STK-IN1050 - Statistikk for informatikere

21. Klassifikasjon, del II

Matematisk institutt

2026-05-04

Logistisk regresjon

I logistisk regresjon modellerer vi log-odds som en lineær funksjon av forklaringsvariabler:

$$\log\left(\frac{P(Y=1 \mid \mathbf{x})}{1-P(Y=1 \mid \mathbf{x})}\right) = \beta_0 + \beta_1 x_1 + \cdots + \beta_k x_k$$

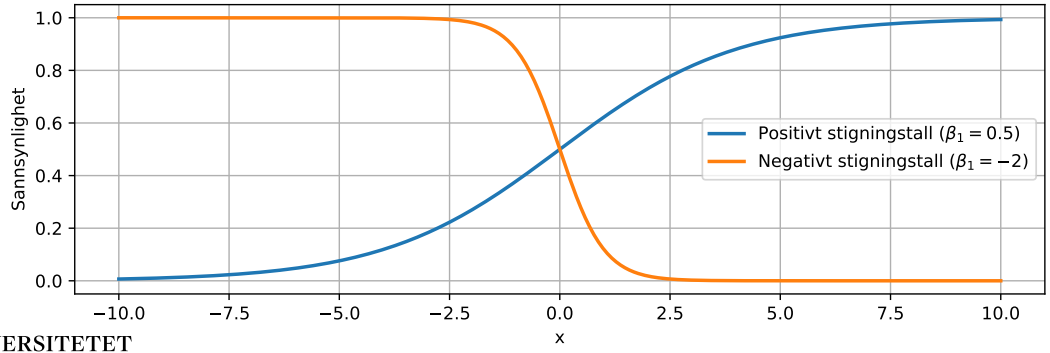
Vi skal nå diskutere logistisk regresjon i praksis:

- estimering
- tolkning av koeffisienter/odds
- prediksjon
- modellvalg og evaluering
- naiv Bayes som et alternativ

Tolkning av sammenheng mellom x og y

Anta at vi har en forklaringsvariabel x. Da blir modellen

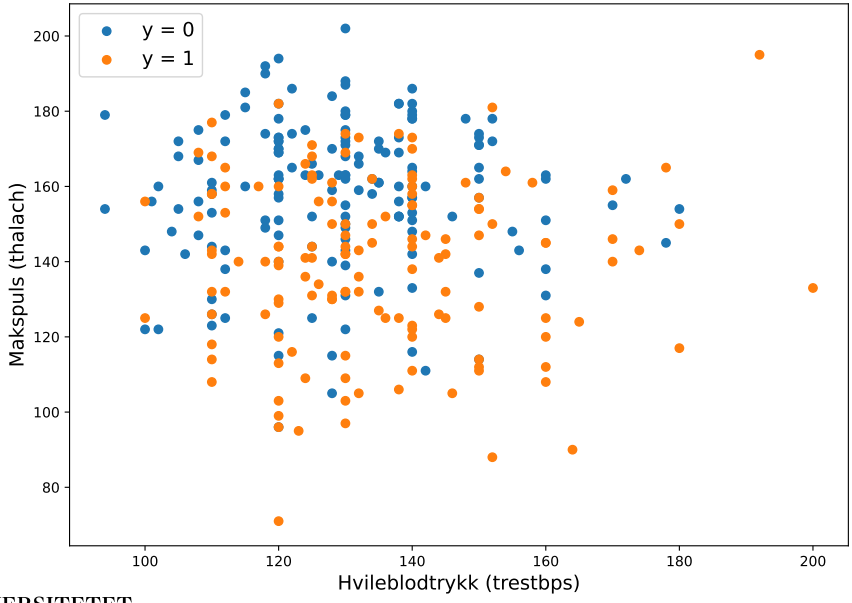
$$P(Y = 1 \mid x) = \frac{\exp(\beta_0 + \beta_1 x)}{1 + \exp(\beta_0 + \beta_1 x)}$$



Datasett: 302 pasienter, 13 forklaringsvariabler

```
HDdata = pd.read_csv("../data/Heart-disease-Cleveland.txt", sep=",")
HDdata.columns = ["age","sex","cp","trestbps","chol","fbs","restecg",
                  "thalach","exang","oldpeak","slope","ca","thal","num"]
HDdata["y"] = (HDdata["num"]>0).astype(int)
HDdata = HDdata.drop(columns=["num"])
factor_cols = ["sex", "cp","restecg","exang","slope","thal"]
HDdata[factor_cols] = HDdata[factor_cols].astype("category")
print(HDdata.head(8))
```

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	y
0	67.0	1.0	4.0	160.0	286.0	0.0	2.0	108.0	1.0	1.5	2.0	3.0	3.0	1
1	67.0	1.0	4.0	120.0	229.0	0.0	2.0	129.0	1.0	2.6	2.0	2.0	7.0	1
2	37.0	1.0	3.0	130.0	250.0	0.0	0.0	187.0	0.0	3.5	3.0	0.0	3.0	0
3	41.0	0.0	2.0	130.0	204.0	0.0	2.0	172.0	0.0	1.4	1.0	0.0	3.0	0
4	56.0	1.0	2.0	120.0	236.0	0.0	0.0	178.0	0.0	0.8	1.0	0.0	3.0	0
5	62.0	0.0	4.0	140.0	268.0	0.0	2.0	160.0	0.0	3.6	3.0	2.0	3.0	1
6	57.0	0.0	4.0	120.0	354.0	0.0	0.0	163.0	1.0	0.6	1.0	0.0	3.0	0
7	63.0	1.0	4.0	130.0	254.0	0.0	2.0	147.0	0.0	1.4	2.0	1.0	7.0	1


y = 0
y = 1
Makspuls (thalach)
Hvileblodtrykk (trestbps)
200
180
160
140
120
100
80
100
120
140
160
180
200

Estimering av modellen

Anta $\mathbf{x} \in \mathbb{R}^p$ og $Y \in \{0, 1\}$. Da kan vi skrive modellen som

$$\Pr(Y = 1 \mid \mathbf{x}) = \frac{\exp(\beta_0 + \boldsymbol{\beta}^T \mathbf{x})}{1 + \exp(\beta_0 + \boldsymbol{\beta}^T \mathbf{x})}$$

og

$$\Pr(Y = 0 \mid \mathbf{x}) = 1 - \Pr(Y = 1 \mid \mathbf{x}) = \frac{1}{1 + \exp(\beta_0 + \boldsymbol{\beta}^T \mathbf{x})}$$

$\Rightarrow$ For en observasjon $(\mathbf{x}_i, y_i)$ kan vi koble sammen $y_i = 0$ og $y_i = 1$ i én formel

$$\Pr(Y_i = y_i \mid \mathbf{x}_i) = \frac{\exp\left(y_i(\beta_0 + \boldsymbol{\beta}^T \mathbf{x}_i)\right)}{1 + \exp(\beta_0 + \boldsymbol{\beta}^T \mathbf{x}_i)}$$

Maksimum likelihood (ML) estimering

Anta at vi observerer data $\mathcal{D} = \{(y_1, \mathbf{x}_1), \ldots, (y_n, \mathbf{x}_n)\}$.

Hvis disse er uavhengige blir likelihood-funksjonen:

$$L(\beta_0, \boldsymbol{\beta} \mid \mathcal{D}) = \prod_{i=1}^{n} \Pr(Y_i = y_i \mid \mathbf{x}_i) = \prod_{i=1}^{n} \frac{\exp\left(y_i(\beta_0 + \boldsymbol{\beta}^T \mathbf{x}_i)\right)}{1 + \exp(\beta_0 + \boldsymbol{\beta}^T \mathbf{x}_i)}$$

ML estimering: finner $\hat{\beta}_0, \hat{\boldsymbol{\beta}}$ som maksimerer L (eller $\log L$)

Ingen lukket formel $\rightarrow$ numeriske algoritmer

I praksis: f.eks. `statsmodels.logit` i Python

Eksempel: enkel tilpasning i Python

```
import statsmodels.formula.api as smf

model = smf.logit('y ~ trestbps + thalach', data=HDdata)
result = model.fit()
print(result.params.round(3).to_dict())
```

```
Optimization terminated successfully.
         Current function value: 0.584775
         Iterations 6
{'Intercept': 3.909, 'trestbps': 0.019, 'thalach': -0.044}
```

Gir oss parametre: $\hat{\beta}_0$, $\hat{\beta}_{\texttt{trestbps}}$, $\hat{\beta}_{\texttt{thalach}}$

Tolkning via odds og odds ratio

Vi har i forrige forelesning definert odds:

$$\text{odds}(\mathbf{x}) = \frac{\Pr(Y = 1 \mid \mathbf{x})}{\Pr(Y = 0 \mid \mathbf{x})} = \exp(\beta_0 + \boldsymbol{\beta}^T \mathbf{x})$$

og log-odds (logit):

$$\log \text{odds}(\mathbf{x}) = \beta_0 + \boldsymbol{\beta}^T \mathbf{x}$$

Dvs. modellen er lineær i log-odds, ikke direkte i sannsynlighet

Odds ratio og tolkning av β_j

Anta av vi ønsker å sammenlikne $\mathbf{x}$ og $\mathbf{x}^*$ som er like, unntatt $x_j^* = x_j + 1$.

Forhold mellom odds:

$$\frac{\text{odds}(\mathbf{x}^*)}{\text{odds}(\mathbf{x})} = \frac{\exp\left(\beta_0 + \beta_1 x_1 + \ldots + \beta_j(x_j + 1) + \ldots \beta_p x_p\right)}{\exp\left(\beta_0 + \beta_1 x_1 + \ldots + \beta_j x_j + \ldots \beta_p x_p\right)} = \exp(\beta_j)$$

$\Longrightarrow$ Når x_j øker med 1 (alt annet likt), multipliseres odds med faktoren $\exp(\beta_j)$

Odds ratio og tolkning av β_j

Vi har

$$\frac{\text{odds}(\mathbf{x}^*)}{\text{odds}(\mathbf{x})} = \exp(\beta_j)$$

```
result = model.fit(disp=False)
print(result.params.round(3).to_dict())
```

```
{'Intercept': 3.909, 'trestbps': 0.019, 'thalach': -0.044}
```

```
import numpy as np
or_params = np.exp(result.params)
print(or_params.round(2).to_dict())
```

```
{'Intercept': 49.87, 'trestbps': 1.02, 'thalach': 0.96}
```

Konfidensintervall for β_j og odds ratio

Vi antar at $\hat{\beta}_j \approx N(\beta_j, \hat{\sigma}^2_{\hat{\beta}_j})$.

Da har vi standard formel for konfidensintervall for β_j:

$$\hat{\beta}_j \pm z_{\alpha/2}\hat{\sigma}_{\hat{\beta}_j}$$

Det gir tilsvarende intervall for odds ratio e^{β_j}:

$$\left[e^{\hat{\beta}_j - z_{\alpha/2}\hat{\sigma}_{\hat{\beta}_j}}, e^{\hat{\beta}_j + z_{\alpha/2}\hat{\sigma}_{\hat{\beta}_j}}\right]$$

Konfidensintervall for β_j og odds ratio

```
from IPython.display import display, Math

ci = result.conf_int(alpha=0.05).round(3)
display(Math(
    r"95\% \text{ confidence intervals for } "
    r"\beta_j"
))
display(ci)
```

95% confidence intervals for β_j

	0	1
Intercept	1.264	6.554
trestbps	0.004	0.033
thalach	-0.056	-0.031

```
or_ci = np.exp(ci).round(2)
display(Math(
    r"95\% \text{ confidence intervals for } "
    r"\exp(\beta_j)"
))
display(or_ci)
```

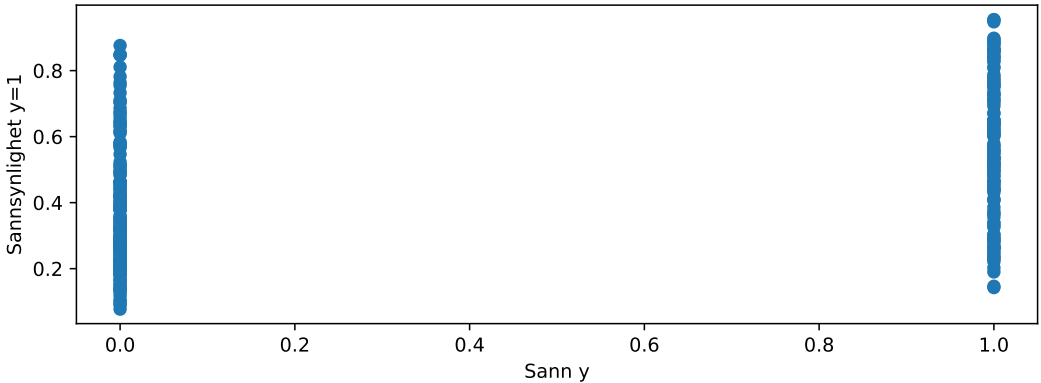
95% confidence intervals for $\exp(\beta_j)$

	0	1
Intercept	3.54	702.05
trestbps	1.00	1.03
thalach	0.95	0.97

Predikert sannsynlighet

For ny observasjon **x**:

$$\hat{p}(\mathbf{x}) = \Pr(Y = 1 \mid \mathbf{x}, \hat{\beta}_0, \hat{\boldsymbol{\beta}})$$



Fra sannsynlighet til klasse

Klassifikasjonsregel:

$$\hat{y}(\mathbf{x}) = \arg \max_{k \in \{0,1\}} \Pr(Y = k \mid \mathbf{x})$$

For to klasser:

$$\hat{y}(\mathbf{x}) = \begin{cases} 1 & \text{hvis } \Pr(Y = 1 \mid \mathbf{x}) > 0.5 \\ 0 & \text{ellers} \end{cases}$$

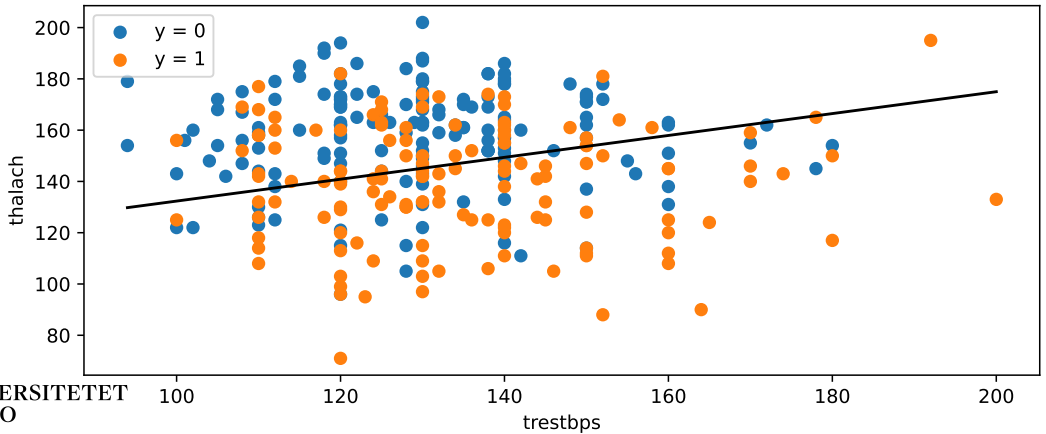
Ekvivalent (se kapitel 12 i bok, eller tavla):

$$\hat{y}(\mathbf{x}) = \begin{cases} 1 & \text{hvis } \beta_0 + \boldsymbol{\beta}^T \mathbf{x} > 0 \\ 0 & \text{ellers} \end{cases}$$

Beslutningsgrense i 2D

For modell med to forklaringsvariabler har vi en lineær beslutningsgrense

$$\beta_0 + \beta_1 \texttt{trestbps} + \beta_2 \texttt{thalach} = 0$$



Hvor mange variabler skal vi ha med?

Problem:

- Mange potensielle forklaringsvariabler
- Ønsker enklest mulig modell som forklarer data godt

Én mulig fremgangsmåte:

- Start med “stor” modell med 8 variabler
- Bruk *p*-verdier til å fjerne minst viktige variabler stegvis

Modelloutput og *p*-verdier

Først: deler data i treningsett og testsett

```
import numpy as np
import statsmodels.formula.api as sm

# Create a random boolean mask
np.random.seed(42)
mask = np.random.rand(len(HDdata)) < 0.5   # ~50% train, 50% test

df_train = HDdata[mask]
df_test  = HDdata[~mask]
```

Vi kjører nå modellen på treningsdataene og bruker enkel terskel (f.eks. 0.05) for å beholde variabler, hvor variabelen med høyest *p*-verdi fjernes først.

Modell med 8 variabler

```
model = sm.logit("y ~ age+sex+trestbps+chol+fbs+thalach+exang+oldpeak",
                 data=df_train)
result = model.fit(disp=False)
display(result.summary2().tables[1].iloc[:,:4].round(3))
```

	Coef.	Std.Err.	z	P>|z|
Intercept	-0.316	3.266	-0.097	0.923
sex[T.1.0]	1.334	0.538	2.478	0.013
exang[T.1.0]	1.318	0.479	2.754	0.006
age	0.028	0.029	0.941	0.347
trestbps	0.014	0.014	0.996	0.319
chol	0.004	0.004	0.863	0.388
fbs	-1.058	0.716	-1.478	0.139
thalach	-0.040	0.013	-3.081	0.002
oldpeak	0.699	0.241	2.904	0.004

Modell med 7 variabler

```
model = sm.logit("y ~ age+sex+trestbps+fbs+thalach+exang+oldpeak",
                 data=df_train)
result = model.fit(disp=False)
display(result.summary2().tables[1].iloc[:,:4].round(3))
```

	Coef.	Std.Err.	z	P>|z|
Intercept	0.224	3.162	0.071	0.944
sex[T.1.0]	1.249	0.523	2.389	0.017
exang[T.1.0]	1.328	0.478	2.779	0.005
age	0.034	0.028	1.210	0.226
trestbps	0.013	0.014	0.892	0.373
fbs	-1.062	0.707	-1.502	0.133
thalach	-0.038	0.013	-3.015	0.003
oldpeak	0.716	0.244	2.937	0.003

Modell med 6 variabler

```
model = sm.logit("y ~ age+sex+fbs+thalach+exang+oldpeak",
                 data=df_train)
result = model.fit(disp=False)
display(result.summary2().tables[1].iloc[:,:4].round(3))
```

	Coef.	Std.Err.	z	P>|z|
Intercept	1.558	2.768	0.563	0.574
sex[T.1.0]	1.185	0.510	2.321	0.020
exang[T.1.0]	1.356	0.476	2.849	0.004
age	0.037	0.028	1.337	0.181
fbs	-0.872	0.658	-1.324	0.185
thalach	-0.038	0.013	-2.995	0.003
oldpeak	0.756	0.237	3.192	0.001

Modell med 5 variabler

```
model = sm.logit("y ~ age+sex+thalach+exang+oldpeak",
                 data=df_train)
result = model.fit(disp=False)
display(result.summary2().tables[1].iloc[:,:4].round(3))
```

	Coef.	Std.Err.	z	P>|z|
Intercept	1.566	2.710	0.578	0.563
sex[T.1.0]	1.139	0.507	2.249	0.025
exang[T.1.0]	1.411	0.472	2.988	0.003
age	0.031	0.027	1.127	0.260
thalach	-0.036	0.012	-3.005	0.003
oldpeak	0.749	0.234	3.207	0.001

Modell med 4 variabler

```
model = sm.logit("y ~ sex+thalach+exang+oldpeak",
                 data=df_train)
result = model.fit(disp=False)
display(result.summary2().tables[1].iloc[:,:4].round(3))
```

	Coef.	Std.Err.	z	P>|z|
Intercept	3.826	1.835	2.085	0.037
sex[T.1.0]	1.039	0.493	2.106	0.035
exang[T.1.0]	1.418	0.470	3.017	0.003
thalach	-0.039	0.012	-3.404	0.001
oldpeak	0.777	0.234	3.320	0.001

Variablene er kjønn, makspuls, trening, indikator for redusert oksygentilførsel i blodet.

Oppsummering: Fra stor til liten modell

Vi bruker en stegvis “backward elimination”:

- Fjern variabel med størst *p*-verdi > 0.05 (f.eks. `chol`)
- Estimer ny modell og sjekk *p*-verdier
- Gjenta til alle variabler har *p*-verdi < 0.05

Vi ender med modell: `y ~ sex + thalach + exang + oldpeak`

Dette er en enkel, “naiv” seleksjonsmetode, men den virker OK i praksis

Hvor godt fungerer modellen?

Gitt en regel til å gå fra $\hat{p}(\mathbf{x})$ til $\hat{y}(\mathbf{x})$ og data $(\mathbf{x}_i, y_i)$ kan vi måle hvor godt modellen fungerer.

Et slikt mål er **ACC**, eller **andel korrekt klassifisert**:

$$\text{ACC} = \frac{1}{n} \sum_{i=1}^{n} I(\hat{y}(\mathbf{x}_i) = y_i)$$

MEN, der er problematisk å bruke samme data til både trening og testing:

- Vi blir for optimistiske (“vi jukser”)
- Modellen er tilpasset akkurat disse dataene $\Longrightarrow$ overtilpasning

Trenings- og testdata

Del data tilfeldig i to deler:

- Treningssett: lær parametre og velg modell
- Testsett: evaluer endelig modell

```
import numpy as np
import statsmodels.formula.api as sm

# Create a random boolean mask
np.random.seed(42)
mask = np.random.rand(len(HDdata)) < 0.5   # ~50% train, 50% test

df_train = HDdata[mask]
df_test  = HDdata[~mask]

model = sm.logit("y ~ sex+exang+thalach+oldpeak", data=df_train)
result = model.fit(disp=False)
```

Trenings- og testdata

```
# Accuracy based on the same data for training and test
p_train = result.predict(df_train)
y_train_hat = (p_train >= 0.5).astype(int)
y_train_true = df_train["y"]
ACCtrain_percent = 100*np.mean(y_train_hat == y_train_true)
print(f"Training-nøyaktighet: {ACCtrain_percent:.2f}%")

# Accuracy based on independent test data
p_test = result.predict(df_test)
y_test_hat = (p_test >= 0.5).astype(int)
y_test_true = df_test["y"]
ACCtest_percent = 100*np.mean(y_test_hat == y_test_true)
print(f"Test-nøyaktighet: {ACCtest_percent:.2f}%")
```

```
Training-nøyaktighet: 80.00%
Test-nøyaktighet: 77.07%
```

- Training-nøyaktighet > Test-nøyaktighet (som regel)
- Test-nøyaktighet gir bedre estimat på ytelse på *nye* data

Fallgruver ved dårlig splitting

Hva kan gå galt hvis vi tar “første halvdel” som trening, “andre halvdel” som test?

Kan det være situasjoner hvor dette faktisk er bra?

Hva kan være en bedre måte å dele opp dataene i andre tilfeller?

Kryssvalidering: bedre utnyttelse av data

1. Del datasettet i K folds (f.eks. $K = 5$ eller 10)
2. For hver fold k:
   a) Tren modell på de $K - 1$ andre foldene
   b) Test på fold k

Fordeler:

- Hver observasjon brukes til test én gang og til trening $K - 1$ ganger
- Bedre utnyttelse av lite/moderat datasett

Ulemper:

- Må estimere modellen K ganger (dyrt hvis modellen er tung)
- Må ha nok data

Eksempel: 5-fold kryssvalidering i praksis

```python
import numpy as np
import pandas as pd
import statsmodels.formula.api as smf

def perform_cross_validation(df, train_idx, test_idx, fold_number=None):
    df_train = df.iloc[train_idx]
    df_test  = df.iloc[test_idx]
    # Fit logistic regression on training data
    model = smf.logit("y ~ sex+exang+thalach+oldpeak", data=df_train)
    result = model.fit(disp=False)
    # Predict probabilities on test data
    p_test = result.predict(df_test)
    # Convert to class predictions with threshold 0.5
    y_test_hat = (p_test >= 0.5).astype(int)
    y_test_true = df_test["y"].values
    # Compute accuracy for this fold
    acc_k = np.mean(y_test_hat == y_test_true)
    if fold_number is not None:
        print(f"Fold {fold_number} accuracy: {acc_k:.3f}")
    return acc_k
```

```python
df = HDdata[["y", "sex", "exang", "thalach", "oldpeak"]] # relevant data
K = 5  # number of folds
np.random.seed(42) # set seed
indices = np.arange(len(df)) # find indices
np.random.shuffle(indices) # shuffle indices
folds = np.array_split(indices, K) # split into folds
cv_accuracies = [] # list of results
for k in range(K):
    test_idx = folds[k] # test
    train_idx = np.concatenate([folds[j] for j in range(K) if j != k]) # train
    acc_k = perform_cross_validation(df, train_idx, test_idx, fold_number=k+1) # run model
    cv_accuracies.append(acc_k) # add results to list
# Overall cross-validated accuracy
mean_acc = np.mean(cv_accuracies)
std_acc  = np.std(cv_accuracies, ddof=1)
print(f"\nMean CV accuracy ({K}-fold): {mean_acc:.3f} ± {std_acc:.3f}")
```

```
Fold 1 accuracy: 0.770
Fold 2 accuracy: 0.770
Fold 3 accuracy: 0.783
Fold 4 accuracy: 0.783
Fold 5 accuracy: 0.750

Mean CV accuracy (5-fold): 0.772 ± 0.014
```

Naiv Bayes som alternativ klassifikator

I logistisk regresjon modellerer vi direkte:

$$\Pr(Y \mid \mathbf{x})$$

Alternativ: modellere simultanfordelingen til X og Y:

$$p(\mathbf{X}, Y) = \Pr(Y)\, p(\mathbf{X} \mid Y)$$

Bruk Bayes' formel:

$$\Pr(Y = k \mid \mathbf{x}) = \frac{\Pr(Y = k) p(\mathbf{x} \mid Y = k)}{p(\mathbf{x})}, \quad k = 0, 1$$

Klassifikasjon:

$$\hat{y}(\mathbf{x}) = \arg \max_{k \in \{0,1\}} \Pr(Y = k) p(\mathbf{x} \mid Y = k)$$

Gaussisk Naive Bayes

Naiv antakelse:

$$p(\mathbf{x} \mid Y) = \prod_{j=1}^{p} p(x_j \mid Y)$$

Hver $x_j \mid Y = k$ antas normalfordelt:

$$x_j \mid Y = k \sim N(\mu_{k,j}, \sigma^2_{k,j})$$

I praksis:

- Beregner estimater for $\mu_{k,j}$, $\sigma^2_{k,j}$ fra data
- Bruker Bayes for å få $\Pr(Y = k \mid \mathbf{x})$

Naiv Bayes vs. logistisk regresjon

```
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score

df_train = HDdata[mask]
df_test  = HDdata[~mask]
# Split into X (predictors) and y (response)
X_train = df_train[["sex","thalach","exang","oldpeak"]].values
y_train = df_train["y"].values
X_test  = df_test[["sex","thalach","exang","oldpeak"]].values
y_test  = df_test["y"].values
# Fit Gaussian Naive Bayes on training data
gnb = GaussianNB()
gnb.fit(X_train, y_train)
# Predict on test data
y_pred = gnb.predict(X_test)
# Accuracy
acc_perc = 100*accuracy_score(y_test, y_pred)
print(f"Naive Bayes accuracy: {acc_perc:.3f}")
```

```
Naive Bayes accuracy: 77.707
```

Oppsummering

- Logistisk regresjon:
  - Modell for $\Pr(Y = 1 \mid \mathbf{x})$ med logistisk lenkefunksjon
  - Lineær i log-odds
- Estimering:
  - Maksimum likelihood, numerisk optimalisering
- Tolkning:
  - Koeffisienter som log-odds
  - $\exp(\beta_j)$ som odds ratio
- Evaluering:
  - Nøyaktighet
  - Train/test-splitt og K-fold-kryssvalidering
- Alternativ:
  - Naiv Bayes som enkel, probabilistisk klassifikator