Lecture 5: Regression

Linear, multi-linear, non-linear and logistic

Nils Olovsson
nils.olofsson@igp.uu.se

Introduction to data science and Python programming for medical research
Uppsala University

2025

UPPSALA
UNIVERSITET

Content

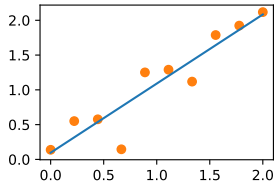
Introduction

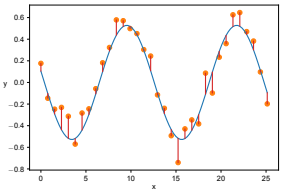
Introduction: Problem

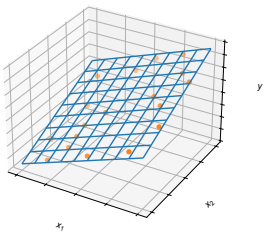
We want to **fit** a **function**, **curve** or a **model** to our data in some **optimal** manner.

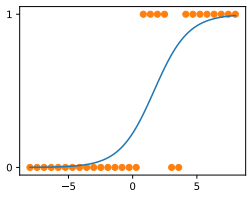
Regression methods:

- Linear regression
- Multi-linear regression
- Non-linear regression
- Logistic regression









Introduction: Regression and classification

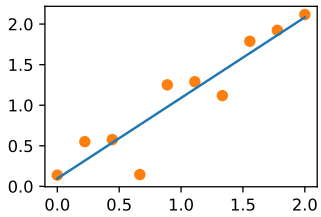
Two families of problems and methods:

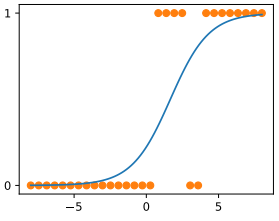
Regression

Continuous input $\rightarrow$ *continuous output*
Example: Medication dosage (mg) and blood pressure (hg).

Classification

Continuous input $\rightarrow$ *discrete/binary output*
Example: Medication dosage (mg) and treatment outcome (successful or not)





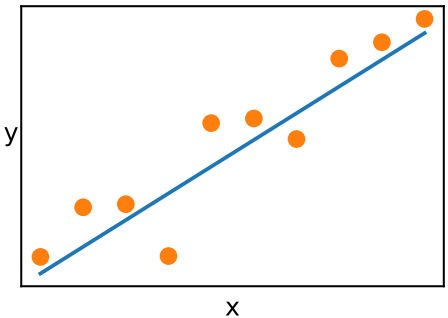
Linear regression

Linear regression: Straight line

Assume a **linear relationship** between x and y.

We want to **fit a straight line** to data such that we can **predict** y from x.

$$y = kx + m$$



Linear regression: Parameters

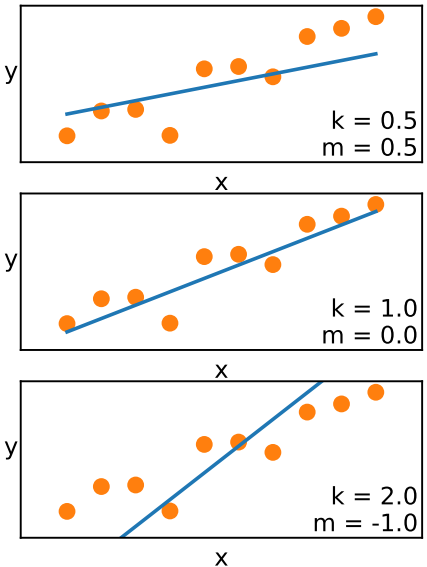
We have N data points with x and y coordinates.

$$\begin{cases} x = [x_1, \ldots, x_N] \\ y = [y_1, \ldots, y_N] \end{cases}$$

Equation for straight line have **two parameters** we can adjust to fit the line to our data.

$$y = kx + m$$

$$\begin{cases} k: \text{ slope} \\ m: \text{ intercept} \end{cases}$$



Linear regression: Least squares fit

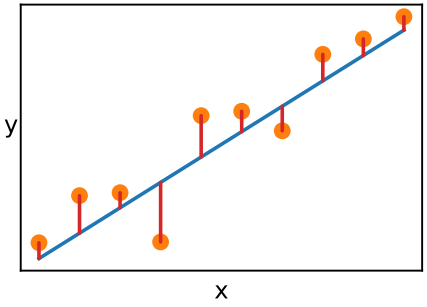
What is a **good fit** of a line to our data?

What is a **bad fit**?

We define an **error** function, e, that compares the model to the data.

Sum of squared vertical distances, d, between the line and the N data points, e.

$$e = \sum_{i}^{N} {d_i}^2 = \sum_{i}^{N} (kx_i + m - y_i)^2$$



Linear regression: Optimality

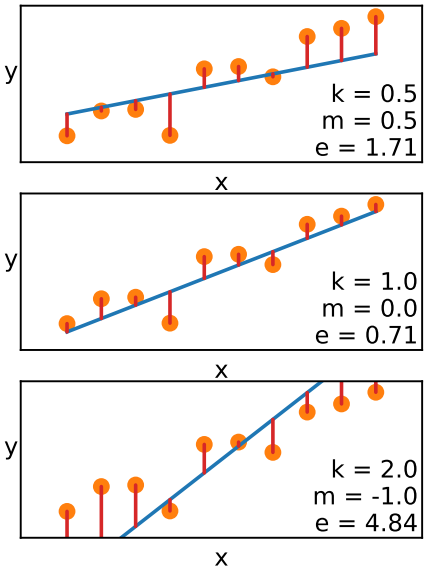
Line that best fits data should be the one with the **smallest possible error**

$$e(k, m; x, y) = \sum_{i}^{N} {d_i}^2 = \sum_{i}^{N} (kx_i + m - y_i)^2 .$$

How do we optimal values for the two parameters k and m?

An **optimization problem** that states that we want to **minimize the error** e over the **two parameters** k **and** m

$$\arg\min_{k,m} e(k, m; x, y)$$

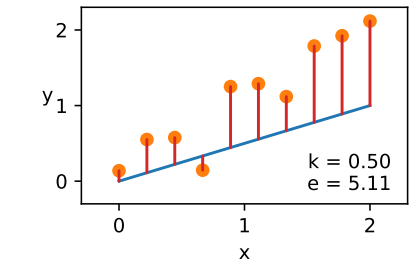


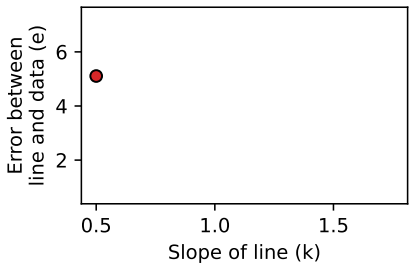
Linear regression: Optimal slope

First look at a simpler problem. Have a line with fixed intercept of $m = 0$. Try to find the optimal slope k to fit it to our data.

$$\arg\min_k e(k; x, y)$$

By looking at several possible values for the slope we can see that there is an optimal value at the bottom of the error function!



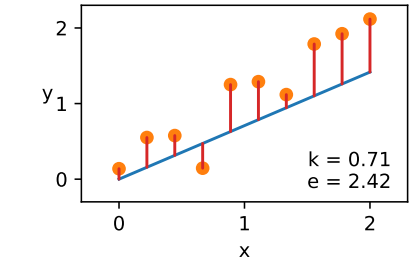


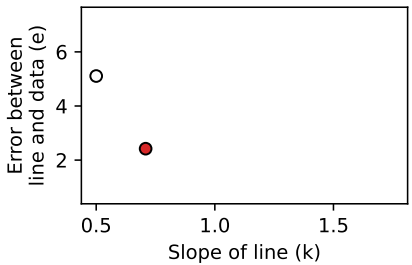
Linear regression: Optimal slope

First look at a simpler problem. Have a line with fixed intercept of $m = 0$. Try to find the optimal slope k to fit it to our data.

$$\arg\min_k e(k; x, y)$$

By looking at several possible values for the slope we can see that there is an optimal value at the bottom of the error function!



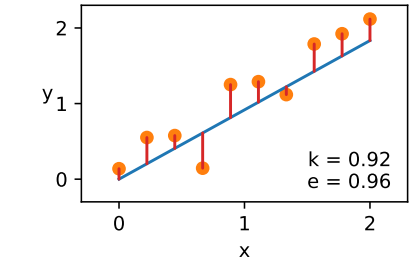


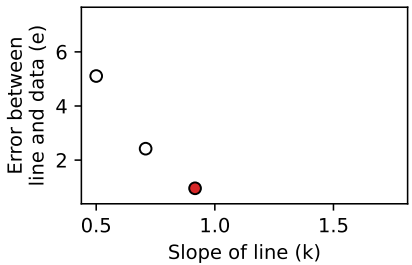
Linear regression: Optimal slope

First look at a simpler problem. Have a line with fixed intercept of $m = 0$. Try to find the optimal slope k to fit it to our data.

$$\arg\min_k e(k; x, y)$$

By looking at several possible values for the slope we can see that there is an optimal value at the bottom of the error function!



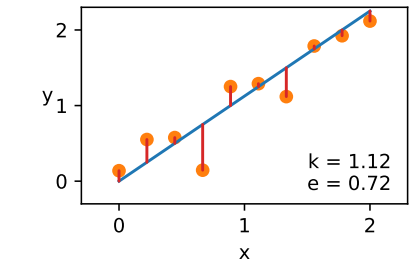


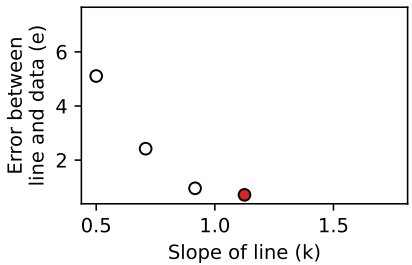
Linear regression: Optimal slope

First look at a simpler problem. Have a line with fixed intercept of $m = 0$. Try to find the optimal slope k to fit it to our data.

$$\arg\min_k e(k; x, y)$$

By looking at several possible values for the slope we can see that there is an optimal value at the bottom of the error function!



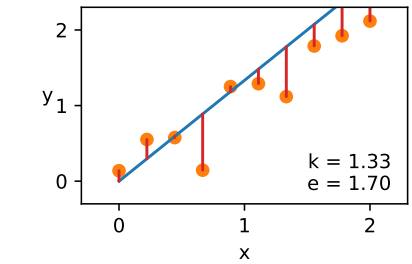


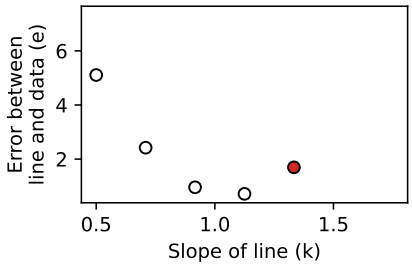
Linear regression: Optimal slope

First look at a simpler problem. Have a line with fixed intercept of $m = 0$. Try to find the optimal slope k to fit it to our data.

$$\arg\min_k e(k; x, y)$$

By looking at several possible values for the slope we can see that there is an optimal value at the bottom of the error function!



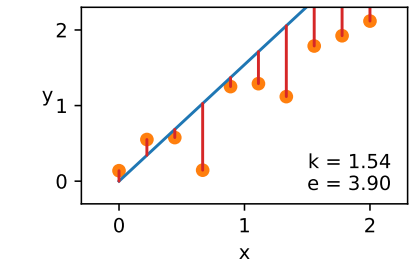


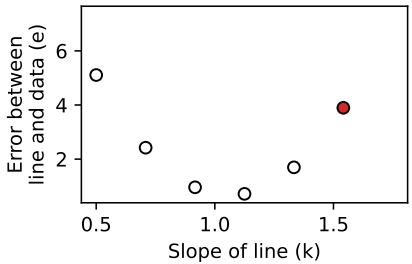
Linear regression: Optimal slope

First look at a simpler problem. Have a line with fixed intercept of $m = 0$. Try to find the optimal slope k to fit it to our data.

$$\arg\min_k e(k; x, y)$$

By looking at several possible values for the slope we can see that there is an optimal value at the bottom of the error function!



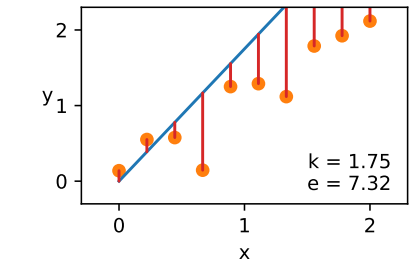


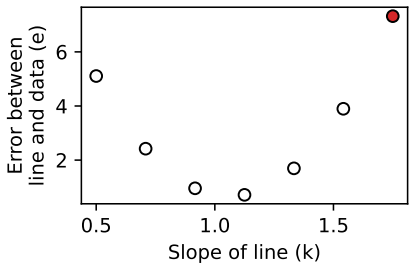
Linear regression: Optimal slope

First look at a simpler problem. Have a line with fixed intercept of $m = 0$. Try to find the optimal slope k to fit it to our data.

$$\arg\min_k e(k; x, y)$$

By looking at several possible values for the slope we can see that there is an optimal value at the bottom of the error function!



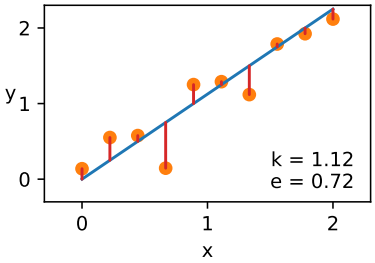


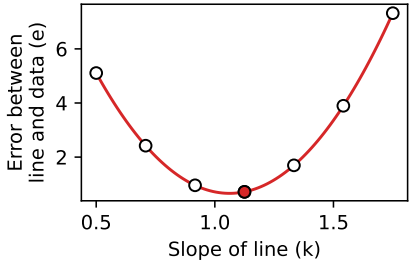
Linear regression: Optimal slope

First look at a simpler problem. Have a line with fixed intercept of $m = 0$. Try to find the optimal slope k to fit it to our data.

$$\arg\min_k e(k; x, y)$$

By looking at several possible values for the slope we can see that there is an optimal value at the bottom of the error function!





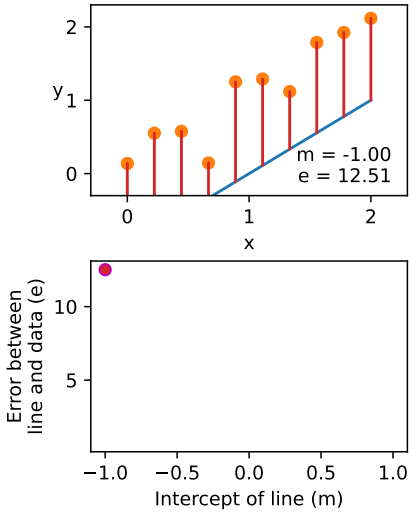
Linear regression: Optimal intercept

Now, the alternative problem. A line with fixed slope ($k = 1$) for which we try to find the **optimal intercept** m.

$$\arg\min_{m} e(m; x, y)$$

Looking at values for the intercept, see that there is an **optimal value** at the **bottom of the error function** for this problem as well!

The **convex** shape of these error functions is not by accident.
We use the sum of the **square** of the distance d_i^2!



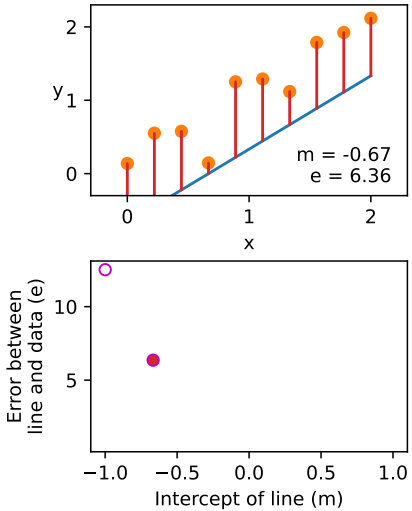
Linear regression: Optimal intercept

Now, the alternative problem. A line with fixed slope ($k = 1$) for which we try to find the **optimal intercept** m.

$$\arg\min_{m} e(m; x, y)$$

Looking at values for the intercept, see that there is an **optimal value** at the **bottom of the error function** for this problem as well!

The **convex** shape of these error functions is not by accident.
We use the sum of the **square** of the distance d_i^2!



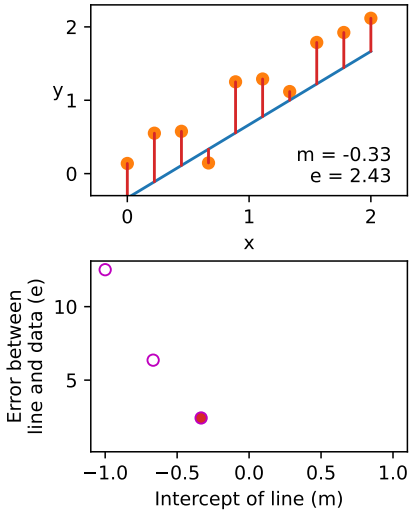
Linear regression: Optimal intercept

Now, the alternative problem. A line with fixed slope ($k = 1$) for which we try to find the **optimal intercept** m.

$$\arg\min_{m} e(m; x, y)$$

Looking at values for the intercept, see that there is an **optimal value** at the **bottom of the error function** for this problem as well!

The **convex** shape of these error functions is not by accident.
We use the sum of the **square** of the distance d_i^2!



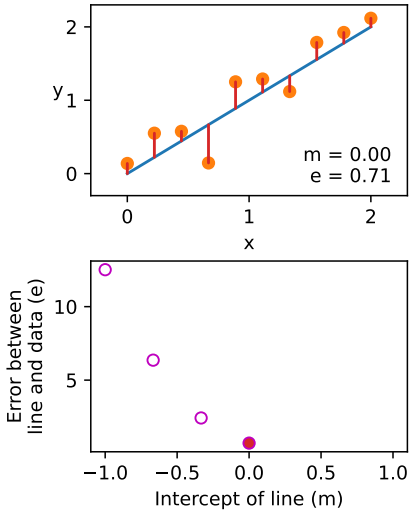
Linear regression: Optimal intercept

Now, the alternative problem. A line with fixed slope ($k = 1$) for which we try to find the **optimal intercept** m.

$$\arg\min_m e(m; x, y)$$

Looking at values for the intercept, see that there is an **optimal value** at the **bottom of the error function** for this problem as well!

The **convex** shape of these error functions is not by accident.
We use the sum of the **square** of the distance $d_i{}^2$!



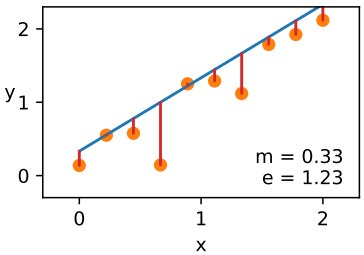
Linear regression: Optimal intercept

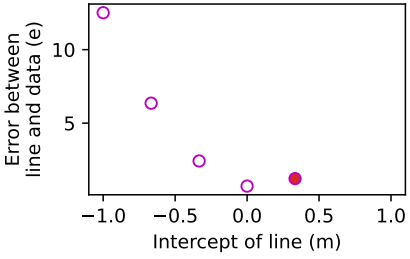
Now, the alternative problem. A line with fixed slope ($k = 1$) for which we try to find the **optimal intercept** m.

$$\arg\min_m e(m; x, y)$$

Looking at values for the intercept, see that there is an **optimal value** at the **bottom of the error function** for this problem as well!

The **convex** shape of these error functions is not by accident.
We use the sum of the **square** of the distance $d_i{}^2$!





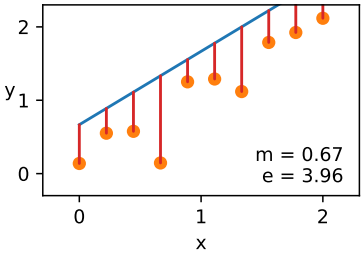
Linear regression: Optimal intercept

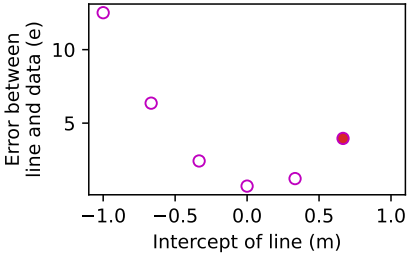
Now, the alternative problem. A line with fixed slope ($k = 1$) for which we try to find the **optimal intercept** m.

$$\arg\min_{m} e(m; x, y)$$

Looking at values for the intercept, see that there is an **optimal value** at the **bottom of the error function** for this problem as well!

The **convex** shape of these error functions is not by accident.
We use the sum of the **square** of the distance $d_i{}^2$!





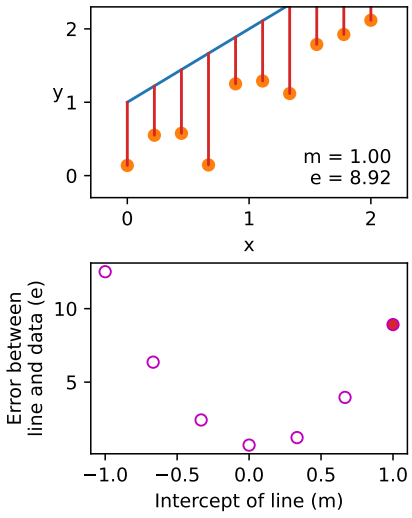
Linear regression: Optimal intercept

Now, the alternative problem. A line with fixed slope ($k = 1$) for which we try to find the **optimal intercept** m.

$$\arg\min_m e(m; x, y)$$

Looking at values for the intercept, see that there is an **optimal value** at the **bottom of the error function** for this problem as well!

The **convex** shape of these error functions is not by accident.
We use the sum of the **square** of the distance $d_i{}^2$!



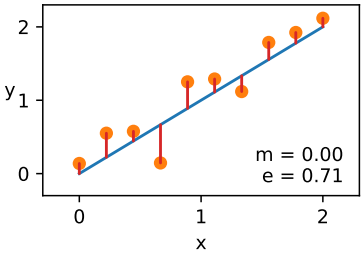
Linear regression: Optimal intercept

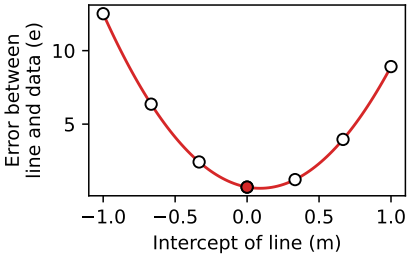
Now, the alternative problem. A line with fixed slope ($k = 1$) for which we try to find the **optimal intercept** m.

$$\arg\min_m e(m; x, y)$$

Looking at values for the intercept, see that there is an **optimal value** at the **bottom of the error function** for this problem as well!

The **convex** shape of these error functions is not by accident.
We use the sum of the **square** of the distance $d_i{}^2$!





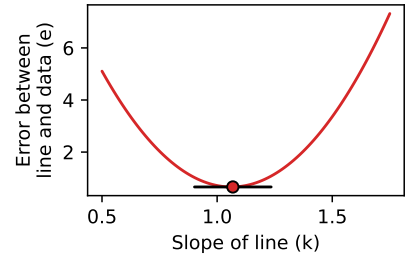
Linear regression: Extreme values

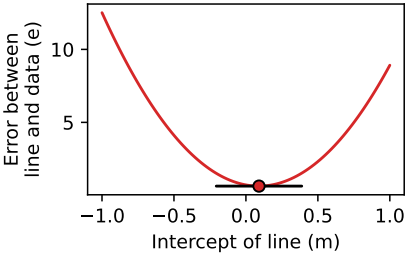
We have seen that the optimal parameter values for the slope and the intercept both have been at the bottom of a, convex, well shaped error function.

The optimal values for our parameters are at the **extreme value** of these functions.

I.e. where the **derivative**, the *slope*, of the error function is **zero**!

$$\begin{cases} \dfrac{\partial e(k)}{\partial k} = 0 \\ \dfrac{\partial e(m)}{\partial m} = 0 \end{cases}$$





Linear regression: Gradient

In the real optimization problem

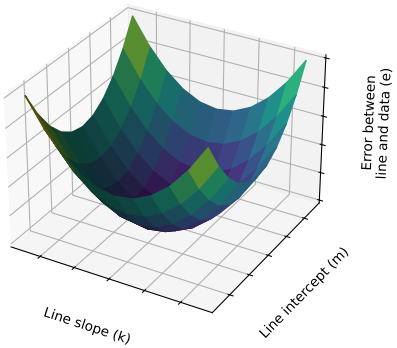
$$\arg\min_{k,m} e(k, m; x, y)$$

we need to **solve for both** the slope, k, and the intercept, m, **simultaneously**.

Look for an extreme value of the 2D derivative, *i.e.* the **gradient** of e, ∇e that we want to be **zero**.

$$\nabla e(k, m) = \frac{\partial e(k, m)}{\partial k}\hat{k} + \frac{\partial e(k, m)}{\partial m}\hat{m} = 0$$

We note that this is just two regular derivatives!



Linear regression: Optimal parameters

We started by defining our **squared distance error** function for the **line equation**

$$e(k, m; x, y) = \sum_{i}^{N} {d_i}^2 = \sum_{i}^{N} (kx_i + m - y_i)^2 \ .$$

and formulated an **optimization problem**

$$\arg\min_{k,m} e(k, m; x, y)$$

that told us we want to find where the **gradient of the error**, ∇e, is **zero**

$$\nabla e(k, m) = \frac{\partial e(k, m)}{\partial k} \hat{k} + \frac{\partial e(k, m)}{\partial m} \hat{m} = 0$$

We see that for ∇e to be zero **both its components need to be zero**

$$\begin{cases} \dfrac{\partial e(k, m)}{\partial k} = 0, & \dfrac{\partial e(k, m)}{\partial m} = 0 \end{cases}$$

Linear regression: Expanding $\partial e/\partial k = 0$

$$\frac{\partial e(k,m)}{\partial k} = 0$$

$$\frac{\partial}{\partial k}\sum_i (kx_i + m - y_i)^2 = 0$$

$$\sum_i \frac{\partial}{\partial k}(kx_i + m - y_i)^2 = 0$$

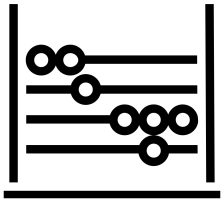
$$\sum_i 2(kx_i + m - y_i)x_i = 0$$

$$\sum_i \cancel{2}(kx_i^2 + mx_i - x_i y_i) = 0$$

$$k\sum_i x_i^2 + m\sum_i x_i - \sum_i x_i y_i = 0$$

Derivation rules

- $\frac{\partial}{\partial x} f(g(x)) = f'(g(x))g'(x)$
- $\frac{\partial}{\partial x}\sum_i f_i(x) = f_1'(x) + \cdots + f_2'(x)$



Linear regression: Expanding $\partial e/\partial m = 0$

$$\frac{\partial e(k,m)}{\partial m} = 0$$

$$\frac{\partial}{\partial m}\sum_i (kx_i + m - y_i)^2 = 0$$

$$\sum_i \frac{\partial}{\partial m}(kx_i + m - y_i)^2 = 0$$

$$\sum_i 2(kx_i + m - y_i) = 0$$

$$\sum_i \cancel{2}(kx_i + m - y_i) = 0$$

$$k\sum_i x_i + m\sum_i 1 - \sum_i y_i = 0$$

Linear regression: System of two equations

By substitution we obtain a **system of two equations** for our two unknowns, k and m.

$$\begin{cases} ck + am - d = 0 \\ ak + Nm - b = 0 \end{cases} \qquad a = \sum_i x_i,\ b = \sum_i y_i,\ c = \sum_i x_i^2,\ d = \sum_i x_i y_i$$

With some effort (or the Python script on the right) we can solve this to finally find our optimal parameter values for the line as

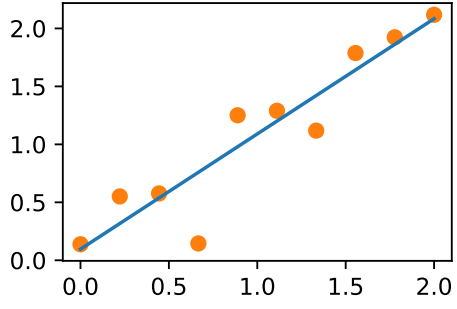
$$\begin{cases} k = \dfrac{Nd - ab}{Nc - a^2} \\ m = \dfrac{ad - bc}{a^2 - Nc} \end{cases}$$

```
1 import sympy
2 k, m, a, b, c, d, N = sympy.symbols('k m a b c
      d N')
3 sol = sympy.solve([c * k + a * m - d,
4                    a * k + N * m - b],
5                   [k, m],
6                   )
7 print("Solution to system of equations:")
8 print(sol)
```

Linear regression: Python implementation

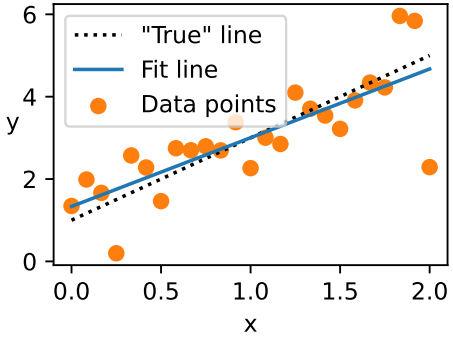
```python
import numpy as np
import matplotlib.pyplot as plt

def linear_regression(
    x: np.ndarray,
    y: np.ndarray,
) -> Tuple[float, float]:
    N = len(x)
    a = np.sum(x);    b = np.sum(y)
    c = np.sum(x**2); d = np.sum(x * y)
    k = (N * d - a * b) / (N * c - a * a)
    m = (a * d - b * c) / (a * a - N * c)
    return k, m

# Define a straight line y = x
x = np.linspace(0, 2, 10)
y = x
# Sample some random y errors
rng = np.random.default_rng(seed=1)
y_e = y + rng.normal(0, 0.4, size=y.shape)
# Fit line (our linear regression)
k, m = linear_regression(x, y_e)
y = k * x + m
# Plot
ax = plt.figure(figsize=(3,2)).gca()
ax.plot(x, y, color='tab:blue')
ax.scatter(x, y_e, marker='o', color='tab:
    orange')
#plt.show()
```

$$\begin{cases} k = \dfrac{Nd - ab}{Nc - a^2} \\ m = \dfrac{ad - bc}{a^2 - Nc} \end{cases}$$



Linear regression: Python

```python
import numpy as np
import statistics
import scipy.stats
import sklearn

def print_result(
    name: str,
    k: float,
    m: float,
) -> None:
    print(f"Linear regression ({name}):")
    print(f"k = {lr.slope:.5f}")
    print(f"m = {lr.intercept:.5f}")
    return
# Define points x, true y and y with errors
x = np.linspace(0, 2, 25)
yt = 2 * x + 1
rng = np.random.default_rng(seed=1)
y_e = yt + rng.normal(0, 1.0, size=yt.shape)
# Fit line (Std. library, scipy, sklearn)
k, m = statistics.linear_regression(x, y_e)
lr = scipy.stats.linregress(x, y_e)
lm = sklearn.linear_model.LinearRegression()
lm.fit(x.reshape(-1, 1), y_e)
# Print results and compare
print_result('scipy', lr.slope, lr.intercept)
print_result('sklearn',
             lm.coef_[0], lm.intercept_)
print_result('Python', k, m)
```



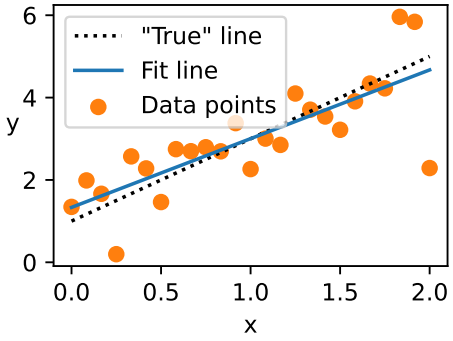

```
>> Linear regression (scipy):
>> k = 1.66938
>> m = 1.33349
>> Linear regression (sklearn):
>> k = 1.66938
>> m = 1.33349
>> Linear regression (Python):
>> k = 1.66938
>> m = 1.33349
```

Linear regression: Statistical significance

```python
import numpy as np
import scipy.stats
import matplotlib.pyplot as plt

# Define points x, true y and y with errors
x = np.linspace(0, 2, 25)
yt = 2 * x + 1
rng = np.random.default_rng(seed=1)
y_e = yt + rng.normal(0, 1.0, size=yt.shape)

# Fit line (Std. library, scipy, sklearn)
lr = scipy.stats.linregress(x, y_e)
print(f"p value: {lr.pvalue:.8f}")

# Calculate values on the line and plot
y = lr.slope * x + lr.intercept
ax = plt.figure(figsize=(3,2)).gca()
ax.plot(x, yt, color='k', linestyle=':',
        label='"True" line')
ax.plot(x, y, color='tab:blue',
        label='Fit line')
ax.scatter(x, y_e, marker='o',
           color='tab:orange',
           label='Data points')
ax.set_ylabel('y', rotation=0)
ax.set_xlabel('x')
ax.legend()
#plt.show()
```

P values not reported by all packages. Can be calculated with `scipy.stats` or `statsmodels`.

```
>> p value: 0.00000361
```



Multilinear regression

Multilinear regression: Fit a plane

If y depends linearly on **more than one feature** we have to **instead of fitting a line**

$$y = kx + m$$

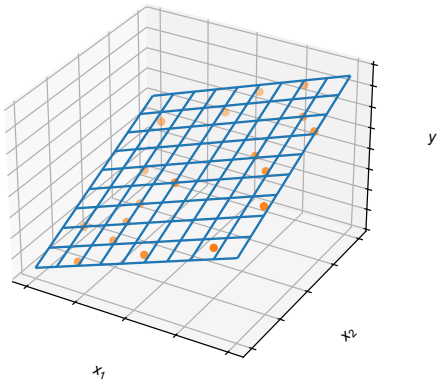
fit a plane in 2D

$$y = k_1 x_1 + k_2 x_2 + m$$

or more generally a **hyperplane**

$$y = k_1 x_1 + \cdots + k_M x_M + m$$

using **multilinear regression**.



Multilinear regression: Python

```python
import numpy as np
from sklearn.linear_model import LinearRegression
rng = np.random.default_rng(seed=1)

# Define data sampled on a plane
M = 20
x = np.linspace(0, 2, M)
y = np.linspace(0, 2, M)
x, y = np.meshgrid(x, y)
z = 2 * x + 1.5 * y + 0.5
# Select random points on the grid
ind_r = rng.choice(M, size=2 * M)
ind_c = rng.choice(M, size=2 * M)
x_sel = x[ind_r, ind_c]
y_sel = y[ind_r, ind_c]
z_sel = z[ind_r, ind_c]
# Reshape to [(2*M), 2] shaped data matrix
X = np.concatenate([x_sel.reshape(1,-1),
                    y_sel.reshape(1,-1)]).T
# Sample some random errors
z_sel_e = z_sel + rng.normal(0, 0.4, size=x_sel.
    shape)
# Fit linear model
lr = LinearRegression()
lr.fit(X, z_sel_e)
# Print results
print("Multilinear regression result.")
print(f"Intercept:    {lr.intercept_}")
print(f"Coefficients: {lr.coef_}")
```

```
>> Multilinear regression result.
>> Intercept:    0.32293560817917344
>> Coefficients: [2.0323346 1.5930564]
```

We can see that the regression succeeds in *reconstructing* the plane from which the noisy samples where generated.

That plane was defined to have slopes 2 and 1.5 and interception of 0.5 as seen on line 10 in the code.

Outliers

Outliers

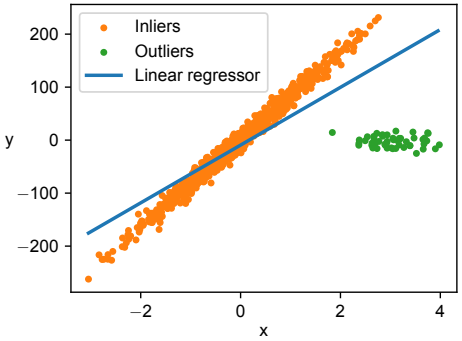
Outliers: Outliers in data

Outliers in the data can **negatively affect our models**.

Manually removing or filtering outliers is both laborious and subjective so it is better to **use standardized methods**.

Will look at two such methods.

- RANSAC
- Theil-Sen



https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.RANSACRegressor.html

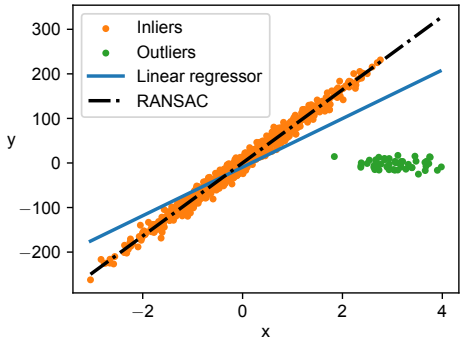
Outliers: RANSAC

RANdom SAmple Consensus (RANSAC) works by **random sub sampling of the data** points.

A regression **model is fit** to this **subset** and evaluated for all data points.

Procedure is then **repeated** for a specified number of iterations.

```
# Fit linear model with RANSAC algorithm
ransac = linear_model.RANSACRegressor()
ransac.fit(X, y)
inlier_mask = ransac.inlier_mask_
outlier_mask = np.logical_not(inlier_mask)
```



https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.RANSACRegressor.html

Outliers: Theil-Sen

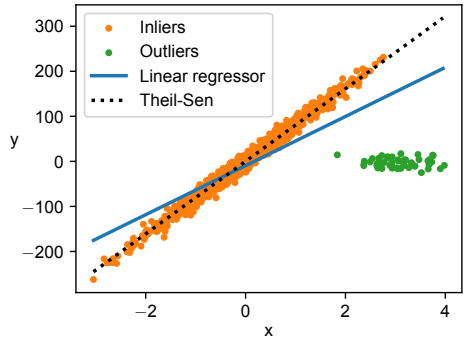
Instead of using a regression model the slope, k, of the line is calculated as the **median slope between all pairs of points** in the data.

$$k_{ij} = \frac{y_i - y_j}{x_i - x_j}$$

Division by zero is avoided by not comparing points with the same x. The **intercept** is calculated as the median of $kx_i - y_i$.

This is **less sensitive to outliers** than linear regression.

```
# Fit linear model with Theil-Sen algorithm
tsr = linear_model.TheilSenRegressor()
tsr.fit(X, y)
```



https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.TheilSenRegressor.html

Outliers: RANSAC or Theil-Sen

For estimating line fit to data with outliers both these methods perform well.

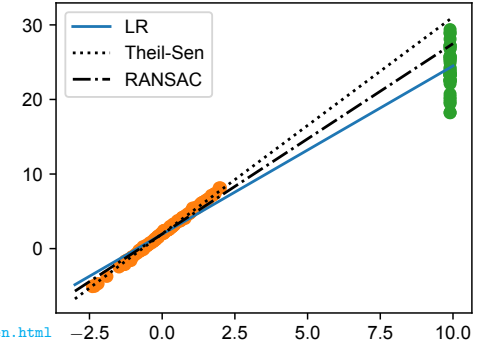
Differences between them to consider:

- Theil-Sen is a special method only for fitting a line/plane.
  RANSAC is a more general method that can be applied to other models.
- RANSAC is a *random* method so we might get different answers when we use it, even if data is the same.
  Theil-Sen is *deterministic*, it will always give the same answer.
- An exact **Theil-Sen** estimator is **computationally intensive** for very large data sets since all pairs of data points must be compared.

Outliers: RANSAC or Theil-Sen (Example)

```python
import matplotlib.pyplot as plt
import numpy as np
from sklearn.linear_model import
      LinearRegression, RANSACRegressor,
      TheilSenRegressor
estimators = [
('LR', LinearRegression()),
('Theil-Sen', TheilSenRegressor(random_state
      =0)),
('RANSAC', RANSACRegressor(random_state=0))]
colors = {'LR': ('tab:blue', '-'),
          'Theil-Sen': ('k', ':'),
          'RANSAC': ('k', '-.')}
n = 200
m = int(0.2 * n)
rng = np.random.default_rng(seed=0)
x = rng.standard_normal(n)
noise = 0.1 * rng.standard_normal(n)
y = 3 * x + 2 + noise
x[-m:] = 9.9; y[-m:] += 22
X = x[:, np.newaxis]
# Plot data and line estimates
ax = plt.figure(figsize=(4,3)).gca()
ax.scatter(x[:-m], y[:-m], color='tab:orange')
ax.scatter(x[-m:], y[-m:], color='tab:green')
line_x = np.array([-3, 10]).reshape(2, 1)
```

```python
for name, estimator in estimators:
    estimator.fit(X, y)
    y_pred = estimator.predict(line_x)
    ax.plot(line_x, y_pred,
            color=colors[name][0],
            linestyle=colors[name][1],
            label=name)
ax.legend(loc="upper left")
#plt.show()
```



https://scikit-learn.org/stable/auto_examples/linear_model/plot_theilsen.html

Non-linear regression

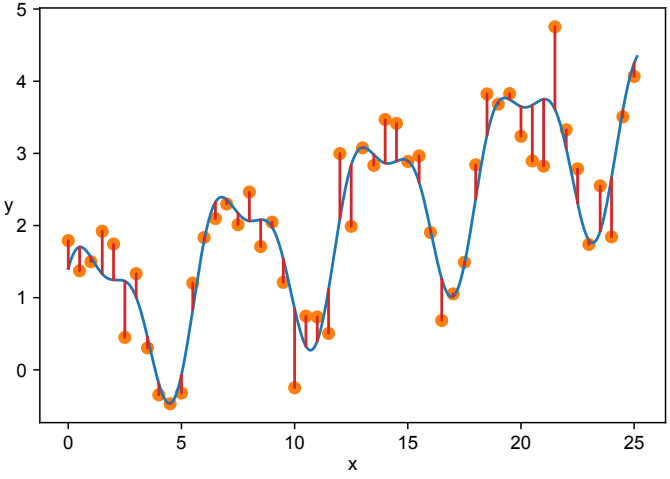
Non-linear regression

Non-linear regression: Problem

Want to fit **another function** besides a line to data.

Use the **same error function** definition with **sum of squared distances**.

This is called **non-linear least squares regression**.



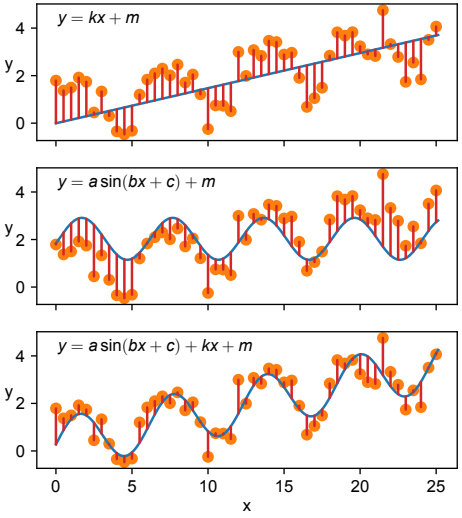
https://en.wikipedia.org/wiki/Non-linear_least_squares

Non-linear regression: Model and parametrization

For **linear regression** we had **two parameters**, k and m. We say that it has **two degrees of freedom**.

In the more general case of **non-linear regression** we can have however **many parameters** are needed to describe the function, model, we want to fit to the data.

However, should **not have more parameters than needed**.



Non-linear regression: Optimization

Same **sum of squared distances error** function as before

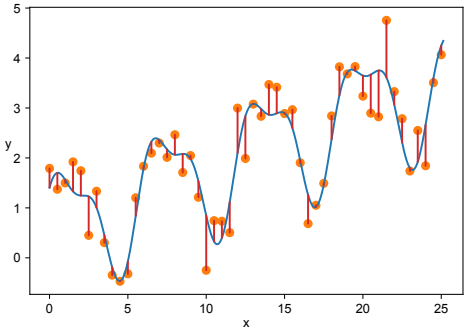
$$e(p; x, y) = \sum_i (f(p; x_i) - y_i)^2$$

Analogous **optimization problem**

$$\arg\min_p e(p; x, y)$$

Smallest error where the **gradient of the error is zero**!

$$\nabla e(p) = 0$$



Non-linear regression: Gradient descent

For **linear regression** we could **calculate formulas for the optimal values** of our two parameters.

In general this can not be done. Instead, find a **solution numerically**.

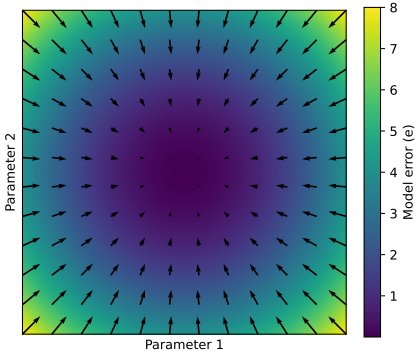
To find the bottom of the error function where $\nabla e \approx 0$, **follow the negative gradient** downwards! This is called **gradient descent**.

The general algorithm for this follows two steps that are repeated:

- For the current iteration j, compute $\nabla e(p_j; x, y)$.
- Update the parameters $p_{j+1} = p_j - \alpha \nabla e(p_j; x, y)$

where α is a step size.

(α is sometimes called *learning rate*.)

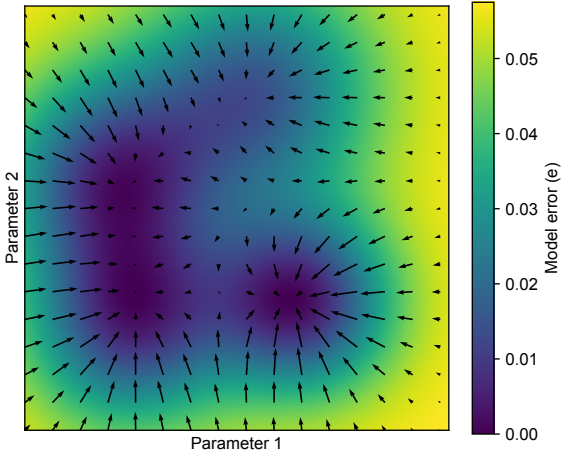


Non-linear regression: Initial values and global optimum

We need to pick a **starting point** for our gradient descent.

If the shape of the **error function landscape** has many smaller pits, **local optima**, these can trap our gradient descent algorithm and we **fail to find the global optimum**.

There is **no general solution** to solve this problem.
However, there are **many strategies**, often based on some stochastic component, to overcome it.



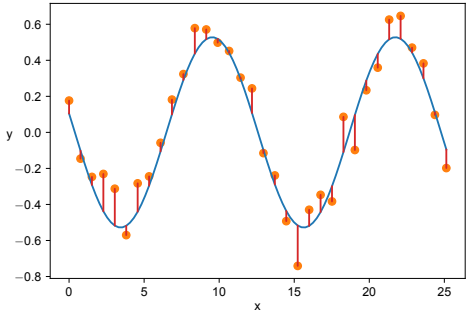
Non-linear regression: Example using scipy

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import least_squares

np.random.seed(0)
y0 = None; x0 = None

def sin_fun(
    x: np.ndarray, p: np.ndarray,
) -> np.ndarray:
    a = p[0]; b = p[1]; c = p[2]
    return  a * np.sin(b * x + c)

def error(par) -> np.ndarray:
    return y0 - sin_fun(x0, par)

p_true = np.array([0.5, 0.5, np.pi])

x = np.linspace(0, 8*np.pi, 100, endpoint=True)
x0 = x[0::3]
e0 = np.random.normal(0.0, 0.1, len(x0))
y0 = sin_fun(x0, p_true) + e0

p_start = np.zeros(p_true.shape)
p_start = np.array([0.1, 0.1, 3.0])
res = least_squares(error, p_start, method='lm')
```

https://docs.scipy.org/doc/scipy/tutorial/optimize.html

```python
e = error(res.x)
y = sin_fun(x, res.x)
y0_est = sin_fun(x0, res.x)

fig = plt.figure(figsize=(6,4))
ax = fig.gca()

# Plot offset lines with different colors
```

...



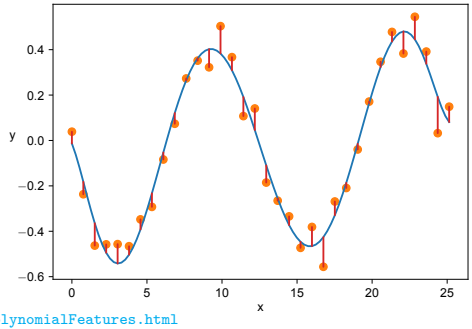
Non-linear regression: Example using sklearn (Polynomial features)

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import make_pipeline

def sin_fun(
    x: np.ndarray,
    p: np.ndarray,
):
    a = p[0]; b = p[1]; c = p[2]
    y = a * np.sin(b * x + c)
    return y

p_true = np.array([0.5, 0.5, np.pi])

x = np.linspace(0, 8*np.pi, 100, endpoint=True)
x0 = x[0::3]
e0 = np.random.normal(0.0, 0.1, len(x0))
y0 = sin_fun(x0, p_true) + e0
```

```python
poly = make_pipeline(PolynomialFeatures(7),
    LinearRegression())
poly.fit(x0[:, np.newaxis], y0)
y0_est = poly.predict(x0[:, np.newaxis])
y = poly.predict(x[:, np.newaxis])

fig = plt.figure(figsize=(6,4))
ax = fig.gca()
```

Dataset with one feature, x.

Expanding this using **polynomial features** of degree 7 gives eight features $1,\ x,\ x^2, \cdots, x^7$



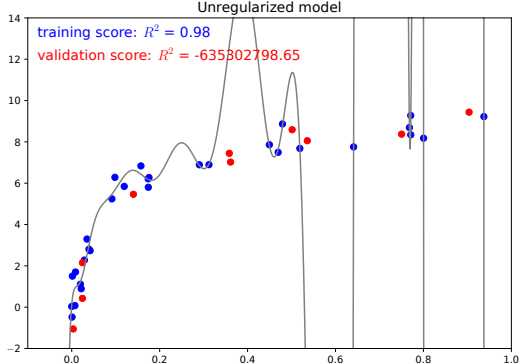
https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.PolynomialFeatures.html

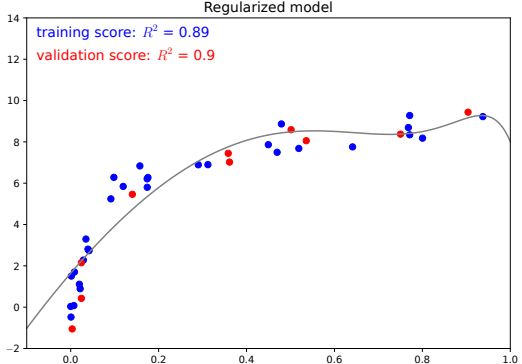
Regularization

Regularization

Regularization: Problem

If we have too many parameters or too *flexible* of a model we risk **overfitting**. This means that our model will near perfectly predict the data used during fitting but will fail to generalize and perform poorly on new data. Solution is to add regularization.





Regularization: Regularization term

We add an extra term to our error function.

$$\arg\min_{p} \underbrace{e(p; x, y)}_{\text{Data term}} + \underbrace{\alpha r(p)}_{\text{Regularization term}}$$

where α is a **hyper parameter** that we set depending on how much regularization we want.

This is sometimes called a **penalty function**.

Two common regularization methods used are:

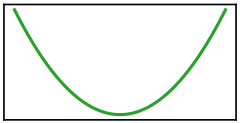
- **Ridge regression**
- **Lasso regression**

Regularization: Ridge and Lasso

Both ridge and lasso regularization **penalize large parameter values**.

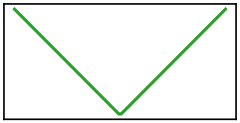
Ridge regression

$$r(p) = \sum_i p_i^2$$



Lasso regression

$$r(p) = \sum_i |p_i|$$



Regularization: Linear regression with regularization

Our **original error function** for the **straight line** looked like

$$e(k, m; x, y) = \sum_i (kx_i + m - y_i)$$

If we **add ridge regression** we get

$$e(k, m; x, y) = \sum_i (kx_i + m - y_i) + \alpha(k^2 + m^2)$$

There are **many other types of regularization** used in image processing, machine learning and deep learning.

https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Ridge.html
https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Lasso.html

Logistic regression

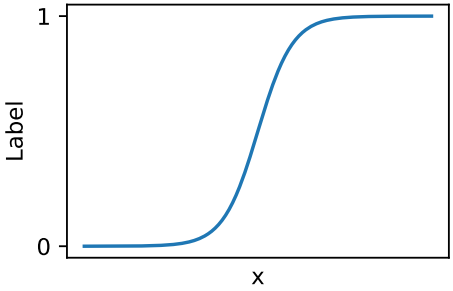
Logistic regression

Logistic regression: Classification

Looked at continuous input → continuous output.

Now look at **continuous input** → **binary output**, *i.e.* **classification**.

These are often modeled with some s-shaped **logistic** function.



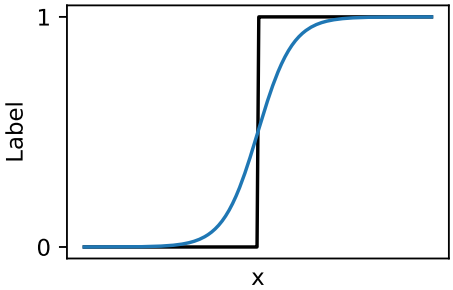
https://en.wikipedia.org/wiki/Logistic_function

Logistic regression: Sigmoid

Why not use a sharp cutoff function, *i.e. Heaviside function*, since we want binary output?

Want that are **continuous** such that we can **calculate derivatives** to find an extreme value optimum in our error function.

Another benefit is that we can **interpret** the output **as a probability** of belonging to a class or not.



$$\sigma(x) = \frac{1}{1 + e^{-(kx+m)}}$$

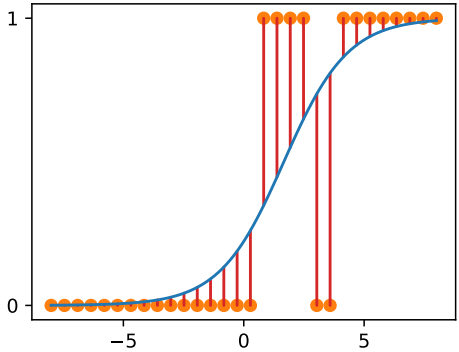
Logistic regression: Error function

For **regression problems** we used **sum of squared distances**. We can use this for logistic regression as well.

However, for **classification** a more common error metric is binary **cross entropy**.

This error metric is more rooted in the **probabilistic interpretation**.

$$\text{BCE}(y, p) = -\frac{1}{N}\sum_{i=1}^{N}(y_i \log(p_i) + (1 - y_i)\log(1 - p_i))$$



Distance error like this not typically used for classification.

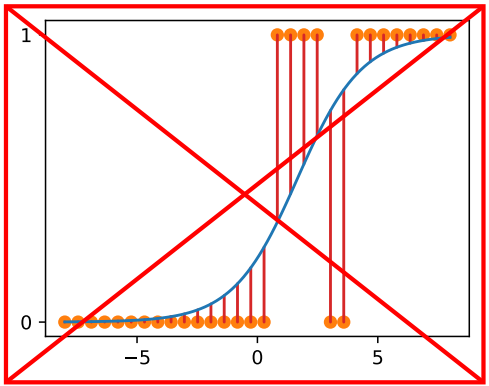
Logistic regression: Error function

For **regression problems** we used **sum of squared distances**. We can use this for logistic regression as well.

However, for **classification** a more common error metric is binary **cross entropy**.

This error metric is more rooted in the **probabilistic interpretation**.

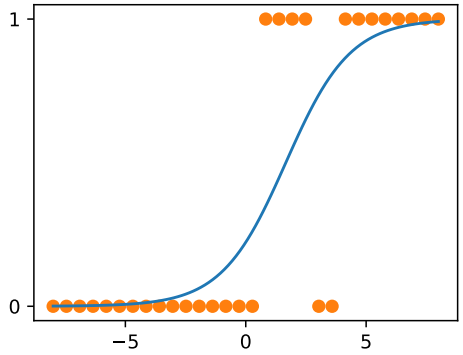
$$\text{BCE}(y, p) = -\frac{1}{N}\sum_{i=1}^{N}(y_i \log(p_i) + (1 - y_i)\log(1 - p_i))$$



Distance error like this not typically used for classification.

Logistic regression: Python (linear, sklearn)

```python
import numpy as np
from sklearn.linear_model import
    LogisticRegression
import matplotlib.pyplot as plt
# Generate some 1D classification data
rng = np.random.default_rng(seed=0)
X = np.linspace(-8, 8, 30, endpoint=True)
v = np.tanh(X) + rng.uniform(-1.0, 1.0, X.
    shape)
y = np.zeros(v.shape)
y[v > 0.5] = 1
X = X.reshape(-1, 1)
# Fit logistic regression classifier
lr = LogisticRegression()
lr.fit(X, y)
# Create sigmoid array for plot
k = lr.coef_[0]
m = lr.intercept_
x_curve = np.linspace(-8, 8, 100, endpoint=
    True)
y_curve = 1.0 / (1.0 + np.exp(-(k * x_curve +
    m)))
#Plot
ax = plt.figure(figsize=(4,3)).gca()
ax.scatter(X, y, color='tab:orange')
ax.set_yticks([0, 1])
ax.plot(x_curve, y_curve)
#plt.show()
```



Logistic regression: Multivariate classification

One **single feature** will usually **not** be a very **strong predictor** in the real world.

Similarly to multilinear regression,can fit a **logistic linear classifier** to data with **multiple features**.

In 2D

$$\sigma(x_1, x_2) = \frac{1}{1 + e^{-(k_1 x_1 + k_2 x_2 + m)}}$$

Or more generally

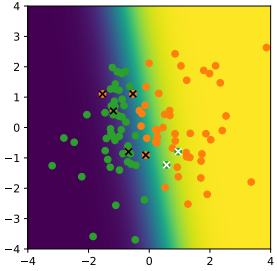
$$\sigma(x_1, \cdots, x_M) = \frac{1}{1 + e^{-(k_1 x_1 + \cdots + k_M x_M + m)}}$$

In **scikit learn** this is just as **easy** as 1D logistic regression!

Logistic regression: Python (2 feature dimensions) (1/2)

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import
    make_classification
from sklearn.linear_model import
    LogisticRegression
# Create data and fit logitistic model
X, y = make_classification(n_samples=100,
                           n_features=2,
                           n_informative=2,
                           n_redundant=0,
                           random_state=0)
lr = LogisticRegression()
lr.fit(X, y)

# Predict if sample is class or not
y_pred = lr.predict(X)
fp =  y_pred & ~y # False positives
fn = ~y_pred &  y # False negatives

# Create 2D decision function
m = lr.intercept_
k1 = lr.coef_.flatten()[0]
k2 = lr.coef_.flatten()[1]
x1 = np.linspace(-4, 4, 200, endpoint=True)
x2 = np.linspace(-4, 4, 200, endpoint=True)
X1, X2 = np.meshgrid(x1, x2)
z = 1.0 / (1.0 + np.exp( -1.0 * (k1 * X1 + k2
    * X2 + m) ))
```

```python
ax = plt.figure(figsize=(4,4)).gca()
ax.scatter(X[y==1,0], X[y==1,1], color='tab:
    orange')
ax.scatter(X[y==0,0], X[y==0,1], color='tab:
    green')
ax.scatter(X[fp==1,0], X[fp==1,1], color='
    white', marker='x')
ax.scatter(X[fn==1,0], X[fn==1,1], color='k',
    marker='x')
ax.imshow(z, origin='lower', extent=[-4, 4,
    -4, 4])
#plt.show()
```

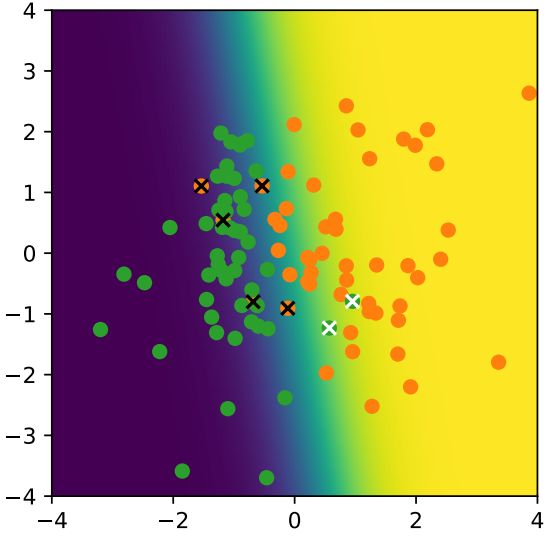


Logistic regression: Python (2 feature dimensions) (2/2)

Logistic regression gives a **soft separating boundary along a line**.

(In higher dimensions this line would be a separating hyperplane.)

The wrongly classified samples are indicated by an x.



Validation of classification

Validation of classification: Basic metrics

How do we know how good our **classification** model is?

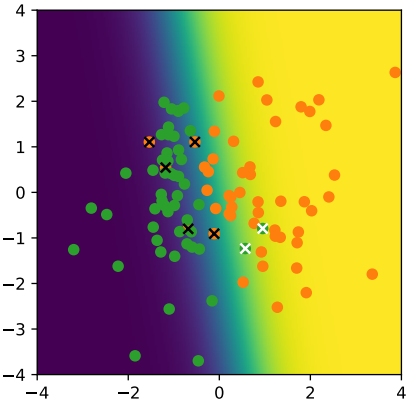
Let us start by defining four basic **validation scores**.

TP True positives

FP False positives (x)

TN True negatives

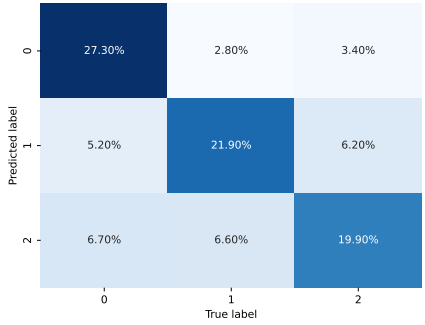
FN False negatives (x)



Validation of classification: Confusion matrix

```python
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import
    make_classification
from sklearn.linear_model import
    LogisticRegression
from sklearn.metrics import confusion_matrix
# Create data and fit logitistic model
X, y = make_classification(n_samples=1000,
                           n_classes=3,
                           n_features=4,
                           n_informative=4,
                           n_redundant=0,
                           random_state=0)
lr = LogisticRegression()
lr.fit(X, y)
y_pred = lr.predict(X)
# Plot confusion matrix using seaborn
cm = confusion_matrix(y, y_pred)
sns.heatmap(cm/np.sum(cm),
            annot=True,
            fmt='.2%',
            cmap='Blues',
            cbar=False)
plt.xlabel('True label')
plt.ylabel('Predicted label')
#plt.show()
```

These metrics can be summarized in a **confusion matrix**.



(The linear classifier struggles to separate these three classes.)

https://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusion_matrix.html

Validation of classification: Metrics

The basic validation scores can be used to define some commonly used metrics.

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

$$\text{F1 score)} = \frac{2\text{TP}}{2\text{TP} + \text{FP} + \text{FN}}$$

TP True positives

FP False positives

TN True negatives

FN False negatives

https://scikit-learn.org/stable/api/sklearn.metrics.html
https://scikit-learn.org/stable/modules/generated/sklearn.metrics.recall_score.html
https://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision_score.html
https://scikit-learn.org/stable/modules/generated/sklearn.metrics.f1_score.html

Validation of regression

Validation of regression: Metrics

How do we know how good our regression model is?

sklearn has **plenty of metrics** implemented.

E.g. the R^2 score (coefficient of determination).

Harder to interpret than for classification.

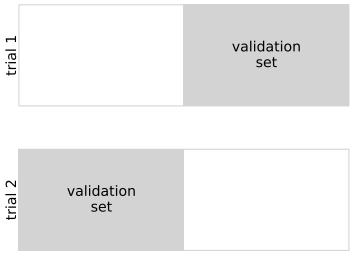
d2_absolute_error_score	D^2 regression score function, fraction of absolute error explained.
d2_pinball_score	D^2 regression score function, fraction of pinball loss explained.
d2_tweedie_score	D^2 regression score function, fraction of Tweedie deviance explained.
explained_variance_score	Explained variance regression score function.
max_error	The max_error metric calculates the maximum residual error.
mean_absolute_error	Mean absolute error regression loss.
mean_absolute_percentage_error	Mean absolute percentage error (MAPE) regression loss.
mean_gamma_deviance	Mean Gamma deviance regression loss.
mean_pinball_loss	Pinball loss for quantile regression.
mean_poisson_deviance	Mean Poisson deviance regression loss.
mean_squared_error	Mean squared error regression loss.
mean_squared_log_error	Mean squared logarithmic error regression loss.
mean_tweedie_deviance	Mean Tweedie deviance regression loss.
median_absolute_error	Median absolute error regression loss.
r2_score	R^2 (coefficient of determination) regression score function.
root_mean_squared_error	Root mean squared error regression loss.
root_mean_squared_log_error	Root mean squared logarithmic error regression loss.

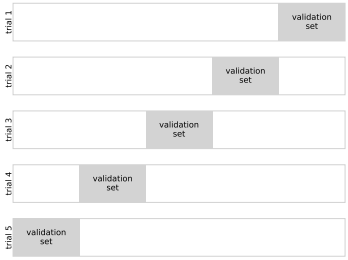
https://scikit-learn.org/stable/api/sklearn.metrics.html
https://scikit-learn.org/stable/modules/generated/sklearn.metrics.r2_score.html

Validation of regression: Holdout sets

Split data into **train** and **test** sets.

(Extreme case when we put only one data point in the test set and do this for each data point in the data set.
This way of testing is called **leave one out**.)





https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html
https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.cross_validate.html
https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.LeaveOneOut.html

Validation of regression: Splitting data into train and test

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_regression
from sklearn.linear_model import
     LinearRegression
from sklearn.model_selection import
     train_test_split
from sklearn.metrics import r2_score

# Create data and split into subsets
X, y = make_regression(n_samples=100,
     n_features=1, noise=5.0)
X_train, X_test, y_train, y_test =
     train_test_split(X, y, test_size=0.2)
# Create linear regression
lm = LinearRegression()
lm.fit(X_train, y_train)
# Apply model on the train and test data
y_test_pred = lm.predict(X_test)
y_train_pred = lm.predict(X_train)
# Compute metrics and report
r2_test  = r2_score(y_test, y_test_pred)
r2_train = r2_score(y_train, y_train_pred)
print(f"Training score: {r2_train}")
print(f"Test score:     {r2_test}")
```

We can easily split data into training and testing sets using sklearn.

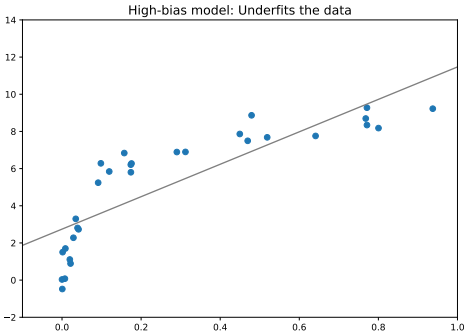
After model fitting we compute and report the R^2 score of each.

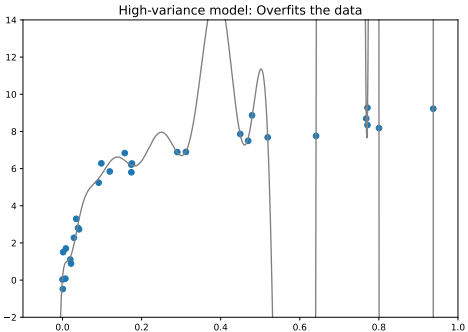
```
>> Training score: 0.9901669782979793
>> Test score:     0.9892174984807733
```

Validation of regression: Bias and variance

Trade off between bias and variance in the model.

I.e., how well does the model generalize and perform on the test data and how well does it fit the training data.

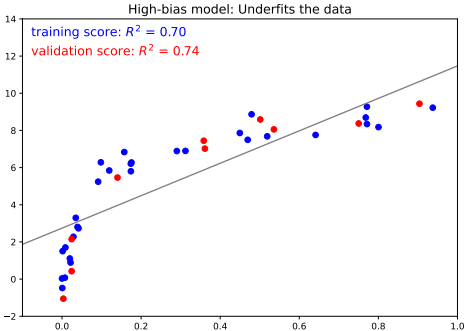


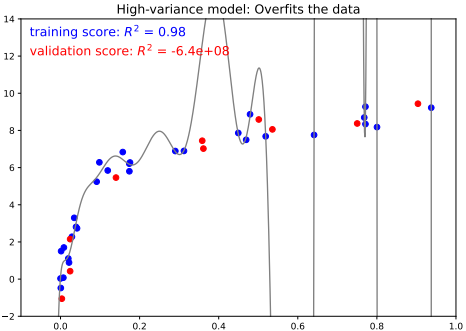


Validation of regression: Bias and variance

Trade off between bias and variance in the model.

I.e., how well does the model generalize and perform on the test data and how well does it fit the training data.



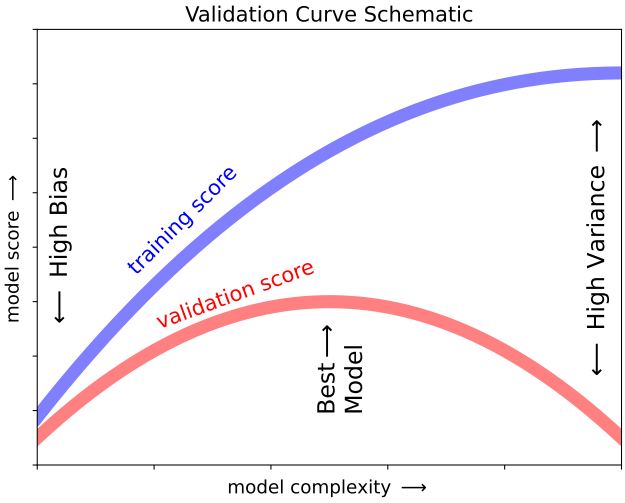


Validation of regression: Validation

Due to overfitting the training score will increase as the model complexity increases.

The validation score will increase at first and then start to decrease as the model starts to have rapid oscillations.

The optimal model complexity is found at the peak of the validation curve.

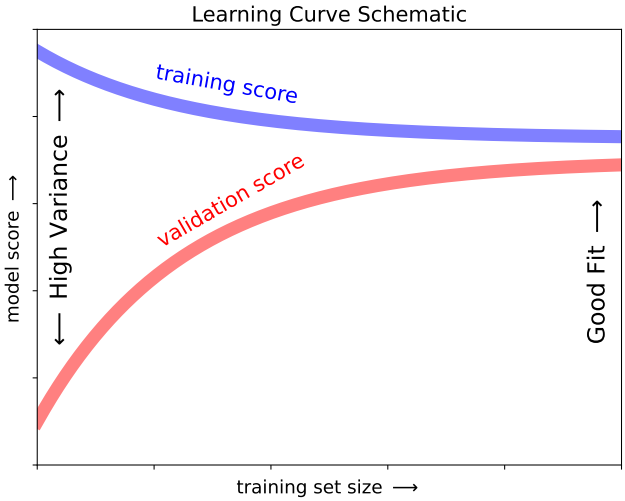


Validation of regression: Validation

Common that both training score and validation score converges to some value after the training data ha reached a certain size.

At this point adding more data will not improve the model.

Model performance can be improved by increased complexity.



Machine learning

Machine learning: Conclusions

Minimize error

Want to find parameters of our model that minimize error when compared to data.

Validation

We should validate our models by at least splitting data into training and test sets.