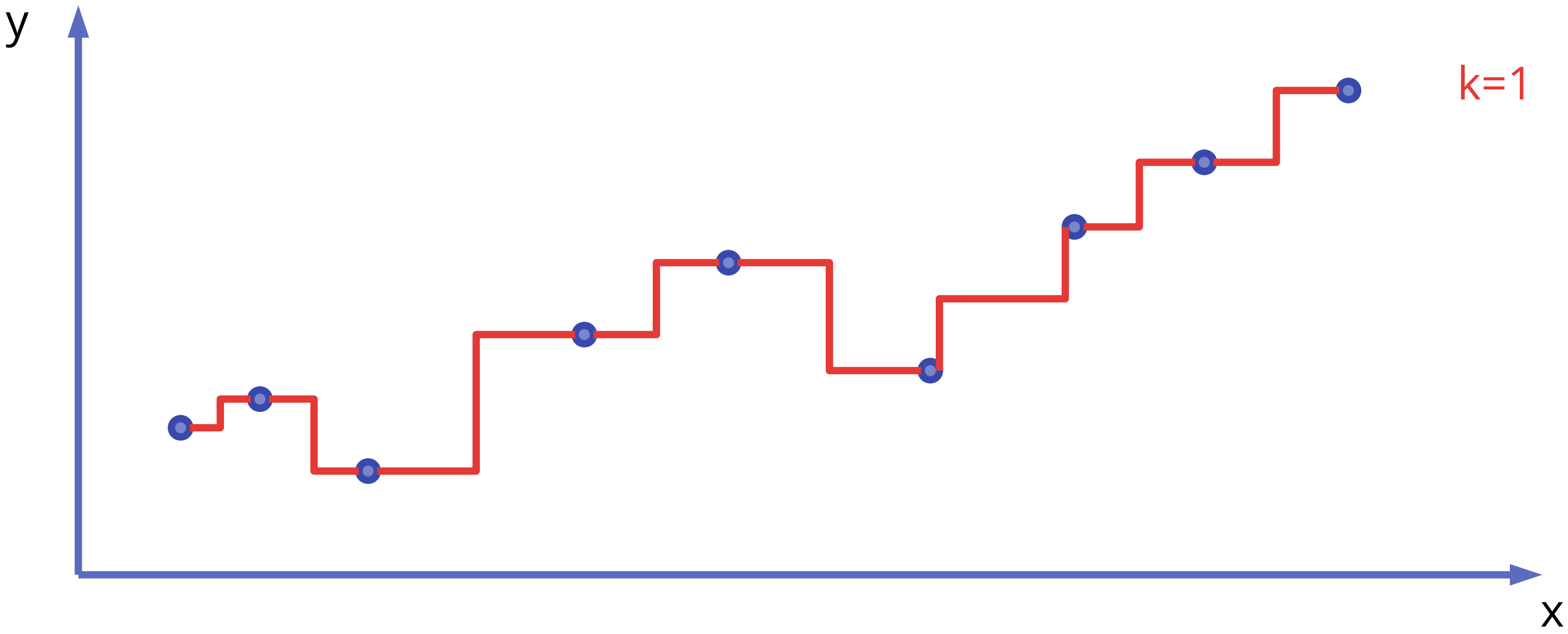
k-nearest neighbor regression

Nearest neighbor methods can also be implemented as regressors that are able to **interpolate** between and **smoothen** available data.



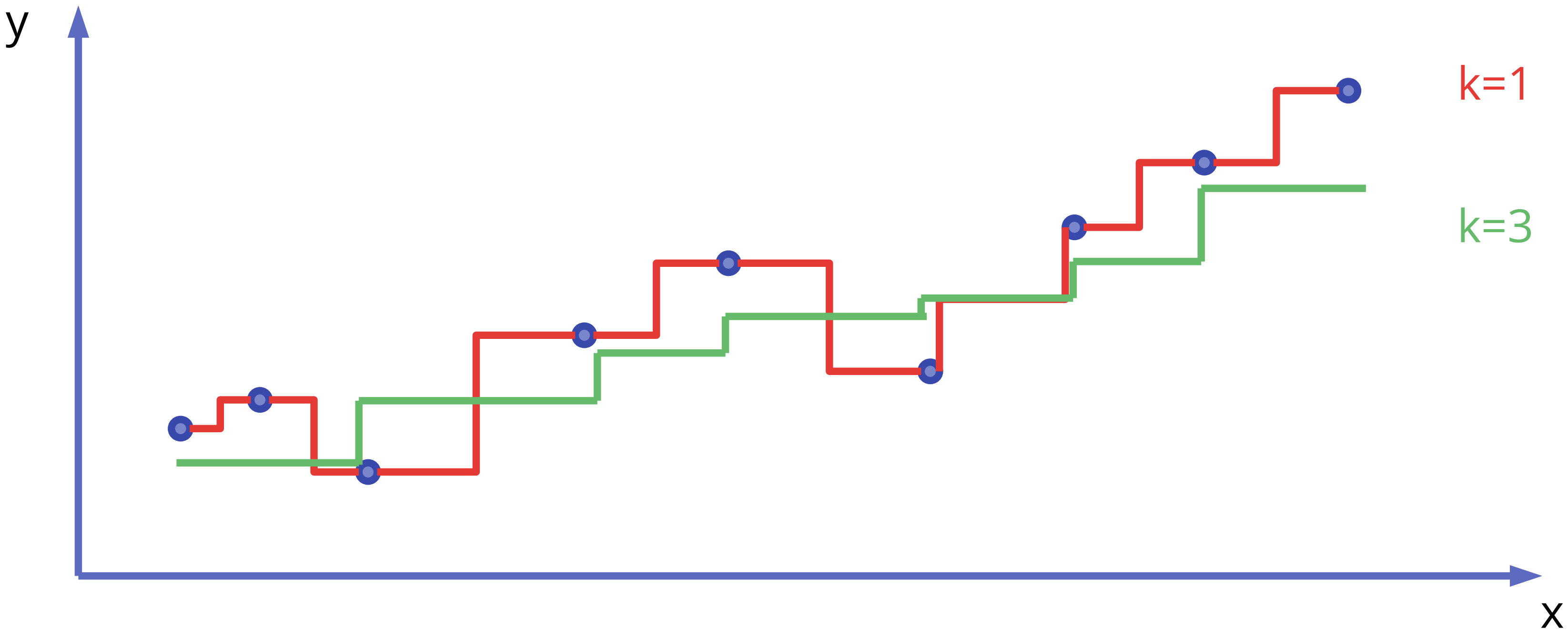
k-nearest neighbor regression

Nearest neighbor methods can also be implemented as regressors that are able to **interpolate** between and **smoothen** available data.



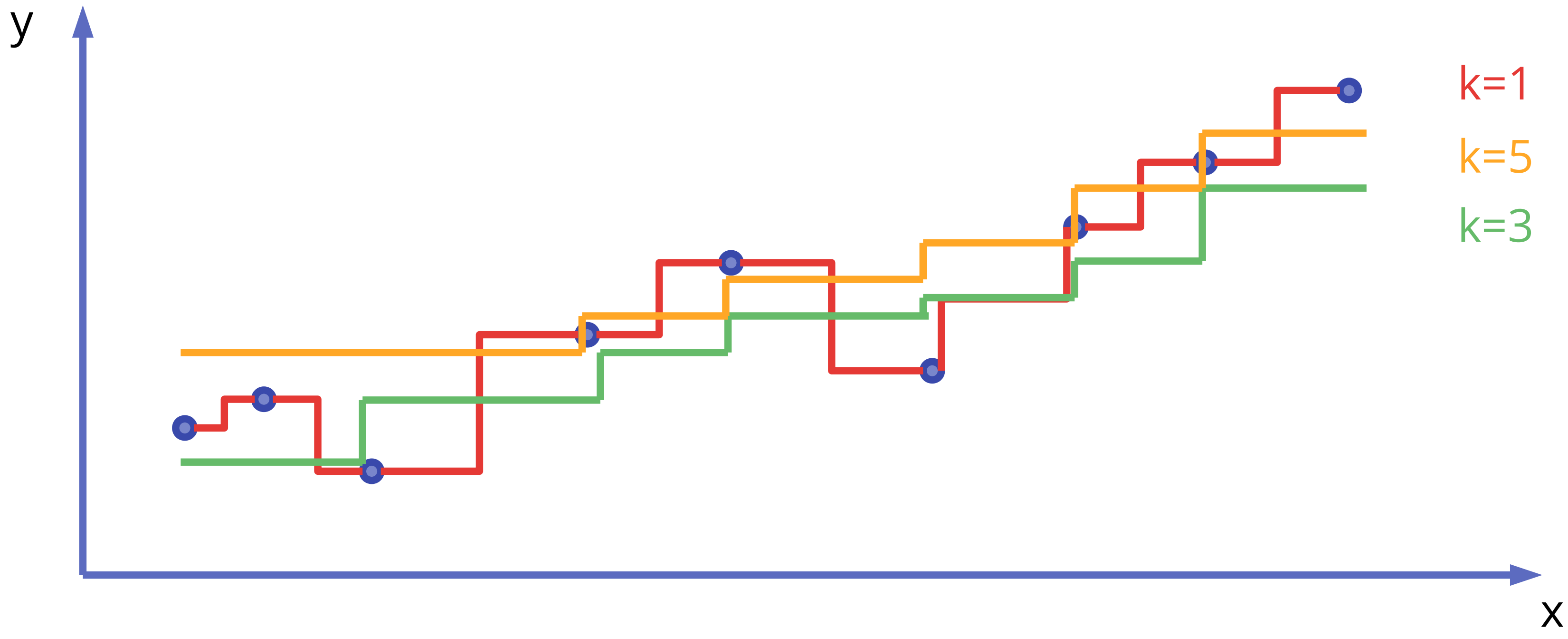
k-nearest neighbor regression

Nearest neighbor methods can also be implemented as regressors that are able to **interpolate** between and **smoothen** available data.



k-nearest neighbor regression

Nearest neighbor methods can also be implemented as regressors that are able to **interpolate** between and **smoothen** available data.



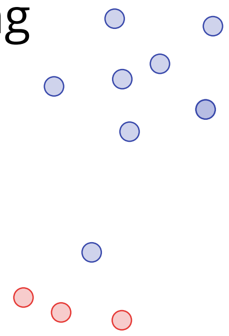
Regularizing nearest neighbor methods

Regularizing nearest neighbor methods is simply done through varying its hyperparameter, k.

Regularizing nearest neighbor methods

Regularizing nearest neighbor methods is simply done through varying its hyperparameter, k.

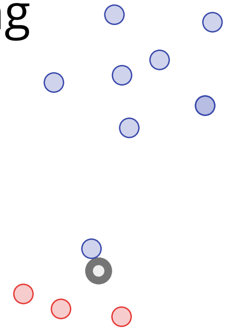
Training data:



Regularizing nearest neighbor methods

Regularizing nearest neighbor methods is simply done through varying its hyperparameter, k.

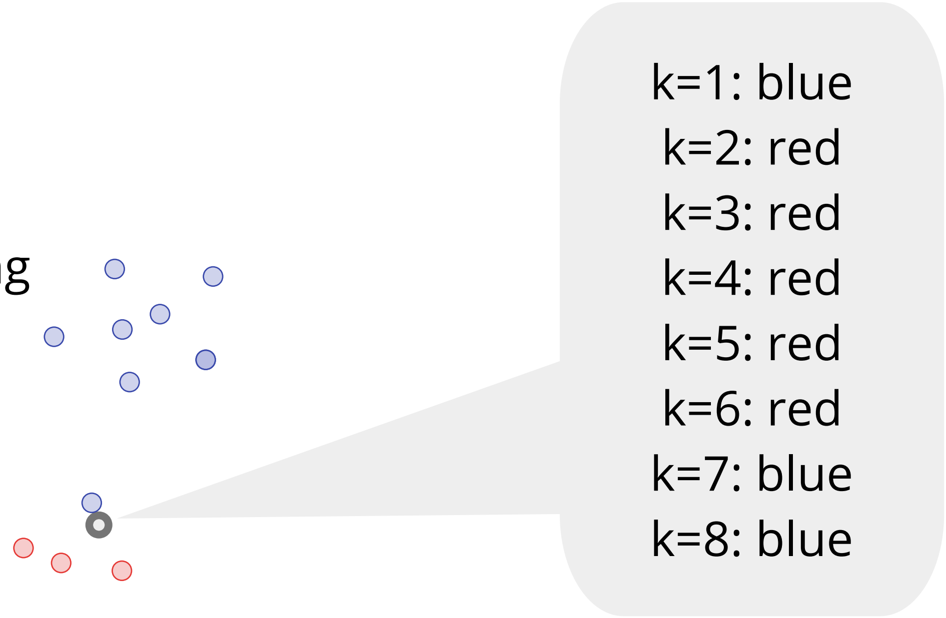
Training data:



Regularizing nearest neighbor methods

Regularizing nearest neighbor methods is simply done through varying its hyperparameter, k.

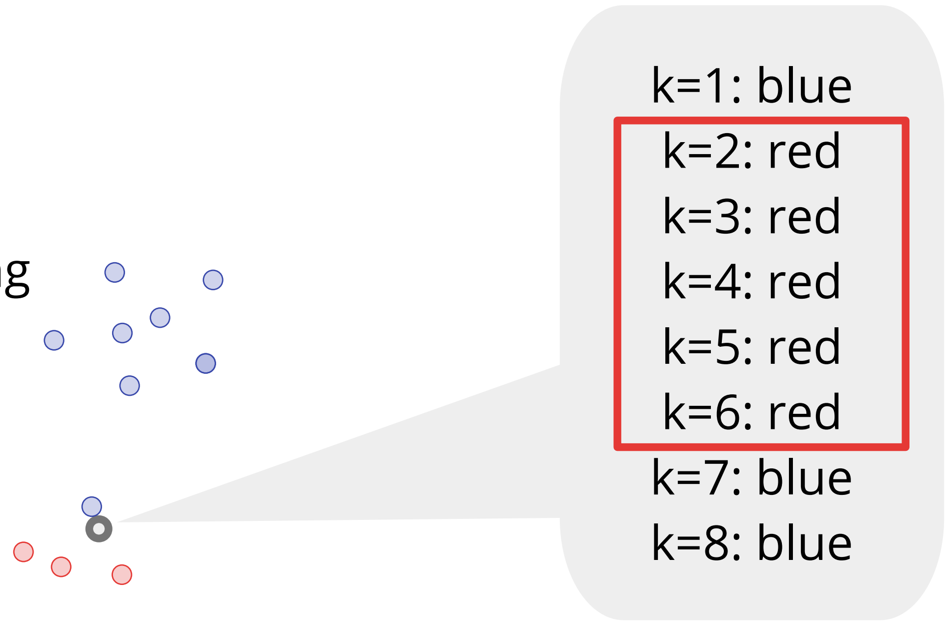
Training data:



Regularizing nearest neighbor methods

Regularizing nearest neighbor methods is simply done through varying its hyperparameter, k.

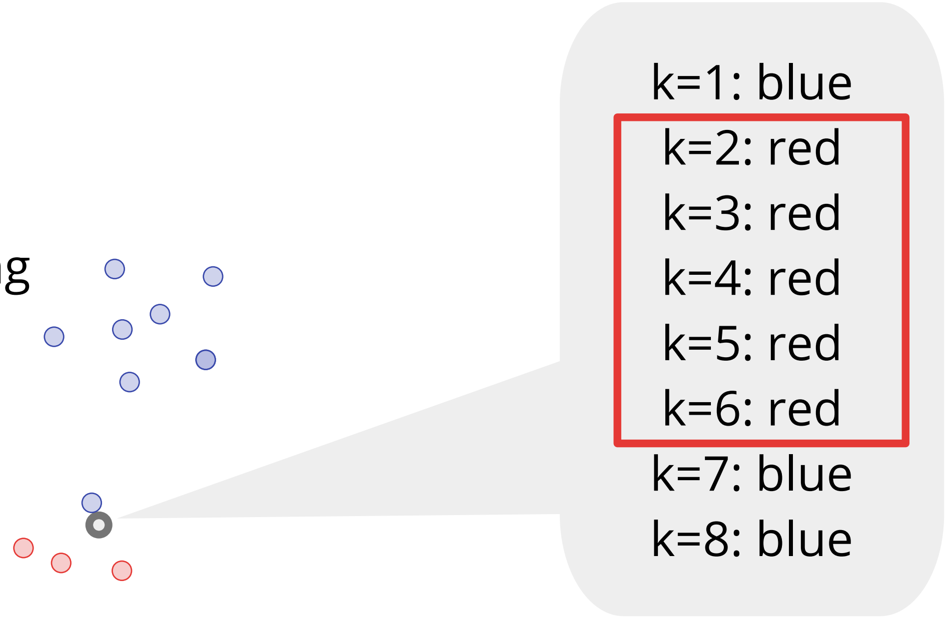
Training data:



Regularizing nearest neighbor methods

Regularizing nearest neighbor methods is simply done through varying its hyperparameter, k.

Training data:

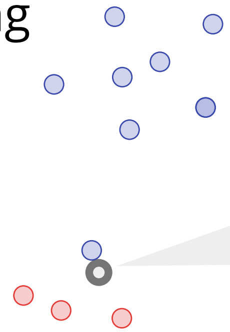


A low k is susceptible to small-scale variations and noise.

Regularizing nearest neighbor methods

Regularizing nearest neighbor methods is simply done through varying its hyperparameter, k.

Training data:



k=1: blue
k=2: red
k=3: red
k=4: red
k=5: red
k=6: red
k=7: blue
k=8: blue

A low k is susceptible to small-scale variations and noise.

A high k may miss local details.

k-nearest neighbor classification – pros and cons

k-nearest neighbor classification – pros and cons

Pros:

k-nearest neighbor classification – pros and cons

Pros:

- Easy to understand, implement and results are highly interpretable.

k-nearest neighbor classification – pros and cons

Pros:

- Easy to understand, implement and results are highly interpretable.
- Non-parameteric

k-nearest neighbor classification – pros and cons

Pros:

- Easy to understand, implement and results are highly interpretable.
- Non-parameteric
- Works reliably even with small data sets.

k-nearest neighbor classification – pros and cons

Pros:

- Easy to understand, implement and results are highly interpretable.
- Non-parameteric
- Works reliably even with small data sets.

Cons:

k-nearest neighbor classification – pros and cons

Pros:

- Easy to understand, implement and results are highly interpretable.
- Non-parameteric
- Works reliably even with small data sets.

Cons:

- Calculating distances computationally intensive for large data sets.

k-nearest neighbor classification – pros and cons

Pros:

- Easy to understand, implement and results are highly interpretable.
- Non-parameteric
- Works reliably even with small data sets.

Cons:

- Calculating distances computationally intensive for large data sets.
- Performs poor on sparse data sets; prone to the **curse of dimensionality.**

k-nearest neighbor classification – pros and cons

Pros:

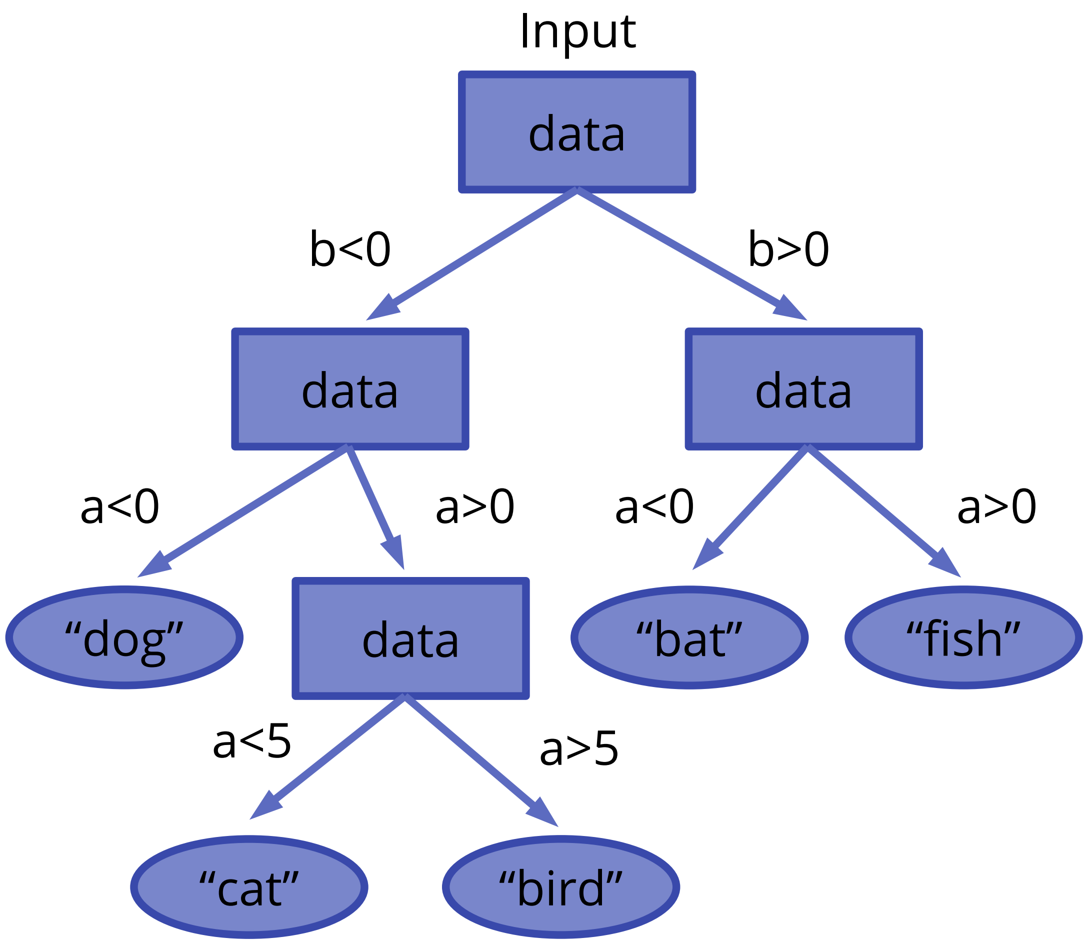
- Easy to understand, implement and results are highly interpretable.
- Non-parameteric
- Works reliably even with small data sets.

Cons:

- Calculating distances computationally intensive for large data sets.
- Performs poor on sparse data sets; prone to the **curse of dimensionality.**

> Number of data points should grow exponentially with data dimensionality.
>
> If parameter space is insufficiently sampled, the model does not have enough data points for training properly.

Tree-based models
(a high-level introduction)



Decision tree classification

A decision tree is a **rule-based structure** for prediction of scalar output from (potentially) multi-dimensional input data.

Tree properties (hyperparameters):

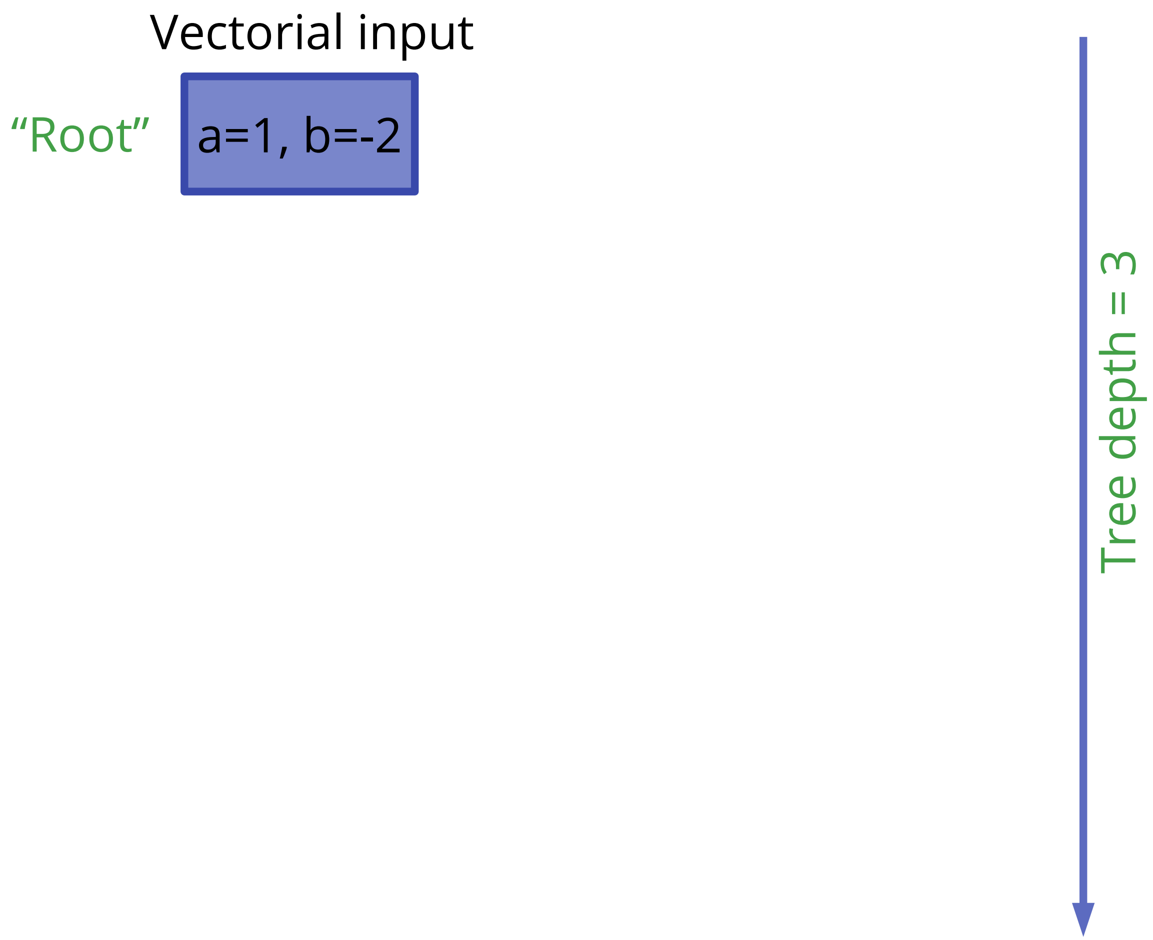
- Tree depth
- Number of leaves

Decision tree classification

A decision tree is a **rule-based structure** for prediction of scalar output from (potentially) multi-dimensional input data.

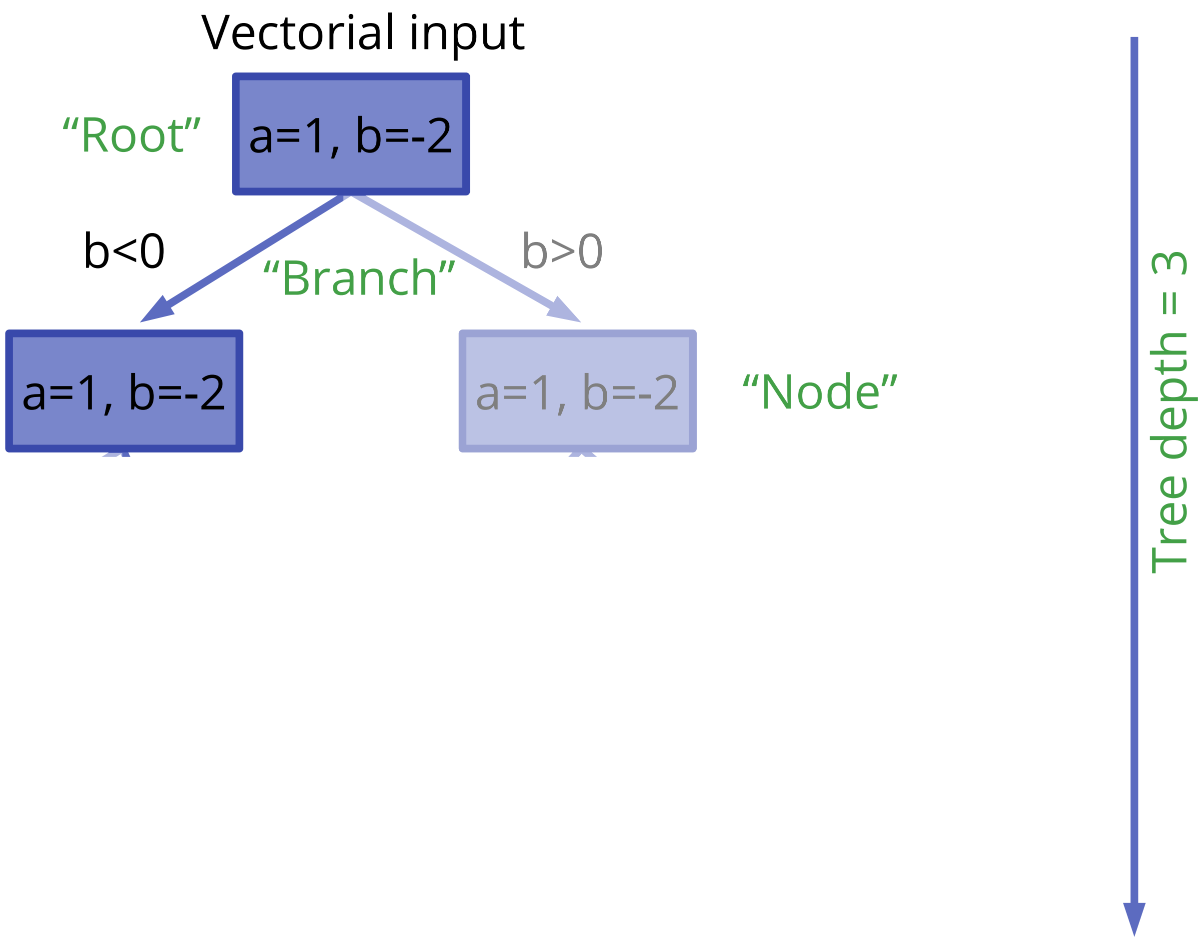
Tree properties (hyperparameters):

- Tree depth
- Number of leaves



Decision tree classification

A decision tree is a **rule-based structure** for prediction of scalar output from (potentially) multi-dimensional input data.



Tree properties (hyperparameters):

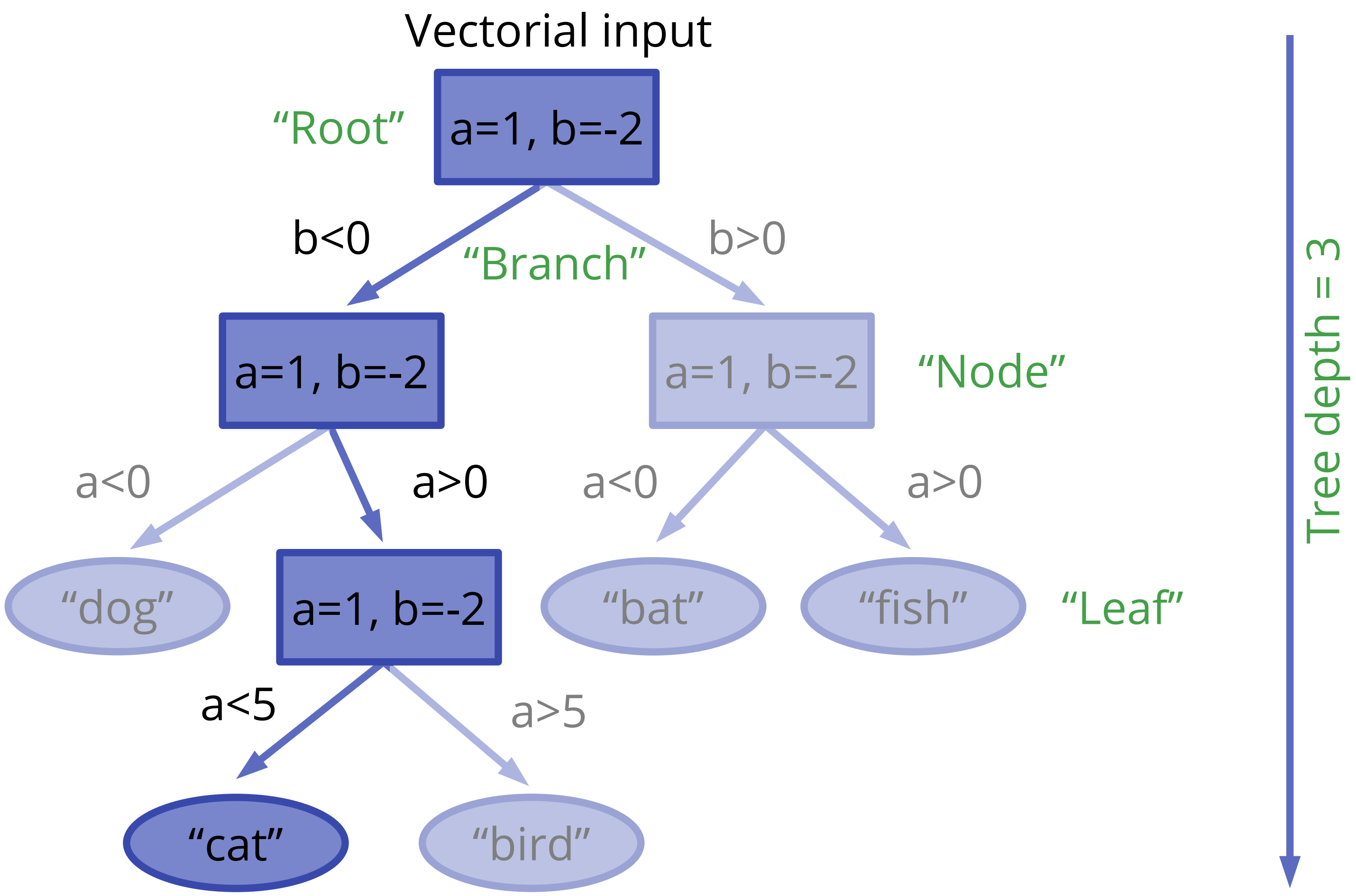
- Tree depth
- Number of leaves

Decision tree classification

A decision tree is a **rule-based structure** for prediction of scalar output from (potentially) multi-dimensional input data.

Tree properties (hyperparameters):

- Tree depth
- Number of leaves

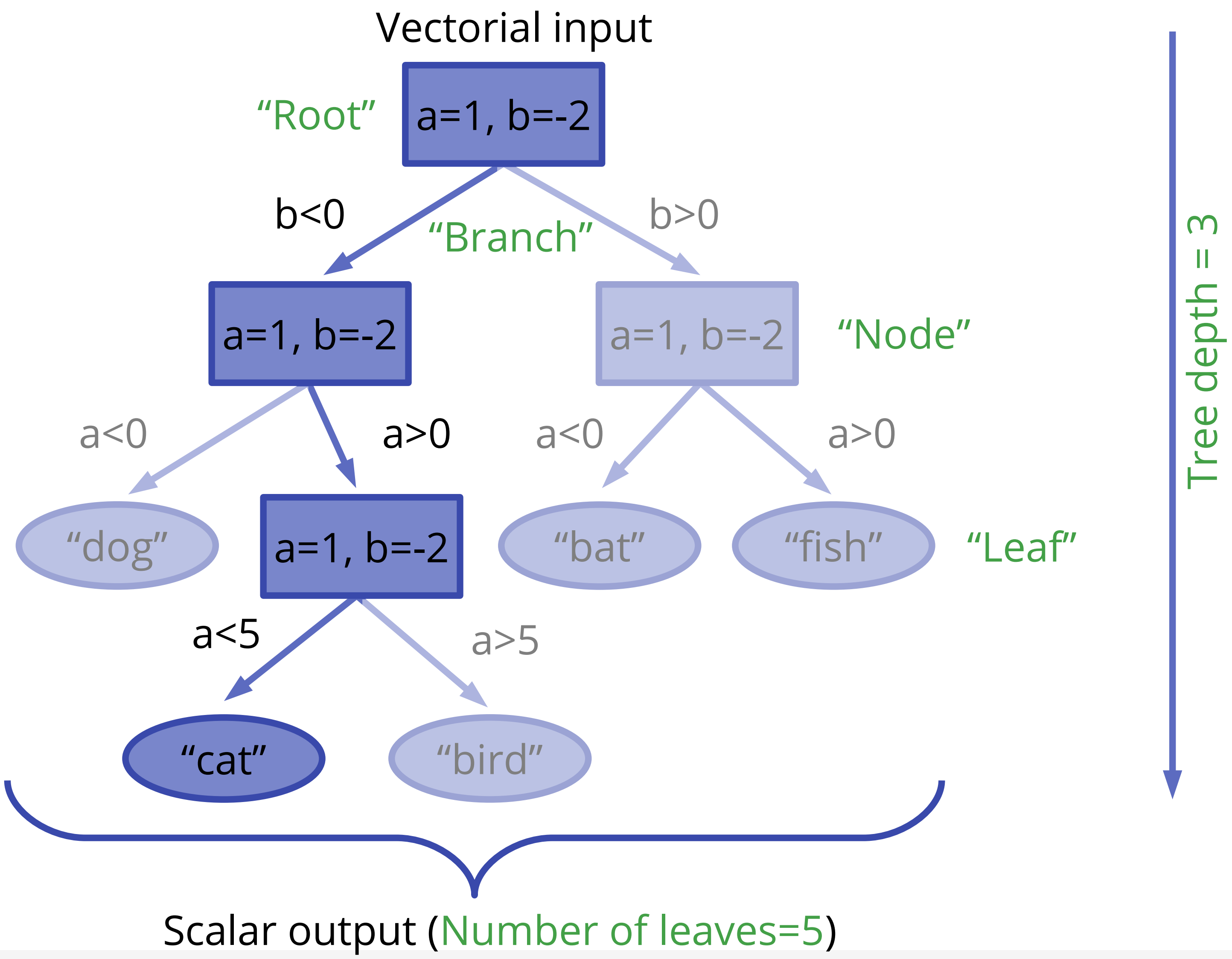


Decision tree classification

A decision tree is a **rule-based structure** for prediction of scalar output from (potentially) multi-dimensional input data.

Tree properties (hyperparameters):

- Tree depth
- Number of leaves



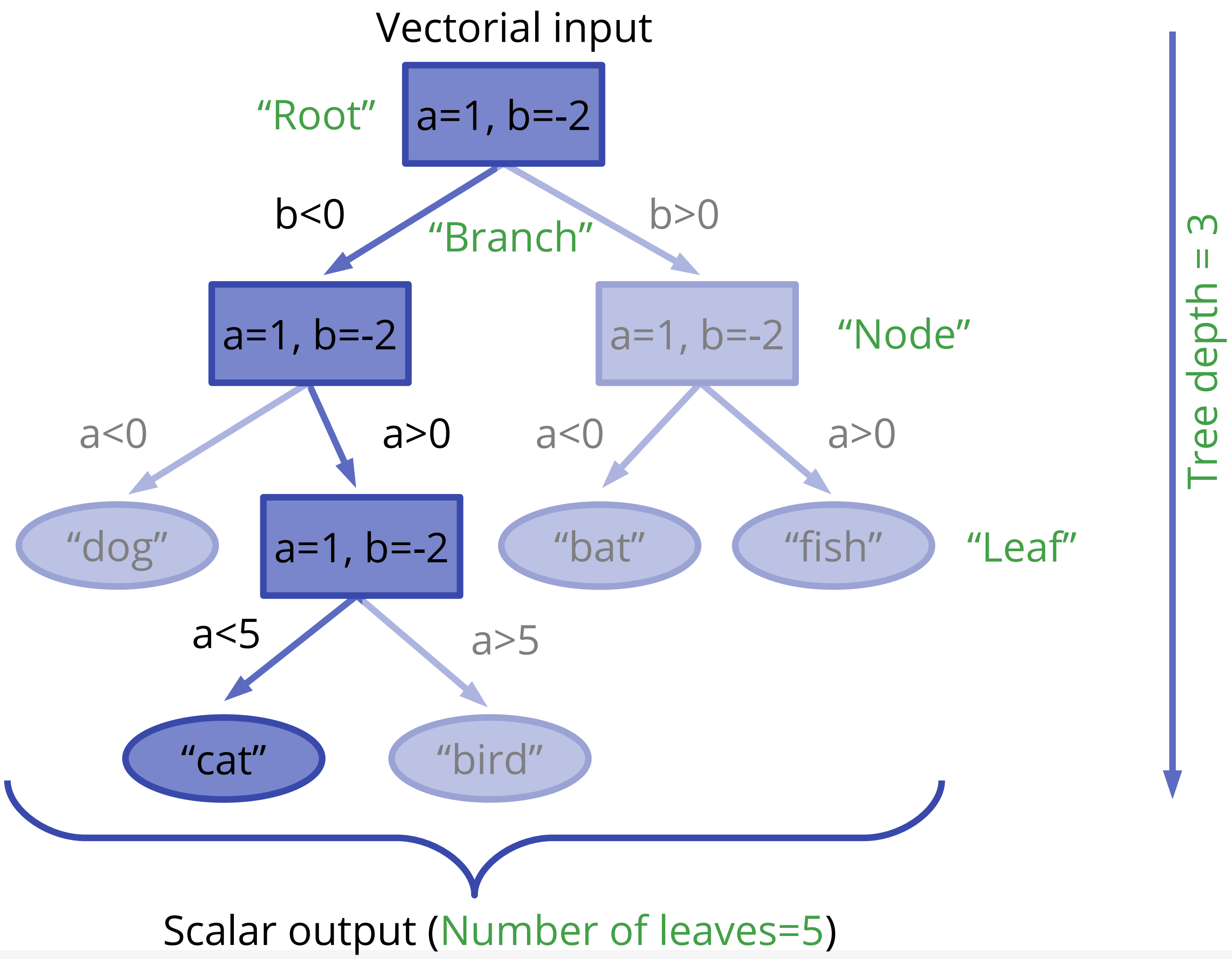
Decision tree classification

A decision tree is a **rule-based structure** for prediction of scalar output from (potentially) multi-dimensional input data.

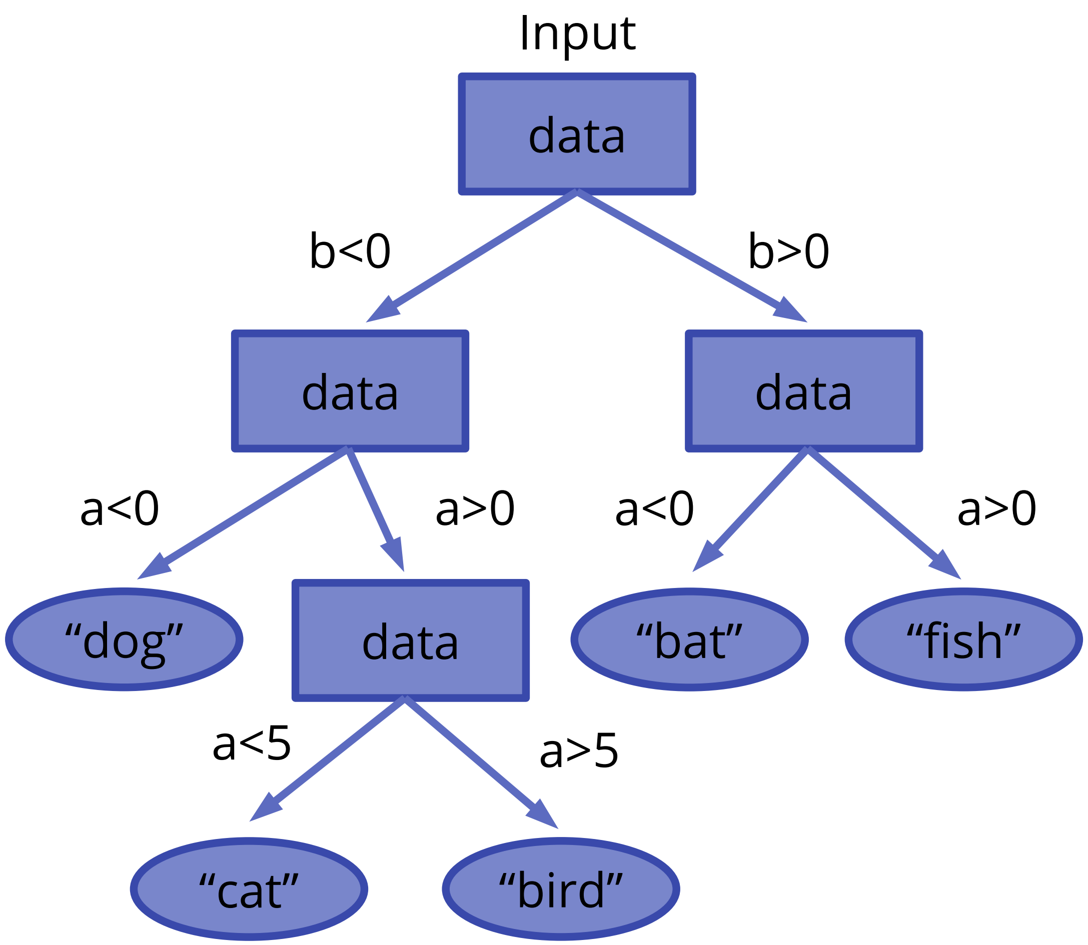
Tree properties (hyperparameters):

- Tree depth
- Number of leaves

How can the rules stored in the nodes be learned?

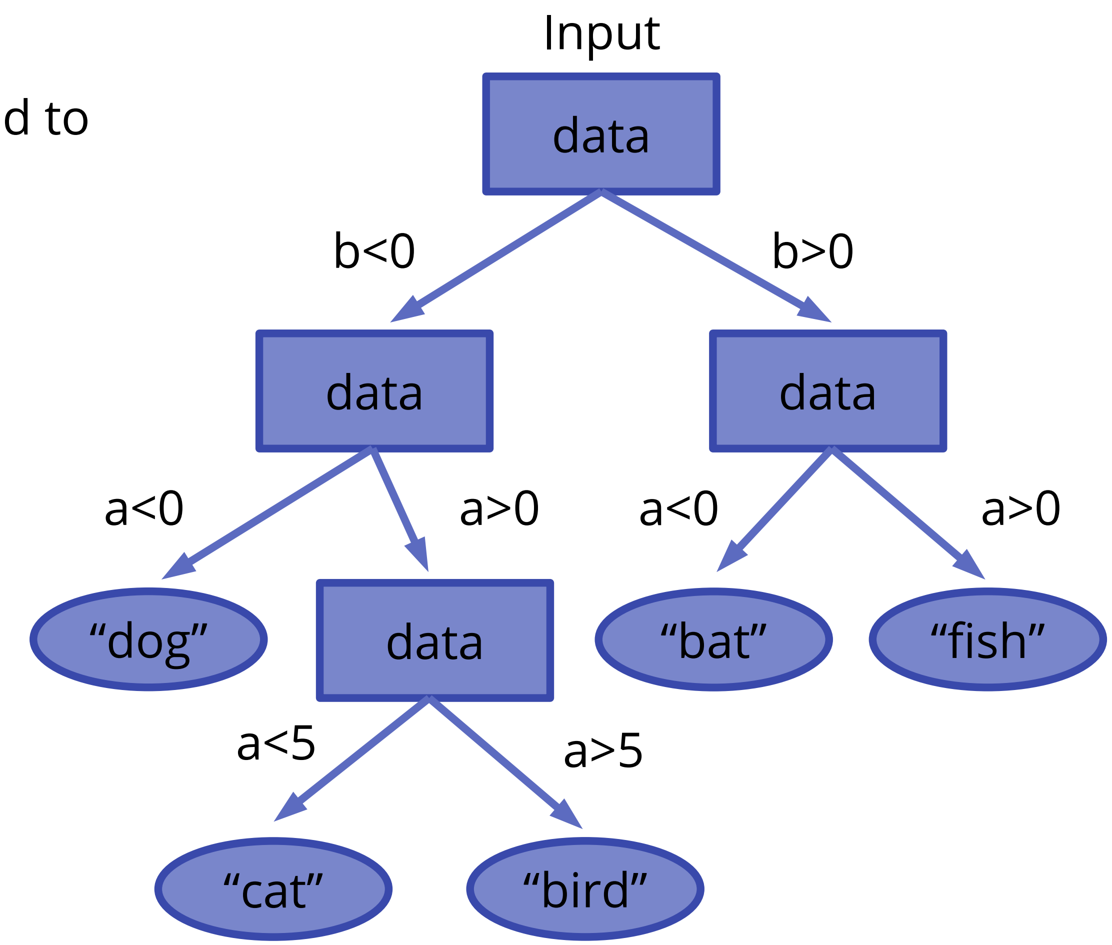


Decision tree learning



Decision tree learning

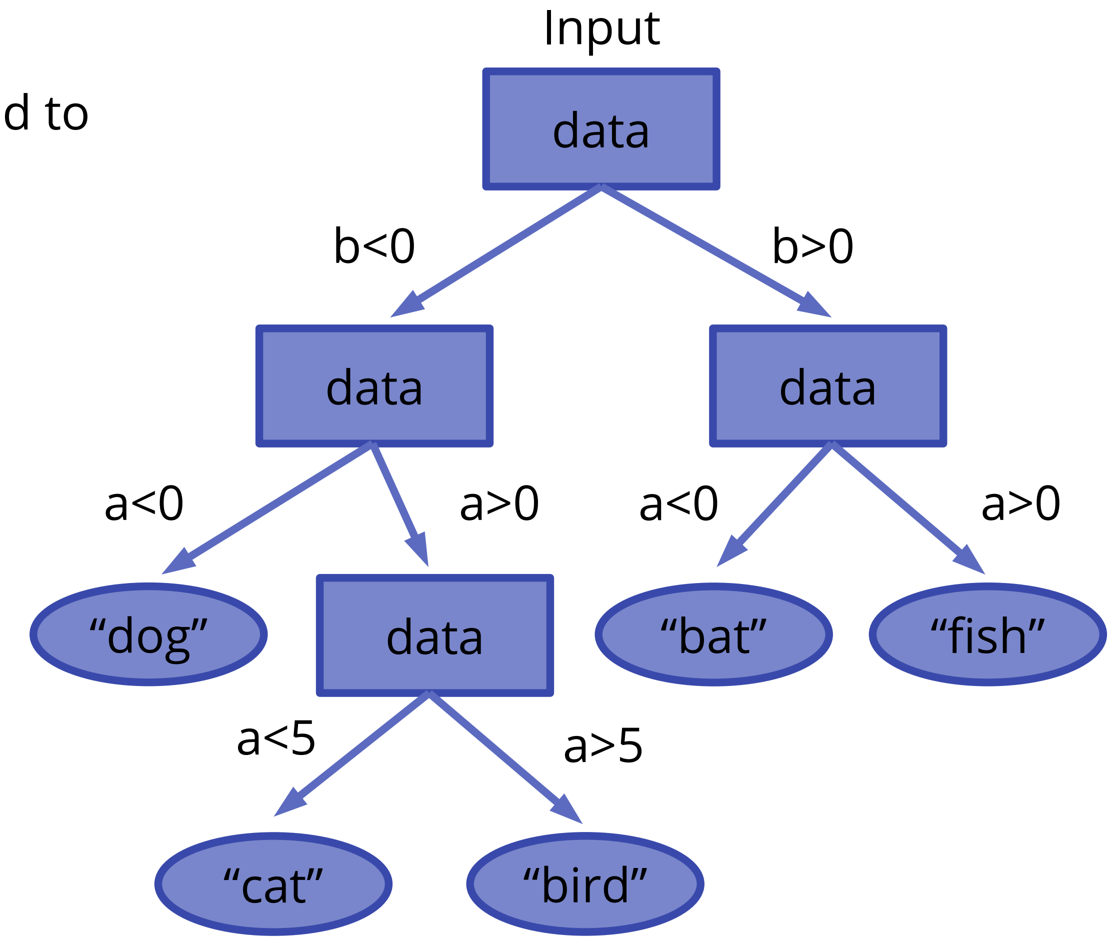
A **greedy divide-and-conquer strategy** is adopted to train decision trees on a training data set in a recursive fashion:



Decision tree learning

A **greedy divide-and-conquer strategy** is adopted to train decision trees on a training data set in a recursive fashion:

1) Identify the "most important feature" (greedy)

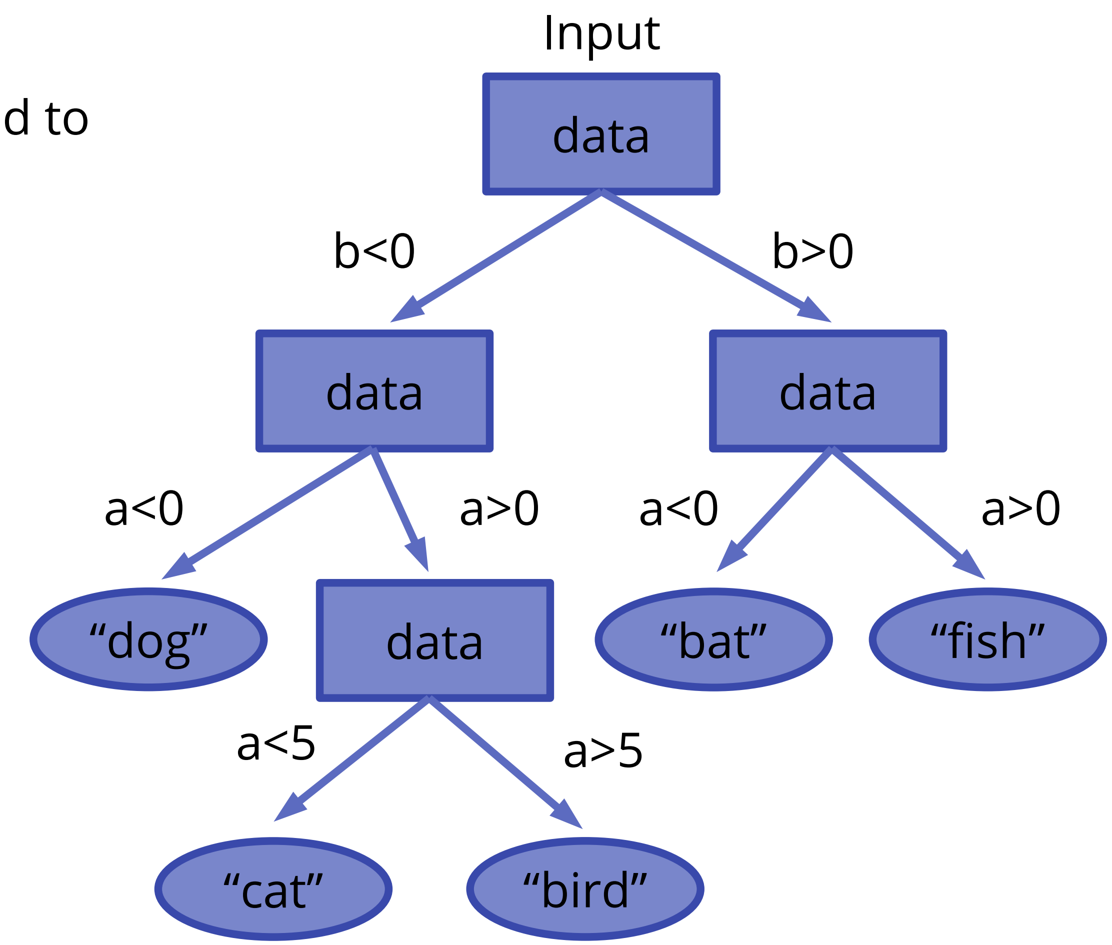


Decision tree learning

A **greedy divide-and-conquer strategy** is adopted to train decision trees on a training data set in a recursive fashion:

1) Identify the "most important feature" (greedy)

2) Split the samples across this feature (divide)



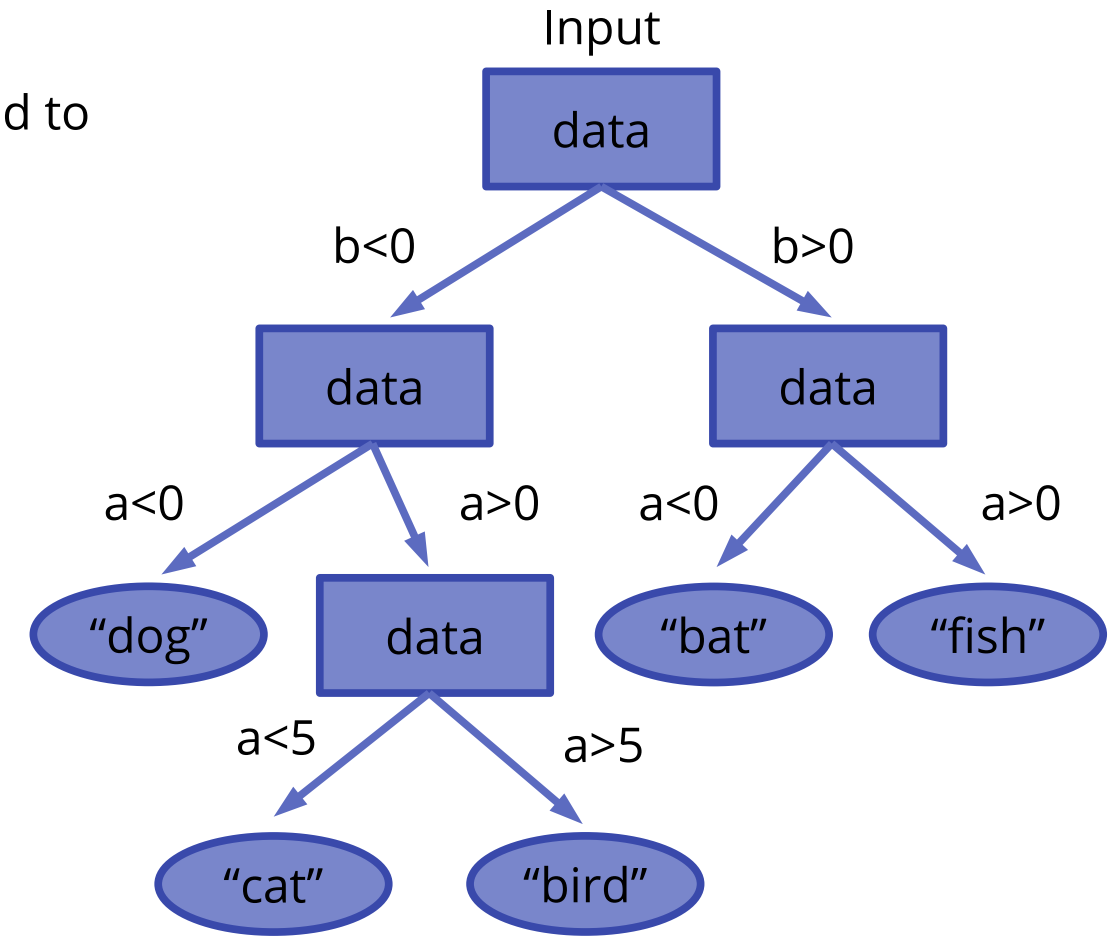
Decision tree learning

A **greedy divide-and-conquer strategy** is adopted to train decision trees on a training data set in a recursive fashion:

1) Identify the “most important feature” (greedy)

2) Split the samples across this feature (divide)

3) If all samples of a branch are of the same class, create a leaf, unless number of leafs has been reached, and stop.



Decision tree learning

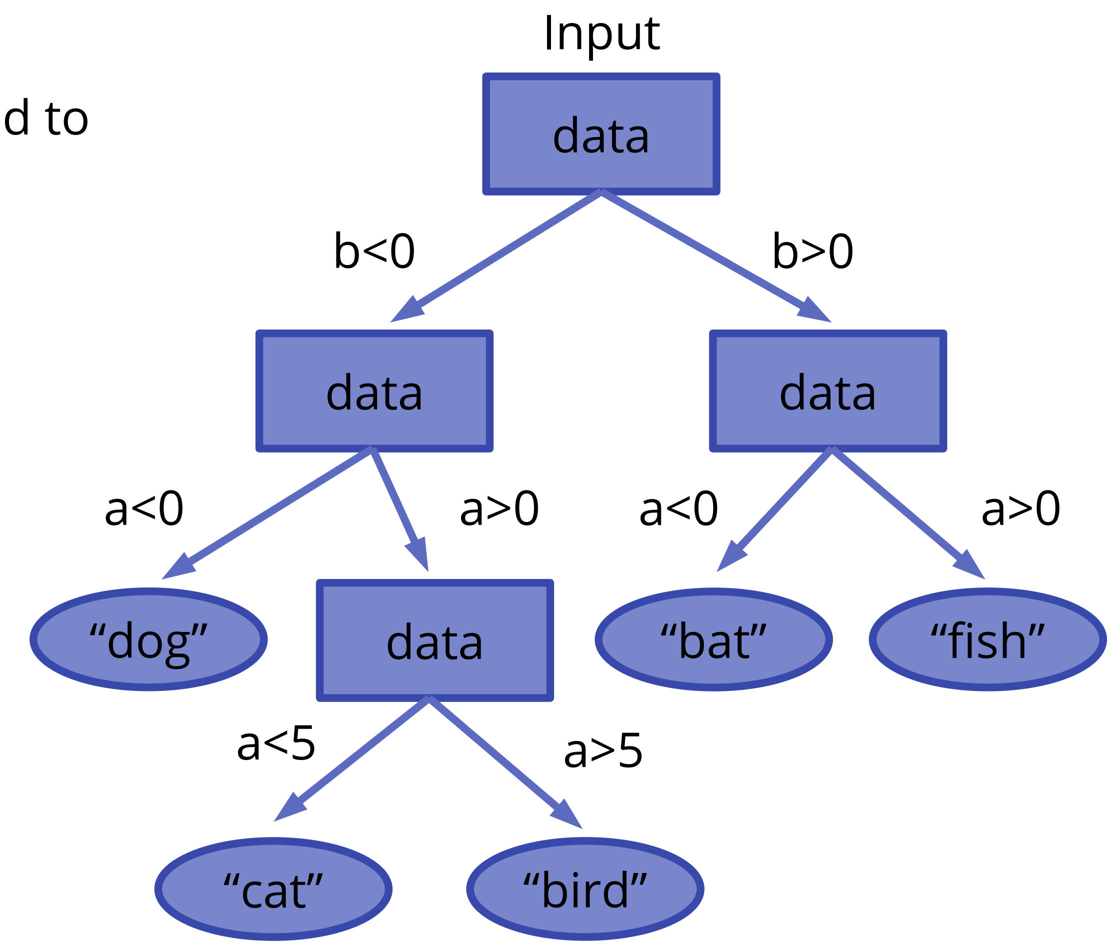
A **greedy divide-and-conquer strategy** is adopted to train decision trees on a training data set in a recursive fashion:

1) Identify the "most important feature" (greedy)

2) Split the samples across this feature (divide)

3) If all samples of a branch are of the same class, create a leaf, unless number of leafs has been reached, and stop.

4) If not all samples of a branch are of the same class, recursively apply algorithm again to that branch until maximum depth reached.



Decision tree learning

A **greedy divide-and-conquer strategy** is adopted to train decision trees on a training data set in a recursive fashion:

1) Identify the “most important feature” (greedy)
2) Split the samples across this feature (divide)
3) If all samples of a branch are of the same class, create a leaf, unless number of leafs has been reached, and stop.
4) If not all samples of a branch are of the same class, recursively apply algorithm again to that branch until maximum depth reached.

But what is the “most important feature”?

Generally, it means that feature that makes the most difference to the classification of a single sample.

There different implementation of this definition, e.g., utilizing **information entropy** or other useful measures.

Random forests - decision trees as “weak learners”

Random forests - decision trees as “weak learners”

Single decision trees typically generalize only to some extent due to their limited depth and size; they have **low model capacity**.

Random forests - decision trees as “weak learners”

Single decision trees typically generalize only to some extent due to their limited depth and size; they have **low model capacity**.

Ensemble methods help trees to perform much better: by combining a large number of decision trees and letting them make decisions in an averaged vote or majority vote, thereby **increasing capacity**.

Random forests - decision trees as "weak learners"

Single decision trees typically generalize only to some extent due to their limited depth and size; they have **low model capacity**.

Ensemble methods help trees to perform much better: by combining a large number of decision trees and letting them make decisions in an averaged vote or majority vote, thereby **increasing capacity**.

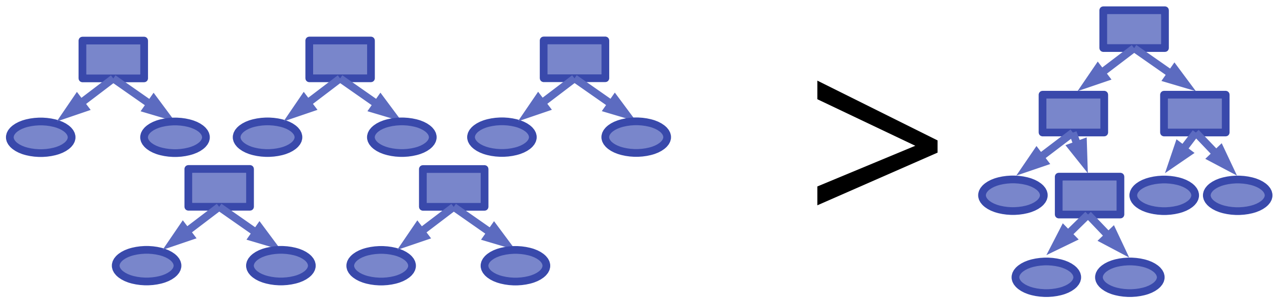
Trees in a **random forest** are shallower than other decision tree models. The trees therefore act as "**weak learners**" that perform badly by themselves. However, combining a large number of weak learners performs much better than individual trees. The intuition behind is that weak learners "on average" compensate for their individual shortcomings.

Random forests - decision trees as “weak learners”

Single decision trees typically generalize only to some extent due to their limited depth and size; they have **low model capacity**.

Ensemble methods help trees to perform much better: by combining a large number of decision trees and letting them make decisions in an averaged vote or majority vote, thereby **increasing capacity**.

Trees in a **random forest** are shallower than other decision tree models. The trees therefore act as “**weak learners**” that perform badly by themselves. However, combining a large number of weak learners performs much better than individual trees. The intuition behind is that weak learners “on average” compensate for their individual shortcomings.



Gradient-boosted tree-based models

Gradient-boosted tree-based models are random forests (decision tree ensembles) that are built successively in such a way that *every newly created tree compensates for the shortcomings of the previous trees*.

The term **gradient-boosting** refers to the fact that new base learners (individual decision trees) are fitted to the model's pseudo-residuals, based on the gradient of the loss of the ensemble:

Gradient-boosted tree-based models

Gradient-boosted tree-based models are random forests (decision tree ensembles) that are built successively in such a way that *every newly created tree compensates for the shortcomings of the previous trees*.

The term **gradient-boosting** refers to the fact that new base learners (individual decision trees) are fitted to the model's pseudo-residuals, based on the gradient of the loss of the ensemble:

$$f(\mathbf{x}) = \sum_{m}^{M} \beta_m h(\mathbf{x}, \theta_m)$$

Ensemble model with learning rate β_m, base learners $h(\mathbf{x}, \theta_m)$ with parameters θ_m.

$$\mathbf{r}_m^i = -\left[\frac{\partial L(y^i, f(x^i))}{\partial f(x^i)}\right]_{f=f_{m-1}}$$

Pseudo-residuals to which the next base learner will be fitted; by default, the loss of the updated ensemble will be less or equal to the loss of the current ensemble.

Gradient-boosted tree-based models

Gradient-boosted models are very successful in regression and classification tasks and still represent state-of-the-art in traditional ML.

If you have a classification or regression problem, it is always worth trying out gradient-boosted methods.

Common implementations:

- XGBoost
- LightGBM

Tree-based models – pros and cons

Tree-based models – pros and cons

Pros:

Tree-based models – pros and cons

Pros:

- Extremely versatile and robust.

Tree-based models – pros and cons

Pros:

- Extremely versatile and robust.
- Can be trained on small amounts of data

Tree-based models – pros and cons

Pros:

- Extremely versatile and robust.
- Can be trained on small amounts of data
- Non-parametric

Tree-based models – pros and cons

Pros:

- Extremely versatile and robust.
- Can be trained on small amounts of data
- Non-parametric
- Interpretability: tree-based models are able to compute “**feature importances**”

Tree-based models – pros and cons

Pros:

- Extremely versatile and robust.
- Can be trained on small amounts of data
- Non-parametric
- Interpretability: tree-based models are able to compute “**feature importances**”

Cons:

Tree-based models – pros and cons

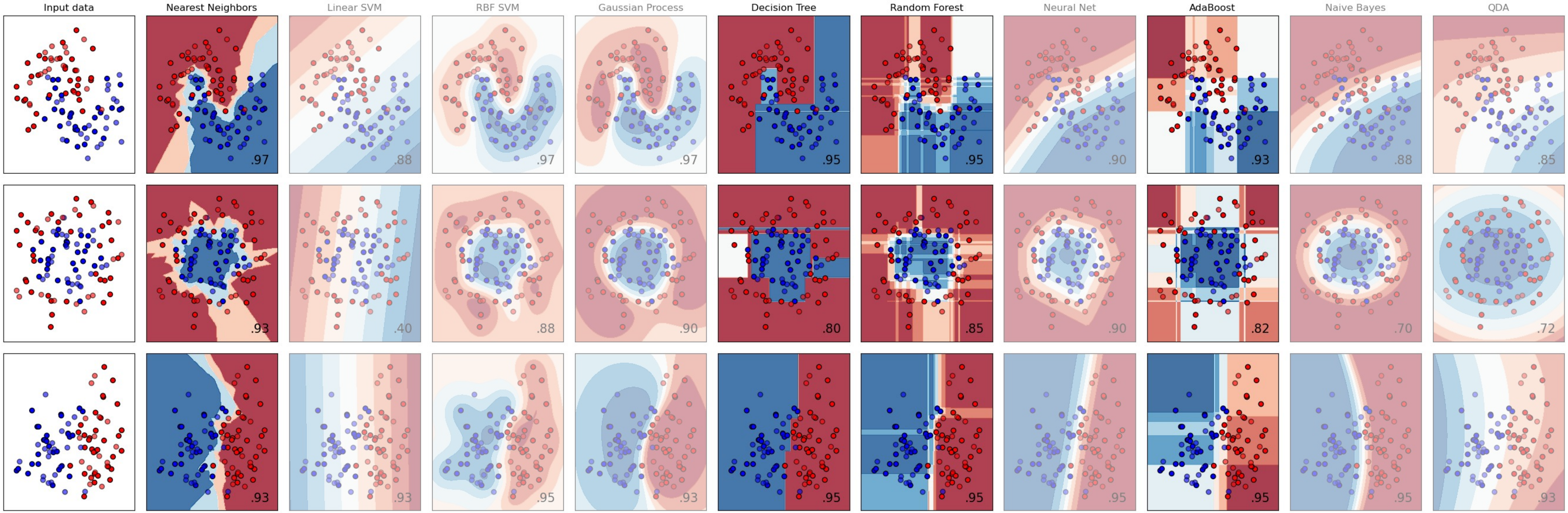
Pros:

- Extremely versatile and robust.
- Can be trained on small amounts of data
- Non-parametric
- Interpretability: tree-based models are able to compute “**feature importances**”

Cons:

- Decision boundaries and regression predictions may be discrete instead of continuous (see next slide)

Supervised learning - summary



(Similar to XGBoost)

Scikit-learn classifier comparison

That’s all folks!