

Linear Regression

Linear model, gradient descent, feature expansion

Programming and Statistical Analysis, 2025

Majid Sohrabi

National Research University Higher School of Economics



MMCP

April 25, 2025

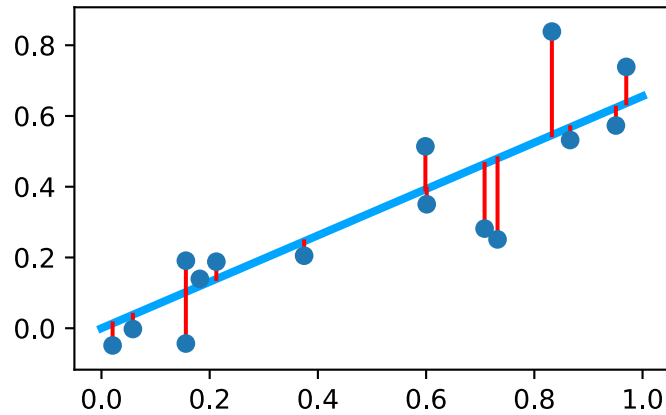
Why study linear models?



Linear models in a nutshell

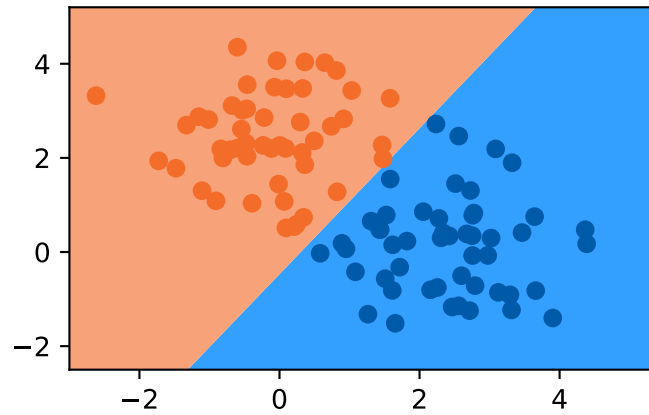
Regression:

$$\hat{f}(x) = \theta^T x$$



Classification:

$$\hat{f}(x) = \mathbb{I}[\theta^T x > 0]$$

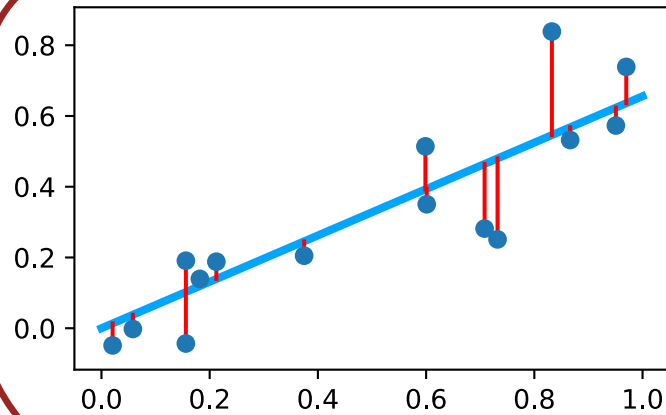


**Outputs linear in
inputs**

Linear models in a nutshell

Regression:

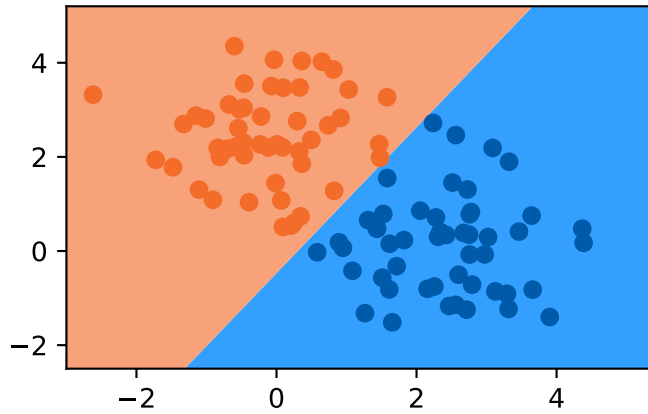
$$\hat{f}(x) = \theta^T x$$



Targets $\in \mathbb{R}$ (or even \mathbb{R}^m in the multidimensional case)

Classification:

$$\hat{f}(x) = \mathbb{I}[\theta^T x > 0]$$

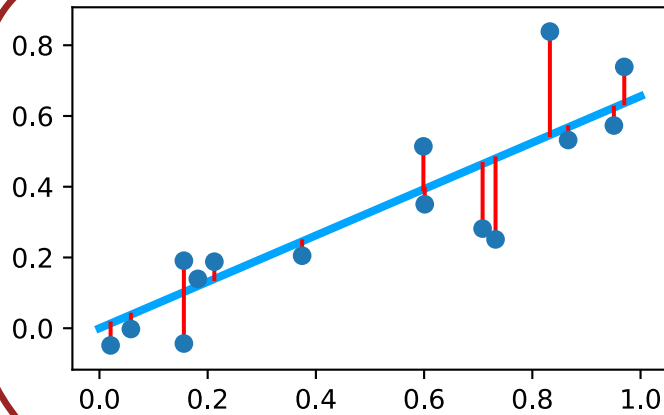


Outputs linear in inputs

Linear models in a nutshell

Regression:

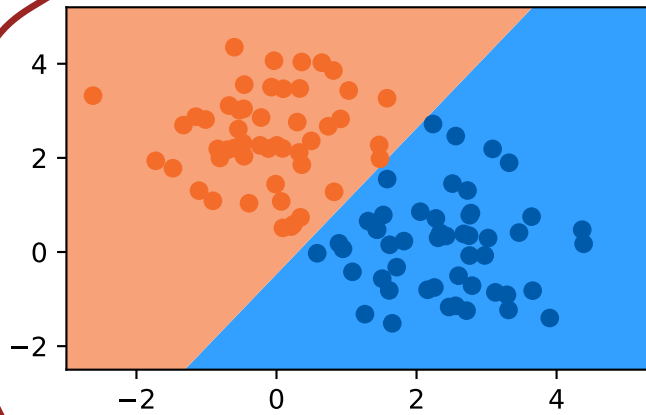
$$\hat{f}(x) = \theta^T x$$



Targets $\in \mathbb{R}$ (or even \mathbb{R}^m in the multidimensional case)

Classification:

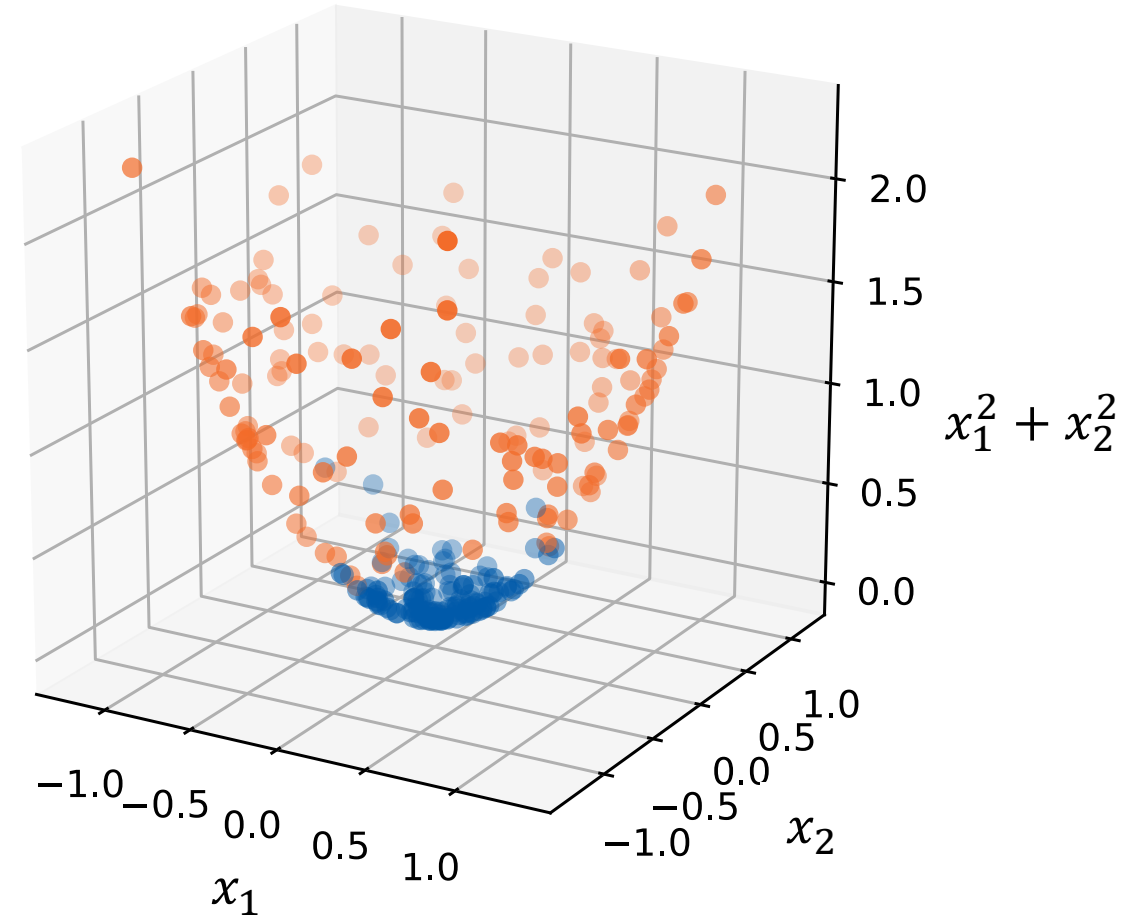
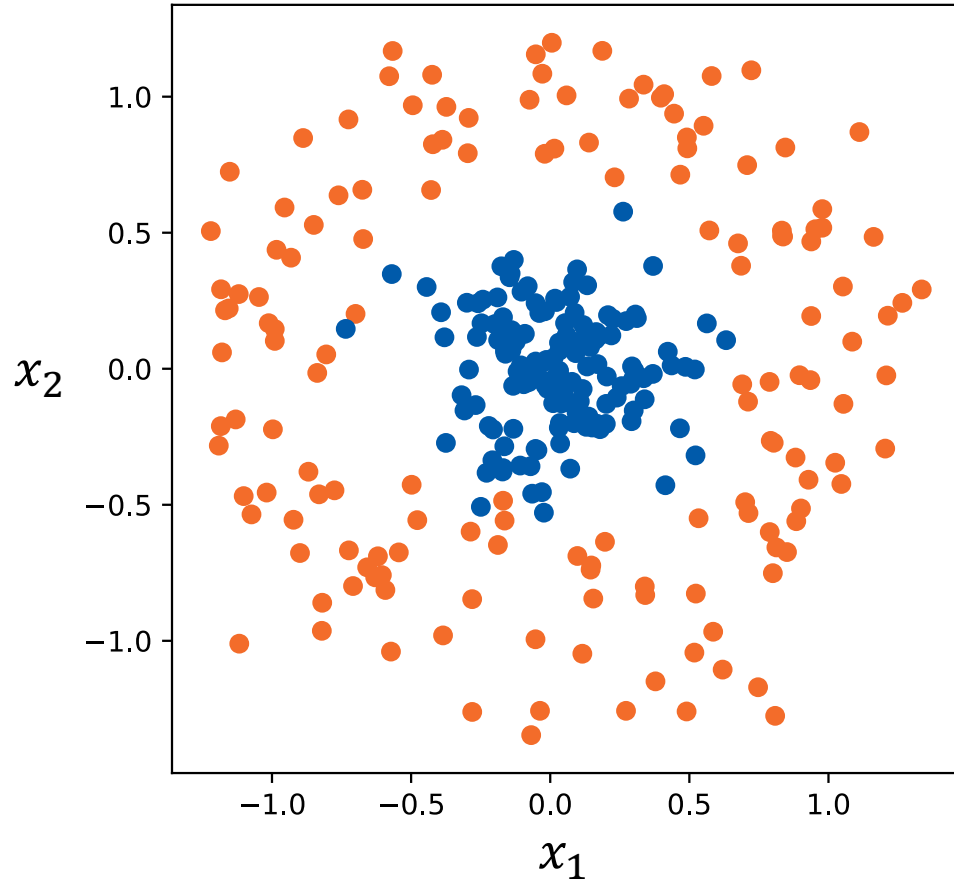
$$\hat{f}(x) = \mathbb{I}[\theta^T x > 0]$$



Targets \in some finite set

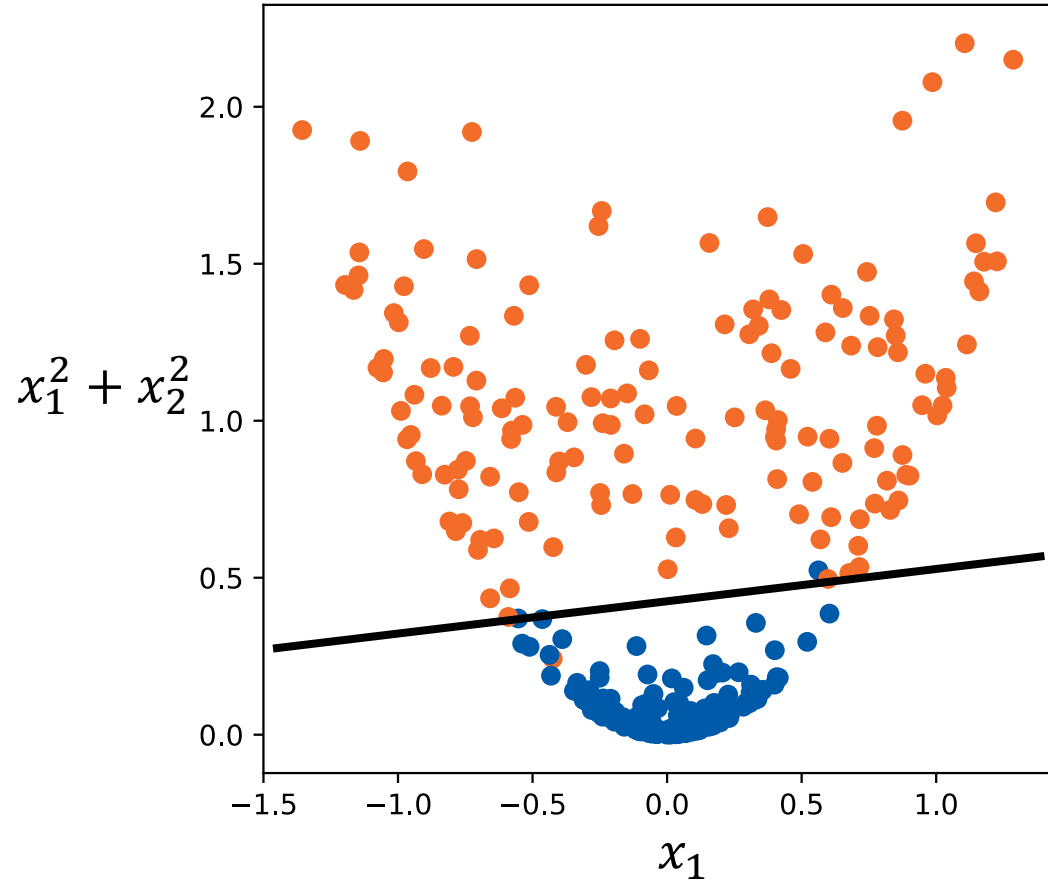
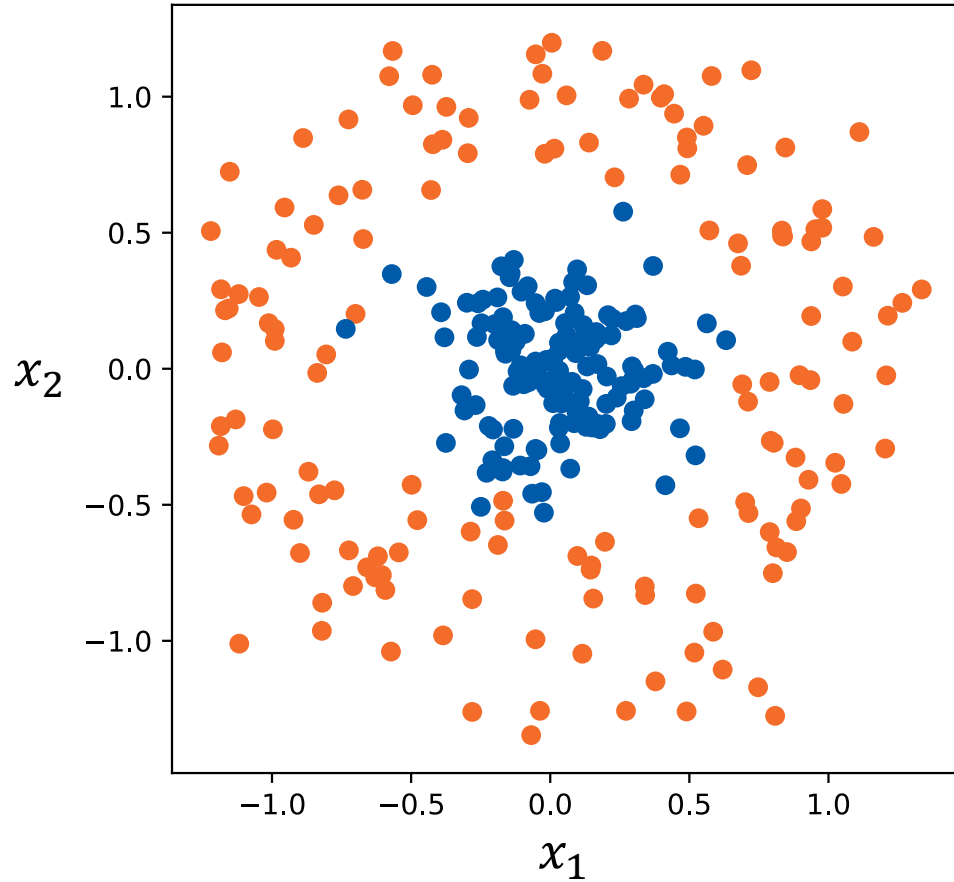
Outputs linear in inputs

The hidden power



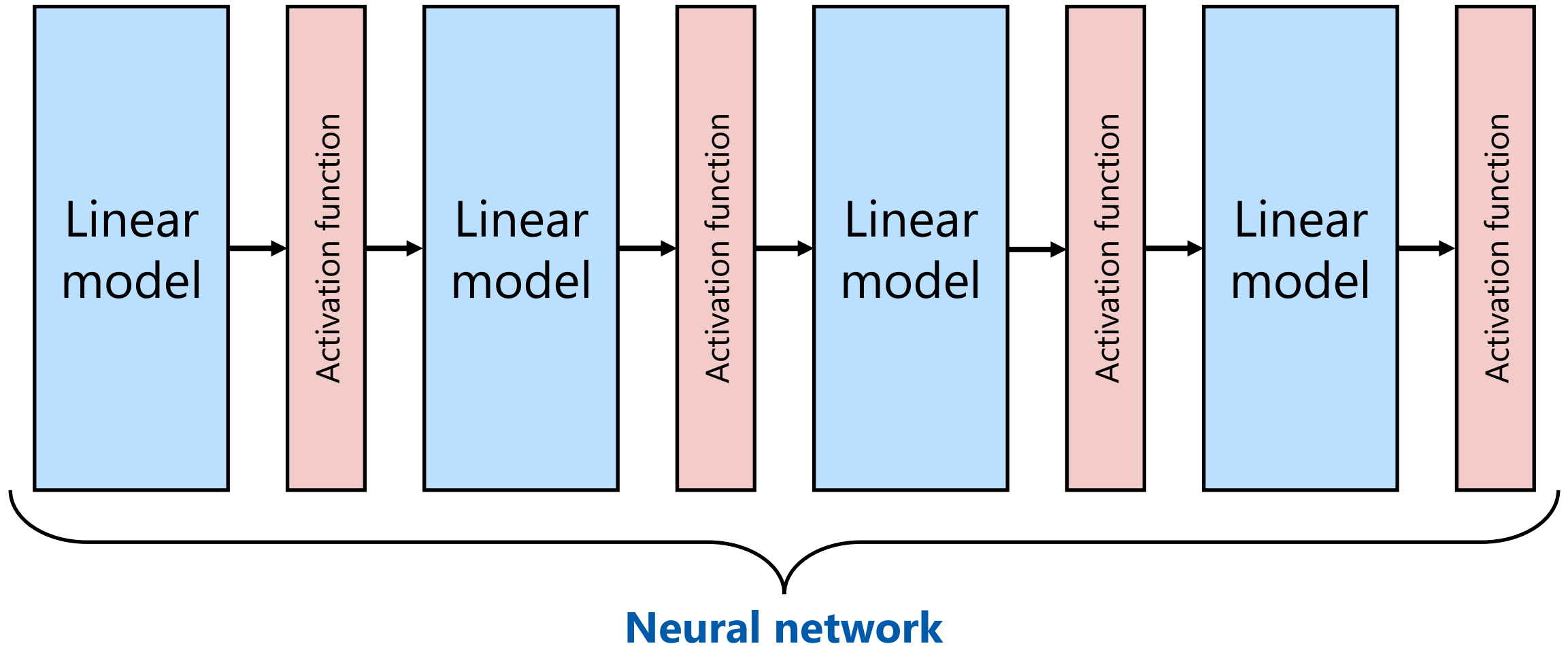
Linearly **inseparable** \rightarrow **separable** by **transforming the features**

The hidden power



Linearly **inseparable** \rightarrow **separable** by **transforming the features**

Building block for deep models



Better intuition for deep neural networks training

Interpretability

One can take a look at the weights corresponding to each of the features separately, e.g.:

Note: this example is not based on real data, but rather taken from the lecturer's imagination

$$\text{PredictedIncome} = 1000 + 10 \times \text{Age} + 1000 \times \text{HasHigherEducation} - 0.1 \times \text{DistanceFromCapital}$$

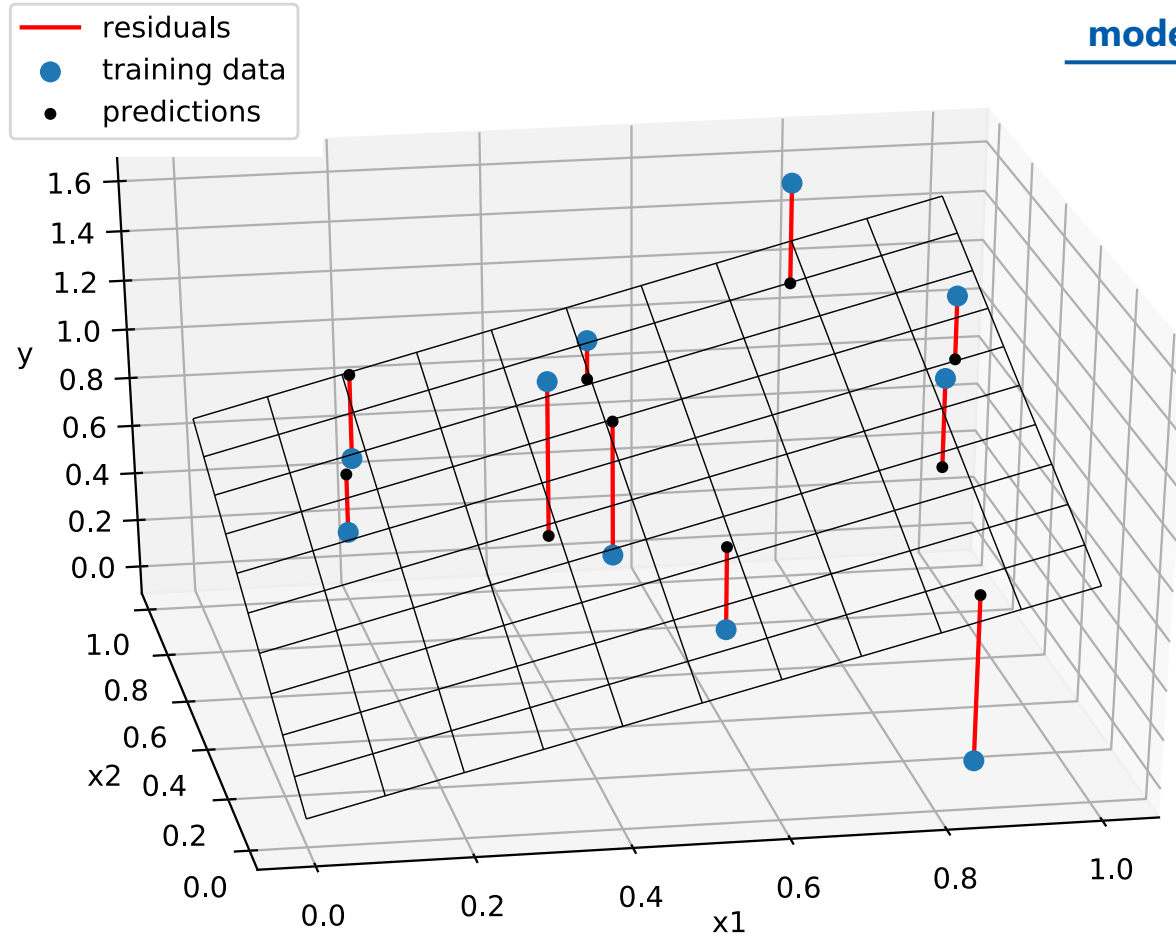
Higher weight means more impact on the prediction

- Note: features need to be of same scale if you want to compare their weights

Linear Regression



Linear Regression model



model prediction

$$\hat{f}_{\theta}(x) = \theta^T x$$

parameters vector

$$\theta \in \mathbb{R}^d$$

features vector

$$x \in \mathcal{X} \subset \mathbb{R}^d$$

$$\frac{1}{N} \sum_{i=1 \dots N} \left(y_i - \hat{f}_{\theta}(x_i) \right)^2 \xrightarrow{\theta} \min$$

**Mean Squared Error
(MSE loss)**

Common loss functions

Mean squared error (MSE):

$$\frac{1}{N} \sum_{i=1 \dots N} \left(y_i - \hat{f}_{\theta}(x_i) \right)^2$$

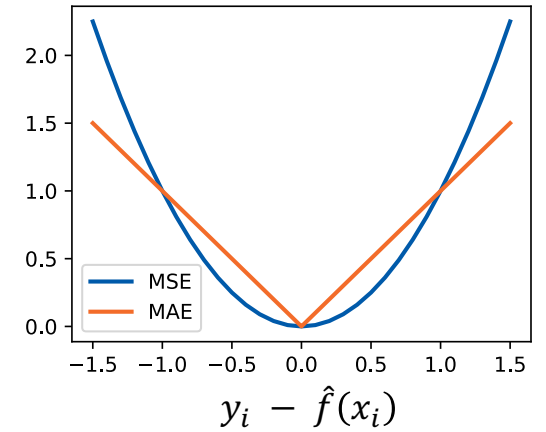
Common loss functions

Mean squared error (MSE):

$$\frac{1}{N} \sum_{i=1 \dots N} \left(y_i - \hat{f}_{\theta}(x_i) \right)^2$$

Mean absolute error (MAE):

$$\frac{1}{N} \sum_{i=1 \dots N} |y_i - \hat{f}_{\theta}(x_i)|$$



Common loss functions

Mean squared error (MSE):

$$\frac{1}{N} \sum_{i=1 \dots N} \left(y_i - \hat{f}_{\theta}(x_i) \right)^2$$

Mean absolute error (MAE):

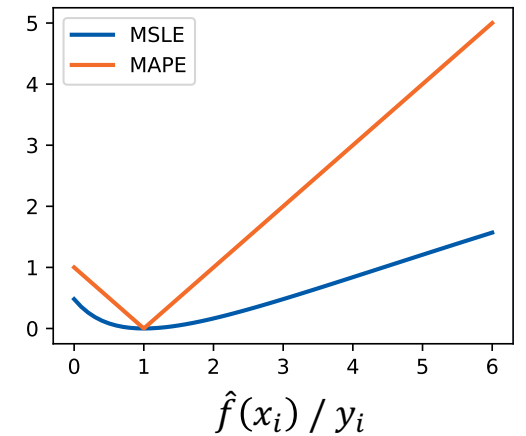
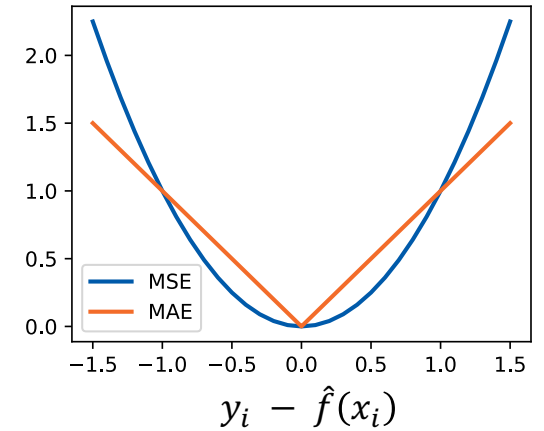
$$\frac{1}{N} \sum_{i=1 \dots N} |y_i - \hat{f}_{\theta}(x_i)|$$

Mean absolute percentage error (MAPE):

$$\frac{1}{N} \sum_{i=1 \dots N} \left| \frac{y_i - \hat{f}_{\theta}(x_i)}{y_i} \right|$$

Mean squared logarithmic error (MSLE):

$$\frac{1}{N} \sum_{i=1 \dots N} \left(\log(y_i + 1) - \log(\hat{f}_{\theta}(x_i) + 1) \right)^2$$



Common loss functions

Mean squared error (MSE):

$$\frac{1}{N} \sum_{i=1 \dots N} \left(y_i - \hat{f}_{\theta}(x_i) \right)^2$$

Mean absolute error (MAE):

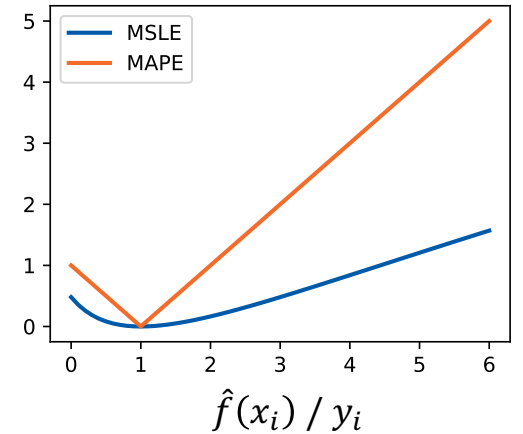
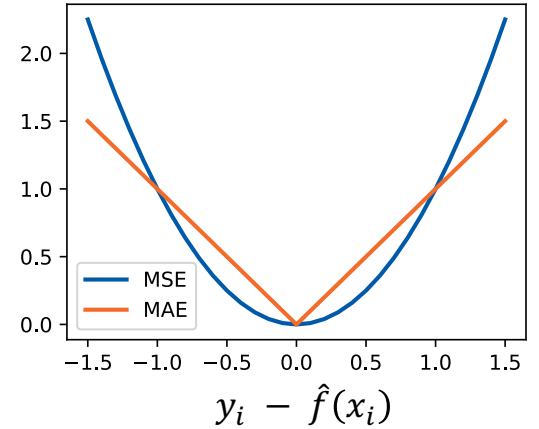
$$\frac{1}{N} \sum_{i=1 \dots N} |y_i - \hat{f}_{\theta}(x_i)|$$

Mean absolute percentage error (MAPE):

$$\frac{1}{N} \sum_{i=1 \dots N} \left| \frac{y_i - \hat{f}_{\theta}(x_i)}{y_i} \right|$$

Mean squared logarithmic error (MSLE):

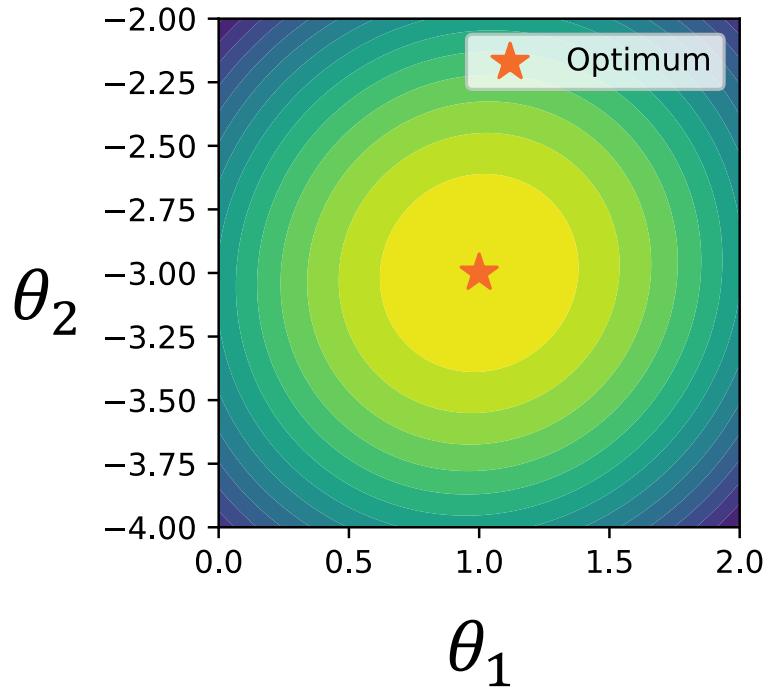
$$\frac{1}{N} \sum_{i=1 \dots N} \left(\log(y_i + 1) - \log(\hat{f}_{\theta}(x_i) + 1) \right)^2$$



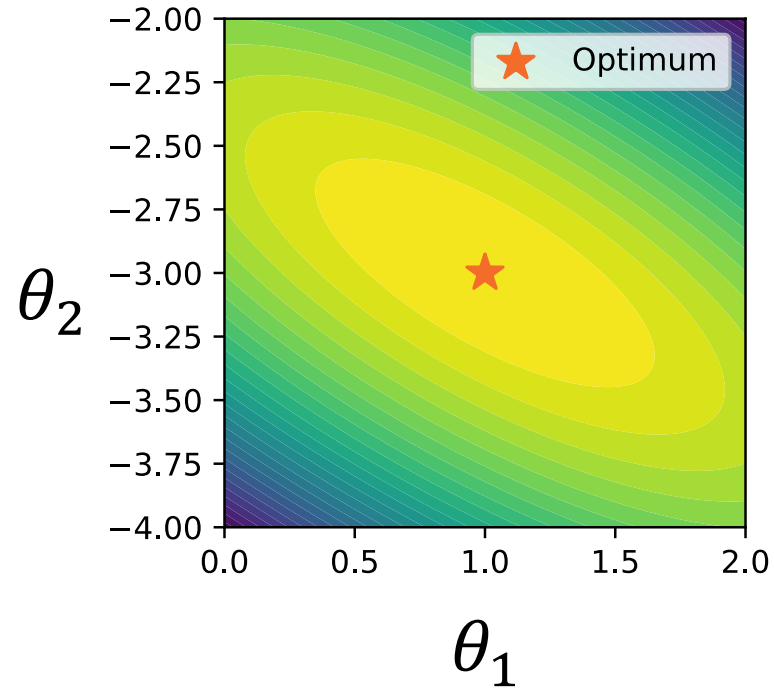
Different **loss functions** also are related to different **assumptions about the data**

Feature correlations matter!

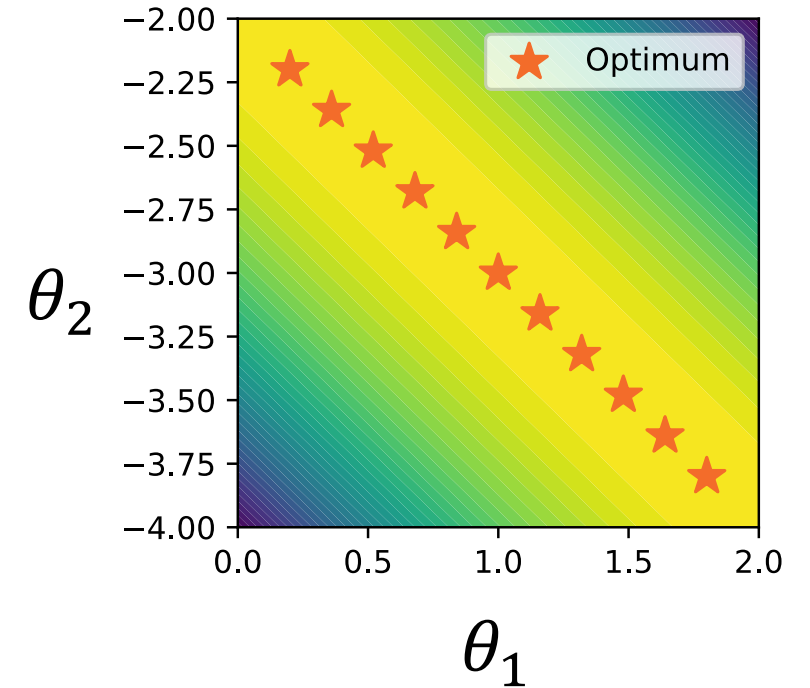
No correlation



50% correlation



100% correlation



MSE level maps

Bias term

a.k.a. intercept term

$$\hat{f}_{\theta}(x) = \theta^T x + \theta_0$$

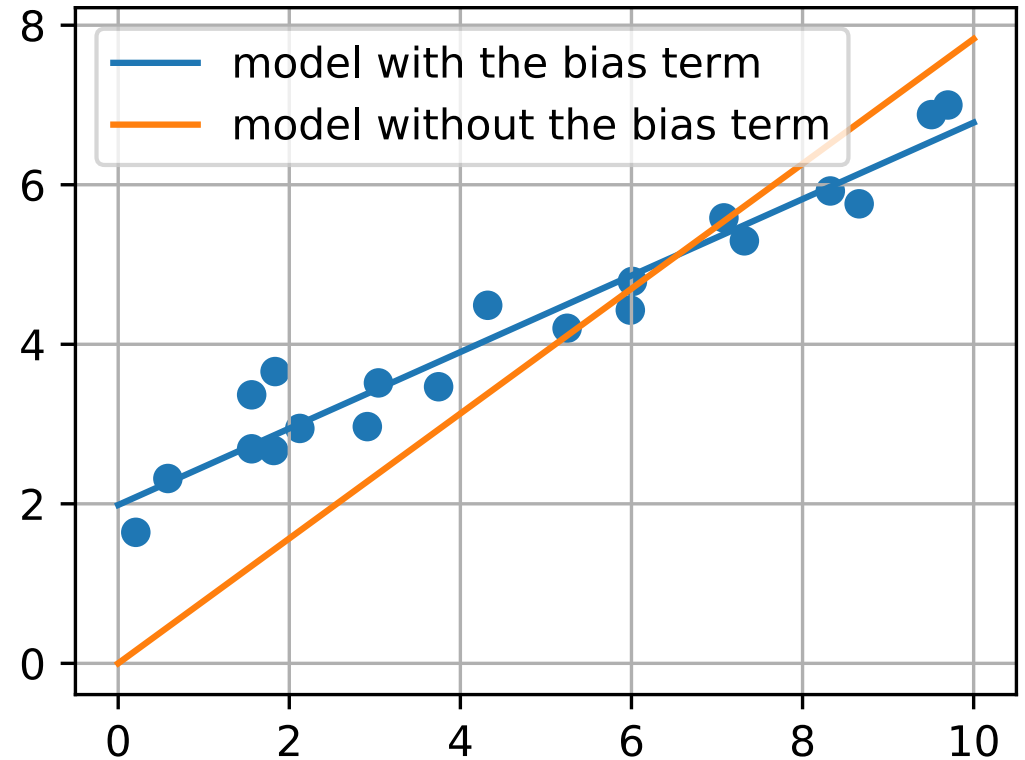
$$\theta \in \mathbb{R}^d$$

$$\theta_0 \in \mathbb{R}$$

$$x \in \mathcal{X} \subset \mathbb{R}^d$$

No need to redo the math – just add a **constant feature** to the design matrix:

$$X = \begin{bmatrix} x_1^1 & x_1^2 & \cdots & x_1^d \\ x_2^1 & x_2^2 & \cdots & x_2^d \\ \vdots & \vdots & \ddots & \vdots \\ x_N^1 & x_N^2 & \cdots & x_N^d \end{bmatrix} \longrightarrow X = \begin{bmatrix} 1 & x_1^1 & x_1^2 & \cdots & x_1^d \\ 1 & x_2^1 & x_2^2 & \cdots & x_2^d \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_N^1 & x_N^2 & \cdots & x_N^d \end{bmatrix}$$



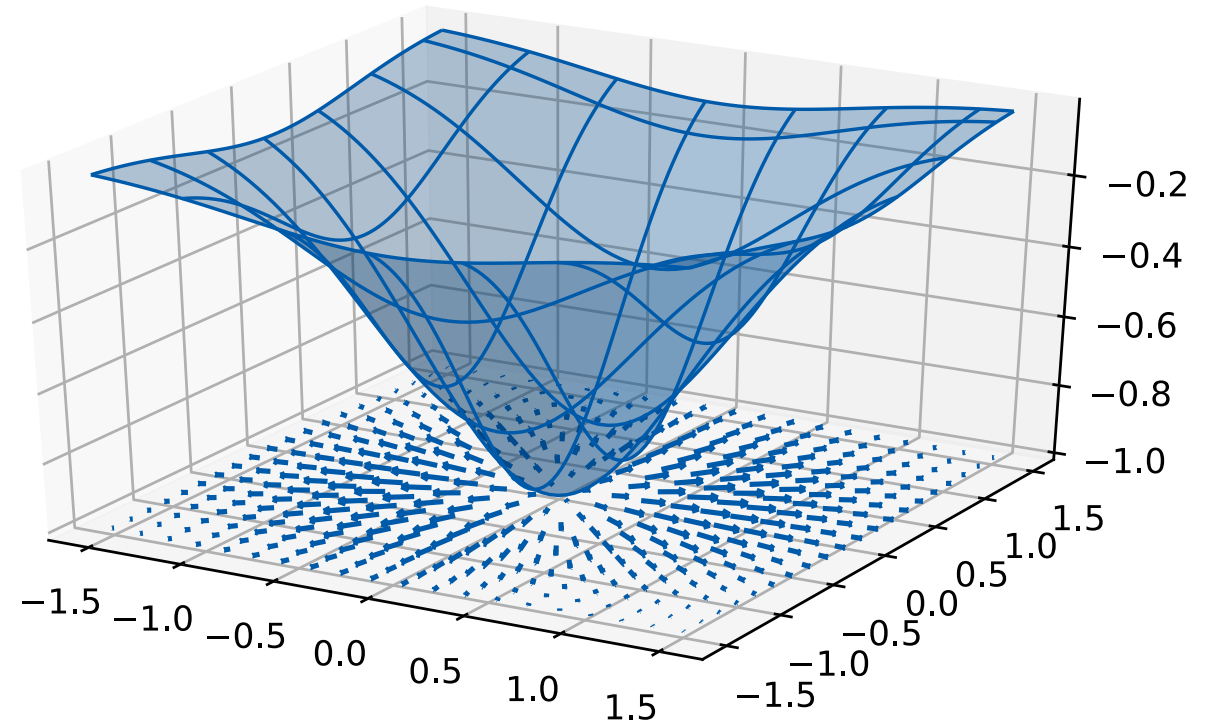
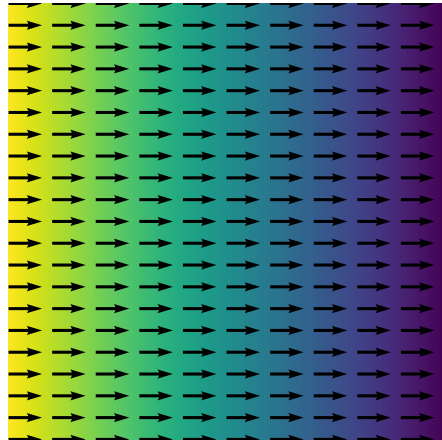
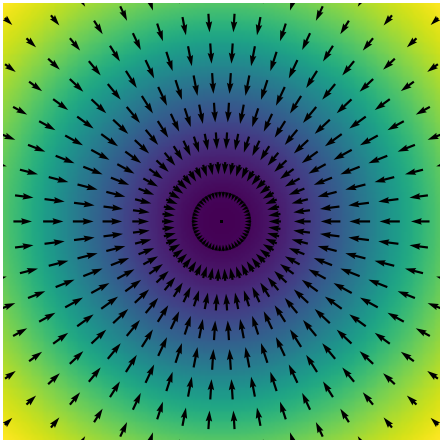
Numerical & Stochastic Optimization



Gradient

Gradient: $\nabla_x f(x) \equiv \left(\frac{\partial f(x)}{\partial x_1}, \dots, \frac{\partial f(x)}{\partial x_d} \right)$

Points towards **steepest function increase**



Gradient Descent Optimization

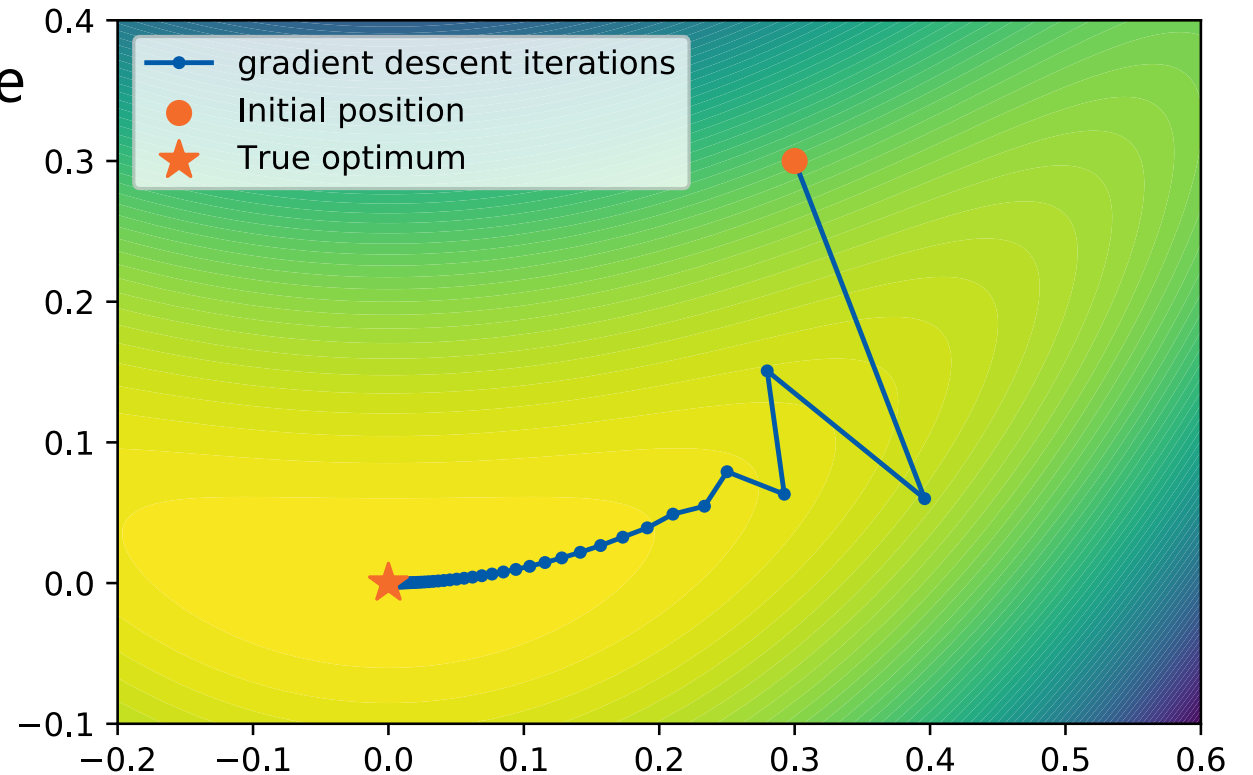
Can optimize functions starting at some initial point $x^{(0)}$ and moving opposite to the gradient:

$$x^{(k)} \leftarrow x^{(k-1)} - \alpha \nabla_x f(x^{(k-1)})$$

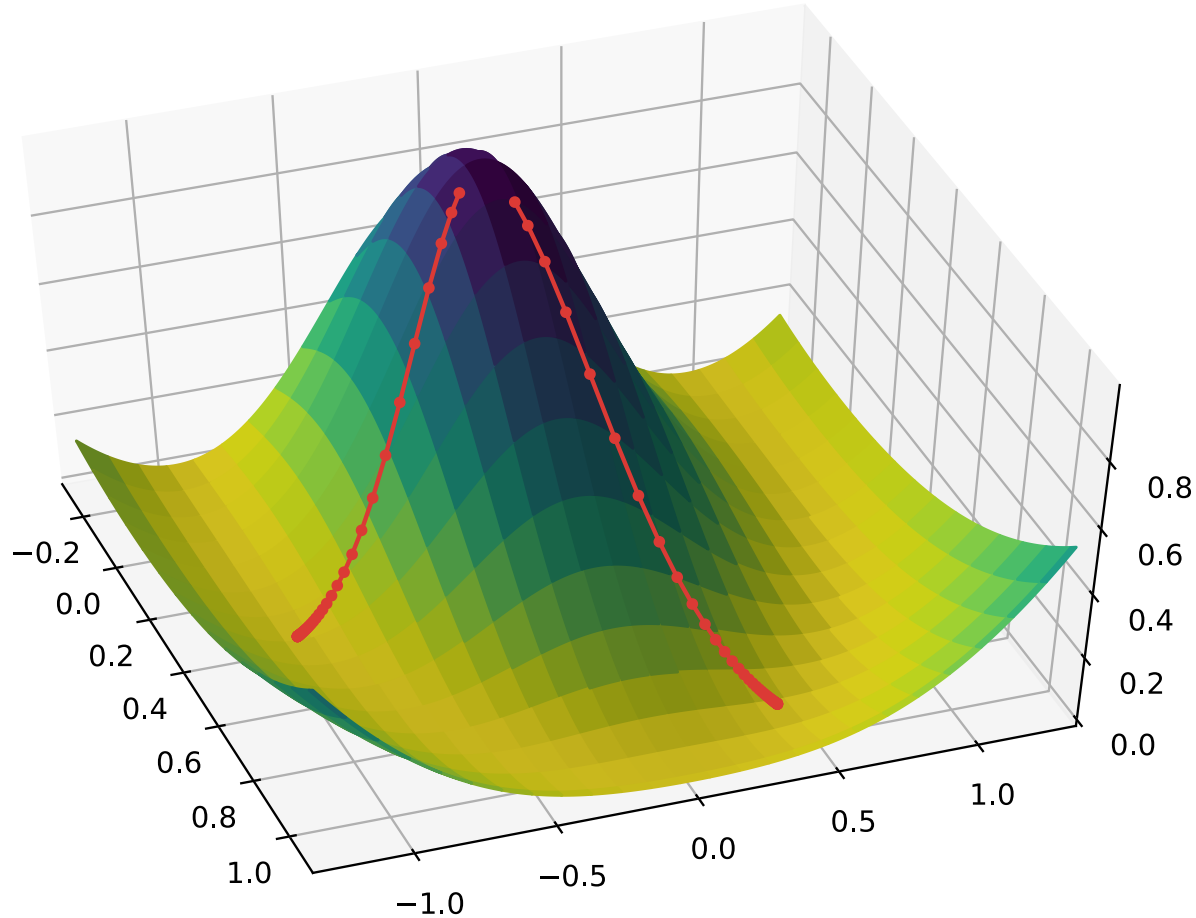
with $\alpha \in \mathbb{R}$, $\alpha > 0$ – learning rate.

For smooth **convex** functions with a single minimum x^* :

$$f(x^{(k)}) - f(x^*) = \mathcal{O}\left(\frac{1}{k}\right)$$



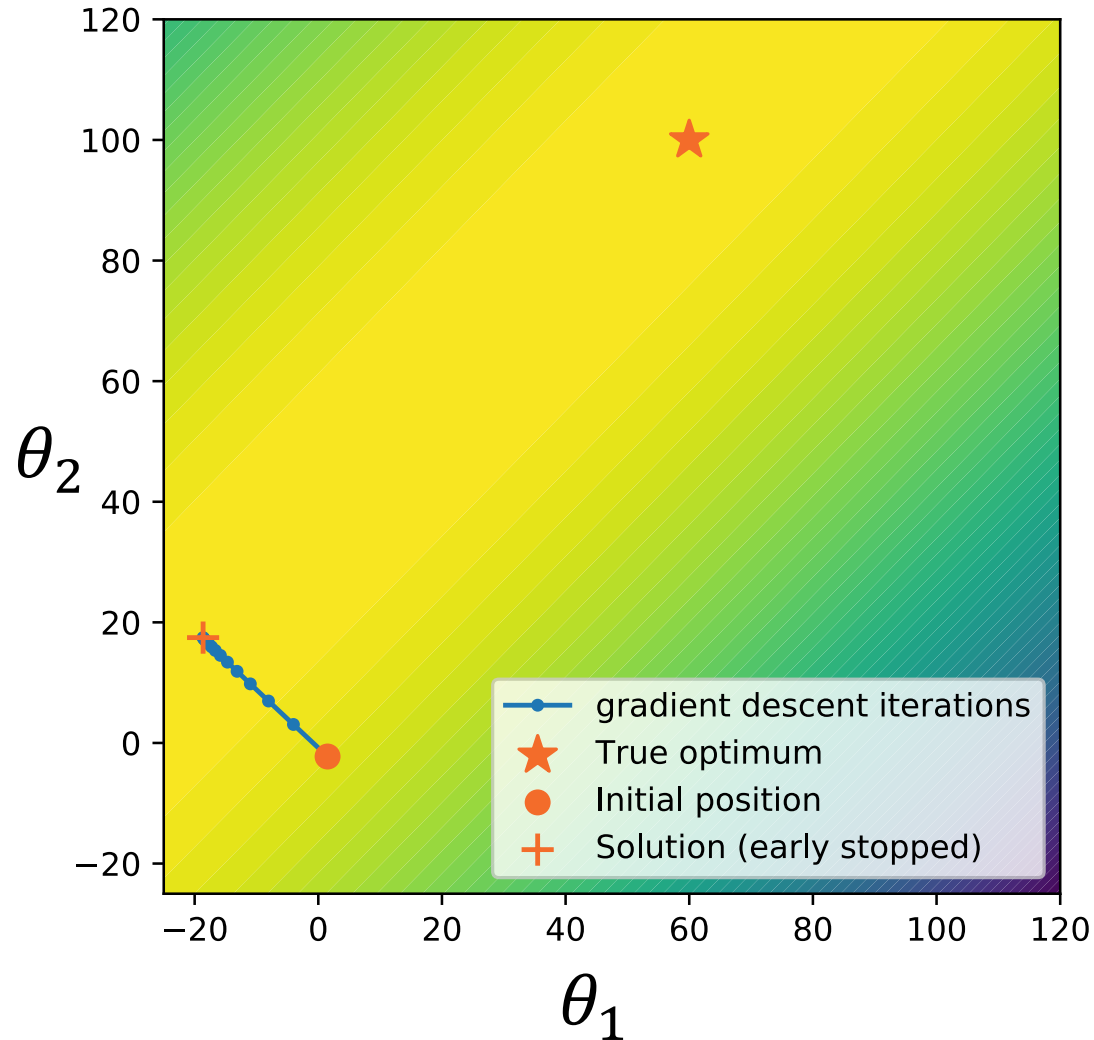
Gradient descent for non-convex functions



May get to a minimum which is not global

Result depends on the starting point

Gradient descent as means for regularisation



Large parameter values typically mean overfitting

You may avoid this problem by initializing parameters with **small values** and **early stopping** the gradient descent

Stochastic Gradient Descent (SGD)

In machine learning we optimize loss functions which are typically averages over objects:

$$L = \frac{1}{N} \sum_{i=1 \dots N} \mathcal{L}(y_i, \hat{f}_{\theta}(x_i))$$

Stochastic Gradient Descent (SGD)

In machine learning we optimize loss functions which are typically averages over objects:

$$L = \frac{1}{N} \sum_{i=1 \dots N} \mathcal{L}(y_i, \hat{f}_{\theta}(x_i))$$

For large N , gradient descent is computationally inefficient and may be unfeasible in terms of memory consumption

Stochastic Gradient Descent (SGD)

In machine learning we optimize loss functions which are typically averages over objects:

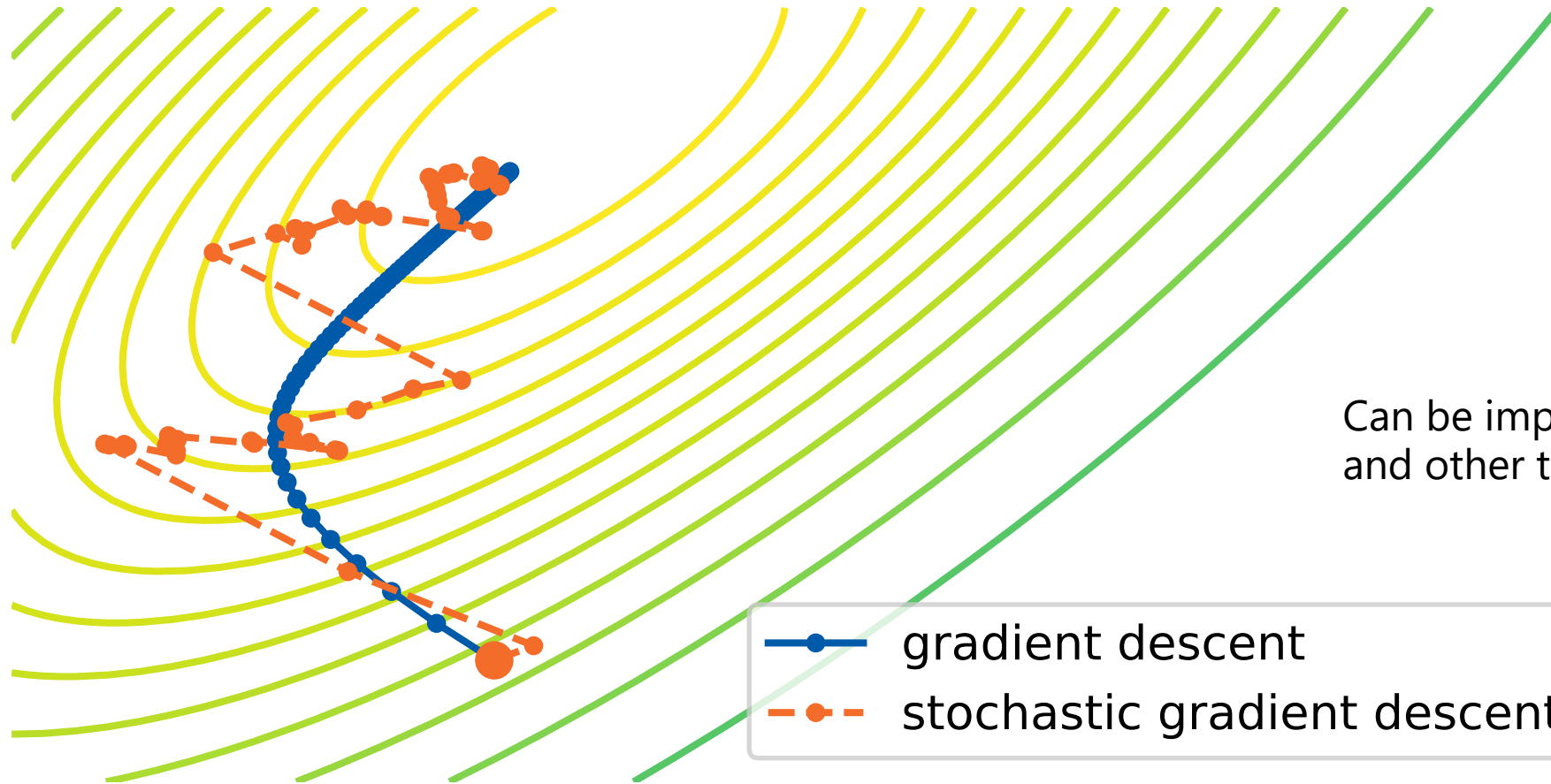
$$L = \frac{1}{N} \sum_{i=1 \dots N} \mathcal{L}(y_i, \hat{f}_\theta(x_i))$$

For large N , gradient descent is computationally inefficient and may be unfeasible in terms of memory consumption

Alternative:

- At each step k pick $l_k \in \{1, \dots, N\}$ at random
- Optimize: $\theta^{(k)} \leftarrow \theta^{(k-1)} - \alpha \nabla_{\theta} \mathcal{L}(y_{l_k}, \hat{f}_{\theta}(x_{l_k})) \Big|_{\theta = \theta^{(k-1)}}$

Stochastic Gradient Descent (SGD)



SGD convergence rate for smooth **convex** functions with a single minimum: $\mathcal{O}\left(\frac{1}{\sqrt{k}}\right)$

Can be improved by batching and other tricks

Feature Expansion



Feature expansion

One can perform **feature transformations** with any function $\Phi: \mathbb{R}^d \rightarrow \mathbb{R}^{d'}$

$$X = \begin{bmatrix} x_1^1 & x_1^2 & \cdots & x_1^d \\ x_2^1 & x_2^2 & \cdots & x_2^d \\ \vdots & \vdots & \ddots & \vdots \\ x_N^1 & x_N^2 & \cdots & x_N^d \end{bmatrix} \longrightarrow \Phi(X) = \begin{bmatrix} \Phi^1(x_1^1, \dots, x_1^d) & \cdots & \Phi^{d'}(x_1^1, \dots, x_1^d) \\ \Phi^1(x_2^1, \dots, x_2^d) & \cdots & \Phi^{d'}(x_2^1, \dots, x_2^d) \\ \vdots & \ddots & \vdots \\ \Phi^1(x_N^1, \dots, x_N^d) & \cdots & \Phi^{d'}(x_N^1, \dots, x_N^d) \end{bmatrix}$$

Feature expansion

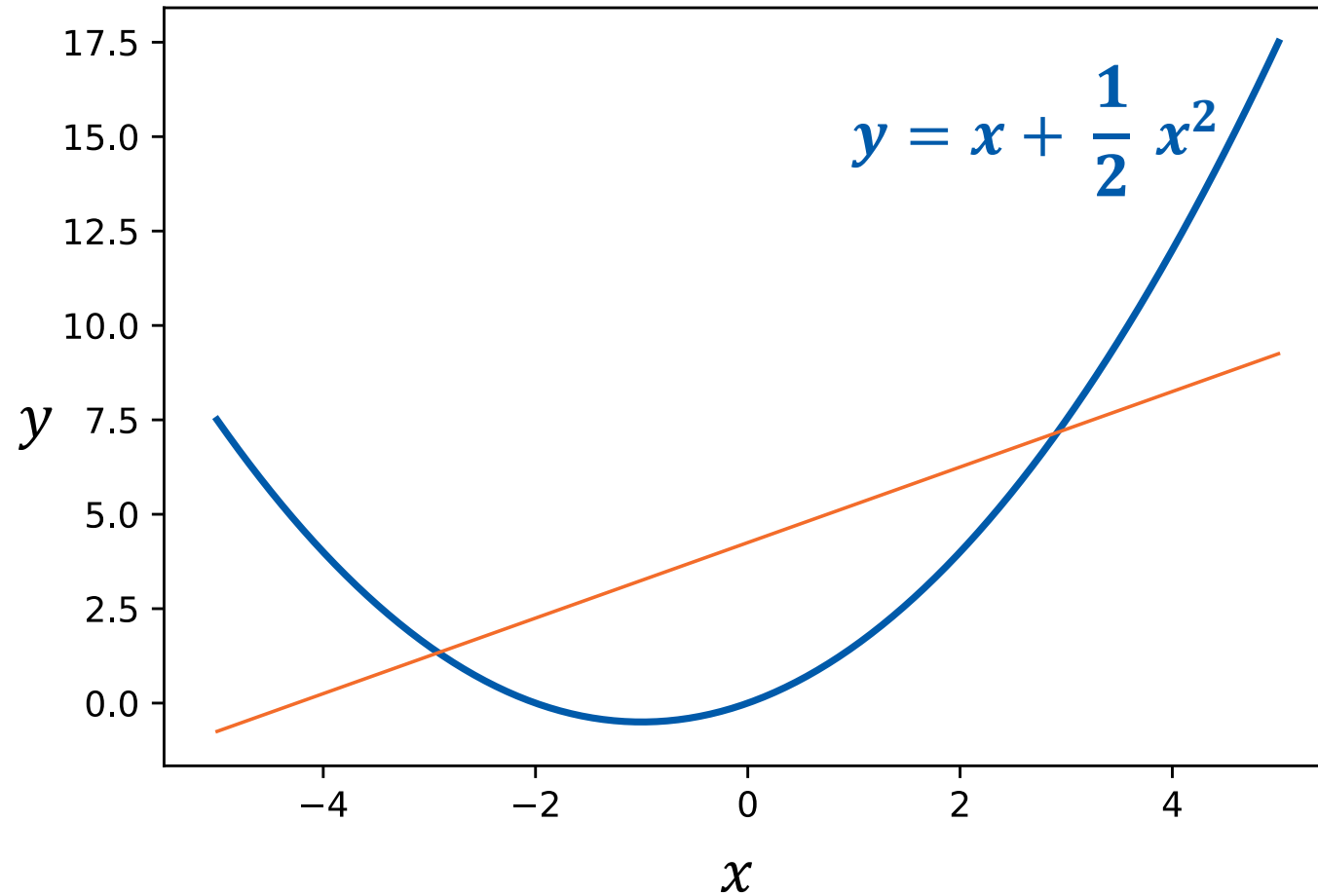
One can perform **feature transformations** with any function $\Phi: \mathbb{R}^d \rightarrow \mathbb{R}^{d'}$

$$X = \begin{bmatrix} x_1^1 & x_1^2 & \cdots & x_1^d \\ x_2^1 & x_2^2 & \cdots & x_2^d \\ \vdots & \vdots & \ddots & \vdots \\ x_N^1 & x_N^2 & \cdots & x_N^d \end{bmatrix} \longrightarrow \Phi(X) = \begin{bmatrix} \Phi^1(x_1^1, \dots, x_1^d) & \cdots & \Phi^{d'}(x_1^1, \dots, x_1^d) \\ \Phi^1(x_2^1, \dots, x_2^d) & \cdots & \Phi^{d'}(x_2^1, \dots, x_2^d) \\ \vdots & \ddots & \vdots \\ \Phi^1(x_N^1, \dots, x_N^d) & \cdots & \Phi^{d'}(x_N^1, \dots, x_N^d) \end{bmatrix}$$

Finding the best function Φ is called **feature engineering**

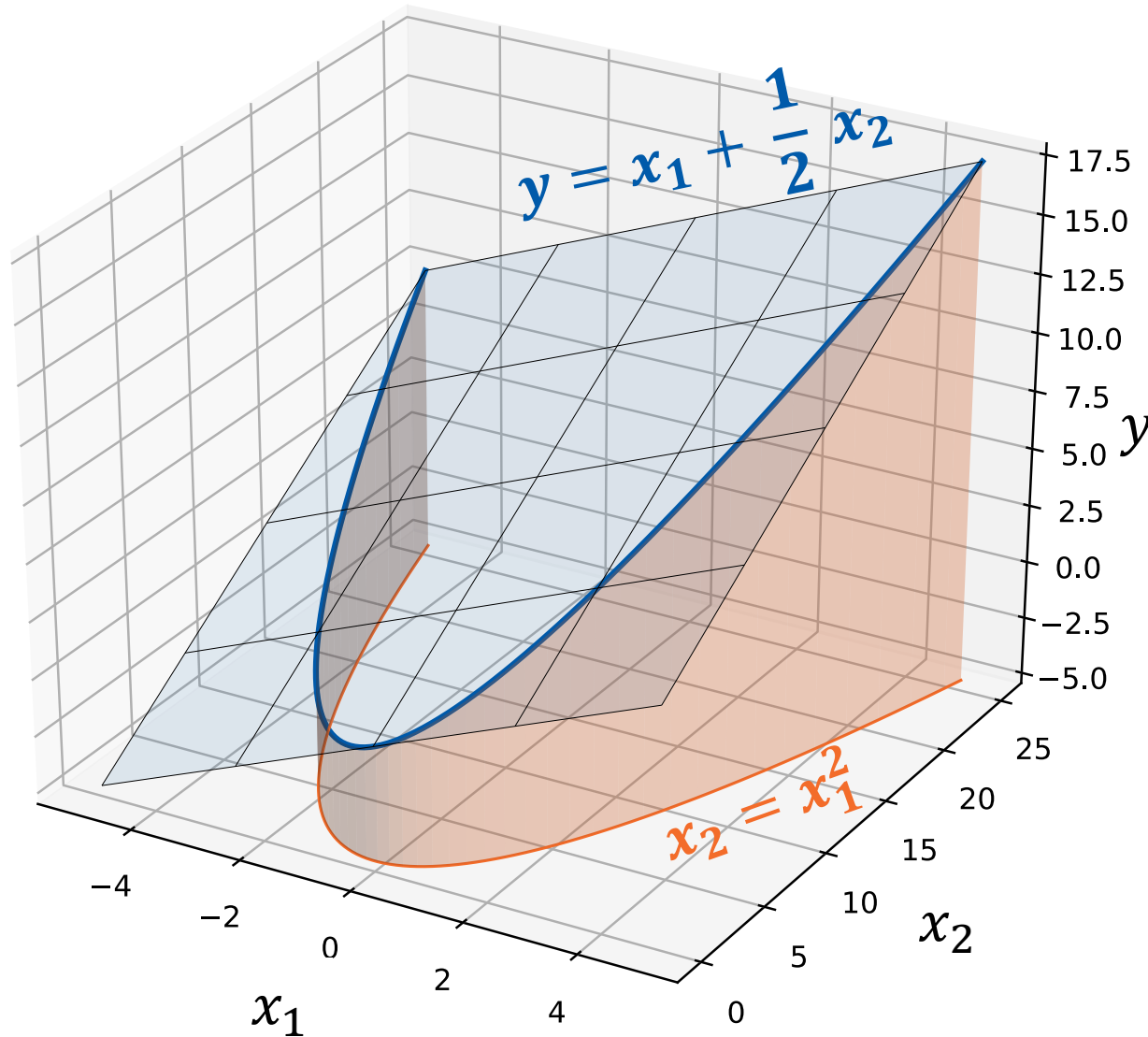
- It is an important part of machine learning and requires deep understanding of the underlying problem and the data

Example: polynomial features



Can't be solved with the only linear feature (x)

Example: polynomial features



Introducing another feature does the job:

$$(x_1, x_2) \equiv (x, x^2)$$

Now our estimate is:

$$\hat{f}(x) = \theta_1 x + \theta_2 x^2$$

Polynomial features of degree p (general case)

For the original features:

$$(x_i^1, x_i^2, \dots, x_i^d)$$

introduce all unique multiplicative combinations of the form:

$$(x_i^{k_1})^{p_1} \cdot (x_i^{k_2})^{p_2} \cdot \dots \cdot (x_i^{k_m})^{p_m}$$

with $p_1 + p_2 + \dots + p_m \leq p$

Example: degree 3 polynomial features

For the original features (a, b, c) :

$(1, a, b, c, a^2, ab, ac, b^2, bc, c^2, a^3, a^2b, a^2c, ab^2, abc, ac^2, b^3, b^2c, bc^2, c^3)$

Summary

Understanding linear models gives **useful insights** into more complicated machine learning algorithms and optimization

Linear Regression with MSE loss allows for **analytical solution**

The stability of the solution depends on the **feature correlations**

Linear models can be optimized with gradient descent and stochastic gradient descent

- In some cases this can **regularize** the solution

Feature transformations allow for very powerful use of the linear models

Thank you!



msohrabi@hse.ru

Majid Sohrabi