

---

# LÄR DIG AI FRÅN GRUNDEN - TILLÄMPAD MASKININLÄRNING MED PYTHON

---

Antonio Prgomet  
Terese Johnson  
Amanda Solberg  
Linus Rundberg Streuli

Detta verk är skyddat av upphovsrättslagen.

Den som bryter mot upphovsrättslagen kan åtalas av allmän åklagare och dömas till böter eller fängelse i upp till två år samt bli skyldig att erlägga ersättning till upphovsman eller rättsinnehavare.

© Pedagogicus Publishing

# Innehållsförteckning

<b>Förord</b>	<b>7</b>
Bokens hemsida . . . . .	7
Bokens målgrupp . . . . .	7
Element i boken . . . . .	7
Språk . . . . .	8
Bokens uppställning . . . . .	8
Lycka till . . . . .	9
<b>I Introduktion till maskininlärning</b>	<b>11</b>
<b>1 Introduktion till maskininlärning</b>	<b>13</b>
1.1 Artificiell intelligens (AI), maskininlärning (ML) och djupinlärning (DL)	14
1.2 Fyra problemkategorier inom maskininlärning: regression, klassificering, dimensionsreducering och klustring	16
1.3 Fundamentala koncept . . . . .	19
1.3.1 Träningsdata, valideringsdata och testdata . . . . .	21
1.3.2 K-delad korsvalidering . . . . .	30
1.3.3 <i>RMSE</i> . . . . .	33
1.3.4 Hyperparametrar och parametrar . . . . .	35
1.3.5 <i>Grid search</i> . . . . .	36
1.3.6 Kategorisk data . . . . .	42
1.3.7 <i>Feature engineering</i> . . . . .	47
1.4 Tvärsnittsdata, tidsseriedata och paneldata . . . . .	51
1.5 Övningsuppgifter . . . . .	54

<b>2 Ett ML-projekt från början till slut</b>	<b>55</b>
2.1 En checklista för Maskininlärning . . . . .	55
2.1.1 Definiera problemet och skapa en helhetsbild . . . . .	56
2.1.2 Få tillgång till datan . . . . .	56
2.1.3 Utforska datan, gör en <i>exploratory data analysis (EDA)</i> . . . . .	57
2.1.4 Bearbeta datan . . . . .	58
2.1.5 ML-modellering . . . . .	59
2.1.6 Presentera din lösning för intressenter . . . . .	60
2.1.7 Produktionssättning av modellen och övervakning av implementeringen . . . . .	60
2.2 Ett kodexempel från början till slut - Huspriser i Kalifornien . . . . .	63
2.3 Utmaningar inom ML . . . . .	90
2.4 <i>Scikit-learn</i> . . . . .	90
2.4.1 Designprinciper . . . . .	92
2.4.2 <i>Estimators, predictors, transformers</i> . . . . .	95
2.4.3 <i>Pipelines</i> . . . . .	97
2.4.4 Ett avstick till <i>TensorFlow</i> och <i>Keras</i> . . . . .	102
2.5 Övningsuppgifter . . . . .	102
<b>II Maskininlärning</b>	<b>103</b>
<b>3 Regression</b>	<b>105</b>
3.1 Regressionsproblem . . . . .	105
3.2 Utvärderingsmått . . . . .	106
3.2.1 <i>Root mean squared error (RMSE)</i> . . . . .	106
3.2.2 <i>Mean squared error (MSE)</i> . . . . .	107
3.2.3 <i>Mean absolute error (MAE)</i> . . . . .	107
3.3 Regressionsmodeller . . . . .	109
3.3.1 Linjär regression . . . . .	111
3.3.2 Optimeringsalgoritm - <i>Gradient descent</i> . . . . .	122
3.3.3 <i>The bias variance trade-off</i> . . . . .	128
3.3.4 <i>Ridge regression</i> (L2-regularisering) . . . . .	129
3.3.5 <i>Lasso regression</i> (L1-regularisering) . . . . .	133
3.3.6 <i>Elastic net</i> (kombination av <i>Lasso</i> & <i>Ridge</i> ) . . . . .	135
3.3.7 <i>Support vector machines (SVM)</i> . . . . .	136
3.3.8 Beslutsträd . . . . .	139
3.3.9 <i>Ensemble learning</i> . . . . .	144
3.3.10 <i>Random forest</i> . . . . .	149

3.4	Övningsuppgifter . . . . .	150
<b>4</b>	<b>Klassificering</b>	<b>151</b>
4.1	Klassificeringsproblem . . . . .	151
4.1.1	<i>OvR-</i> och <i>OvO</i> -algoritmerna . . . . .	154
4.2	Utvärderingsmått . . . . .	154
4.2.1	<i>Confusion matrix</i> . . . . .	155
4.2.2	<i>Accuracy</i> . . . . .	158
4.2.3	<i>Precision</i> och <i>Recall</i> . . . . .	159
4.2.4	<i>F1-score</i> . . . . .	162
4.2.5	<i>ROC</i> -kurvan . . . . .	164
4.3	Klassificeringsmodeller . . . . .	166
4.3.1	Logistisk regression . . . . .	168
4.3.2	<i>Support vector machines (SVM)</i> . . . . .	176
4.3.3	Beslutsträd . . . . .	190
4.3.4	<i>Ensemble learning</i> . . . . .	195
4.3.5	<i>Random forest</i> . . . . .	203
4.4	Två kodexempel . . . . .	206
4.4.1	Kodexempel 1 - Klassificering av <i>MNIST</i> . . . . .	206
4.4.2	Kodexempel 2 - Välja önskad <i>precision</i> och <i>recall</i> . . . . .	213
4.5	Övningsuppgifter . . . . .	216
<b>5</b>	<b>Dimensionsreducering</b>	<b>217</b>
5.1	<i>Curse of dimensionality</i> . . . . .	217
5.2	Vad är dimensionsreducering? . . . . .	220
5.2.1	Effekter av dimensionsreducering . . . . .	220
5.3	Principalkomponentanalys ( <i>PCA</i> ) . . . . .	223
5.3.1	<i>Kernel PCA</i> . . . . .	229
5.4	Övningsuppgifter . . . . .	232
<b>6</b>	<b>Klustering</b>	<b>233</b>
6.1	Vad är klustering? . . . . .	233
6.1.1	Exempel på tillämpningsområden för klustering . . . . .	234
6.2	<i>K-Means</i> . . . . .	237
6.2.1	Hur genomförs klustering med <i>K-Means</i> ? . . . . .	240
6.2.2	Hur tränas modellen? . . . . .	243
6.2.3	Val av antalet kluster . . . . .	246
6.3	Två kodexempel . . . . .	252
6.3.1	Kodexempel 1 - Kundsegmentering . . . . .	252

6.3.2	Kodexempel 2 - Semi-vägledd inlärning . . . . .	256
6.4	Övningsuppgifter . . . . .	259
<b>III</b>	<b>Djupinlärning</b>	<b>261</b>
<b>7</b>	<b>Artificiella neurala nätverk (ANN)</b>	<b>263</b>
7.1	Bakgrund . . . . .	264
7.2	Modellarkitektur . . . . .	264
7.2.1	Hur fungerar ett neutralt nätverk . . . . .	265
7.2.2	Riktlinjer för modellarkitektur . . . . .	272
7.2.3	Intuition för antal layers och noder . . . . .	274
7.3	Regularisering . . . . .	275
7.4	Träning av neurala nätverk - <i>Backpropagation</i> . . . . .	280
7.5	Två kodexempel . . . . .	284
7.5.1	Kodexempel 1 - Bildklassificering av <i>Fashion MNIST</i> . . . . .	284
7.5.2	Kodexempel 2 - Optimering av hyperparametrar med <i>KerasTuner</i> . . . . .	292
7.6	Övningsuppgifter . . . . .	296
<b>8</b>	<b>Convolutional neural network (CNN)</b>	<b>297</b>
8.1	Bakgrund . . . . .	297
8.2	Hur fungerar <i>CNN</i> ? . . . . .	298
8.2.1	<i>Convolution layers</i> . . . . .	298
8.2.2	<i>Pooling layers</i> . . . . .	301
8.2.3	<i>Padding</i> . . . . .	301
8.3	Exempel på en modellarkitektur för <i>CNN</i> . . . . .	302
8.4	<i>Data augmentation</i> . . . . .	305
8.5	Tre kodexempel . . . . .	305
8.5.1	Kodexempel 1 - Bildklassificering av <i>CIFAR-100</i> . . . . .	306
8.5.2	Kodexempel 2 - Förtränaade modeller . . . . .	310
8.5.3	Kodexempel 3 - <i>Transfer learning</i> . . . . .	312
8.6	Övningsuppgifter . . . . .	316
<b>9</b>	<b>Recurrent neural network (RNN)</b>	<b>317</b>
9.1	Bakgrund . . . . .	317
9.2	Modellarkitekturen . . . . .	319
9.2.1	<i>Long short-term memory (LSTM)</i> . . . . .	320
9.2.2	<i>Gated recurrent unit (GRU)</i> . . . . .	320
9.3	<i>Natural language processing (NLP)</i> . . . . .	321

9.3.1	<i>Embeddings</i>	321
9.4	Kodexempel - Sentimentanalys	323
9.5	Övningsuppgifter	327
<b>10</b>	<b>Chattbottar</b>	<b>329</b>
10.1	ChatGPT och <i>prompt engineering</i>	329
10.2	Molnbaserade och lokala språkmodeller	330
10.2.1	Molnbaserade modeller	331
10.2.2	Lokala modeller	331
10.3	Bygga en chattbot	331
10.4	<i>Retrieval Augmented Generation (RAG)</i>	333
10.4.1	Läs in data	334
10.4.2	Chunking	335
10.4.3	<i>Embeddings</i>	336
10.4.4	Semantisk sökning	337
10.4.5	Generera svar	340
10.4.6	Evaluering	341
10.5	Vector store	343
10.6	Vidare arbete med chattbottar	346
10.7	Övningsuppgifter	346



# Förord

## Bokens hemsida

Boken har en tillhörande hemsida där material kopplat till boken finns uppladdat.  
[https://github.com/AntonioPrgomet/ai\\_tillaempad\\_ml](https://github.com/AntonioPrgomet/ai_tillaempad_ml)

## Bokens målgrupp

Boken är avsedd för kurser av olika slag och används bland annat inom yrkeshögskolan, andra läroinstitut, företag och organisationer. Boken går även utmärkt att använda för självstudier.

Boken förutsätter att läsaren har grundläggande kunskaper i pythonprogrammering. Om det saknas går boken fortfarande att läsa efter att man gjort grundkursen i Python som finns tillgänglig på bokens hemsida. På bokens hemsida finns det även ett dokument där summasymbolen ( $\Sigma$ ) samt grundläggande vektor- och matrisalgebra går igenom. Den läsare som saknar de kunskaperna kan läsa igenom det dokumentet.

Boken kan användas på olika sätt.

- En elementär kurs i AI/ML kan inkludera kapitel 1-4.
- En grundkurs i AI/ML kan inkludera kapitel 1-6.
- En grundkurs i AI/Djupinlärning kan inkludera kapitel 1-2 och kapitel 7-10.
- En heltäckande kurs inom ML kan inkludera hela boken, kapitel 1-10.

## Element i boken

Boken använder sig av två element som är värdiga att notera.

Det finns informationsrutor som ser ut enligt nedan.

**i** Så här ser en informationsruta ut. I de här rutorna skriver vi korta fördjupande förklaringar, reflektioner och kommentarer.

Ofta är bokens kodexempel annoterade. På den högra kanten förekommer siffror som motsvarar en förklarande text under kodexemplet. Efter den förklarande texten följer utskrifter från själva koden. Vi ser ett exempel här nedanför.

```
print('An annotated code example.')
```

(1)

① Siffran (1) till höger i kodexemplet motsvaras av en förklarande text direkt under kodexemplet.

An annotated code example.

## Språk

Boken är skriven på svenska. Bokens kodexempel är dock på engelska eftersom det är en stark praxis världen över att kod skrivs på engelska. Vi har använt svensk terminologi i största möjliga utsträckning men vissa ord, exempelvis *pipeline* brukar benämnas så även i svenska och därför har vi behållit det. Engelska ord *kursiveras* i texten.

## Bokens upplägg

Boken är uppdelad i tre delar.

1. Introduktion till maskininlärning. Här introduceras maskininlärning där vi får en helhetsbild av ämnet och lär oss fundamentala koncept som används genom hela boken.
2. Maskininlärning. Här lär vi oss grunderna inom maskininlärning som består av de fyra områdena regression, klassificering, dimensionsreducering och klustering. Vi använder primärt biblioteket *scikit-learn* för modellering.
3. Djupinlärning. Här lär vi oss grunderna inom djupinlärning där modeller som kallas för neurala nätverk används. Djupinlärning är ett delämne inom maskininlärning. Här använder vi primärt biblioteken *TensorFlow* och *Keras* för modellering.

## **Lycka till**

AI och maskininlärning är högaktuella ämnen och kunskaper inom dem kan skapa många möjligheter. Vi författare hoppas att du ska tycka att boken är lika rolig att läsa som vi har tyckt det är att skriva den.

Lycka till!

Antonio Prgomet - <https://www.linkedin.com/in/antonioprgomet/>

Terese Johnson

Amanda Solberg

Linus Rundberg Streuli



## **Del I**

# **Introduktion till maskininlärning**



# Kapitel 1

## Introduktion till maskininlärning

I detta kapitel kommer vi att få en första helhetsbild över maskininlärning. Vi kommer börja med att lära oss hur artificiell intelligens (AI), maskininlärning (ML) och djupinlärning (DL) hänger ihop. Därefter fortsätter vi att kolla på fyra centrala problemkategorier inom ML, fundamentala koncept som vi kommer ha nyttå av genom hela boken och avslutningsvis kollar vi på hur data kan kategoriseras som tvärsnittsdata, tidsseriedata och paneldata.

I nästa kapitel kommer vi gå igenom ett ML-projekt från början till slut vilket ytterligare konkretisrar den helhetsbild vi får i detta kapitel. Dessa två kapitel har alltså en nära koppling till varandra. Under den första läsningen av dessa två kapitel bör läsaren inte försöka förstå alla detaljer eftersom många av koncepten kommer användas senare in i boken och då bli mer naturliga. Det är därför en god idé att göra en första genomläsning och försöka få en helhetsbild, när man sedan kommit längre in i boken kan man återkomma för att repetera och testa sin förståelse.

## 1.1 Artificiell intelligens (AI), maskininlärning (ML) och djupinlärning (DL)

AI är ett relativt nytt ämne och begreppet *artificiell intelligens* myntades först 1956. Men vad handlar AI om? AI handlar om förmågan hos datorprogram och robotar att efterlikna människors eller djurs intelligens. Några exempel på tillämpningsområden, som vi kommer lära oss i denna bok är:

- Utföra prediktioner. Exempelvis prediktera en persons inkomst baserat på dennes ålder och utbildningsnivå.
- Datorseende. Exempelvis kunna se vad det är för siffra någon skrivit på ett papper.
- Kommunicera. Exempel på detta är chatbottar såsom ChatGPT.

AI som ämne har flera delämnen där ML utgör en central del. Det finns flera definitioner och nedan ser vi två vanligt förekommande:

*“Machine Learning is the field of study that gives computers the ability to learn without being explicitly programmed”* (Arthur Samuel, 1959).

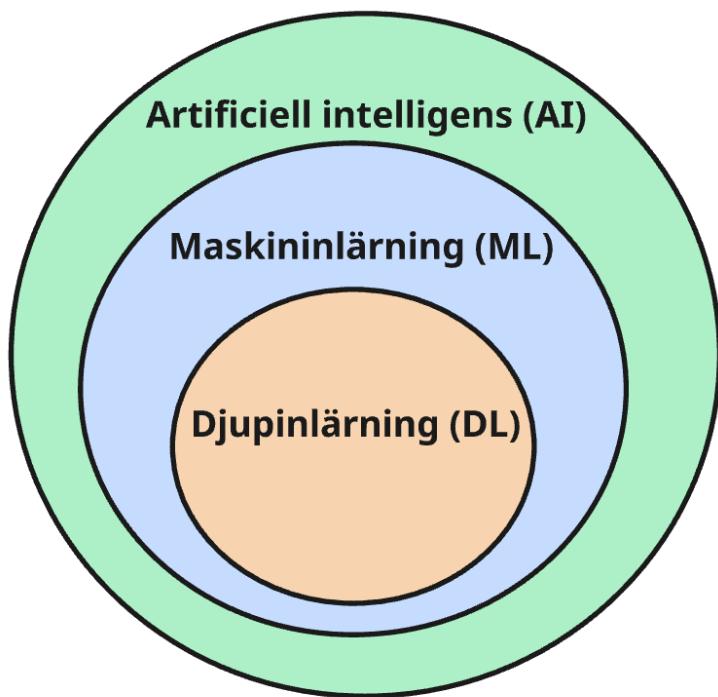
*“The science and art of programming computers so they can learn from data”*  
(Tom Mitchell, 1997).

Den första definitionen av Samuel betonar datorers förmåga *att lära sig* utan att vara explicit programmerade. Ett exempel är att vi baserat på data om människors löner och åldrar kan få datorn att lära sig att förstå vilket samband som råder mellan lön och ålder. Därefter kan vi, när vi träffar en ny människa, prediktera dennes lön baserat på dennes ålder. Denna förmåga, att datorn lärt sig ett samband mellan lön och ålder kan sättas i kontrast till traditionell programmering där vi explicit instruerar datorn vad den ska göra i olika situationer. Exempelvis genom konditionell logik såsom if-satser.

Den andra definitionen betonar att det är såväl en vetenskap som konst. Det innebär att när vi till exempel ska skapa en ML-modell så finns det rätta svar på frågor som ”hur funkar en modell rent matematiskt” eller ”hur funkar modellens träningsalgoritm”. Detta är vetenskapsdelen. Däremot finns det inte universella rätta svar på frågor såsom ”är modellen lämplig att använda” eller ”hur ska vi hantera diverse utmaningar som uppstår i modelleringen?” Detta är konstdelen. Notera, även om det inte finns universellt rätta svar så finns det svar som är mer eller mindre väl-underbyggda.

Inom ML finns det i sin tur ett delämne som kallas för djupinlärning (förkortas ofta DL efter engelskans *deep learning*) där modeller som kallas för neurala nätverk används. Neurala nätverk har en modellarkitektur som ursprungligen var inspirerade av hur bland annat människohjärnan fungerar. Vi kommer lära oss mer om neurala nätverk i bokens

tredje del som startar med Kapitel 7. Förhållandet mellan AI, ML och DL är som ses i Figur 1.1.



Figur 1.1: En schematisk bild över förhållandet mellan AI, ML och DL.

Inom ML kan vi definiera fyra olika typer av problemkategorier, dessa är *regression*, *klassificering*, *dimensionsreducering* och *klustring*. I Kapitel 3 - Kapitel 6 kommer respektive kategori gås igenom på djupet. Härnäst kollar vi på vad dessa fyra problemkategorier innebär.

## 1.2 Fyra problemkategorier inom maskininlärning: regression, klassificering, dimensionsreducering och klustering

För att träna/skapa maskininlärningsmodeller behöver vi ha tillgång till data. Många gånger har företag och/eller mäniskor tränat modeller som sedan andra kan använda utan att ha direkt tillgång till data. Detta är exempelvis fallet med publik tillgängliga chatbottar såsom ChatGPT. Men även den modellen har av upphovs-företaget, "OpenAI" i detta fallet, ursprungligen blivit tränad på data.

Data kan generellt sett delas in i *indata* och *utdata*. Indata är den information som ML-modellen får tillgång till och utdata är det svar som ML-modellen förväntas ge efter att ha bearbetat indatan. Generellt sett betecknas utdata med  $y$  och indatan med  $x$ .

Inom *regression* antar utdata,  $y$ , kontinuerliga värden. Exempel på kontinuerliga värden är inkomst då den kan anta alla värden inom ett spann, exempelvis 47 831 kr eller 32 421.5 kr.

Inom *klassificering* antar utdata,  $y$ , diskreta värden. Om det är två möjliga värden kallas det för *binär klassificering* och om det är fler möjliga värden kallas det för *multiklass klassificering*. Binär klassificering hade kunnat vara om vi vill prediktera om någon är en man eller en kvinna. I detta fallet är klasserna {man, kvinna}. Multiklass klassificering hade kunnat vara om vi vill prediktera om ett fordon är en cykel, bil, buss eller övrigt. I detta fall finns det alltså fyra klasser, dessa är {cykel, bil, buss, övrigt}.

Det vi kallat för utdata,  $y$ , kallas också för den *beroende variabeln*. De engelska begreppen *target* och *label* är också vanligt förekommande. Det vi kallat för indata,  $x$ , kallas också för den *oberoende variabeln*. Det engelska begreppet *feature* är också vanligt förekommande.

Kollar vi på datan nedan och exempelvis hade velat skapa en modell där en persons inkomst predikteras baserat på dennes ålder så är det ett regressionsproblem eftersom utdata (inkomst) är en kontinuerlig variabel.

```
regression_data = {
    "Inkomst (y)": [58500, 42000, 34000, 39000],
    "Ålder (x)": [58, 29, 35, 42]
}
```

①

```
df_reg = pd.DataFrame(regression_data)          ②
df_reg
```

- ① Rådata skapas som kommer konverteras till en *Pandas dataframe* i nästa steg.  
 ② Vi konverterar rådatan till en *Pandas dataframe*.

	Inkomst (y)	Ålder (x)
0	58500	58
1	42000	29
2	34000	35
3	39000	42

### i Vad är indata ( $x$ ) respektive utdata ( $y$ )?

Generellt sett är det inte givet vad som är indata ( $x$ ) respektive utdata ( $y$ ), det är något vi som modellerare behöver ta ställning till där vi betraktar  $y$  som beroende av  $x$  och inte tvärtom. I exemplet ovan är det relativt naturligt att åldern påverkar inkomst och inte tvärtom. Vi blir inte yngre eller äldre om vi får en ökad inkomst, däremot påverkar åldern inkomsten. Många gånger förstår vi ganska snabbt vad som är utdata respektive indata men det kan också vara utmanande att bestämma vad vi ska betrakta som vad vilket är en del av konsten inom ML.

Önskar vi betona rad och/eller kolumn för datan så kan vi använda indexering. Datatan ovan hade då kunnat karakteriseras enligt Ekvation 1.1.

$$\begin{cases} y_1, & x_1 \\ y_2, & x_2 \\ y_3, & x_3 \\ y_4, & x_4 \end{cases} \quad (1.1)$$

Vi ser exempelvis att  $y_1 = 58500$  och  $x_1 = 58$ . Notera, i Python börjar indexering från 0 medan vi för ekvationerna har valt att använda indexering som börjar från 1.

En hel rad, t.ex.  $y = 42000$  och  $x = 29$ , kallas tillsammans för en observation eller en datapunkt. Att det kallas för en datapunkt är eftersom vi kan se det som en vektor, som i sin tur kan representeras som en punkt i ett flerdimensionellt rum.

Rent generellt, om vi antar att vi har  $n$  stycken rader och  $p$  stycken variabler kan vi karakterisera datan enligt Ekvation 1.2.

$$\begin{pmatrix} y_1, & x_{11}, & x_{12}, & \dots, & x_{1p} \\ y_2, & x_{21}, & x_{22}, & \dots, & x_{2p} \\ \vdots \\ y_n, & x_{n1}, & x_{n2}, & \dots, & x_{np} \end{pmatrix} \quad (1.2)$$

Kollar vi på datan nedan och exempelvis hade velat skapa en modell där en persons kön predikteras baserat på inkomsten så hade det varit ett klassificeringsproblem eftersom utdata (kön) antar diskreta värden. Notera att i datan nedan så hade vi kunnat använda såväl inkomst som ålder för att prediktera kön. Generellt sett så är frågan om vilka variabler som ska inkluderas i en modell såväl en vetenskap som en konst. Det finns olika tillvägagångssätt och det är ett viktigt delområde inom ML som kallas för *variabelselektion*. Vi kommer lära oss mer om det i Avsnitt 1.3.7.

```
classification_data = {
    "Kön (y)": ["Man", "Kvinna", "Man", "Kvinna"],
    "Inkomst (x1)": [84000, 36750, 33000, 53000],
    "Ålder (x2)": [58, 29, 35, 42]
}

df_classification = pd.DataFrame(classification_data)
df_classification
```

	Kön (y)	Inkomst (x1)	Ålder (x2)
0	Man	84000	58
1	Kvinna	36750	29
2	Man	33000	35
3	Kvinna	53000	42

**i** Notera, i datan ovan har vi de två kategorierna “Man” och “Kvinna”. En annan kategorisering hade kunnat vara “Man”, “Kvinna” samt “Vill ej uppge”. Då har vi gått från ett binärt klassificeringsproblem till ett multiklass klassificeringsproblem. Hur gör vi i praktiken? Det är något den som skapar modellerna behöver ta ställning till, bland annat med avseende på vad som vill uppnås.

Inom *dimensionsreducering* förenklar vi indata genom att överföra den till en rymd med lägre dimensionalitet, exempelvis genom en modell som kallas för *principalkomponentanalys*, ofta benämnt PCA efter engelskans *Principal Component Analysis*. En vanlig anledning till att minska datans dimensionalitet är för att modellträningen (där vi använder regressionsmodeller eller klassificeringsmodeller) ska gå snabbare.

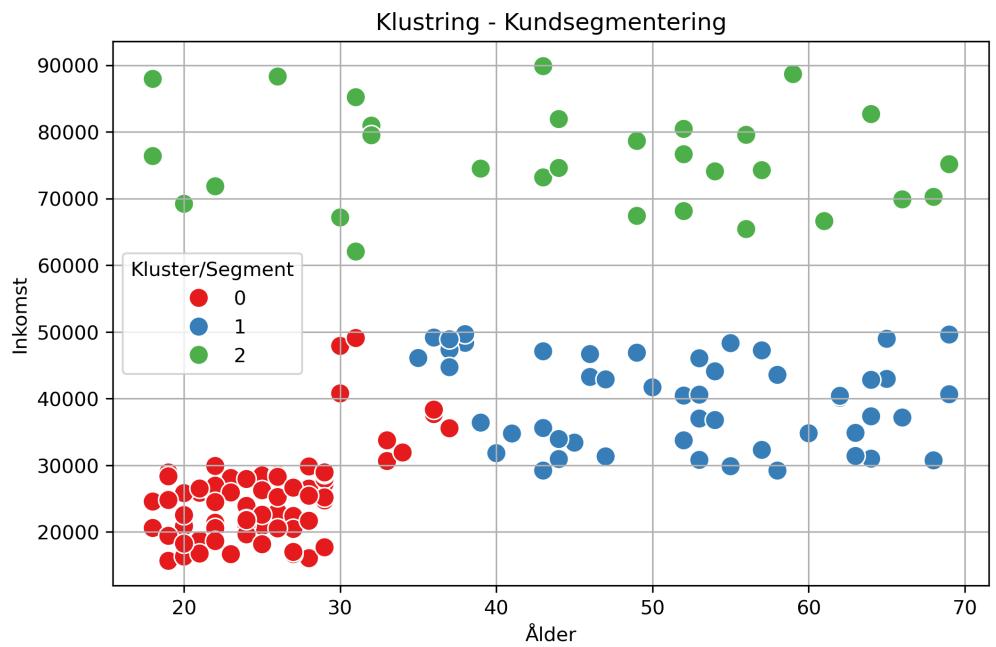
Inom *klustering* har vi endast indata och vill dela in denna i olika grupper/kluster. Ett tillämpningsområde inom detta är kundsegmenteringsmodeller. Exempelvis i Figur 1.2 kan vi tänka oss att kunder från de olika klustren/segmenten kan ha olika preferenser och köpkraft. Vilka preferenser de har kan exempelvis undersökas via intervjuer och/eller via dataanalys om data finns tillgänglig. Detta kan företag använda för att bland annat anpassa produkterbjudanden och marknadsföring. Om vi tänker oss att ett bilföretag producerar både familjebilar och sportbilar så kanske sportbilarna hade marknadsförts mot det andra segmentet eftersom de har högst inkomst. Vi kanske också kommer fram till att många inom det första segmentet hade önskat ha en sportbil, då kanske det kan vara en bra affärsmöjlighet för företaget att försöka utveckla en billigare form av sportbil?

Vi avslutar med att nämna att ett gemensamt namn för regressionsproblem och klassificeringsproblem är *väglett lärande* (eller övervakat lärande) och ett gemensamt namn för klustering och dimensionsreducering är *icke-väglett lärande* (eller oövervakat lärande). Anledningen till att det kallas för väglett lärande är på grund av att vi kan tänka oss att  $y$  vägleder  $x$  när modellerna tränas. För icke-väglett lärande använder vi endast indata,  $x$ , innebärande att inget  $y$  vägleder träningen.

## 1.3 Fundamentalta koncept

Inom maskininlärning finns det ett antal olika koncept som är fundamentala och väldigt ofta används. Nedan listar vi dessa.

- Uppdelning av data i träningsdata, valideringsdata och testdata. En variant av detta är k-delad korsvalidering som vi också kommer lära oss.



Figur 1.2: Exempel på en kundsegmenteringsmodell.

- Utvärderingsmått, för regressionsproblem är det vanligast förekommande det som kallas för *Root Mean Squared Error (RMSE)*. För klassificeringsproblem finns det andra utvärderingsmått såsom *accuracy*, *precision*, *recall* och *confusion matrix*. Här kommer vi introducera *RMSE* och andra utvärderingsmått presenteras senare i boken.
- Hyperparametrar och parametrar. Hyperparametrar styr hur en modell lär sig och svarar på frågan ”hur vi lär oss”. Parametrar är saker som en modell har lärt sig och svarar på frågan ”vad har vi lärt oss”.
- *Grid search* används för att optimera hyperparametrar.
- Hantering av kategorisk data i modeller, det vill säga data som är icke-numerisk. Om vi antar att en variabel kan anta färgerna {blå, grön, röd} så är det exempel på kategorisk data.
- *Feature engineering* handlar om vilka variabler som ska väljas i en modell, skapandet av nya variabler och transformering av variabler. Generellt sett åsyftas de oberoende variablerna (*features*) men även exempelvis transformering av de beroende variablerna kan göras.

Vi går nu igenom varje koncept i mer detalj.

### 1.3.1 Träningsdata, valideringsdata och testdata

När vi tränar ML-modeller så behöver vi data som modellerna kan tränas på. Mer specifikt brukar datan, det vill säga hela datasetet, delas in i träningsdata, valideringsdata och testdata.

- På träningsdatan tränar vi olika modeller.
- På valideringsdatan utvärderar vi de olika modellerna som tränats på träningsdatan för att se vilken modell som presterar bäst. Hur vet vi vilken modell som presterar bäst? För att veta det använder vi ett eller flera utvärderingsmått. För regressionsproblem används ofta *RMSE* som vi snart kommer lära oss om i Avsnitt 1.3.3.
- När vi valt den bäst presterande modellen från valideringsdatan ska vi, givet att vi är nöjda med resultatet, träna om den modellen på träningsdatan och valideringsdatan ihopslagen innan modellen slutligen testas på testdatan för att få en uppskattning på modellens generaliseringsförmåga på ny osedd data. Om vi inte är nöjda med resultatet får vi exempelvis prova att tränna andra modeller på träningsdatan, bearbeta datan på olika sätt eller samla in ny/mer data.

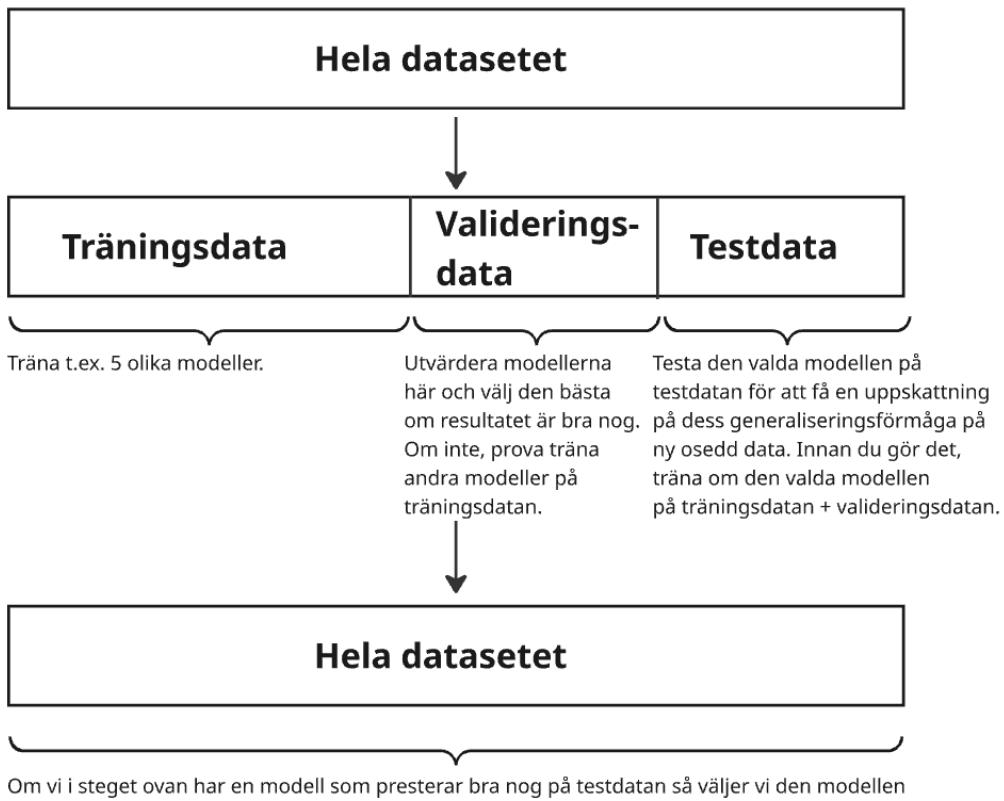
- Om modellen som testats på testdatan bedöms vara tillräckligt bra för att börja användas/produktionssättas så ska modellen tränas om på hela datasetet, det vill säga träningsdelen, valideringsdelen och testdelen ihopslagen. Det är också anledningen till att vi ska träna om modellen på träningsdelen och valideringsdelen i steget ovan innan vi testar den på testdelen. Detta eftersom det är närmre mängden data som modellen slutligen kommer tränas på innan produktionssättning och därmed får vi generellt sett också en mer precis bild av hur bra modellen kan tänkas prestera i skarp läge när den är i produktion.

**i** I praktiken ser vi ibland att en modell inte tränas om på träningsdelen och valideringsdelen ihopslagen innan den testas på testdelen eller att modellen inte tränas om på hela datasetet innan den börjar användas skarpt. Anledningen är att det kan ta lång tid att träna om modellen och den som genomför modelleringen tycker helt enkelt inte att det är värt det. Så länge man förstår vad man gör så kan man avvika från det som är teoretiskt korrekt.

En tumregel är att data används i proportionerna 60-20-20 eller 70-15-15, det vill säga 60% (70%) av datan används för träning, 20% (15%) av datan används för validering och 20% (15%) av datan används för test. Denna uppdelning är endast en tumregel. Om vi exempelvis har för lite träningsdata kan träningen av modellerna bli dålig och har vi för lite valideringsdata kan valet av den bäst presterande modellen bli dålig. Det finns alltså en *trade-off* men denna *trade-off* beror på den *absoluta* mängden data snarare än procentsatser. Det innebär att om vi har mycket data (vad som är mycket data beror på tillämpningsområde och modell) kan det sakna praktisk betydelse om vi använder en 60-20-20 uppdelning, 50-25-25 uppdelning eller ens en 98-1-1 uppdelning. Om modellerna tar lång tid att träna kan vi föredra en uppdelning där vi har mindre data i träningsdelen på grund av att modellerna då tränas på mindre data vilket går snabbare. Om vi har lite data kanske vi väljer att använda en uppdelning där vi har mer data i träningsdelen för att modellerna ska ha en rimlig chans att bli tränade på ett tillräckligt stort dataset.

Se Figur 1.3 för en schematisk illustration av hur vi använder träningsdata, valideringsdata och testdata.

Varför exakt behöver vi använda valideringsdata? Anledningen är att vi på träningsdelen kan skapa perfekta modeller som kan genomföra alla prediktioner korrekt. Detta är dock till ingen nytta om modellen inte kan prediktera ny, osedd data korrekt. Därför utvärderar vi modellerna som tränats på träningsdelen via valideringsdelen. Modeller som är bra på träningsdelen men dåliga på valideringsdelen sägs vara överanpassade



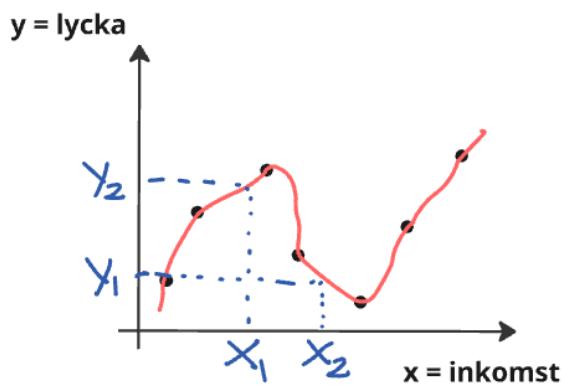
Figur 1.3: En schematisk illustration av hur träningsdata, valideringsdata och testdata används.

(*overfitted* på engelska). Modeller som är för enkla för att kunna fånga strukturen i träningsdatan sägs vara underanpassade (*underfitted* på engelska). Generellt sett presterar underanpassade modeller dåligt på både träningsdatan och valideringsdatan. Se Figur 1.4 för ett schematiskt exempel på hur en överanpassad modell, underanpassad modell och en rimlig modell hade kunnat se ut. I figuren antar vi att den beroende variabeln,  $y$ , mäter en persons lycka och den oberoende variabeln,  $x$ , mäter en persons inkomst. Generellt sett är det rimligt att tänka sig att en ökad inkomst ger en ökad lycka. Kollar vi på det översta exemplet “a) överanpassad modell” så ser vi att modellen, som representeras av den röda linjen följer träningsdatan, som representeras av de svarta punkterna, exakt. Det leder till orimliga resultat. Anledningen är att om två personer har de fixa inkomstnivåerna  $x_1$  och  $x_2$  där  $x_1 < x_2$  i figuren så blir den predikterade lyckan  $y_2 > y_1$ . I ord, personen med lägre inkomst förväntas ha en högre lycka i just det intervallet vi kollar på. Det är ett orimligt mönster. Modellen är alltså perfekt för träningsdatan men leder till orimliga resultat när vi genomför nya prediktioner. Den är överanpassad. Kollar vi på det mittersta exemplet “b) underanpassad modell” så ser vi en underanpassad modell som inte lyckas fånga datans underliggande struktur. Oavsett inkomstnivå så är den predikterade lyckan densamma. I det nedersta exemplet, “c) rimlig modell” så ser vi en modell som verkar följa den generella trenden i träningsdatan utan att följa den för noga. Vi ser också att om två personer med de fixa inkomsterna  $x_3$  och  $x_4$  där  $x_3 < x_4$  så förväntas personen med den högre inkomsten ha en högre lycka  $y_4$  än personen med lägre inkomst som förväntas ha lyckan  $y_3$ . Det är rimligt ur ett modelleringsperspektiv även om det i verkligheten kan finnas många andra faktorer som påverkar lyckan. Detta är alltså ett förenklat exempel.

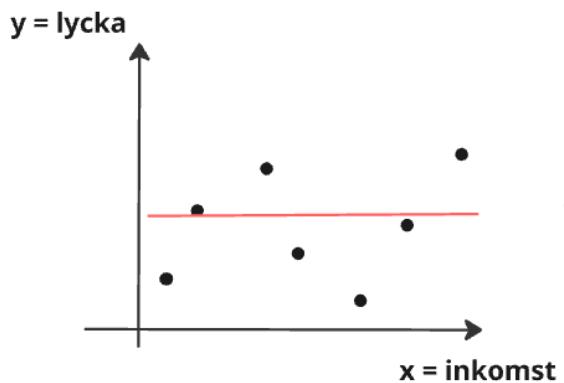
### **i En modell är en förenkling av verkligheten**

I exemplet ovan skapade vi en modell där lycka modellerades med avseende på inkomst. Det är en förenkling av verkligheten eftersom vi vet att många andra faktorer såsom relationer och hälsa också påverkar lycka. Faktum är att alla modeller är en förenkling av verkligheten, de varken är eller bör betraktas som “sanna”. Målet är alltså inte att skapa en modell som ger en “exakt bild av verkligheten” utan snarare en modell som hjälper oss att uppnå det syfte vi har. Därför kan en viss modell vara användbar i ett sammanhang men inte i ett annat.

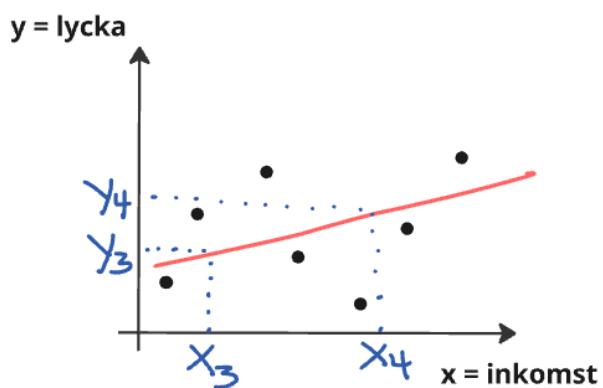
Varför testar vi en modell på testdatan? Räcker det inte att modellen utvärderas på valideringsdatan? Det faktum att man inom ML ofta skapar flera olika modeller och väljer den modell som presterar bäst på valideringsdatan gör att det av ren slump kan hända att modellen råkar prestera bra på valideringsdatan. För att minska denna risk



**a) Överanpassad modell**



**b) Underanpassad modell**



**c) Rimlig modell**

Figur 1.4: En schematisk figur som visar hur en överanpassad modell, underanpassad modell och en rimlig modell hade kunnat se ut.

så genomförs alltså ett test på testdatan. Notera att testdatan tar man ut i början av sitt arbetsflöde och därefter ska man inte använda det förrän man kommit till slutet av arbetsflödet och det är dags att slutgiltigt testa den valda modellen. Anledningen är att om vi testar olika modeller på testdatan och fortsätter att göra så tills vi hittar något som funkar bra så finns det en risk att vi till slut endast hittat en modell som presterar bra på den specifika testdatan som vi har. Det är meningslöst eftersom syftet med testdatan är att testa en modells generaliséringsförmåga på ny, osedd data.

Notera att idealt sett så önskar vi att datan som används är *representativ* för det vi vill modellera. Om vi exempelvis vill skapa en prediktionsmodell för att prediktera inkomster för personer med olika utbildningsnivå i Sverige så vore det inte bra om vi endast har data för personer som bor i Stockholm. Anledningen kan exempelvis vara att utbildningsnivå har olika stor påverkan på inkomst i olika delar av landet.

Har vi ett representativt urval av data i grunden så gör *scikit-learn* per automatik slumpmässiga urval när data delas in i träningsdata, valideringsdata ochtestdata (i kodexemplet nedan kommer vi se `train_test_split`). Det slumpmässiga urvalet leder oftast till att även träningsdatan, valideringsdatan och testdatan blir representativ. Vi nämner för den intresserade läsaren att det är möjligt att genomföra *stratifierade urval* när data delas in i träningsdata, valideringsdata ochtestdata. Detta är inget vi går djupare in på i denna bok.

Vi kollar nu på ett kodexempel för att konkretisera det vi gått igenom. Vi använder demonstrationsdatan `load_diabetes` från *scikit-learn* för detta ändamål.

```
from sklearn.datasets import load_diabetes  
from sklearn.model_selection import train_test_split  
  
diabetes = load_diabetes(as_frame=True)  
  
df = diabetes.frame  
print(df.head())  
  
X = df.drop(columns='target')  
y = df['target']  
  
X_train_full, X_test, y_train_full, y_test = train_test_split(X,  
                                                               y, test_size=0.2, random_state=42)
```

```
X_train, X_val, y_train, y_val = train_test_split(X_train_full,
    ↳ y_train_full, test_size=0.25, random_state=42)           ⑦
```

- ① Importerar de två biblioteken vi använder.
- ② Ladda in datan.
- ③ `.frame` specificerar att vi vill ha en *dataframe* som innehåller både de oberoende variablerna (*x*) och den beroende variabeln (*y*) som på engelska benämns *target*.
- ④ Skriver ut de fem första raderna så vi ser hur datan ser ut.
- ⑤ Vi delar in vår data i *X* och *y* för att därefter, i nästa steg, dela upp vår data i träningsdata, valideringsdata och testdata.
- ⑥ Vi delar in vår data, *X* och *y*, i *X\_train\_full*, *X\_test*, *y\_train\_full* och *y\_test*. `random_state=42` leder till att vi får reproducerbara resultat eftersom vilken data som hamnar i *train\_full* respektive *test* är slumpmässigt. Siffran 42 valdes godtyckligt.
- ⑦ *X\_train\_full* och *y\_train\_full* delas nu in i *X\_train*, *X\_val*, *y\_train* och *y\_val*.

	age	sex	bmi	bp	s1	s2	s3
0	0.038076	0.050680	0.061696	0.021872	-0.044223	-0.034821	-0.043401
1	-0.001882	-0.044642	-0.051474	-0.026328	-0.008449	-0.019163	0.074412
2	0.085299	0.050680	0.044451	-0.005670	-0.045599	-0.034194	-0.032356
3	-0.089063	-0.044642	-0.011595	-0.036656	0.012191	0.024991	-0.036038
4	0.005383	-0.044642	-0.036385	0.021872	0.003935	0.015596	0.008142

	s4	s5	s6	target
0	-0.002592	0.019907	-0.017646	151.0
1	-0.039493	-0.068332	-0.092204	75.0
2	-0.002592	0.002861	-0.025930	141.0
3	0.034309	0.022688	-0.009362	206.0
4	-0.002592	-0.031988	-0.046641	135.0

Sammanfattningsvis har vi med koden ovan skapat följande:

1. Vi har hela datasetet som representeras av *X* och *y*.
2. Vi har träningsdatan och valideringsdatan ihopslagen som representeras av *X\_train\_full* och *y\_train\_full*.
3. Vi har träningsdatan som representeras av *X\_train* och *y\_train*.
4. Vi har valideringsdatan som representeras av *X\_val* och *y\_val*.
5. Vi har testdatan som representeras av *X\_test* och *y\_test*.

Med datan på plats kan vi demonstrera hur processen ser ut vid modellval. Kom ihåg att processen var:

1. Träna olika modeller på träningsdatan.
2. Utvärdera modellerna på valideringsdatan och välj den bäst presterande.
3. Om prestationen är bra nog, träna om den bäst presterande modellen på träningsdatan och valideringsdatan ihopslagen innan den utvärderas på testdatan.
4. Träna om modellen på hela datasettet innan vi börjar använda den skarpt.

För detta ändamål skapar vi två modeller, en linjär regressionsmodell och ett beslutssträd, som vi kommer lära oss mer om senare i boken, Kapitel 3.

```
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeRegressor
from sklearn.metrics import root_mean_squared_error

# 1. Träna två modeller på träningsdatan.
linreg = LinearRegression()                                ①
tree = DecisionTreeRegressor(random_state=42)             ②

linreg.fit(X_train, y_train)
tree.fit(X_train, y_train)                                ③

# 2. Utvärdera modellerna på valideringsdatan och välj den bäst
#     ↵ presterande.
rmse_linreg = root_mean_squared_error(y_val,
    ↵ linreg.predict(X_val))
rmse_tree = root_mean_squared_error(y_val, tree.predict(X_val)) ④

print(f"RMSE på validering - Linjär Regression:
    ↵ {rmse_linreg:.2f}")
print(f"RMSE på validering - Beslutsträd: {rmse_tree:.2f}")

# Välj den bäst presterande modellen på valideringsdatan. Den
#     ↵ modellen som har lägre RMSE är bättre.
if rmse_linreg < rmse_tree:
    best_model = LinearRegression()
    best_model_name = "Linjär Regression"
else:
```

```

best_model = DecisionTreeRegressor(random_state=42)
best_model_name = "Beslutsträd"

# 3. Om prestationen är bra nog, träna om den bäst presterande
    ↵ modellen på träningsdatan och valideringsdatan ihopslagen
    ↵ innan den utvärderas på testdatan.
best_model.fit(X_train_full, y_train_full)

# Utvärdera modellen på testdatan.
rmse_test = root_mean_squared_error(y_test,
    ↵ best_model.predict(X_test))

print(f"Bästa modell baserat på validering: {best_model_name}")
print(f"RMSE på testdatan: {rmse_test:.2f}")

# 4. Träna om modellen på hela datasetet innan vi börjar använda
    ↵ den skarpt.
best_model.fit(X, y)

```

- ① Vi instantierar en linjär regressionsmodell.
- ② Vi instantierar ett beslutsträd.
- ③ Vi tränar respektive modell genom att använda `.fit()` metoden.
- ④ Vi beräknar *RMSE* för respektive modell. Vad det utvärderingsmåttet innehåller kommer vi gå igenom i Avsnitt 1.3.3.

RMSE på validering - Linjär Regression: 51.19

RMSE på validering - Beslutsträd: 65.81

Bästa modell baserat på validering: Linjär Regression

RMSE på testdatan: 53.85

---

fit_intercept	True
copy_X	True
tol	1e-06
n_jobs	None
positive	False

---

Att dela in datan i träningsdata, valideringsdata och testdata är fundamentalt och

kommer återkomma genom hela boken. En variant av detta är det som kallas för *k*-delad korsvalidering och det går vi igenom härnäst.

### 1.3.2 K-delad korsvalidering

Som ett alternativ till uppdelningen i träningsdata, valideringsdata och testdata så används ofta *k-delad korsvalidering*. Det är en metod där datan först delas in i träningsdata ochtestdata. En vanlig tumregel är en 80-20 uppdelning eller 70-30 uppdelning. Träningsdatan delas därefter upp i *k* lika stora delar där modellen tränas *k* gånger, varje gång på *k*-1 delar (dessa *k*-1 delarna är den nya träningsdatan för varje iteration) och den återstående delen är valideringsdatan för varje iteration. *K* är vanligtvis fem eller tio. Se Figur 1.5 för en visualisering av *k*-delad korsvalidering. För varje iteration fås ett värde på det utvärderingsmåttet vi använder, exempelvis *RMSE* om vi arbetar med regressionsproblem. Om vi använder 5-delad korsvalidering, som i figuren, kommer vi totalt få fem olika siffror som vi tar medelvärdet av vilket gör att vi får en slutgiltig siffra som används för att mäta hur bra vår modell presterar.

Logiken bakom att använda *k*-delad korsvalidering är att om valideringsdatan är för liten så kan det leda till ett dåligt modellval. Om valideringsdatan är för stor kan det leda till lite träningsdata vilket kan leda till att modellerna blir dåligt tränade. Men även i de fall valideringsdatan varken är för liten eller för stor så kombinerar *k*-delad korsvalidering flera estimat på utvärderingsmåttet som används (t.ex. *RMSE* i regressionsfallet) vilket kan ge en bättre skattning på hur bra modellen faktiskt presterar. Nackdelen är att varje modell behöver tränas *k* gånger, något som kan ta lång tid.

Vi demonstrerar *K*-delad korsvalidering med ett kodexempel.

```
import numpy as np
from sklearn.datasets import load_diabetes
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.svm import LinearSVR
from sklearn.model_selection import cross_validate

X, y = load_diabetes(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y,
    test_size=0.2, random_state=42)

lr = LinearRegression()
```

## Träningsdata

**Iteration 1.** Validering



**Iteration 2.** Validering



**Iteration 3.** Validering



**Iteration 4.**



**Iteration 5.**



Figur 1.5: Illustration över hur en 5-delad korsvalidering ser ut. I varje iteration så genomförs en validering på den data som representeras av rutan där det står "Validering". Daten som representeras med övriga rutor används för att träna modellen. För varje iteration får ett värde för det utvärderingsmått som används på valideringen, exempelvis RMSE för regressionsproblem. Totalt får fem värden som används för att beräkna ett medelvärde, medelvärdet blir den siffran som används för att utvärdera hur bra modellen presterat.

```

svm = LinearSVR()                                ①

cv_lr = cross_validate(lr, X, y, cv=5, scoring =
    ↵ 'neg_root_mean_squared_error')               ②
cv_svm = cross_validate(svm, X, y, cv=3, scoring =
    ↵ 'neg_root_mean_squared_error')

print(cv_lr.keys())                               ③

print(cv_lr['test_score'])                      ④
print(cv_svm['test_score'])                     ⑤

if np.mean(cv_lr['test_score']) > np.mean(cv_svm['test_score']):
    ↵
        print("The best model is linear regression.")
elif np.mean(cv_lr['test_score']) ==
    ↵ np.mean(cv_svm['test_score']):
        print("The models are equally good.")
else:
    print("The best model is support vector machine.") ⑥

```

- ① Vi instantierar den linjära regressionsmodellen och en modell som heter *Support Vector Machine (SVM)* som vi kommer lära oss om senare i boken, Kapitel 3.
- ② Vi genomför 5-delad korsvalidering. I kodraden nedan genomför vi 3-delad korsvalidering så läsaren ska kunna se skillnaden. Vi specificerar hyperparametern `scoring = 'neg_root_mean_squared_error'`, i Avsnitt 1.3.5 kommer det förklaras varför vi använder *negative root mean squared error*. I korthet så vill vi generellt sett ha så lågt fel som möjligt, det vill säga låg *RMSE*. I *scikit-learn* är dock "högre värden bättre" och för att rangordningen då ska bli korrekt använder vi negativt tecken.
- ③ Med denna koden ser vi att vi bland annat kan få ut `test_score` vilket är det vi är intresserade av i detta kodexempel.
- ④ Vi får fem estimerade fel eftersom vi använde 5-delad korsvalidering vilket specificeras med `cv=5` i koden ovan.
- ⑤ Vi får tre estimerade fel eftersom vi använde 3-delad korsvalidering vilket specificeras med `cv=3` i koden ovan.
- ⑥ Med korsvalidering fick vi flera estimat på felet som modellerna gör. För att jämföra dem beräknar vi medelvärdet av felet för respektive modell. Notera, vi vill

generellt sett ha låga fel men eftersom "högre värden är bättre" i *scikit-learn* så använde vi `neg_root_mean_squared_error` och då är alltså den modell med högre värde bättre.

```
dict_keys(['fit_time', 'score_time', 'test_score'])
[-52.72497937 -55.03486476 -56.90068179 -54.85204179 -53.94638716]
[-90.2051448 -93.11857727 -94.32004223]
The best model is linear regression.
```

### 1.3.3 RMSE

*Root Mean Squared Error (RMSE)* är ett utvärderingsmått för regressionsmodeller och dess matematiska uttryck ser vi i Ekvation 1.3.

$$\text{RMSE} = \sqrt{\frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2} \quad (1.3)$$

Notera att  $\hat{y}_i$  är predikterade värden för observation/datapunkt  $i$ , och  $y_i$  är de sanna värdena för observation/datapunkt  $i$ . Det är en väletablerad konvention att tillägget av en "hatt",  $\hat{y}$ , generellt sett betecknar predikterade värden.

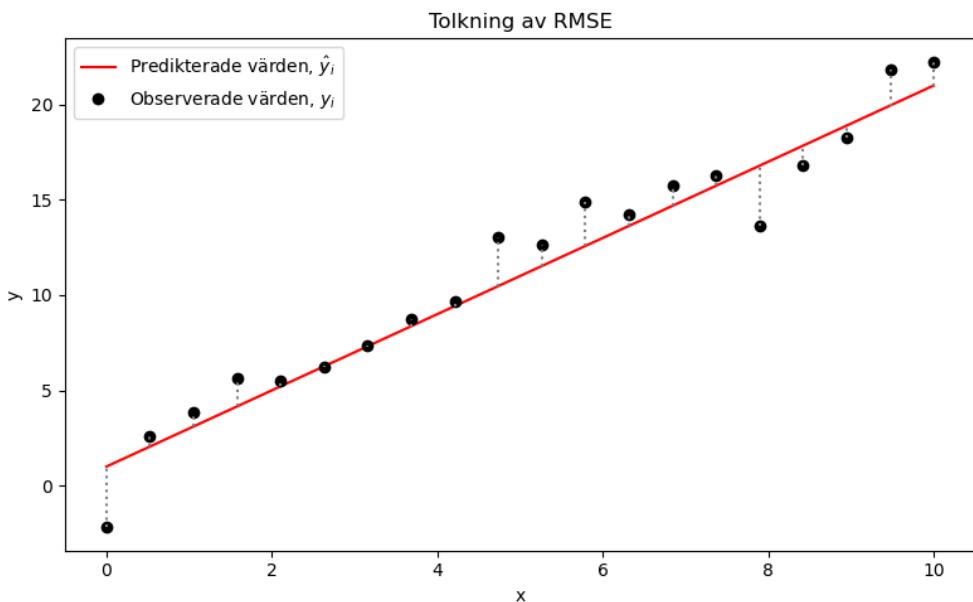
Vi går nu igenom formeln steg för steg.

- Vi beräknar skillnaden mellan det sanna värdet,  $y_i$  och det predikterade värdet  $\hat{y}_i$ ;  $(y_i - \hat{y}_i)$ . Detta kallas för *Error*.
- Eftersom vi beräknar skillnader kan dessa vara positiva och negativa vilket i sin tur kan leda till att skillnaderna tar ut varandra när de summeras. Antag att  $y_1 = 10, y_2 = 8$  och  $\hat{y}_1 = 12, \hat{y}_2 = 6$ . Då blir summan av skillnaderna  $(y_1 - \hat{y}_1) + (y_2 - \hat{y}_2) = (10 - 12) + (8 - 6) = (-2) + (2) = 0$ . Summan av skillnaderna blir alltså noll trots att båda prediktionerna är fel. Detta är inte en önskvärd egenskap och därför kvadrerar vi skillnaderna och tar alltså  $(y_i - \hat{y}_i)^2$ . Detta kallas för *Squared Error*.
- Vi beräknar medelvärdet för för de kvadrerade felet (*Squared Error*);  $\frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$ . Detta kallas för *Mean Squared Error*.
- När vi beräknade *Squared Error* så kvadrerade vi skillnaderna. Om vi exempelvis predikterar en persons lön i svenska kronor (kr) så blir enheten  $(\text{kr})^2$  vilket är svårtolkat. Därför använder vi kvadratroten för att återföra värdena till den

ursprungliga enheten och får formeln  $\sqrt{\frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2}$ . Detta kallas för *Root Mean Squared Error*.

Vi ser alltså att namnet *Root Mean Squared Error* är ett beskrivande namn på formeln.

Vi kan, lite slarvigt, tolka *RMSE* som de observerade värdenas ( $y$ ) medelavstånd från de predikterade värdena ( $\hat{y}$ ). Att det är en slarvig tolkning beror på att vi tar medelvärdet av *Squared Error* och därefter tar vi kvadratroten ur det. Se Figur 1.6.



Figur 1.6: En visualisering som hjälper oss att tolka RMSE.

Avslutningsvis, vill vi kunna tolka felet som modeller gör använder vi oss av *RMSE*. Vill vi rangordna modeller, det vill säga jämföra dem för att se vilken som är bättre kan vi också använda oss av *RMSE* men vi kan även använda *Mean Squared Error* (*MSE*). Anledningen är att det endast skiljer ett rotens-ur tecken och det är en monoton transformation innebärande att rangordningen förblir densamma. Exempelvis,  $100 > 9 \Leftrightarrow \sqrt{100} > \sqrt{9} \Leftrightarrow 10 > 3$ . Att man ofta använder *MSE* för modell-rangordning, som vi bland annat kommer se senare i boken, beror på att kvadratrotten är en extra operation som kan ta längre tid att genomföra.

Vi demonstrerar hur beräkningen av  $RMSE$  kan se ut i *scikit-learn*.

```
from sklearn.metrics import root_mean_squared_error  
y_true = [4, 10, 8] (1)  
y_pred = [2, 13, 5] (2)  
root_mean_squared_error(y_true, y_pred) (3)
```

- ① Vi antar att vi har några specificerade värden som är korrekta ( $y$ ).
- ② Vi antar att vi har några värden som motsvarar predikterade värden ( $\hat{y}$ ).
- ③ Vi beräknar  $RMSE$ .

2.70801280154532

### 1.3.4 Hyperparametrar och parametrar

En hyperparameter svarar på frågan “hur vi lär oss” och en parameter svarar på frågan “vad vi lärt oss”.

I koden nedan har vi skapat två olika modeller, `lin_reg_1` och `lin_reg_2`. Notera att båda modellerna är linjära regressionsmodeller vars modellspecifikation är den räta linjens ekvation i detta fallet, det vill säga  $y = kx + m$  där  $k$  är lutningen och  $m$  är interceptet eller skärningspunkten med y-axeln. Skillnaden är att i den första modellen är hyperparametern `set_intercept` satt till `False` medan i den andra modellen är hyperparametern `set_intercept` satt till `True`. Detta styr alltså hur vi lär oss och i den första modellen specificerar vi att vi inte vill att modellen ska skatta ett intercept utifrån datan och istället anta att det är 0 medan i den andra modellen vill vi att modellen ska skatta ett intercept utifrån datan. Därmed kommer interceptet, som är en parameter, att vara 0 för den första modellen och något annat för den andra modellen (teoretiskt sett kan interceptet bli 0 även för den andra modellen ifall det är vad modellen lär sig).

```
from sklearn.linear_model import LinearRegression (1)  
  
# Data  
example_data = {  
    "y": [-10, 3, 2, 7, -9, 10, 9, 7, 4, 7, 3, 2, -4, -5, -9, -4],  
    "x": [-3, 0, 1, 2, 4, 2, 3, 5, 7, 8, 9, 6, -2, -5, -1, -2]  
}
```

```

example_data = pd.DataFrame(example_data)                                ②

lin_reg_1 = LinearRegression(fit_intercept=False)                         ③
lin_reg_1.fit(example_data[["x"]], example_data["y"])                      ④

lin_reg_2 = LinearRegression(fit_intercept=True)                           ⑤
lin_reg_2.fit(example_data[["x"]], example_data["y"])

print('Parameters for lin_reg_1.', 'Intercept:',
      ↵ round(lin_reg_1.intercept_, 2), 'Slope',
      ↵ round(lin_reg_1.coef_[0], 2))                                         ⑥
print('Parameters for lin_reg_2.', 'Intercept:',
      ↵ round(lin_reg_2.intercept_, 2), 'Slope',
      ↵ round(lin_reg_2.coef_[0], 2))                                         ⑦

```

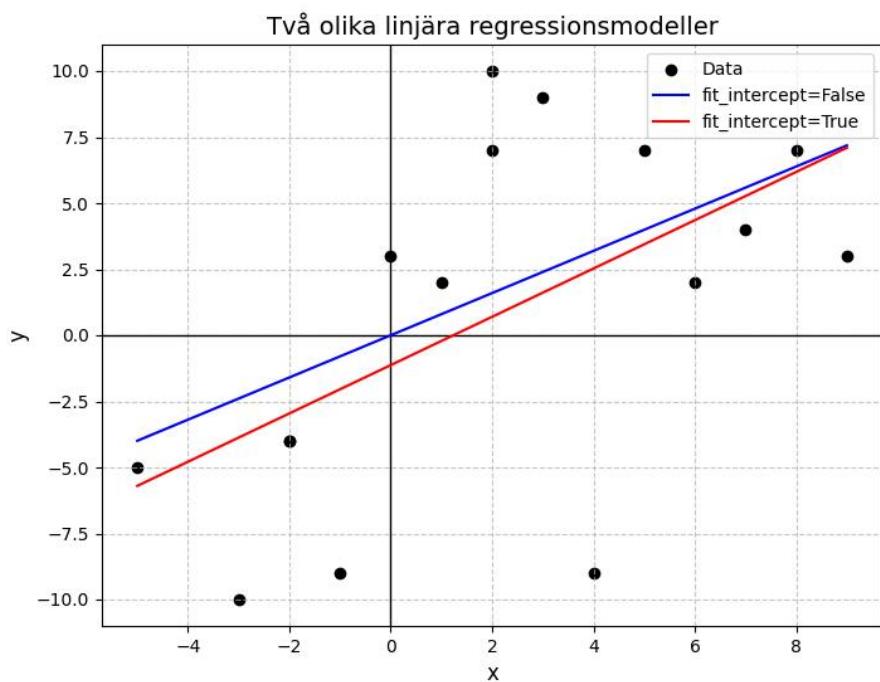
- ① Importera den linjära regressionsmodellen från *scikit-learn* biblioteket.
- ② Skapa ett dataset som vi kommer använda för demonstrationssyfte.
- ③ Instantiera en linjär regressionsmodell där hyperparametern `fit_intercept` är satt till `False`.
- ④ Träna modellen på den skapade exempeldatan.
- ⑤ En linjär regressionsmodell där hyperparametern `fit_intercept` är satt till `True` instantieras och tränas.
- ⑥ Vi skriver ut interceptet ( $m$  i den räta linjens ekvation  $y = kx + m$ ) och lutningen ( $k$  i den räta linjens ekvation  $y = kx + m$ ) för `lin_reg_1` modellen.
- ⑦ Vi skriver ut interceptet och lutningen för `lin_reg_2` modellen.

Parameters for lin\_reg\_1. Intercept: 0.0 Slope 0.8  
Parameters for lin\_reg\_2. Intercept: -1.13 Slope 0.91

Från resultatet av koden ser vi att den första modellen har interceptet 0 och lutningen 0.8 medan den andra modellen har interceptet -1.13 och lutningen 0.91. Vi kan visualisera de två modellerna vi skapat. Se Figur 1.7.

### 1.3.5 Grid search

Ovan provade vi manuellt två olika värden på en hyperparameter, `fit_intercept=False` och `fit_intercept=True`. Senare i boken kommer vi se att vi kommer kunna välja bland många olika värden för flera olika typer av hyperparametrar och det vore krångligt att behöva göra det manuellt då det snabbt kan bli väldigt mycket kod. Vad



Figur 1.7: Visualisering av två olika linjära regressionsmodeller.

vi istället använder, och detta kommer vi frekvent använda under bokens gång, kallas för `GridSearch` i *scikit-learn*. Detta är ett automatiserat sätt att välja hyperparametrar på. Vad algoritmen gör är att den för varje unik kombination av hyperparametrar tillämpar k-delad korsvalidering och ser vilken kombination som är optimal.

I koden nedan undersöker vi endast en hyperparameter `fit_intercept` som antar värdet `False` eller `True`.

```
from sklearn.model_selection import GridSearchCV          ①
model = LinearRegression()                                ②
hyperparameters = {'fit_intercept': [True, False]}        ③
grid_search = GridSearchCV(estimator=model,
                           ↵ param_grid=hyperparameters, scoring='neg_mean_squared_error',
                           ↵ cv=2)                         ④
grid_search.fit(example_data[["x"]], example_data["y"])    ⑤
```

- ① Importera `GridSearchCV` från *scikit-learn* biblioteket.
- ② Instantiera en linjär regressionsmodell.
- ③ Här specificerar vi vilka hyperparametrar vi vill undersöka. I detta fallet är det hyperparametern `fit_intercept` där vi vill prova värdena `True` och `False`. Senare i boken kommer vi för mer komplexa modeller se att vi kan undersöka flera kombinationer av hyperparametrar.
- ④ Vi instantierar en `GridSearch`. Notera att hyperparametern `cv` har i *scikit-learn* standardvärdet fem men vi har valt värdet två. Anledningen är eftersom vi senare kommer att skriva ut resultaten och vill kunna få plats med allt på bokens sidor. Vi specificerade även '`neg_mean_squared_error`', förklaringen till varför vi använder negativa värden på *MSE* ges i informationsrutan nedan med rubriken "Högre är bättre i *scikit-learn scoring*".
- ⑤ Vi genomför en *grid search* genom att använda `.fit()` metoden. Vi använder `example_data` från kodexemplet i Avsnitt 1.3.4.

**i** Högre är bättre i *scikit-learn scoring*

Notera att i kodexemplet ovan användes `scoring='neg_mean_squared_error'` i den fjärde annoteringen där ”neg” står för *negative*. Varför *negative*? Anledningen är att i *scikit-learn* är det en allmän princip att ”högre värden är bättre” i samband med att *score* beräknas. Det förstas lättast med ett exempel.

Antag att vi har två modeller och att modell 1 har en *MSE* på 100 och modell 2 har en *MSE* på 150. Då är modell 1 bättre eftersom felet (*MSE*) är lägre. Men i *scikit-learn* hade modell 2 betraktats som bättre eftersom *scikit-learn* följer den allmänna principen att ”högre värden är bättre” i samband med *scoring* vilket det uppenbart inte är i detta fallet. För att hantera detta tar vi de negativa värdena (*negative mean squared error*) och får -100 och -150. Nu är -100 större än -150 eftersom det är närmre noll. *Scikit-learn* hade därmed betraktat modell 1 som bättre. Nu blir det alltså korrekt rangordning av modellerna!

Vi kan inspektera resultaten av den GridSearch vi genomfört. Vi kommer kolla på några kolumner i taget eftersom samtliga kolumner av utrymmesskäl inte får plats på bokens sida.

**i** Notera att det finns en viss slumpmässighet i GridSearch eftersom vilken data som inkluderas i korsvalideringen som genomförs inte är deterministiskt. Det innebär att läsaren lär få andra siffror och eventuellt även en annan modellrangordning.

```
df_gs_results = pd.DataFrame(grid_search.cv_results_)  
df_gs_results.iloc[:, [0, 1, 5]]
```

- ① Spara resultaten från den *grid search* vi genomfört i en *pandas dataframe* och döp variabeln till `df_gs_results`.
- ② Skriv ut den första, andra och sjätte kolumnen från variabeln `df_gs_results`. Kom ihåg att indexering i Python börjar från 0.

	mean_fit_time	std_fit_time	params
0	0.000502	0.000502	{'fit_intercept': True}
1	0.001019	0.000020	{'fit_intercept': False}

Från tabellen ovan ser vi medelvärdet och standardavvikelsen för träningstiden för de två modellerna där den ena hade `fit_intercept = True` och den andra hade `fit_intercept = False`. I detta fallet ser vi att det gick snabbt att träna modellerna, inte ens en sekund tog det. För stora dataset och komplexa modeller kan modellträningen ta längre tid där det kan dröja några minuter, timmar eller dagar. Vi fortsätter att inspektera ytterligare kolumner från den *grid search* vi genomfört.

```
df_gs_results.iloc[:, [5, 6, 7, 8]]
```

	params	split0_test_score	split1_test_score	mean_test_score
0	{'fit_intercept': True}	-59.095576	-28.876354	-43.985965
1	{'fit_intercept': False}	-47.703118	-28.890138	-38.296628

Från tabellen ovan ser vi att vi har två resultat, `split0_test_score` och `split1_test_score`. Detta eftersom vi satte hyperparametern `cv=2` i koden tidigare. Högst resultat, det vill säga det värde som är närmast noll (eftersom vi har negativa värden), har den andra modellen där `fit_intercept = False`. Därför bör vi välja den modellen. Från resultattabellen nedan ser vi kolumnen `rank_test_score` och att den modellen alltså är rankad som nummer ett.

```
df_gs_results.iloc[:, [5, 8, 10]]
```

	params	mean_test_score	rank_test_score
0	{'fit_intercept': True}	-43.985965	2
1	{'fit_intercept': False}	-38.296628	1

Läsaren uppmanas att själv inspektera resultaten i sin helhet genom att exekvera nedanstående kod. Notera, vi visar inte fullständiga resultat från koden på grund av att de inte rymms på bokens sida.

```
pd.DataFrame(grid_search.cv_results_)
```

Läser vi den officiella dokumentationen för `GridSearch` från *scikit-learn* ser vi att följande hyperparametrar finns:

```
class sklearn.model_selection.GridSearchCV(estimator, param_grid,
    ↵ *, scoring=None, n_jobs=None, refit=True, cv=None, verbose=0,
    ↵ pre_dispatch='2*n_jobs', error_score=np.nan,
    ↵ return_train_score=False)
```

Vi ser där att hyperparametern `refit` har default-värdet `True`. Fortsätter vi läsa dokumentationen ser vi att det innebär följande: “*Refit an estimator using the best found parameters on the whole dataset.*” När vi genomfört vår *grid search*, det vill säga använt `.fit()` metoden, så tränas modellen om med de bästa hyperparametrarna på hela datasettet och sparar så vi kan använda den, exempelvis för att genomföra prediktioner.

```
from sklearn.model_selection import GridSearchCV

model = LinearRegression()

hyperparameters = {'fit_intercept': [True, False]

grid_search = GridSearchCV(estimator=model,
    ↵ param_grid=hyperparameters, scoring='neg_mean_squared_error',
    ↵ cv=3)

grid_search.fit(example_data[["x"]], example_data["y"])           ①

grid_search.predict([[6]])                                         ②
```

- ① En *grid search* genomförs och när de optimala hyperparametrarna hittats så tränas modellen om på hela datasettet med de optimala hyperparametrarna och sparar. I detta fallet sparas den optimala modellen i variabeln `grid_search`.
- ② Vi använder vår optimala modell för att genomföra en prediktion. Notera, vi genomför alltså en prediktion för  $x = 6$  och får ett värde enligt nedan, att resultatet är rimligt kan vi verifiera med hjälp av Figur 1.7.

```
array([4.78915663])
```

Koden nedan demonstrerar ytterligare några inspektioner som kan utföras för en `GridSearch`.

```

print("Best Hyperparameters from GridSearchCV:",
      ↵ grid_search.best_params_)
print("Best Cross-Validation MSE:", -grid_search.best_score_)

best_model = grid_search.best_estimator_
print(f"Best Model Coefficients. Intercept:
      ↵ {round(best_model.intercept_, 2)}, Slope:
      ↵ {round(best_model.coef_[0], 2)}")

```

Best Hyperparameters from GridSearchCV: {'fit\_intercept': False}  
 Best Cross-Validation MSE: 28.554233641184407  
 Best Model Coefficients. Intercept: 0.0, Slope: 0.8

Vi avslutar detta avsnitt med att nämna att det finns en klass `RandomizedSearchCV` i `scikit-learn`. Denna klass är motsvarigheten till `GridSearchCV` men skillnaden är att den slumpmässigt väljer vilka hyperparametervärden som utvärderas. I praktiken kan den vara användbar om det finns ett stort antal möjliga hyperparametervärden att undersöka. Den intresserade läsaren kan läsa dokumentationen från `scikit-learn` för `RandomizedSearchCV`.

### 1.3.6 Kategorisk data

När vi arbetar med ML-modeller, exempelvis den linjära regressionsmodellen, så behöver modellerna numerisk indata. Trots detta så vet vi från verkligheten att kategorisk data som antar värden i olika kategorier såsom färg och kön kan vara viktiga. Hur kan vi inkorporera sådan data? Vi omvandlar den kategoriska datan till numerisk data och kan därefter använda den i våra modeller. När vi arbetar med kategorisk data kan vi skilja mellan nominal data och ordinal data.

Nominal data kan anta ett fixt antal specifika värden där värdena saknar inbördes rangordning. Exempelvis {röd, grön, blå, svart} kan vi tolka som nominal data. Denna typ av data kan vi omvandla till numerisk data genom metoder som kallas för *one-hot-encoding* och *dummy-variable-encoding*. Ordinal data kan anta ett fixt antal specifika värden där värdena har en inbördes rangordning. Om vi exempelvis har data från en kundundersökning och frågar kunderna om deras upplevelse så kan möjliga svar vara {Ej nöjd, Neutral, Nöjd}. Denna typ av data kan vi omvandla till numerisk data genom en metod som kallas för *ordinal-encoding*. Notera, huruvida datan är nominal eller ordinal beror på situationen och hur vi väljer att tolka den. Tidigare saade vi att färgerna {röd, grön, blå, svart} kan tolkas som att de är på nominal skala. Om vi däremot exempelvis

vet att röda bilar är mer populära än gröna bilar som i sin tur är mer populära än blåa bilar som i sin tur är mer populära än svarta bilar, då finns det plötsligt en inbördes rangordning och datan kan då tolkas som att den är på ordinal skala. Här behöver alltså den som genomför ML-modellerings själv göra ett val vilket alltså är en del av konsten.

I koden nedan skapar vi exempeldata för demonstrationssyfte och sparar det i variabeln `nominal_df`.

```
nominal_data = {'färg': ['rød', 'grön', 'blå', 'svart', 'rød',
    ↵ 'grön', 'grön', 'rød']}
nominal_df = pd.DataFrame(nominal_data)
nominal_df
```

	färg
0	rød
1	grön
2	blå
3	svart
4	rød
5	grön
6	grön
7	rød

Datan från tabellen ovan kommer vi nu att omvandla med hjälp av *one-hot-encoding*. Eftersom vi har fyra distinkta kategorier {röd, grön, blå, svart} kommer vi att få fyra nya kolumner efter att vi har genomfört en *one-hot-encoding*. Radsumman för varje rad blir ett eftersom en färg endast kan vara en färg i taget. För att genomföra vår *one-hot-encoding* använder vi *Pandas* funktionen `get_dummies()`. Funktionen heter så eftersom variabler/kolumner som endast kan anta värdet 0 eller 1 kallas för dummyvariabel.

```
oh_encoding = pd.get_dummies(nominal_df, dtype = int)
oh_encoding
```

	färg_blå	färg_grön	färg_röd	färg_svart
0	0	0	1	0

	färg_bla	färg_grön	färg_röd	färg_svart
1	0	1	0	0
2	1	0	0	0
3	0	0	0	1
4	0	0	1	0
5	0	1	0	0
6	0	1	0	0
7	0	0	1	0

Ovan demonstrerade vi *one-hot-encoding*. Nu kommer vi demonstrera *dummy-variable-encoding*. Generellt gäller att om vi använder *dummy-variable-encoding* för en variabel med k kategorier så kommer vi få k-1 nya variabler/kolumner. I vårt fall har vi fyra kategorier och får därför 3 nya variabler. Detta räcker eftersom i kodexemplet nedan markeras grön med kolumnen `färg_grön = 1`, röd med kolumnen `färg_röd = 1`, svart med kolumnen `färg_svart = 1` och i de fall alla tre kolumnerna har värdet 0 vet vi att det är färgen bla som åsyftas. Se exempelvis raden med index 2 där alla kolumnerna har värdet 0, då vet vi att det är färgen bla.

```
dummy_encoding = pd.get_dummies(nominal_df, dtype = int,
                                drop_first = True)
dummy_encoding
```

①

- ① Notera att för såväl *one-hot-encoding* som för *dummy-variable-encoding* kan vi använda pandas funktionaliteten `pd.get_dummies()`. I det fallet vi önskar genomföra en *dummy-variable-encoding* specificerar vi hyperparametern `drop_first = True`, precis som vi gjorde i detta kodexemplet.

	färg_grön	färg_röd	färg_svart
0	0	1	0
1	1	0	0
2	0	0	0
3	0	0	1
4	0	1	0
5	1	0	0
6	1	0	0
7	0	1	0

Men vad är skillnaden mellan *one-hot-encoding* och *dummy-variable-encoding*? Rent informationsmässigt ger metoderna samma information men när vi ska använda nominal data i den linjära regressions-modellen så ska vi använda *dummy-variable-encoding* medan vi för övriga modeller använder *one-hot-encoding*. Varför det är så går att matematiskt motivera men det är inget vi går djupare in på i denna bok.

Nedan demonstrerar vi *ordinal encoding* för kolumnen `Pris` där vi ser att det finns en tydlig rangordning, låg, medel och hög.

```
ordinal_data = {'Produkt': ['XXX', 'YYY', 'XXY', 'XYX', 'YXX'],
                'Pris': ['hög', 'medel', 'låg', 'hög', 'låg']}
ordinal_df = pd.DataFrame(ordinal_data)                                     ①
ordinal_df
```

- ① Vi skapar exempeldata för demonstrationssyfte och sparar den i variabeln `ordinal_df`.

	Produkt	Pris
0	XXX	hög
1	YYY	medel
2	XXY	låg
3	XYX	hög
4	YXX	låg

```
mapping = {"låg": 1, "medel": 2, "hög": 3}                                ①
ordinal_df["Pris"] = ordinal_df["Pris"].map(mapping)                      ②
ordinal_df
```

- ① Vi specificerar vilken korrespondens vi önskar. Exempelvis kommer `låg` korrespondera med 1.  
 ② Vi använder `map()` funktionen från Pandas för att ersätta värdena `låg`, `medel` och `hög` med deras korresponderade värden sparade i variabeln `mapping` i detta fall.

	Produkt	Pris
0	XXX	3
1	YYY	2

	Produkt	Pris
2	XXY	1
3	XYX	3
4	YXX	1

Önskar vi lägga till en kolumn istället för att ersätta den ursprungliga **Pris** kolumnen kan vi göra det också.

```
ordinal_df_2 = pd.DataFrame(ordinal_data)

ordinal_df_2['PrisOrdinalKodad'] =
    ↵ ordinal_df_2['Pris'].map(mapping)
ordinal_df_2
```

	Produkt	Pris	PrisOrdinalKodad
0	XXX	hög	3
1	YYY	medel	2
2	XXY	låg	1
3	XYX	hög	3
4	YXX	låg	1

I exemplen ovan har vi arbetat med data som är i formatet *Pandas dataframe*. Arbetar vi med data som är i *NumPy array* format kan vi använda funktionerna **OneHotEncoder** och **OrdinalEncoder** från *scikit-learn*. Den intresserade läsaren uppmanas att läsa dokumentationen. Dessa funktionerna går att använda för *Pandas dataframes* också men kommer då konvertera den ursprungliga *dataframen* till en *NumPy array*. Vi demonstrerar detta nedan.

```
nominal_df
```

	färg
0	röd
1	grön

<hr/> <hr/> färg	
2	blå
3	svart
4	röd
5	grön
6	grön
7	röd

---

```
from sklearn.preprocessing import OneHotEncoder
OneHotEncoder(sparse_output=False).fit_transform(nominal_-
↳ df[['färg']])

array([[0., 0., 1., 0.],
       [0., 1., 0., 0.],
       [1., 0., 0., 0.],
       [0., 0., 0., 1.],
       [0., 0., 1., 0.],
       [0., 1., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.]])
```

### 1.3.7 Feature engineering

*Feature engineering* handlar om vilka variabler som ska väljas (variabelselektion) i en modell, skapandet av nya variabler och transformering av variabler. Generellt sett åsyftas de oberoende variablerna (*features*) men även exempelvis transformering av den beroende variabeln kan göras. Nedan skapar vi exempeldata som vi kommer använda oss av.

```
example_data = {'pris (tkr)': [4200, 2500, 7900, 6100, 6500],
                'kvm': [120, 80, 250, 200, 150],
                'antal rum': [4, 3, 7, 5, 4],
                'byggår': [2005, 2016, 1980, 1990, 2020]}
example_data = pd.DataFrame(example_data)
example_data
```

	pris (tkr)	kvm	antal rum	byggår
0	4200	120	4	2005
1	2500	80	3	2016
2	7900	250	7	1980
3	6100	200	5	1990
4	6500	150	4	2020

När vi skapar en modell behöver vi ta ställning till vilka variabler som ska användas, detta benämns *variabelselektion*. I en modell kanske vi väljer att modellera hur variabeln **pris** (tkr) beror på variabeln **kvm** som står för kvadratmeter. I en annan modell kanske vi väljer att modellera hur variabeln **pris** (tkr) beror på variablerna **kvm** och **antal rum**. När vi tränat dessa två modeller på träningsdatan så utvärderar vi dem på valideringsdatan och går vidare med den som presterar bäst.

Variabelselektion är generellt sett mycket viktigt eftersom oavsett hur komplicerade modeller vi använder oss av så kommer de inte att prestera bra om fel variabler används. ”*Shit in, shit out*” är ett uttryck för detta, det vill säga stoppar vi in dåliga variabler i modellerna får vi ut dåliga prediktioner. Hur väljer vi då vilka variabler som ska användas? Den kanske viktigaste metodiken är att använda sitt omdöme och eventuell teori som säger att en viss variabel påverkar den beroende variabeln ( $y$ ). Exempelvis finns det forskning som visar att motion påverkar hälsan, om vi då hade velat modellera en människas hälsa så finns det teoretiska skäl till att använda motion som en oberoende variabel. Om vi tror att flera olika uppsättningar av variabler kan vara användbara så kan vi träna olika modeller på träningsdatan och därefter på valideringsdatan utvärdera vilken som presterar bäst. Vi betonar här att vi behöver använda vårt omdöme på vilka uppsättningar av variabler som används. Detta eftersom ifall vi exempelvis tränar 1000 olika modeller på träningsdatan så är sannolikheten troligtvis ganska hög att det av ren slump blir någon modell som presterar bra. Det finns även algoritmer som kan användas för att genomföra variabelselektion på ett mer automatiserat sätt. Exempelvis, när vi lär oss om beslutsträd och *random forest* modeller senare i boken, kommer vi se att dessa modeller kan användas för att bedöma hur viktiga olika variabler är genom något som benämns för *feature importance* (gås igenom i Avsnitt 4.3.5). Poängen att man kan använda de variabler som har högst *feature importance* eftersom de enligt modellen är viktigast. Den läsare som vill fördjupa sig inom algoritmisk variabelselektion kan googla på *Feature selection* och läsa *scikit-learns* dokumentation, se följande länk: [https://scikit-learn.org/stable/modules/feature\\_selection.html](https://scikit-learn.org/stable/modules/feature_selection.html). Vi poängterar dock att metoden där omdömet och teori används är viktig, förkasta alltså inte omdömet bara för att eventuella algoritmer används för variabelselektion.

**i** Man kan fråga sig om det inte vore bäst att helt enkelt använda alla tillgängliga variabler i en modell? Rent generellt är det en dålig idé eftersom modellen troligtvis kommer generalisera dåligt till ny, osedd data. Den blir alltså överanpassad.

En generell princip inom modellering benämns *principle of parsimony* och den säger att givet en viss prestationsnivå så försöker man ha en så enkel modell som möjligt. Rent generellt, inom vetenskapliga sammanhang, benämns det Ockhams rakkniv.

Vi kan även skapa nya variabler. Se kodexemplet nedan där vi skapar en ny variabel som beräknar genomsnittligt antal kvadratmeter per rum och en dummyvariabel som är 1 ifall huset är byggt senare än år 2000 och 0 annars. Vi kanske vet att hus byggda efter år 2000 är mer attraktiva då ny typ av byggtteknik började användas. Vi ser alltså att vi kan använda vår domänkunskap, om hus i detta fall, för att skapa en ny variabel. Det gäller generellt att domänkunskap är användbart i modellering. Ibland kanske vi redan har det från början av projektet och om vi inte har det lär vi behöva läsa in oss för att faktiskt få det, åtminstone grundläggande sådan.

```
example_data['snitt_kvm_per_rum'] = example_data['kvm'] /  
    ↵ example_data['antal rum']  
  
example_data['ny_husmodell'] = (example_data['byggår'] >  
    ↵ 2000).astype(int)  
  
example_data
```

	pris (tkr)	kvm	antal rum	byggår	snitt_kvm_per_rum	ny_husmodell
0	4200	120	4	2005	30.000000	1
1	2500	80	3	2016	26.666667	1
2	7900	250	7	1980	35.714286	0
3	6100	200	5	1990	40.000000	0
4	6500	150	4	2020	37.500000	1

Variablerna vi skapade ovan kan därefter användas i en modell och genom att utvärdera modellen på valideringsdatan kan vi se om variablerna är till hjälp.

Ett annat sätt att få nya variabler är att samla in mer data. Hade vi exempelvis velat prediktera huspriser så vet vi att geografiskt läge och närhet till havet är två faktorer som kan ha stor påverkan på priset. I praktiken kan det vara en kostsam process att samla in mer data och en bedömning kring om det är värt det eller inte behöver göras.

Variabler kan även transformeras. Inom ML standardiseras man ofta variabler för att de ska vara på samma skala. Logiken är att om variablerna är på samma skala kan det bli lättare för modellen att se mönster. I koden nedan demonstreras `StandardScaler` i *scikit-learn* som standardiseras variablerna till att ha medelvärdet 0 och standardavvikelsen 1. Notera, när vi säger att variablerna har medelvärdet 0 och standardavvikelsen 1 menar vi att respektive kolumn har medelvärdet 0 och standardavvikelse 1.

```
from sklearn.preprocessing import StandardScaler
import numpy as np

example_data = np.array([[1.0, 2.0], [3.0, 6.0], [5.0, 10.0]])      ①

scaler = StandardScaler()
example_data_scaled = scaler.fit_transform(example_data)            ② ③

print("Original data:\n", example_data)
print("Scaled data:\n", example_data_scaled)
print("Medelvärde efter skalning:", np.mean(example_data_scaled,
    axis=0))                                                       ④
print("Standardavvikelse efter skalning:",
    np.std(example_data_scaled, axis=0))                                ⑤
```

- ① Vi skapar exempeldata för demonstrationssyfte.
- ② Vi instantierar `StandardScaler` från *scikit-learn*.
- ③ Vi transformeras vår exempeldata med `StandardScaler` till att ha medelvärdet 0 och standardavvikelsen 1. Resultatet sparas i den nya variabeln `example_data_scaled`.
- ④ Vi demonstrerar att respektive kolumn nu har medelvärdet 0.
- ⑤ Vi demonstrerar att respektive kolumn nu har standardavvikelsen 1.

Original data:

```
[[ 1.  2.]
 [ 3.  6.]
 [ 5. 10.]]
```

Scaled data:

```

[[ -1.22474487 -1.22474487]
 [ 0.          0.        ]
 [ 1.22474487  1.22474487]]
Medelvärde efter skalning: [0. 0.]
Standardavvikelse efter skalning: [1. 1.]

```

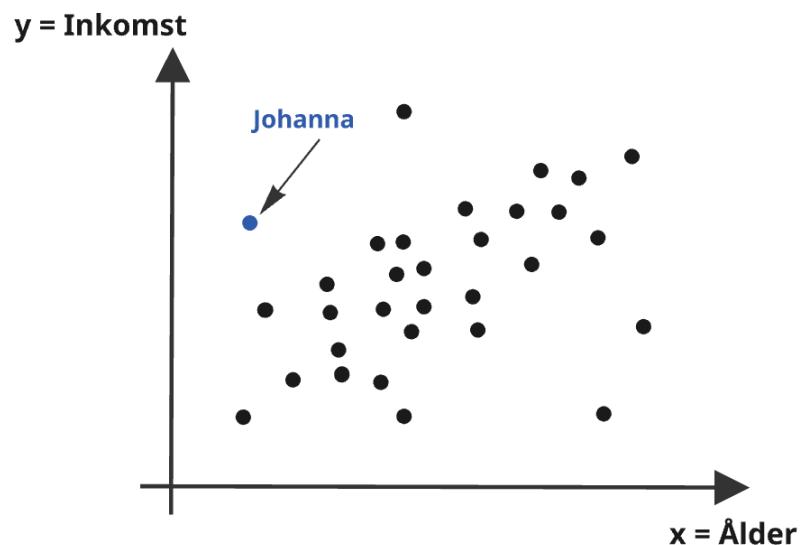
Det går att standardisera variabler på fler sätt, exempelvis kan den intresserade läsaren läsa *scikit-learn* dokumentationen för *MinMaxScaler* som gör att variablerna befinner sig i ett visst intervall, exempelvis mellan 0 – 1. Inom ML förekommer det också att variabler logaritmeras (log-transformeras), exempelvis för att de ska se mer normalfördelade ut. Vi går dock inte djupare in på det utan nämner endast det för den läsare som har kunskaper i mer avancerad statistik. Avslutningsvis nämner vi att senare i boken, Kapitel 5, kommer vi lära oss en modell som heter *Principal Component Analysis (PCA)* som också möjliggör oss att transformera variabler.

## 1.4 Tvärnittsdata, tidsseriedata och paneldata

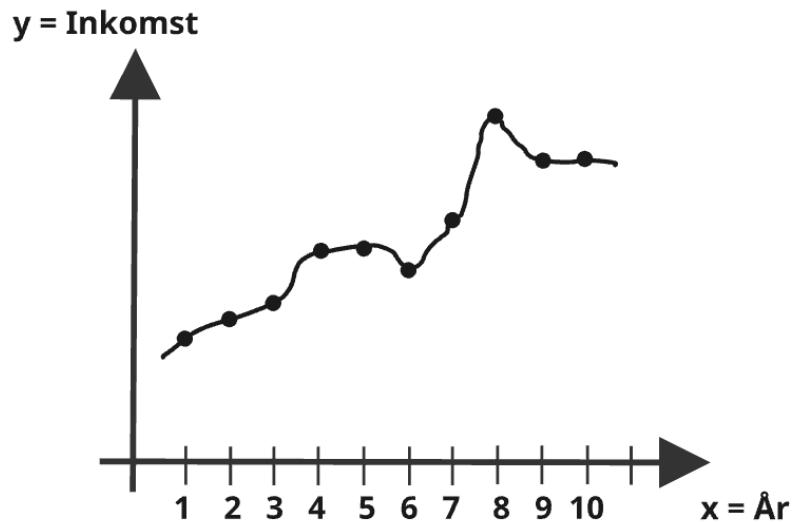
Data kan kategoriseras på olika sätt. Bra kategorier som är bra att känna till är *tvärnittsdata*, *tidsseriedata* och *paneldata*.

- *Tvärnittsdata* är sådan data där observationer för olika individer (det kan exempelvis vara människor, företag eller länder) samlas in vid *en tidpunkt*. Se Figur 1.8 på hur tvärnittsdata kan se ut.
- *Tidsseriedata* är sådan data där observationer över tid samlas in för en individ (det kan exempelvis vara människor, företag eller länder). Se Figur 1.9 på hur tidsseriedata kan se ut.
- *Paneldata* är sådan data där observationer för individer (det kan exempelvis vara människor, företag eller länder) samlas in över tid. Notera alltså att tvärnittsdata och tidsseriedata kan betraktas som specialfall av paneldata. *Tvärnittsdata* är en tidpunkt från paneldatan och *tidsseriedatan* är en individ från paneldatan. Se Figur 1.10 på hur paneldata kan se ut.

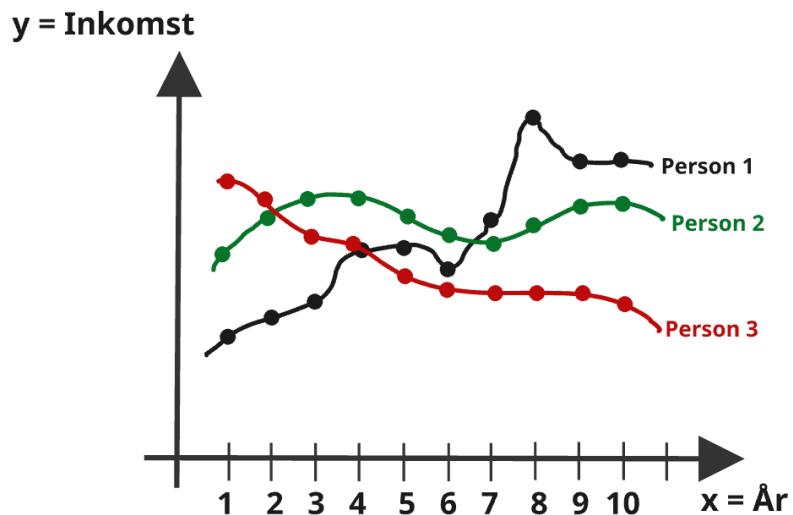
I boken kommer vi mestadels arbeta med tvärnittsdata, bland annat när vi tillämpar regressionsmodeller och klassificeringsmodeller. Tidsseriedata kommer vi mycket översiktligt nämna i Kapitel 9. Paneldata kommer vi inte gå in på i denna bok. Det finns mycket information på bland annat internet där analysmetoder och modeller för respektive data-kategori behandlas som den intresserade läsaren kan läsa in sig på.



Figur 1.8: Exempel på hur tvärsnittsdata kan se ut. Vi har ålder och inkomst för olika personer. Exempelvis har vi i figuren markerat en observation där personen heter Johanna.



Figur 1.9: Ett exempel på hur tidsseriedata kan se ut. Vi följer inkomsten över tid för en person.



Figur 1.10: Ett exempel på hur paneldata kan se ut. Vi har följt inkomsterna över tid för tre olika personer.

### **i Tidsserier - ARMA**

Den läsare som vill lära sig om tidsseriemodeller kan börja kolla på de modeller som benämns för *ARMA* (*Autoregressive Moving Average*) där *AR*- och *MA*-modellerna är specialfall.

## 1.5 Övningsuppgifter

Övningsuppgifter för kapitlet finns på bokens hemsida:  
[https://github.com/AntonioPrgomet/ai\\_tillaempad\\_ml](https://github.com/AntonioPrgomet/ai_tillaempad_ml)

# Kapitel 2

## Ett ML-projekt från början till slut

I detta kapitel kommer vi börja med att gå igenom en praktisk checklista som kan vara till hjälp när vi genomför ett ML-projekt. Denna checklista kommer vi sedan att använda när vi genomför ett kodexempel, från början till slut, där huspriser i Kalifornien kommer modelleras. Därefter tar vi ta upp några vanligt förekommande utmaningar inom ML-projekt. Kapitlet avslutas med ett avsnitt om *scikit-learn*, ett mycket populärt och använt bibliotek inom AI/ML branschen som vi kommer att använda genom hela boken.

### 2.1 En checklista för Maskininlärning

Vi presenterar här en checklista som ska ge en orientering över möjliga steg att genomföra i ett ML-projekt. Checklistan har sju steg och det är lätt att få intrycket att dessa steg görs i en rak progression - vi gör steg ett, går vidare till steg två för att därefter fortsätta med steg tre och så vidare. Så är det inte. Ofta kan vi behöva hoppa mellan stegen, exempelvis för att vi fått en insikt som påverkar vår analys från föregående steg eller för att det vi egentligen ville göra inte är möjligt. Så i verkligheten är vi beredda på att vi behöver arbeta iterativt där vi går ”fram-och-tillbaka” och hoppar mellan stegen. Notera också att checklistan ska utgöra ett stöd i vårt arbete och inget som vi varken behöver eller ska följa slaviskt.

### 2.1.1 Definiera problemet och skapa en helhetsbild

Det första steget handlar om att förstå vad vi vill uppnå och försöka skapa sig en helhetsbild. Några frågor vi kan ställa oss i detta steget är:

- Vad är målet med projektet?
- Om vi uppfyller målet, kommer det vi skapar vara användbart?
- Finns det befintliga lösningar eller “*workarounds*” vi kan använda?
- Hur mäter vi projektets framgång?
- Vilka personer är det bra att involvera i projektet och exempelvis diskutera med?

### 2.1.2 Få tillgång till data

Data är av central betydelse för att skapa ML-modeller och ett första steg är att få tillgång till den. När vi laddar in data i olika sammanhang är det ofta en god idé att automatisera flödet så att vi kontinuerligt kan få uppdaterad data. Några frågor vi kan ställa oss i detta steget är:

- Vilken data behöver vi för projektet och hur får vi tillgång till den?
- Hanterar vi data på ett lagenligt sätt? Exempelvis kan det finnas skyddade uppgifter eller information som behöver anonymiseras och rent allmänt hanteras på ett juridiskt korrekt sätt.
- Om vi ska använda data för ML-modellering kan vi här lägga undan ett dataset som vi använder som testdata. Testdata ska inte användas förrän på slutet. Vi skapar även träningsdata och valideringsdata. Alternativt så har vi endast träningsdata och testdata om vi senare använder k-delad korsvalidering.

**i Finns datan för det vi ursprungligen ville göra? En iterativ arbetsprocess.**

I steg ett från checklistan hade vi kanske ett ursprungligt mål. I detta steget, när vi ska få tillgång till data, kanske vi upptäcker att det vi vill göra inte går att göra på grund av att data saknas. Då kan vi behöva gå tillbaka till steg ett från checklistan och fundera på om vi behöver ändra målsättning med projektet. Det kan vara att målet ändras eller så kan målet rentav vara att börja samla in den data som krävs för att vi ska kunna uppnå vårt ursprungliga mål.

### 2.1.3 Utforska datan, gör en *exploratory data analysis (EDA)*

Att utforska och analysera data genom att göra en *exploratory data analysis (EDA)* är ett viktigt steg för att förstå vad det är för data vi arbetar med. I detta steget kan vi genomföra olika beräkningar såsom att beräkna medelvärdet, medianer, typvärdet och liknande. Vi kan även genomföra olika visualiseringar för att förstå olika samband och se mönster. I detta steget, när vi börjar arbeta med datan bör vi alltid arbeta med en kopia av datan så att vi inte av misstag ändrar ursprungsdatan. Notera att *EDA* genomförs på träningsdatan. Några frågor vi kan undersöka i detta steget är:

- Finns det mönster som syns i datan? Vad säger dessa? Vad kan de tänkas bero på?
- Finns det några visualiseringar som kan vara särskilt användbara för att hjälpa oss förstå datan och upptäcka mönster?
- Finns det något som verkar fel i datan? Hur kan vi verifiera om det faktiskt är fel eller inte?
- Finns det saknade värden, *missing values*, i datan?

### **i** Behövs ML-modellering eller räcker en *EDA*?

I sammanhang där vi vill lära oss mer om ML, exempelvis på en utbildning, är det ganska självklart att vi inkluderar ML-modellering i olika uppgifter eller arbeten. I verkligheten är det dock många problem som kan lösas genom att genomföra en *EDA*. Exempelvis kanske en marknadsföringsavdelning vill veta typiska beteenden och attribut hos dess kunder. Då kan medelålder, medelinkomst, antal köp kunderna gjort och dylikt kanske räcka? Det behövs alltså ingen ML-modellering i det fallet. Det kan dock mycket väl vara så att med tiden utvecklas frågorna från marknadsföringsavdelningen som kanske börjar undra vad sannolikheten att olika kunder kommer *churna* är (med det engelska ordet *churn* menar vi kunder som slutar köpa företagets produkter), då blir ML-modellering relevant.

Det faktum att många tycker det är spännande och kul med ML-modellering kan i verkligheten lätt leda till en överdriven benägenhet att vilja skapa modeller. En sund fråga att fråga sig är därför, *“krävs det ML-modellering för det vi vill uppnå?”*.

Att träna på att genomföra *EDA* är en bra idé, dels för att det löser många problem i verkligheten, dels för att det behöver göras innan vi genomför eventuell ML-modellering.

#### 2.1.4 Bearbeta datan

Liksom i föregående steg bör vi i detta steget arbeta med en kopia av datan för att inte ändra eller förstöra ursprungsdatan. När vi bearbetar datan och gör olika transformationer är det en god idé att skriva funktioner för de transformationer som görs. Detta eftersom vi då kan genomföra samma transformér när vi får uppdaterad data och dessutom hantera våra data-transformér som en hyperparameter om vi skapar ML-modeller. Fördelen med att betrakta data-transformér som en hyperparameter är att vi exempelvis kan använda oss av `GridSearch` för att utvärdera det. Några frågor vi kan ställa oss i detta steget är:

- Finns det outliers i datan? Det vill säga värden som kraftigt skiljer sig från andra värden? Vad kan det bero på? Hur ska vi hantera det? Detta steg är en viktig del i det som brukar benämñas *data cleaning*.
- Om det finns *missing values* i datan, hur ska vi hantera det? Vi kanske väljer att ta bort de raderna eller ersätta de saknade värdena med motsvarande medelvärdén

eller median för övriga värden i den kolumnen. Detta steg är också en viktig del i det som brukar benämñas *data cleaning*.

- Det kan finnas data/variabler som inte är användbara för vårt ändamål, dessa kan vi ta bort.
- Om vi genomför någon *feature engineering*, exempelvis skapa nya variabler eller transformera befintliga, så görs det vanligtvis här. Det kan även ske i samband med ML-modelleringen som är nästa steg. I praktiken är det inte en skarp gräns då man ofta arbetar iterativt mellan dessa två stegen.

### **i** *Data cleaning*

I pedagogiska exempel är datan vi ska genomföra ML-modelleringen på ofta tillrättalagd eller bearbetad och datan behöver därför inte städas (*data cleaning*). I verkligheten är det dock ett viktigt steg som ofta kan vara tidskrävande och svårt.

## 2.1.5 ML-modellering

I föregående steg har vi bearbetat datan så att den kan användas för ML-modellering. När vi nu ska skapa ML-modeller kan vi tänka på följande:

- Det är rimligt att i början prova flera olika modeller med standardvärden för hyperparametrarna. Detta för att se vilken typ av modeller som verkar prestera bra. Med erfarenhet kan vi bilda oss en intuition för vilka modeller som kan prestera bra. Exempelvis kan vissa modeller vara användbara för vissa typer av problem och tillämpningar, beroende på hur komplicerat datasetet är så kan mer eller mindre komplexa modeller också behövas. Ytterst så utvärderar vi dock modellerna på valideringsdatan för att se vilka modeller som faktiskt presterar bra.
- Från steget ovan kan vi fortsätta justera hyperparametrarna på de modeller vi vill fortsätta undersöka. Vi kan även prova att inkludera olika variabler, d.v.s. arbeta med variaelselektion. Notera, detta är en iterativ process.
- Till slut kommer vi ha en modell som presterar bäst på valideringsdatan. Den modellen kan vi börja använda, givet att den presterar tillräckligt bra och uppfyller de krav vi har, men innan dess bör vi slutligen utvärdera modellens generaliseringsförmåga på testdatan. Detta för att se ungefärligt hur bra resultatet vi kan förvänta oss när vi börjar använda modellen skarpt.

### **i** Överanpassa inte modellen till testdatan

Vi ska inte justera den slutgiltiga modellen (t.ex. genom att ändra hyperparametrar eller välja en helt annan modell) efter att provat dess generaliseringsförmåga på test-datan. Det enda vi då åstadkommer är att vi överanpassar vår modell till test-datan. När vi valt en modell så utvärderar vi alltså dess generaliseringsförmåga på testdatan som ett *sista steg*. År vi inte nöjda behöver vi börja om från början. Notera skillnaden mellan att börja om från början och justera hyperparametrar tills modellen presterar bra nog på testdatan.

## 2.1.6 Presentera din lösning för intressenter

Ofta ska vi presentera en lösning för olika intressenter. Det kan exempelvis vara kollegor inom ett team, marknadsförare, chefer, projektledare eller ledningsgrupp. Fundera på vem som är målgruppen och vilken information de har nyttja av. Anpassa kommunikationen utifrån det. Några saker vi kan tänka på i detta steget är:

- Vilken abstraktionsnivå vi ska använda. Presenterar vi en lösning för tekniskt orienterade personer så kanske de är intresserade av detaljer kring modelleringsval och dylikt. Presenterar vi en lösning för en ledningsgrupp så är de ofta intresserade av konkreta resultat, ageranden som arbetet kan tänkas leda till samt ekonomiska konsekvenser.
- Innan en presentation bör vi fundera igenom vad budskapet är och hur vi kan förmedla det på ett effektivt sätt.
- Visualiseringar av olika slag kan ofta vara hjälpsamma.

## 2.1.7 Produktionssättning av modellen och övervakning av implementeringen

Om vi lyckas skapa en modell och faktiskt ska produktionssätta den så kan följande vara bra att tänka på:

- Exempelvis kan vi skapa olika enhetstest för att säkerställa att funktionaliteten är som den ska vara.
- Det är en god idé att bevaka modellens prestanda eftersom modeller kan bli sämre med tiden, till exempel för att datan ser annorlunda ut till följd av förändringar inom innovation, ekonomi och beteenden hos människor.
- Ofta tränas modeller regelbundet om på ny data, exempelvis månadsvis eller veckovis.

När vi tränat en modell som vi tänkt återanvända så sparas modellen så att den inte behöver tränas om varje gång vi ska använda den. Det är enkelt att göra, vi kan exempelvis använda oss av biblioteken `joblib` eller `pickle`. Vi demonstrerar `joblib` nedan.

```
from sklearn.linear_model import LinearRegression  
from sklearn.datasets import make_regression  
from sklearn.model_selection import train_test_split  
import joblib  
  
x, y = make_regression(n_samples=100, n_features=1, noise=10,  
    ↵ random_state=42)  
x_train, x_test, y_train, y_test = train_test_split(x, y,  
    ↵ test_size=0.2, random_state=42)  
  
model = LinearRegression()  
model.fit(x_train, y_train)  
  
joblib.dump(model, 'our_linear_model.pkl')  
print("Model saved!")  
  
loaded_model = joblib.load('our_linear_model.pkl')  
print("Model loaded!")  
  
prediction = loaded_model.predict([x_test[5]])  
print(f"Predicted value: {prediction}")
```

- ① Vi importerar biblioteken vi behöver för exemplet.
- ② Vi skapar ett enkelt dataset som används i detta kodexemplet för demonstration.
- ③ Vi delar upp datan i träningsdata och testdata. Vi använder `random_state=42` för att få reproducerbara resultat.
- ④ Vi instantierar en linjär regressionsmodell.
- ⑤ Vi tränar den instantierade modellen.
- ⑥ Vi sparar modellen genom att använda oss av `joblib.dump(model, 'our_linear_model.pkl')`. I mappen där vi har sparat vårt kodskript kommer det sparas en fil som heter "our\_linear\_model.pkl", den filen innehåller den sparade modellen. Vi kan välja att spara modellen på en annan sökväg, exempelvis hade vi då kunnat skriva: `joblib.dump(model, 'C:/Users/antonio/Downloads/test/our_linear_model.pkl')`. Notera, du behöver alltså då ändra sökvägen till

- motsvarigheten på din dator.
- ⑦ Vi skriver ut *Model saved!* för att det ska vara pedagogiskt att se när modellen har sparats.
  - ⑧ Om vi exempelvis stänger av datorn så kan vi nu ladda in modellen, utan att behöva träna om den, vilket vi gjorde i denna kodraden.
  - ⑨ Vi skriver ut *Model loaded!* för att det ska vara pedagogiskt att se när modellen har laddats in.
  - ⑩ Vi använder vår laddade modell för att genomföra en prediktion.
  - ⑪ Vi skriver ut det predikterade värdet.

**Model saved!**

**Model loaded!**

**Predicted value:** [-12.80652919]

## i MLOps

När vi säger att vi produktionssätter en modell menar vi helt enkelt att den börjar användas. Detta kan ske på olika sätt. Exempelvis:

- En modell kan användas på en hemsida vilket är fallet med chattbotar såsom ChatGPT som nås via en hemsida.
- En modell kan användas i applikationer såsom sociala medier där ML-modeller rekommenderar flöde som gör att användarna spenderar mer tid på appen.
- En modell kan användas i olika apparater såsom kameror för att exempelvis detektera fel i en industriell produktionsprocess eller för självkörande bilar. Rent generellt är ML-modeller användbara inom det område som benämns *internet of things* (IoT) eller på svenska, *sakernas internet*.
- En modell kan användas för att skapa en ny kolumn med predikterade värden i en SQL-databas. Denna kolumn med predikterade värden kan därefter användas för olika ändamål. Om man till exempel predikterar sannolikheten att en kund kommer *churna* så kan man på ett företag sätta in åtgärder, såsom marknadsföring, för alla kunder som exempelvis har över 35% risk att *churna*.

Arbetet med att produktionssätta och underhålla maskininlärningsmodeller är en del av det område som benämns för *MLOps*. Det finns specialiserad information om detta, exempelvis via internet eller litteratur, som den intresserade läsaren kan fördjupa sig inom.

### **i De flesta AI/ML-projekten misslyckas**

De flesta AI/ML-projekten uppnår inte de ursprungligen satta målen eller att ens passera någon form av prototyp-stadie. Det kan bero på anledningar såsom brist på behövlig data, dåliga modeller eller att rätt kompetens och resurser saknas. Oavsett anledning är det bra att ha i åtanke för att bilda sig rimliga förväntningar. Det cirkulerar olika uppskattningar om att cirka 85% av alla ML-projekt misslyckas. Det är därmed orimligt att i verkligheten förvänta sig att alla projekt vi arbetar med faktiskt kommer till steget att de börjar användas. Men ofta får vi ändå ut något från projekten som skapar värde. Tänk kreativt - vad kan vi göra? Vårt ursprungliga mål var kanske att skapa en ML-modell men även om det inte lyckas kanske den *EDA* vi gjorde är användbar? Undvik därför att tänka för binärt i termer av att projekt "lyckas" eller "misslyckas" och försök istället se hur det som gjorts kan vara användbart eller vilka lärdomar som tagits och hur det kan komma till nytta för framtiden.

## **2.2 Ett kodexempel från början till slut - Huspriser i Kalifornien**

I detta avsnitt kommer vi gå igenom ett kodexempel från början till slut. Syftet är att läsaren ska se hur ett ML-projekt, från början till slut, kan se ut snarare än att skapa en så bra modell som möjligt. Vi kommer gå igenom de sju stegen från checklistan i föregående avsnitt. Vi börjar med att förstå problemet och skapa oss en helhetsbild.

### **1. Förstå problemet och skapa en helhetsbild**

Vi kommer arbeta med ett välkänt och ofta använt dataset, som heter "*California Housing dataset*". Datan finns tillgänglig på bokens hemsida. Du kan även läsa mer om datasettet på följande länk:

<https://www.kaggle.com/datasets/camnugent/california-housing-prices>

**i** Kaggle är en användbar och välkänd sida där du kan hitta många olika dataset och tillämpningar med färdig kod inom *data science* och maskininlärning. Majoriteten av de som arbetar med ML känner till Kaggle och har använt det någon gång. Kolla gärna igenom sidan och gör några projekt för att få erfarenhet och bygga upp portföljprojekt som du kan lägga upp på din GitHub-profil och till exempel använda i ditt CV.

Den beroende variabeln ( $y$ ) är en variabel som heter `median_house_value` och visar medianvärdet för hus i olika distriktsnivå i Kalifornien, USA, på 1990-talet. Notera att datan är på distriktsnivå och inte för enskilda hus. Därför kan det exempelvis finnas 129 sovrum för en datapunkt eftersom det i det distriktet då finns 129 sovrum.

Målet är att skapa en ML-modell för att kunna prediktera medianvärdet för hus i olika distriktsnivå.

## 2. Få tillgång till datan

Datan finns sparad som en *csv*-fil på bokens hemsida. Vi kommer ladda in den och göra en första inspektion för att kolla att allting är ok. Notera att vi här inte gör en direkt analys som vi kommer använda för vår ML-modellering, det görs i nästa steg via *EDA*. Innan vi laddar in datan börjar vi dock med att importera bibliotek som vi kommer använda. Notera, detta är placerat i början av koden men i praktiken så märker vi i efterhand vilka bibliotek vi behöver och lägger till dessa. Det är alltså inte något vi direkt vet och skriver in även om man kan få det intycket eftersom koden ligger i början. Som så ofta är fallet är det en iterativ process.

```
import numpy as np  
import matplotlib.pyplot as plt  
import pandas as pd  
  
import seaborn as sns  
  
from sklearn.model_selection import train_test_split  
from sklearn.metrics import root_mean_squared_error  
from sklearn.linear_model import LinearRegression  
from sklearn.ensemble import RandomForestRegressor  
from sklearn.model_selection import GridSearchCV
```

- ① *Numpy*, *Matplotlib* och *Pandas* är standardbibliotek inom ML och brukar generellt sett alltid laddas in rent slentrianmässigt.
- ② *Seaborn* är ett visualiseringssbibliotek baserat på matplotlib som många gånger kan göra det enklare att skapa mer sofistikerade visualiseringar. Den läsare som vill lära sig om *Seaborn* kan göra så genom att läsa guiderna på bibliotekets hemsida.
- ③ Importer av diverse funktionalitet från *scikit-learn* som vi kommer använda genom kodexempellets gång.

Vi laddar nu in datan och inspekterar den.

```
housing_original = pd.read_csv("housing.csv")
```

④

- ① Datafilen "housing.csv" är i det här fallet i samma mapp som kodfilen. Därför behövs ingen fullständig sökväg och det räcker att skriva `pd.read_csv("housing.csv")`. Hade vi använt en fullständig sökväg hade det kunnat se ut enligt följande:  
`pd.read_csv("C:/Users/antonio/Downloads/testing_antonio/housing.csv")`  
Notera, du behöver alltså då ändra sökvägen till motsvarigheten på din dator.

Genom att använda `.info()` metoden får vi en koncis summering av datan.

```
housing_original.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 10 columns):
 #   Column           Non-Null Count  Dtype  
--- 
 0   longitude        20640 non-null   float64 
 1   latitude         20640 non-null   float64 
 2   housing_median_age 20640 non-null   float64 
 3   total_rooms      20640 non-null   float64 
 4   total_bedrooms   20433 non-null   float64 
 5   population       20640 non-null   float64 
 6   households       20640 non-null   float64 
 7   median_income    20640 non-null   float64 
 8   median_house_value 20640 non-null   float64 
 9   ocean_proximity  20640 non-null   object  
dtypes: float64(9), object(1)
```

```
memory usage: 1.6+ MB
```

Vi ser från resultatet att vi har 20640 observationer/rader och 10 stycken variabler/kolumner. Samtliga kolumner förutom `ocean_proximity` är av datatypen `float64`. Kolumnen `ocean_proximity` har data-typen `object` och som vi kommer se är det textsträngar i den kolumnen. Vi ser även att kolumnen `total_bedrooms` har 20433 *non-null* värden innehållande att det finns  $20640 - 20433 = 207$  saknade värden. Detta kommer behöva hanteras, antingen genom att ta bort raderna där värden saknas eller fylla i enskilda värden där det saknas värden. För att få en känsla för hur datan ser ut är det bra att skriva ut några rader. Det gör vi härnäst. Notera, vi gör två separata utskrifter eftersom samtliga kolumner av utrymmesskäl inte får plats på bokens sida.

```
housing_original.iloc[:, :6].head()
```

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population
0	-122.23	37.88	41.0	880.0	129.0	322.0
1	-122.22	37.86	21.0	7099.0	1106.0	2401.0
2	-122.24	37.85	52.0	1467.0	190.0	496.0
3	-122.25	37.85	52.0	1274.0	235.0	558.0
4	-122.25	37.85	52.0	1627.0	280.0	565.0

```
housing_original.iloc[:, 5:].head()
```

	population	households	median_income	median_house_value	ocean_proximity
0	322.0	126.0	8.3252	452600.0	NEAR BAY
1	2401.0	1138.0	8.3014	358500.0	NEAR BAY
2	496.0	177.0	7.2574	352100.0	NEAR BAY
3	558.0	219.0	5.6431	341300.0	NEAR BAY
4	565.0	259.0	3.8462	342200.0	NEAR BAY

Från de fem första raderna i datan ovan är samtliga värden för kolumnen `ocean_proximity` = NEAR BAY. För att se om det finns fler kategorier och vilka dessa i sådana fall är kan vi använda metoden `value_counts()`.

```
housing_original['ocean_proximity'].value_counts()
```

```
ocean_proximity
<1H OCEAN      9136
INLAND        6551
NEAR OCEAN     2658
NEAR BAY       2290
ISLAND          5
Name: count, dtype: int64
```

Vi ser att kategorin ISLAND endast har fem stycken observationer. För att göra våra analyser mer koncisa kommer vi ta bort de rader som har värdet ISLAND för kolumnen ocean\_proximity.

```
housing = housing_original[housing_original['ocean_proximity'] !=  
    ↪ 'ISLAND']  
housing['ocean_proximity'].value_counts()
```

- ① Vår originaldata sparade vi i variabeln `housing_original`. Nu när vi tar bort vissa rader vill vi rent principiellt inte göra det på originaldatan. Därför sparar vi datan där dessa rader är raderade i en ny variabel som vi kallar för `housing` i detta fall.
- ② Detta steg görs för att vi ska kunna verifiera att det inte finns några rader med värdet ISLAND kvar.

```
ocean_proximity
<1H OCEAN      9136
INLAND        6551
NEAR OCEAN     2658
NEAR BAY       2290
Name: count, dtype: int64
```

Vi vet från Avsnitt 1.3.6 att kategorisk data generellt sett behöver omvandlas för att kunna användas i maskininlärningsmodeller. Genom att använda metoden `.get_dummies()` genomför vi *one-hot-encoding* och får fyra dummyvariabler eftersom vi har de fyra kategorierna <1H Ocean, INLAND, NEAR OCEAN och NEAR BAY.

```
housing = pd.get_dummies(housing, columns = ['ocean_proximity'],  
    ↪ dtype = int, prefix = 'dmy')
```

- ① Vi använder prefixet "dmy" för de dummyvariabler vi skapar. I nästa kodblock ser vi hur dummyvariabel-kolumnerna får namn på formen `dmy_xxx`. Exempelvis `dmy_-<1H OCEAN`.

```
housing.iloc[[1, 200, 1000, 1850, 5000], 9:]
```

①

- ① Vi väljer ut specifika rader för att även demonstrera det faktum att radsumman av alla dummyvariabler alltid är 1. Anledningen är att endast en dummyvariabel kan vara "uppfylld" i taget och därmed vara 1 medan resterande dummyvariabler alltså är 0.

	dmy_-<1H OCEAN	dmy_INLAND	dmy_NEAR BAY	dmy_NEAR OCEAN
1	0	0	1	0
200	0	0	1	0
1000	0	1	0	0
1850	0	0	0	1
5000	1	0	0	0

Härnäst skapar vi träningsdata, valideringsdata och testdata. Vi kommer även skapa ett dataset som vi kallar för `train_full` som består av träningsdatan och valideringsdatan kombinerad.

- På träningsdatan kommer vi träna olika modeller.
- På valideringsdatan kommer vi utvärdera våra tränade modeller och eventuellt utse en vinnare om resultatet är bra nog.
- Om vi utsett en vinnarmodell så ska vi träna om den modellen på träningsdatan och valideringsdatan ihopslagen. Att göra denna ihopslagningen kan bli praktiskt bökigt rent kodmässigt och därför kommer vi redan nu förbereda det genom skapandet av `train_full` datan.
- Testdatan "stoppar vi undan" och ska inte använda förrän på slutet för att slutgiltigt utvärdera en modells generaliseringsförmåga på ny osedd data. Annars överanpassar vi endast våra modeller till testdatan och det är inte syftet.

```
train_full, test = train_test_split(housing, test_size=0.2,
                                   random_state=40)
train, val = train_test_split(train_full, test_size=0.25,
                             random_state=36)
```

①

- ① När vi delar upp vår data finns det en slumpmässighet i hur det sker. För att få samma resultat varje gång koden körs så kan vi specificera ett värde på hyperparametern `random_state`. Vi har godtyckligt valt värdet 40, vi hade likväld kunnat välja värdet 45 eller något annat, även om det troligtvis hade lett till andra resultat. Syftet är dock att resultaten blir samma, oavsett vad de blir, när vi kör om koden.

Härnäst fortsätter vi med att genomföra en *exploratory data analysis (EDA)* och bearbeta datan. Vi gör alltså steg tre och steg fyra från checklistan. Notera, nu ska vi endast använda träningsdatan eftersom vi inte vill få insikter om eller påverka den data som vi slutligen ska utvärdera och testa våra modeller på.

### 3-4. EDA och databearbetning

Vi börjar med att skapa oss en överblick över vår träningsdata genom att använda `.info()` metoden.

```
train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 12381 entries, 12054 to 14298
Data columns (total 13 columns):
 #   Column           Non-Null Count  Dtype  
--- 
 0   longitude        12381 non-null   float64 
 1   latitude         12381 non-null   float64 
 2   housing_median_age 12381 non-null   float64 
 3   total_rooms      12381 non-null   float64 
 4   total_bedrooms   12261 non-null   float64 
 5   population       12381 non-null   float64 
 6   households       12381 non-null   float64 
 7   median_income    12381 non-null   float64 
 8   median_house_value 12381 non-null   float64 
 9   dmy_<1H OCEAN    12381 non-null   int64  
 10  dmy_INLAND       12381 non-null   int64  
 11  dmy_NEAR BAY    12381 non-null   int64  
 12  dmy_NEAR OCEAN   12381 non-null   int64  
dtypes: float64(9), int64(4)
memory usage: 1.3 MB
```

Vi ser från resultatet att det saknas värden för kolumnen `total_bedroom`. Vi väljer här att ta bort de rader där det saknas värden.

```
train = train.dropna()  
train.info()
```

①

- ① Vi kör om `info` metoden för att verifiera att raderna där det saknades värden har tagits bort.

```
<class 'pandas.core.frame.DataFrame'>  
Index: 12261 entries, 12054 to 14298  
Data columns (total 13 columns):  
 #   Column           Non-Null Count  Dtype     
 ---  --     
 0   longitude        12261 non-null   float64  
 1   latitude         12261 non-null   float64  
 2   housing_median_age 12261 non-null   float64  
 3   total_rooms      12261 non-null   float64  
 4   total_bedrooms   12261 non-null   float64  
 5   population       12261 non-null   float64  
 6   households       12261 non-null   float64  
 7   median_income    12261 non-null   float64  
 8   median_house_value 12261 non-null   float64  
 9   dmy_<1H OCEAN    12261 non-null   int64  
 10  dmy_INLAND       12261 non-null   int64  
 11  dmy_NEAR BAY    12261 non-null   int64  
 12  dmy_NEAR OCEAN   12261 non-null   int64  
dtypes: float64(9), int64(4)  
memory usage: 1.3 MB
```

Vi gör samma sak för valideringsdatan och testdatan.

```
val = val.dropna()  
test = test.dropna()
```

## i Ersätta saknade värden

Vi valde ovan att ta bort raderna där det saknas värden för kolumnen `total_bedrooms`. Vi hade till exempel även kunnat fylla i de saknade värdena med medelvärdet eller medianvärdet för kolumnen. Då kan vi använda oss av `SimpleImputer` i *scikit-learn*. Se nedan där detta demonstreras i kod.

```
from sklearn.impute import SimpleImputer

imputer_demo_df = pd.DataFrame({
    'A': [7, 4, 10],
    'B': [2, np.nan, 5],
    'C': [3, 6, 9]
})①

print(imputer_demo_df)

imputer = SimpleImputer(strategy='mean')②
df_imputed = imputer.fit_transform(imputer_demo_df)

print(df_imputed)

print("Before imputation, type:", type(imputer_demo_df))
print("After imputation, type:", type(df_imputed))③
```

	A	B	C
0	7	2.0	3
1	4	NaN	6
2	10	5.0	9
	[ 7. 2. 3.]	[ 4. 3.5 6.]	[10. 5. 9.]

Before imputation, type: <class 'pandas.core.frame.DataFrame'>  
After imputation, type: <class 'numpy.ndarray'>

1. Vi skapar en *dataframe* som används för demonstration. Notera, det saknas ett värde i den andra kolumnen.
2. Vi specificerar att vi använder medelvärdet, `mean`, för att ersätta det saknade värde.

det. I vårt fall har vi värdena 2 och 5, medelvärdet av det är  $(2+5)/2 = 3.5$  vilket är det värde som kommer fyllas i när vi i nästa rad exekverar koden `imputer.fit_transform(imputer_demo_df)`. Vi hade även kunnat använda exempelvis medianen genom att specificera värdet `median` för hyperparametern `strategy`.

3. Notera, efter att vi använt `SimpleImputer` så transformeras vår *pandas dataframe* till en *numpy ndarray*. Det är bra att känna till så vi inte blir överraskade.

Med koden nedan genomför vi en visualisering som innehåller mycket information. Vi ser en geografisk bild över Kalifornien, populationsstorlekarna där större populationer representeras med större punkter samt priserna som representeras med olika färger. Notera att denna typ av komplexa visualiseringar inte är något som vi generellt sett ”enkelt gör” utan det kräver ofta att vi googlat eller sett någon annan göra dem för att vi ska få inspiration. Från visualiseringen ser vi ett tydligt mönster, hus nära havet tenderar att vara dyrare.

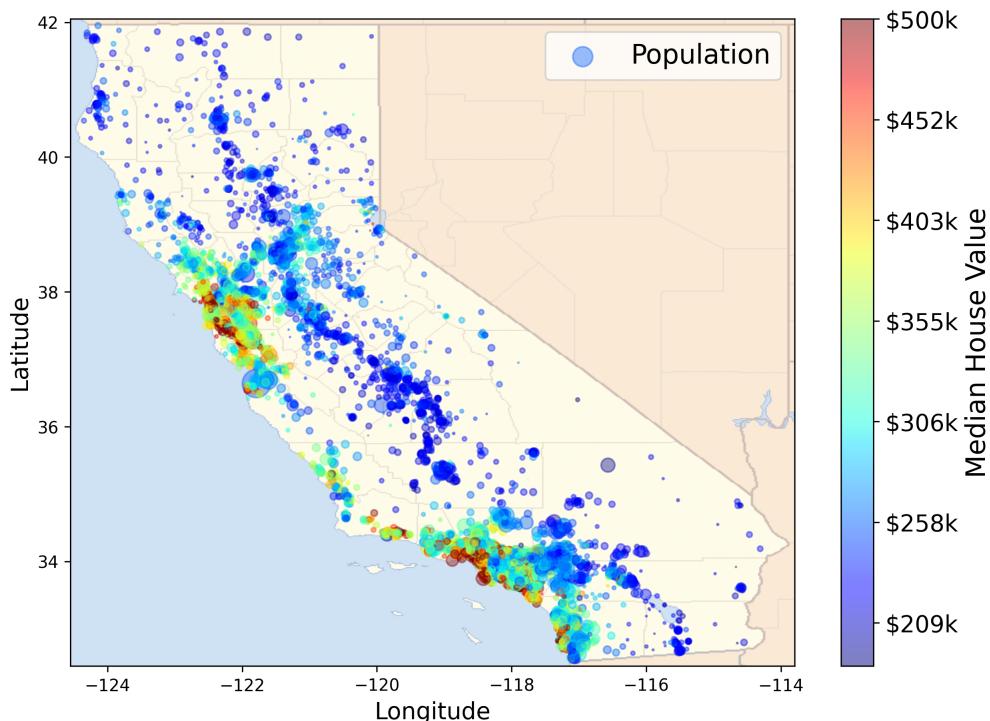
```
import matplotlib.image as mpimg
california_img=mpimg.imread('california.png')①
ax = train.plot(kind="scatter", x="longitude", y="latitude",
                 figsize=(10,7),
                 s=train['population']/100, label="Population",
                 c="median_house_value",
                 cmap=plt.get_cmap("jet"),
                 colorbar=False, alpha=0.4)
plt.imshow(california_img, extent=[-124.55, -113.80, 32.45,
                                    42.05], alpha=0.5,
                 cmap=plt.get_cmap("jet"))
plt.ylabel("Latitude", fontsize=14)
plt.xlabel("Longitude", fontsize=14)

prices = train["median_house_value"]
tick_values = np.linspace(prices.min(), prices.max(), 11)
cbar = plt.colorbar(ticks=tick_values/prices.max())
cbar.ax.set_yticklabels(["${:.0f}M".format(v) for v in
                           tick_values], fontsize=14)
cbar.set_label('Median House Value', fontsize=16)

plt.legend(fontsize=16)
```

① För att visualisera kartan över Kalifornien använder vi oss av bilden “california.png”

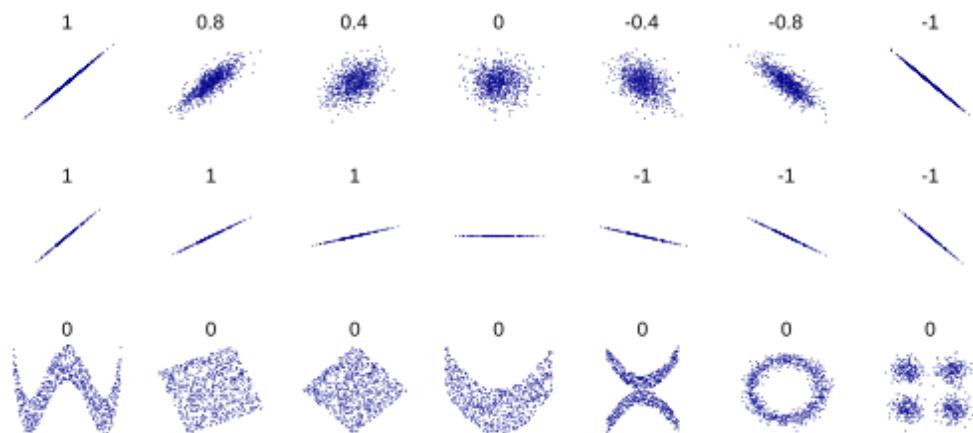
som finns på bokens hemsida.



När vi senare ska skapa modeller behöver vi välja vilka kolumner vi ska inkludera i modellen. Ett sätt att göra detta på är att kolla på korrelationen mellan den beroende variabeln, `median_house_value` i detta fallet, samt de oberoende variablerna. Vi kommer dock när vi kommer till själva modelleringen inkludera alla variabler för att begränsa kodexempletts omfattning.

## i Korrelation

Korrelationskoefficienten eller korrelationen mellan två variabler mäter graden av linjärt samband och befinner sig mellan  $-1$  och  $+1$ . Ett perfekt positivt linjärt samband har korrelationskoefficienten  $+1$  medan ett perfekt negativt linjärt samband har korrelationskoefficienten  $-1$ . Korrelationskoefficienten mellan en variabel och sig själv är alltid  $+1$ . I de fall korrelationskoefficienten är  $0$  finns det inget linjärt samband. Däremot kan det finnas andra samband som är icke linjära. Notera att korrelationen endast mäter graden av linjärt samband. Vi ska inte glömma att det kan finnas andra samband som är icke-linjära. Se Figur 2.1.



Figur 2.1: En figur som visar olika samband och beräknade korrelationskoefficienter. I den mittersta raden ser vi att korrelationskoefficienten inte säger något om lutningen. I den nedersta raden ser vi att det finns väldigt tydliga och starka samband men att korrelationskoefficienten fortfarande är 0. Anledningen är att det är icke-linjära samband och korrelationskoefficienten mäter endast graden av linjära samband.

```
corr_matrix = train.corr()
corr_matrix["median_house_value"].sort_values(ascending=False)
```

```
median_house_value      1.000000
```

```

median_income          0.689648
dmy_<1H OCEAN        0.253893
dmy_NEAR BAY         0.159601
dmy_NEAR OCEAN       0.148101
total_rooms           0.132945
housing_median_age   0.099810
households            0.063797
total_bedrooms        0.048099
population            -0.025675
longitude             -0.043444
latitude              -0.147407
dmy_INLAND            -0.485680
Name: median_house_value, dtype: float64

```

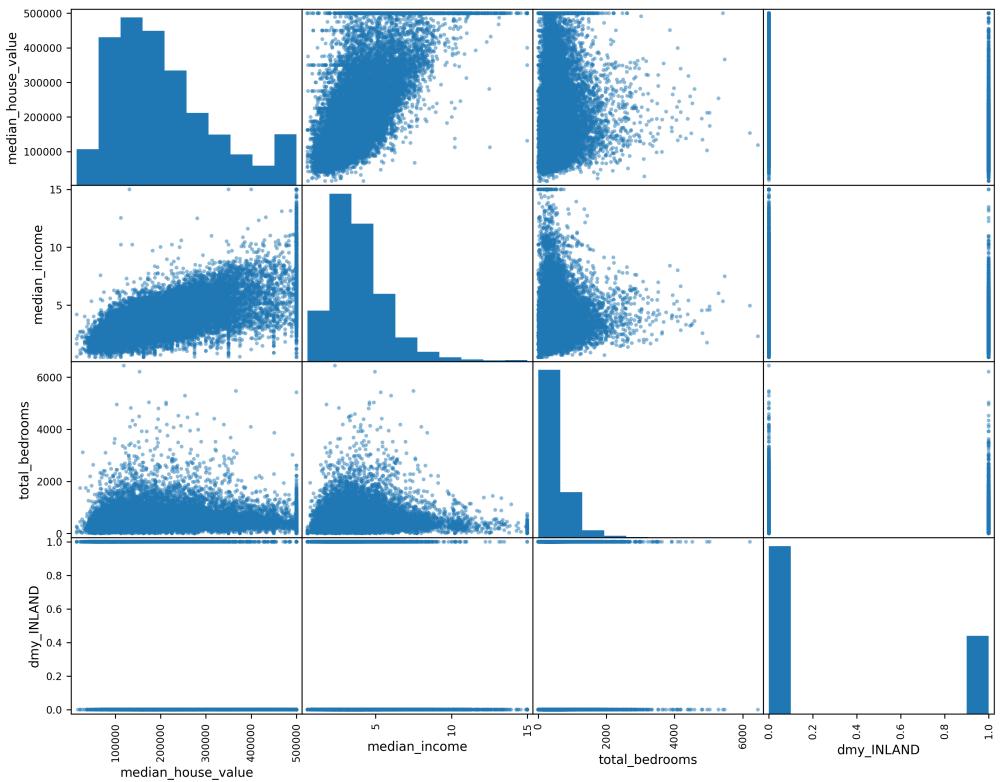
I tabellen ovan har vi beräknat korrelationskoefficienten mellan `median_house_value` och övriga kolumner. Föga förvånande ser vi exempelvis att korrelationskoefficienten mellan `median_house_value` och `median_income` är positiv och relativt nära +1. Det är rimligt att ökad inkomst leder till dyrare hus i ett område och att sambandet är relativt starkt. Hade vi skapat en modell som endast inkluderar två variabler hade vi kunnat prova att använda variablerna `median_income` och `dmy_INLAND` då de har relativt stark korrelation med den beroende variabeln. Notera, tecknet, +/--, spelar alltså ingen roll, det vi är ute efter är alltså att det finns ett samband och riktningen spelar mindre roll. Som nämnts tidigare kommer vi dock använda alla variabler när vi kommer till modelleringen för att begränsa kodexempletts omfattning.

Nedan gör vi några visualiseringar för att få en grafisk representation vilket ofta kan ge en bättre överblick än vad endast siffror kan göra.

```

attributes = ["median_house_value", "median_income",
    "total_bedrooms", "dmy_INLAND"]
pd.plotting.scatter_matrix(housing[attributes], figsize=(13, 10))
plt.show()

```

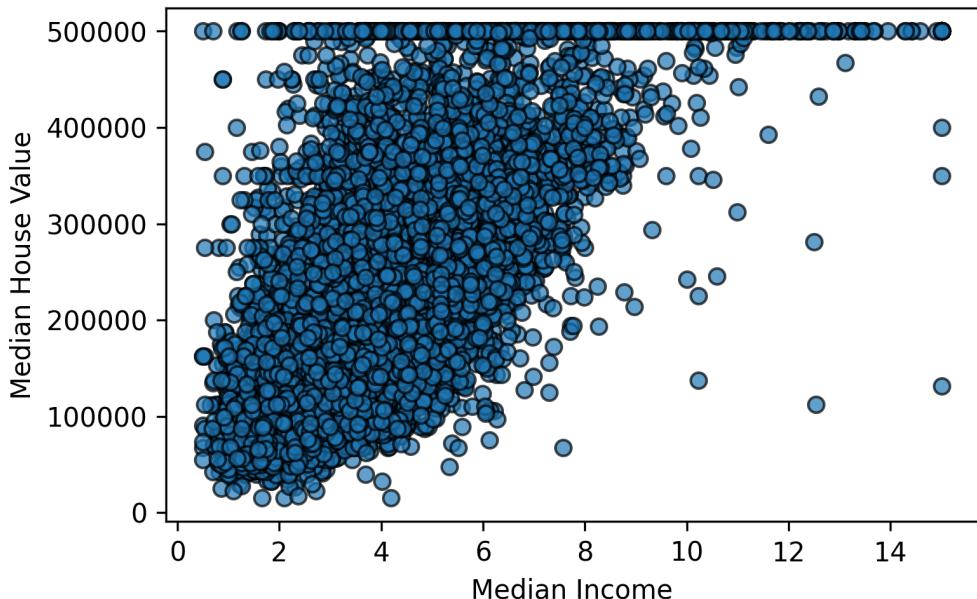


Är vi nyfikna på sambandet mellan inkomst och medianvärden på husen så kan vi göra en separat visualisering enligt nedan för att få en ännu tydligare bild. Från figuren ser vi tydligt att det är ett positivt samband mellan inkomst och husvärde. Men vi ser också något mer som är intressant. Medianhuspriserna verkar ha en övre gräns på 500000, innebärande att om priset är över det så sätts det till 500000. Det kan vara ett medvetet val av de som tillhandahållit dataen, exempelvis för att om några distrikter har extremt höga priser kan de distrikten, som inte är representativa, påverka hela analysen på ett substantiellt sätt vilket kanske inte är önskvärt. Det kan också vara ett fel som skett i data-inmatningen. Vi vet egentligen inte. Oavsett behöver vi som modellerare ta ställning till vad som ska göras. Vi väljer här för enkelhetens skull att låta de datapunkterna vara som de är.

```

plt.scatter(train['median_income'], train['median_house_value'],
           alpha=0.7, edgecolor='k')
plt.xlabel('Median Income')
plt.ylabel('Median House Value')
plt.show()

```



Nedan skapar vi lådagram för att analysera hur våra dummyvariabler påverkar vår beroende variabel som är `median_house_value`. Exempelvis ser vi att om ett distrikt befinner sig på `INLAND`, vilket ses genom att dummyvariabeln `dmy_INLAND = 1`, så verkar värdet på husen i distriktet vara mycket mindre. Det är en intressant observation som är rimlig då vi från erfarenhet vet att hus nära havet, generellt sett, är dyrare.

```

fig, axes = plt.subplots(2, 2, figsize=(12, 10))

sns.boxplot(data=train, x='dmy_<1H OCEAN', y='median_house_value',
             ax=axes[0, 0])

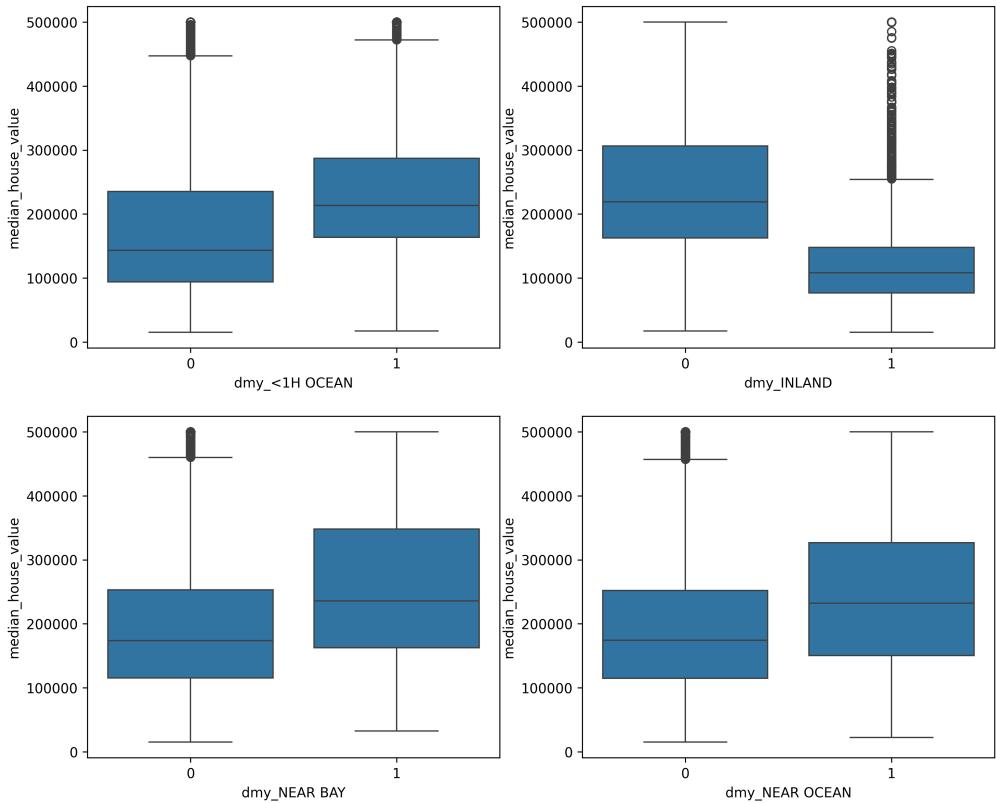
```

```

sns.boxplot(data=train, x='dmy_INLAND', y='median_house_value',
            ax=axes[0, 1])
sns.boxplot(data=train, x='dmy_NEAR BAY', y='median_house_value',
            ax=axes[1, 0])
sns.boxplot(data=train, x='dmy_NEAR OCEAN',
            y='median_house_value', ax=axes[1, 1])

```

- ① Här använder vi *seaborn* biblioteket som vi importerat med aliaset “*sns*”.



Efter att ha utforskat datan och gjort lite bearbetningar, exempelvis tagit bort rader där det saknas data, så fortsätter vi nu med ML-modelleringen som är steg fem i vår checklista.

## 5. ML-modellering

Det första vi kommer göra nu är att dela in datan i  $X$  och  $y$  där det som vanligt gäller att  $y$  betecknar den beroende variabeln och  $X$  betecknar de oberoende variablerna. Att vi delar in vår data i  $X$  och  $y$  först nu beror på att många av de saker vi genomfört tidigare kan göras enklare om all data är samlad i en *pandas dataframe*. Men när vi gör själva modelleringen kan det underlätta om datan är uppdelad i  $X$  och  $y$ . Detta är dock ytterst en subjektiv preferens och det går i vanlig ordning att göra på olika sätt. Viktigast är att vi vet vad vi gör.

```
X_train_full = train_full.drop(columns=['median_house_value'])      ①
y_train_full = train_full['median_house_value']                      ②

X_train, y_train = train.drop(columns=['median_house_value']),
    ↵  train['median_house_value']
X_val, y_val = val.drop(columns=['median_house_value']),
    ↵  val['median_house_value']
X_test, y_test = test.drop(columns=['median_house_value']),
    ↵  test['median_house_value']
```

- ① Vårt  $X$  innehåller alla kolumner förutom `median_house_value` som är vår beroende variabel. Vi ”*droppar*” därför den kolumnen när vi skapar vårt  $X$ .
- ② Vår beroende variabel  $y$  innehåller endast kolumnen `median_house_value`.

Det är ofta en god idé att säkerställa att den data vi har är korrekt. Exempelvis kan vi kolla så att dimensionerna är korrekta.

```
print(X_train.shape)
print(y_train.shape)

(12261, 12)
(12261,)
```

Vi ser från resultatet ovan att vi för båda  $X$  och  $y$  datan har 12261 rader men att  $X$  också har 12 kolumner eftersom det är tolv olika variabler. Det är alltså precis som det ska vara.

### **i** Stort $X$ eller litet $x$ ?

Enligt konvention brukar vektorer betecknas med små bokstäver (gemener) och matriser med stora bokstäver (versaler). Eftersom vi har flera oberoende variabler så blir det en matrisform på  $X$  och därmed en stor bokstav som används. Hade vi endast haft en oberoende variabel så blir det en vektorform på  $x$  och därmed en liten bokstav som används.

Det är också vanligt förekommande att man använder ett stort  $X$  även om det endast finns en oberoende variabel.

Vi fortsätter nu med att skapa två olika ML-modeller, en linjär regressionsmodell och en *random forest* modell. Notera, hur dessa modeller rent praktiskt fungerar kommer vi lära oss i nästa kapitel. Vi börjar med den linjära regressionsmodellen.

```
from sklearn.linear_model import LinearRegression          ①  
lin_reg = LinearRegression()                            ②  
lin_reg.fit(X_train, y_train)                          ③
```

- ① Importera `LinearRegression` från *scikit-learn*.
- ② Instantiera en linjär regressionsmodell.
- ③ Träna den instantierade regressionsmodellen.

fit_intercept	True
copy_X	True
tol	1e-06
n_jobs	None
positive	False

Nedan kommer vi träna en *random forest* modell, vi kommer också använda `time` biblioteket för att mäta hur lång tid träningen tar för att demonstrera att det går att göra ifall vi önskar.

```
import time                                              ①  
from sklearn.ensemble import RandomForestRegressor
```

```

start_time = time.time()                                ②

rf = RandomForestRegressor()
hyperparam_grid = {'max_depth': [5, 10, 15, 50], 'n_estimators': ③
    ↵ [1, 5, 10]}                                     ④
grid_search = GridSearchCV(rf, hyperparam_grid,
    ↵ scoring='neg_root_mean_squared_error', cv=5)      ⑤
grid_search.fit(X_train, y_train)                      ⑥

end_time = time.time()

execution_time = end_time - start_time
print(f"GridSearchCV fitting took {execution_time:.4f} seconds.") ⑦

```

- ① Vi importerar `time` biblioteket.
- ② Koden mellan `start_time = time.time()` och `end_time = time.time()` är det som vi mäter tiden på.
- ③ Vi instantierar en *random forest* modell.
- ④ Vi specificerar vilka hyperparametrar vi vill utvärdera i en `GridSearch`.
- ⑤ Vi instantierar vår `GridSearch`. Notera att vi använder `scoring='neg_root_mean_squared_error'` eftersom ”högre är bättre” i *scikit-learn scoring*. Hyperparametern `cv=5` specificerar att vi använder 5-delad korsvalidering.
- ⑥ Vi genomför en *grid search*. Efter att de optimala hyperparametrarna hittats kommer modellen, det vill säga `RandomForestRegressor`, att tränas om med de optimala hyperparametrarna. Anledningen är att i *scikit-learn* är standardvärdet för hyperparametern `refit True`. Se *scikit-learn* dokumentationen för detaljer.
- ⑦ Vi skriver ut resultatet för hur lång tid vår kod tog att exekvera.

`GridSearchCV fitting took 12.5460 seconds.`

```

print("Best Hyperparameters from GridSearchCV:",
    ↵ grid_search.best_params_)                         ①
# pd.DataFrame(grid_search.cv_results_)               ②

```

- ① Vi skriver ut vilka de optimala hyperparametrarna är.
- ② Denna kodraden är utkommenderad då utskriften inte får plats på bokens sida.

Läsaren uppmanas att själv köra koden och inspektera resultatet.

```
Best Hyperparameters from GridSearchCV: {'max_depth': 50,  
'n_estimators': 10}
```

Nu har vi tränat två olika modeller på träningsdatan. Härnäst ska vi se vilken som är bättre genom att utvärdera dem på valideringsdatan.

```
lr_pred_val = lin_reg.predict(X_val)                                ①  
rf_pred_val = grid_search.predict(X_val)                            ②  
  
print('RMSE Linear Regression:', root_mean_squared_error(y_val,  
    ↵ lr_pred_val))                                                 ③  
print('RMSE Random Forest Regression:',  
    ↵ root_mean_squared_error(y_val, rf_pred_val))                  ④
```

- ① Vi predikterar valideringsdatan med vår linjära regressionsmodell. Dessa prediktorer används för att beräkna *RMSE*.
- ② Vi predikterar valideringsdatan med vår **random forest** modell där vi optimerade hyperparametrarna med **GridSearch**. Dessa prediktioner används för att beräkna *RMSE*.
- ③ Beräknar *RMSE* för vår linjära regressionsmodell.
- ④ Beräknar *RMSE* för vår **random forest** modell.

```
RMSE Linear Regression: 69442.09533293563  
RMSE Random Forest Regression: 52277.96578719621
```

Från resultaten ovan ser vi att **random forest** modellen gör mindre fel (lägre *RMSE*) än vad den linjära regressionsmodellen gör. Alltså är **random forest** modellen bättre och den vi kommer gå vidare med. Vi vet alltså att **random forest** modellen är bättre, men är modellen "bra"? För att kunna svara på det behöver vi ha någon form av referenspunkt och det kan vi få genom att kolla på medelvärdet för huspriserna.

```
print(np.mean(y_val))                                              ①  
print(root_mean_squared_error(y_val, rf_pred_val)/np.mean(y_val))  
    ↵                                                               ②
```

- ① Vi beräknar medelvärdet för huspriserna i valideringsdatan.
- ② Prediktionsfelet modellen gör, *RMSE*, är ungefär 25% stort i förhållande till medelvärdet för huspriserna. Är det bra eller dåligt? Det beror på sammanhanget. Vi

hade troligtvis inte varit glada om vi själva sälde ett hus och får cirka 25% lägre pris än vad vi hade kunnat få. Men om vi kanske vill sälja ett hus och vill få en snabb första uppskattning på vad huspriset kan ligga på, kanske det är bra nog? Vi går djupare in på detta i nästa steg från checklistan ”6. Presentera din lösning för intressenter”.

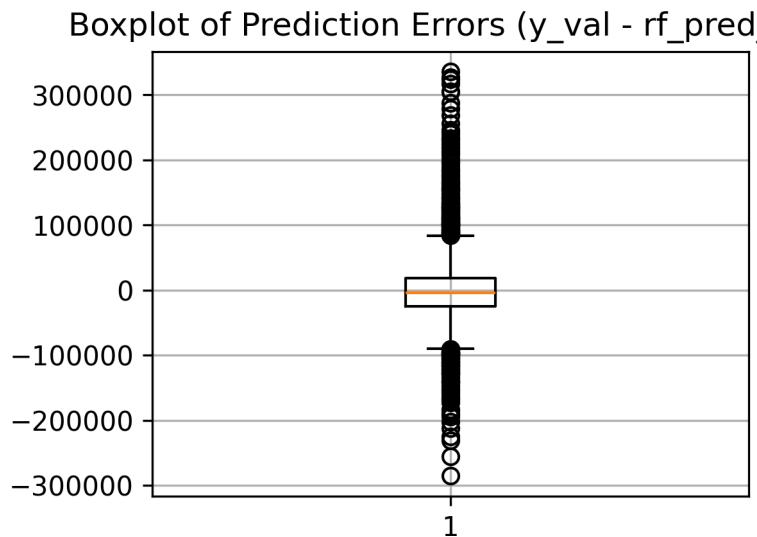
207517.1138589618

0.2519212262306555

Nedan gör vi ett lådagram (boxplot på engelska) för att få en känsla över felens storlek där vi med felen menar  $y\_val - rf\_pred\_val$  som alltså är de sanna värdena från valideringsdatan subtraherat med de predikterade värdena för valideringsdatan. De beräknade differanserna benämns ofta för residualer.

```
errors = y_val - rf_pred_val

plt.figure(figsize=(4, 3))
plt.boxplot(errors)
plt.title('Boxplot of Prediction Errors (y_val - rf_pred_val)')
plt.grid(True)
```



Från visualiseringen ser vi att lådan, som innehåller 50% (mellan den 25:e och 75:e percentilern) av observationerna är relativt liten vilket är bra. Samtidigt ser vi att det sker en hel del fel som är stora vilket inte är bra.

Vi utgår nu ifrån att vi har en modell vi är nöjda med och har tänkt produktionssätta. Då är nästa steg att träna om den valda modellen på träningsdatan och valideringsdatan ihopslagen (kom ihåg att vi tidigare förberedde `train_full` datan så det är förberett) för att sedan utvärdera den på testdatan.

I koden nedan kollar vi på vilka de optimala hyperparametrarna är. Dessa kommer sedan användas när vi tränar om modellen på träningsdatan och valideringsdatan ihopslagen.

```
best_params = grid_search.best_params_
best_params
{'max_depth': 50, 'n_estimators': 10}
```

```
best_rf = RandomForestRegressor(**best_params)           ①
best_rf.fit(X_train_full, y_train_full)                 ②
```

- ① Vi instantierar en modell med de optimala hyperparametrarna. `**best_params` betyder att innehållet i *dictionaryn best\_params* packas upp och skickas som *keyword arguments* till `RandomForestRegressor` modellen.
- ② Modellen tränas på träningsdatan och valideringsdatan ihopslagen.

n_estimators	10
criterion	'squared_error'
max_depth	50
min_samples_split	2
min_samples_leaf	1
min_weight_fraction_leaf	0.0
max_features	1.0
max_leaf_nodes	None
min_impurity_decrease	0.0
bootstrap	True
oob_score	False
n_jobs	None
random_state	None
verbose	0

warm_start	False
ccp_alpha	0.0
max_samples	None
monotonic_cst	None

Med koden nedan verifierar vi att de optimala hyperparametrarna används i modellen. Notera, det finns även andra hyperparametrar och det är eftersom standardvärdena för de användes vilket skedde eftersom vi inte specificerade att några andra skulle användas, då används alltså standardvärdena vilket är en designprincip i *scikit-learn*, vi kommer lära oss mer om detta i Avsnitt 2.4.1.

```
best_rf.get_params()
```

```
{'bootstrap': True,
 'ccp_alpha': 0.0,
 'criterion': 'squared_error',
 'max_depth': 50,
 'max_features': 1.0,
 'max_leaf_nodes': None,
 'max_samples': None,
 'min_impurity_decrease': 0.0,
 'min_samples_leaf': 1,
 'min_samples_split': 2,
 'min_weight_fraction_leaf': 0.0,
 'monotonic_cst': None,
 'n_estimators': 10,
 'n_jobs': None,
 'oob_score': False,
 'random_state': None,
 'verbose': 0,
 'warm_start': False}
```

Slutligen så utvärderar vi modellens prestanda på testdatan. Vi ser att vi får liknande resultat som för valideringsdatan vilket är ett tecken på att modellen generaliseras bra på ny osedd data. Det är bra.

```

rf_pred_test = best_rf.predict(X_test)
rmse_test = root_mean_squared_error(y_test, rf_pred_test)
print('RMSE Random Forest on Test data:', rmse_test)
print(rmse_test/np.mean(y_test))

```

RMSE Random Forest on Test data: 50877.971548985646  
0.2460790722693711

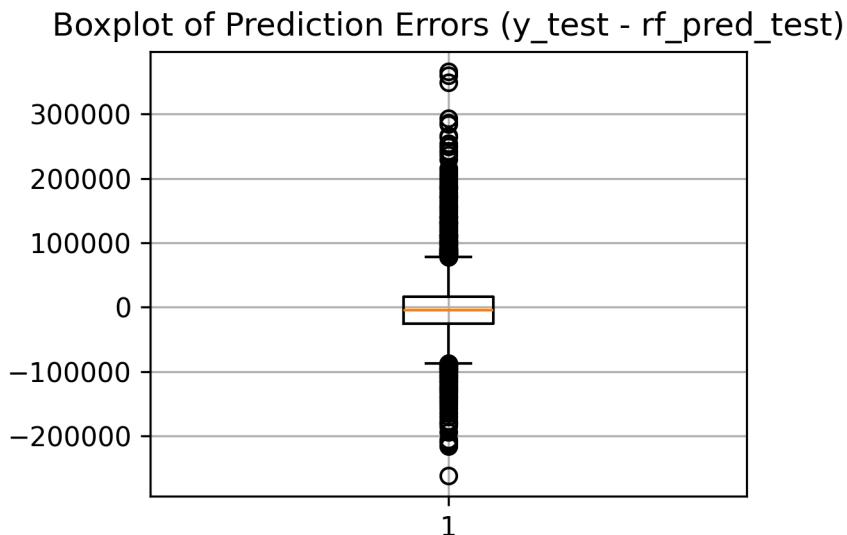
Nedan gör vi också ett lådagram där vi kan analysera felens storlek.

```

errors = y_test - rf_pred_test

plt.figure(figsize=(4, 3))
plt.boxplot(errors)
plt.title('Boxplot of Prediction Errors (y_test - rf_pred_test)')
plt.grid(True)

```



Vi antar att vi är nöjda med vår modell. Då kan modellen tränas om på hela datasetet och därefter sparas så den inte behöver tränas om varje gång den ska användas. Vi gör det i koden nedan.

```

X_full = housing.drop(columns=['median_house_value'])
y_full = housing['median_house_value']

saved_model = RandomForestRegressor(**best_params)
saved_model.fit(X_full, y_full)

joblib.dump(saved_model, 'rf_saved_model.pkl')

```

```
['rf_saved_model.pkl']
```

När vi nu har en sluttgiltig modell kan den börja användas. I koden nedan kan vi tänka oss att en användare är intresserad av att prediktera priset för två områden där det ena till exempel har `longitude = -118.3, latitude = 34.2, housing_median_age = 35.0` och så vidare medan det andra har `longitude = -117.85, latitude = 33.9, housing_median_age = 20.0` och så vidare.

```

new_districts = pd.DataFrame({
    'longitude': [-118.30, -117.85],
    'latitude': [34.20, 33.90],
    'housing_median_age': [35.0, 20.0],
    'total_rooms': [880.0, 1200.0],
    'total_bedrooms': [200.0, 300.0],
    'population': [500.0, 750.0],
    'households': [220.0, 280.0],
    'median_income': [4.2, 5.1],
    'dmy_<1H OCEAN': [0, 1],
    'dmy_INLAND': [1, 0],
    'dmy_NEAR BAY': [0, 0],
    'dmy_NEAR OCEAN': [0, 0]
})

```

```
predicted_values = saved_model.predict(new_districts)
```

```

for i, value in enumerate(predicted_values, start=1):
    print(f"Predicted median house value for district {i}:
        ${value:.2f}")

```

① Data för två distrikter vi vill prediktera har matats in i en *pandas dataframe*.

② Vi predikterar `median_house_value` för de två distrikten.

- ③ Vi skriver ut predikterade värden för de två distrikten.

```
Predicted median house value for district 1: $266,810.00  
Predicted median house value for district 2: $204,910.00
```

## 6. Presentera din lösning för intressenter

Nu har vi kommit till steget att vi ska presentera vårt arbete för intressenter. Beroende på vem vi presenterar för så anpassar vi presentationen. Vi bör tänka igenom vilka budskap vi vill förmedla och hur det görs på ett effektivt sätt.

Vi kommer inte göra en komplett presentation här utan kommer istället diskutera om modellen går att använda. När vi utvärderade modellen på testdatan fick vi en *RMSE* på cirka 50000 vilket är ungefär ett 25% fel i förhållande till medelvärdet på testdatan. Är det bra nog? Går det att använda?

Om vi föreställer oss att vi är en mäklarbyrå så vore de säljande kunderna nog inte så nöjda om vi sålde ett hus för billigt. På motsvarande sätt vore de köpande kunderna nog inte så nöjda om vi sålde ett hus för dyrt. Sammantaget så är felet modellen gör troligtvis för stort för att kunna användas till att sätta utgångspris på hus. Men, modellen kanske hade kunnat användas för att snabbt få ett pris och att varje mäklare sedan manuellt justerar priset så det blir mer anpassat utifrån marknadspriserna? Genom kreativt tänkande har vi alltså hittat en möjlig användning för modellen som hade kunnat leda till effektivare och snabbare processer för att prissätta hus. Vi kan även tänka oss att modellen hade kunnat användas på en hemsida där potentiella kunder kan mata in information om ett hus och få ett uppskattat huspris där kunderna informeras om att priset endast ska tolkas som en grov uppskattning och att de kommer bli kontaktade av en mäklare för att få en mer precis prissättning. På så sätt kan kunderna få en snabb uppskattning samtidigt som mäklarbyrån kan få nya potentiella kunder.

Generellt sett behöver vi alltid fråga oss om en modell är bra nog för att kunna användas. Vi kommer inom maskininlärning aldrig få perfekta resultat och vi behöver därför fundera kring om modellen kan skapa nytta och i sådana fall hur.

Sammanfattningsvis har vi på kort tid och utan att egentligen ha arbetat så iterativt, som vi hade gjort i verkligheten, skapat en modell som ger ett resultat som indikerar att vi kan ta fram något som kan skapa värde.

**i Det är inte säkert att det ens är möjligt att uppnå resultat som är bra nog**

När vi skapar modeller kan vi alltid försöka förbättra resultaten genom att arbeta antingen med datan eller själva modellerna. I verkligheten vet vi dock inte om det faktiskt är möjligt att få resultat som uppfyller de krav vi har givet den datan vi har. Exempelvis, om vi vill prediktera en persons lön så vore det väldigt svårt att göra det på ett bra sätt om vi saknar information om ålder, utbildningsnivå och arbetslivserfarenhet. Datan är alltså central.

Trots att vi har all data vi önskar så är det heller inte säkert att vi kan uppnå de resultat vi önskar eftersom datan påverkas för mycket av slumpen snarare än faktiska mönster som kan upptäckas av ML-modeller. Detta är exempelvis fallet med aktiepriser där det diskuteras huruvida det faktiskt är möjligt att ”slå marknaden”. Den intresserade läsaren kan läsa mer om *efficient market hypothesis*, en teori som legat till grund för Sveriges Riksbanks pris i ekonomisk vetenskap till Alfred Nobels minne år 2013 (i vardagligt tal kallat ”nobelpriset i ekonomi”).

## 7. Produktionssättning av modellen och övervakning av implementeringen

Om vi antar att modellen vi skapat ska börja användas så ska den sättas i produktion och övervakas.

Produktionssättningen hade kunnat ske på olika sätt beroende på hur modellen ska användas. Exempelvis hade modellen kunnat implementeras i en app eller hemsida så att kunder kan använda den för att exempelvis få en prisuppskattning för ett hus. Den intresserade läsaren kan ganska enkelt göra det genom att använda ett bibliotek som heter *Streamlit*. Se dokumentationen för detaljer <https://streamlit.io/> .

Modellen hade även behövt övervakas så att de som använder den inte plötsligt börjar få märkliga resultat. Detta kan ske, exempelvis för att det ekonomiska läget ändras vilket gör att mönstren på bostadsmarknaden ändras. Vi hade även kunnat sätta upp riktlinjer för om/när vi vill träna om modellen på ny data. Det hade till exempel kunnat ske kvartalvis eller när vi märker att modellen börjar prestera sämre på grund av att datan modellen bygger på börjar bli inaktuell.

## 2.3 Utmaningar inom ML

I detta avsnitt kommer vi nämna några vanligt förekommande utmaningar som kan uppstå när vi arbetar med ML-projekt.

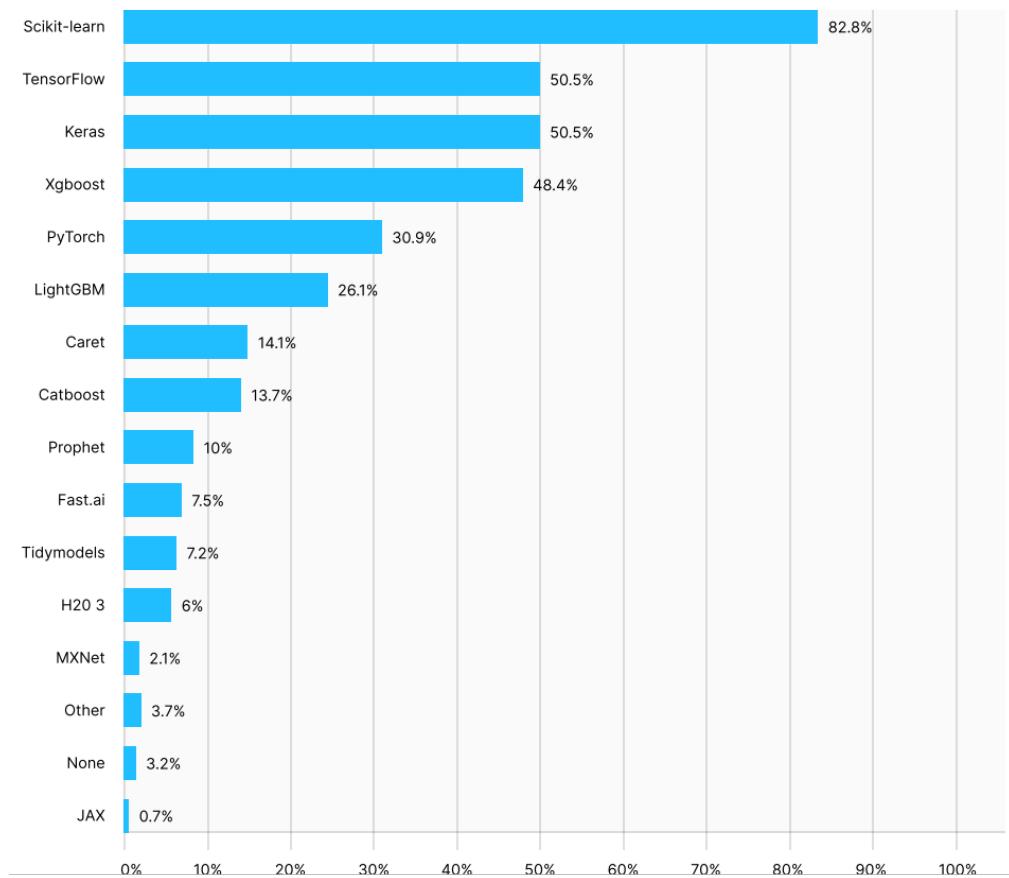
- För lite data. Många tillämpningsområden inom maskininlärning kräver tusentals observationer för att en modell ska kunna tränas till att prestera tillräckligt bra. I vissa fall, exempelvis inom bildanalys, kan det krävas miljontals bilder för att modellen ska bli praktiskt användbar. Oavsett hur skickliga modellerare vi är går det inte att uppnå goda resultat utan tillräckligt mycket data.
- Icke-representativ data. Datan vi tränar våra modeller på behöver generellt sett vara *representativ* för den datan vi senare ska prediktera om modellen ska kunna prestera bra.
- Dålig data kvalitet. Om det finns fel i datan exempelvis för att den blivit slarvigt insamlad eller felaktigt inmatad så kommer ML-modellerna få det svårt att lära sig de underliggande mönstren. Det är exempelvis inte ovanligt att det saknas data, något som då måste hanteras. Exempelvis kanske vi väljer att ta bort observationerna helt eller fylla i saknade värden med exempelvis medelvärdet. Generellt sett kan vi träna en modell där datan blivit bearbetad på ett sätt och träna en annan modell där datan blivit bearbetad på ett annat sätt. Därefter kan vi genom att utvärdera modellerna på valideringsdatan se vad som verkar bäst.
- Irrelevanta *features*. Det finns ett uttryck som säger “*shit in-shit out*”. Har vi inte bra *features* så kommer det generellt sett inte gå att skapa bra ML-modeller.
- Överanpassning och underanpassning. När vi skapar ML-modeller finns det alltid en risk att vi överanpassar eller underanpassar modellerna.

Förhoppningsvis framgår det tydligt att datan är av central betydelse inom ML-projekt. I praktiken är arbete med data något som kan vara utmanande och ta mycket tid.

## 2.4 *Scikit-learn*

*Scikit-learn* är ett *open source* maskininlärningsbibliotek för Python. Biblioteket har bland annat olika algoritmer för regression, klassificering, klustering, modellval och data-bearbetning. Den första publika lanseringen skedde år 2010 och det är idag ett mycket populärt och använt bibliotek inom ML, både inom näringslivet och akademiska sammanhang. I Figur 2.2 ser vi en undersökning som visar vilka ML-bibliotek som används mest bland yrkesverksamma inom *data-science* branschen där *scikit-learn* alltså var det mest populära. Namnet *scikit-learn* kommer från att projektet var ämnat att vara ett “*scientific toolkit for machine learning*”.

#### MACHINE LEARNING FRAMEWORK USAGE



Figur 2.2: Undersökning som visar vilka ML-bibliotek som används mest bland yrkesverksamma inom data science branschen år 2020. Statistiken är tagen från <https://www.kaggle.com/c/kaggle-survey-2020>.

Den läsare som vill fördjupa sig i hur *scikit-learn* är uppbyggt kan läsa nedanstående två artiklar. Detta avsnittet om *scikit-learn* har också till stor del baserats på de två artiklarna.

- Scikit-learn: Machine Learning in Python, Pedregosa et al., JMLR 12, pp. 2825–2830, 2011.
- API design for machine learning software: experiences from the scikit-learn project, Buitinck et al., 2013.

### 2.4.1 Designprinciper

*Scikit-learn* är designat för att vara enkelt, effektivt och tillgängligt även för icke-expporter. Några viktiga design-principer som biblioteket följer är det som vi benämner för konsistens, inspektion, begränsning av klasser och rimliga standardvärden för hyperparametrarna. Vi kollar på var och en av dessa designprinciper härnäst.

- Konsistens. Alla objekt ska ha ett konsistent gränssnitt och ett begränsat antal metoder (metoder är helt enkelt funktioner i klasser). Dokumentationen är också konsistent då den följer samma mönster.

Kollar vi på koden nedan så tränar vi två olika modeller, en linjär regressionsmodell och ett beslutsträd. Trots att det är två helt olika modeller är processen densamma. Vi börjar med att (1) instantiera en modell, (2) träna modellen genom att använda `.fit()` metoden och därefter kan vi använda modellerna för att (3) göra prediktioner. Detta är ett exempel som visar på hur konsistent gränssnittet för ML-modeller är.

```
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeRegressor
from sklearn.datasets import make_regression
from sklearn.model_selection import train_test_split

x, y = make_regression(n_samples=100, n_features=1, noise=10,
    ↵ random_state=42)
x_train, x_test, y_train, y_test = train_test_split(x, y,
    ↵ test_size=0.2, random_state=42)

# Instantiate models
lin_model = LinearRegression()
tree_model = DecisionTreeRegressor(random_state=42)
```

①

```

# Fit models
lin_model.fit(x_train, y_train)
tree_model.fit(x_train, y_train)                                ②

# Predict
lin_preds = lin_model.predict(x_test)
tree_preds = tree_model.predict(x_test)                         ③

print("Linear Regression predictions:", lin_preds[:3])
print("Decision Tree predictions:      ", tree_preds[:3])      ④

```

- ① Vi instantierar respektive modell.
- ② Vi trärar respektive modell med `.fit()` metoden.
- ③ Vi använder respektive modell för att utföra prediktioner med `.predict()` metoden.
- ④ Vi skriver ut de tre första prediktioner från respektive modell.

```
Linear Regression predictions: [-58.66528327  65.48743554  36.04876271]
Decision Tree predictions:      [-63.16609268  64.57600193  51.39997923]
```

Läser vi dokumentationen för respektive modell, se följande länkar:

- [https://scikit-learn.org/stable/modules/generated/sklearn.linear\\_model.LinearRegression.html](https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html)
- <https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeRegressor.html>

Kommer vi också se att dokumentationen är uppbyggd på samma sätt och därmed konsistent. Exempelvis, i början av den kan vi se vilka hyperparametrar som finns och dokumentationen har också kodexempel som kan vara mycket användbara.

- Inspektion. Konstruktorparametrar (hyperparametrar, kallas för parametrar i *scikit-learn* dokumentationen) nås genom att använda `.get_params()` metoden. Lärda parametrar (parametrar, kallas för attribut i *scikit-learn* dokumentationen) är lagrade och nåbara som publika attribut, de nås genom att använda ett understräck som suffix. Vi demonstrerar detta i kodexemplet nedan.

**i** I *scikit-learn* kallas hyperparametrar för parametrar och parametrar för attribut

Det kan vara något förvirrande att konstruktörparametrar kallas för parametrar i *scikit-learn* dokumentationen. Rent generellt benämns det dock för hyperparametrar inom maskininlärning då det svarar på frågan ”hur vi lär oss”. Lärda parametrar kallas för attribut i *scikit-learn* dokumentationen. Rent generellt benämns det dock för parametrar inom maskininlärning då det svarar på frågan ”vad vi lärt oss”.

För att sammanfatta: i *scikit-learn* kallas hyperparametrar för parametrar och parametrar för attribut.

```
print(lin_model.get_params())①  
print(lin_model.intercept_)  
print(lin_model.coef_)②
```

- ① Vi kollar på hyperparametrarna genom att använda metoden `.get_params()`. Standardvärdet visas eftersom vi inte specificerade något annat i instantieringen av modellen i föregående kodblock.
- ② Vi kollar på parametrarna `.intercept_` och `.coef_`.

```
{'copy_X': True, 'fit_intercept': True, 'n_jobs': None, 'positive': False, 'tol': 1e-06}  
0.09922221422587718  
[44.24418216]
```

- Begränsning av klasser. Endast modellalgoritmer, såsom `LinearRegression`, är representerade genom byggda klasser. Data representeras som *NumPy arrays* eller *SciPy sparse matrices* och hyperparametrar representeras som vanliga Python strängar eller siffror när det är möjligt.
- Rimliga standardvärdet för hyperparametrarna. När det behöver specificeras vilka värden som ska användas för hyperparametrar så har *scikit-learn* specificerat rimliga standardvärdet vilket innebär att typanvändaren inte behöver tänka på det. Detta gör det enkelt att använda och implementera olika modeller. Det möjliggör även fler att faktiskt arbeta med ML då man inte behöver förstå teoretiska detaljer som i många fall inte har någon praktisk betydelse.

Kollar vi på koden nedan så ser vi att för ett beslutsträd så har exempelvis standardvärdet `squared_error` specificerats för hyperparametern `criterion` och standardvärdet `best` för hyperparametern `splitter` och så vidare. Hade vi behövt specificera allt sådant själva hade det krävts mycket arbete utan att det ger så mycket i majoriteten av fallen. Att *scikit-learn* använder rimliga standardvärden för hyperparametrarna är något som väldigt mycket underlättar praktiskt arbete och möjliggör fler att tillämpa och praktiskt arbeta med ML.

```
class sklearn.tree.DecisionTreeRegressor(*,
    ↵ criterion='squared_error', splitter='best', max_depth=None,
    ↵ min_samples_split=2, min_samples_leaf=1,
    ↵ min_weight_fraction_leaf=0.0, max_features=None,
    ↵ random_state=None, max_leaf_nodes=None,
    ↵ min_impurity_decrease=0.0, ccp_alpha=0.0, monotonic_cst=None)
```

## 2.4.2 *Estimators, predictors, transformers*

Det centrala objektet i *scikit-learn* benämns *estimators* och de lär sig (estimerar) från data genom `.fit()` metoden. *Estimators* (1) instantieras först innan (2) de tränas med `.fit()` metoden. I koden nedan demonstreras hur vi (1) instantierar en linjär regressionsmodell och (2) tränar modellen. Notera, det är samma kod som vi använt tidigare.

```
lin_model = LinearRegression()
lin_model.fit(x_train, y_train)
```

①

②

① Modellen instantieras.

② Modellen tränas genom `.fit()` metoden.

fit_intercept	True
copy_X	True
tol	1e-06
n_jobs	None
positive	False

*Estimators* som kan göra prediktioner genom `.predict()` metoden kallas för *predictors*. *Predictors* är alltså en delmängd av *estimators* och `LinearRegression()` som vi kollat

på ovan är alltså både en *estimator* och en *predictor*. *Predictors* måste alltid ha en *score* metod som kvantifierar hur bra prediktioner som görs. Högre *score* är alltid bättre i *scikit-learn*.

```
lin_model.predict(x_test)  
print(lin_model.score(x_test, y_test))
```

①  
②

- ① Vi använder *.predict()* metoden för att genomföra en prediktion.  
② Vi använder *.score()* metoden för att beräkna *score*.

0.9374151607623286

*Estimators* som kan transformera data genom *.transform()* metoden kallas för *transformers*. *Transformers* är alltså en delmängd av *estimators* och *StandardScaler()* som vi kollar på nedan är alltså både en *estimator* och en *transformer*. *.transform()* metoden tar datan vi vill transformera som ett argument och returnerar den transformerede datan.

```
scaler = StandardScaler()  
scaler.fit(x_train)  
x_transformed = scaler.transform(x_train)  
print(np.round(x_transformed.mean(), 1))  
print(x_transformed.std())  
  
# Transformers have a convenience method called fit_transform()  
# that does both fitting and transforming in one step  
scaler_2 = StandardScaler()  
x_transformed_2 = scaler_2.fit_transform(x_train)
```

①  
②  
③  
④

- ① Vi instantierar *StandardScaler()* som är en *transformer*.  
② Vi tränar vår *transformer*.  
③ Vi transformerar *x\_train* datan så att den har medelvärdet 0 och standardavvikelsen 1.  
④ Alla *transformers* i *scikit-learn* har metoden *.fit\_transform()* som möjliggör oss att göra två steg i ett. Kodens med den fjärde annoteringen gör alltså exakt samma sak som kodens med den andra och tredje annoteringen kombinerat.

-0.0  
1.0

### 2.4.3 Pipelines

En sekvens av databearbetningssteg kombinerade kallas för en data-*pipeline* eller bara *pipeline*. När vi skapar *pipelines* i *scikit-learn* så behöver alla *estimators* i den, förutom den sista, vara *transformers*. Den sista *estimatorn* kan vara antingen en *transformer* eller en *predictor*. Som vi kommer se är två fördelar med *pipelines* i *scikit-learn*:

- Bekvämlighet och effektivitet. Använder vi en *pipeline* behöver vi endast använda *.fit()* och *.predict()* metoderna en gång för hela *pipelinen*. Använder vi inte en *pipeline* måste vi kalla på metoderna för respektive *estimator* i sekvensen.
- Gemensam optimering av hyperparametrar. Vi kan använda *GridSearch* över samtliga hyperparametrar i *pipelinen* på en gång. Använder vi inte en *pipeline* måste vi göra det för varje separat *estimator* i sekvensen.

Vi kollar nu på två kodexempel där vi kommer använda oss av *pipelines*. I det första kodexemplet kommer vi hantera saknade värden och skapa *polynomial features*. Först gör vi båda stegen individuellt och sen kombinerar vi dem i en *pipeline*. Vi kommer se att resultatet blir detsamma men koden blir mer kompakt och lätthanterad genom att göra dem i en *pipeline*.

#### i Polynomial features

*Polynomial features* är ett sätt att skapa nya variabler. Detta kan användas i linjära regressionsmodeller för att exempelvis modellera icke-linjära samband med det som benämns för *polynom regression*. Det kommer gås igenom i Avsnitt 3.3.1.

```
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import Pipeline

A = np.array([[np.nan, np.nan, 2], [4, np.nan, 1], [10, 5, 9]]) ①
print(A)

print(np.nanmean(A, axis=0)) ②

# Doing two transformations
imp_mean = SimpleImputer(missing_values=np.nan, strategy='mean')
A_transformed = imp_mean.fit_transform(A)
```

```

print(A_transformed)

poly = PolynomialFeatures(2)                                     ③
A_transformed = poly.fit_transform(A_transformed)
print(A_transformed)

# Doing two transformations in one step with a pipeline
my_pipe = Pipeline([
    ('missing_values', SimpleImputer(missing_values=np.nan,
        strategy='mean')),
    ('polynomial', PolynomialFeatures(2)),
])
A_transformed_2 = my_pipe.fit_transform(A)                         ④
print(A_transformed_2)

```

- ① Vi skapar demonstrationsdata.
- ② Vi beräknar medelvärdet för respektive kolumn där vi ignorerar saknade värden, *nan*. Detta medelvärdet är vad som kommer fyllas i för de saknade värden när vi exekverar koden `imp_mean.fit_transform(A)` i kodraden nedan.
- ③ Vi specificerar att vi vill generera en ny *feature*-matris som består av alla polynom-kombinationer upp till och med den andra graden. I vår matris, A, har vi tre variabler/*features* som kan benämñas för  $x_1$ ,  $x_2$  och  $x_3$ . Dessa kommer transformeras enligt följande:  $(x_1, x_2, x_3) \rightarrow (1, x_1, x_2, x_3, x_1^2, x_1x_2, x_1x_3, x_2^2, x_2x_3, x_3^2)$ .
- ④ Vi skapar en pipeline.

```

[[nan nan  2.]
 [ 4. nan  1.]
 [10.  5.  9.]]
[7.  5.  4.]
[[ 7.  5.  2.]
 [ 4.  5.  1.]
 [10.  5.  9.]]
[[ 1.   7.   5.   2.  49.  35.  14.  25.  10.   4.]
 [ 1.   4.   5.   1. 16.  20.   4.  25.   5.   1.]
 [ 1.  10.   5.   9. 100.  50.  90.  25.  45.  81.]]
[[ 1.   7.   5.   2.  49.  35.  14.  25.  10.   4.]
 [ 1.   4.   5.   1. 16.  20.   4.  25.   5.   1.]
 [ 1.  10.   5.   9. 100.  50.  90.  25.  45.  81.]]

```

Vi ser att slutresultatet blir detsamma men att det är mer kompakt att använda en *pipeline*. Vi kan föreställa oss hur stor skillnaden blir om till exempel 10 stycken transformationer hade gjorts. Då blir det avsevärt mycket mindre kod genom att använda en *pipeline*.

Vi kollar nu på det andra kodexemplet. Vi kommer börja med att (1) generera slumpmässig data som vi använder för demonstration, (2) använda en *pipeline* där vi inte optimerar hyperparametrarna med *grid search*, detta kan jämföras med (3) där vi använder oss av en *pipeline* men optimerar hyperparametrarna med en *grid search* och slutligen kommer vi (4) visualisera båda modellerna.

```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import Ridge
from sklearn.model_selection import GridSearchCV          ①

# 1. Generating random data that will be used for demonstration
np.random.seed(42)
x = 5 * np.random.rand(100, 1) - 3
y = 0.6*x**3 + 0.6 * x**2 + 0.7*x + 2 + np.random.randn(100, 1)

x_new = np.linspace(-3, 2.5, 20).reshape(-1, 1) # This data will
    be used for prediction

# 2. Creating a pipeline with "arbitrarily" set hyperparameters
pipe = Pipeline(steps=[
    ('poly_features', PolynomialFeatures(degree=2,
                                         include_bias=False)),
    ('regression', Ridge(alpha=1))           ②
])                                         ③

pipe.fit(x, y)
y_ridge_pred = pipe.predict(x_new)

# 3. Creating a pipeline with hyperparameters chosen by GridSearch
print("Naming of hyperparameters:", pipe.get_params(), "\n")      ④
```

```

param_grid = [
    {'poly_features__degree': [1, 2, 3, 4], 'regression__alpha':
     [0.5, 1, 1.5]}
]
grid_search = GridSearchCV(pipe, param_grid, cv = 3, scoring =
                           'neg_mean_squared_error')
grid_search.fit(x, y)

print("Optimal hyperparameters:", grid_search.best_estimator_) ⑤

y_ridge_pred_gs = grid_search.predict(x_new)

# 4. Plotting both models
fig, ax = plt.subplots()
ax.set_title("Linear Regression")
ax.scatter(x, y, color='black', label = "data")
ax.plot(x_new, y_ridge_pred, 'b--', label = 'Pipeline without grid
           search')
ax.plot(x_new, y_ridge_pred_gs, 'r--', label = 'Pipeline with grid
           search')
ax.legend()

```

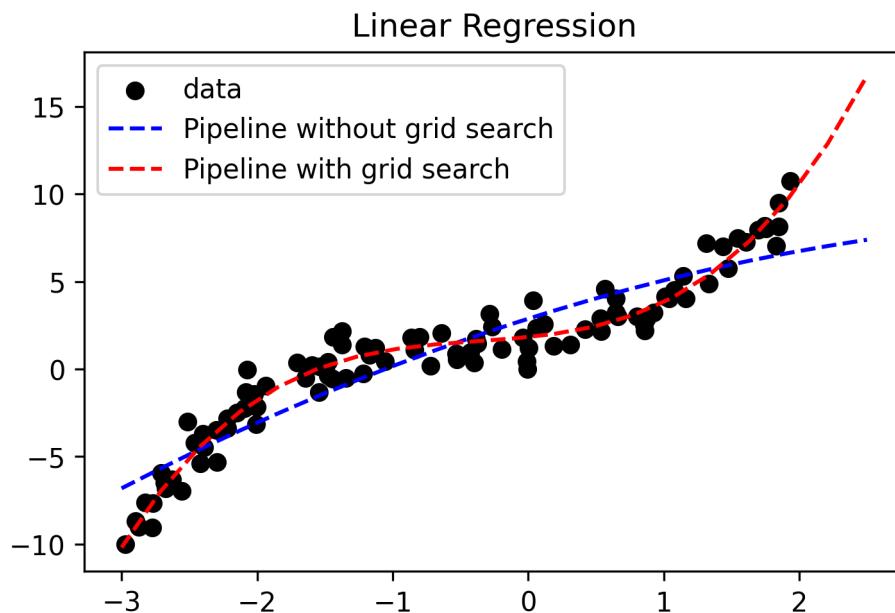
- ① Vi importerar de bibliotek som används.
- ② *Ridge* är en regulariserad regressionsmodell där regulariseringssstyrkan styrs av hyperparametern *alpha*. Vi kommer lära oss om denna modell i Avsnitt 3.3.4.
- ③ Vi skapar en *pipeline* där den sista *estimatore*n är en *predictor*.
- ④ Vi kollar vilka hyperparametrar som finns och hur de benämns. Notera namngivningen med två understräck, exempelvis *poly\_features\_\_degree*.
- ⑤ Vi är nyfikna på vilka hyperparametrar som är optimala och skriver ut dessa.

```

Naming of hyperparameters: {'memory': None, 'steps': [('poly_features',
PolynomialFeatures(include_bias=False)), ('regression',
Ridge(alpha=1))], 'transform_input': None, 'verbose': False,
'poly_features': PolynomialFeatures(include_bias=False), 'regression':
Ridge(alpha=1), 'poly_features__degree': 2,
'poly_features__include_bias': False, 'poly_features__interaction_only':
False, 'poly_features__order': 'C', 'regression__alpha': 1,
'regression__copy_X': True, 'regression__fit_intercept': True,
'regression__max_iter': None, 'regression__positive': False,
'regression__random_state': None, 'regression__solver': 'auto',
'regression__tol': 0.0001}

Optimal hyperparameters: Pipeline(steps=[('poly_features',
PolynomialFeatures(degree=3, include_bias=False)),
('regression', Ridge(alpha=1.5))])

```



Vi ser från figuren att de prediktioner som genomfördes med den *pipeline* där hyperparametrarna optimerades med *grid search* generellt sett följer datan bättre vilket är

precis som förväntat. Huruvida modellen faktiskt är bättre på ny, osedd data hade vi behövt undersöka genom att jämföra modellerna på valideringsdata.

#### 2.4.4 Ett avstick till *TensorFlow* och *Keras*

Vi har i detta avsnitt pratat om *scikit-learn* som är ett mycket populärt och använt bibliotek inom ML. Kollar vi på Figur 2.2 så ser vi att de bibliotek som är på delad andra och tredje plats (de har exakt samma procent, 50.5%), är *TensorFlow* och *Keras*. Dessa bibliotek kommer vi använda senare i bokens tredje del (Djupinlärning) med start från Kapitel 7.

Nedan citerar vi från källan <https://www.tensorflow.org/guide/keras> som på ett mycket bra sätt förklrar hur de två biblioteken, *Tensorflow* och *Keras*, hänger ihop.

##### **Keras: The high-level API for TensorFlow.**

*“Keras is the high-level API of the TensorFlow platform. It provides an approachable, highly-productive interface for solving machine learning (ML) problems, with a focus on modern deep learning. Keras covers every step of the machine learning workflow, from data processing to hyperparameter tuning to deployment. It was developed with a focus on enabling fast experimentation.”*

##### **Who should use Keras?**

*“The short answer is that every TensorFlow user should use the Keras APIs by default. Whether you’re an engineer, a researcher, or an ML practitioner, you should start with Keras.”*

Sammantaget så kan vi tänka på *Keras* som ratten och *Tensorflow* som motorn i en bil och vi kommer fokusera på att använda *Keras*.

*Scikit-learn* har också funktionalitet för djupinlärning men det är inte i närheten lika sofistikerat som *TensorFlow* och *Keras* inom detta.

## 2.5 Övningsuppgifter

Övningsuppgifter för kapitlet finns på bokens hemsida:  
[https://github.com/AntonioPrgomet/ai\\_tillaempad\\_ml](https://github.com/AntonioPrgomet/ai_tillaempad_ml)

## **Del II**

# **Maskininlärning**



# Kapitel 3

# Regression

I detta kapitel kommer vi att fördjupa oss inom regressionsproblem där målet är att prediktera kontinuerlig utdata med hjälp av indata. Kapitlet inleder med att förklara vad regressionsproblem är och ger exempel på diverse tillämpningsområden. Därefter presenteras olika utvärderingsmått följt av olika regressionsmodeller som används för att genomföra prediktioner.

I kapitlet kommer vi göra två avstickare till optimeringsalgoritmen *gradient descent* (Avsnitt 3.3.2) och *the bias-variance-tradeoff* (Avsnitt 3.3.3) efter att vi gått igenom den linjära regressionsmodellen (Avsnitt 3.3.1). *Gradient descent* är en allmän optimeringsalgoritm som används inom hela ML. Det är dock pedagogiskt att förklara det i samband med att den linjära regressionsmodellen gåtts igenom eftersom vi då har något konkret att utgå ifrån. *The bias-variance-tradeoff* är ett mycket viktigt koncept som förklarar varför mer komplexa modeller inte alltid är bättre. Vi kommer börja använda det för att förstå hur den linjära regressionsmodellen med regularisering kan utvecklas till *ridge regression*, *lasso regression* och *elastic net*. Men vi betonar att *the bias-variance-tradeoff* är ett generellt resultat och något vi kommer ha nytta av i hela boken.

## 3.1 Regressionsproblem

Regressionsproblem handlar om att prediktera en kontinuerlig beroende variabel ( $y$ ) med hjälp av oberoende variabler ( $x$ ). Det kan exempelvis handla om att prediktera en

persons lön ( $y$ ) med avseende på personens ålder ( $x$ ). Det finns flera tillämpningsområden inom olika branscher och några exempel tagna från verkligheten ges nedan.

- **Bilvärdering** - Prediktera försäljningspriset på en bil beroende på variabler som miltal, ålder, bränsletyp och märke.
- **Fastighetsvärdering** - Prediktera försäljningspriset på en fastighet beroende på variabler som storlek, byggnadsår, geografiskt läge och antal rum.
- **Jordbruksproduktion** - Prediktera hur stor skörd (exempelvis potatis) blir i kilogram baserat på variabler som temperatur, mängden regn och bruket av gödningsmedel.
- **Försäkringar** - Prediktera förväntad skadekostnad i kronor för en kund baserat på variabler som ålder, utbildning, geografiskt läge och tidigare historik.
- **Efterfrågan på produkter** - Prediktera hur många enheter av en produkt som kommer säljas (volym) baserat på pris, säsong, kampanjer och tidigare försäljningsdata.
- **Lönenivå** - Prediktera en persons lön i kronor beroende på variabler som utbildningsnivå, ålder, erfarenhet, bransch och geografiskt område.

Härnäst kommer vi lära oss om olika utvärderingsmått som används för att utvärdera regressionsmodeller.

## 3.2 Utvärderingsmått

För att utvärdera regressionsmodeller är nedanstående mått vanligt förekommande:

- *Root Mean Square Error (RMSE)*
- *Mean Squared Error (MSE)*
- *Mean Absolute Error (MAE)*

Vi går nu igenom dessa.

### 3.2.1 *Root mean squared error (RMSE)*

*RMSE* är det mått som används i störst utsträckning i samband med att regressionsmodeller utvärderas. Måttet gicks igenom i Avsnitt 1.3.3 och i Ekvation 3.1 ser vi återigen

dess beräkningsformel.

$$\text{RMSE} = \sqrt{\frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2} \quad (3.1)$$

### 3.2.2 Mean squared error (MSE)

Generellt sett använder vi *RMSE* om vi vill kunna tolka hur stort fel en modell gör. Vi kan även använda det för att rangordna olika modeller där en modell är bättre än en annan om den har en lägre *RMSE*. År vi endast intresserade av rangordning av modeller är det dock generellt sett mer effektivt att använda *MSE* eftersom rangordningen av modellerna blir densamma men det är beräkningsmässigt billigare att beräkna *MSE* eftersom inga beräkningar av kvadratroten behöver göras vilket kan spara tid.

Beräkningsformeln för *MSE* ser vi i Ekvation 3.2.

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2 \quad (3.2)$$

Vi kan använda *scikit-learn* för att beräkna *MSE* enligt koden nedan.

```
from sklearn.metrics import mean_squared_error  
y_true = [3, -0.5, 2, 7] ①  
y_pred = [2.5, 0.0, 2, 8] ②  
  
mean_squared_error(y_true, y_pred) ③  
④
```

- ① Importera *MSE* från *scikit-learn* biblioteket.
- ② Vi antar att vi har sanna värden,  $y_i$ , som representeras av *y\_true*.
- ③ Vi antar att vi har predikterade värden,  $\hat{y}_i$ , som representeras av *y\_pred*.
- ④ Vi beräknar *MSE*.

0.375

### 3.2.3 Mean absolute error (MAE)

Precis som för *RMSE* och *MSE* så gäller det också för *MAE* att ett lägre tal indikerar en bättre modell. Beräkningsformeln för *MAE* ser vi i Ekvation 3.3.

$$\text{MAE} = \frac{1}{n} \sum_{i=1}^n |y_i - \hat{y}_i| \quad (3.3)$$

Notera att  $|x|$  är det som benämns för absolutbeloppet av  $x$  där det exempelvis gäller att  $|4| = 4$  och  $|-4| = 4$ . Absolutbeloppet är alltså en funktion som alltid returnerar den positiva motsvarigheten av ett tal. Matematiskt definieras absolutbeloppet enligt Ekvation 3.4.

$$|x| = \begin{cases} x, & \text{if } x \geq 0 \\ -x, & \text{if } x < 0 \end{cases} \quad (3.4)$$

Vi kan använda *scikit-learn* för att beräkna *MAE* enligt koden nedan.

```
from sklearn.metrics import mean_absolute_error
y_true = [3, -0.5, 2, 7]
y_pred = [2.5, 0.0, 2, 8]
mean_absolute_error(y_true, y_pred)
```

0.5

Det går att matematiskt bevisa att  $RMSE \geq MAE$  eftersom  $RMSE$  lägger ett större straff på stora avvikelse än vad  $MAE$  gör. Det betyder att i de fall vi vill undvika stora fel är  $RMSE$  ett lämpligt mått att använda. I de fall alla fel värderas linjärt med avseende på felens storlek är  $MAE$  ett lämpligt mått att använda. I koden nedan exemplifieras detta.

```
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import root_mean_squared_error

y_true = [100, 80]                                     ①
y_pred_model_1 = [90, 75]                             ②
y_pred_model_2 = [90, 15]                             ③

MAE1 = mean_absolute_error(y_true, y_pred_model_1)
MAE2 = mean_absolute_error(y_true, y_pred_model_2)

RMSE1 = root_mean_squared_error(y_true, y_pred_model_1)
RMSE2 = root_mean_squared_error(y_true, y_pred_model_2)

print("MAE1 =", MAE1)
```

```
print("MAE2 =", MAE2)
print("RMSE1 =", RMSE1)
print("RMSE2 =", RMSE2)
```

- ① Vi antar att vi har två sanna värden som är 100 och 80.
- ② Vi antar att vi har en modell som predikterar värdena 90 och 75.
- ③ Vi antar att vi har en modell som predikterar värdena 90 och 15.

```
MAE1 = 7.5
MAE2 = 37.5
RMSE1 = 7.905694150420948
RMSE2 = 46.502688094345686
```

Från resultatet ser vi att den första modellen gör ett fel på  $(100 - 90) + (80 - 75) = 15$  och den andra modellen gör ett fel på  $(100 - 90) + (80 - 15) = 75$ . Felet skiljer sig alltså med en faktor 5 eftersom  $15 \cdot 5 = 75$ . Kollar vi på *MAE* så går den från 7.5 till 37.5 vilket är en ökning med faktorn 5 ( $7.5 \cdot 5 = 37.5$ ) medan *RMSE* ökar med en faktor som är cirka 5.89 ( $\frac{46.5}{7.9} \approx 5.89$ ). *MAE* ökade alltså linjärt medan *RMSE* ökade icke-linjärt där det senare måttet alltså lägger ett större straff på stora avvikelser.

Generellt sett är *RMSE* det mått som används mest men beroende på vad man önskar uppnå kan även *MAE* vara ett lämpligt mått att använda.

### 3.3 Regressionsmodeller

Några vanligt förekommande modeller för regressionsproblem är:

- Linjär regression
- *Ridge* regression
- *Lasso* regression
- *Elastic net*
- *Support vector machines*
- Beslutsträd
- *Ensemble learning*
- *Random forest*

## i Hur data kan se ut

I samband med att vi arbetar med data så kan det se ut enligt nedan.

	Inkomst (y)	Ålder (x1)	Utbildningsnivå (x2)
1	35000	34	2
2	40000	33	5
3	95000	58	3
4	21000	19	0

Vi har alltså tre variabler (kolumner) och totalt fyra observationer (rader). Hade vi använt index notation för att beteckna värdena så hade det skrivits enligt nedan.

$$\begin{cases} y_1, & x_{11}, & x_{12} \\ y_2, & x_{21}, & x_{22} \\ y_3, & x_{31}, & x_{32} \\ y_4, & x_{41}, & x_{42} \end{cases}$$

Notera alltså att exempelvis  $y_1$  representerar 35000,  $x_{11}$  representerar 34 och  $x_{12}$  representerar 2.

För att demonstrera hur de olika modellerna fungerar kommer vi skapa ett syntetiskt dataset med `make_regression` funktionaliteten från *scikit-learn*. Att använda syntetiska dataset är smidigt för att snabbt kunna demonstrera och testa olika modeller.

```
from sklearn.datasets import make_regression
from sklearn.model_selection import train_test_split

X, y = make_regression(n_samples=5000, n_features=3, noise=5,
    ↵ random_state=42)                                ①
print(X[:4, :])                                    ②
print(y[:4])                                       ③

X_train, X_test, y_train, y_test = train_test_split(X, y,
    ↵ test_size=0.2, random_state=42)                  ④
```

① Vi genererar ett syntetiskt dataset som innehåller 5000 observationer (rader) och tre

oberoende variabler (kolumner). `random_state=42` har specificerats så att vi får reproducerbara resultat.

- ② Vi skriver ut de fyra första värdena från våra tre oberoende variabler ( $x$ ).
- ③ Vi skriver ut de fyra första värdena från vår beroende variabel ( $y$ ).
- ④ Dela upp datan i träningsdata och testdata.

```
[[ 0.67796997 -1.28472777 -0.33102433]
 [ 1.03138053  0.3881858 -0.97027133]
 [-1.21689671  1.36337651 -0.60515624]
 [-0.54429615 -0.50442268 -1.5198928 ]]
[ -1.88062329 -46.40116933 -95.83251096 -155.05295711]
```

Härnäst börjar vi med att gå igenom linjär regression. Därefter kommer två avstick göras där det ena är optimeringsalgoritmen *gradient descent* och det andra är *the bias variance trade-off*. Dessa två koncept används rent generellt inom ML men de är lättare att förstå och förklara direkt efter att den linjära regressionsmodellen gäts igenom. Efter det fortsätter vi med att gå igenom resterande modeller. För samtliga modeller vi går igenom kommer det syntetiska datasetet vi skapade ovan med `make_regression` att användas för demonstration.

### 3.3.1 Linjär regression

Den linjära regressionsmodellen är en enkel modell som är väldigt central. Den används ofta i verkligheten och är en bra modell för att rent allmänt förstå modellering.

Den enklaste regressionsmodellen benämns *enkel linjär regression* och den antar att det endast finns en oberoende variabel  $x$ . Modellens prediktioner,  $\hat{y}$ , beräknas enligt Ekvation 3.5.

$$\hat{y} = \hat{\theta}_0 + \hat{\theta}_1 x. \quad (3.5)$$

Notera att  $\hat{\theta}_0$  och  $\hat{\theta}_1$  är de skattade parametrarna, det vill säga skattningar på de fixa men okända parametrarna  $\theta_0$  och  $\theta_1$ . Mer om detta i Avsnitt 3.3.2. Skillnaden mellan det sanna värdet ( $y$ ) och det predikterade värdet ( $\hat{y}$ ) kallas för residualen och betecknas med  $e$ , det vill säga  $e = y - \hat{y}$ .

Den observanta läsaren noterar kanske att formeln,  $\hat{y} = \hat{\theta}_0 + \hat{\theta}_1 x$  helt enkelt är den räta linjens ekvation som brukar skrivas  $y = kx + m$  där  $m$  i vårt fall är  $\hat{\theta}_0$  och  $k$  är  $\hat{\theta}_1$ .

För att konkretisera ovanstående, kan vi tänka oss att vi har ett dataset för olika personer där vi vet deras inkomst och ålder. Några rader från den datan kan vi se i Tabell 3.2.

Tabell 3.2: Exempeldata för enkel linjär regression där vi skriver ut de åtta första raderna från datasetet.

	Inkomst (y)	Ålder (x)
0	35000	34
1	40000	33
2	95000	58
3	22000	19
4	26000	20
5	55000	40
6	42000	66
7	33000	34

Om vi tränar en linjär regressionsmodell på den datan hade det kunnat se ut som i Figur 3.1. Notera att de svarta punkterna representerar data ( $y_i$ ), den blåa linjen är de predikterade värdena,  $\hat{y} = \hat{\theta}_0 + \hat{\theta}_1 x$ , enligt vår enkla linjära regressionsmodell. Vi ser att  $\hat{\theta}_0$  visar interceptet, det vill säga var vi skär y-axeln och  $\hat{\theta}_1$  är lutningen. Lutningen svarar på frågan ”hur mycket ökar inkomsten när vi ökar  $x$  (åldern) med en enhet”. De röda linjerna representerar residualer, det vill säga  $e_i = y_i - \hat{y}_i$ .

Vi demonstrerar nu ett kodexempel där vi ser hur vi kan träna en linjär regressionsmodell, skriva ut värdena på parametrarna, prediktera en ny observation och slutligen även visualisera modellen.

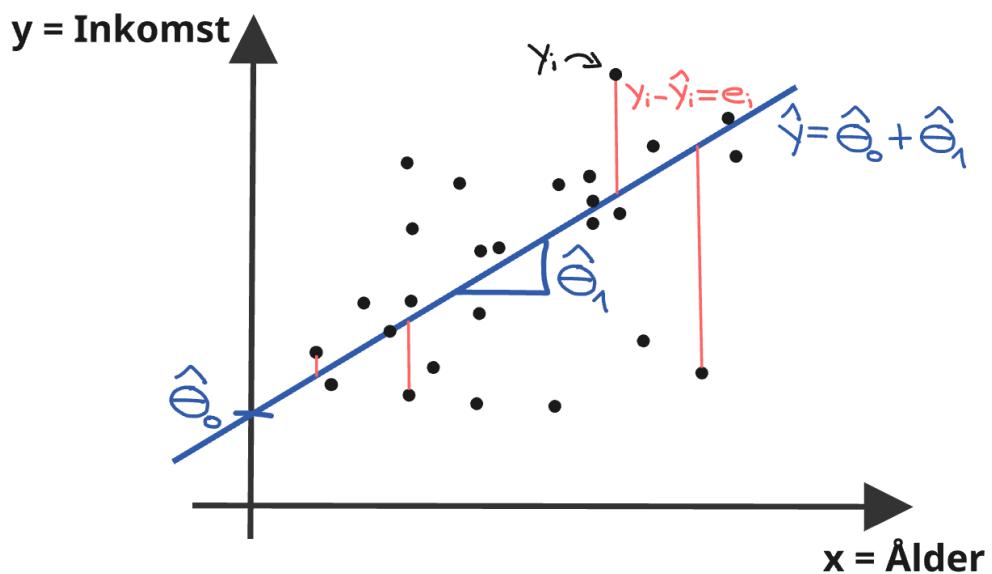
```

import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression      ①

# Data
x = np.array([6, 7, 8, -4, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4,
    ↵ 5]).reshape(-1, 1)                                ②
y = np.array([25, 40, 30, 40, 28, 35, 12, 20, 15, 3, 8, 18, 5,
    ↵ 22, 26, 33])                                     ③

# Fit model
lin_reg = LinearRegression()
lin_reg.fit(x, y)                                    ④

```



Figur 3.1: En schematisk bild över den enkla linjära regressionsmodellen. De svarta punkterna representerar data ( $y_i$ ) och den blåa linjen är de predikterade värdena,  $\hat{y} = \hat{\theta}_0 + \hat{\theta}_1 x$ , enligt vår regressionsmodell. De röda vertikala linjerna är residualer,  $y_i - \hat{y}_i = e_i$ .

```

# Print parameters
print("Intercept:", lin_reg.intercept_) (5)
print("Slope:", lin_reg.coef_) (6)

# Predicting a new observation
x_new = np.array(2).reshape(-1, 1) (7)
print("Predicting a new observation with model",
    ↪ lin_reg.predict(x_new)) (8)
print("Predicting a new observation manually",
    ↪ lin_reg.intercept_ + lin_reg.coef_*x_new) (9)

# Create a plot
x_values = np.linspace(x.min(), x.max(), 100).reshape(-1, 1)
y_pred = lin_reg.predict(x_values)

fig, ax = plt.subplots(figsize=(8, 5))
ax.scatter(x, y, color='black', label='Data points')
ax.plot(x_values, y_pred, color='blue', label='Predictions from
    ↪ linear regression')
ax.set_xlabel("x")
ax.set_ylabel("y")
ax.set_title("Linear Regression ")
ax.legend() (10)

```

- ① Bibliotek som används importeras.
- ② Vi skapar data för demonstrationssyfte. Vi använder `.reshape(-1, 1)` så datan har korrekt form. -1 innebär att datan kommer ha så många rader som det behövs för att datan ska få plats och 1 representerar att vi har en kolumn, detta eftersom vi endast har en oberoende variabel i vårt exempel.
- ③ Vi instantierar en linjär regressionsmodell.
- ④ Vi tränar modellen på datan.
- ⑤ När vi tränar modellen så skattar den parametrarna  $\theta_0$  och  $\theta_1$  vilket ger oss  $\hat{\theta}_0$  och  $\hat{\theta}_1$ . Denna kodrad skriver ut  $\hat{\theta}_0$ .
- ⑥ Denna kodrad skriver ut  $\hat{\theta}_1$ .
- ⑦ Vi antar att vi vill prediktera en ny observation. Vi sparar värdet i variabeln `x_new`.
- ⑧ Vi genomför prediktionen genom att använda `.predict()` metoden.
- ⑨ Vi visar att vi även kan beräkna prediktioner manuellt efter att modellen tränats.

Vi beräknar alltså  $\hat{y} = \hat{\theta}_0 + \hat{\theta}_1 \cdot 2$ .

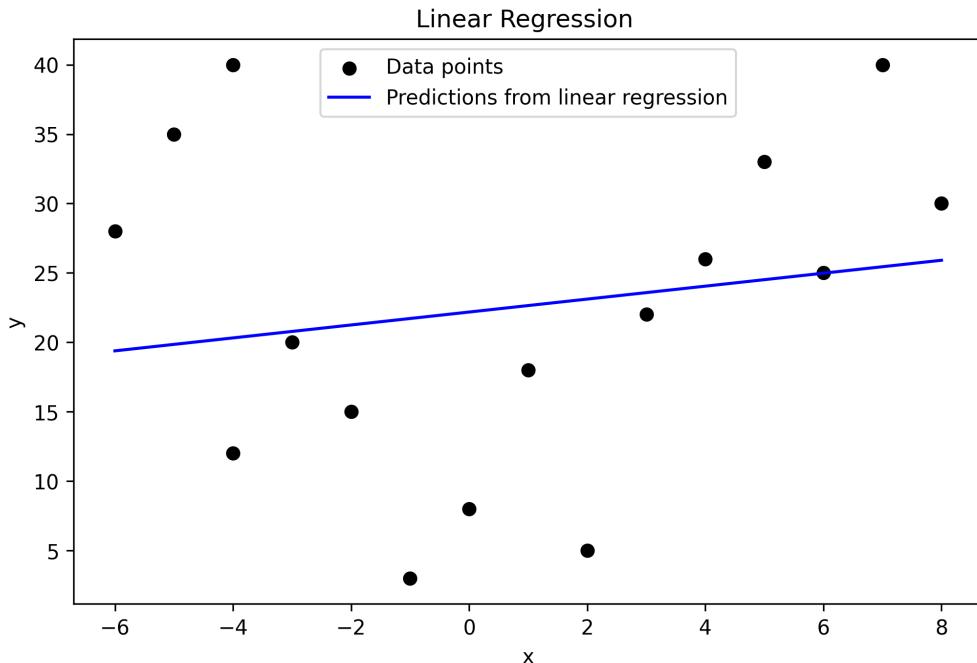
- ⑩ Vi genomför en visualisering av vår modell.

Intercept: 22.179402677651904

Slope: [0.46632338]

Predicting a new observation with model [23.11204943]

Predicting a new observation manually [[23.11204943]]



Om vi kollar på datan från kodexemplet ovan så ser vi att den ser ut att ha ett kvadratiskt mönster. Vi kanske kan prova att lägga till en variabel  $x^2$  för att se om det bättre fångar mönstret i datan? Vi har då modellspecifikationen enligt Ekvation 3.6.

$$\hat{y} = \hat{\theta}_0 + \hat{\theta}_1 x + \hat{\theta}_2 x^2 \quad (3.6)$$

När vi lägger till polynomtermer, exempelvis  $x^2$ , som vi gjorde ovan kallas det för *polynomregression*.

För att tillämpa detta i kod har vi `PolynomialFeatures` från *scikit-learn* till vår hjälp.

Se kodexemplet nedan.

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

# Data
x = np.array([6, 7, 8, -4, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4,
    ↵ 5]).reshape(-1, 1)
y = np.array([25, 40, 30, 40, 28, 35, 12, 20, 15, 20, 8, 18, 5,
    ↵ 22, 26, 33])

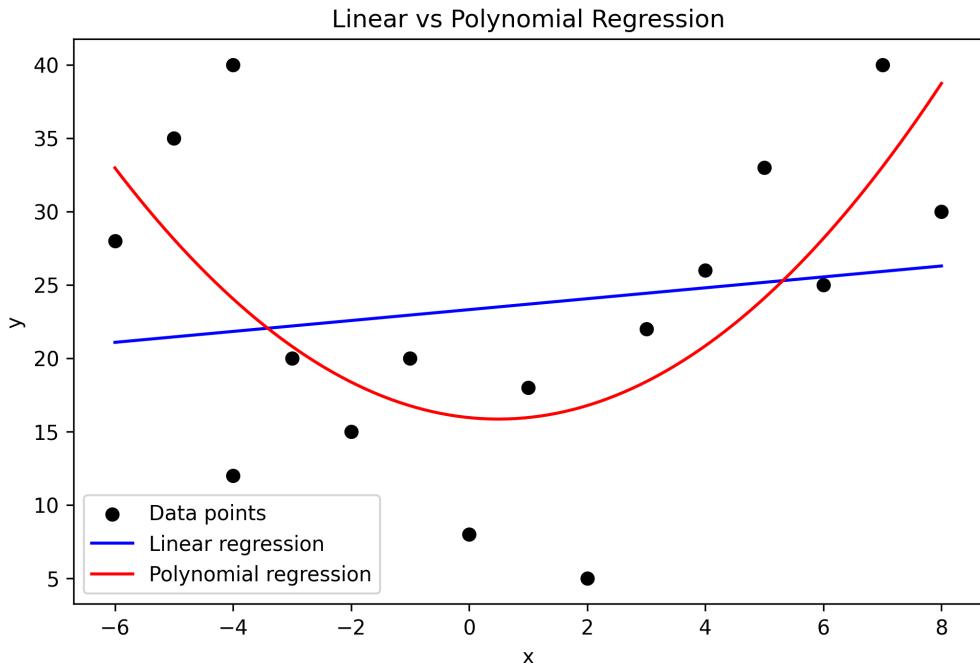
# Linear regression
lin_reg = LinearRegression()
lin_reg.fit(x, y)

# Polynomial regression
poly = PolynomialFeatures(degree=2, include_bias=False)
x_poly = poly.fit_transform(x)
poly_reg = LinearRegression()
poly_reg.fit(x_poly, y)

# Create a plot
x_values = np.linspace(x.min(), x.max(), 100).reshape(-1, 1)
y_pred_linear = lin_reg.predict(x_values)
x_values_poly = poly.transform(x_values)
y_pred_poly = poly_reg.predict(x_values_poly)

fig, ax = plt.subplots(figsize=(8, 5))
ax.scatter(x, y, color='black', label='Data points')
ax.plot(x_values, y_pred_linear, color='blue', label='Linear
    ↵ regression')
ax.plot(x_values, y_pred_poly, color='red', label='Polynomial
    ↵ regression')
ax.set_xlabel("x")
ax.set_ylabel("y")
ax.set_title("Linear vs Polynomial Regression")
```

```
ax.legend()
```



Från visualiseringen ser vi att *polynomregression*-modellen verkar fungera bra.

Vad vi gjorde i exemplet med polynomregression var att vi lade till en oberoende variabel. Detta är en generalisering och rent generellt så kan vi generalisera den enkla linjära regressionsmodellen genom att inkludera fler oberoende variabler. Om vi antar att det finns  $p$  stycken oberoende variabler blir modellens predikterade värden,  $\hat{y}$ , enligt Ekvation 3.7. Det kallas ofta för den linjära regressionsmodellen men vill man betona att det är fler variabler kan vi kalla det för *multipel linjär regression*.

$$\hat{y} = \hat{\theta}_0 + \hat{\theta}_1 x_1 + \hat{\theta}_2 x_2 + \dots + \hat{\theta}_p x_p \quad (3.7)$$

**i Polynomregression är fortfarande en linjär regressionsmodell**

Om vi exempelvis har en linjär regressionsmodell på formen  $\hat{y} = \hat{\theta}_0 + \hat{\theta}_1 x_1 + \hat{\theta}_2 x_2$  så hade vi kunnat lägga till polynomtermer så att vår modell får formen:

$$\hat{y} = \hat{\theta}_0 + \hat{\theta}_1 x_1 + \hat{\theta}_2 x_2 + \hat{\theta}_3 x_1 x_2 + \hat{\theta}_4 x_1^2 + \hat{\theta}_5 x_2^2.$$

Detta är en polynomregression-modell men det är fortfarande ett exempel på en linjär regressionsmodell. När vi säger att den är linjär så menar vi alltså att den är linjär med avseende på parametrarna  $\hat{\theta}_i$ . Vi hade också kunnat kalla  $x_1 x_2 = x_3$ ,  $x_1^2 = x_4$  och  $x_2^2 = x_5$ . Då hade vår modell haft formen:

$$\hat{y} = \hat{\theta}_0 + \hat{\theta}_1 x_1 + \hat{\theta}_2 x_2 + \hat{\theta}_3 x_3 + \hat{\theta}_4 x_4 + \hat{\theta}_5 x_5$$

vilket gör det väldigt tydligt att det fortfarande är en linjär regressionsmodell.

Om vi rent generellt vill betona att vi kan prediktera specifika datapunkter/observationer kan vi använda oss av ett index  $i$ . Då skrivs Ekvation 3.7 om till Ekvation 3.8 där  $\hat{y}_i$  representerar det predikterade värdet för observation  $i$ .

$$\hat{y}_i = \hat{\theta}_0 + \hat{\theta}_1 x_{i1} + \hat{\theta}_2 x_{i2} + \dots + \hat{\theta}_p x_{ip} \quad (3.8)$$

Om vi antar att vi har  $n$  stycken datapunkter som predikterats och vi explicit vill betona detta så kan vi använda notationen enligt Ekvation 3.9.

$$\begin{cases} \hat{y}_1 = \hat{\theta}_0 + \hat{\theta}_1 x_{11} + \hat{\theta}_2 x_{12} + \dots + \hat{\theta}_p x_{1p} \\ \hat{y}_2 = \hat{\theta}_0 + \hat{\theta}_1 x_{21} + \hat{\theta}_2 x_{22} + \dots + \hat{\theta}_p x_{2p} \\ \vdots \\ \hat{y}_n = \hat{\theta}_0 + \hat{\theta}_1 x_{n1} + \hat{\theta}_2 x_{n2} + \dots + \hat{\theta}_p x_{np} \end{cases} \quad (3.9)$$

Detta kan på ett kompakt sätt skrivas i matrisform enligt Ekvation 3.10.

$$\hat{\mathbf{y}} = \mathbf{X}\hat{\boldsymbol{\theta}} \quad (3.10)$$

där

$$\hat{\mathbf{y}} = \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \vdots \\ \hat{y}_n \end{bmatrix}, \quad \mathbf{X} = \begin{bmatrix} 1 & x_{11} & x_{12} & \cdots & x_{1p} \\ 1 & x_{21} & x_{22} & \cdots & x_{2p} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n1} & x_{n2} & \cdots & x_{np} \end{bmatrix}, \quad \hat{\boldsymbol{\theta}} = \begin{bmatrix} \hat{\theta}_0 \\ \hat{\theta}_1 \\ \hat{\theta}_2 \\ \vdots \\ \hat{\theta}_p \end{bmatrix}.$$

När vi använder två oberoende variabler, innebärande att vi har formen,  $\hat{y} = \hat{\theta}_0 + \hat{\theta}_1 x_1 + \hat{\theta}_2 x_2$ , så kan vi visualisera det. Regressionslinjen i Figur 3.1 generaliseras då till ett plan, se Figur 3.2. I högre dimensioner, när vi har tre eller fler oberoende variabler, kan vi inte visualisera det på ett direkt sätt.

Vi använder datasetet skapat med `make_regression` för att skapa en linjär regressionsmodell och en polynomregression-modell som sedan utvärderas med *RMSE*.

```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import Pipeline
from sklearn.metrics import root_mean_squared_error
from sklearn.linear_model import LinearRegression

lin_reg = LinearRegression()

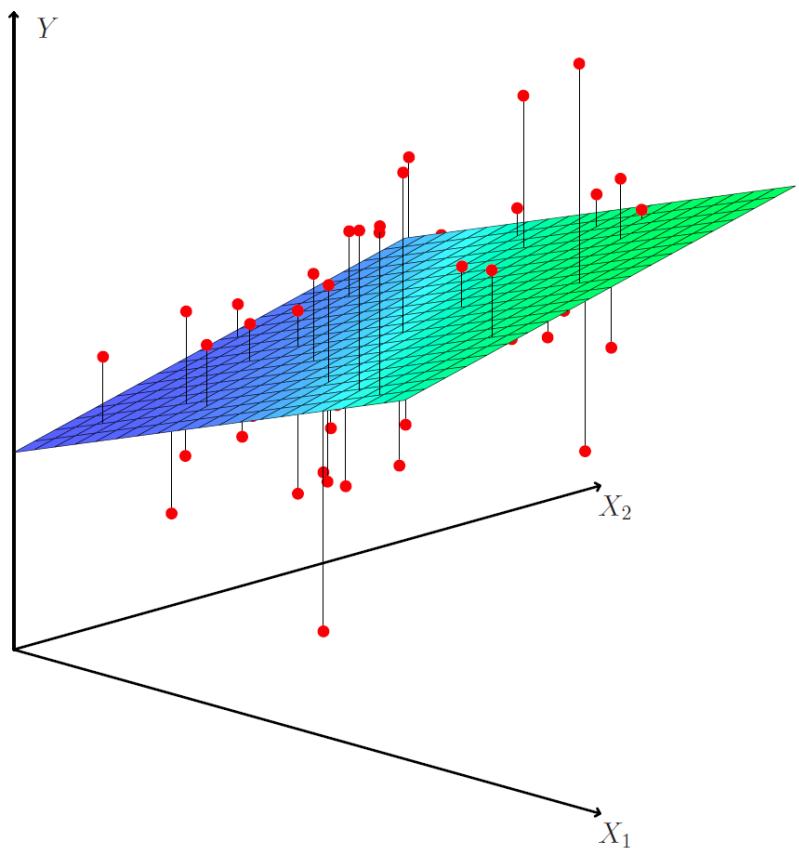
poly_reg_pipeline = Pipeline([
    ('poly', PolynomialFeatures(degree=2, include_bias=False)),
    ('polyreg', LinearRegression())
])(1)

lin_reg.fit(X_train, y_train)
poly_reg_pipeline.fit(X_train, y_train)

y_pred_lin_reg = lin_reg.predict(X_test)
y_pred_poly_reg = poly_reg_pipeline.predict(X_test)

RMSE_lin_reg = root_mean_squared_error(y_test, y_pred_lin_reg)
RMSE_poly_reg = root_mean_squared_error(y_test, y_pred_poly_reg)(2)

print("RMSE for Linear Regression:", RMSE_lin_reg)
```



Figur 3.2: En visualisering över en linjär regressionsmodell som har två oberoende variabler,  $x_1$  och  $x_2$ . Bilden är tagen från An Introduction to Statistical Learning with Applications in R (2:a upplagan) av James et al. (2023). <https://www.statlearning.com/>.

```

print("RMSE for Polynomial Regression:", RMSE_poly_reg)           (3)

if RMSE_lin_reg < RMSE_poly_reg:
    print("The linear regression model has lower RMSE and is hence
          ↵ better.")
elif RMSE_lin_reg == RMSE_poly_reg:
    print("Both models have the same RMSE meaning they are equally
          ↵ good.")
else:
    print("The polynomial regression model has lower RMSE and is
          ↵ hence better.")                                         (4)

```

- ① Vi skapar en *pipeline* för att kombinera skapandet av polynomtermer och modellträningen i ett steg.
- ② Vi beräknar *RMSE*.
- ③ Vi skriver ut *RMSE*.
- ④ Vi skapar kod för att skriva ut vilken modell som har lägst *RMSE* och därmed är bättre.

RMSE for Linear Regression: 5.09730997273096  
 RMSE for Polynomial Regression: 5.101743953679334  
 The linear regression model has lower RMSE and is hence better.

### **i Samma modell men olika syften - Den linjära regressionsmodellen i statistik och maskininlärning**

Den läsare som har erfarenhet av mer avancerad statistik känner säkert igen den linjära regressionsmodellen även därifrån. Inom såväl statistik som maskininlärning så används den linjära regressionsmodellen. Även om själva modellen är densamma, är syftet med modellen ofta annorlunda inom de två ämnena.

Inom statistik är syftet med en linjär regressionsmodell ofta att genomföra inferens; det vill säga förstå samband mellan variabler, beräkna konfidensintervall och genomföra statistisk hypotesprövning. Inom maskininlärning är syftet med en linjär regressionsmodell ofta att genomföra prediktioner på ny, osedd data. Den läsare som är intresserad av statistisk modellering kan exempelvis kolla på sidan <https://www.statlearning.com/> där boken “An introduction to Statistical Learning” finns tillgänglig.

### **3.3.2 Optimeringsalgoritm - *Gradient descent***

Inom maskininlärning tränar vi modeller där målet med träningen är att hitta optimala parametrar för modellerna. Om vi tänker oss den enkla linjära regressionsmodellen, för att ha ett konkret exempel, så utgår den modellen ifrån att data följer sambandet enligt Ekvation 3.11

$$y = \theta_0 + \theta_1 x_1 + \epsilon \quad (3.11)$$

där  $\theta_0$  och  $\theta_1$  är fixa men okända parametrar och  $\epsilon$  är en slumpmässig felterm.

Med hjälp av data,  $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$ , så försöker vi skatta de fixa men okända parametrarna  $\theta_0$  och  $\theta_1$ . För att göra detta definierar vi en *loss function / cost function / penalty function* (alla tre termer används) som är *mean squared error* (MSE) för den linjära regressionsmodellen. Mer specifikt har vi:

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2 \quad (3.12)$$

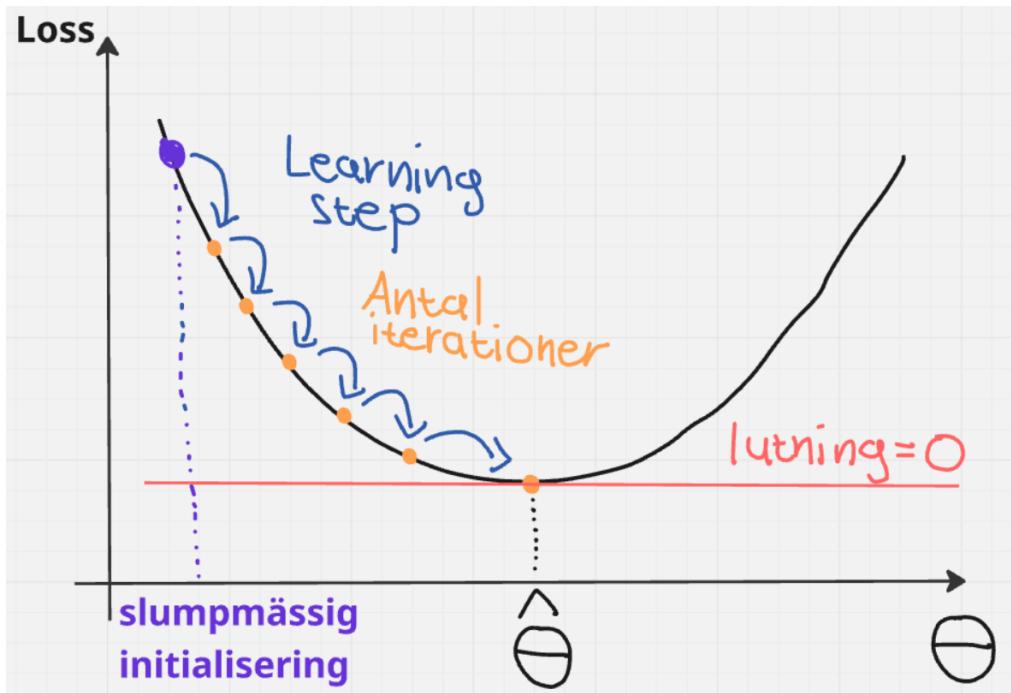
där  $y_i$  är sanna observerade värden och  $\hat{y}_i = \hat{\theta}_0 + \hat{\theta}_1 x_i$  är motsvarande prediktioner för dessa värden. Att vi ”lägger på en hatt”,  $\hat{\theta}_0$  och  $\hat{\theta}_1$ , visar att det är skattade parametrar.

Hur skattar vi nu vad  $\hat{\theta}_0$  och  $\hat{\theta}_1$  är? Svaret är enkelt och väldigt rimligt. Välj de  $\hat{\theta}_0$  och  $\hat{\theta}_1$  som minimerar *loss function*, det vill säga välj de  $\hat{\theta}_0$  och  $\hat{\theta}_1$  som minimerar MSE i Ekvation 3.12. I ord, så tränas alltså ML-modellerna genom att de försöker lära sig de parametrar som gör att felet mellan de sanna värdena,  $y_i$  och de predikterade värdena  $\hat{y}_i$ , blir så litet som möjligt. Själva minimeringen av *MSE* sker genom en algoritm som heter *gradient descent*. Den algoritmen är central och används väldigt mycket inom ML. Själva arbetet och de matematiska detaljerna sköter bibliotek såsom *scikit-learn* åt oss, men vi kommer nu ändå gå igenom algoritmen så att vi får en intuition för hur den fungerar.

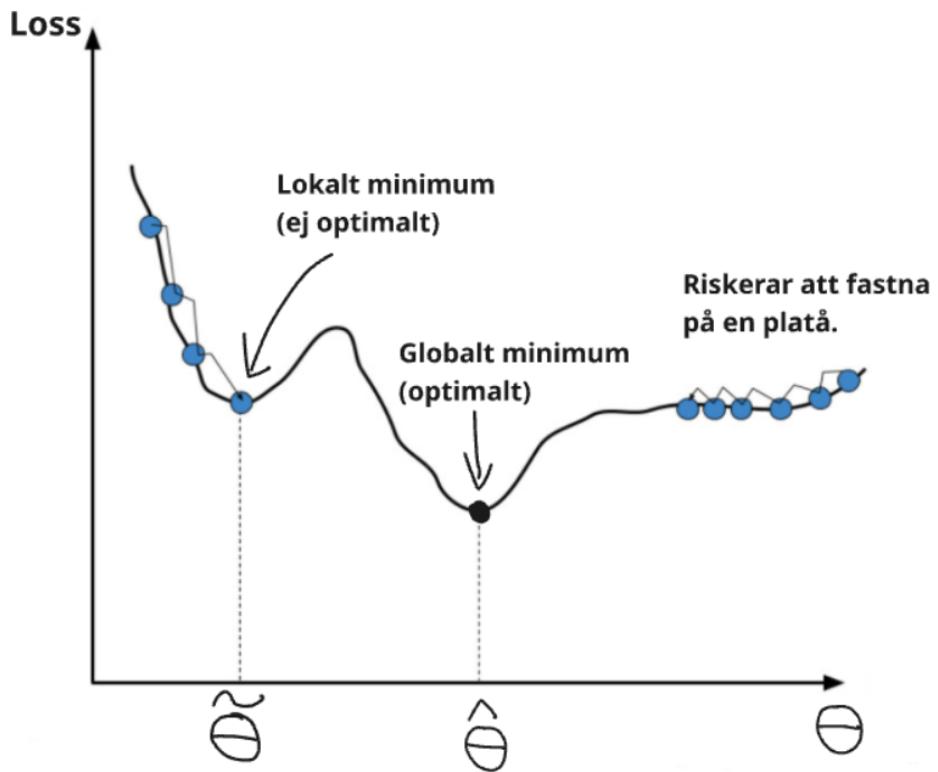
*Gradient descent* är en optimeringsalgoritm som används för att hitta en funktions minimum eller maximum. I exemplet ovan med parameterskattning ville vi hitta de värden för  $\hat{\theta}_0$  och  $\hat{\theta}_1$  som minimerade *MSE*. Intuitionen för algoritmen är att i de fall vi vill hitta minimum för en funktion så går vi i den riktningen som har en negativ lutning och fortsätter gå tills lutningen är 0. Se Figur 3.3.

- För att hitta minimum måste vi börja någonstans. Det sker genom en *slumpmässig initialisering* som representeras med den lila punkten längst upp till vänster i figuren.
- För att hitta minimum går vi i den riktningen där lutningen är negativ. Vi behöver dock bestämma hur stora steg vi ska ta, det benämns *learning step* eller *learning rate* och syns också i figuren.
- Vi behöver också bestämma hur många steg, vi som max, är villiga att ta. Det benämns *antal iterationer* och representeras av de orange prickarna i figuren.
- Minimipunkten för *loss-funktionen*, *MSE* i vårt fall, är där den optimala parametern finns. En minimipunkt har alltid lutningen 0 vilket representeras av den röda linjen i figuren. Där minimipunkten är så ser vi på x-axeln den optimala parametern som vi benämner  $\hat{\theta}$ .

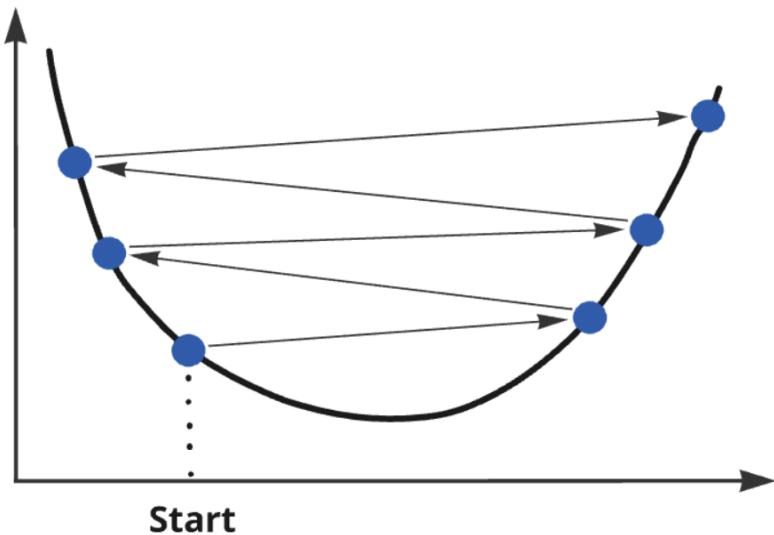
Det går matematiskt att visa att *MSE*, i fallet för linjär regression, är det som benämns för en *konvex funktion* och sådana funktioner har endast det som benämns för ett globalt minimum. Så när ett minimum väl hittas vet vi att det är det optimala. Rent generellt behöver det dock inte vara så. I Figur 3.4 ser vi att algoritmen kan hamna i lokala optimum vilket leder till icke-optimala parametrar som vi betecknar med tilda symbolen,  $\tilde{\theta}$ . Algoritmen riskerar även att fastna på en platå om exempelvis för små steg och/eller för få iterationer görs. Det finns även en risk att för stora steg tas (*learning step / learning rate*) vilket då leder till att algoritmen hoppar omkring optimumet. Se Figur 3.5. I Figur 3.6 ser vi hur algoritmen påverkas av olika *learning rates*.



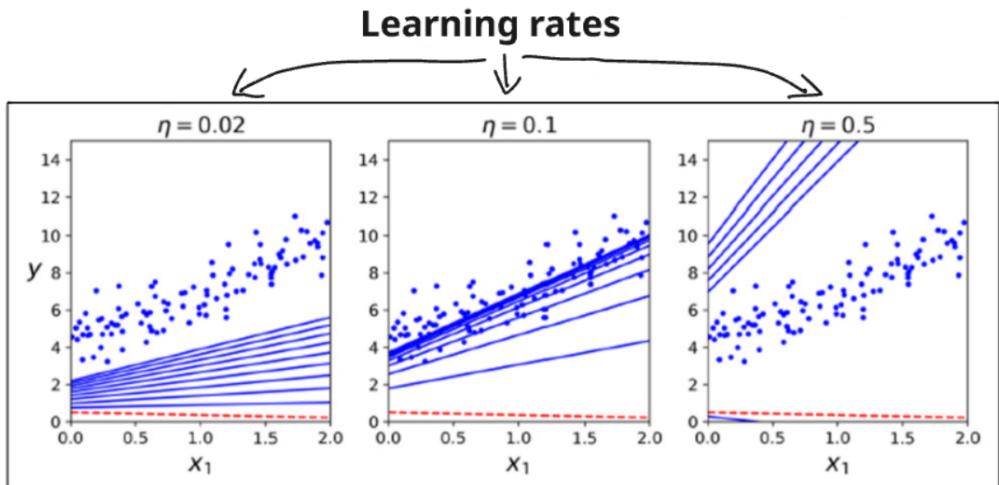
Figur 3.3: Gradient descent algoritmen. För att hitta en funktions minimum går vi i den riktningen där lutningen är negativ och stannar när lutningen är 0. Algoritmen börjar gå från den slumpmässiga initialiseringen, storleken på stegen bestäms av "learning step" och max antal steg som gåras ser vi genom "antal iterationer".



Figur 3.4: När gradient descent tillämpas finns det en risk att algoritmen hittar lokalt minimum,  $\tilde{\hat{\theta}}$  istället för globalt minimum,  $\hat{\theta}$ , vilket inte är optimalt. Algoritmen kan också fastna på en platå.



Figur 3.5: Tas för stora steg i gradient descent algoritmen riskerar det leda till att de optimala parametrarna inte nås.

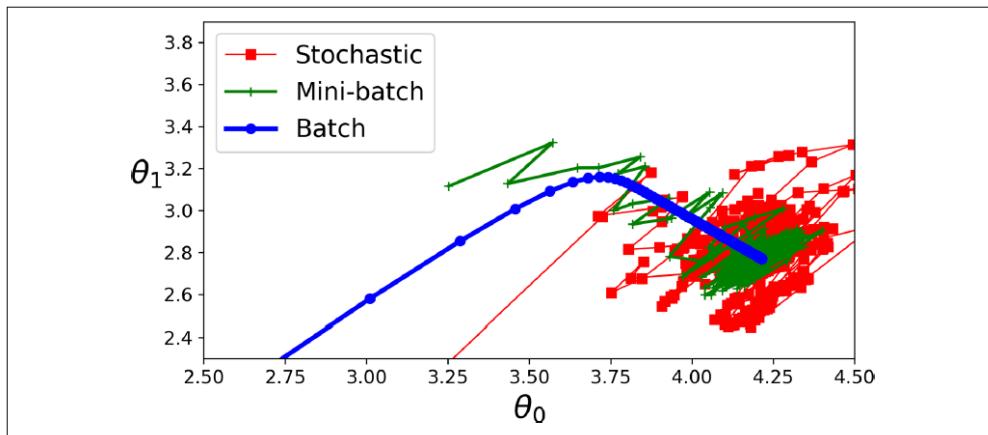


Figur 3.6: Här ser vi den påverkan som olika learning step / learning rates har på gradient descent algoritmen.

Det finns olika varianter av *gradient descent*. Vi kollar på tre sådana.

1. *Batch gradient descent*. Denna variant använder hela träningsdata för varje iteration vilket innebär att det kan ta lång tid innan algoritmen är klar.
2. *Stochastic gradient descent*. Denna variant väljer slumpmässigt en observation i taget vilket medför att den generellt sett är snabbare än *batch gradient descent*. Den är även mer irreguljär vilket kan vara en fördel för att ta sig ur ett lokalt minimum.
3. *Mini batch gradient descent*. Denna variant väljer slumpmässigt en mindre ”batch” av data för varje iteration. Det är därför en blandning av *Batch gradient descent* och *Stochastic gradient descent*.

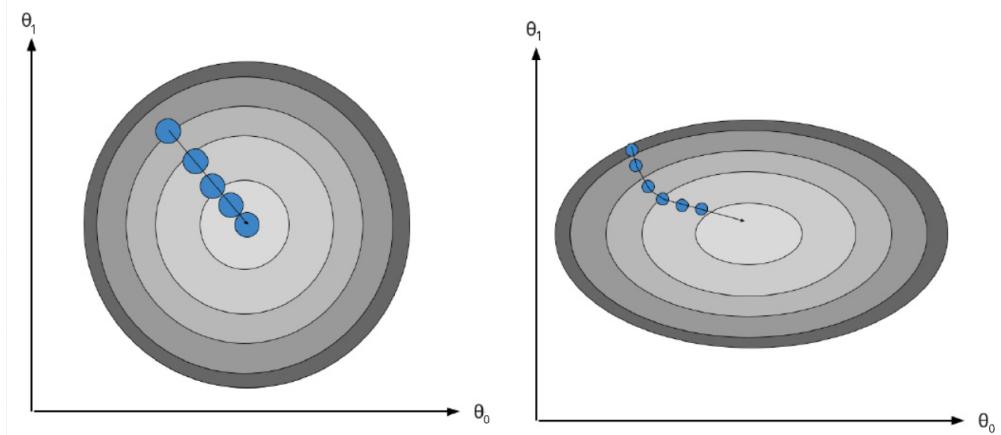
För en illustration av de tre algoritmerna, se Figur 3.7.



Figur 3.7: Illustration av tre varianter av gradient descent.

Tidigare i boken, Avsnitt 1.3.7, lärde vi oss om **StandardScaler** som standardiserade datan. En av anledningarna till att detta görs är för att det blir lättare för *gradient descent* algoritmen att nå optimum. I Figur 3.8, vänstra bilden har de två variablerna samma skala och vi ser hur algoritmen går ”direkt mot optimum”. I den högra bilden har de två variablerna olika skala och algoritmen behöver gå en krokigare väg mot optimum. I praktiken kan denna ”krokigare väg” alltså leda till att optimum inte nås, exempelvis för att det kräver fler iterationer.

Namnet *gradient descent* kommer från det faktum att algoritmen försöker hitta optimum där lutningen är noll. För att se var lutningen är noll kan konceptet om *derivatan*



Figur 3.8: Till vänster, gradient descent med feature scaling och till höger gradient descent utan feature scaling. Att algoritmen lättare kan hitta optimum när variablerna skalats/standardiseras är en av anledningarna till att detta ofta görs i samband med ML-modellering, exempelvis med `StandardScaler` från scikit-learn.

användas (något man lär sig i kursen matematik 3 på gymnasiet). En generalisering av derivatan för funktioner av fler variabler benämns *gradienten* och därav namnet.

Avslutningsvis, *gradient descent* är en viktig algoritm vars namn vi kan se i olika sammanhang när vi exempelvis läser dokumentation. Vi behöver dock, generellt sett, inte tänka på det när vi arbetar då datorn via bibliotek såsom *scikit-learn* automatiskt gör jobbet åt oss. Det är dock bra att ha en intution för hur det fungerar vilket vi nu har fått.

### 3.3.3 *The bias variance trade-off*

Ett viktigt teoretiskt resultat från statistik och maskininlärning är det som benämns *the bias variance trade-off*. Det går att matematiskt bevisa resultatet men det är överkurs och inget vi behöver i denna bok.

*The bias variance trade-off* säger att felet en modell gör när den ska prediktera ny, osedd data kan delas in i tre delar. Dessa tre delarna är:

1. *Bias*. Detta fel beror på felaktiga modellantaganden. Exempelvis kanske vi antar att datan är linjär men i verkligheten är den kvadratisk. Om vi då använder en

vanlig enkel linjär regressionsmodell ( $\hat{y} = \hat{\theta}_0 + \hat{\theta}_1 x_1$ ) istället för en polynomregression med kvadratiska termer ( $\hat{y} = \hat{\theta}_0 + \hat{\theta}_1 x_1 + \hat{\theta}_2 x_1^2$ ) får vi alltså bias. En modell med hög bias kan leda till underanpassning/*underfitting*.

2. *Variance*. Detta fel beror på att modellen är känslig för fluktuationer i tränings-datan när den tränas. Exempelvis, har vi väldigt flexibla modeller, till exempel en hög-gradig polynomregression kan små ändringar i datan leda till att modellen ser väldigt annorlunda ut vilket gör att den har hög varians. En modell med hög varians kan leda till överanpassning/*overfitting*.
3. Irreducerbart fel. Detta är fel som beror på att det helt enkelt finns slumpmässighet i datan, sådana fel är irreducibla och inget vi direkt kan göra något åt.

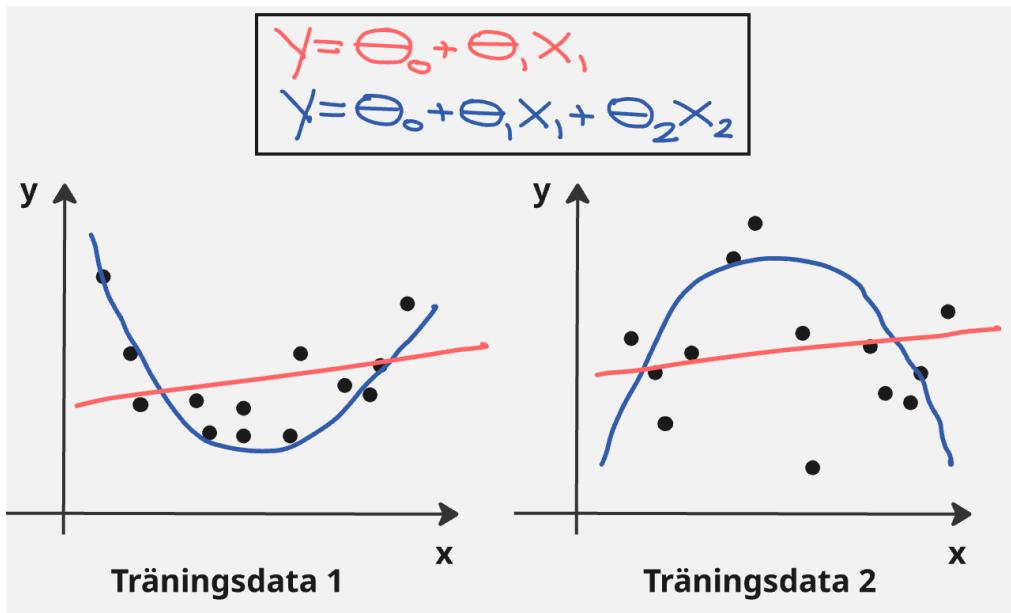
När vi ökar en modells komplexitet, exempelvis går från vanlig enkel linjär regression till polynomregression, ökar vi generellt sett dess varians men minskar dess bias och vice versa när vi minskar en modells komplexitet, då minskar vi generellt sett dess varians men ökar dess bias. Det är därför inte alltid bättre med en mer komplex modell, ökningen i varians kan kosta mer än vad minskningen i bias smakar. Det är anledningen till att det benämns en *trade-off*. Sammanfattningsvis, *the bias variance trade-off* är ett viktigt resultat som hjälper oss att förstå varför mer komplexa modeller inte alltid är bättre. Se även Figur 3.9 för en grafisk illustration.

Härnäst kommer vi lära oss om *ridge regression*, *lasso regression* samt *elastic net*. Tre modeller som är en modifiering av den linjära regressionsmodellen där vi inför ytterligare en begränsning, det sägs att vi *regularisera*r modellen. Anledningen till att vi kan vilja regularisera en modell, det vill säga begränsa den, beror helt enkelt på att vi då minskar variansen. Modellen blir mindre känslig för fluktuationer på träningsdata. Priset vi får betala för den lägre variansen blir generellt sett högre bias men minskningen i varians hoppas vi då blir större än vad ökningen i bias blir. *The bias variance trade-off* förklarar alltså varför vi kan vilja regularisera en modell, dels i fallen *ridge regression*, *lasso regression* samt *elastic net* som vi nu kommer att titta på men även för andra modeller som vi kommer lära oss längre fram, exempelvis beslutsträd.

### 3.3.4 Ridge regression (L2-regularisering)

*Ridge regression* är en regularisera variant av den linjära regressionsmodellen där överanpassning motverkas genom att minska värdet på vikterna  $\hat{\theta}_i$ . Genom att minska värdet på vikterna och därmed begränsa modellen så blir den mindre flexibel, något som ökar bias men minskar variansen. Förhoppningen är alltså då att minskningen i varians blir större än vad ökningen i bias blir.

För den linjära regressionsmodellen, såg vi i Avsnitt 3.3.2, att de optimala paramet-



Figur 3.9: I figuren ser vi två dataset, "träningsdata 1" till vänster och "träningsdata 2" till höger. Vi ser även två modeller där den röda är en vanlig enkel linjär regression,  $\hat{y} = \hat{\theta}_0 + \hat{\theta}_1 x_1$ , och den blåa är en polynomregression,  $\hat{y} = \hat{\theta}_0 + \hat{\theta}_1 x_1 + \hat{\theta}_2 x_1^2$ . Vi ser hur den mer komplexa modellen (blå) är mer flexibel och ändras mycket när träningsdatan ser annorlunda ut. Jämför vi det med den röda modellen är den mer stabil. Sammantaget har den blåa modellen högre varians och lägre bias jämfört med den röda modellen. För att veta vilken modell som faktiskt är bättre hade vi behövt utvärdera dem på valideringsdata.

rarna hittades genom att minimera kostnadsfunktionen  $MSE$ . Betecknar vi kostnadsfunktionen med  $J(\theta)$  så var den alltså  $J(\theta) = MSE(\theta) = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$ . För *ridge* regression lägger vi till en straffterm,  $\alpha \frac{1}{2} \sum_{i=1}^p \theta_i^2$  (denna straffterm benämns ofta för L2-regularisering), innebärande att kostnadsfunktionen blir enligt Ekvation 3.13

$$J(\theta) = MSE(\theta) + \alpha \frac{1}{2} \sum_{i=1}^p \theta_i^2 \quad (3.13)$$

där  $\alpha$  är regulariseringssstyrkan. Ju högre värdet på  $\alpha$  är desto starkare blir krympningen av  $\hat{\theta}_i$  vikterna. För att välja ett värde på  $\alpha$  så använder vi generellt sett *grid search* i *scikit-learn*. I Figur 3.10 ser vi effekten av  $\alpha$ . I den högra bilden där en polynom modell används ser vi tydligt hur en ökning i  $\alpha$  gör att modellen blir mindre flexibel.

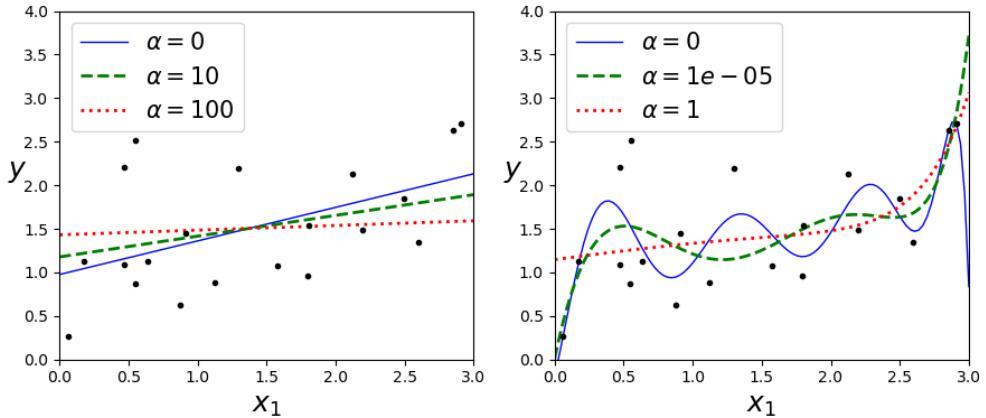
För att intuitivt förstå Ekvation 3.13 så kan vi se det som att flexibla vikter som tillåts bli större kan leda till att  $MSE$  minskar, men det leder också till att den extra strafftermen  $\alpha \frac{1}{2} \sum_{i=1}^p \theta_i^2$  ökar. För att det då ska vara värt att öka storleken på våra vikter så måste minskningen i  $MSE$  bli större än vad ökningen i det extra straffet blir. Vi ser också att det extra straffet blir större ju större  $\alpha$  är vilket leder till att det i färre fall är värt att öka storleken på vikterna, något som alltså krymper dem ner mot 0 i högre utsträckning.

Vi demonstrerar nu modellen med ett kodexempel där vi använder oss av datasetet skapat med `make_regression`.

```
from sklearn.linear_model import Ridge
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import root_mean_squared_error

hyperparams = {'alpha': [0.1, 1, 10]}
ridge = Ridge()
ridge_gs = GridSearchCV(estimator=ridge, param_grid=hyperparams,
    scoring='neg_mean_squared_error', cv= 5)
ridge_gs.fit(X_train, y_train)                                ①

print("Optimized value for alpha:", ridge_gs.best_estimator_) ②
print("Coefficients for ridge regression:",
    np.round(ridge_gs.best_estimator_.coef_, 5))                ③
```



Figur 3.10: I figuren ser vi effekten av olika värden på hyperparametern  $\alpha$ , som styr regulariseringssyrkan, för ridge-regularisering. I den vänstra bilden använder vi en linjär modell och i den högra bilden använder vi en polynom modell.

```

print("Coefficients for linear regression:",
      np.round(lin_reg.coef_, 5))

y_pred = ridge_gs.predict(X_test)
rmse_ridge = root_mean_squared_error(y_test, y_pred)

print("RMSE for Ridge Regression:", np.round(rmse_ridge, 4))
print("RMSE for Linear Regression:", np.round(RMSE_lin_reg, 4))

```

- ① Vi genomför en *grid search* där olika värden på regulariseringssyrkan  $\alpha$  utvärderas.
- ② Vi skriver ut det optimerade värdet på  $\alpha$  som hittats med hjälp av *grid search*.
- ③ Vi skriver ut  $\hat{\theta}_i$  vikterna från *ridge* regressionen.
- ④ Vi skriver ut  $\hat{\theta}_i$  vikterna från den vanliga linjära regressionsmodellen från Avsnitt 3.3.1.

```

Optimized value for alpha: Ridge(alpha=0.1)
Coefficients for ridge regression: [38.99838  1.29336 89.84231]
Coefficients for linear regression: [38.99942  1.29342 89.84458]
RMSE for Ridge Regression: 5.0971

```

RMSE for Linear Regression: 5.0973

Från resultatet ser vi att det optimerade värdet på  $\alpha$  var relativt litet innehållande att det blev en liten regularisering som gjordes. Detta har effekten att vikterna,  $\hat{\theta}_i$ , endast krympas lite för *ridge* regression i förhållande till vikterna för den vanliga linjära regressionsmodellen.

### 3.3.5 *Lasso* regression (L1-regularisering)

Lasso (*Least Absolute Shrinkage and Selection Operator*) är en regulariserad variant av linjär regression vars kostnadsfunktion är enligt Ekvation 3.14.

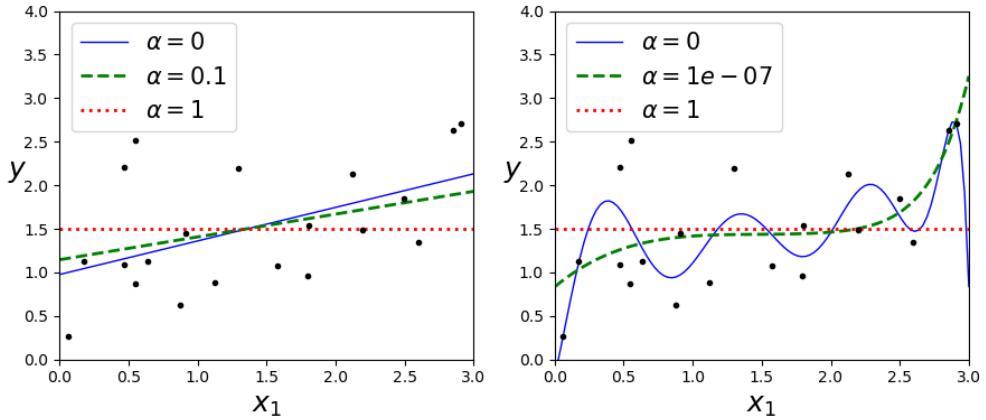
$$J(\theta) = MSE(\theta) + \alpha \sum_{i=1}^p |\theta_i| \quad (3.14)$$

Den extra kostnadstermen,  $\alpha \sum_{i=1}^p |\theta_i|$  benämns ofta för L1-regularisering och det går att matematiskt bevisa att införandet av den tenderar att sätta vikterna för mindre viktiga variabler till 0. Det kan jämföras med *ridge* regression där vikterna generellt sett krympes så de närmade sig 0. Det faktum att *lasso* regression alltså tenderar till att sätta vikterna för mindre viktiga variabler till 0 innehåller att den på ett automatiserat sätt genomför *variabelselektion* eller *feature selection*. Detta kan exempelvis vara användbart i de fall vi har ett dataset med väldigt många variabler (ett högdimensionellt dataset) och vi misstänker att några av dem kan vara oviktiga för att prediktera den beroende variabeln  $y$ .

#### i Intuition för *lasso* regression

En intuitiv förklaring på hur L1-regularisering fungerar ges med följande exempel. Antag att vi har en budget på hur stora vikterna för våra oberoende variabler får vara. Det kommer då innehålla att vissa variabler som inte är tillräckligt användbara då är bättre att sätta till noll för att spara budgeten till de variabler som i högre utsträckning bidrar till att vi kan genomföra bra prediktioner. Det innehåller alltså att features som anses vara "starka" överlever och övriga sätts till noll.

I Figur 3.11 ser vi effekten av  $\alpha$ . I den högra bilden där en polynom modell används ser vi tydligt hur en ökning i  $\alpha$  gör att modellen blir mindre flexibel.



Figur 3.11: I figuren ser vi effekten av olika värde på hyperparametern  $\alpha$ , som styr regulariseringsstyrkan, för lasso-regularisering. I den vänstra bilden använder vi en linjär modell och i den högra bilden använder vi en polynom modell.

Vi demonstrerar nu *lasso regression* med ett kodexempel där vi använder oss av datasetet skapat med `make_regression`.

```

from sklearn.linear_model import Lasso

lasso_reg = Lasso(alpha=9)
lasso_reg.fit(X_train, y_train)
y_pred_lasso = lasso_reg.predict(X_test)

RMSE_lasso = root_mean_squared_error(y_test, y_pred_lasso)

print("Coefficients for lasso regression:",
      np.round((lasso_reg.coef_), 2))
print("RMSE for lasso regression:", np.round(RMSE_lasso))

```

- ① Normalt sett optimerar vi  $\alpha$  med hjälp av *grid search* men här har vi själv satt ett värde som i detta sammanhanget är relativt stort. Detta för att demonstrera hur det sätter variabler som modellen anser är mindre viktiga till 0.
- ② Vi skriver ut  $\hat{\theta}_i$  vikterna från *lasso* regressionen.

```
Coefficients for lasso regression: [29.78  0.   80.6 ]
RMSE for lasso regression: 13.0
```

Från resultatet ser vi den andra vikten satts till 0. På grund av att vi manuellt satt `alpha=9` för att demonstrera hur vikter kan dras ned till 0 så ser vi också att *RMSE* i detta fall ökade avsevärt jämfört med tidigare exempel för de andra modellerna.

### 3.3.6 *Elastic net* (kombination av *Lasso & Ridge*)

*Elastic Net* är en kombination av *lasso* (L1) och *ridge* (L2) regularisering. Den balanserar båda teknikerna med hjälp av en *mix ratio*,  $r$ , dess kostnadsfunktion är enligt Ekvation 3.15.

$$J(\theta) = MSE(\theta) + r\alpha \sum_{i=1}^p |\theta_i| + \frac{1-r}{2}\alpha \sum_{i=1}^p \theta_i^2. \quad (3.15)$$

Vi ser att när *mix ratio*,  $r$ , är satt till noll är kostnadsfunktionen för *elastic net* samma som för *ridge* regression. Men om  $r$  är satt till 1 är den istället samma som för *lasso* regression. Denna kombination gör det möjligt för *Elastic Net* att välja viktiga *features* (som *lasso*) samtidigt som den kan ha kvar några små vikter som inte är noll (som *ridge*).

Se Tabell 3.3 där två hyperparametrar för *elastic net* presenteras. Läsaren uppmanas att läsa scikit-learn dokumentationen för att få en helhetsbild.

Tabell 3.3: Ett urval av hyperparametrar för elastic net från scikit-learns dokumentation.

Hyperparameter	Beskrivning	Standardvärde
<code>alpha</code>	Är ett icke-negativt värde mellan 0 till oändligheten som styr styrkan av regulariseringen. Av typen float.	1
<code>l1_ratio</code>	Kontrollerar mixen av L1- och L2-regularisering (0 = Ridge, 1 = Lasso). Av typen float med ett värde mellan 0 och 1.	0.5

Vi demonstrerar nu *elastic net* med ett kodexempel där vi använder oss av datasetet skapat med `make_regression`.

```
from sklearn.linear_model import ElasticNet

elastic_net = ElasticNet()
hyperparams = {
    'alpha': [0.01, 0.1, 1.0, 10.0],
    'l1_ratio': [0.1, 0.5, 0.7, 0.9, 1.0]}
elastic_net_gs = GridSearchCV(estimator=elastic_net,
    param_grid=hyperparams,
    scoring='neg_mean_squared_error', cv=5)
elastic_net_gs.fit(X_train, y_train)

print("Optimized values from grid search:",
    elastic_net_gs.best_estimator_)
print("Coefficients for elastic net:",
    np.round(elastic_net_gs.best_estimator_.coef_, 5))

y_pred = elastic_net_gs.predict(X_test)
rmse_elastic_net = root_mean_squared_error(y_test, y_pred)

print("RMSE for Elastic Net:", np.round(rmse_elastic_net, 4))
```

```
Optimized values from grid search: ElasticNet(alpha=0.01, l1_ratio=1.0)
Coefficients for elastic net: [38.98905  1.28335 89.83427]
RMSE for Elastic Net: 5.097
```

### 3.3.7 *Support vector machines (SVM)*

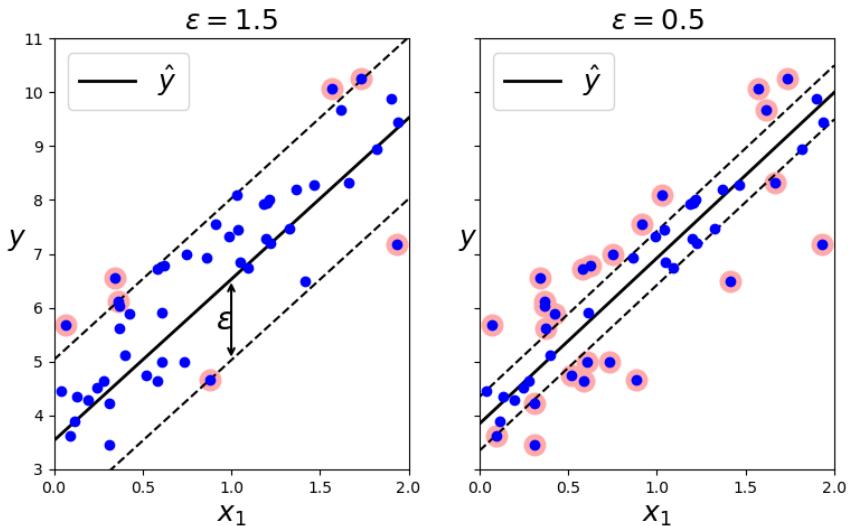
*Support Vector Machines (SVM)* är en modell som kan användas både för regressionsproblem och för klassificeringsproblem. I detta avsnitt kommer vi ge en överblick över hur den fungerar för regressionsproblem och vissa aspekter kommer vi gå in på i mer detalj när vi går igenom den för klassificeringsproblem i Avsnitt 4.3.2. Det är därför en god idé att läsa igenom innevarande avsnitt och sen komma tillbaka igen efter att Avsnitt 4.3.2 lästs igenom.

För regressionsproblem försöker *SVM* modellen skapa en väg, med en given bredd, som innehåller så många observationer som möjligt. Mitten-linjen i vägen är det som blir

modellens predikterade värden,  $\hat{y}$ . Se Figur 3.12. Från figuren ser vi att vägens bredd styrs av hyperparametern  $\epsilon$ . Samtliga observationer som är innanför vägen, det vill säga prediktionsfel som är mindre än  $\epsilon$ , ignoreras av modellen innehållande att de inte påverkar modellens parametrar i samband med träning. På engelska säger man att modellen är  $\epsilon$ -insensitive. Detta har följande implikationer:

- Om vi minskar värdet på  $\epsilon$  så påverkar träningsdatan modellen mer. Om vi vill ha en mer flexibel modell (exempelvis för att den kan vara underanpassad) minskar vi alltså värdet på  $\epsilon$ .
- Om vi ökar värdet på  $\epsilon$  så påverkar träningsdatan modellen mindre. Om vi vill ha en mindre flexibel modell (exempelvis för att den kan vara överanpassad) ökar vi alltså värdet på  $\epsilon$ .

I praktiken så väljer vi ett värde på  $\epsilon$  med hjälp av *grid search*.



Figur 3.12: Visualisering av hur  $\epsilon$  påverkar SVM modellen.

Generellt sett är det bra att standardisera data när vi använder *SVM*-modeller. Modellerna är också generellt sett lämpliga att använda för små och medelstora dataset eftersom de kan ta för lång tid att träna på större dataset. För *SVM*-modeller finns det något som benämns för *kernel trick*. Detaljer är överkurs men vi kommer att gå djupare in på och demonstrera *the kernel trick* i Avsnitt 4.3.2. I korthet räcker det att

veta att det ger samma effekt som att lägga till transformerade variabler (exempelvis polynomtermer) utan att det faktiskt behöver göras. Det leder till att vi kan dra nytta av effekterna utan att behöva lägga till variabler vilket exempelvis hade påverkat tiden det tar att träna modellen.

I Tabell 3.4 presenteras ett urval av hyperparametrar för *SVR* (*Support Vector Regression*). Läsaren uppmanas att läsa scikit-learn dokumentationen för att få en helhetsbild.

*Tabell 3.4: Ett urval av hyperparametrar för SVR (Support Vector Regression) från scikit-learns dokumentation.*

Hyperparameter	Beskrivning	Standardvärde
C	Regulariseringssvärde som måste vara större än 0, är av typen float.	1.0
epsilon	Påverkar bredden på vägen. De observationer innanför vägen påverkar inte träningen av modellen.	0.1
kernel	Anger typen av <i>kernel</i> som ska användas i algoritmen, <code>{'linear', 'poly', 'rbf', 'sigmoid', 'precomputed'}</code> eller callable.	'rbf'
gamma	<i>Kernel</i> koefficient för 'rbf', 'poly' och 'sigmoid', <code>{'scale', 'auto'}</code> eller float.	'scale'

I *scikit-learn* finns klassen `LinearSVR` och `SVR`. Den senare används om vi vill använda oss av en *kernel*, det gör vi dock inte i detta avsnitt och i kodexemplet nedan demonstrerar vi `LinearSVR` på datasetet skapat med `make_regression`.

```
from sklearn.svm import LinearSVR
from sklearn.pipeline import Pipeline
```

```

from sklearn.preprocessing import StandardScaler

svr_pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('svr', LinearSVR())
])
①

hyperparams = {'svr__epsilon': [0.01, 0.1, 1.0]} ②

svr_grid_search = GridSearchCV(estimator=svr_pipeline,
    param_grid=hyperparams, scoring='neg_mean_squared_error')

svr_grid_search.fit(X_train, y_train)
y_pred = svr_grid_search.predict(X_test)
rmse_svr = root_mean_squared_error(y_test, y_pred)

print("Optimized SVR pipeline:", svr_grid_search.best_estimator_)
print("Root Mean Squared Error (RMSE):", rmse_svr)

```

- ① Vi skapar en *pipeline* som först skalar datan innan modellen tränas.  
 ② Vi specificerar vilka värden på  $\epsilon$  vi vill undersöka i en *grid search*.

```

Optimized SVR pipeline: Pipeline(steps=[('scaler', StandardScaler()), ('svr', LinearSVR(epsilon=1.0))])
Root Mean Squared Error (RMSE): 5.086958033549762

```

### 3.3.8 Beslutsträd

I detta avsnitt kommer vi kolla på beslutsträd, en modell som på engelska benämns *decision tree*. Modellen kan användas för både regressionsproblem och klassificeringsproblem.

För att förstå hur modellen genomför prediktioner kan vi kolla på Figur 3.13. Modellen kommer gå genom trädet från den översta noden, som benämns *rotnod*, och vidare genom *inre noder* för att slutligen komma till de yttersta noderna som benämns *lövnoder*. Antag nu att vi har en observation med värdet  $x_1 = -1$  och  $x_2 = 0.55$ . För att genomföra en prediktion så ser vi från figuren att följande steg görs:

- Frågan “är  $x_2 \leq 0.438$  ?” ställs. Om svaret är sant så går vi till vänster, annars till höger. Eftersom vi antar att  $x_2 = 0.55$  är svaret falskt och vi går alltså till

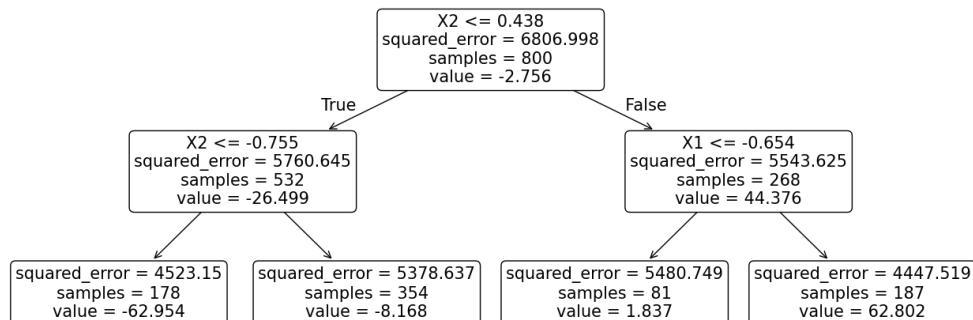
höger.

2. Nu ställs frågan “är  $x_1 \leq -0.654$  ?”. Eftersom vi antar att  $x_1 = -1$  är svaret sant och vi går alltså till vänster.
3. I lövnoden ser vi att det står *value* = 1.837 och det blir det predikterade värdet för observationen med  $x_1 = -1$  och  $x_2 = 0.55$ .

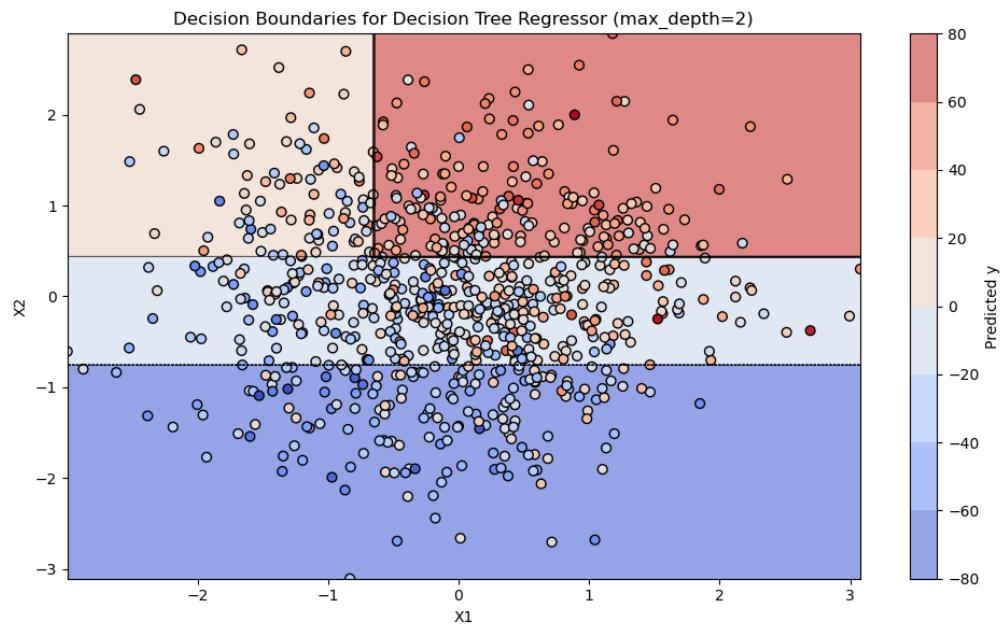
I varje nod står det *samples* och det visar hur många observationer det finns i respektive nod. Med *squared error* menas *MSE*.

I frågan “ $x_2 \leq 0.438$ ” är  $x_2$  den oberoende variabeln som används och 0.438 är tröskelvärdet. Hur väljer modellen vilken oberoende variabel respektive tröskelvärde som ska användas i varje fråga? Svaret är att den väljer den variabel och det tröskelvärde som gör att *MSE (squared error)* blir så lågt som möjligt i respektive steg. Praktisk implementering sköter *scikit-learn* automatiskt åt oss.

Kollar vi på Figur 3.14 så ser vi hur den korresponderar mot Figur 3.13. Värdena  $x_2 = 0.438$ ,  $x_2 = -0.755$  och  $x_1 = -0.654$  från den första figuren ser vi är linjerna i den andra figuren. Vi ser också hur det bildas fyra olika rutor i den andra figuren, alla observationer som hamnar i en viss ruta får alltså samma predikterade värde. Prickarnas färg i Figur 3.14 visar de sanna värdena för observationerna där färgkodningen ses i den stapel som finns längst till höger i figuren.



Figur 3.13: Visualisering av ett beslutsträd. Denna typ av visualisering kan enkelt göras genom att använda `sklearn.tree.plot_tree` från scikit-learn.



Figur 3.14: En visualisering för ett beslutsträd, notera hur denna figur korresponderar med Figur 3.13.

## **i** Modelltolkning - *White box* modeller kontra *black box* modeller

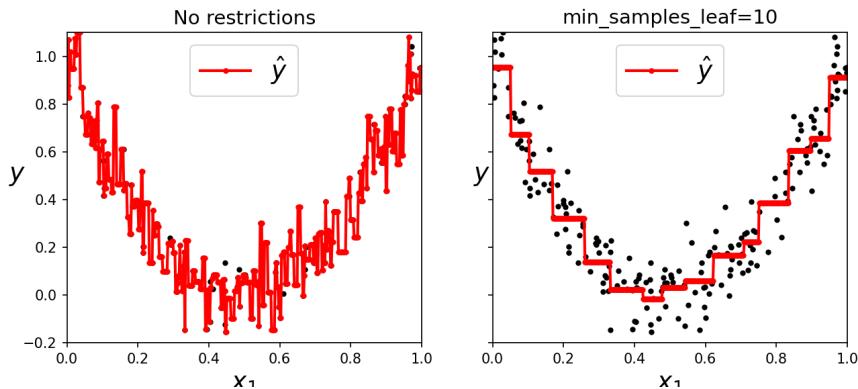
Beslutsträd är intuitiva och det är lätt att se varför modellen gör en viss prediktion. Därför kallas de ofta för *white box* modeller. För exempelvis *random forest* och neurala nätverk, två modeller vi kommer lära oss om senare i boken, kan vi rent matematiskt följa beräkningarna och se vilka prediktioner de utför men det är svårt att intuitivt förklara i ord. Denna typ av modeller benämns ofta som *black box* modeller.

Exempelvis kan vi med ett beslutsträd manuellt stegar igenom och se vilken prediktion som görs. Det kan vi generellt sett inte göra med *Random Forest* och neurala nätverk.

Generellt sett så bör beslutsträd regulariseras för att de inte ska bli överanpassade. Se Figur 3.15. Exempel på hyperparametrar som kan användas för regularisering ser vi i Tabell 3.5. Läsaren uppmanas att läsa scikit-learn dokumentationen för att få en helhetsbild. Ett enkelt sätt att regularisera ett träd kan exempelvis vara genom att välja en nivå på `max_depth` som alltså styr hur djupt beslutsträdet kan bli. Trädet i Figur 3.13 har exempelvis djupet 2.

Tabell 3.5: Ett urval av hyperparametrar för beslutsträd från scikit-learns dokumentation.

Hyperparameter	Beskrivning	Standardvärde
<code>max_depth</code>	Trädets maximala djup, av typen int.	None
<code>min_samples_split</code>	Minsta antal observationer som krävs för att dela en intern nod, typen är int eller float.	2
<code>min_samples_leaf</code>	Det minsta antal observationer som behöver finnas i en lövnod, typen är int eller float.	1
<code>max_leaf_nodes</code>	Största antalet lövnoder ett träd får ha. Av typen int.	None



Figur 3.15: I den vänstra bilden ser vi ett beslutsträd som är överanpassat, det följer träningsdata för noggrant för att det ska kunna generalisera till ny, osedd data på ett bra sätt. Den högra bilden visar på ett beslutsträd som blivit regulariserat och som ser bättre ut. Generellt sett bör beslutsträd regulariseras.

I kodexemplet nedan demonstrerar vi `DecisionTreeRegressor` på datasetet skapat med `make_regression`.

```
from sklearn.tree import DecisionTreeRegressor

dec_tree = DecisionTreeRegressor(random_state=42)

hyperparams = {
    'max_depth': [2, 3, 5, 10],
    'min_samples_split': [2, 5, 10]
}
dec_tree_gs = GridSearchCV(estimator=dec_tree,
                           param_grid=hyperparams, scoring='neg_mean_squared_error')
dec_tree_gs.fit(X_train, y_train)

y_pred = dec_tree_gs.predict(X_test)
rmse_tree = root_mean_squared_error(y_test, y_pred)

print("Optimized hyperparameters with grid search:",
      dec_tree_gs.best_params_)
```

```
print("RMSE on test set:", rmse_tree)

Optimized hyperparameters with grid search: {'max_depth': 10,
'min_samples_split': 2}
RMSE on test set: 8.958522839814108
```

### 3.3.9 Ensemble learning

*Ensemble learning* handlar om att kombinera flera modeller med förhoppningen att de gemensamt kan prestera bättre än vad varje enskild modell kan göra. Vi kommer kolla på *voting regression* samt *bagging* och *pasting* som är tre olika sätt att göra detta på.

Med *voting regression* utgår vi från att vi har ett dataset. Med det datasetet kan vi skapa flera olika modeller som gör individuella prediktioner. För att få en slutgiltig prediktion tar vi medelvärdet av de individuella prediktionerna. Se Figur 3.16 för en illustration av processen.

I kodexemplet nedan demonstrerar vi hur *voting regression* genomförs, först manuellt och sedan med hjälp av `VotingRegressor` från *scikit-learn*. Vi använder datasetet skapat med `make_regression` för demonstrationen.

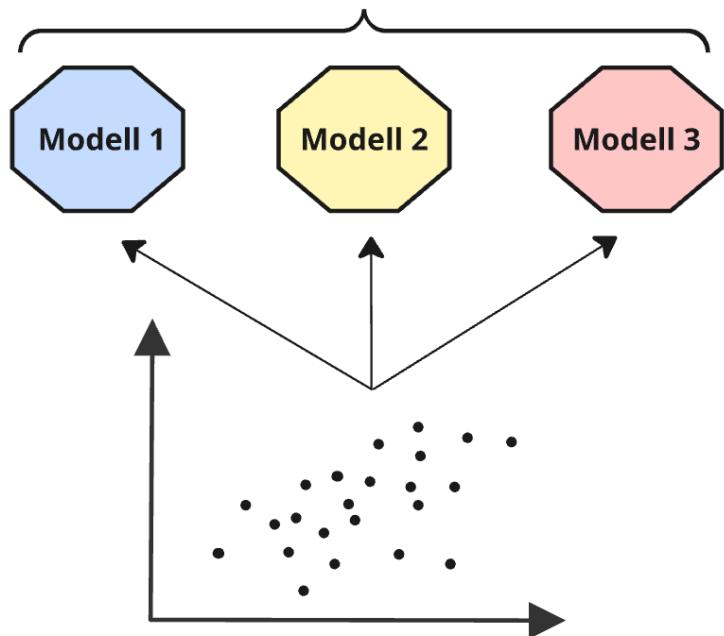
```
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeRegressor
from sklearn.svm import LinearSVR
from sklearn.ensemble import VotingRegressor

### Voting regression done manually
# Initialize regressors
m1 = LinearRegression()
m2 = DecisionTreeRegressor(max_depth=3, random_state=2)
m3 = LinearSVR(random_state=2)

# Fit each model individually
m1.fit(X_train, y_train)
m2.fit(X_train, y_train)
m3.fit(X_train, y_train)

# Predict with each model individually
```

**Ta medelvärdet av varje modells individuella prediktion för att få en slutgiltig prediktion.**



Figur 3.16: Schematisk bild över hur voting regression sker. I detta exempel har vi använt tre individuella modeller som kombinerats, rent generellt kan dock ett godtyckligt antal modeller kombineras.

```

pred1 = m1.predict(X_test)
pred2 = m2.predict(X_test)
pred3 = m3.predict(X_test)

# The ensemble prediction is obtained by averaging the individual
# predictions
avg_pred = (pred1 + pred2 + pred3) / 3
print(avg_pred[0:5])                                         ①

### Using VotingRegressor
voting_regressor = VotingRegressor([('lin_reg', m1), ('dec_tree',
    ↵ m2), ('svm', m3)])
voting_regressor.fit(X_train, y_train)
voting_regressor_pred = voting_regressor.predict(X_test)
print(voting_regressor_pred[0:5])

```

- ① Vi skriver ut de fem första prediktionerna.

```
[ -41.03912946  -18.99966521   68.61782186 -127.66927334  -24.1281992 ]
[ -41.03912946  -18.99966521   68.61782186 -127.66927334  -24.1281992 ]
```

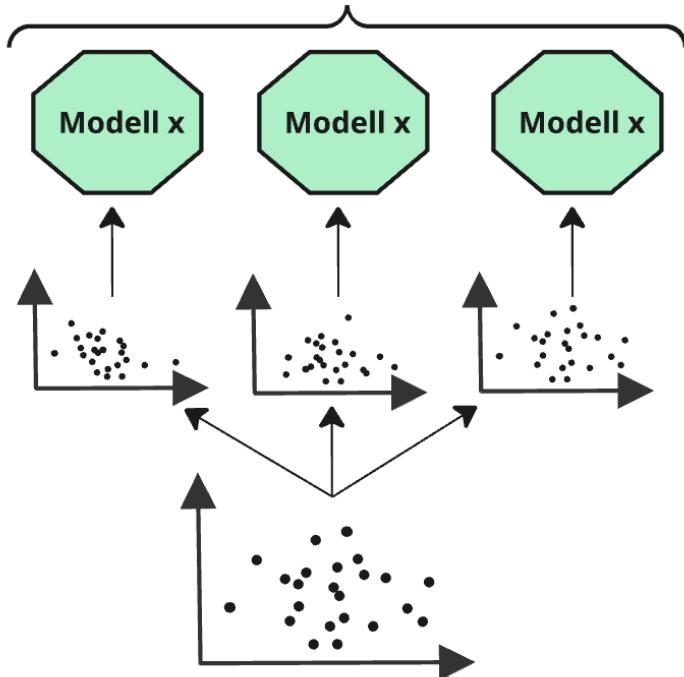
Vi ser att resultatet blir detsamma när vi gjort det manuellt respektive med `VotingRegressor`.

Nedan förklaras hur *bagging* och *pasting* fungerar, till vår hjälp har vi Figur 3.17.

1. Vi utgår ifrån att vi har ett givet dataset.
2. Från det givna datasetet kan vi skapa nya dataset genom antingen *bagging* eller *pasting*.
  - Med *bagging* gör vi ett slumpmässigt urval med återläggning (innebärande att samma observation kan förekomma flera gånger för det nya datasetet som skapas) för att skapa ett nytt dataset.
  - Med *pasting* gör vi ett slumpmässigt urval utan återläggning (innebärande att samma observation *inte* kan förekomma flera gånger för det nya datasetet som skapas) för att skapa ett nytt dataset.
3. Välj *en* modell, exempelvis en linjär regressionsmodell, och träna den på varje dataset som skapats.
4. Den slutgiltiga prediktionen fås genom att kombinera varje individuell prediktion från respektive modell genom att beräkna medelvärdet

För att genomföra *bagging* eller *pasting* använder vi oss av `BaggingRegressor` i *scikit-*

## Ta medelvärdet av varje modells individuella prediktion för att få en slutgiltig prediktion.



Figur 3.17: Schematisk bild över hur bagging och pasting fungerar. Vi har ett givet dataset som vi tar slumpmässiga urval från för att skapa nya dataset. Om urvalet sker med återläggning benämns det bagging och om urvalet sker utan återläggning benämns det pasting. Därefter kan vi välja en modell, exempelvis en linjär regressionsmodell, och träna modellen för varje dataset. I figuren hade vi alltså tränat tre linjära regressionsmodeller. När vi ska prediktera en ny observation kan vi kombinera prediktionerna från de tre modellerna genom att beräkna medelvärdet.

*learn*. Kollar vi på dokumentationen så ser vi att följande hyperparametrar finns:

```
class sklearn.ensemble.BaggingRegressor(estimator=None,  
    ↵ n_estimators=10, *, max_samples=1.0, max_features=1.0,  
    ↵ bootstrap=True, bootstrap_features=False, oob_score=False,  
    ↵ warm_start=False, n_jobs=None, random_state=None, verbose=0)
```

Från hyperparametrarna ovan kan vi bland annat notera följande:

- Koden `bootstrap=True` innebär att vi får *bagging*. Om `False` specificeras får vi *pasting*.
- `n_estimators=10` styr hur många dataset som skapas från den ursprungliga datan, och fölaktligen hur många modeller som tränas då varje skapat dataset används för att träna en modell.
- `max_samples=1.0` specificerar att varje dataset som skapas är lika stort som det ursprungliga. Man kan alltså ha att de skapade dataseten t.ex. är 80% av de ursprungliga, då hade vi skrivit `max_samples=0.8`.
- Använder vi oss av *bagging* går det matematiskt att visa att cirka 37% av observationerna aldrig kommer att bli dragna, dessa benämns *out of bag observations*. Dessa hade då kunnat användas för att utvärdera modellen. Det styrs med `oob_score=False`.
- När hela datasetet används (`bootstrap=False`) men endast en slumpmässig andel av de oberoende variablerna så kallas detta för *random subspaces*. Kan uppnås genom att specificera `max_samples=1.0` och `max_features<1.0`. Skulle vi exempelvis ha `max_features=0.5` innebär det att modellerna som tränas endast är tränade på 50% av de oberoende variablerna där valet har skett slumpmässigt för varje dataset.
- När en slumpmässig andel av datasetet och de oberoende variablerna används kallas det för *random patches*. Kan uppnås genom att specificera `max_samples<1.0` och `max_features<1.0`.

*Random subspaces* och *random patches* kan vara användbart när vi arbetar med högdimensionella dataset (det vill säga att det finns många oberoende variabler), vilket kan ske när vi exempelvis arbetar med bilder.

Vi demonstrerar nu `BaggingRegressor` genom att använda datasetet som vi skapat med `make_regression`.

```

from sklearn.linear_model import LinearRegression
from sklearn.ensemble import BaggingRegressor

bagging_reg= BaggingRegressor(estimator=LinearRegression(),
    ↵ n_estimators=15)
bagging_reg.fit(X_train, y_train)

y_pred = bagging_reg.predict(X_test)
rmse_bagging = root_mean_squared_error(y_test, y_pred)

print("RMSE on test set:", rmse_bagging)

```

RMSE on test set: 5.096743981380514

### 3.3.10 Random forest

Random forest är ett *ensemble* av beslutsträd som oftast skapas genom *bagging* även om *pasting* också förekommer. Vi hade manuellt kunnat implementera modellen med *BaggingRegressor* men det finns en specifik klass för det i *scikit-learn*.

Precis som för beslutsträd är det ofta bra att regularisera *random forest*. I koden nedan skapar vi en *random forest* modell. Vi använder datasetet som vi skapat med `make_regression`.

```

from sklearn.ensemble import RandomForestRegressor

random_forest = RandomForestRegressor(random_state=42)

hyperparams = {
    'n_estimators': [10, 50, 100, 150],
    'max_depth': [2, 3, 5, 10],
    'min_samples_split': [2, 5, 10]
}
random_forest_gs = GridSearchCV(estimator=random_forest,
    ↵ param_grid=hyperparams, scoring='neg_mean_squared_error')
random_forest_gs.fit(X_train, y_train)

y_pred = random_forest_gs.predict(X_test)

```

```
rmse_forest = root_mean_squared_error(y_test, y_pred)

print("Optimized hyperparameters with grid search:",
      random_forest_gs.best_params_)
print("RMSE on test set:", rmse_forest)
```

```
Optimized hyperparameters with grid search: {'max_depth': 10,
'min_samples_split': 2, 'n_estimators': 150}
RMSE on test set: 6.446240033273218
```

### i Extra Tree

I Avsnitt 4.3.5 kommer vi gå igenom en modell som heter *Extra Tree* som även kan användas för regressionsproblem. Modellen importeras med koden: `from sklearn.ensemble import ExtraTreesRegressor.`

## 3.4 Övningsuppgifter

Övningsuppgifter för kapitlet finns på bokens hemsida:  
[https://github.com/AntonioPrgomet/ai\\_tillaempad\\_ml](https://github.com/AntonioPrgomet/ai_tillaempad_ml)

# Kapitel 4

## Klassificering

I föregående kapitel lärde vi oss om regression där målet var att prediktera kontinuerlig utdata med hjälp av indata. I detta kapitel kommer vi att fördjupa oss inom *klassificering*, en problemkategori inom maskininlärning där målet istället är att prediktera kategorisk utdata med hjälp av indata.

Kapitlet börjar med att förklara vad klassificeringsproblem är och ger bland annat exempel på diverse tillämpningsområden. Därefter presenteras olika utvärderingsmått följt av olika klassificeringsmodeller som används för att genomföra klassificering. Kapitlet avslutas med två kodexempel.

### 4.1 Klassificeringsproblem

Klassificeringsproblem handlar om att prediktera kategorisk data där datan kan anta två eller flera klasser. Det finns flera tillämpningsområden inom olika branscher och några exempel tagna från verkligheten ges nedan.

- **Kundanalys** - Prediktera om en kund kommer att *churna* (alltså att kunden lämnar företaget) eller ej baserat på variabler såsom antal köp, ålder, kön och antal supportärenden.
- **Sjukvård** - Prediktera om en patient har en viss sjukdom eller inte (sjuk eller frisk) baserat på medicinsk data såsom blodtryck, puls och så vidare.

- **Bildigenkänning** - Identifiera objekt i en bild baserat på variabler såsom färg, form och textur. Det kan exempelvis handla om att identifiera skadedjur i jordbruksmiljöer eller på golfbanor.
- **Anomalidetektion** - Detta används bland annat inom tillverkningsprocesser i företag för att med hjälp av bilder från kameror på ett automatiserat sätt identifiera defekta produkter.
- **Finans** - Prediktera om en aktie kommer att stiga i värde eller ej baserat på exempelvis företagsinformation såsom lönsamhet och omsättning samt makroekonomisk data såsom inflation och räntor.
- **Kreditrisk** - Bedöma om en bankkund ska beviljas ett huslån eller inte baserat på data såsom inkomst, ålder, civilstånd och eventuella betalningsanmärkningar.

Den enklaste formen av klassificering, där data kan anta två klasser, heter *binär klassificering*. Om vi ska prediktera ifall en kund kommer lämna ett företag (churna) eller inte (stanna) så hade de två klasserna varit {churna, stanna}. De två klasserna kan representeras på olika sätt. Om vi ställer oss frågan ”*kommer en kund att churna?*” kan klasserna representeras som {ja, nej}. Ofta används en numerisk representation där en klass representeras som 1 och den andra som 0. I detta exempel vore alltså klasserna {1, 0}. Om vi har ett logiskt uttryck används ofta ”*sant*” om en egenskap är uppfyllt och ”*falskt*” om den inte är det. I detta fall vore klasserna {sant, falskt}. Vilken representation som används beror på sammanhanget.

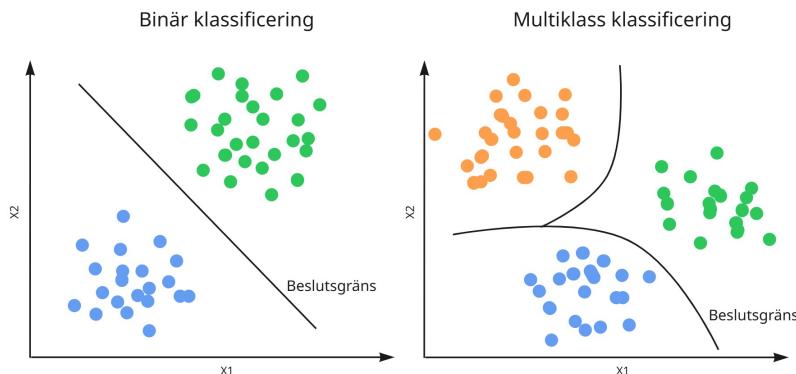
När antalet klasser är fler än två kallas detta för *multiklass klassificering*. Exempelvis kan målet vara att prediktera vilket djur vi ser på en bild där de möjliga alternativen kan vara hund, katt, häst och fågel. I detta fall är klasserna {hund, katt, häst, fågel}.

I de fall vi ska prediktera två eller flera utdata variabler samtidigt, kallas det för *multioutput klassificering*. Det kan handla om att vi ska prediktera om en person är över 18 år {ja, nej} samt om det är en {man, kvinna}. I detta fall finns det två utdata variabler, om personen är över 18 år samt vilket kön personen har. I de fall det finns flera *outputs* och minst en *output* kan anta fler än två klasser så kallas det för *multioutput multiklass klassificering*. Om vi exempelvis vill prediktera om en person är över 18 år, vilket kön den har samt var den bor, där vi antar att personen kan bo i {södra Sverige, mellersta Sverige, norra Sverige}, så vore detta ett exempel på *multioutput multiklass klassificering*. Vi kommer i denna bok inte gå djupare in på de typer av klassificering som nämns i detta stycke. Vi håller oss alltså till binär klassificering och multiklass klassificering.

För att genomföra klassificering så används klassificeringsmodeller som tränas för att de ska lära sig kunna särskilja olika klasser. Det finns flera olika modeller där två exempl

är de som heter *logistisk regression* och *random forest*. I Avsnitt 4.3 kommer vi lära oss hur dessa modeller fungerar där vi kommer använda dem för att genomföra prediktioner.

I samband med att klassificeringsmodeller tränas på data så lär de sig en *beslutsgräns*, vilket benämns för *decision boundary* på engelska, alltså en gräns som skiljer klasser från varandra över det rum som spänns upp av indata-variablene. Det är vid denna gräns som modellen ändrar beslut om hur den ska klassificera från en klass till en annan. En beslutsgräns kan vara både linjär och olinjär beroende på vilken modell som används. I enklare modeller, såsom logistisk regression, är beslutsgränsen linjär, medan den i mer komplexa och flexibla modeller, såsom *random forest*, kan vara olinjär. Om en beslutsgräns är för enkel kan detta leda till underanpassning, och tvärtom ifall en beslutsgräns är för komplex, då kan detta leda till överanpassning. Vi återkommer till hur beslutsgränser för olika modeller ser ut och hur utmaningar såsom under- och överanpassning hanteras senare i kapitlet. Se Figur 4.1 för en schematisk illustration av beslutsgränser. Kollar vi exempelvis på den vänstra bilden i Figur 4.1 så ser vi att vi har ett binärt klassificeringsproblem där klasserna är {blå, grön}. Beslutsgränsen är linjär och om vi ska prediktera en ny datapunkt så kommer alla datapunkter som hamnar under eller på den linjära beslutsgränsen klassificeras som blå och de som hamnar över beslutsgränsen klassificeras som grön.



Figur 4.1: Schematisk bild av binär klassificering och multiklass klassificering där även beslutsgränserna är inkluderade. I den vänstra bilden ser vi en linjär beslutsgräns och i den högra bilden ser vi en icke-linjär beslutsgräns.

#### 4.1.1 *OvR*- och *OvO*-algoritmerna

Binära klassificeringsmodeller kan med hjälp av två algoritmer som heter *One-vs-Rest* (*OvR*) och *One-vs-One* (*OvO*) utvidgas till att även kunna hantera multiklass klassificering.

*OvR* innebär att för varje klass i ett multiklass problem så tränas en modell för att kunna särskilja om en datapunkt tillhör den klassen eller inte. När en ny observation ska klassificeras så körs alla modeller där den slutgiltiga prediktionen blir det som den modell med högst *score* (såsom sannolikhet) predikterar. Vi exemplifierar. Antag att vi ska klassificera siffrorna  $\{0, 1, 2, \dots, 9\}$ . *OvR* tränar då en klassificerare för varje siffra. Alltså en klassificerare predikterar om en siffra är en 0:a eller inte, en klassificerare predikterar om en siffra är en 1:a eller inte och så vidare för alla siffror. När vi sen ska klassificera en siffra, kör vi alla modeller där prediktionen blir den siffran som fått högst score.

När vi använder oss av *OvO* tränar vi istället en modell för varje par av klasser. Det innebär att om det finns  $N$  stycken klasser kommer  $(N \times (N - 1))/2$  stycken modeller behöva tränas. Varje modell jämför ett klasspar och predikterar en av de två klasserna. Den klass som predikteras flest gånger totalt väljs slutligen. Vi exemplifierar. Antag återigen att vi ska klassificera siffrorna  $\{0, 1, 2, \dots, 9\}$ . *OvO* tränar en binär klassificerare för varje par av siffror:  $\{0, 1\}, \{0, 2\}, \{0, 3\}, \dots, \{0, 9\}, \{1, 2\}, \{1, 3\}, \dots, \{1, 9\}, \dots, \{8, 9\}$ . När en prediktion ska göras för en ny observation, kör vi alla modeller och ser vilken siffra som predikteras flest gånger. Det blir vår slutgiltiga prediktion.

**i** Använder vi en binär klassificeringsmodell för multiklass data så kommer *scikit-learn* automatiskt köra *OvO* eller *OvR* algoritmerna åt oss.

## 4.2 Utvärderingsmått

För att utvärdera klassificeringsmodeller är nedanstående mått vanligt förekommande:

- *Confusion matrix*
- *Precision*
- *Recall*
- *F1-score*
- *ROC*-kurvan

Som vi kommer att se kan det ofta vara mer komplext att utvärdera klassificeringsmodeller än regressionsmodeller på grund av att olika typer av fel kan göras. Därför kan

det ofta vara användbart att använda sig av flera mått i samband med analys för att få en djupare och mer nyanserad förståelse för hur bra modellen presterar och vilka typer av fel den gör.

#### 4.2.1 *Confusion matrix*

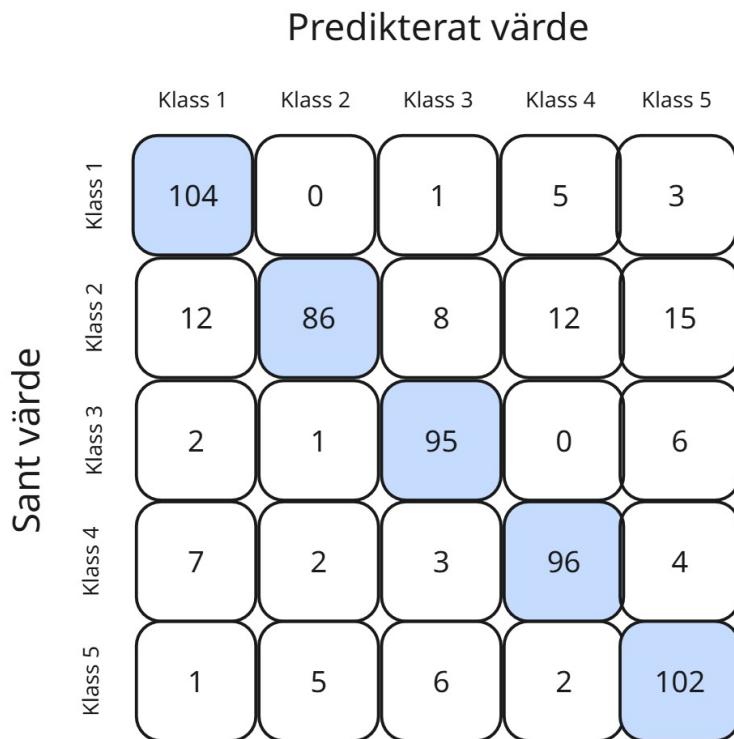
En *confusion matrix* är en matris som visualiseras hur väl en modell presterar genom att jämföra sanna och predikterade värden. Varje rad representerar den sanna klassen, medan varje kolumn representerar den predikterade klassen. Det kan också vara tvärtom. Antalet gånger varje klass har predikterats som respektive klass av modellen beräknas och skrivs i matrisen. Diagonalen, från övre vänstra hörnet till nedre högra hörnet, i matrisen visar alla datapunkter som blivit korrekt klassificerade. I Figur 4.2 ser vi hur en *confusion matrix* för ett binärt klassifieringsproblem ser ut. Notera särskilt de olika definitionerna i figuren, alltså *TP* (*True Positive*), *TN* (*True Negative*), *FN* (*False Negative*) och *FP* (*False Positive*). Vi återkommer till dessa när vi går igenom de resterande utvärderingsmåtten i detta avsnitt.

		Predikterat värde	
		Positiv	Negativ
Sant värde	Positiv	<b>True Positive (TP)</b> Antalet gånger modellen korrekt predikterat en positiv klass	<b>False Negative (FN)</b> Antalet gånger modellen felaktigt predikterat en negativ klass
	Negativ	<b>False Positive (FP)</b> Antalet gånger modellen felaktigt predikterat en positiv klass	<b>True Negative (TN)</b> Antalet gånger modellen korrekt predikterat en negativ klass

Figur 4.2: En *confusion matrix* för ett binärt klassifieringsproblem.

När vi har ett multiklass klassifieringsproblem, alltså fler än två klasser, utökas vår *confusion matrix*. Se Figur 4.3. I figuren ser vi att klass 1 predikterats som klass 1, alltså korrekt, 104 gånger. Klass 1 har dock predikterats felaktigt som klass 3 en gång, som klass 4 fem gånger och som klass 5 tre gånger. Diagonalen från övre vänstra hörnet

till högra nedre hörnet visar antalet gånger som modellen gjort korrekta prediktioner för respektive klass. Exempelvis har klass 2 korrekt blivit klassificerad som klass 2, 86 gånger. Om vi fortsätter att kolla på figuren ser vi att de största felet sker för klass 2. Vi ser att det felaktigt klassificerats som klass 1, 12 gånger, som klass 4, 12 gånger och som klass 5, 15 gånger. Reflekterar vi över det så kan vi tänka oss att för handskrivna siffror så kan det lätt bli så att en tvåa ser ut som en femma. Kanske vore det möjligt att förbättra modellen genom att samla in mer träningsdata för just klass 2 för att modellen bättre ska kunna särskilja tvåor? Eller kanske *data augmentation*, som gås igenom i Avsnitt 8.4 hade kunnat vara användbart?



*Figur 4.3: Exempel på en confusion matrix för ett multiklass klassifieringsproblem.*

Med hjälp av *scikit-learn* kan vi skapa en *confusion matrix* i Python. Se kodexemplet nedan.

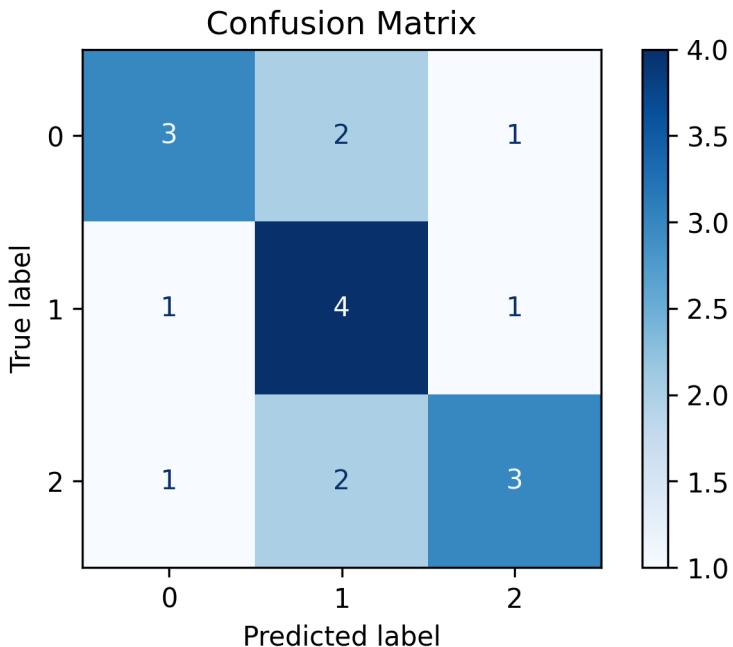
```
from sklearn.metrics import confusion_matrix          ①  
  
y_true = [0, 1, 2, 0, 1, 2, 1, 0, 2, 1, 0, 2, 2, 1, 0, 1, 2, 0]  
y_pred = [0, 2, 2, 0, 0, 1, 1, 0, 2, 1, 1, 0, 1, 1, 2, 1, 2, 1] ②  
  
cm = confusion_matrix(y_true, y_pred)                ③  
print(cm)                                         ④
```

- ① Importera *confusion matrix* från *scikit-learn* biblioteket.
- ② Skapa variabler för sanna och predikterade värden. Dessa kommer att återanvändas i kodexemplen för de kommande utvärderingsmåtten.
- ③ Skapa en *confusion matrix*.
- ④ Skriv ut *confusion matrix*.

```
[[3 2 1]  
 [1 4 1]  
 [1 2 3]]
```

I koden nedan visualiseras vi den *confusion matrix* som vi skapade ovan.

```
from sklearn.metrics import ConfusionMatrixDisplay          ①  
import matplotlib.pyplot as plt  
  
disp = ConfusionMatrixDisplay(confusion_matrix=cm,  
    ↵ display_labels=["0", "1", "2"])  
disp.plot(cmap="Blues", values_format="d")  
plt.title("Confusion Matrix")  
plt.show()
```



Sammanfattningsvis, en *confusion matrix* ger oss värdefull information. Bland annat innehåller den information om  $TP$ ,  $TN$ ,  $FN$  och  $FP$ , dessa används i mått som *accuracy*, *precision*, *recall* och *F1-score*. Härnäst kollar vi på dessa mått.

#### 4.2.2 Accuracy

*Accuracy* är ett mått på hur stor andel av alla observationer som klassificerats korrekt. Beräkningsformeln är enligt Ekvation 4.1.

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \quad (4.1)$$

Ett högt värde på *accuracy* innehåller alltså att en stor andel av observationerna har klassificerats korrekt. Innebär detta att vår modell presterar väl? Inte nödvändigtvis. Om vårt dataset är obalanserat, alltså att en klass är betydligt vanligare än andra, kan måttet riskera att vara missvisande. Om ett dataset har datapunkter där 95% tillhör klass 1 och vår modell alltid förutsäger klass 1 kommer *accuracy score* att vara 95%,

trots att modellen inte alls klarar av att prediktera klass 0. Beroende på hur datan vi arbetar med ser ut kan *accuracy*-måttet alltså vara mer eller mindre lämpligt.

Vi kan använda *scikit-learn* för att beräkna *accuracy* enligt koden nedan.

```
from sklearn.metrics import accuracy_score  
accuracy = accuracy_score(y_true, y_pred)  
print("Accuracy:", round(accuracy, 2))
```

- ① Importera *accuracy score* från *scikit-learn* biblioteket.
- ② Beräkna *accuracy score*, där vi använt *y\_true* och *y\_pred* från föregående kodexempel.
- ③ Skriv ut *accuracy* avrundat till två decimaler.

Accuracy: 0.56

#### 4.2.3 *Precision* och *Recall*

Nu kommer vi kolla på utvärderingsmåtten *precision* och *recall*. De hör ihop genom det samband som benämns *precision-recall tradeoff* som vi strax kommer gå igenom, vi börjar dock med att kolla på respektive mått.

*Precision* mäter andelen av de positiva prediktionerna som faktiskt är korrekta. Beräkningsformeln är enligt Ekvation 4.2.

$$Precision = \frac{TP}{TP + FP} \quad (4.2)$$

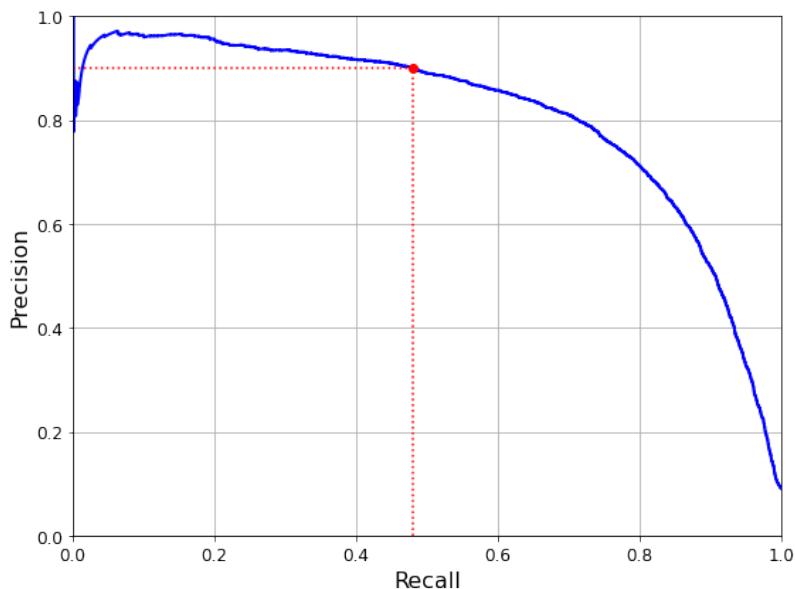
Teoretiskt sett, om ett högt värde för *precision* är önskvärt är det möjligt att bygga en modell som endast klassificerar en observation som positiv om den är väldigt säker. Detta kommer dock, generellt sett, ofta leda till att många värden som faktiskt är positiva felaktigt predikteras som negativa. Det leder då till att *recall*, som vi härnäst definierar, blir lågt.

*Recall* mäter andelen av den positiva klassen som predikteras korrekt. Beräkningsformeln är enligt Ekvation 4.3.

$$Recall = \frac{TP}{TP + FN} \quad (4.3)$$

Teoretiskt sett, om vi bygger en modell som alltid klassificerar en observation som positiv kommer vi att få en 100% *recall* (under antagandet att det finns minst en positiv observation i datasetet). Det leder dock, generellt sett, till att *precision* minskar eftersom andelen av våra positiva prediktioner som faktiskt är korrekta minskar (vi predikterar ju alltid positivt oavsett om observationen faktiskt är positiv eller negativ). Detta fenomen, att en högre *recall* generellt sett leder till en lägre *precision* och vice versa, benämns för *precision-recall tradeoff*.

I Figur 4.4 ser vi hur en ökad *precision* ger en minskad *recall* och vice versa. Den röda sträckade linjen visar att för en given *recall* får vi en given *precision* och vice versa. Kollar vi i figurens övre vänstra hörn, ser vi ett litet undantag där både *precision* och *recall* stiger, det är dock ett undantag och inget vi bryr oss om eftersom det generella sambandet är att det finns en *tradeoff*.



Figur 4.4: En visualisering av precision-recall tradeoff.

Vi kollar nu på två exempel.

Det första exemplet är kopplat till mejl där ett mejl kan klassificeras som spam-mejl eller vanligt mejl.

- $TP$  = Spam-mejl som korrekt klassificerats som spam-mejl.
- $FP$  = Vanligt mejl som felaktigt klassificerats som spam-mejl. Detta innebär att ett vanligt mejl hamnar i skräpposten vilket alltså inte är önskvärt.
- $FN$  = Spam-mejl som felaktigt klassificerats som vanligt mejl. Detta innebär att ett spam-mejl hamnar i den vanliga inkorgen vilket alltså inte är önskvärt.

Om vår modell har en hög *precision* innebär det att om ett mejl markerats som spam så är det ofta ett spam-mejl, men förmodligen hamnar även många spam-mejl i vår inkorg. Detta leder då till ett lågt värde på *recall*.

Det andra exemplet är kopplat till patienter inom vården där en patient kan klassificeras som sjuk eller frisk.

- $TP$  = Sjuka patienter som korrekt klassificerats som sjuka.
- $FP$  = Friska patienter som felaktigt klassificerats som sjuka.
- $FN$  = Sjuka patienter som felaktigt klassificerats som friska. Detta innebär att patienten inte kommer få den behandling den bör få.

Om vår modell har ett högt *recall* innebär det att patienter som faktiskt är sjuka identifieras. Men det leder då även till att *precision* blir lågt innehållande att många av de som vi klassificerar som sjuka egentligen är friska. Om vi fortsätter analysera exemplet inser vi att för allvarliga sjukdomar som kan vara livshotande är det viktigt att vi har en mycket hög *recall* så de som drabbas av sjukdom faktiskt får vård. För icke allvarliga sjukdomar behöver man generellt sett acceptera en lägre *recall* eftersom annars kan mycket resurser förbrukas till att behandla patienter som inte behöver vård. Det leder då till att de som kanske verkligen hade behövt vård då inte kan få den. Det faktum att det finns begränsade resurser leder alltså till att man generellt sett måste acceptera en *recall* som är lägre än 100%. Detta exemplet belyser också det faktum att man behöver analysera den situation som man tar beslut kring.

I koden nedan demonstreras hur *scikit-learn* används för att beräkna *precision* och *recall*.

```
from sklearn.metrics import precision_score, recall_score          ①

precision = precision_score(y_true, y_pred, average = "macro")    ②
recall = recall_score(y_true, y_pred, average = "macro")
print("Precision:", round(precision, 3))
print("Recall:", round(recall, 3))                                ③
```

① Importera *precision* och *recall* från *scikit-learn* biblioteket.

- ② Beräkna *precision. average* = "macro" innebär att *precision* beräknas för varje klass i vårt multiklass klassificeringsproblem och sedan tas ett medelvärde för att få en siffra för *precision*. Det finns andra alternativ, se *scikit-learn* dokumentationen för detaljer.
- ③ Skriv ut *precision* och *recall* avrundat till tre decimaler.

*Precision:* 0.567

*Recall:* 0.556

Hur kan vi då när vi modellerar styra *precision recall trade-off*? Vid klassificering utgår många modeller från hur stor sannolikheten är att en datapunkt tillhör en klass. Som standard kommer modellen att prediktera att datapunkten tillhör en klass om sannolikheten är lika med eller högre än 50% och den andra klassen om sannolikheten är lägre än 50%. Denna gräns kallas för *threshold*. Det är möjligt att justera värdet för denna och på så sätt öka eller minska *precision* respektive *recall*:

- Om vi **ökar** *threshold* kommer endast de mer säkra fallen att klassificeras som positiva, alltså får vi färre falska positiva, vilket i sin tur leder till att *precision* ökar. Samtidigt leder detta till att vi missar positiva fall, det blir alltså fler falska negativa, vilket leder till att *recall* minskar.
- Om vi **minskar** *threshold* tillåter vi att fler datapunkter klassificeras som positiva, vilket leder till färre falska negativa och således ökar *recall*. Detta leder även till att negativa fall felaktigt kommer att klassificeras som positiva, alltså får vi fler falska positiva, vilket leder till att *precision* minskar.

Hur vi gör detta rent praktiskt kommer demonstreras i Avsnitt 4.4.2.

Vid tillfällen när en balans mellan *precision* och *recall* önskas kan *F1-score* användas. Vi kollar på det härnäst.

#### 4.2.4 *F1-score*

*F1-score* är det som benämns det *harmoniska medelvärdet* för *precision* och *recall*. Beräkningsformeln är enligt Ekvation 4.4.

$$F_1 = \frac{2}{\text{recall}^{-1} + \text{precision}^{-1}} = 2 \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} = \frac{2\text{TP}}{2\text{TP} + \text{FP} + \text{FN}} \quad (4.4)$$

För att få ett högt värde på *F1-score* krävs det att både *precision* och *recall* är högt. Vi kan få en intuition för det genom att göra ett räkneexempel, se nedan.

	Precision	Recall	F_1 Score
0	1.00	1.00	1.00
1	1.00	0.01	0.02
2	0.01	1.00	0.02
3	0.90	0.40	0.55
4	0.60	0.70	0.65
5	0.90	0.10	0.18
6	0.60	0.40	0.48

*F1-score* är därför ett lämpligt mått att använda när vi önskar en balans mellan *precision* och *recall*. *F1-score* kan beräknas i *scikit-learn* enligt koden nedan.

```
from sklearn.metrics import f1_score

f1 = f1_score(y_true, y_pred, average="macro")
print("F1-score:", round(f1, 2))
```

F1-score: 0.55

I koden nedan demonstrerar vi *classification\_report* från *scikit-learn* där *precision*, *recall* och *f1-score* skrivs ut. Läsaren uppmanas att läsa dokumentationen för att förstå vad allting betyder i den utskrivna tabellen.

```
from sklearn.metrics import classification_report
y_true = [0, 1, 2, 2, 2]
y_pred = [0, 0, 2, 2, 1]
target_names = ['class 0', 'class 1', 'class 2']
print(classification_report(y_true, y_pred,
                           target_names=target_names))
```

	precision	recall	f1-score	support
class 0	0.50	1.00	0.67	1
class 1	0.00	0.00	0.00	1
class 2	1.00	0.67	0.80	3
accuracy			0.60	5
macro avg	0.50	0.56	0.49	5
weighted avg	0.70	0.60	0.61	5

### 4.2.5 ROC-kurvan

*ROC*-kurvan (från engelskans *receiver operating characteristic curve*) liknar *precision-recall* kurvan. Men istället för att visualisera sambandet mellan *precision* och *recall* visualiseras sambandet mellan *TPR*, från engelskans *True Positive Rate*, och *FPR*, från engelskans *False Positive Rate*, se Ekvation 4.5 och Ekvation 4.6. Notera att *TPR* alltså är en annan benämning för *recall*.

Vi vill också uppmärksamma läsaren på begreppen *sensitivity* och *specificity*. *Sensitivity* anger hur väl modellen hittar positiva fall, alltså *recall*, och *specificity* anger hur väl modellen hittar negativa fall. Om vi skulle identifiera vilka som är sjuka anger *sensitivity* alltså hur bra vi hittar de som faktiskt är sjuka och *specificity* hur bra vi hittar de som inte är sjuka (friska). *FPR* kan alltså beräknas som  $1 - \text{specificity}$ . Vi kan säga att *ROC*-kurvan visualiseras sambandet mellan *sensitivity* och *1-specificity*.

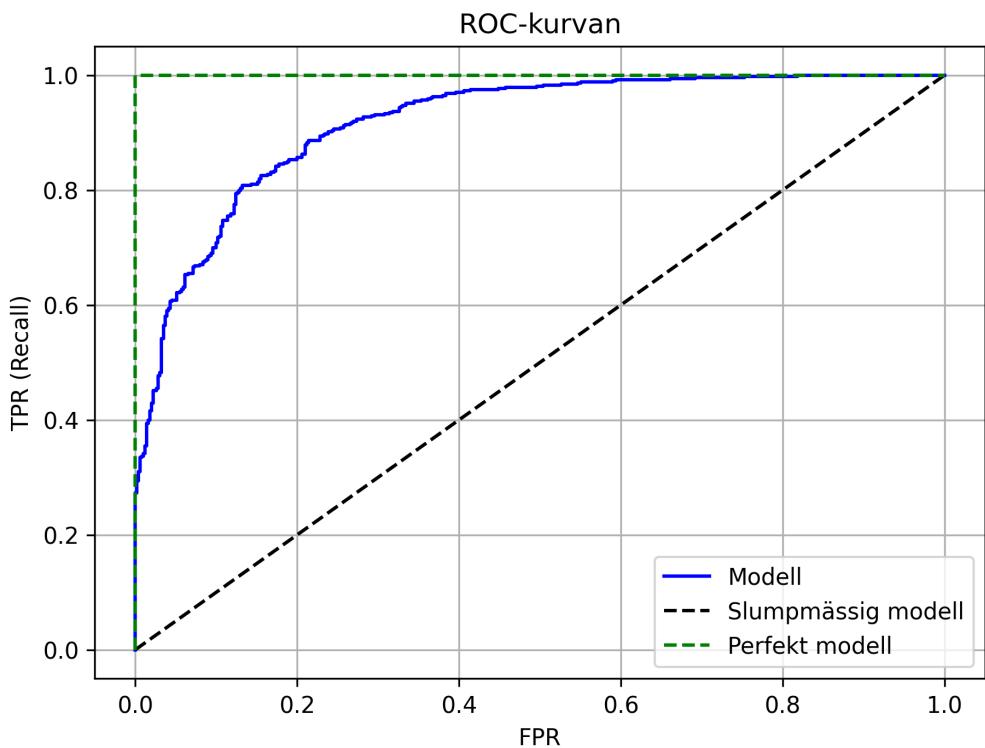
$$TPR = \frac{TP}{TP + FN} = recall \quad (4.5)$$

$$FPR = \frac{FP}{FP + TN} \quad (4.6)$$

Precis som för *precision* och *recall* kan vi ändra på *threshold* för att minska och öka *TPR* respektive *FPR* beroende på vad vi vill att modellen ska prioritera.

- Om vi **ökar** *threshold* kommer endast de fall där modellen är säkrare att klassificeras som positiva, alltså får vi färre falska positiva och sanna positiva, vilket i sin tur leder till att *FPR* och *TPR* minskar.
- Om vi **minskar** *threshold* tillåter vi att fler datapunkter klassificeras som positiva, vilket leder till färre falska negativa och således ökar *FPR* och *TPR*.

I Figur 4.5 visas hur *ROC*-kurvor kan se ut för olika *thresholds*. För en “perfekt modell” skulle kurvan gå rakt upp till koordinaten  $(0, 1)$  — vilket betyder att alla positiva fall identifierats korrekt utan att några negativa klassificerats felaktigt ( $TPR = 1$ ,  $FPR = 0$ ). Om *threshold* därefter sänks kommer även negativa exempel börja klassificeras som positiva, vilket ökar *FPR*. Detta visas som en horisontell linje från  $(0, 1)$  till  $(1, 1)$ . Detta segment motsvarar inte längre ett perfekt resultat, men modellen är fortfarande “perfekt” eftersom vi kan välja en optimal tröskel som leder oss till  $(0, 1)$ . En modell som helt slumpmässigt gissar vilken klass en datapunkt tillhör benämner vi “slumpmässig modell”. Eftersom slumprövningen har 50% chans att gissa rätt (givet balanserade klasser) kommer *TPR* och *FPR* att förändras i samma takt, och *ROC*-kurvan blir därför en diagonal linje, se den svarta streckade linjen i figuren. Den svarta linjen representerar hur en *ROC*-kurva, rent schematiskt, kan se ut för en godtycklig modell.



Figur 4.5: Visualiseringar av ROC-kurvor för olika typer av modeller.

Arean under *ROC*-kurvan benämns *ROC-AUC* från engelskans *Area under the curve*. Det är ett mått på hur bra modellen presterar. Om denna är 1.0 är klassificeringen perfekt, om den är 0.5 är klassificeringen lika bra som en slumppräglad modell och om den är lägre än 0.5 presterar modellen sämre än slumpen. *ROC-AUC* är ett mått som kan användas för att jämföra modeller mot varandra. Om två modeller jämförs är den modell med högre *ROC-AUC* bättre. Två modeller kan ha samma *ROC-AUC* trots att *ROC*-kurvorna för modellerna ser olika ut.

Koden nedan demonstrerar beräkningen av *ROC-AUC*.

```
from sklearn.metrics import roc_auc_score  
  
auc_score = roc_auc_score(y_true, y_scores)  
print("ROC-AUC Score:", round(auc_score, 2))
```

①  
②

- ① Beräkna ROC-AUC.  
② Skriv ut ROC-AUC.

ROC-AUC Score: 0.92

Det finns situationer när *ROC*-kurvan och tillhörande mått *ROC-AUC* kan vara missvisande och andra utvärderingsmått bör användas istället. Om vi exempelvis har ett obalanserat dataset där den negativa klassen domineras i antal kommer *FPR* alltid att bli låg, trots att modellen är dålig på att hitta den positiva klassen (en låg *FPR* indikerar att vi har få negativa exemplar som felaktigt klassificeras som positiva).

## 4.3 Klassificeringsmodeller

Precis som för regressionsproblem finns det flera olika modeller för att hantera klassificeringsproblem. Några vanligt förekommande är:

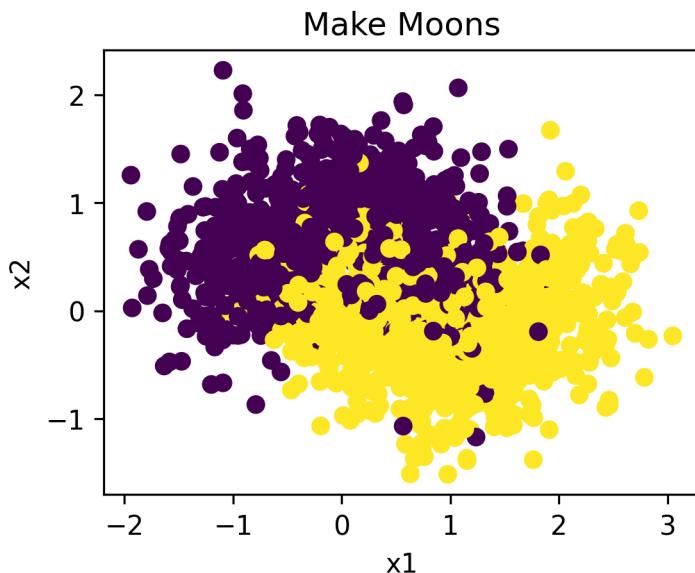
- Logistisk regression
- *Support vector machines*
- Beslutsträd
- *Ensemble learning*
- *Random forest*

För att demonstrera hur de olika modellerna fungerar kommer vi skapa ett syntetiskt dataset med `make_moons` funktionaliteten från *scikit-learn*. Att använda syntetiska dataset är smidigt för att snabbt kunna demonstrera och testa olika modeller. Datasetet

som skapas med hjälp av `make_moons` består av två oberoende variabler,  $x_1$  och  $x_2$ , samt två klasser,  $\{0, 1\}$ . Koden nedan visualiseras datan och vi ser att den formar två halvmånar, därför namnet `make_moons`.

```
from sklearn.datasets import make_moons          ①  
  
X, y = make_moons(n_samples=2000, noise=0.4, random_state=42) ②  
  
X_train, X_test, y_train, y_test = train_test_split(X, y,  
    test_size=0.2, random_state=42)                  ③  
  
fig, ax = plt.subplots(figsize=(4, 3))  
ax.scatter(X_train[:, 0], X_train[:, 1], c=y_train)  
ax.set(title="Make Moons", xlabel="x1", ylabel="x2")      ④
```

- ① Importera `make_moons` från *scikit-learn* biblioteket.
- ② Skapa ett dataset med `make_moons` funktionen.
- ③ Dela upp datan i träningsdata och testdata.
- ④ Visualisera datan.



Notera hur de två klasserna skapar varsin halvmåne och att dessa har ett visst överlapp. Det är därför inte ett trivialt mönster och en bra klassificeringsmodell behöver lära sig att utnyttja detta mönster för att prestera bra.

### 4.3.1 Logistisk regression

Logistisk regression är en modell som trots namnet “regression” används för klassificeringsproblem. Det är en binär klassificerare och modellen utför klassificering genom att estimera sannolikheten för att en datapunkt tillhör en klass. Om sannolikheten är större än eller lika med 50% kommer modellen prediktera att datapunkten tillhör den klassen, vi kallar det för den positiva klassen, som ofta representeras med 1. Om sannolikheten är lägre än 50% kommer modellen istället att prediktera att datapunkten tillhör den andra klassen, alltså den negativa klassen som ofta representeras med 0.

**i** Även om den logistiska regressionsmodellen är en binär klassificerare så kan den användas för multiklass klassificering med hjälp av *OvR* eller *OvO* algoritmerna, som presenterades i Avsnitt 4.1.1. Detta sker automatiskt i *scikit-learn*.

Den logistiska regressionsmodellen kan även utvidgas till att i sig själv kunna hantera multiklass klassificering utan att använda *OvR* eller *OvO*. Det benämns då för *multinomial logistic regression* eller *softmax regression*. De tekniska detaljerna är överkurs och inget vi går igenom i denna bok. Vi nämner det endast för fullständigheten skull.

Precis som för linjär regression, som presenterades i föregående kapitel, utgår den logistiska regressionsmodellen från att utdata kan modelleras som en linjär kombination av indata, plus en slumpmässig felterm,  $\epsilon$ . Vi kan tänka oss att vi hade kunnat definiera den logistiska regressionsmodellen enligt Ekvation 4.7.

$$p = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_p x_p + \epsilon \quad (4.7)$$

**i** Notera att när vi pratar om predikterade värden så läggs det på en hatt och  $\epsilon$  försvinner eftersom det inte finns något slumpmässigt i genomförda prediktioner. Slumpmässigheten finns endast när vi kollar på faktisk data. Ekvation 4.7 hade då blivit enligt ekvationen nedan.

$$\hat{p} = \hat{\theta}_0 + \hat{\theta}_1 x_1 + \hat{\theta}_2 x_2 + \dots + \hat{\theta}_p x_p$$

Problemet är då att vänsterledet,  $p$ , som vi tolkar som en sannolikhet endast kan vara mellan  $0 <= p <= 1$  medan högerledet kan anta värden mellan  $-\infty < \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_p x_p + \epsilon < \infty$ . Det är alltså inte likhet. Se Figur 4.6 för ett exempel på detta. I figuren antar vi att vi har olika kunder och att dessa antingen kan *churna* (1) eller inte (0). Som oberoende variabel har vi ålder. Om vi vill modellera sannolikheten att en kund *churnar* så måste detta vara mellan 0 – 1 vilket gör att den gröna linjen inte är lämplig som modell. Den kan passera både 0 och 1 vilket gör att vi inte kan tolka det som en sannolikhet. Modellen kan därför inte användas rakt av. Någon form av justering behöver göras.

För att komma tillräffa med detta hade vi därför kunnat prova att skriva enligt nedan:

$$\frac{p}{1-p} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_p x_p + \epsilon \quad (4.8)$$

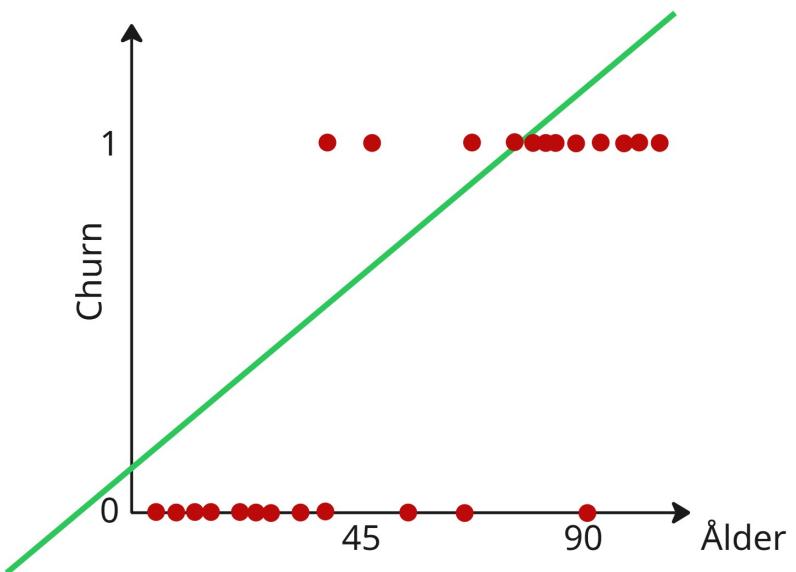
Vänsterledet kan nu anta värden mellan  $0 <= \frac{p}{1-p} < \infty$  och högerledet kan anta värden mellan  $-\infty < \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_p x_p + \epsilon < \infty$ . Det är ännu inte helt rätt, men det är närmare målet att det ska råda likhet mellan de två ledens. Vi testar slutligen nedanstående där log betecknar den naturliga logaritmen och är inversen till talet  $e$ .<sup>1</sup>

$$\log\left(\frac{p}{1-p}\right) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_p x_p + \epsilon \quad (4.9)$$

Vänsterledet kan nu anta värden mellan  $-\infty <= \log\left(\frac{p}{1-p}\right) < \infty$  och högerledet kan anta värden mellan  $-\infty < \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_p x_p + \epsilon < \infty$ . Likhet gäller alltså mellan de två ledens!

---

<sup>1</sup>Talet  $e$  är en matematisk konstant som går igenom i gymnasiets matematik 3 kurs. Den läsare som inte läst den nivån av matematik kan ignorera nedanstående beräkningar och endast skumma igenom det. Börja därefter läsa som vanligt igen från och med informationsrutan med rubriken ”*Matematik är viktigt och användbart*”.



Figur 4.6: Modell för hur sannolikheten att churna påverkas av ålder. Den gröna linjen representerar en linjär modell och de röda prickarna representerar olika kunder som antingen churnat (1) eller inte churnat och alltså stannat kvar i företaget (0). Eftersom den beroende variabeln churn är antingen 1 eller 0 kan vi tänka oss att vi hade velat modellera sannolikheten att en kund churnar, den bör då ligga mellan 0-1. Den gröna linjen har då problemet att den kan passera både 0 och 1. Vi kan därför inte använda modellen rakt av. Någon form av justering behöver göras.

När modellen tränas så används alltså Ekvation 4.9 av rent matematiska skäl, det ska vara likhet mellan båda leden. Vi är dock (generellt sett) intresserade av att få ut sannolikheter,  $p$ , snarare än  $\log\left(\frac{p}{1-p}\right)$ , som benämns för log-oddset. För att få ut sannolikheter efter att modellen tränats kan vi med lite algebran lösa ut för sannolikheten  $p$ , se ekvationerna nedan.

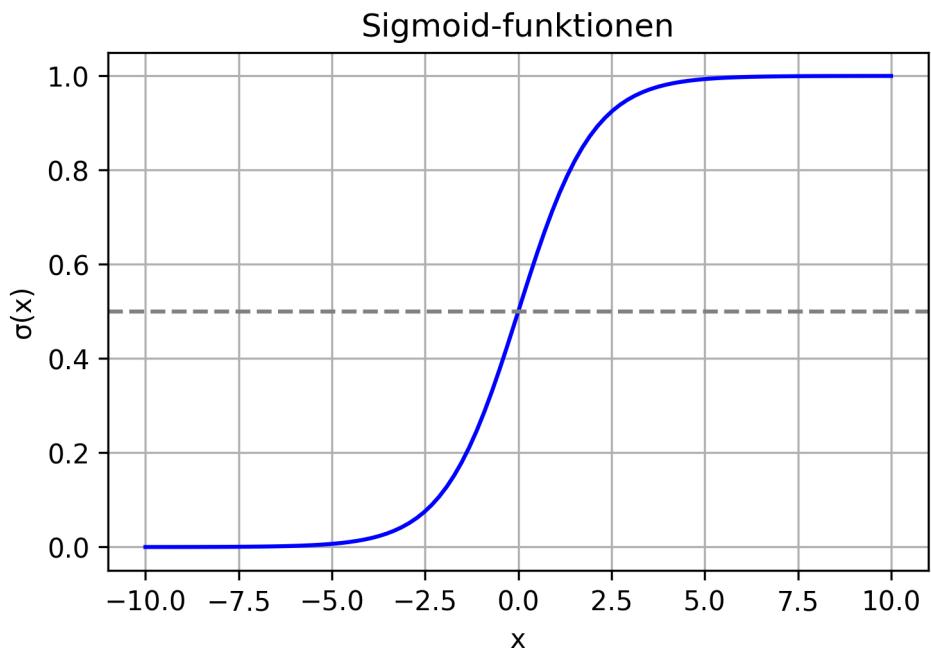
$$\begin{aligned} \log\left(\frac{p}{1-p}\right) &= \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_p x_p + \epsilon \\ \frac{p}{1-p} &= e^{\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_p x_p + \epsilon} \\ p &= (1-p) e^{\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_p x_p + \epsilon} \\ p + p e^{\theta_0 + \theta_1 x_1 + \cdots + \theta_p x_p + \epsilon} &= e^{\theta_0 + \theta_1 x_1 + \cdots + \theta_p x_p + \epsilon} \\ p(1 + e^{\theta_0 + \theta_1 x_1 + \cdots + \theta_p x_p + \epsilon}) &= e^{\theta_0 + \theta_1 x_1 + \cdots + \theta_p x_p + \epsilon} \\ p &= \frac{e^{\theta_0 + \theta_1 x_1 + \cdots + \theta_p x_p + \epsilon}}{1 + e^{\theta_0 + \theta_1 x_1 + \cdots + \theta_p x_p + \epsilon}} \\ p &= \frac{1}{1 + e^{-(\theta_0 + \theta_1 x_1 + \cdots + \theta_p x_p + \epsilon)}} \end{aligned}$$

Den sista ekvationen,  $p = \frac{1}{1 + e^{-(\theta_0 + \theta_1 x_1 + \cdots + \theta_p x_p + \epsilon)}}$ , kallas för den logistiska funktionen och är det som benämns för en sigmoid-funktion, dessa kännetecknas av ett s-format utseende. Den logistiska funktionen kan generellt sett skrivas enligt Ekvation 4.10 där  $x$  i vårt fall var  $(\theta_0 + \theta_1 x_1 + \cdots + \theta_p x_p + \epsilon)$ . Visualiseringar vi den logistiska funktionen enligt Figur 4.7 så ser vi den karakteristiska s-formen. Vi ser även att funktionen antar värden mellan 0 – 1 innehållande att vi kan tolka det som en sannolikhet - precis som vi önskade.

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{4.10}$$

### i Matematik är viktigt och användbart

Denna bok fokuserar på praktiska tillämpningar vilket innebär att vi inte går in på matematiska detaljer. Däremot är det bra att vara medveten om att både AI i allmänhet och ML i synnerhet är områden som i hög grad är uppbyggda av matematik. Det är en av anledningarna till att det är ett viktigt och användbart ämne.



Figur 4.7: En visualisering av den logistiska funktionen som är en sigmoid-funktion. Vi ser hur funktionen har värden mellan 0 – 1 innebärande att vi kan tolka det som en sannolikhet.

Ovan gick vi igenom en del matematiska detaljer. I praktiken, när vi praktiskt tillämpar modellen så kommer vi i vanlig ordning använda oss av *scikit-learn*.

För logistisk regression finns det flera justerbara hyperparametrar som påverkar modellens beteende och prestanda, dock färre än för mer komplexa modeller som vi kommer att lära oss om i resten av kapitlet. Exempel på två hyperparametrar som kan justeras, via exempelvis *GridSearch*, ser vi i Tabell 4.1. Det finns fler hyperparametrar såsom *solver* och *max\_iter* men det är kopplat till optimeringsalgoritmen som tränar modellen och är inget vi vanligtvis använder oss av. Vi litar alltså på att standardvärdena i *scikit-learn* fungerar bra vilket de i de flesta praktiska sammanhang gör. Läsaren uppmanas att läsa *scikit-learn* dokumentationen för att få en helhetsbild.

Tabell 4.1: Ett urval av hyperparametrar för logistisk regression från scikit-learns dokumentation.

Hyperparameter	Beskrivning	Standardvärde
penalty	Regulariseringstyp. {’l1’, ’l2’, ’elasticnet’, None}.	’l2’
C	Regulariseringsstyrka. Ju lägre C, desto starkare regularisering. Av typen float.	1.0

**i** Vi påminner läsaren om *the bias variance trade-off*, se Avsnitt 3.3.3, som förklrar varför vi kan tänkas vilja regularisera en modell.

För den logistiska regressionsmodellen kan vi exempelvis använda oss av *grid search* i *scikit-learn* för att hitta ett optimalt värde på *penalty* eller regulariseringsstyrkan *C*.

Vi använder datasetet skapat med *make\_moons* för att illustrera hur en logistisk regressionsmodell kan instantieras, användas för prediktion och slutligen utvärderas.

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

model_lr = LogisticRegression()                                ①
model_lr.fit(X_train, y_train)                                ②
y_pred = model_lr.predict(X_test)                             ③
```

```
accuracy = accuracy_score(y_test, y_pred)  
print(f"Test Accuracy: {accuracy:.2f}")
```

④

⑤

- ① Instantiera den logistiska regressionsmodellen.
- ② Träna modellen.
- ③ Använd den tränade modellen för att prediktera testdata.
- ④ Beräkna *accuracy score*.
- ⑤ Skriv ut *accuracy score* avrundat till två decimaler.

Test Accuracy: 0.83

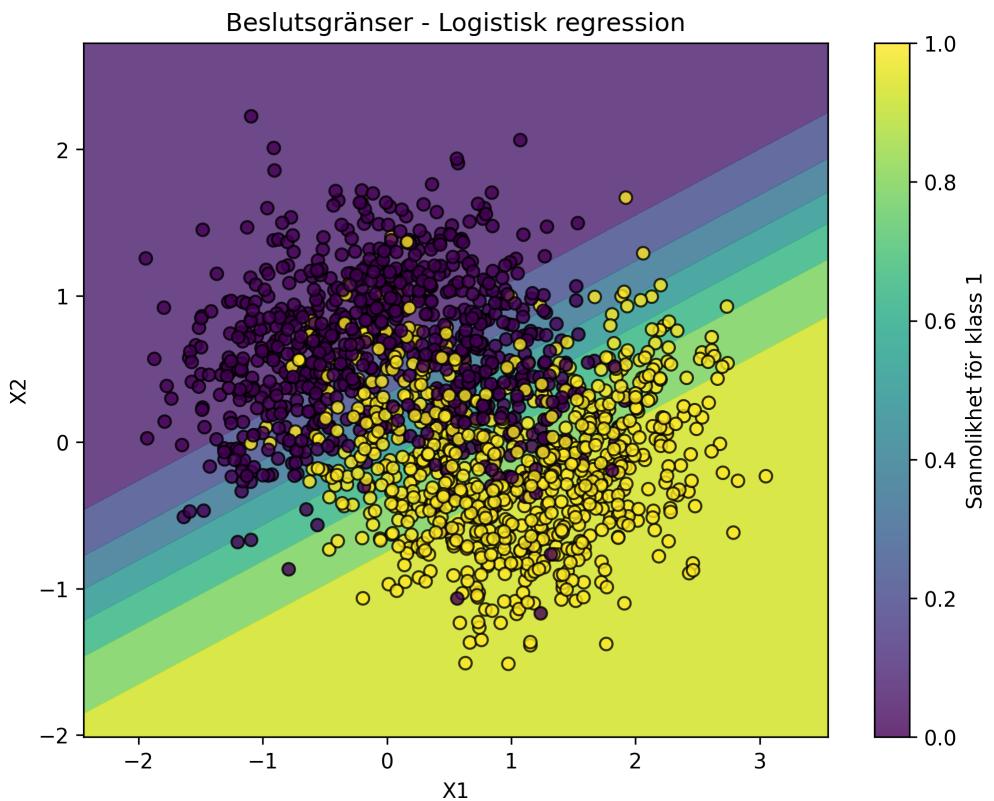
Den logistiska regressionsmodellen lyckas alltså klassificera korrekt klass för 83% av alla datapunkter i testdatan. I Figur 4.8 ser vi hur beslutsgränserna som modellen skapat ser ut.

Logistisk regression är en linjär modell, och således är beslutsgränserna också linjära vilket vi ser i Figur 4.8. Modellen fungerar generellt sett bra när sambandet mellan utdata och indata är ungefärligt linjärt, detta är inte fallet med datasetet skapat med `make_moons` vilket vi ser i figuren, detta leder till att många datapunkter ”i mitten” felklassificeras. Notera, vi skrev *beslutsgränser* ovan som alltså är plural, eftersom man själv kan välja vilken beslutsgräns som ska användas, bland annat för att styra önskad nivå på *precision* och *recall* (kom ihåg *precision-recall tradeoff*). Om inget annat sägs så är *default* för beslutsgränsen där sannolikheten för den positiva klassen (klass 1) är 50%, det är alltså beslutsgränsen som *scikit-learn* automatiskt använder för att genomföra klassificeringen.

När ett dataset har en icke-linjär struktur kan det vara en bra idé att testa logistisk regression med polynomvariabler eller andra mer komplexa modeller, såsom *SVM* eller *Random Forest*, som kan fånga icke-linjära samband. Koden nedan demonstrerar hur polynomvariabler kan användas i den logistiska regressionsmodellen. Figur 4.9 visar hur de tillhörande beslutsgränserna ser ut där vi alltså ser att de blir icke-linjära.

```
from sklearn.preprocessing import PolynomialFeatures  
from sklearn.pipeline import make_pipeline  
  
model_poly = make_pipeline(  
    PolynomialFeatures(degree=3),  
    LogisticRegression())
```

①



Figur 4.8: Beslutsgränser för en logistisk regressionsmodell som klassificerar observationer i en av två klasser (0/lila eller 1/gul) baserat på variablerna  $x_1$  och  $x_2$ . Bakgrundsfärgen visar sannolikheten (mellan 0 och 1) att en punkt tillhör klass 1 och färgskalan är graderad till höger.

```

)
model_poly.fit(X_train, y_train)

y_pred = model_poly.predict(X_test)

accuracy = accuracy_score(y_test, y_pred)
print(f"Test Accuracy: {accuracy:.2f}")

```

- ① Notera att vi importerade `make_pipeline` istället för `Pipeline` vilket vi gjorde i Avsnitt 2.4.3, från dokumentationen kan vi utläsa följande: This is a shorthand for the Pipeline constructor; it does not require, and does not permit, naming the estimators. Instead, their names will be set to the lowercase of their types automatically.

Test Accuracy: 0.86

Vi fick en något högre *accuracy* när vi använde oss av polynomvariabler. Kanske hade ännu högre grad av polynomvariablene ytterligare förbättrat resultatet? Prova!

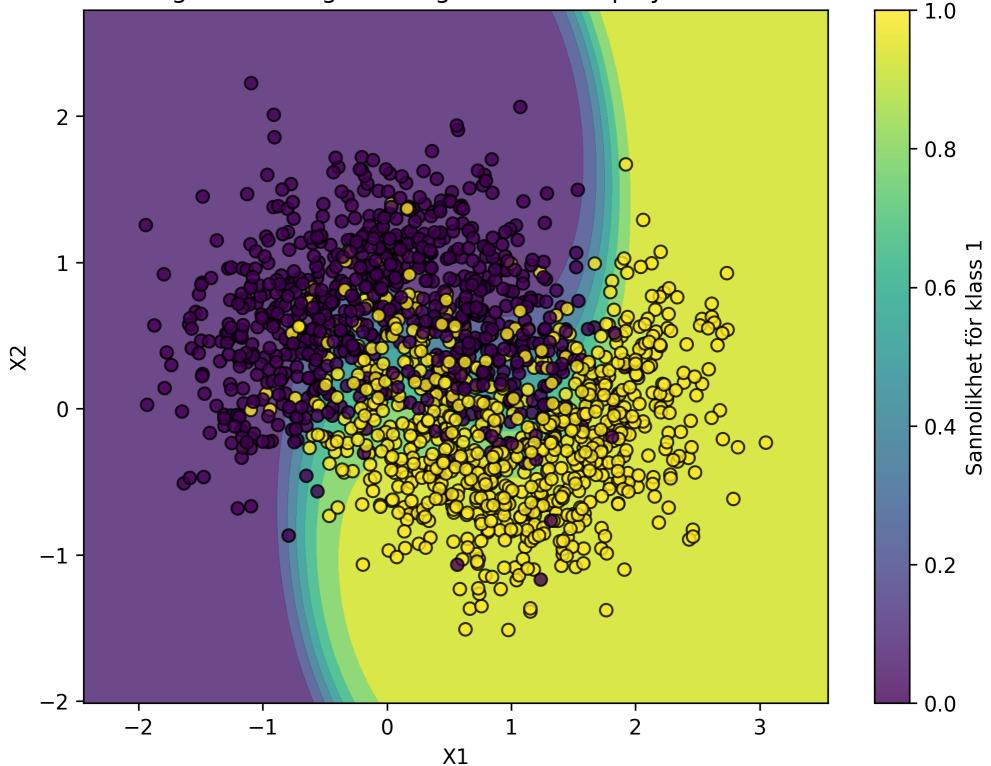
Vi noterar att även när den logistiska regressionsmodellen inte är den mest lämpade modellen för ett problem kan den med fördel användas som en baslinjemodell (på engelska, *baseline modell*) som ger en första indikation i ett klassificeringsproblem och något man kan jämföra andra mer komplexa modeller med. Detta eftersom modellen är lätt att förstå och enkel att träna där träningen ofta brukar gå relativt fort. Om logistisk regression presterar bra kan det indikera att datan är linjärt separerbar. Om den presterar dåligt kan det signalera att en mer flexibel modell kan behövas.

### 4.3.2 *Support vector machines (SVM)*

*SVM*, från engelskans *Support vector machines*, introducerades redan i föregående kapitel om regression. Som vi kommer se i detta avsnitt kan modellen även användas för att hantera klassificeringsproblem. För regressionsproblem var målet att hitta en ”väg” där så många datapunkter som möjligt från vårt dataset hamnade på. För klassificeringsproblem vill vi istället separera datapunkter som tillhör olika klasser och målet är därför istället att hitta en så bred ”väg” som möjligt mellan datapunkter från de två olika klasserna.

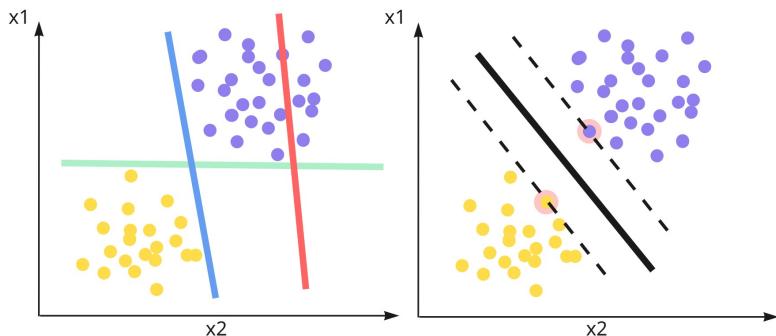
Med hjälp av Figur 4.10 kan vi skapa en förståelse för hur *SVM* för klassificering fungerar. I figuren representerar de två färgerna, gul och lila, två olika klasser i ett dataset.

Beslutsgränser - Logistisk regression med polynomvariabler



Figur 4.9: Beslutsgränser för den logistiska regressionsmodellen med polynomvariabler. Vi ser att beslutsgränserna är icke-linjära.

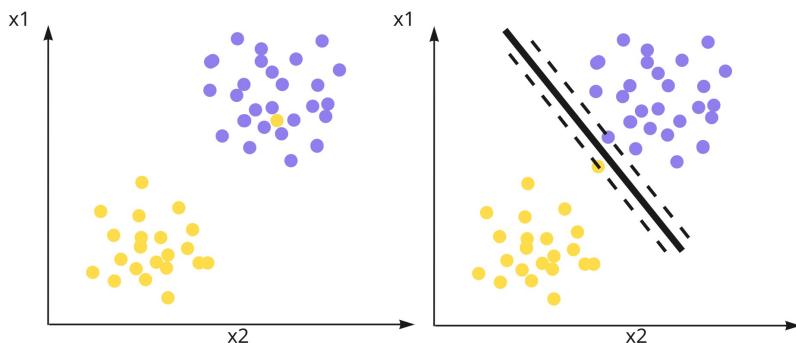
Vårt mål är att hitta ett hyperplan som separerar datapunkterna, eftersom vi i detta fall jobbar i två dimensioner så är hyperplanet en linje. Detta kommer att vara vår beslutsgräns. I den vänstra bilden finns tre olika alternativ på beslutsgränser. Den röda linjen misslyckas med att separera klasserna och är därför olämplig att använda. Däremot separerar både den gröna och den blåa linjen klasserna och hade kunnat användas. Teoretiskt sett hade vi kunnat skapa oändligt många linjer som separerar klasserna, så frågan är, hur ska vi välja *en* linje på ett optimalt sätt? Svaret är att SVM-modellen skapar en så bred ”väg” som möjligt mellan klasserna och då får vi även en unik linje (det kan bevisas matematiskt) vilket gör att det endast finns *ett* val. Detta ser vi i den högra bilden där ”vägen” representeras av sträckade linjer. Datapunkterna som de streckade linjerna passerar igenom, som i figuren är markerade med röda ringar, kallas för *support vectors*. De ”*supportar vägen*”.



Figur 4.10: I den vänstra bilden är den röda linjen olämplig att använda som beslutsgräns eftersom den misslyckas med att separera klasserna. De två andra linjerna separerar både klasserna men har problemet att de inte är ”unika”. Det finns teoretiskt sett oändligt många sådana linjer vi hade kunnat välja bland. För att kunna välja en unik linje som separerar klasserna så skapar SVM en så bred ”väg” som möjligt mellan klasserna. Detta visualiseras i den högra bilden.

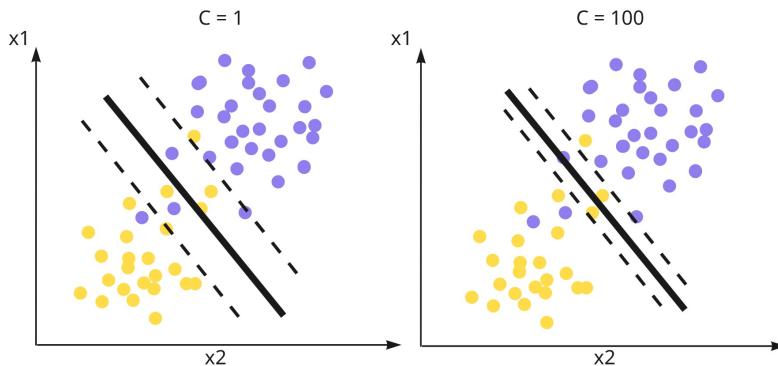
Om vi när vi skapar vår modell säger att alla datapunkter strikt måste vara utanför och på rätt sida om ”vägen” kallas detta för *hard margin classification*. Detta är dock mestadels något som skapar en teoretisk förståelse eftersom det i praktiken inte alltid är möjligt att strikt separera samtliga datapunkter. Detta illustreras i Figur 4.11. I den vänstra bilden är det inte möjligt att skapa en beslutsgräns som separerar klasserna. I den högra bilden blir ”vägen” väldigt smal. Detta leder förmodligen till att modellen generaliseringar dåligt på ny data vilket inte är önskvärt. För att hantera dessa problem

används i praktiken det som benämns *soft margin classification*. Detta innebär att datapunkter tillåts befina sig innanför eller på fel sida om ”vägen”. Målet är att ha en så bred ”väg” som möjligt och begränsa antalet *margin violations*. Som vi senare kommer se finns det en hyperparameter,  $C$ , som styr hur mycket vi tillåter *margin violations*. Högre  $C$  leder till mindre *margin violations* och vice versa. I de fall modellen anses vara överanpassad, det vill säga att modellen presterar bra på träningsdata men generaliseringen dåligt på ny osedd data (exempelvis på valideringsdata), så kan vi öka regulariseringen genom att minska  $C$ . Då följer modellen träningsdatan mindre exakt (fler *margin violations* tillåts) med målet att den ska generalisera till ny, osedd data, på ett bättre sätt. Se Figur 4.12. I samband med modellering måste en avvägning mellan att kunna skapa en så bred ”väg” som möjligt och hur många datapunkter som tillåts hamna på fel sida om ”vägen” göras. Detta kan vi styra med hjälp av hyperparametern  $C$ . För att välja ett optimalt värde på  $C$  har vi `GridSearch` från *scikit-learn* till vår hjälp.



Figur 4.11: I den vänstra bilden är det inte möjligt att välja en beslutsgräns som separerar datapunkterna från de två olika klasserna. I den högra bilden är det möjligt men resultatet blir krystat. Denna figur motiverar varför soft margin classification snarare än hard margin classification är det som i praktiken används.

SVM är, precis som logistisk regression, en binär klassificerare i grunden men med hjälp av *OvO* och *OvR* algoritmerna kan modellen i vanlig ordning även hantera multiklass klassificering. Detta sker automatiskt i *scikit-learn*.



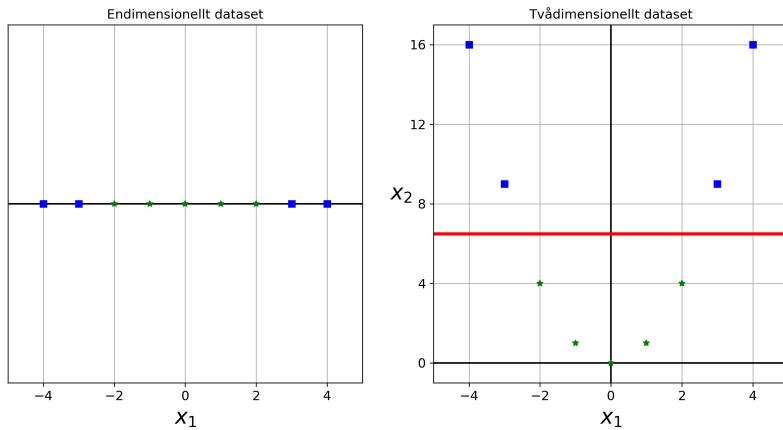
Figur 4.12: Visualisering över hur olika värden på hyperparametern  $C$  påverkar SVM-modellen. För överanpassade modeller minskar vi generellt sett värdet på  $C$  för att modellen ska följa träningsdatan mindre noggrant i hopp om att den ska generalisera bättre på ny, osedd data. För underanpassade modeller ökar vi generellt sett värdet på  $C$  för att modellen ska följa träningsdatan mer noggrant i hopp om att den ska lära sig mönstret från datan bättre. Vi kan optimera värdet på  $C$  med hjälp av grid search från scikit-learn.

### i SVM är lämplig för små och medelstora dataset

Generellt sett är *SVM* en modell som lämpar sig väl för små och medelstora dataset. För stora dataset kan modellen ta väldigt lång tid att träna.

Hittills har våra exempel behandlat linjär klassificering med *SVM*. Det är också möjligt att använda *SVM* för icke-linjära dataset. Ett sätt att göra detta på är att lägga till polynomiska termer, vilket kan leda till att datasetet blir linjärt separerarbart. Att lägga till polynomiska termer är ett enkelt sätt att hantera icke-linjära samband i data. Se Figur 4.13. I den vänstra bilden från figuren ser vi att vi har ett dataset med endast en variabel,  $x_1$ . Denna variabel kan vi använda för att skapa polynomtermen  $x_2 = x_1^2$  som alltså är y-axeln i den högra bilden. Vi ser att datan blir linjärt separerbar efter att variabeln lagts till där beslutsgränsen representeras av den röda linjen. Att lägga till data, genom att använda polynomiska variabler, kan dock göra att modellträningen tar lång tid. För att hantera detta kan *kernel trick* användas. Hur det fungerar går att matematiskt bevisa men det är överkurs. När vi praktiskt implementerar det räcker det att känna till att det ger samma effekt som att lägga till fler oberoende variabler (polynom variabler i vårt exempel) utan att det faktiskt görs. Effekten blir alltså

densamma förutom att träning av modellen inte tar lika lång tid. Så istället för att manuellt lägga till polynomvariabler kommer vi i *scikit-learn* specificera hyperparametern `kernel='poly'`. Andra *kernels* vi kan använda för att hantera icke-linjär data är `rbf` (*RBF*) och `sigmoid`. Hur gör vi valet rent praktiskt? Vi använder `GridSearch` i *scikit-learn* för att välja optimal *kernel*. Vi går nu igenom hur *RBF*, som är en förkortning av “*radial basis function*”, fungerar.



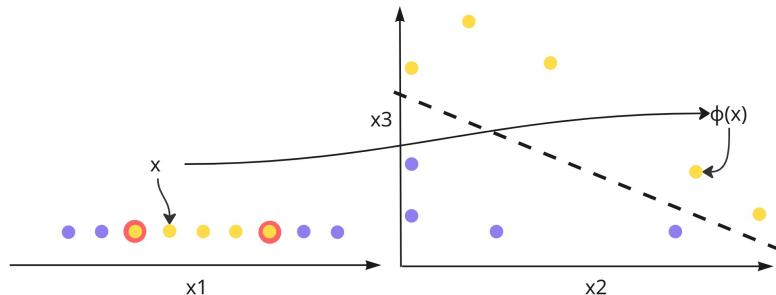
Figur 4.13: I den vänstra bilden, där det endast finns en variabel,  $x_1$ , är datan inte linjärt separerbar. Genom att lägga till polynomtermen  $x_2 = x_1^2$ , som representeras av  $y$ -axeln i den högra bilden ser vi att datan är linjärt separerbar. Beslutsgränsen är den röda linjen.

Genom att mäta avståndet från vissa punkter som vi kallar för *landmarks* kan vi skapa nya variabler. I *scikit-learn* är *RBF* den funktion som används för att mäta just detta avstånd om hyperparametern `kernel='rbf'` är specificerad. Det är också *default*-värdet innebörande att det är vad som används om inget annat specificeras. I Figur 4.14 ser vi i den vänstra bilden ett dataset som inte är linjärt separerbart. Vi skapar två stycken *landmarks*. Dessa representeras av de röda ringarna. För varje observation/datapunkt beräknar vi avståndet med hjälp av *RBF*-funktionen, se Ekvation 4.11 nedan, till våra båda *landmarks*.

$$\phi_\gamma(\vec{x}, \ell) = e^{-\gamma \|\vec{x} - \ell\|^2} \quad (4.11)$$

där  $\|\vec{x} - \ell\|$  är avståndet från en datapunkt,  $\vec{x}$ , till en *landmark*,  $\ell$ .  $\gamma$  (symbolen är den grekiska bokstaven “gamma”) är en hyperparameter som vi strax kommer gå igenom.

De beräknade avstånden, med *RBF*-funktionen, representerar koordinaterna i det nya koordinatsystemet. Den högra bilden i Figur 4.14 visar datapunkterna från den vänstra bilden efter att de transformeras med hjälp av *RBF* till en högre dimension. Daten är nu linjärt separerbar. Vi gör ett räkneexempel för att konkretisera hur data transformeras när vi använder *RBF*.



Figur 4.14: Transformering av data med hjälp av RBF. I den vänstra bilden har ett dataset som endast har en variabel,  $x_1$ , detta dataset transformeras (se pilen) till att ha två variabler,  $x_2$  och  $x_3$ . Efter transformationen blir datasetet linjärt separerbart.

Antag att datapunkten  $x$  från den vänstra bilden i Figur 4.14 har värdet  $x_1 = -1$  och att de två *landmarks* har värdet  $x_1 = -2$  och  $x_1 = 1$ . Vi antar att  $\gamma = 0.3$ . Då är avståndet från  $x_1 = -1$  till  $x_1 = -2$  lika med 1 och avståndet från  $x_1 = -1$  till  $x_1 = 1$  är lika med 2. Använder vi Ekvation 4.11 får vi då att:  $x_2 = e^{-0.3(1^2)} \approx 0.74$  och  $x_3 = e^{-0.3(2^2)} \approx 0.3$ . Det innebär att den datapunkt som representerades av  $x_1 = -1$  i den vänstra bilden nu alltså transformeras till koordinaten  $(0.74, 0.3)$  i den högra bilden.

Hur väljer vi *landmarks*? Ett enkelt tillvägagångssätt är att sätta varje observation till en landmark. Har vi  $M$  stycken observationer och  $N$  stycken variabler så kommer vårt dataset bli ”transformerat” (vi gör inte en egentlig transformation, vi har ju ”*kernel trick*”) till att ha  $M$  stycken observationer och  $M$  stycken variabler eftersom vi antar att de ursprungliga variablerna inte används. I praktiken kan detta dock leda till att det blir väldigt många variabler.

## i Intuition för varför nya variabler skapas

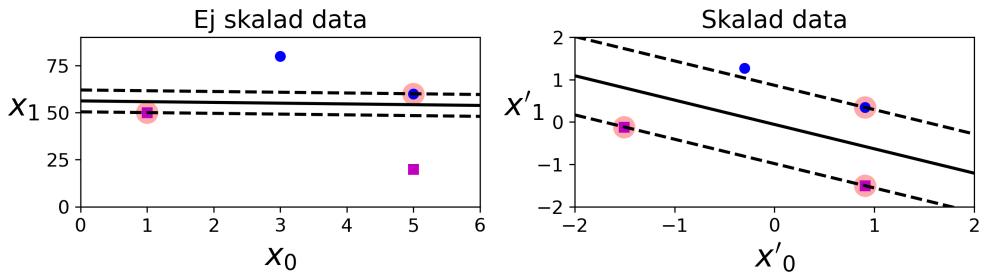
I exemplet vi gått igenom har vi lagt till variabler, antingen genom att använda polynomvariabler eller genom att skapa nya med hjälp av *RBF*. Detta är intuitivt rimligt. Anledningen är att när vi skapar nya variabler så får vi ett högre dimensionellt dataset vilket gör att det finns "mer plats", detta ökar chanserna att datan blir linjärt separerbar.

C, *kernel* och *gamma* tillhör de vanligaste hyperparametrarna att justera under modellering med *SVM*, se Tabell 4.2 där ett urval av hyperparametrar presenteras. Läsaren uppmanas att läsa *scikit-learn* dokumentationen för att få en helhetsbild.

Tabell 4.2: Ett urval av hyperparametrar för *SVC* från *scikit-learns* dokumentation.

Hyperparameter	Beskrivning	Standardvärde
C	Regulariseringssstyrka. Ju lägre C, desto mjukare marginal (tillåter fler fel). Av typen float.	1.0
<i>kernel</i>	Val av kernel. {'linear', 'poly', 'rbf', 'sigmoid', 'precomputed'} eller callable.	"rbf"
<i>gamma</i>	Gäller för "rbf", "poly" och "sigmoid". {'scale', 'auto'} eller float.	"scale"
<i>degree</i>	Polynomgrad. Gäller för kernel "poly". Av typen int.	3
<i>decision_function_shape</i>	Multiklasstrategi. {'ovo', 'ovr'}	"ovr"

Vi använder återigen datan skapad med `make_moons` för att illustrera hur en *SVM*-modell kan instantieras, användas för prediktion och slutligen utvärderas. Detta kan göras på liknande sätt som för logistisk regression. För att göra exemplet något mer avancerat och demonstrera hyperparametrar inkluderas även `GridSearch`. När vi använder oss av en *SVM*-modell är det en tumregel att data ofta standardiseras eftersom *SVM* modeller är känsliga för skalaten på data. Kollar vi på Figur 4.15 ser vi varför standardisering är intuitivt rimligt att genomföra. Vi börjar med att titta på en linjär *SVM*-modell.



Figur 4.15: I den vänstra bilden ser vi ett exempel på oskalad data och att det blir svårt att skapa en bra ”väg” på grund av att skalan på datan skiljer sig åt. När datan standardiseras genom skalning så blir det lättare att skapa en bra ”väg”, se den högra bilden. En tumregel är därför att datan skalas, till exempel genom att använda `StandardScaler` från scikit-learn, i samband med SVM modellering.

```

from sklearn.svm import LinearSVC
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import GridSearchCV          ①

model_linear_svc = make_pipeline(
    StandardScaler(),
    LinearSVC()
)                                                       ②

hyperparam_grid = {
    'linearsvc__C': [0.01, 0.1, 1]
}                                                       ③

model_linear_svc_grid_search = GridSearchCV(model_linear_svc,
                                             hyperparam_grid, cv=5, scoring='accuracy') ④

model_linear_svc_grid_search.fit(X_train, y_train)        ⑤

y_pred = model_linear_svc_grid_search.predict(X_test)    ⑥

accuracy = accuracy_score(y_test, y_pred)                 ⑦

```

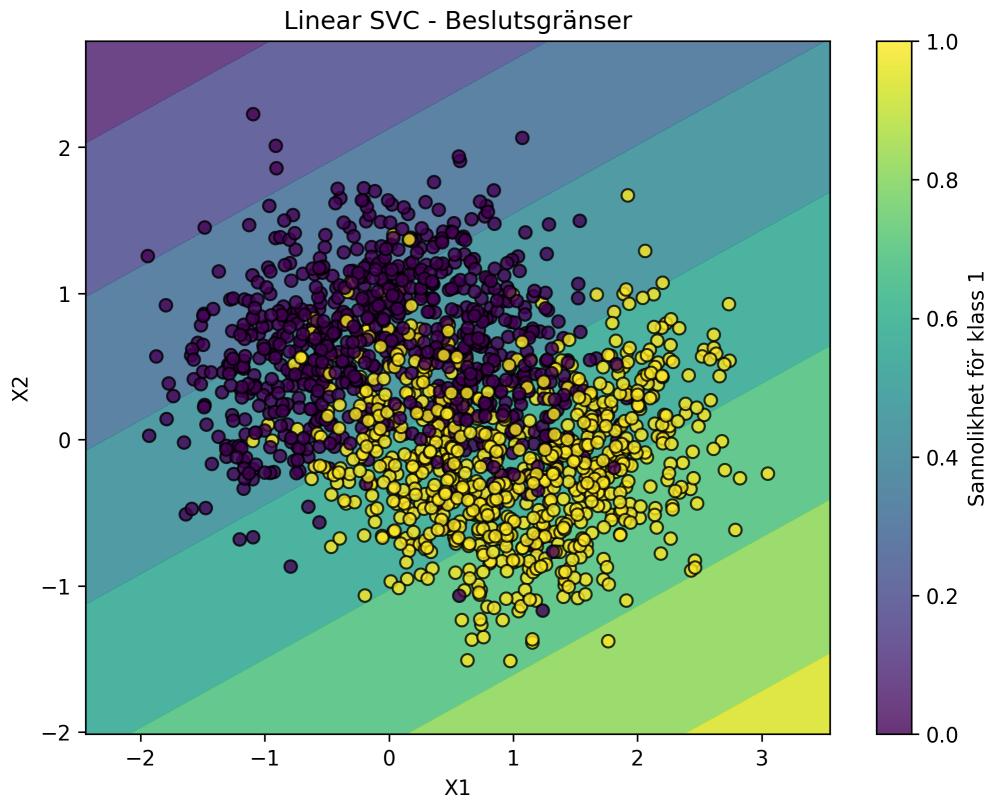
```
print(f"Test Accuracy: {accuracy:.2f}")          ⑧
print("Best hyperparameters:",
      ↪ model_linear_svc_grid_search.best_params_) ⑨
```

- ① Importera `LinearSVC`, `make_pipeline` samt `GridSearch` från *scikit-learn* biblioteket. Notera att modellen i *scikit-learn* kallas för *SVC*, *support vector classifier*, till skillnad från tidigare kapitel när *SVR*, *support vector regressor*, användes.
- ② Vi skapar en *pipeline* som först skalar datan innan modellen tränas.
- ③ Specificera hyperparametrar som kommer att utvärderas med `GridSearch`.
- ④ Skapa ett `GridSearchCV`-objekt.
- ⑤ Träna modellen.
- ⑥ Utvärdera modellen på testdata.
- ⑦ Beräkna *accuracy score*.
- ⑧ Skriv ut *accuracy score*.
- ⑨ Skriv ut de bästa hyperparametrarna från `GridSearch`.

```
Test Accuracy: 0.83
Best hyperparameters: {'linearsvc__C': 0.1}
```

*Accuracy score* visar att modellen klassificerar 83% av alla datapunkter i testdata korrekt. Se beslutsgränser i Figur 4.16. Precis som för logistisk regression felklassificerar modellen en relativt stor andel av datapunkterna i och med att vi använt en linjär modell (med linjär beslutsgräns) medan datan uppvisar icke-linjära mönster.

För att även demonstrera hur olinjära dataset kan hanteras kommer följande kodexempel att använda sig av en kernel. Det är möjligt att sätta kernel till `linear` i nedanstående modell och på så sätt få en linjär modell likt ovan. Genom att använda oss av en kernel i detta fall får vi en något högre *accuracy score* jämfört med tidigare kodexempel. Se beslutsgränser i Figur 4.17 .



Figur 4.16: Beslutsgränser för Linear SVC.

```

from sklearn.pipeline import make_pipeline
from sklearn.svm import SVC

model_svc = make_pipeline(
    StandardScaler(),
    SVC()
)

hyperparam_grid = {
    'svc__kernel': ['rbf', 'poly', 'sigmoid'],
    'svc__C': [0.1, 1, 10],
    'svc__gamma': [0.01, 0.1, 1]
}

model_svc_grid_search = GridSearchCV(model_svc, hyperparam_grid,
                                       cv=5, scoring='accuracy')

model_svc_grid_search.fit(X_train, y_train)

y_pred = model_svc_grid_search.predict(X_test)

accuracy = accuracy_score(y_test, y_pred)
print(f"Test Accuracy: {accuracy:.2f}")
print("Best hyperparameters:", model_svc_grid_search.best_params_)

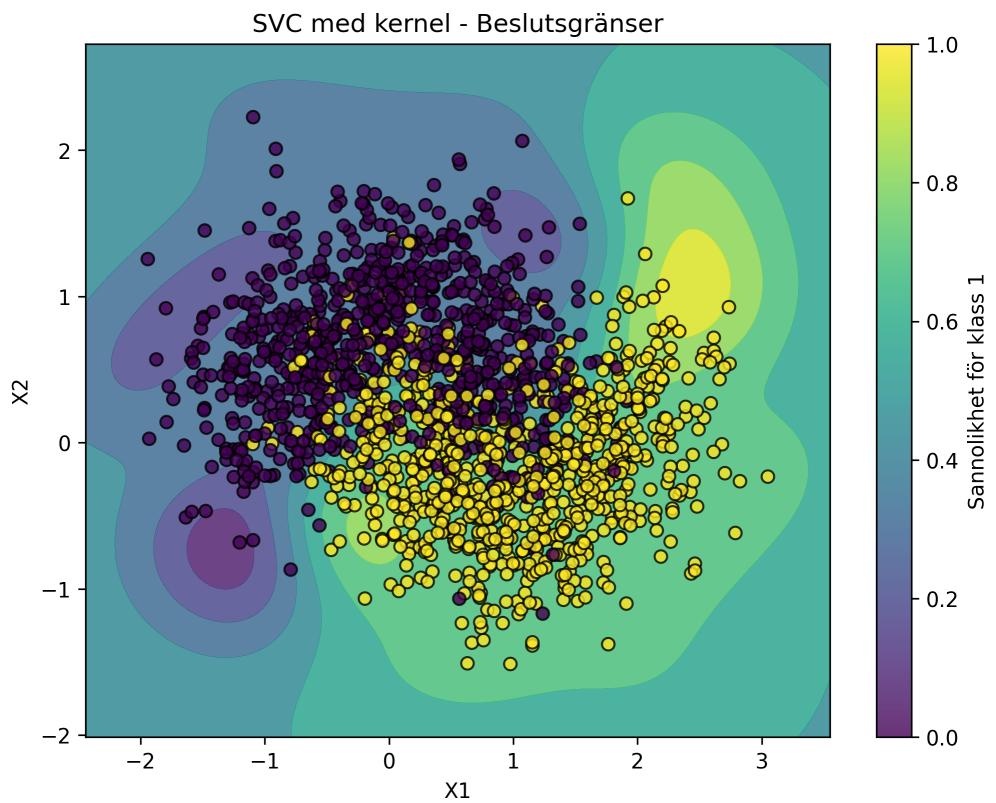
```

Test Accuracy: 0.86

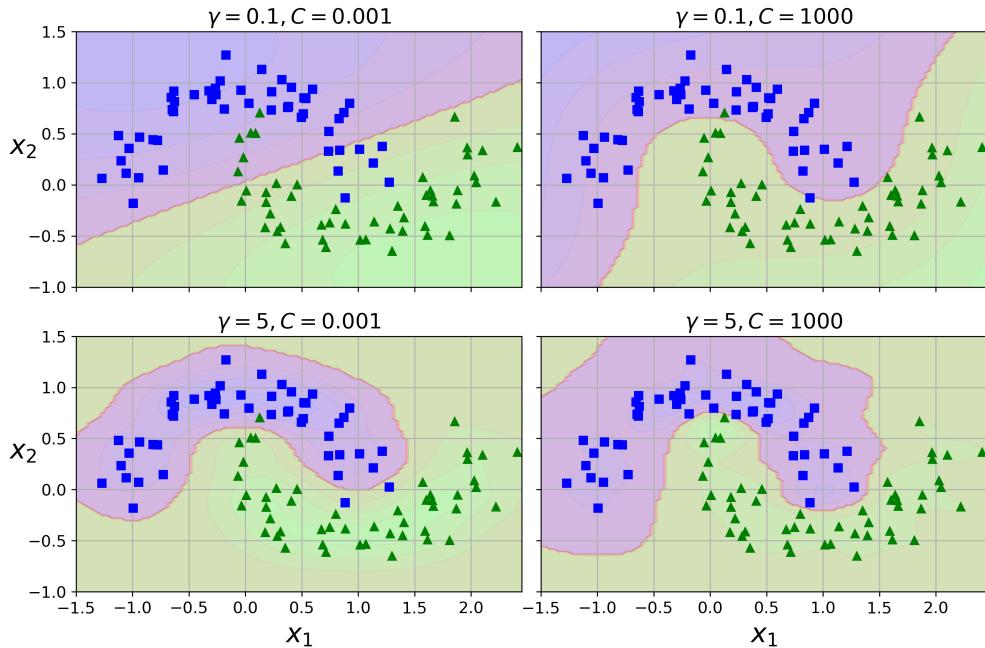
Best hyperparameters: {'svc\_\_C': 10, 'svc\_\_gamma': 1, 'svc\_\_kernel': 'rbf'}

Avslutningsvis, i Figur 4.18 visas exempel på olika värden för **Gamma** och **C** för en *SVM*-modell som använder sig av en *RBF*-kernel. Om en modell är under- eller överanpassad är det möjligt att justera hyperparametrarna **Gamma** och **C** för att förbättra modellens generaliseringsförmåga på ny, osedd data. Rent praktiskt gör vi det i vanlig ordning med hjälp av *grid search* från *scikit-learn*. Från figuren ser vi att:

- När **C** ökar så ökar flexibiliteten. Om modellen är överanpassad minskar vi därför värdet på **C** och vice versa, är den underanpassad ökar vi värdet på **C**.
- När  $\gamma$  ökar så ökar flexibiliteten. Om modellen är överanpassad minskar vi därför värdet på  $\gamma$  och vice versa, är den underanpassad ökar vi värdet på  $\gamma$ .



Figur 4.17: Beslutsgränser för en SVM-modell där kernel används. Vi ser att beslutsgränserna är icke-linjära.



Figur 4.18: Beslutsgränser för SVM-modellen för olika värden på  $C$  och  $\Gamma$ . Är vår modell överanpassad kan vi prova att minska värdet på  $\gamma$  och/eller  $C$  och vice versa om den är underanpassad, då kan vi prova att öka värdet på  $\gamma$  och/eller  $C$ . I praktiken väljer vi optimala värden på  $\gamma$  och  $C$  med hjälp av grid search.

**i** Notera att en linjär *SVM*-modell kan skapas både genom `LinearSVC` och `SVC(kernel=linear)`. Om vi redan i förväg vet att vi ska skapa en linjär *SVM*-modell bör vi använda `LinearSVC` eftersom den är optimerad för detta och därför går snabbare. I de fall vi vill utvärdera om en linjär eller icke-linjär modell är bättre använder vi `SVC` där olika *kernels* (exempelvis “linear” eller “poly”) kan utvärderas.

### 4.3.3 Beslutsträd

Beslutsträd (eng. *decision trees*) kan, precis som *SVM*, hantera både regressions- och klassificeringsproblem och introducerades i föregående kapitel. Ett beslutsträd har samma generella struktur för både regressions- och klassificeringsproblem, där trädet består av noder. Den första noden benämns rotnod och de sista noderna benämns lövnoder. För att genomföra en klassificering med beslutsträd startar processen vid rotnoden och går därefter via inre noder nedåt för att slutligen komma till en lövnod, som representerar en predikterad klass.

När vi skapar beslutsträd via *scikit-learn* bör vi generellt sett använda *grid search* för att optimera hyperparametrar. Anledningen är att trädet annars kan bli väldigt djupt på grund av hur standardvärdena för hyperparametrarna har blivit satta i *scikit-learn*, detta kan leda till att modellen blir överanpassad. Se Tabell 4.3 där de vanligast förekommande hyperparametrarna vi optimerar för beslutsträd presenteras. Läsaren uppmanas att läsa *scikit-learn* dokumentationen för att få en helhetsbild.

Tabell 4.3: Ett urval av hyperparametrar för beslutsträd från scikit-learns dokumentation.

Hyperparameter	Beskrivning	Standardvärde
<code>max_depth</code>	Maximalt antal nivåer ett träd får ha. Ett högre värde gör att modellen kan lära sig fler detaljer, men det finns en risk att modellen blir överanpassad. Av typen int.	“None”
<code>min_samples_split</code>	Minimalt antal samples en nod behöver ha för att kunna delas. Högre värden gör trädet mindre djupt. Av typen int eller float.	2
<code>min_samples_leaf</code>	Minsta antalet observationer en lövnod behöver ha. Av typen int eller float.	1
<code>max_leaf_nodes</code>	Största antalet lövnoder ett träd får ha. Av typen int.	“None”
<code>max_features</code>	Antalet oberoende variabler/features som övervägs när modellen ska hitta den optimala delningen för en nod. int, float eller {“sqrt”, “log2”}.	“None”

Läser vi dokumentationen, kan vi bland annat utläsa nedanstående, vilket förklrar varför överanpassning kan ske om vi inte optimerar hyperparametrar med *grid search*:

**`max_depth` int : default=None**

The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than `min_samples_split` samples.

Vi använder nu datasetet skapat med `make_moons` för att illustrera hur modellering med beslutsträd kan se ut.

```

from sklearn.tree import DecisionTreeClassifier

tree_clf = DecisionTreeClassifier(max_depth=2)
tree_clf.fit(X_train, y_train)
y_pred = tree_clf.predict(X_test)

accuracy = accuracy_score(y_test, y_pred)
print(f"Test Accuracy: {accuracy:.2f}")

y_proba = tree_clf.predict_proba(X_test)
print("Predicted probabilities (first 5 rows):\n",
      np.round(y_proba[:5], 3))
```

①

- ① Beräkna och skriv ut sannolikheten för varje klass. Från Figur 4.20 ser vi hur sannolikheterna för respektive klass beräknas. Vi tar antalet observationer för respektive klass i lövnoderna dividerat med totala antalet observationer i den lövnode. Exempelvis,  $588/665 \approx 0.884$  och  $77/665 \approx 0.116$ .

```

Test Accuracy: 0.85
Predicted probabilities (first 5 rows):
[[0.163 0.837]
 [0.163 0.837]
 [0.884 0.116]
 [0.163 0.837]
 [0.163 0.837]]
```

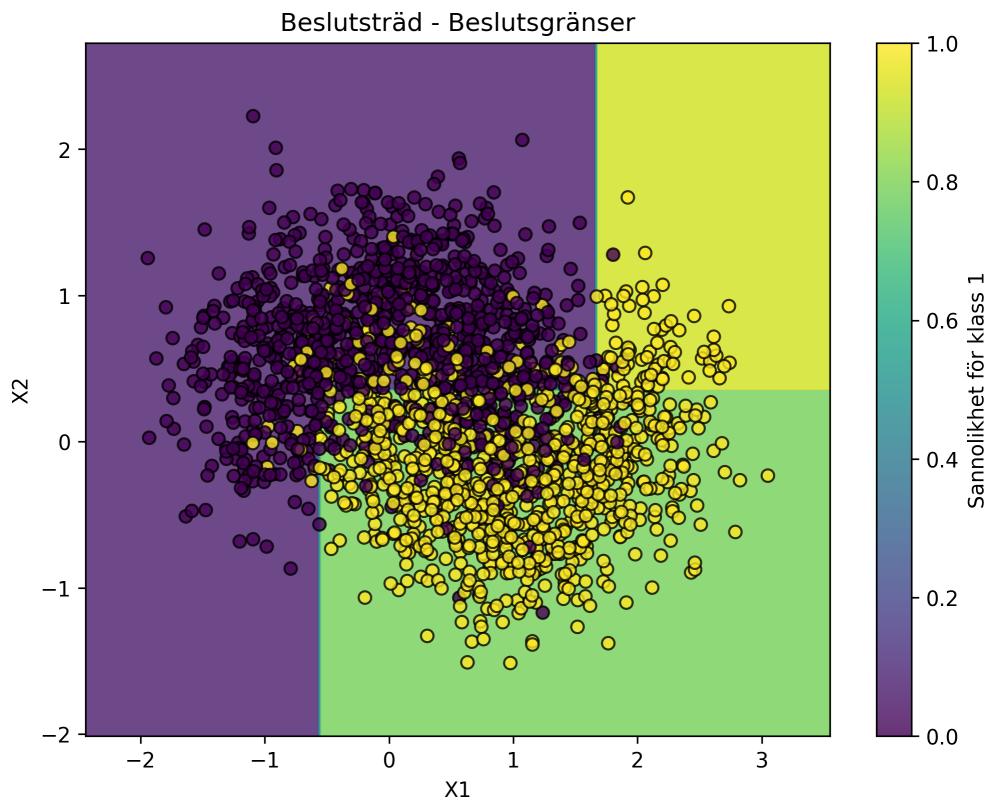
Modellen lyckas med att klassificera 85% av alla datapunkter korrekt. I Figur 4.19 ser vi hur beslutsgränserna för beslutsträdet ser ut.

Koden nedan visualiseras det tränade beslutsträdet, se Figur 4.20.

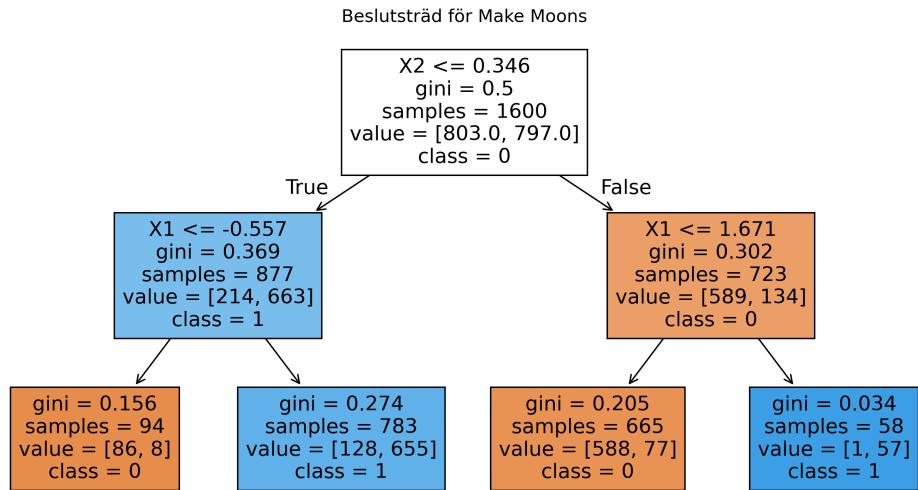
```

from sklearn.tree import plot_tree

plt.figure(figsize=(12, 6))
plot_tree(tree_clf, filled=True, feature_names=["X1", "X2"],
          class_names=["0", "1"])
plt.title("Beslutsträd för Make Moons")
plt.show()
```



Figur 4.19: Beslutsgränser för ett beslutsträd.



Figur 4.20: Visualisering av hur ett beslutsträd kan se ut.

Vid klassificering av nya observationer kommer modellen att vandra genom trädet som visualiseras i Figur 4.20, uppifrån och ned, för att ta ett beslut om vilken klass en datapunkt tillhör. Processen startar i den översta noden, rotnoden, där frågan “är  $X_2$  mindre än eller lika med  $0.346$ ?” utvärderas. Är svaret sant går den till vänster, annars till höger. Om vi antar att den gått till vänster så ställer den frågan “är  $X_1$  mindre än eller lika med  $-0.557$ ?”. Återigen, om svaret är sant går den till vänster, annars till höger. Om vi antar att det är falskt så går vi till höger och där ser vi att den predikterade klassen är 1. Nedan förklaras informationen som finns i noderna mer ingående.

- **Tröskelvillkor** anger en indatavariabel och dess tröskelvärde. Detta används för att dela upp data i två grenar. När uttrycket är sant går modellen till vänster, och när det är falskt går modellen till höger. I den första noden, rotnoden, är tröskelvillkoret  $X_2 \leq 0.346$ .
- **Gini** är ett index som mäter hur “oren” en nod är. Om  $\text{gini} = 0$  tillhör alla datapunkter samma klass och noden är alltså ren (innebärande att orenheten = 0). Ju längre ned i trädet vi går desto mindre blir värdet på gini-koefficienten. Gini-koefficienten för nod  $i$ ,  $G_i$ , beräknas enligt Ekvation 4.12.

$$G_i = 1 - \sum_{k=1}^K p_{i,k}^2 \quad (4.12)$$

där  $p_{i,k}$  är antalet observationer från klass  $k$  (som hämtas från "value" i respektive nod) dividerat med det totala antalet observationer i nod  $i$  (som hämtas från "samples" i respektive nod). Exempelvis, i den högra, mittersta noden blir gini-koefficienten 0.302 som vi ser enligt följande beräkning:

$$G_i = 1 - \left[ \left( \frac{589}{723} \right)^2 + \left( \frac{134}{723} \right)^2 \right] = 0.302 \quad (4.13)$$

När modellen tränas kommer den att välja den variabel (till exempel X1 eller X2) och det tröskelvärde (exempelvis 0.346), som producerar de renaste noderna (mäts med gini koefficienten), för varje förgrening.

- **Samples** anger totala antalet datapunkter som finns i noden.
- **Value** anger antalet datapunkter som finns i respektive klass.
- **Class** anger den klass som flest datapunkter i den specifika noden tillhör. Det är denna klass som trädet kommer att prediktera om vi stannar på den noden.

Med hjälp av beslutsträd kan vi även beräkna sannolikheter. Om vi kollar på den lövnod som är längst till vänster i Figur 4.20, så har vi sannolikheterna  $P(0) = 86/94 \approx 0.915$  och  $P(1) = 8/94 \approx 0.085$ . Vi noterar att sannolikhetsmåttet kan vara "stelt"/"offlexibelt" eftersom alla observationer som har  $X2 \leq 0.346$  och  $X1 \leq -0.557$  kommer att ha samma sannolikheter.

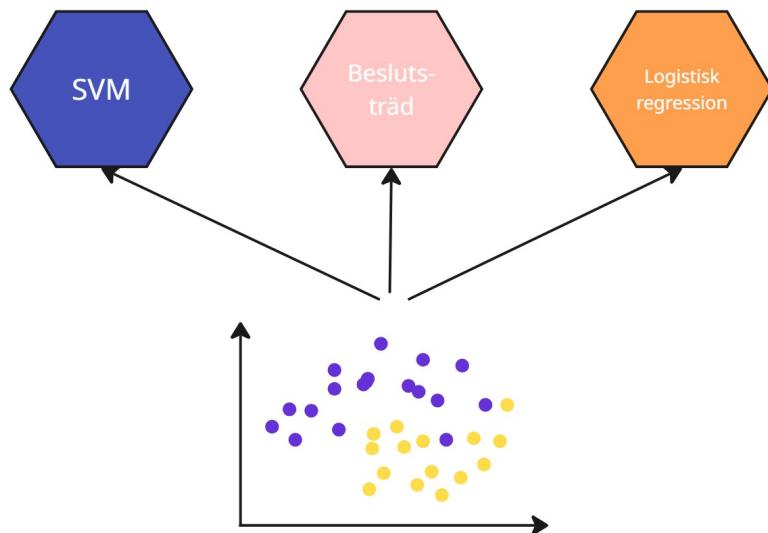
Notera hur Figur 4.20 korresponderar med Figur 4.19. I beslutsträdet från Figur 4.20 ser vi att frågorna:  $X2 \leq 0.346$ ,  $X1 \leq -0.557$ ,  $X1 \leq 1.671$  ställs och dessa värden motsvarar värdena för linjerna i Figur 4.19.

Beslutsträd är den fundamentala byggstenen i *Random Forest*-modeller, som är en *ensemble learning*-metod. Vi fortsätter nu med att kolla på *ensemble learning* innan vi fortsätter med *random forest*.

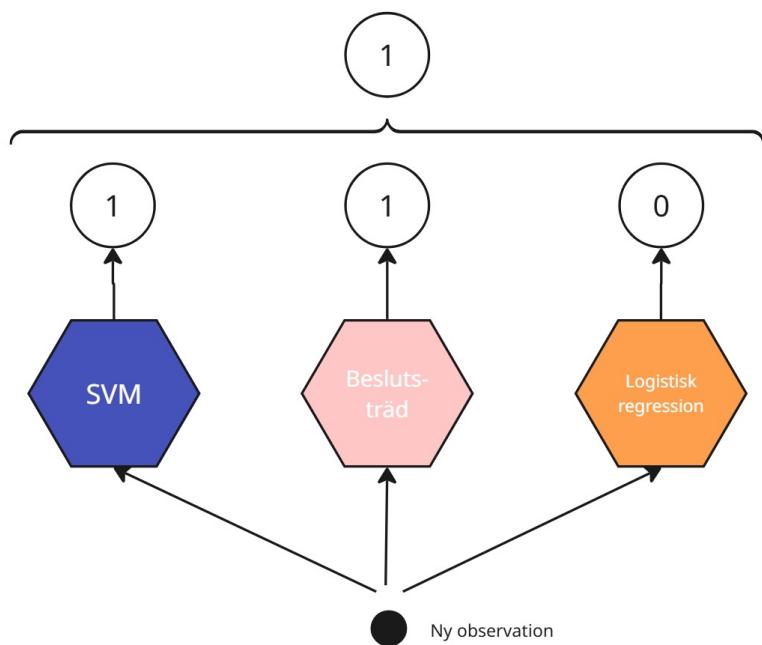
#### 4.3.4 Ensemble learning

I föregående kapitel introducerades *Ensemble Learning*, en metod där flera modeller kombineras med målet att skapa en modell med bättre prediktions- och generaliseringssförmåga än varje enskild modell, se Figur 4.21.

För regression gjordes en prediktion baserat på medelvärdet för samtliga modellers prediktioner. I klassificering använder vi istället det som benämns för *voting classifier* där det finns det som kallas för *hard voting classifier* och *soft voting classifier*. Det finns två typer av *voting*. När varje modell predikterar en klass och den klass som flest modeller predikterar är den som väljs kallas det för majoritetsröstning, *hard voting* på engelska. Detta visas i Figur 4.22 där vi ser att två modeller predikterar 1 medan en modell predikterar 0. Därför har 1 fått flest röster och det blir den slutgiltiga prediktionen. Om varje modell istället ger en sannolikhet för varje klass kan ett genomsnitt beräknas och den klass med högst genomsnittlig sannolikhet väljs. Detta blir alltså en sannolikhetsbaserad röstning, och det benämns för *soft voting* på engelska. Notera att *soft voting* förutsätter att samtliga modeller faktiskt kan uppskatta sannolikheter.



Figur 4.21: Flera olika klassificeringsmodeller tränas separat på träningsdata för att sedan kunna kombineras till en.



Figur 4.22: Exempel på klassificering med majoritetsröstning, hard voting.

## i Intuition för *hard voting* klassificering

Genom att använda kunskap som flera modeller har är det intuitivt rimligt att vi får bättre resultat genom att kombinera dessa med hjälp av *hard voting*. Den läsare som är bekant med mer avancerad statistik och känner till binomialfördelningen, kan även kolla på nedanstående exempel för att få ytterligare intuition.

Antag att vi har 1000 stycken oberoende binära klassificerare, där varje klassificerare har en sannolikhet på 51% att förutsäga rätt klass. Det innebär att varje enskild klassificerare i sig är ganska dålig.

Frågan är: Vad är sannolikheten att vi predikterar rätt klass om vi använder oss av *hard voting classifier*?

- För att den sammanslagna klassificeringen (genom majoritetsröstning) ska bli korrekt, krävs det att fler än 500 klassificerare predikterar korrekt.
- Vi modellerar antalet korrekta klassificerare som en binomialfördelad slumpvariabel.

Låt  $X$  vara antalet klassificerare som predikterar rätt klass. Då gäller det att:

$$X \sim \text{Bin}(n = 1000, p = 0.51)$$

Vi söker sannolikheten:

$$P(X > 500) = 1 - P(X \leq 500)$$

Vi kan använda Python (med `scipy.stats`) för att räkna ut detta:

```
from scipy.stats import binom  
print(1 - binom.cdf(500, n=1000, p=0.51))
```

0.7260985557304961

Sannolikhet är alltså cirka 72.6% att vårt *ensemble* predikterar korrekt, trots att varje individuell modell bara har 51% träffssäkerhet.

I verkligheten är antagandet om oberoende klassificerare orealistiskt, men exemplet ger en bra teoretisk intuition för varför *ensemble learning* kan vara kraftfullt även när varje enskild klassificerare är relativt dålig.

När vi använder oss utav `VotingClassifier` i *scikit-learn* finns det ett antal hyperparametrar som kan justeras. I Tabell 4.4 ser vi några vanligt förekommande. Notera

att hyperparametern `estimators` är obligatorisk att ange. Läsaren uppmanas att läsa *scikit-learn* dokumentationen för att få en helhetsbild.

Tabell 4.4: Ett urval av hyperparametrar för *VotingClassifier* från scikit-learns dokumentation.

Hyperparameter	Beskrivning	Standardvärde
<code>estimators</code>	De modeller, med tillhörande namn, som ska användas. Hyperparametern är obligatorisk, en lista med modeller måste angas. Lista med (str, estimator) tuples.	N/A
<code>voting</code>	Vilken typ av <i>voting</i> . {'hard', "hard" 'soft'}. En lista med vikter per modell om <i>soft voting</i> används.	"hard"
<code>weights</code>		"None"

Koden nedan instantierar och tränar tre olika modeller. De tre modellerna används också för att skapa en *voting classifier*. Precis som tidigare används datan skapad med `make_moons`.

```
from sklearn.ensemble import VotingClassifier          ①

log_clf = LogisticRegression()
tree_clf = DecisionTreeClassifier()
svm_clf = SVC()                                       ②

voting_clf = VotingClassifier(
    estimators=[('lr', log_clf), ('tree', tree_clf), ('svc',
        svm_clf)],
    voting='hard')                                     ③

for clf in (log_clf, tree_clf, svm_clf, voting_clf):
    clf.fit(X_train, y_train)                         ④

y_pred = clf.predict(X_test)                          ⑤
```

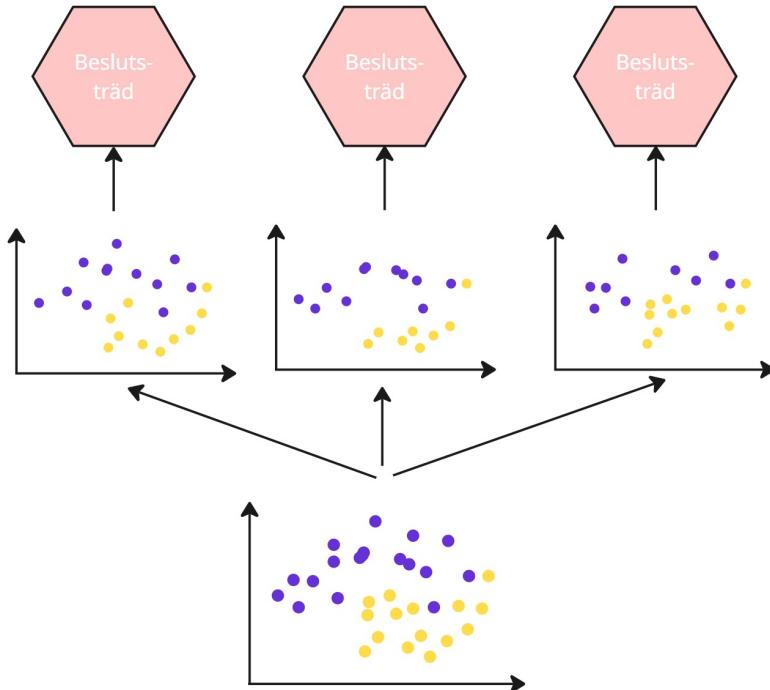
```
print(clf.__class__.__name__, accuracy_score(y_test, y_pred))  
⑥
```

- ① Importera *voting classifier* från *scikit-learn* biblioteket.
- ② Instantiera modellerna logistisk regression, beslutsträd och *SVM*.
- ③ Instantiera en *voting classifier* där hyperparametern voting specificerats till **hard** vilket innebär att vi kommer använda oss av en *hard voting* klassificerare.
- ④ Träna modellerna.
- ⑤ Utvärdera modellerna på testdatan.
- ⑥ Beräkna och skriv ut *accuracy* för respektive modell.

```
LogisticRegression 0.835  
DecisionTreeClassifier 0.7825  
SVC 0.86  
VotingClassifier 0.85
```

Två andra ensemblemetoder är det som kallas för *bagging* och *pasting*, vilket introducerades i föregående kapitel om regression. Vi repeterar kort vad de innebär här. Ovan tränade vi olika modeller på samma uppsättning av träningsdata. I denna metod används samma sorts modell för varje prediktion, men varje modell tränas på olika slumpmässigt valda urval av träningsdata. När urval sker med återläggning kallas det för *bagging*, och när urval görs utan återläggning kallas det för *pasting*. Se Figur 4.23 som illustrerar detta.

För *bagging* och *pasting* används **BaggingClassifier** i *scikit-learn*. Använd hyperparametern **bootstrap** för att välja vilken metod som ska användas. Den och ett urval av andra hyperparametrar kan ses i Tabell 4.5. Läsaren uppmanas att läsa *scikit-learn* dokumentationen för att få en helhetsbild.



*Figur 4.23: Vi har ett givet dataset som vi tar slumpmässiga urval från för att skapa nya dataset. Om urvalet sker med återläggning benämns det bagging och om urvalet sker utan återläggning benämns det pasting. Därefter kan vi välja en modell, exempelvis ett beslutsträd, och träna modellen för varje dataset. I figuren har vi alltså tränat tre beslutsträd. När vi ska prediktera en ny observation kan vi kombinera prediktionerna från de tre beslutsträderna genom hard voting eller soft voting.*

Tabell 4.5: Ett urval av hyperparametrar för *BaggingClassifier* från scikit-learns dokumentation.

Hyperparameter	Beskrivning	Standardvärde
<code>bootstrap</code>	Vilken metod som ska användas, med ( <i>bagging</i> ) eller utan ( <i>pasting</i> ) återläggning. <code>True</code> = <i>bagging</i> . <code>False</code> = <i>pasting</i> . Av typen bool.	<code>True</code>
<code>estimator</code>	Vilken typ av grundmodell som ska replikeras.	<code>None</code> -> <code>DecisionTreeClassifier</code>
<code>n_estimators</code>	Antalet modeller som tränas. Fler modeller ger en högre stabilitet, men gör att träningen går längsammare. Av typen int.	10
<code>max_samples</code>	Antal datapunkter som ska dras för varje urval. Av typen int eller float.	default=1.0

Koden nedan visar ett exempel på när *BaggingClassifier* används. `bootstrap` har satts till `True` och det är således ensemblemetoden *bagging* som används.

```
from sklearn.ensemble import BaggingClassifier          ①

bag_clf = BaggingClassifier(
    DecisionTreeClassifier(), n_estimators=100,
    bootstrap=True)                                     ②

bag_clf.fit(X_train, y_train)
y_pred = bag_clf.predict(X_test)

accuracy = accuracy_score(y_test, y_pred)
print(f"Test Accuracy: {accuracy:.2f}")
```

- ① Importera *bagging classifier* från *scikit-learn* biblioteket.
- ② Instantiera en *bagging classifier*. Modellen gör urval med återläggning eftersom hyperparametern `bootstrap` satts till `True` och totalt kommer 100 stycken dataset

skapas eftersom `n_estimators=100`, det innebär att vi kan träna 100 stycken modeller.

Test Accuracy: 0.84

I koden ovan har vi alltså kombinerat flera olika beslutsträd, som vi kommer se i nästa avsnitt är det alltså en typ av *random forest*-modell vi manuellt har skapat. Notera liknelsen, flera träd (beslutsträd) blir en skog (*random forest*).

En sista ensemblemetod som vi kommer att nämna är *boosting*. Syftet är endast att nämna det så att den intresserade läsaren känner till det och kan fördjupa sig i området. För de metoderna vi hittills har gått igenom, har modellerna tränats *parallel*. *Boosting* tränar istället modeller *sekventiellt*. Detta gör det möjligt för varje modell att lära sig från tidigare modell och på så sätt kan den korrigera fel och bli bättre. En *boosting*-modell som blivit mycket populär är XGBoost (*Extreme Gradient Boosting*). Denna finns tillgänglig i Python-biblioteket `xgboost`. Koden nedan visar ett exempel på hur en `XGBClassifier` används och vi ser att det följer samma metodologi som för `scikit-learn`. Notera att det även finns `XGBRegressor` som hanterar regressionsproblem.

```
import xgboost as xgb  
①  
xgb_clf = xgb.XGBClassifier()  
②  
xgb_clf.fit(X_train, y_train)  
y_pred = xgb_clf.predict(X_test)  
  
accuracy = accuracy_score(y_test, y_pred)  
print(f"Test Accuracy: {accuracy:.2f}")
```

- ① Importera `xgboost`.
- ② Instantiera en `XGBClassifier`.

Test Accuracy: 0.84

#### 4.3.5 Random forest

*Random forest* är ett *ensemble* av beslutsträd som oftast skapas genom *bagging* även om *pasting* också förekommer. I föregående avsnitt hade vi koden nedan. Kollar vi på koden ser vi att det är flera beslutsträd kombinerade, vi har alltså manuellt skapat en *random forest*-modell.

```

from sklearn.ensemble import BaggingClassifier

bag_clf = BaggingClassifier(
    DecisionTreeClassifier(), n_estimators=100,
    bootstrap=True)
bag_clf.fit(X_train, y_train) (2)

y_pred = bag_clf.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f"Test Accuracy: {accuracy:.2f}")

```

Vill vi använda oss av en *random forest*-modell så finns det dock en specifik klass för detta i *scikit-learn*, `RandomForestClassifier`, detta är en optimerad implementation för just *random forest*-modeller. En viktig aspekt skiljer sig dock.

I Avsnitt 4.3.3, om beslutsträd, skrev vi följande:

*“När modellen tränas kommer den att välja den variabel (till exempel X1 eller X2) och det tröskelvärde (exempelvis 0.346), som producerar de renaste noderna (mäts med gini koefficienten), för varje förgrening.”*

Kollar vi på Figur 4.20, innebär det att modellen väljer den variabel och det tröskelvärde som producerar de renaste noderna för varje förgrening/split. Renheten av noderna mäts med gini-koefficienten, Ekvation 4.12.

Kollar vi på dokumentationen från *scikit-learn* så ser vi att nedanstående hyperparametrar finns för `RandomForestClassifier`.

```

class sklearn.ensemble.RandomForestClassifier(n_estimators=100, *,
      ↴ criterion='gini', max_depth=None, min_samples_split=2,
      ↴ min_samples_leaf=1, min_weight_fraction_leaf=0.0,
      ↴ max_features='sqrt', max_leaf_nodes=None,
      ↴ min_impurity_decrease=0.0, bootstrap=True, oob_score=False,
      ↴ n_jobs=None, random_state=None, verbose=0, warm_start=False,
      ↴ class_weight=None, ccp_alpha=0.0, max_samples=None,
      ↴ monotonic_cst=None)

```

Det står alltså `max_features='sqrt'`. Det innebär att endast ett urval av *features* betraktas för varje förgrening. Specificerar vi dock `None` kommer samtliga features betraktas för varje förgrening, precis som för beslutsträd.

Om vi för varje förgrening/split, förutom en slumpmässig grupp av *features* även väljer ett slumpmässigt värde på *tröskelvärdet* får vi den modell som benämns *Extra Tree* (*Extremely Randomized Trees*). Den implementeras genom `ExtraTreesClassifier` i *scikit-learn*. Eftersom optimalt tröskelvärde inte behöver hittas går denna modellen generellt sett snabbare att träna. En naturlig fråga att ställa sig är varför man skulle vilja använda en slumpmässig grupp av *features* och ett slumpmässigt tröskelvärde för varje förgrening? Anledningen är att detta medför att vi generellt sett får en högre bias men lägre varians, totalt sett kan alltså slutresultatet bli bättre på grund av *bias-variance trade-off*, se Avsnitt 3.3.3.

Vi demonstrerar modellerna med datasetet som skapades via `make_moons`.

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import ExtraTreesClassifier

rf_clf = RandomForestClassifier(n_estimators=100,
    ↵ max_leaf_nodes=16, random_state=12)
ert_clf = ExtraTreesClassifier(n_estimators=500,
    ↵ max_leaf_nodes=16, random_state=22)

rf_clf.fit(X_train, y_train)
ert_clf.fit(X_train, y_train)

y_pred_rf = rf_clf.predict(X_test)
y_pred_ert = ert_clf.predict(X_test)

accuracy_rf = accuracy_score(y_test, y_pred_rf)
accuracy_ert = accuracy_score(y_test, y_pred_ert)

print("Accuracy (Random forest):", round(accuracy_rf, 2))
print("Accuracy (Extra tree):", round(accuracy_ert, 2))
```

```
Accuracy (Random forest): 0.85
Accuracy (Extra tree): 0.86
```

Med hjälp av beslutsträd, *random forest* eller *extra tree* kan vi få en uppskattning på hur viktiga olika *features* är genom att använda oss av `feature_importances_`. Då får vi ut siffror, där högre siffra indikerar att variabeln är viktigare. Tillsammans summerar även siffrorna till 1. Se koden nedan.

```
print(rf_clf.feature_importances_)
print(ert_clf.feature_importances_)

[0.43042283 0.56957717]
[0.46523419 0.53476581]
```

Har vi ett dataset med flera variabler så kan alltså `feature_importances_` hjälpa oss att välja vilka variabler vi vill inkludera. Trots att, exempelvis en *random forest*-modell, inte ska användas för prediktioner så kan en sådan modell tränas för att vi ska få ut `feature_importances_` som kan hjälpa oss genomföra variabelselektion.

## 4.4 Två kodexempel

I detta avsnitt kommer vi kolla på två kodexempel. I det första genomförs en klassificering av datasetet *MNIST* som består av handskrivna siffror, och i det andra demonstreras hur vi kan få önskade värden på *precision* eller *recall* med våra modeller.

### 4.4.1 Kodexempel 1 - Klassificering av *MNIST*

*MNIST* är ett mycket vanligt förekommande dataset inom maskininlärning för att testa och demonstrera olika modeller för klassificering inom bildigenkänning. Datasetet består av handskrivna siffror där varje bild är  $28 \times 28 = 784$  pixlar stor, vilket betyder att datasetet består av 784 olika variabler. Bilderna är ritade i gråskala. Varje bild ska klassificeras till en siffra mellan 0 och 9. Detta innebär att vi har ett multiklass klassifieringsproblem.

Vi börjar med att importera de bibliotek vi kommer använda. Vi definierar även en funktion som vi senare kommer använda för att visualisera *confusion matrices*.

```
# Imports
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib as mpl

from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
```

```

from sklearn.metrics import confusion_matrix,
    ↵ ConfusionMatrixDisplay
from sklearn.metrics import classification_report

from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier,
    ↵ ExtraTreesClassifier
from sklearn.ensemble import VotingClassifier

# Define a function that will be used later
def display_confusion_matrix(y_true, y_pred):
    cm = confusion_matrix(y_true, y_pred)
    ConfusionMatrixDisplay(cm).plot()

```

Därefter laddar vi in datan och gör några inspektioner för att få en känsla för den.

```

# Load data and inspect it
mnist = fetch_openml('mnist_784', version=1, cache=True,
    ↵ as_frame=False)                                ①
X = mnist["data"][:10000]
y = mnist["target"][:10000].astype(np.uint8)
print(X.shape)                                     ②
print(y.shape)

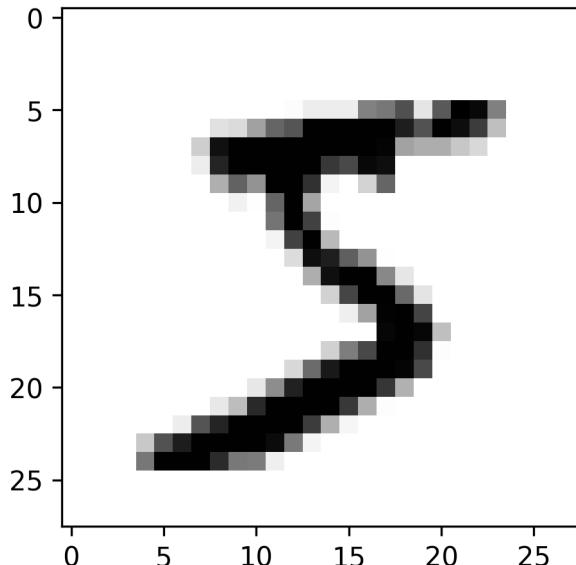
some_digit = X[0]
some_digit_image = some_digit.reshape(28, 28)
plt.imshow(some_digit_image, cmap=mpl.cm.binary)    ③
print("True label for the plotted image is", y[0])  ④

# print(mnist.DESCR)                           ⑤

```

- ① Vi laddar endast in 10000 observationer så kodexemplet går fortare att köra. Läsaren uppmanas att prova använda hela datasetet.
- ② Vi ser att vi har 10000 observationer och 784 oberoende variabler.
- ③ Vi visualisera en bild, vi ser att det ser ut som en femma.
- ④ Vi skriver ut bildens etikett/*label*. Det är en femma.
- ⑤ Läsaren uppmanas att exekvera koden för att få en beskrivning av datasetet. Vi har kommenterat ut koden i boken av utrymmesskäl.

```
(10000, 784)
(10000,)
True label for the plotted image is 5
```



I koden nedan delar vi upp vår data i träning, validering och test. Notera, vi har även `X_train_val` som alltså är träningsdatan och valideringsdatan ihopslagen, vi tränar om vår modell på det datasetet innan modellen utvärderas på testdatan senare. Vi skalar/standardiseringar även datan.

```
# Splitting data
X_train_val, X_test, y_train_val, y_test = train_test_split(X, y,
    ↵ test_size=2000, random_state=42)
X_train, X_val, y_train, y_val = train_test_split(X_train_val,
    ↵ y_train_val, test_size=2000, random_state=42)

# Scaling data
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
```

```
X_val = scaler.transform(X_val)
X_test = scaler.transform(X_test)
```

①

- ① På träningsdatan använder vi `.fit_transform()`, på valideringsdatan och testdatan använder vi endast `.transform()`.

I koden nedan tränar vi tre olika modeller, logistisk regression, *random forest* och *extra tree*. Därefter kombinerar vi även dem i en *voting classifier*.

```
logreg_clf = LogisticRegression(max_iter=1000)
random_forest_clf = RandomForestClassifier(n_estimators=100,
    ↵ random_state=42)
extra_trees_clf = ExtraTreesClassifier(n_estimators=100,
    ↵ random_state=42)

named_estimators = [
    ("logreg_clf", logreg_clf),
    ("random_forest_clf", random_forest_clf),
    ("extra_trees_clf", extra_trees_clf)
]
voting_clf = VotingClassifier(named_estimators, voting='hard')

models = [logreg_clf, random_forest_clf, extra_trees_clf,
    ↵ voting_clf]
for model in models:
    model.fit(X_train, y_train)

print("Accuracy for each model")
model_names = ["Logistic Regression", "Random Forest", "Extra
    ↵ Trees", "Voting Classifier"]
for name, model in zip(model_names, models):
    score = model.score(X_val, y_val)
    print(f"{name:20s}: {score:.4f}")
```

①

- ① Detta ger oss *accuracy* för varje modell. Läs dokumentationen för detaljer.

```
Accuracy for each model
Logistic Regression : 0.8800
Random Forest      : 0.9380
```

```
Extra Trees      : 0.9400
Voting Classifier : 0.9395
```

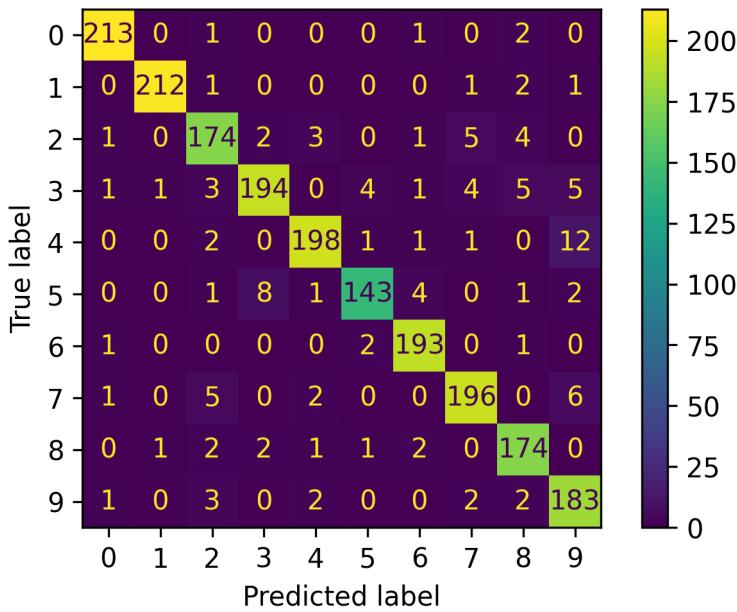
Den bäst presterande modellen, givet att vi använder *accuracy* som mått, ser vi från resultatet ovan är *extra tree*. Med koden nedan kan vi inspektera modellens resultat ytterligare. Här använder vi funktionen vi definierade i början av kodexemplet.

```
et_pred_val = extra_trees_clf.predict(X_val)

display_confusion_matrix(y_val, et_pred_val)          ①
print(classification_report(y_val, et_pred_val))      ②
```

- ① Vi använder funktionen vi skapade för att skapa en *confusion matrix*.  
② Vi skriver ut en *classification report*, läs *scikit-learn* dokumentationen för detaljer.

	precision	recall	f1-score	support
0	0.98	0.98	0.98	217
1	0.99	0.98	0.98	217
2	0.91	0.92	0.91	190
3	0.94	0.89	0.92	218
4	0.96	0.92	0.94	215
5	0.95	0.89	0.92	160
6	0.95	0.98	0.96	197
7	0.94	0.93	0.94	210
8	0.91	0.95	0.93	183
9	0.88	0.95	0.91	193
accuracy			0.94	2000
macro avg	0.94	0.94	0.94	2000
weighted avg	0.94	0.94	0.94	2000



Slutligen tränar vi om modellen på träningsdata och valideringsdata ihopslagen innan vi utvärderar den på testdata. Notera att vi även standardiseringar data med `StandardScaler`.

```
# Refit model on train + validation data
X_train_val = scaler.fit_transform(X_train_val)
extra_trees_clf.fit(X_train_val, y_train_val)

# Evaluate it on test data
et_pred_test = extra_trees_clf.predict(X_test)
# display_confusion_matrix(y_test, et_pred_test)           ①
# print(classification_report(y_test, et_pred_test))      ②
```

- ① Med denna kod skapar vi en *confusion matrix* för att analysera modellens prestanda på testdata. Vi gör dock inte det i boken (koden är utkommenterad) för att kodexemplet inte ska ta för många sidor.
- ② Med denna kod skapar vi en *classification report* för att analysera modellens prestanda på testdata. Vi gör dock inte det i boken (koden är utkommenterad) för

att kodexemplet inte ska ta för många sidor.

Om vi är nöjda med modellens prestanda på testdatan så hade vi i verkligheten tränat om den på hela datasetet innan produktionssättning. Det görs i koden nedan.

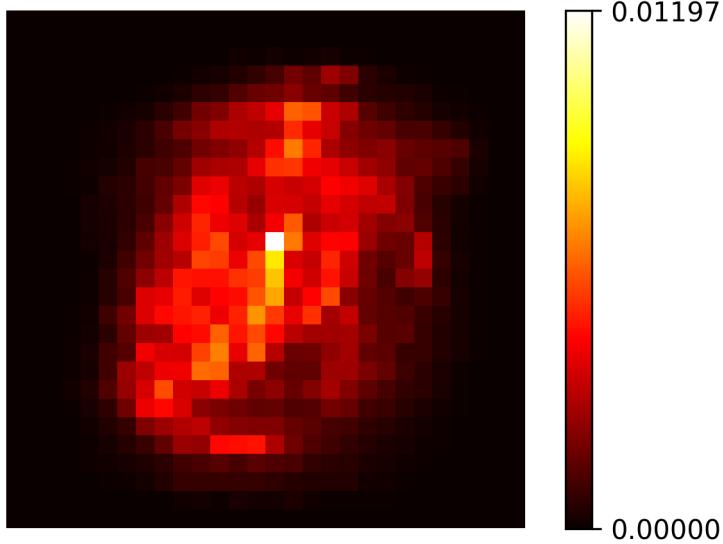
```
scaler_2 = StandardScaler()
X = scaler_2.fit_transform(X)
extra_trees_clf.fit(X, y)
```

Som ett avslutande avstick gör vi en visualisering som på ett fint sätt demonstrerar *feature importance* som vi pratade om i Avsnitt 4.3.4.

```
def plot_digit(data):
    image = data.reshape(28, 28)
    plt.imshow(image, cmap=mpl.cm.hot, interpolation="nearest")
    plt.axis("off")

plot_digit(extra_trees_clf.feature_importances_)

cbar = plt.colorbar(ticks=[extra_trees_clf.feature_importances_-
    .min(), extra_trees_clf.feature_importances_.max()])
```



Från visualiseringen ser vi att de pixlar som är i mitten är viktigast. Det är inte förvånande eftersom det är oftast där siffrorna skrivs, se den första visualiseringen i kodexemplet där en femma visualiseras.

Vi har nu kommit till slutet av kodexemplet där vi har sett hur ett ML-projekt från början till slut kan se ut.

#### 4.4.2 Kodexempel 2 - Välja önskad *precision* och *recall*

I detta kodexempel demonstreras hur vi själva kan välja vilken *precision* respektive *recall* vi önskar.

```

from sklearn.datasets import make_classification
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import precision_score, recall_score

X, y = make_classification(n_samples=1000, n_features=4,
                           n_classes=2, weights=[0.7, 0.3], random_state=42)           ①

```

```

model = LogisticRegression(random_state=42)
model.fit(X, y)

y_scores = model.predict_proba(X)[:, 1]                                ②

thresholds = np.linspace(0.0, 0.99, 100)
precisions = []
recalls = []

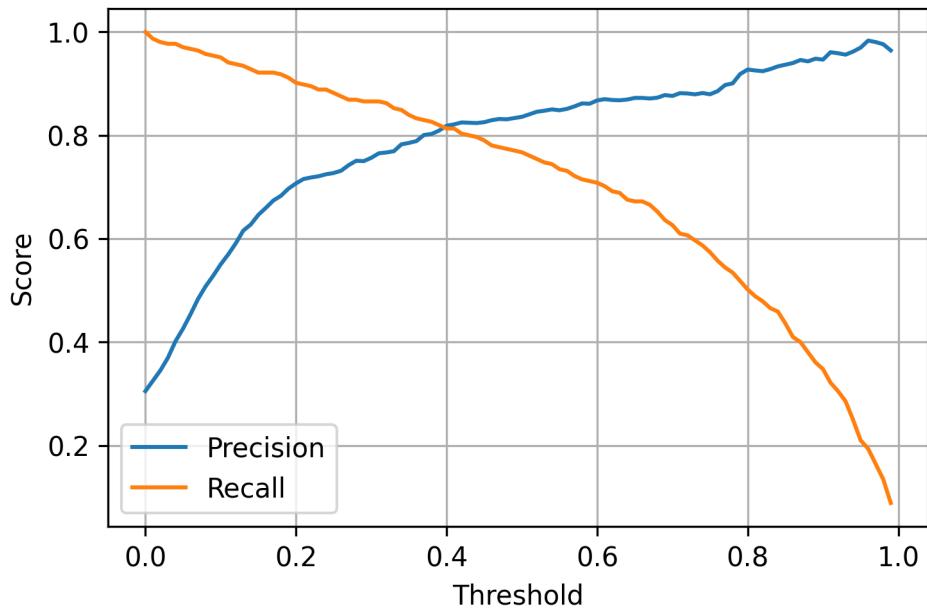
for t in thresholds:
    y_pred = (y_scores >= t).astype(int)
    precisions.append(precision_score(y, y_pred, zero_division=0))
    recalls.append(recall_score(y, y_pred, zero_division=0))           ③

plt.plot(thresholds, precisions, label='Precision')
plt.plot(thresholds, recalls, label='Recall')
plt.xlabel('Threshold')
plt.ylabel('Score')
plt.title('Values for precision and recall with different
          ↴ thresholds')
plt.legend()
plt.grid(True)

```

- ① Vi genererar ett syntetiskt dataset för demonstrationssyfte.
- ② Vi predikterar sannolikheter för den positiva klassen.
- ③ Med koden från den andra annoteringen till hit beräknar vi *precision* och *recall* för olika *thresholds*. Detta använder vi i koden som följer för att skapa en visualisering där vi kan se vilken *precision* respektive *recall* vi får för olika *thresholds*.

Values for precision and recall with different thresholds



Från grafen vi skapade kanske vi tycker det ser bra ut med den *precision* och *recall* vi får när threshold är 0.6. Koden nedan visar vad *precision* och *recall* är vid vårt valda *threshold*. Notera, vi såg det via grafen men nu får vi alltså exakta siffror.

```
threshold = 0.6
y_pred_custom = (y_scores >= threshold).astype(int)

precision = precision_score(y, y_pred_custom)
recall = recall_score(y, y_pred_custom)

print(f"Precision: {precision:.3f}")
print(f"Recall: {recall:.3f}")
```

Precision: 0.867

Recall: 0.708

Med koden nedan visar vi hur vi hade predikterat en ny observation med vår valda

*threshold.*

```
new_observation = np.array([[0.5, -1.2, 0.3, 0.5]])          ①

new_score = model.predict_proba(new_observation)[:, 1][0]
threshold = 0.6
new_pred = int(new_score >= threshold)

print(f"Probability for class 1: {new_score:.3f}")
print(f"Prediction (with threshold={threshold}): {new_pred}")
```

- ① När vi genererade det syntetiska datasetet specificerade vi `n_features=4`, datan har alltså fyra oberoende variabler vilket vår nya observation också behöver ha.

Probability for class 1: 0.461  
Prediction (with threshold=0.6): 0

Sammanfattningsvis har vi i kodexemplet demonstrerat hur vi kan välja *precision* respektive *recall* genom att justera *threshold*.

## 4.5 Övningsuppgifter

Övningsuppgifter för kapitlet finns på bokens hemsida:  
[https://github.com/AntonioPrgomet/ai\\_tillaempad\\_ml](https://github.com/AntonioPrgomet/ai_tillaempad_ml)

# Kapitel 5

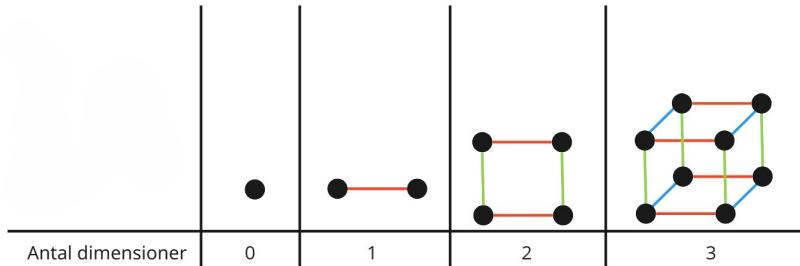
## Dimensionsreducering

I detta kapitel kommer vi börja med att kolla på några utmaningar som kan uppstå för högdimensionella dataset. Detta brukar benämnas *the curse of dimensionality*. Därefter presenteras dimensionsreducering, en metod inom icke-väglett lärande, där vi förenklar indata genom att transformera den till en rymd med lägre dimensionalitet. Som vi kommer att se innebär det rent praktiskt att antalet variabler i datan minskar. Detta görs främst för att möjliggöra snabbare modellträning, ofta på bekostnad av en sämre prediktionsförmåga för de modeller som senare tränas på datan då information går förlorad. Avslutningsvis presenteras principalkomponentanalys som är den vanligast förekommande modellen för att genomföra dimensionsreducering.

### 5.1 *Curse of dimensionality*

Ju fler variabler/*features* vi har i ett dataset, desto högre dimension sägs datasetet ha. Generellt sett kan vi visualisera något i upp till tre dimensioner, för fyra eller fler dimensioner kan vi inte göra en direkt visualisering, se Figur 5.1. Det kan noteras att vi lever i en tredimensionell värld, där vi uppfattar längd, bredd och djup.

Med en given mängd data, det vill säga ett givet antal rader, så kan regressions- och klassificeringsmodeller i snitt prestera bättre genom att någon variabel läggs till. Om vi tänker oss att vi ska skapa en regressionsmodell som kan prediktera en persons lön och vi har 200 observationer, då är det en rimlig gissning att modellen kommer prestera bättre om vi använder variablerna ålder och utbildningsnivå än enbart ålder för att



Figur 5.1: Illustration av hur olika dimensioner, från 0D till 3D, kan se ut. För fyra eller högre dimensioner kan vi inte göra en direkt visualisering.

prediktera lönen. Men denna effekt vänder och börjar vi lägga till ännu fler variabler så brukar modellerna prestera sämre. Exakt hur många variabler som krävs för att denna användning ska ske beror såklart på situationen. En tumregel är att ju högre dimension ett dataset har, desto mer data krävs för att kunna skapa bra modeller. Intuitivt är det rimligt eftersom fler variabler kräver att modellen måste lära sig mer komplexa mönster vilket alltså kräver mer data att träna på.

I högre dimensioner kan data bete sig på ett sätt som vi kanske inte förväntar oss eller är vana vid från lägre dimensioner. Detta benämns *the curse of dimensionality*. Vi kollar nu på två exempel.

Vi börjar med det första exemplet. Antag att vi slumpmässigt väljer en punkt i en tvådimensionell enhetskvadrat ( $1 \times 1$  kvadrat). Då går det matematiskt att visa att det är mindre än 0.4% chans att punkten är närmre än 0.001 längdenheter från en kant. Det är alltså väldigt osannolikt att en punkt är extrem i någon av dimensionerna. Om motsvarande resonemang görs för en 100-dimensionell enhetshyperkub så är sannolikheten strax över 18%, för 1000 dimensioner är den över 86% och för 10000 dimensioner är sannolikheten större än 99.999999%. Exemplet demonstrerar att i högre dimensioner ökar sannolikheten att datan blir mer extrem.

### **i De flesta människor är extrema i något avseende**

Som en rolig notis kan vi tänka oss att en mänsk kan beskrivas i flera dimensioner. Exempelvis hur många steg den tar per dag kan vara en dimension och en annan dimension kan vara hur mycket mjölk den dricker per dag. På så sätt hade vi kunnat beskriva en mänsk med exempelvis 1000 dimensioner och då är sannolikheten väldigt hög att den är extrem i något avseende.

Vi fortsätter med det andra exemplet. Antag att vi slumpmässigt väljer två punkter i en 2-dimensionell enhetskvadrat, då går det att matematiskt beräkna att det genomsnittliga avståndet mellan punkterna blir ungefär 0.52 längdenheter. Motsvarande resonemang i en 3-dimensionell enhetshyperkub gör att det genomsnittliga avståndet mellan punkterna ungefär blir 0.66 längdenheter. I en 1000000-dimensionell enhetshyperkub blir avståndet ungefär 408 längdenheter. Intuitivt kan vi förstå det genom att i fler dimensioner finns det mer plats vilket gör att avståndet mellan datapunkter ökar. Detta kan vi få en känsla för om vi kollar på Figur 5.1 i fallet när vi går från två till tre dimensioner. Exemplet implicerar att om vi har en ny datapunkt som vi ska prediktera så är den i snitt längre ifrån de datapunkter som en modell har tränats på ju högre dimension datasetet har. Det blir alltså svårare att prediktera och risken för överanpassning ökar, det vill säga att modellen presterar bra på den datan den tränats på men generaliseras dåligt på ny, osedd data.

De två exemplen ovan belyser att data generellt sett blir mer svårmodellerad i högre dimensioner. I teorin är en lösning på detta problem att samla in mer data för att på så sätt säkerställa att datapunkterna inte ligger för långt ifrån varandra. I praktiken kan det dock ofta vara svårt eftersom antalet observationer som behövs för att uppnå en viss täthet och pålitliga resultat växer exponentiellt med antalet dimensioner eftersom utrymmet ökar exponentiellt. Vi exemplifierar. Vi kan föreställa oss en linje, i en dimension, som är graderad från 1 till 10. Det är 10 punkter på den linjen. Om vi utökar denna linje till en kvadrat, i två dimensioner, har vi nu  $10 \times 10 = 100$  punkter. Redan vid 80 dimensioner har vi  $10^{80}$  punkter, vilket är ungefär lika mycket som det uppskattade antalet atomer i det observerbara universum. För att sätta det i perspektiv så kan vi notera att MNIST, den datan vi arbetade med i Avsnitt 4.4.1 hade 784 variabler.

Vi vet alltså nu att högdimensionella dataset kan vara fundamentalt mer svårmodellerade innehållande att vi behöver beakta *curse of dimensionality*. Ett sätt att hantera detta på är att genomföra dimensionsreducering vilket är ämnet för avsnittet som följer.

## 5.2 Vad är dimensionsreducering?

Dimensionsreducering innebär att vi transformerar ett dataset med en given dimension till ett lägre antal dimensioner. Följden blir alltså att datasetet har ett mindre antal variabler. Från en given uppsättning variabler får vi alltså nya variabler som är färre till antalet. Generellt sett kan dessa nya variablerna inte tolkas som de ursprungliga variablerna kunde vilket innebär att information går förlorad, det är ett pris vi får betala för att kunna genomföra dimensionsreducering.

Som vi kommer att se i avsnittet om *PCA* nedan, Avsnitt 5.3, är det endast de oberoende variablerna som hanteras för att transformera data. En beroende variabel är inte nödvändig för att göra detta och därfor är dimensionsreducering en metod inom icke-väglett lärande. Notera att när dimensionsreducering används som ett försteg till en modell från väglett lärande, såsom logistisk regression eller beslutsträd, i syfte att reducera antalet dimensioner på data så tillhör processen att reducera antalet dimensioner fortfarande icke-väglett lärande. I Avsnitt 5.3.1 kommer vi se ett exempel på hur vi kan använda *PCA* som ett försteg till en logistisk regressionsmodell.

### 5.2.1 Effekter av dimensionsreducering

Genomför vi en dimensionsreducering görs det ofta i syfte att uppnå en eller flera av effekterna nedan.

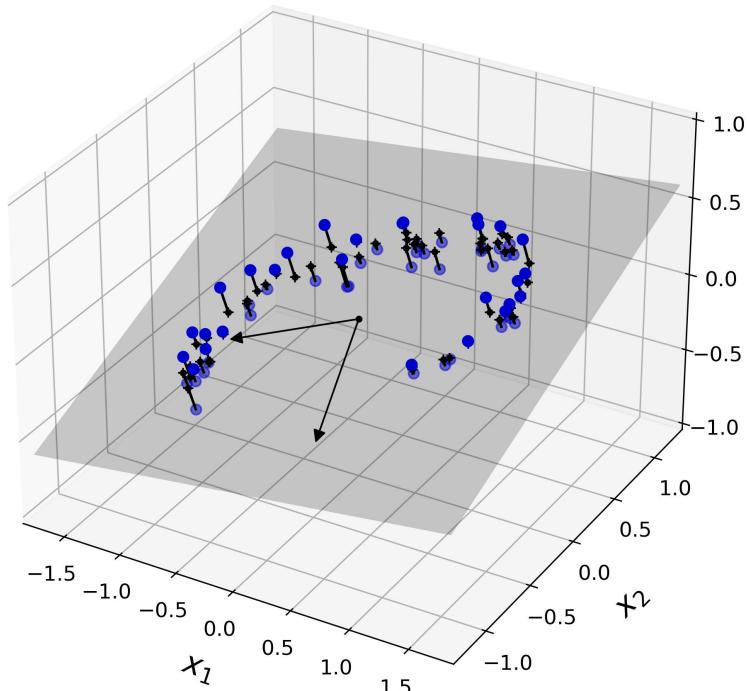
- Minska tiden det tar att träna en modell.
- En modell som tränas på dimensionsreducerad data kan ibland få bättre prediktionsförmåga även om den ofta blir sämre.
- Att kunna visualisera data även om det inte är originaldata.

Generellt sett genomförs dimensionsreducering ofta för att påskynda modellträningen. Exempelvis kanske en modell tränas om varje natt och då behöver modellen helt enkelt kunna tränas på utsatt tid.

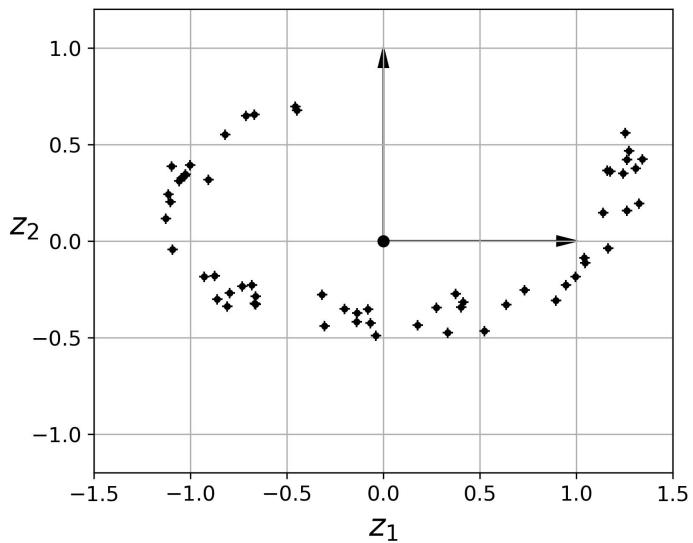
I de fall en modells prediktionsförmåga försämras efter en dimensionsreducering så behöver en avvägning mellan tid och prediktionsförmåga göras. Exempelvis, om en klassificeringsmodell tar 15 timmar att träna och uppnår en 70% *accuracy* kanske det efter dimensionsreducering tar 8 timmar att träna modellen där en 65% *accuracy* uppnås. Huruvida detta är värt det eller inte beror på från fall till fall innehållande att en avvägning behöver göras. Det kan även hända att modeller får bättre prediktionsförmåga när datan som används för att träna modellerna har genomgått en dimensionsreducering. En anledning kan vara att onödigt brus från datan tas bort och att modellen därfor inte blir överanpassad, vilket exempelvis kan ske för att den anpassat sig till slumpmässigt

brus snarare än till underligande mönster i data.

Om ett högdimensionellt dataset reduceras till två eller tre dimensioner kan det visualiseras. Observera dock att det är en transformerad version av data som visualiseras och att göra en direkt tolkning kan vara utmanande. Kollar vi på Figur 5.2 och Figur 5.3 ser vi hur ett tredimensionellt dataset dimensionsreducerats till två dimensioner och därefter visualiseras. I högre dimensioner, exempelvis 10-dimensioner så blir denna typ av tolkningar svår att göra och görs sällan.



Figur 5.2: Ett dataset i 3D som dimensionreduceras ned på ett plan som är tvådimensionellt. I Figur 5.3 ser vi hur datan ser ut efter att en dimensionsreducering genomförs.



Figur 5.3: Ett dataset i 2D efter att dimensionsreducering genomförts. Se Figur 5.2 för hur ursprungsdatan såg ut.

## 5.3 Principalkomponentanalys (*PCA*)

Principalkomponentanalys, ofta förkortat *PCA* efter engelskans *principal component analysis*, är den vanligast förekommande modellen för dimensionsreducering. Vi går nu igenom den.

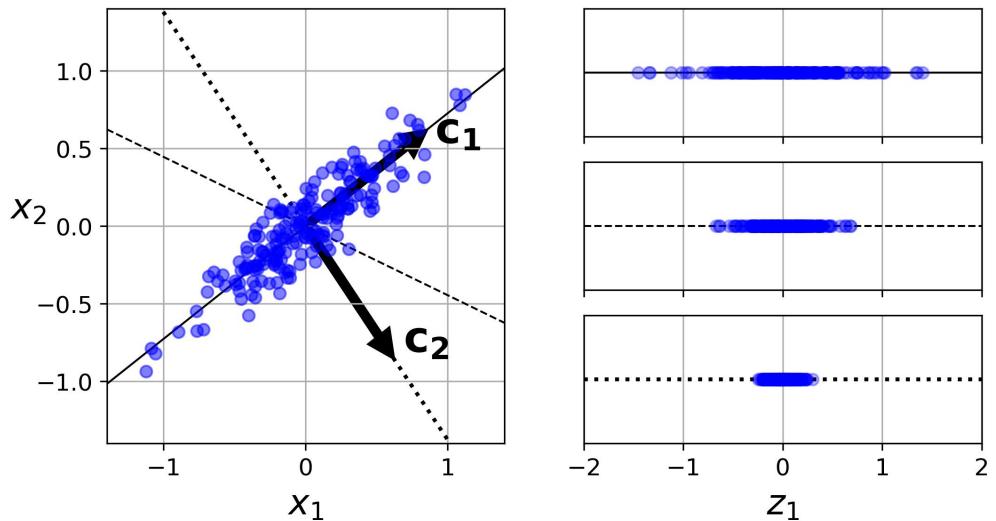
För att genomföra en dimensionsreducering så kan punkter projiceras ned till ett lägre dimensionellt hyperplan. Hur väljer vi hyperplan? Ett rimligt sätt att göra detta på är att välja det hyperplan som bevarar så mycket information från det ursprungliga datasetet som möjligt. För att göra detta så väljer *PCA* det hyperplan som behåller högst andel varians. Vi exemplifierar.

### i Vad är ett hyperplan?

Ett hyperplan är en generalisering av ett tvådimensionellt plan som befinner sig i ett tredimensionellt rum till ett rum av godtycklig dimension. I 2D har vi en linje, i 3D ett plan och i högre dimensioner har vi ett hyperplan. Rent generellt gäller det att i ett n-dimensionellt rum är ett hyperplan en platt hyperytta med n-1 dimensioner.

Antag att vi har ett dataset med två dimensioner, se vänster bild i Figur 5.4. De två dimensionerna är  $x_1$  och  $x_2$ . I figuren finns också tre olika linjer (heldragen, punktmarkerad och sträckad) vilket är motsvarigheten till ett plan i 2D. Höger bild i Figur 5.4 visar hur punkterna har projicerats på respektive linje. Vi ser att det är den heldragna linjen som behåller mest varians åtföljt av den streckade och slutligen den prickade. Eftersom vi vill behålla så mycket varians som möjligt hade datan alltså projicerats ned på den heldragna linjen. Vi noterar att i den ursprungliga datan kanske variablerna  $x_1$  och  $x_2$  betecknar vikt och längd, den nya variabeln  $z_1$  som fås efter projiceringen har ingen direkt tolkning kopplat till de ursprungliga variablerna. Tolkningen försvinner alltså.

Rent generellt, för att hitta det hyperplan som behåller störst andel varians identifierar *PCA*-modellen först den linje/axel som behåller störst andel av variansen i den ursprungliga datan. Denna nya axel behöver inte vara i linje med någon av de ursprungliga axlarna utan går i den riktning där punkterna sprider ut sig som mest, alltså där variansen är störst. Detta är  $C_1$  i Figur 5.4. Modellen fortsätter sedan med att identifiera den axel som behåller den näst största andelen av variansen. Den andra axeln måste vara ortogonal (vinkelrät) mot den första axeln. Detta är  $C_2$  i Figur 5.4. Notera att  $C_2$  är den punktmarkerade linjen trots att den sträckade linjen behåller mer varians. Anledningen är att den sträckade linjen inte är ortogonal mot  $C_1$  vilket den alltså måste



Figur 5.4: Projektion av dataset i 2D på linjer i 1D.

vara. Modellen fortsätter sedan att identifiera ytterligare axlar tills antalet nya axlar är lika många som antalet dimensioner i ursprungliga data. Varje ny axel måste vara ortogonal mot tidigare axlar. Varje axel kallas för principalkomponent.

När samtliga principalkomponenter identifierats är nästa steg att bestämma hur många som ska behållas för att genomföra projektionen på. Hur gör vi det valet? Vanligtvis väljer vi den andel av variansen som vi vill behålla, exempelvis 95%. Antag att vi har ett dataset som består av tio ursprungliga variabler och vi genomför en *PCA*. Vi kommer då att skapa tio stycken principalkomponenter (PK) där vi antar att de olika principalkomponenterna förklrar följande andel av variansen i datan.

- PK1: 40%
- PK2: 30%
- PK3: 20%
- PK4: 5%
- PK5-PK10: 5% (tillsammans)

Om vi exempelvis önskar behålla minst 95% av variansen så kommer det medföra att vi behåller PK1-PK4 ( $40 + 30 + 20 + 5 = 95$ ). Vi har alltså gått från att ha tio variabler till att ha fyra i vårt exempel.

Koden nedan demonstrerar hur *PCA* kan tillämpas.

```
import numpy as np  
from sklearn.decomposition import PCA  
  
np.random.seed(42)  
X = np.random.rand(1000, 10)  
pca = PCA(n_components=0.7)  
X_reduced = pca.fit_transform(X)  
print(f"vi har gått från {X.shape[1]} dimensioner till  
    ↪ {X_reduced.shape[1]} dimensioner genom att genomföra en  
    ↪ dimensionsreducering.")
```

- ① Importera *PCA* från *scikit-learn* biblioteket.
- ② Skapa ett dataset *X* som innehåller 10 kolumner.
- ③ Instantiera en *PCA*-modell som automatiskt väljer antalet komponenter så att minst 70% av den totala variansen i orginaldatan bevaras.
- ④ Träna *PCA*-modellen på datasetet *X* och transformera dataen.

vi har gått från 10 dimensioner till 7 dimensioner genom att genomföra en dimensionsreducering.

Koden nedan visar hur stor andel av variansen som förklaras med respektive principal-komponent.

```
variance_per_PK = pca.explained_variance_ratio_  
print(variance_per_PK)
```

```
[0.11671953 0.11109465 0.1082102  0.10253589 0.10232202 0.0994705  
 0.09834437]
```

I kodexemplet ovan valde vi att behålla 70% av variansen. I verkligheten kan det vara användbart att göra en visualisering för att se om den kumulativa förklarade variansen snabbt avtar efter ett visst antal dimensioner. Detta kan ge en vägledning kring när det inte längre är värt att använda fler dimensioner i dimensionsreduceringen. Se kodexemplet som följer där detta demonstreras.

Vi börjar med att importera bibliotek och ladda in *MNIST*-datan som vi kommer använda för demonstrationssyfte.

```

from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split

mnist = fetch_openml('mnist_784', version=1, as_frame=False)
mnist.target = mnist.target.astype(np.uint8)

X = mnist["data"]
y = mnist["target"]

X_train, X_test, y_train, y_test = train_test_split(X, y,
    test_size = 0.2)

```

Därefter genomför vi en *PCA* och skapar en visualisering. Från visualiseringen ser vi att det bildas en armbåge-likt form vid cirka 70 dimensioner. De första 70 dimensionerna står för cirka 85% av den totala variansen. De resterande 714 dimensionerna står för cirka 15 % av variansen. Kom ihåg att *MNIST* hade  $28 \times 28 = 784$  stycken dimensioner, så vi gjorde beräkningen  $784 - 70 = 714$ .

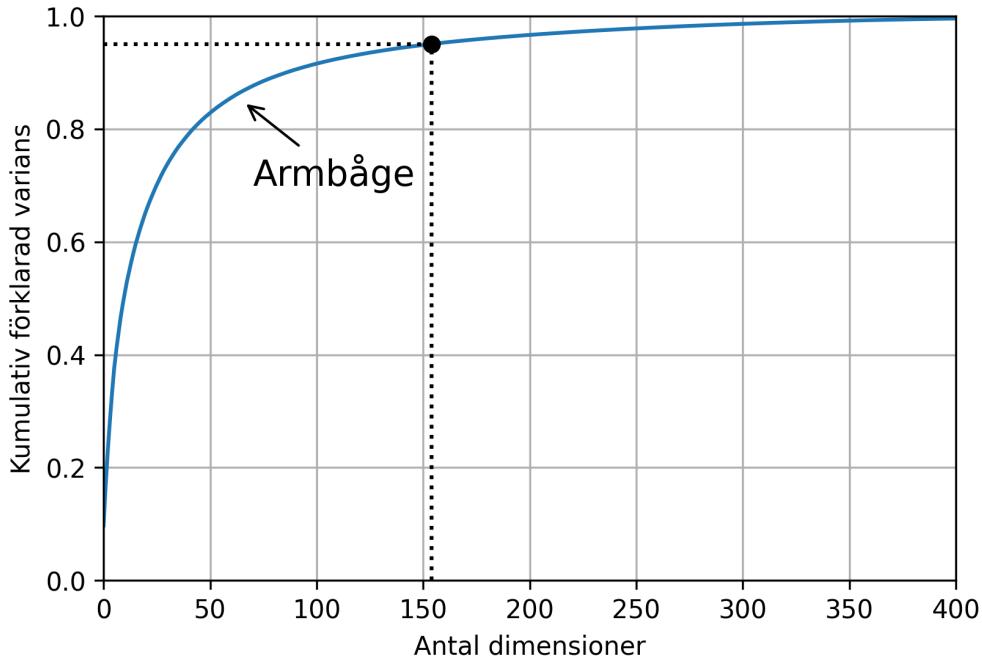
```

pca = PCA()
pca.fit(X_train)
cumsum = np.cumsum(pca.explained_variance_ratio_)
d = np.argmax(cumsum >= 0.95) + 1
print(d)                                     ①

plt.figure(figsize=(6,4))
plt.plot(cumsum)
plt.axis([0, 400, 0, 1])
plt.xlabel("Antal dimensioner")
plt.ylabel("Kumulativ förklarad varians")
plt.plot([d, d], [0, 0.95], "k:")
plt.plot([0, d], [0.95, 0.95], "k:")
plt.plot(d, 0.95, "ko")
plt.annotate("Armbåge", xy=(65, 0.85), xytext=(70, 0.7),
             arrowprops=dict(arrowstyle="->"), fontsize=14)
plt.grid(True)

```

① Vi skriver ut antalet dimensioner som krävs för att behålla 95% av variansen.



I detta fallet vore det därför rimligt att ha cirka 70-150 dimensioner då fler dimensioner inte tillför så mycket till den förklarade variansen. Exakt antal dimensioner som väljs beror dock på och är något vi själva väljer baserat på kontext.

Det är också möjligt att välja antal dimensioner manuellt istället för den andel av variansen som vi vill behålla. Detta är särskilt användbart när syftet med dimensionsreduceringen är att kunna visualisera datan, då vill vi välja två eller tre dimensioner. I koden nedan demonstreras hur vi manuellt väljer ett fixt antal dimensioner. Vi demonstrerar även hur vi kan transformera tillbaka data till ursprunglig dimension genom att använda metoden `.inverse_transform()`. Resultatet blir dock inte detsamma på grund av att information går förlorad när vi genomför en *PCA*.

```
# Create a dataset with 3 features/columns
np.random.seed(42)
X = np.random.rand(1000, 3)
print(X[0:2, :])
```

```

# Reducing the data to 2 dimensions
pca = PCA(n_components=2) (1)
X2D = pca.fit_transform(X)
print(X2D[0:2, :]) (2)

# Using the inverse transform to transform the data back to 3
  ↔  dimensions
X3D_inv = pca.inverse_transform(X2D) (3)
print(X3D_inv[0:2, :])

# The data is not the same since information is lost when
  ↔  performing PCA
print(np.allclose(X3D_inv, X))

```

- ① Vi har specificerat hyperparametern `n_components=2` vilket ger oss två principal-komponenter.
- ② Koden ovan, `print(X[0:2, :])`, skrev ut två rader från ursprungsdatan där vi ser att datan har tre kolumner. `print(X2D[0:2, :])` visar att vi nu har två kolumner efter att vi genomfört *PCA*.
- ③ Vi transformerar tillbaka datan till tre dimensioner. Vi ser dock, när vi skriver ut datan med koden `print(X3D_inv[0:2, :])` i raden nedan, att den inte är densamma som ursprungsdatan. Detta eftersom genomförandet av *PCA* medför att information går förlorad vilket gör att vi inte kan återskapa orginaldatan.

```

[[0.37454012 0.95071431 0.73199394]
 [0.59865848 0.15601864 0.15599452]]
 [[ 0.45459779  0.05941797]
 [-0.36808474 -0.19497059]]
 [[0.62343309 0.91834955 0.62721367]
 [0.37026666 0.18571753 0.25214411]]

```

`False`

När dimensionsreduceringen är genomförd har vi ett nytt dataset. Det är nu möjligt att använda det nya datasetet för att göra visualiseringar eller träna en modell på.

Vi noterar att *PCA* är en linjär metod där modellen antar att den underliggande strukturen/mönstret i datan är linjär. För att hantera icke-linjära strukturer kan vi använda oss av *kernel PCA* (kPCA). Det går vi igenom härnäst.

### 5.3.1 Kernel PCA

I Avsnitt 4.3.2 introducerade vi *kernel trick* vilket möjliggjorde oss att genomföra icke-linjär klassificering. Logiken var att linjär klassificering för den transformerade datan (strikt sett transformeras inte datan men effekten är densamma) motsvarade icke-linjär klassificering i den ursprungliga datan. Samma trick kan användas för *PCA* vilket då ger oss det som benämns *kernel PCA*. Med *kPCA* kan vi alltså genomföra icke-linjär projicering för att dimensionsreducera data.

I Figur 5.5 visualiseras ett dataset som benämns för *swiss roll*. Det laddas in via `from sklearn.datasets import make_swiss_roll`. Det datasetet har vi genomfört icke-linjära dimensionsreduceringar på genom att använda tre olika *kernels*; linjär (som motsvarar ”vanlig” *PCA*), *RBF* och *sigmoid*. Se resultatet i Figur 5.6.

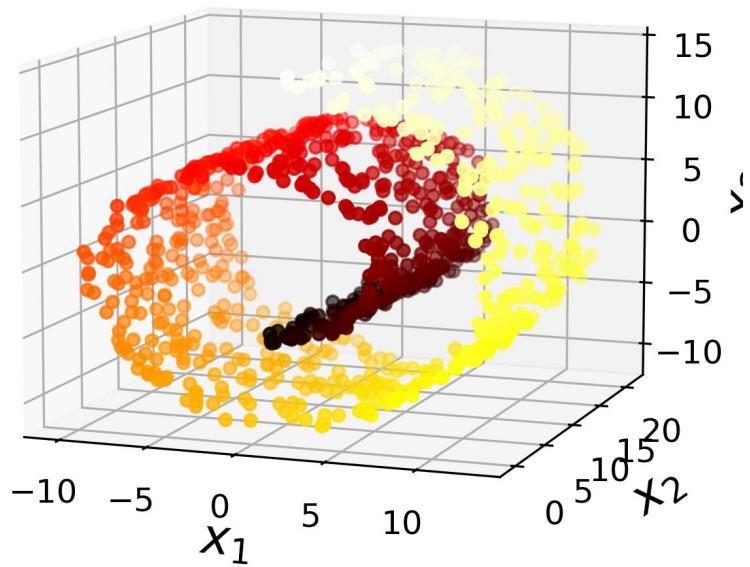
Koden nedan demonstrerar användandet av *kPCA*.

```
from sklearn.decomposition import KernelPCA          ①  
  
X = np.random.rand(10000, 10)  
rbf_pca = KernelPCA(n_components=2, kernel="rbf", gamma=0.04)    ②  
X_reduced = rbf_pca.fit_transform(X)
```

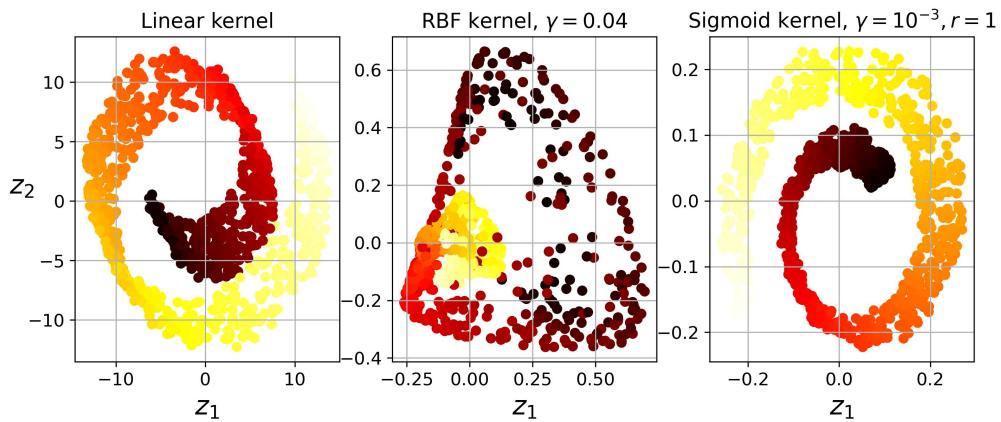
- ① Importera *kPCA* från *scikit-learn*.
- ② Instantiera en *kPCA*-modell som manuellt väljer antalet komponenter till 2, vi använder *RBF* som *kernel* och hyperparametern *gamma* sätts till 0.04.

*kPCA* är en modell inom icke-väglett lärande. Hur ska vi utvärdera modellen för att kunna välja *kernels* och hyperparametrar på ett optimalt sätt? Eftersom dimensionsreducering ofta utförs som ett förberedande steg för att använda en modell inom väglett lärande (regression eller klassificering) kan vi skapa en *pipeline* som innehåller en *kPCA* och en prediktor. Detta gör det möjligt att utnyttja *grid search* för att välja *kernel* och hyperparametrar som ger den bästa prestandan för vår prediktor. Se kodexemplet nedan där en logistisk regressionsmodell används som prediktor.

```
from sklearn.model_selection import GridSearchCV  
from sklearn.linear_model import LogisticRegression  
from sklearn.pipeline import Pipeline  
from sklearn.datasets import make_moons
```



Figur 5.5: Swiss roll dataset.



Figur 5.6: Visualisering av hur datan från Figur 5.5 ser ut efter att tre olika typer av dimensionsreduceringar genomförs.

```
X, y = make_moons(n_samples=1000, noise=0.1, random_state=42)

clf = Pipeline([
    ("kpca", KernelPCA(n_components=3)),
    ("log_reg", LogisticRegression())
])①

hyperparam_grid = [
    {"kpca__gamma": np.linspace(0.01, 1, 10),
     "kpca__kernel": ["rbf", "sigmoid"]}]②

grid_search = GridSearchCV(clf, hyperparam_grid, cv=5)
grid_search.fit(X, y)

print(f"The best parameters: {grid_search.best_params_}")③
```

- ① Skapa en *pipeline* som innehåller en *kPCA* och en logistisk regressionsmodell.
- ② Specificera vilka hyperparametrar som ska undersökas.
- ③ Skriv ut de hyperparametrar som ger den bästa prestandan.

```
The best parameters: {'kpca__gamma': np.float64(1.0), 'kpca__kernel':  
'sigmoid'}
```

## 5.4 Övningsuppgifter

Övningsuppgifter för kapitlet finns på bokens hemsida:

[https://github.com/AntonioPrgomet/ai\\_tillaempad\\_ml](https://github.com/AntonioPrgomet/ai_tillaempad_ml)

# Kapitel 6

## Klustering

I detta kapitel kommer vi att lära oss om klustering, ett område inom icke-väglett lärande. Kapitlet inleds med en genomgång av vad klustering innebär samt olika tillämpningsområden. Därefter behandlas *K-means*, den vanligast förekommande modellen för att utföra klustering. Kapitlet avslutas med två kodexempel.

### 6.1 Vad är klustering?

Klustering innebär att datapunkter delas in i olika grupper/kluster där principen är att datapunkter som liknar varandra ska hamna i samma kluster. Den vanligast förekommande klustringsmodellen heter *K-means*, den går igenom i Avsnitt 6.2.

För att exemplifiera klustering kan vi tänka oss att vi har ett album innehållande självporträtt av olika personer. En klustringsmodell hade kunnat identifiera alla bilder med en viss person och gruppera dessa bilder i ett kluster och bilder som innehåller en annan person hade hamnat i ett annat kluster. Detta är ett problem inom icke-väglett lärande eftersom det inte finns någon beroende variabel som anger vilken person som är i vilken bild. Hade varje bild inom albumet haft en etikett där det framgår vilken person som är på vilken bild hade det varit väglett-lärande (klassificeringsproblem). De två exemplen kan tyckas vara lika men i det första fallet har träningsdata (albumet) inga etiketter där det framgår vilken person som är i vilken bild medan det i det andra fallet finns etiketter på träningsdata där det framgår vilken person som är i vilken bild.

### **i Majoriteten av data saknar etiketter i verkligheten**

Generellt sett är majoriteten av tillämpningar inom ML kopplade till väglett lärande. En utmaning i verkligheten är att majoriteten av data däremot, saknar etiketter, det vill säga datan har  $X$  men saknar  $y$ .

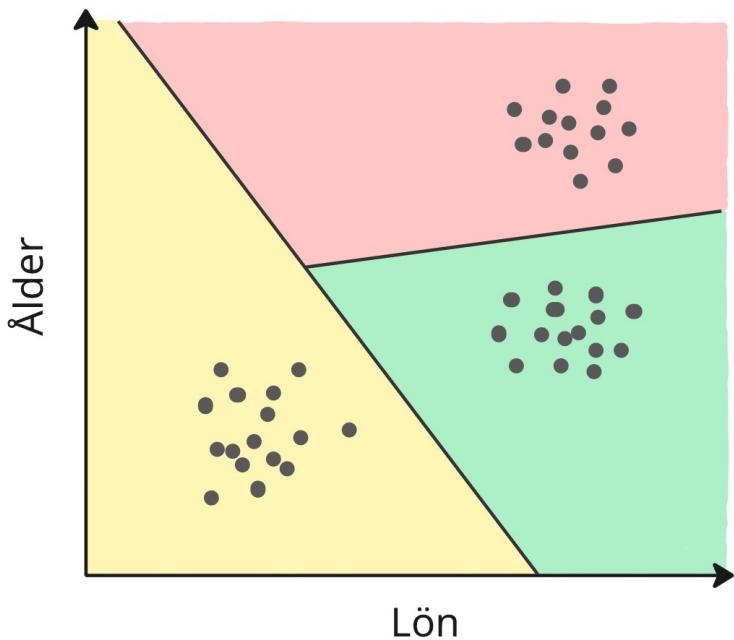
Om vi exempelvis i verkligheten hade velat bygga en ML-modell som kan detektera produktfel på ett löpande band så hade vi ganska enkelt kunnat sätta upp en kamera som tar bilder och sparar dessa. På kort tid hade vi kunnat bygga ett stort dataset bestående av tusentals bilder. Om vi därför hade velat bygga en binär klassificeringsmodell som kan detektera vilka produkter som har fel respektive inte har det hade vi inte kunnat göra det eftersom etiketter på datan saknas. Att manuellt sätta etiketter kräver ofta mänskligt arbete och kan därför ta lång tid och vara dyrt. En potentiell lösning för att hantera detta kan vara att använda sig av det som kallas för *semi-vägledd inlärning*, vi kommer se ett kodexempel kopplat till detta i Avsnitt 6.3.2.

#### **6.1.1 Exempel på tillämpningsområden för klustering**

Klustering kan användas för flera olika tillämpningsområden. Tre exempel är för att bygga kundsegmenteringsmodeller, genomföra anomalidetektion och för semi-vägledd inlärning. Vi förklarar nu varje exempel närmare.

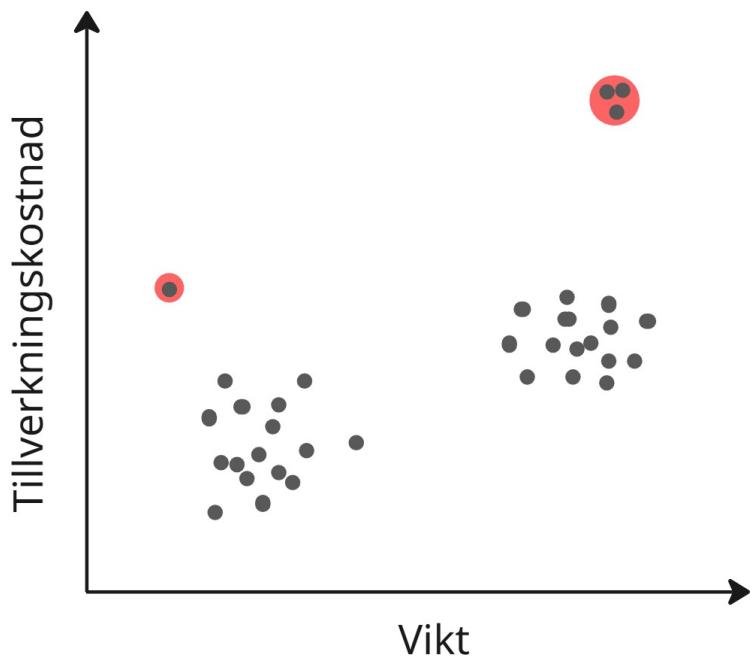
Med kundsegmentering kan kunder som liknar varandra (exempelvis med avseende på variabler såsom köpbeteende, ålder och lönenivå) grupperas. Dessa grupper kan därför undersökas för att exempelvis lära sig förstå vad för typ av produkter de gillar och vilka kommunikationskanaler de föredrar. Företag kan därför anpassa exempelvis produkterbjudanden och kommunikationssätt för kunderna baserat på vilket kluster de tillhör. Vi hade kunnat tänka oss att ett kluster innehåller mestadels pensionärer och ett annat kluster innehåller mestadels studenter. Då kanske klustret med pensionärer föredrar att få fysisk reklam med marknadsföring medan studenterna kanske föredrar reklam via sociala medier. I Figur 6.1 ser vi ett exempel där kunder grupperats med avseende på variablerna lön och ålder för att bilda tre kluster. I Avsnitt 6.3.1 kommer vi kolla på ett kodexempel som demonstrerar hur kundsegmentering kan gå till.

Klustering används även för att upptäcka anomalier eller avvikelse i data. Det finns två typer av anomalier. Antingen kan en eller flera observationer inom ett kluster avvika från övriga observationer i samma kluster och därmed betraktas som anomalier eller så kan ett helt kluster avvika från övriga kluster och därmed betraktas som en anomali.



Figur 6.1: Exempel på hur en kundsegmentering med avseende på variablerna lön och ålder kan se ut. Troligtvis vill ett företag hantera de olika kundgrupperna på olika sätt för att exempelvis öka försäljningen.

Det senare fallet kan till exempel ske om ett kluster innehåller få observationer jämfört med övriga kluster eller om ett kluster har ett stort avstånd till övriga kluster, se Figur 6.2 som illustrerar detta. Anomalidetektion används exempelvis i företag för att upptäcka defekta produkter i en tillverkningsprocess. Det används även av exempelvis banker för att upptäcka avvikande transaktioner som därefter manuellt kan granskas för att upptäcka eventuella bedrägerier och brott.



*Figur 6.2: Illustration av kluster för produkter i en tillverkningsprocess med anomalier. Den vänstra röda ringen demonstrerar en observation som anses vara en anomali eftersom den avviker från övriga observationer i samma kluster. Den högra röda ringen demonstrerar ett kluster som anses vara en anomali eftersom klustret avviker från övriga kluster i den bemärkelsen att det har få observationer.*

Om vi har ett dataset där endast en delmängd av datapunkterna har etiketter så kan vi genomföra en klustring och därefter sätta etiketter baserat på vilket kluster en observation tillhör. Rent generellt, när endast en delmängd av vår träningsdata har etiketter och vissa observationer alltså saknar etiketter så benämns det semi-väglett lärande.

Vanligtvis är det så att antalet observationer som har etiketter i förhållande till antal observationer som saknar etiketter är mycket litet. Vi kommer kolla närmare på detta i Avsnitt 6.3.2.

## 6.2 *K-Means*

*K-Means* är den vanligast förekommande modellen för klustering. Modellen är relativt enkel där ny data grupperas genom att varje datapunkt tilldelas det kluster vars centroid är närmast. För att genomföra de tillämpningsområden som beskrevs i Avsnitt 6.1.1 hade vi kunnat använda *K-Means* modellen.

Generellt sett är *K-means*-modellen snabb och skalbar innehärande att den kan hantera stora dataset. Modellen presterar generellt sett bra när storleken på de olika klustren är ungefär lika stora, klustren har liknande spridning och klustren har en sfärisk form. När detta inte är uppfyllt finns det en risk att modellen presterar dåligt. I Figur 6.3 ser vi hur olika kluster kan se ut baserat på simulerad data. I Figur 6.4 ser vi hur modellen presterar för respektive fall.

Även om *K-means* är den modell som oftast används finns det andra modeller också, den intresserade läsaren kan läsa följande länk för att få mer information: <https://scikit-learn.org/stable/modules/clustering.html>.

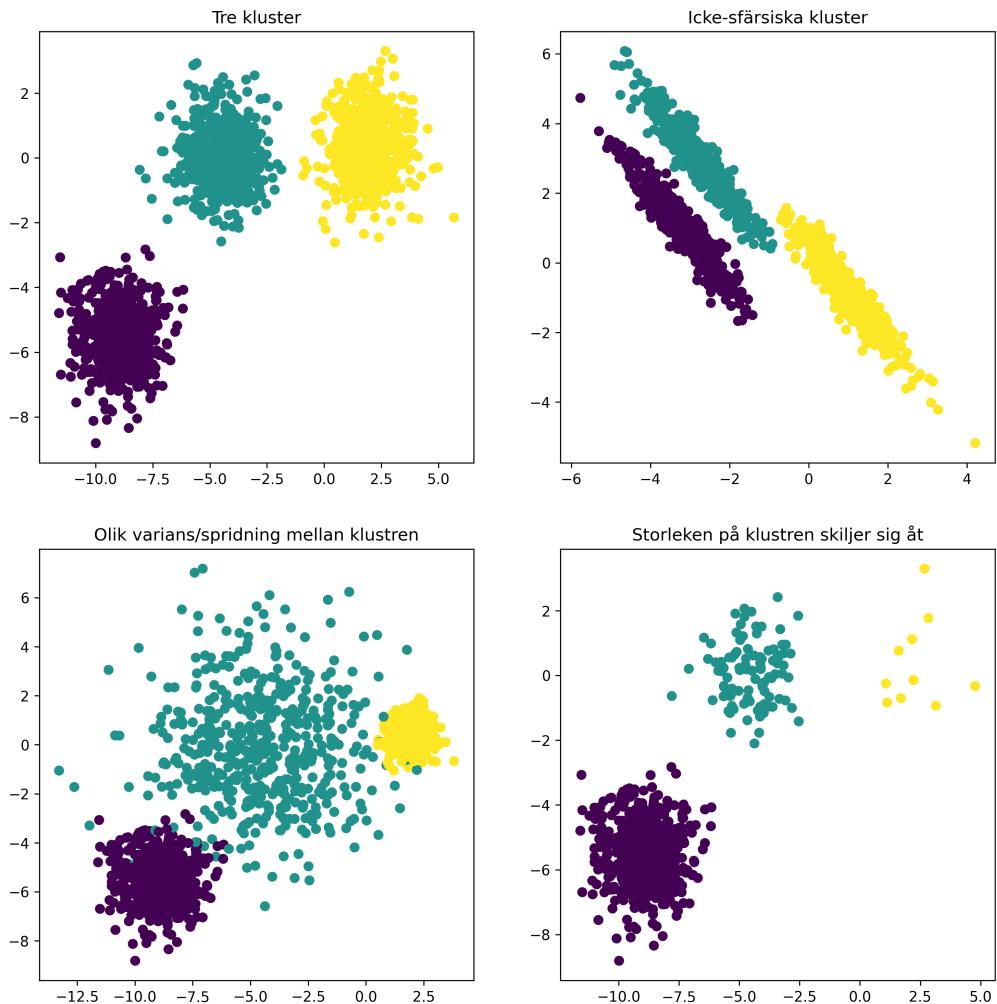
Generellt sett bör data skalas, med exempelvis `StandardScaler()`, innan den klustras. Detta eftersom många klustringsmodeller, inklusive *K-Means*, använder avstånd för att dela upp datapunkter i kluster. Om datan inte är skalad kan det leda till snedvridna resultat.

Härnäst kollar vi på hur klustering med *K-means* genomförs.

### **i** *Curse of dimensionality*

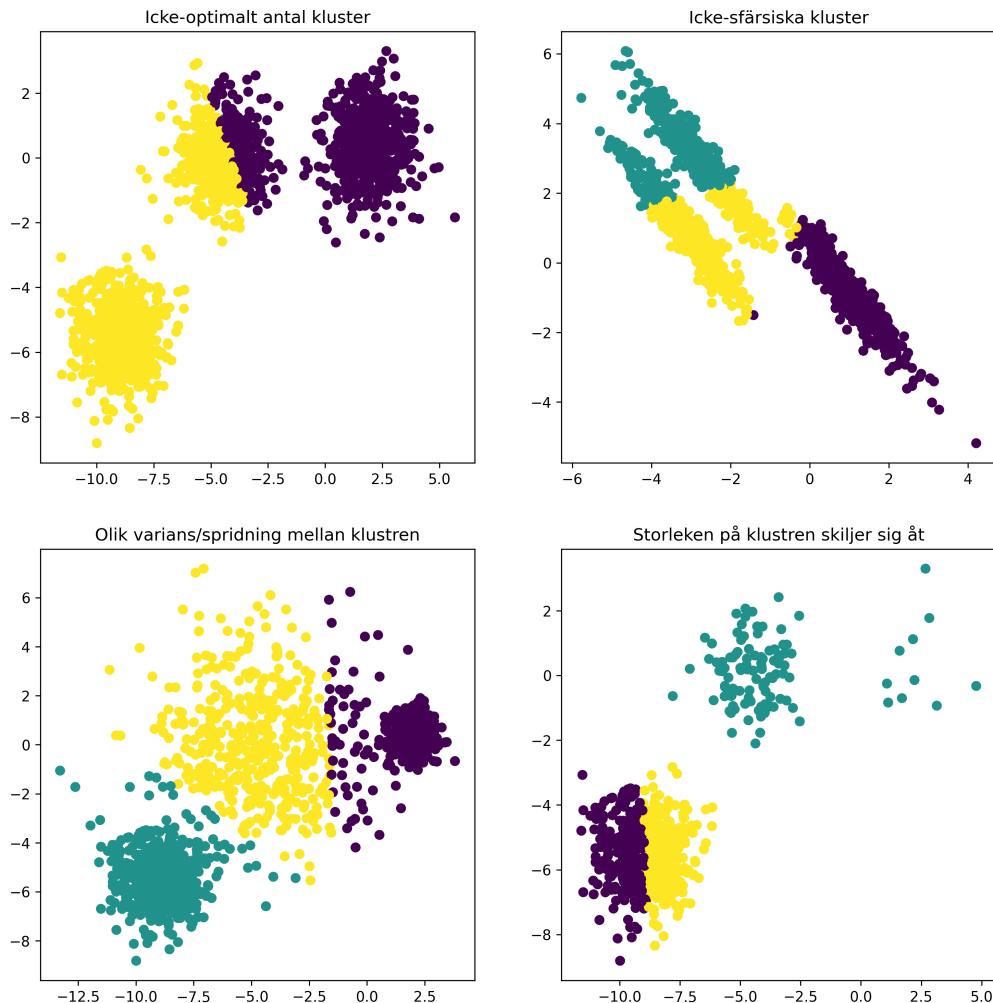
I Avsnitt 5.1 såg vi att i högre dimensioner blir avståndet mellan datapunkter generellt sett större. Det kan leda till utmaningar i samband med klustering varför dimensionsreducering, exempelvis med hjälp av *PCA*, kan vara användbart innan klustering genomförs.

## Sanna kluster



Figur 6.3: Visualisering av sanna klustertillhörigheter för simulerad data.

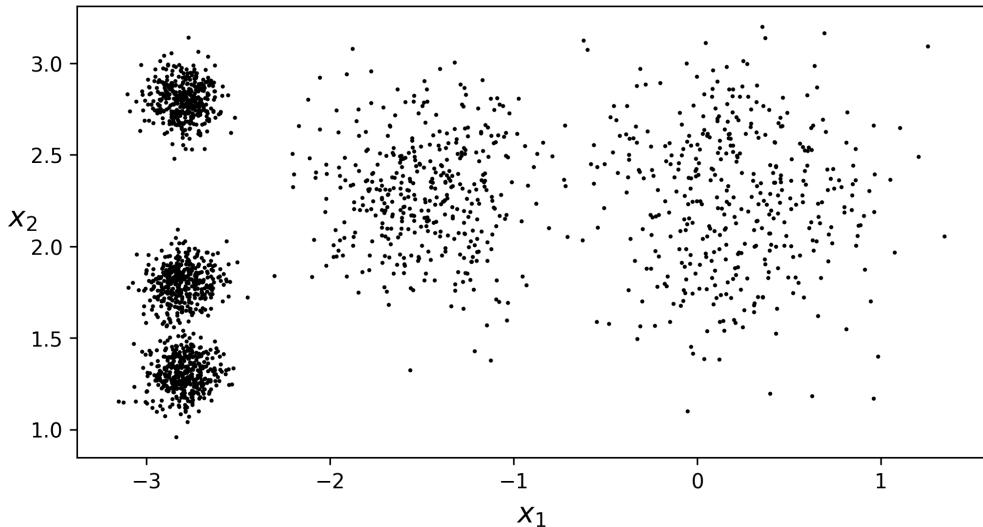
## Dåliga resultat med K-means



Figur 6.4: Visualisering av hur K-means-modellen presterar på datan simulerad från Figur 6.3. I den översta bilden till vänster ser vi att resultatet blir dåligt eftersom vi specificerat fel antal kluster i samband med modellträningen. I den översta bilden till höger ser vi att resultatet blir dåligt eftersom datan inte har en sfärisk form. I den nedre, vänstra bilden ser vi att klustren blir dåliga eftersom variansen/spridningen på datan i de olika klustren skiljer sig åt. I den sista bilden blir klusteringen dålig eftersom storleken på klustren skiljer sig åt.

### 6.2.1 Hur genomförs klustring med *K-Means*?

För att förklara vad modellen gör och hur den fungerar används ett dataset utan etiketter, se Figur 6.5.



Figur 6.5: Ett dataset med fem kluster.

När modellen körs kommer den att lokalisera en centroid för varje kluster och sedan kommer varje observation att tilldelas det kluster vars centroid är närmast. Observera att antalet kluster som modellen ska hitta måste anges manuellt och det görs i samband med modell-instantieringen. I vårt exempel med Figur 6.5 är det tydligt att antalet kluster är fem. För andra dataset, speciellt dataset med ett högre antal dimensioner där datan inte direkt kan visualiseras, kan det vara betydligt svårare att uppskatta antalet kluster. Hur valet av antal kluster sker kommer vi gå igenom i Avsnitt 6.2.3.

*K-Means* har flera justerbara hyperparametrar, se Tabell 6.1. Den hyperparameter som vi vanligtvis specificerar är `n_clusters`, de andra två i tabellen specificeras mer sällan, innehållande att dess standardvärdet används. Vad de innebär kommer vi att förstå efter att vi läst Avsnitt 6.2.2. Läsaren uppmanas att läsa *scikit-learn* dokumentationen för att få en helhetsbild.

Tabell 6.1: Ett urval av hyperparametrar för K-means från scikit-learns dokumentation.

Hyperparameter	Beskrivning	Standardvärde
<code>n_clusters</code>	Antal kluster modellen ska hitta. Av typen int.	8
<code>n_init</code>	Antal gånger modellen initieras med olika startvärden för centroider. 'auto' eller int.	"auto"
<code>max_iter</code>	Max antal iterationer per <code>n_init</code> . Av typen int.	300

Koden nedan tränar en *K-Means*-modell på det dataset som illustrerades i Figur 6.5.

```

from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans
①

blob_centers = np.array(
    [[ 0.2,  2.3],
     [-1.5 ,  2.3],
     [-2.8,  1.8],
     [-2.8,  2.8],
     [-2.8,  1.3]])
blob_std = np.array([0.4, 0.3, 0.1, 0.1, 0.1])
X, y = make_blobs(n_samples=2000, centers=blob_centers,
    cluster_std=blob_std, random_state=7)
②

kmeans = KMeans(n_clusters=5)
③
y_pred = kmeans.fit_predict(X)
④
print(y_pred)
print(kmeans.labels_)
⑤

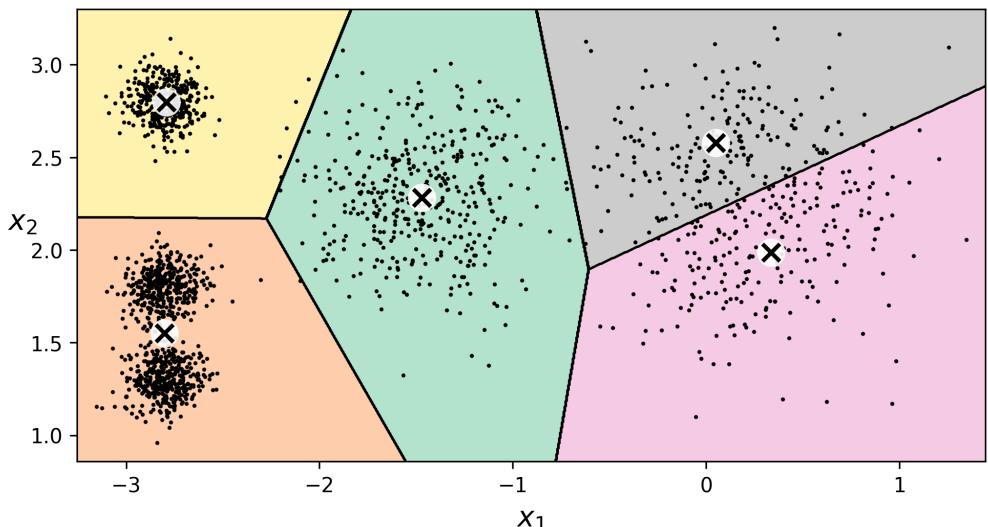
```

- ① Importera *K-Means* från *scikit-learn* biblioteket.
- ② Generera syntetisk data.
- ③ Initiera en *K-Means*-modell där vi sätter antal kluster till 5.
- ④ Träna *K-Means*-modellen på datasetet X och gör en prediktion över vilket kluster varje datapunkt tillhör.
- ⑤ `print(y_pred)` och `print(kmeans.labels_)` ger samma resultat.

```
[1 1 2 ... 3 4 1]  
[1 1 2 ... 3 4 1]
```

När modellen har körts har alla datapunkter tilldelats något av de fem klustren som modellen identifierat. Notera att siffrorna från det utskrivna resultatet (`print(y_pred)`, `print(kmeans.labels_)`) inte är en klassificering utan det visar vilket kluster respektive observation tillhör. Se Figur 6.6 för hur klustren ser ut. I stort kan vi se att modellen har lyckats med att tilldela de flesta observationer till korrekt kluster även om det finns några observationer som verkar hamna i fel kluster.

När varje datapunkt tilldelas ett kluster kallas detta för *hård klustering*. Det är även möjligt att ge varje observation en ”poäng” per kluster, till exempel baserat på avstånd till varje kluster. Detta kallas för *mjuk klustering*.



Figur 6.6: Illustration över hur centroider placeras och hur beslutsgränserna, representerat genom de färgade områdena, ser ut.

För att hitta koordinaterna för varje centroid kan följande kod användas.

```
kmeans.cluster_centers_
```

①

- ① Koordinater för respektive klusters centroid.

```
array([[ -1.47083264,   2.28276928],  
      [-2.80214068,   1.55162671],  
      [ 0.33567783,   1.98622883],  
      [-2.79290307,   2.79641063],  
      [ 0.05089714,   2.57710047]])
```

För att mäta avståndet mellan en observation och samtliga centroider kan följande kod användas.

```
kmeans.transform(X)
```

(1)

- ① Beräkna avstånd mellan varje observation och samtliga centroider i respektive kuster.

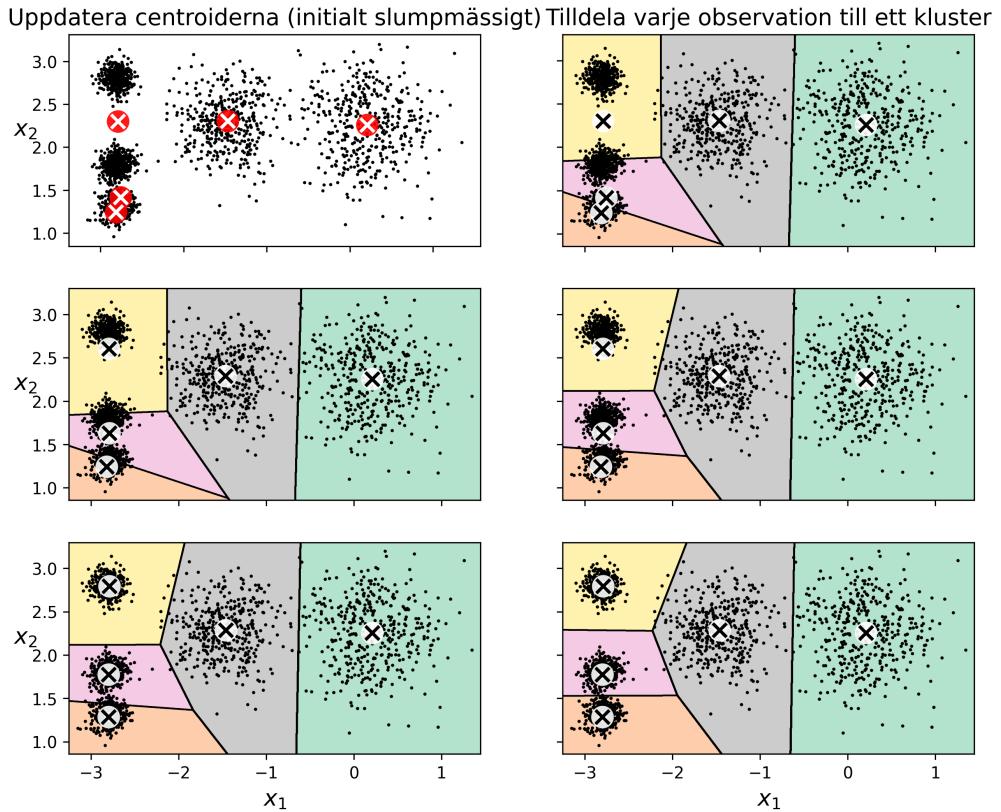
```
array([[ 1.54436296,  0.23085922,  3.1008427 ,  1.45402521,  3.01241845],  
      [1.48131641,  0.26810747,  3.2151229 ,  0.99002955,  3.02445682],  
      [2.67393196,  3.78216716,  1.02771976,  4.09069201,  1.67525714],  
      ...,  
      [1.40510347,  1.17785478,  3.22551467,  0.06769209,  2.85799771],  
      [1.71548744,  3.15905017,  0.42556519,  3.05913478,  0.25880028],  
      [1.21206498,  0.43658314,  2.97390701,  0.85434589,  2.75972638]])
```

## 6.2.2 Hur tränas modellen?

Vi vet nu vad modellen gör, men hur tränas den?

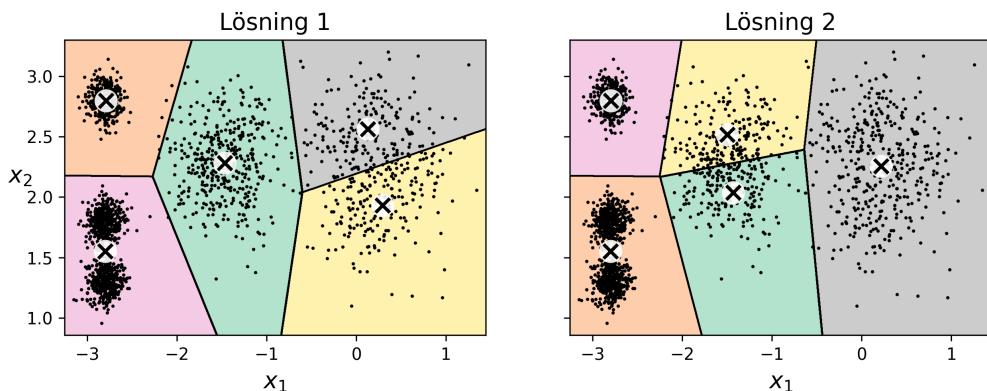
Modellträningen påbörjas genom att en slumpmässig initialisering av centroiderna sker. Varje observation tilldelas därefter det kluster vars centroid är närmast. Därefter så beräknas medelvärdet av alla observationers koordinater i respektive kluster och centroiderna uppdateras till dessa koordinater. Träningsalgoritmen fortsätter att iterera denna process. Antalet iterationer styrs av hyperparametern `max_iter` som vi såg i Tabell 6.1. Se Figur 6.7 för en visuell illustration av hur träningen går till.

I figuren ser vi i den första bilden (längst upp till vänster) att en slumpmässig initialisering gjorts. Därefter tilldelas varje observation ett kluster vilket vi ser i den andra bilden (längst upp till höger). I den tredje bilden (mittersta till vänster) ser vi hur centroiderna har uppdaterats. Efter att centroiderna har uppdaterats så tilldelas observationerna på nytt till det kluster vars centroid ligger närmast. Det ser vi i den fjärde bilden (mittersta till höger). I den femte bilden (längst ner till vänster) uppdateras centroiderna och i den sjätte bilden (längst ner till höger) tilldelas återigen varje observation det kluster



Figur 6.7: Visualisering av hur träningsalgoritmen för K-means fungerar. Initialt sker en slumpmässig placering av centroiderna för att varje observation ska kunna tilldelas det kluster vars centroid ligger närmast. Därefter fortsätter algoritmen att iterativt uppdatera centroiderna baserat på medelvärdet av varje observations koordinat och tilldela varje observation till det kluster vars centroid ligger närmast. Algoritmen konvergerar alltid i ett ändligt antal steg men lösningen, det vill säga det algoritmen konvergerar till, kan vara icke-optimal.

vars centroid ligger närmast. Från den femte bilden ser vi att lösningen verkar ha stabiliseras vilket innebär att inga större förändringar sker i kommande iterationer. Det går att teoretiskt visa att träningsalgoritmen kommer konvergera efter ett ändligt antal iterationer som vanligtvis är relativt få. Det innebär att centroidernas placering alltså inte längre uppdateras. Även om algoritmen alltså kommer att konvergera så kan det ske i ett lokalt optimum innebärande att lösningen inte är optimal. Huruvida lösningen blir optimal eller inte beror på den initiala initialiseringen av centroiderna. Se Figur 6.8 på hur två icke-optimala lösningar kan se ut, dessa har uppstått till följd av att den slumpmässiga initialiseringen av centroiderna i respektive fall varit olycklig.



Figur 6.8: Illustration av två icke-optimala lösningar som skett till följd av en olycklig initialisering av centroiderna i respektive fall.

För att hantera denna problematik med icke optimala lösningar så kan flera slumpmässiga initialiseringar av centroiderna göras. Detta styrs av hyperparametern `n_init` som vi såg i Tabell 6.1. Generellt sett så leder fler slumpmässiga initialiseringar till en bättre modell men det ökar också träningstiden. Om ingen direkt anledning finns så används i vanlig ordning `default`-värdet för hyperparametern.

Om vi genomför flera slumpmässiga initialiseringar så kan vi alltså få olika modeller. Algoritmen kommer själv att välja den bästa lösningen och det är den som returneras. Användaren kommer alltså inte att behöva hantera flera lösningar. Hur vet algoritmen vilken lösning som är den bästa? Den använder ett utvärderingsmått (kom ihåg att exempelvis `RMSE` var ett utvärderingsmått för regressionsproblem) som heter `inertia` som är medelkvadratavståndet mellan varje observation och dess närmaste centroid.

I koden nedan demonstreras hur vi skapar en *K-means*-modell som har fem kluster och

där det görs 20 slumpmässiga initialiseringar, varje initialisering itereras 400 gånger. Vi skriver även ut modellens *inertia*.

```
kmeans = KMeans(n_clusters=5, n_init=20, max_iter = 400)
kmeans = kmeans.fit(X)

print(kmeans.inertia_)
print(kmeans.score(X))
```

```
211.59853725816836
-211.59853725816836
```

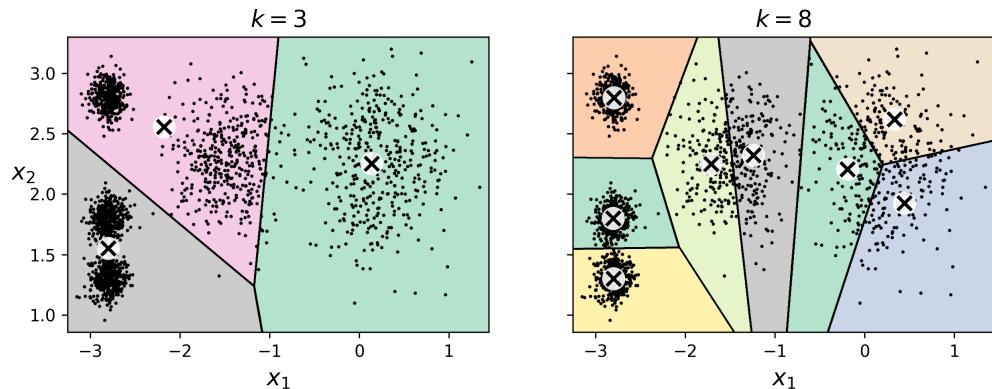
Från resultatet i koden ovan ser vi att `.score()` returnerar det negativa värdet på *inertia*. Varför negativt? Detta eftersom `.score()` metoden i *scikit-learn* alltid måste respektera regeln att ”högre värdet är bättre”, se informationsrutan i Avsnitt 1.3.5 om du behöver en repetition.

### 6.2.3 Val av antalet kluster

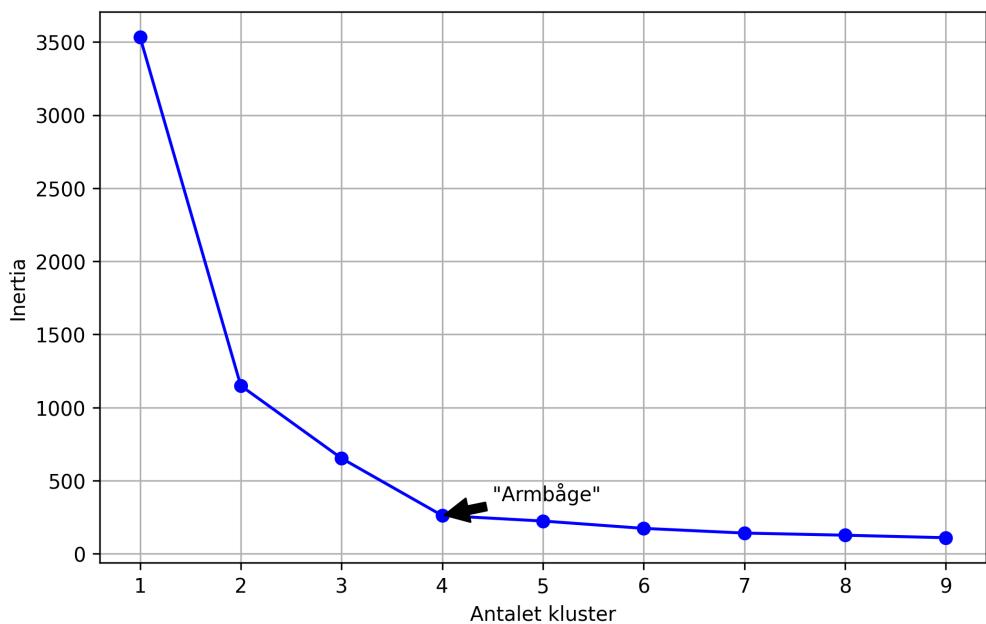
I Figur 6.5 var det enkelt att se att antalet kluster borde sättas till fem. Hade vi exempelvis valt tre eller åtta kluster istället hade resultatet blivit dåligt, vilket vi ser i Figur 6.9. Generellt sett är det dock svårt att avgöra antalet kluster eftersom vi inte på ett direkt sätt kan visualisera data som består av fyra eller fler variabler. Hur väljer vi då antalet kluster? Precis som när vi avgjorde den bästa lösningen i föregående avsnitt kan vi även för detta tänka oss att vi använder utvärderingsmåttet *inertia*.

Eftersom *inertia* är medelkvadratavståndet mellan varje observation och dess närmaste centroid så kommer det alltid att minska när antalet kluster ökar. Att därför välja den modell med lägst *inertia* kommer alltid leda till att vi väljer så många kluster som möjligt, det är alltså inte bra. Vi kan dock fortfarande visualisera hur *inertia* förändras i takt med antal kluster för att göra en bedömning på hur många kluster vi bör välja, se Figur 6.10. Där kan vi tydligt se att *inertia* snabbt sjunker fram till fyra kluster, för att sedan avta långsammare. Kurvan får en ”armbåges-likt” form efter fyra kluster. I detta fall vore det alltså rimligt att välja fyra kluster. Att välja färre kluster skulle ge ett betydligt högre värde på *inertia* och att välja fler kluster påverkar inte *inertia* i någon större utsträckning.

Att använda *inertia* som en metod för att välja rätt antal kluster kräver alltså att vi använder vårt omdöme för att göra ett någorlunda subjektivt val vilket kan vara utmanande. En annan mer precis metod som kan användas heter *silhouette score*.



Figur 6.9: Hade vi använt tre (vänster bild) eller åtta (höger bild) kluster för datan från Figur 6.5 hade resultatet blivit dåligt.



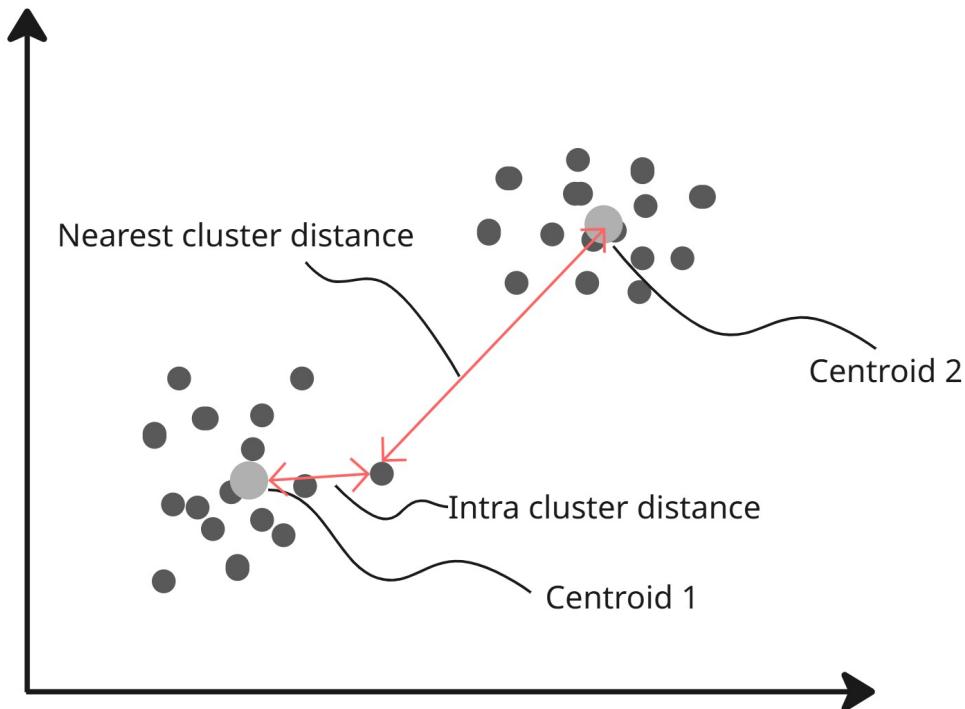
Figur 6.10: Visualisering av inertia som en funktion av antalet kluster.

*Silhouette score* är medelvärdet av varje datapunkts siluettkoefficient där en datapunkts siluettkoefficient ges av Ekvation 6.1

$$SC = \frac{b - a}{\max(a, b)} \quad (6.1)$$

där  $a$  = Medelvärdet av avståndet till alla andra punkter i samma kluster (*intra cluster distance*) och  $b$  = Medelvärdet av avståndet till alla andra punkter i det kluster som ligger närmast (*nearest cluster distance*).

Se Figur 6.11 för en visualisering av *intra cluster distance* och *nearest cluster distance*.



Figur 6.11: Illustration av "intra-cluster distance" samt "nearest cluster distance".

Siluettkoefficienten är alltid ett tal mellan -1 och +1. Om koefficienten är nära +1 betyder det att datapunkten befinner sig inom sitt eget kluster och har ett långt avstånd

till andra kluster. Om koefficienten är nära 0 betyder det att datapunkten ligger nära en klustergräns. Slutligen, om koefficienten är nära -1 betyder det att datapunkten potentiellt blivit tilldelad fel kluster. Målet är att *silhouette score* ska vara så hög som möjligt, alltså så nära +1 som möjligt.

Koden nedan visar hur vi kan beräkna *silhouette score*.

```
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score (1)

# Generating synthetic data
blob_centers = np.array(
    [[ 0.2,  2.3],
     [-1.5 ,  2.3],
     [-2.8,  1.8],
     [-2.8,  2.8],
     [-2.8,  1.3]])
blob_std = np.array([0.4, 0.3, 0.1, 0.1, 0.1])
X, y = make_blobs(n_samples=2000, centers=blob_centers,
                  cluster_std=blob_std, random_state=7)

# Creating two models and computing silhouette scores
kmeans_1 = KMeans(n_clusters=8, random_state = 5).fit(X)
kmeans_2 = KMeans(n_clusters=5, random_state = 5).fit(X)

sc_m1 = silhouette_score(X, kmeans_1.labels_) (2)
sc_m2 = silhouette_score(X, kmeans_2.labels_)

print("Silhouette score for model 1:", sc_m1)
print("Silhouette score for model 2:", sc_m2)

if sc_m1 > sc_m2:
    print("Model 1 is better")
elif sc_m1 == sc_m2:
    print("Models are equally good")
else:
    print("Model 2 is better")
```

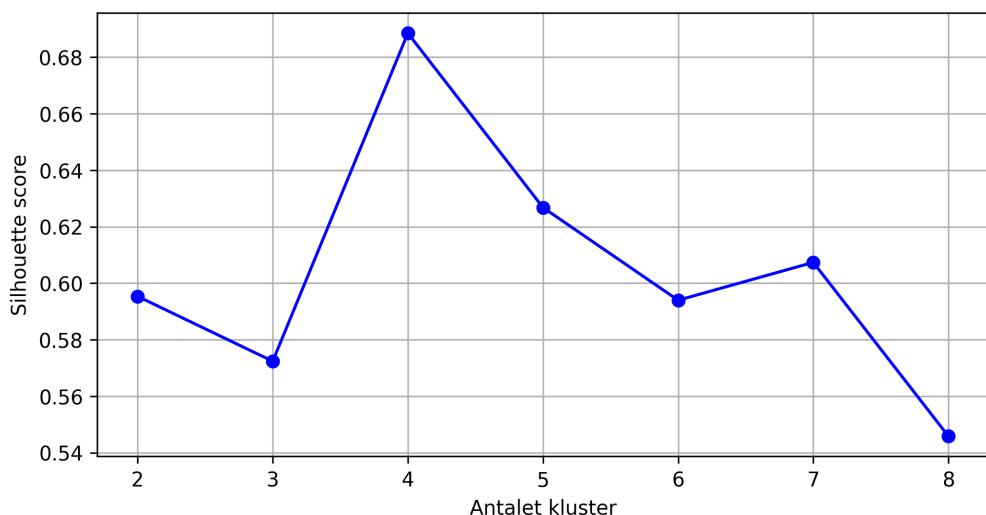
- ① Importera *silhouette score* från *scikit-learn* biblioteket.
- ② Beräkna *silhouette score* för den första modellen.

Silhouette score for model 1: 0.5592826029686692

Silhouette score for model 2: 0.6353926199764364

Model 2 is better

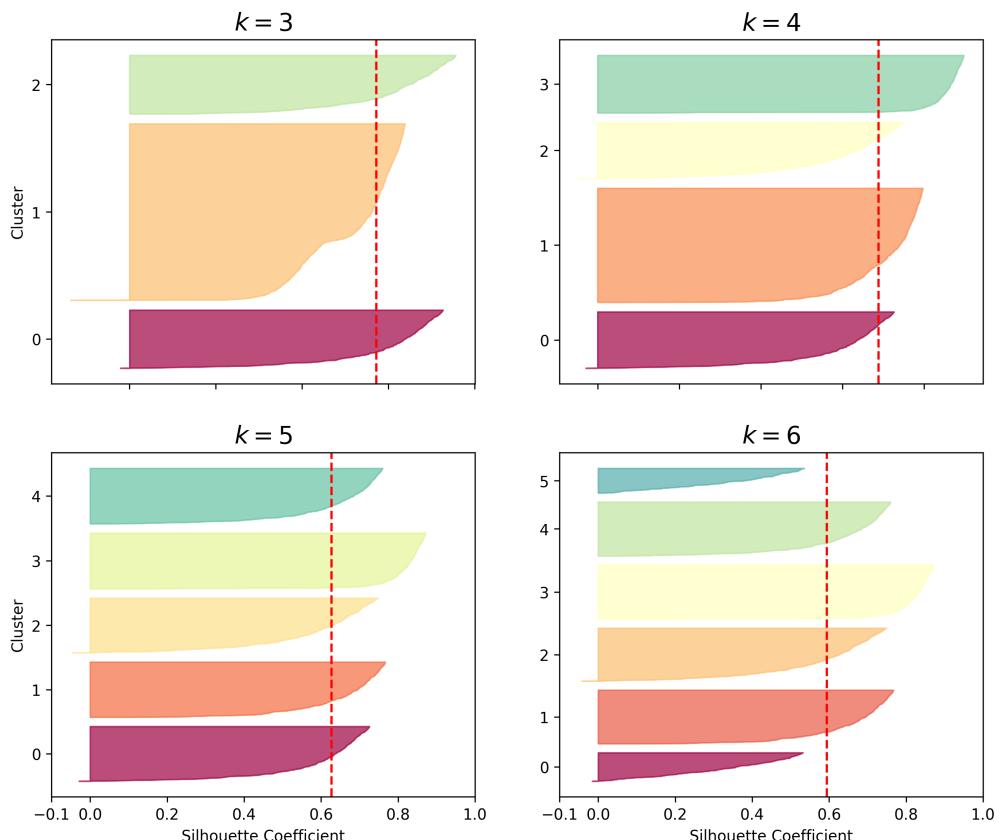
I Figur 6.12 visualiseras vi hur *silhouette score* varierar som en funktion av antal kluster. Även här bekräftas att fyra kluster verkar som ett rimligt val, men även fem kluster kan vara ett bra alternativ då *silhouette score* inte skiljer sig så mycket beroende på om vi väljer fyra eller fem kluster.



Figur 6.12: Silhouette score som en funktion av antalet kluster.

En ytterligare, än mer informativ representation, får vi genom att göra ett så kallat *silhouette diagram*, se Figur 6.13. Varje diagram visar en knivliknande form per kluster. Knivens höjd visar antalet observationer per kluster. Knivens bredd är siluettkoefficienterna sorterade (ju bredare, desto bättre). Den streckade linjen representerar medelvärdet av siluettkoefficienterna, alltså *silhouette score*. Om de flesta observationerna i ett kluster har en lägre siluettkoefficient än vad *silhouette score* är så är klustret troligtvis dåligt eftersom det betyder att många observationer ligger nära ett annat kluster. Därför vill vi generellt sett att knivarna ska passera den streckade linjen.

Om vi kollar på Figur 6.13 ser vi att när vi har sex ( $k = 6$ ) kluster så blir resultatet sämre än i övriga fall, *silhouette score* är lägre och alla knivarna passerar inte den streckade linjen. Vi ser att när vi väljer tre kluster ( $k = 3$ ) så blir det kluster med index 1 (den andra kniven) väldigt stort. Det sker även i det fall vi har fyra kluster ( $k = 4$ ). Trots att *silhouette score* är lägre när vi använder fem kluster så är det ett rimligt val att göra eftersom storleken på klustren är likvärdiga, något vi generellt sett vill i samband med att vi använder *K-means* modellen. Vi väljer alltså att i detta fallet välja fem kluster. I praktiken hade vi dock likväld kunnat välja tre eller fyra, det är alltså ingen exakt vetenskap och vi behöver använda vårt omdöme.



Figur 6.13: Silhouette diagram.

## 6.3 Två kodexempel

Vi kommer nu kolla på två kodexempel där det första demonstrerar hur klustering kan användas för att genomföra en kundsegmentering och det andra demonstrerar semivägledd inlärning.

### 6.3.1 Kodexempel 1 - Kundsegmentering

I detta kodexempel kommer vi demonstrera hur en kundsegmentering hade kunnat gå till.

Vi börjar med att importera bibliotek och generera ett syntetiskt dataset som används för demonstrationssyfte.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler

# Simulating a dataset for demonstration
np.random.seed(42)

# Three groups of customers
age_1 = np.random.randint(18, 30, 60)
age_2 = np.random.randint(30, 70, 60)
age_3 = np.random.randint(18, 70, 30)

income_1 = np.random.randint(15000, 30000, 60)
income_2 = np.random.randint(29000, 50000, 60)
income_3 = np.random.randint(60000, 90000, 30)

# Combine the data
ages = np.concatenate([age_1, age_2, age_3])
incomes = np.concatenate([income_1, income_2, income_3])

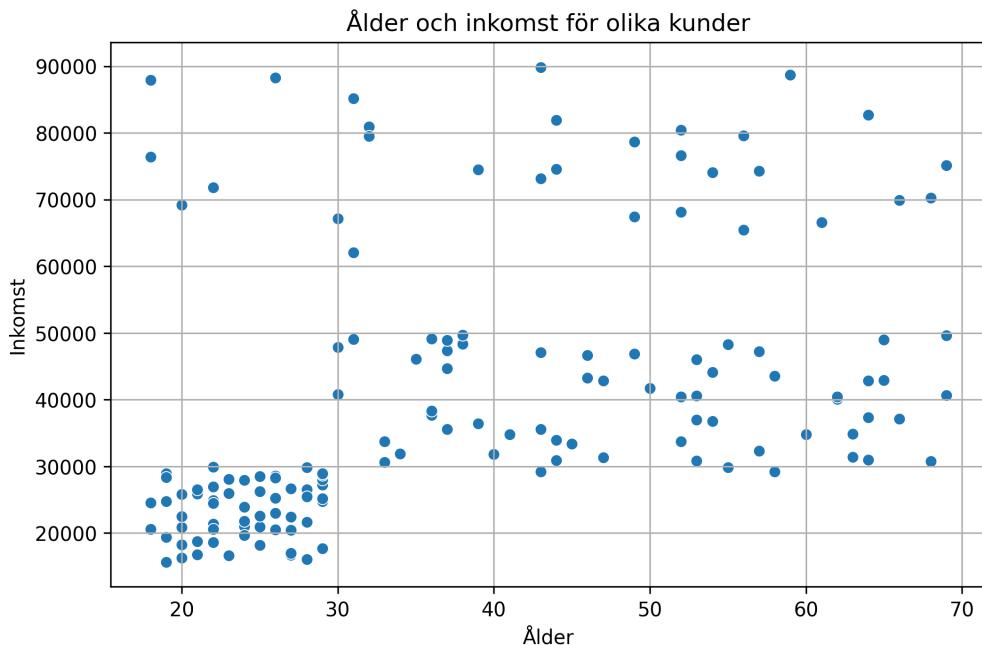
# Store the data in a DataFrame
```

```
data = pd.DataFrame({'Age': ages, 'Income': incomes})
data = data.sample(frac=1, random_state=11).reset_index(drop=True)
①
```

- ① Denna kod gör att raderna för datan sorteras om slumpmässigt. Görs för att vi senare kör koden `data.head(10)` och vill visa olika kluster.

I koden nedan visualiseras vi datan vi simulerat.

```
plt.figure(figsize=(8, 5))
sns.scatterplot(data=data, x='Age', y='Income')
plt.title('Ålder och inkomst för olika kunder')
plt.xlabel('Ålder')
plt.ylabel('Inkomst')
plt.grid(True)
```



Därefter standardiseringar vi vår data och tränar en *K-means*-modell.

```
# Scale the data for K-means
scaler = StandardScaler()
scaled_data = scaler.fit_transform(data)

# Use K-means for clustering
kmeans = KMeans(n_clusters=3, random_state=42)
data['Segment/Cluster'] = kmeans.fit_predict(scaled_data)      ①
```

- ① Vi skapar en ny kolumn i vår data som heter `Segment/Cluster`, det leder till att varje kund får ett segment som den tillhör.

Skriv ut några rader från datan så ser det ut enligt nedan.

```
data.head(10)
```

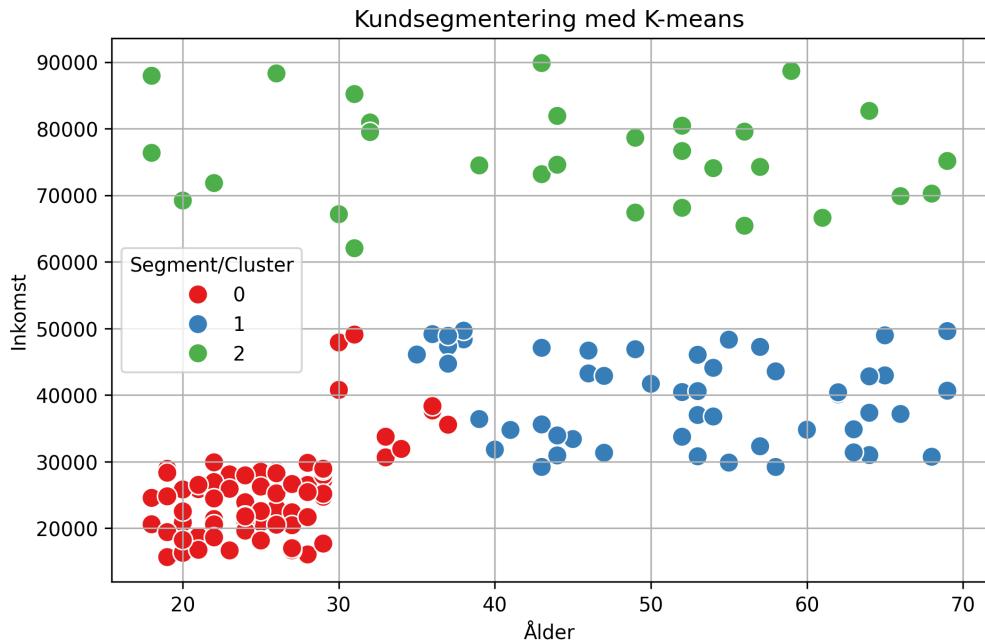
	Age	Income	Segment/Cluster
0	38	48360	1
1	49	67400	2
2	22	71835	2
3	26	22994	0
4	36	37680	0
5	28	16059	0
6	43	35585	1
7	25	20915	0
8	29	24721	0
9	37	47237	1

Visualiseringar vi våra segment ser det ut enligt nedan.

```
plt.figure(figsize=(8, 5))
sns.scatterplot(data=data, x='Age', y='Income',
                 hue='Segment/Cluster', palette='Set1', s=100)
plt.title('Kundsegmentering med K-means')      ①
plt.xlabel('Ålder')
```

```
plt.ylabel('Inkomst')
plt.grid(True)
```

- ① Här har vi använt biblioteket *Seaborn* för att på ett enkelt sätt göra en mer avancerad visualisering.



Från visualiseringen ser vi att vi har tre segment och att det första segmentet (med index 0) generellt sett är yngre män med lägre inkomst, det andra segmentet (med index 1) är män mellan cirka 30-70 år som har medellöner och det tredje segmentet är män mellan cirka 18-70 år som har höga inkomster. Om man exempelvis är en sportbilstillverkare så kan man tänka sig att det segmentet med index 2 är mest aktuellt eftersom de har högst inkomst medan en biltillverkare som kanske säljer familjebilar hade riktat in sig på segmenten med index 0 och 1.

Om vi fortsätter med exemplet så såg vi ovan, när vi körde koden `data.head(10)`, att vi hade tre segment. Dessa kan vi namnge för att personer i ett företag ska kunna prata om de olika segmenten på ett mer naturligt sätt. Det görs i koden nedan.

```

# Map numeric labels to names
cluster_names = {
    0: 'basic',
    1: 'medium',
    2: 'premium'
}

data['segment_name'] = data['Segment/Cluster'].map(cluster_names)
data.head(10)

```

	Age	Income	Segment/Cluster	segment_name
0	38	48360	1	medium
1	49	67400	2	premium
2	22	71835	2	premium
3	26	22994	0	basic
4	36	37680	0	basic
5	28	16059	0	basic
6	43	35585	1	medium
7	25	20915	0	basic
8	29	24721	0	basic
9	37	47237	1	medium

Nästa steg hade nu kunnat vara att lära sig mer om de olika segmenten för att exempelvis kunna anpassa marknadsföringen och produkterna som erbjuds. Exempelvis hade en marknadsavdelning kunnat genomföra intervjuer eller i de fall data om kunderna finns kan olika former av dataanalys göras. Om vi då kommer fram till att exempelvis premium segmentet inte kollar på tv men läser tidningen så vore det mer lämpligt att skicka ut reklam via post än att köpa reklamtid på en tv-kanal om vi vill nå det segmentet.

### 6.3.2 Kodexempel 2 - Semi-vägledd inlärning

I detta kodexempel kommer vi demonstrera hur semi-vägledd inlärning kan se ut. Semi-vägledd inlärning karakteriseras av att endast en delmängd av vår träningsdata har etiketter och vissa observationer alltså saknar etiketter. Vanligtvis är det så att antalet observationer som har etiketter i förhållande till antalet observationer som saknar etiketter är mycket litet.

Vi börjar med att importera bibliotek och ladda in data. Vi kommer använda oss av `load_digits` som i likhet med *MNIST* innehåller handskrivna siffror, men detta dataset är mindre och enklare.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.cluster import KMeans

X_digits, y_digits = load_digits(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X_digits,
    ↵ y_digits, random_state=42)
```

Vi antar nu att vi endast har 50 observationer och ska träna en logistisk regressionsmodell. Då får vi resultatet enligt koden nedan.

```
n_labeled = 50

log_reg = LogisticRegression(random_state=42)
log_reg.fit(X_train[:n_labeled], y_train[:n_labeled])
log_reg.score(X_test, y_test)
```

0.8266666666666667

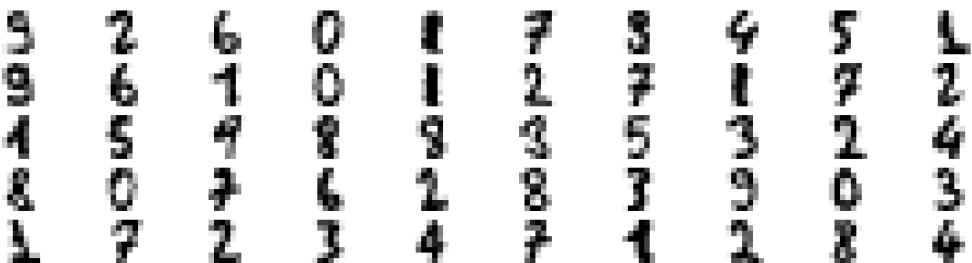
Nu kommer vi klustra vår träningsdata med hjälp av *K-means* där vi kommer skapa femtio kluster. För varje kluster kommer vi därefter hitta den bild som är närmast centroiden. De bilderna blir det vi kallar för representativa bilder (*representative digit* på engelska).

```
k = 50

kmeans = KMeans(n_clusters=k, random_state=42)
X_digits_dist = kmeans.fit_transform(X_train)
representative_digit_idx = np.argmin(X_digits_dist, axis=0)
X_representative_digits = X_train[representative_digit_idx]
```

Vi gör nu en visualisering av de representativa bilderna.

```
plt.figure(figsize=(8, 2))
for index, X_representative_digit in
    enumerate(X_representative_digits):
    plt.subplot(k // 10, 10, index + 1)
    plt.imshow(X_representative_digit.reshape(8, 8),
    cmap="binary")
    plt.axis('off')
```



De representativa bilderna behöver vi nu manuellt sätta en etikett (*label*) på, detta kallas för *data annotation*. Vi kollar på siffrorna från visualiseringen ovan och fyller i etiketter enligt nedan.

```
y_representative_digits = np.array([
    9, 2, 6, 0, 1, 7, 9, 4, 5, 1,
    9, 6, 1, 0, 1, 2, 7, 1, 7, 2,
    1, 5, 9, 8, 8, 3, 5, 3, 2, 4,
    8, 0, 7, 6, 2, 8, 3, 9, 0, 3,
    1, 7, 2, 3, 4, 7, 1, 2, 8, 4
])
```

Nu tränar vi vår modell på dessa 50 representativa bilder istället för de 50 slumpmässigt valda bilder som användes tidigare.

```
log_reg = LogisticRegression(max_iter=5000, random_state=42)
log_reg.fit(X_representative_digits, y_representative_digits)
```

```
log_reg.score(X_test, y_test)
```

```
0.9133333333333333
```

Vi ser att vi fick ett mycket bättre resultat efter att vi använde de representativa bilderna. Generellt sett är det kostsamt att sätta etiketter på data och därför är det ofta en god idé att etiketter sätts på bilder som är så representativa som möjligt.

Ytterligare ett förfarande vi kan prova är att sätta etiketter på samtliga observationer i varje kluster baserat på etiketterna från de representativa bilderna. Detta benämns *label propagation*.

```
y_train_propagated = np.empty(len(X_train), dtype=np.int32)
for i in range(k):
    y_train_propagated[kmeans.labels_==i] =
        y_representative_digits[i]
```

Vi får nu resultatet enligt nedan vilket vi ser är ytterligare lite bättre även om skillnaden är väldigt liten.

```
log_reg = LogisticRegression(max_iter=5000, random_state=42)
log_reg.fit(X_train, y_train_propagated)

log_reg.score(X_test, y_test)
```

```
0.9222222222222223
```

Eftersom vi satte etiketter på samtliga datapunkter i vardera kluster så sattes det alltså också på datapunkter som är nära klustergränserna. Kanske går det att få ännu bättre resultat om vi endast sätter etiketter på de datapunkter som ligger närmast respektive klusters centroid, exempelvis de 75% närmaste? Vi lämnar det åt den intresserade läsaren att prova.

## 6.4 Övningsuppgifter

Övningsuppgifter för kapitlet finns på bokens hemsida:  
[https://github.com/AntonioPrgomet/ai\\_tillaempad\\_ml](https://github.com/AntonioPrgomet/ai_tillaempad_ml)



## Del III

# Djupinlärning



## Kapitel 7

# Artificiella neurala nätverk (ANN)

I detta kapitel kommer vi börja med *djupinlärning* (förkortas ofta DL efter engelskans *deep learning*) där vi kommer lära oss om *artificiella neurala nätverk* (ANN). Djupinlärning är ett delämne inom ML innehållande att vi fortfarande har nyttat av alla de fundamentala koncept (såsom uppdelningen i träningsdata, valideringsdata och testdata samt hyperparametrar och parametrar) vi lärt oss tidigare. Hittills i boken har vi använt *scikit-learn* för att skapa modellerna medan vi nu kommer använda oss av *Keras* och *Tensorflow*. Vi noterar dock att vi fortfarande har nyttat av *scikit-learn*, exempelvis när vi i början av modelleringen delar upp data i träningsdata och testdata. Det vi lär oss i detta kapitel kommer vi ha direkt nyttat av i följande två kapitel, Kapitel 8 och Kapitel 9.

Läsaren kommer med start i detta kapitel märka att implementeringen av neurala nätverk generellt sett är mer komplex än de modellerna vi tidigare lärt oss. Kom därför ihåg att du alltid kan (och kommer behöva) vända dig till officiell dokumentation och diverse källor på internet för att ta reda på mer information och fördjupa din förståelse.

Kapitlet inleder med att ge en bakgrund följt av en genomgång av modellarkitekturen där vi lär oss hur neurala nätverk fungerar. Vi kommer även få riktlinjer för hur modellarkitekturen kan väljas, något vi har nyttat av i praktiskt arbete. Därefter diskuteras regularisering och hur träning av neurala nätverk med *backpropagation* går till innan kapitlet avslutas med två kodexempel.

## 7.1 Bakgrund

Artificiella neurala nätverk (ANN) har funnits sedan 1950-talet. På grund av att datorerna på den tiden inte var tillräckligt kraftfulla och den teoretiska förståelsen för modellerna inte lika utvecklad som den är idag, levde de inte upp till de högt ställda förväntningarna. Detta ledde till minskat intresse och minskad forskningsaktivitet inom området. Under 1970- och 1980-talen var intresset för ANN svalt och perioden brukar ofta benämnas som ”AI-vintern”.

Datorernas tekniska utveckling, framväxten av nya modeller såsom *convolutional neural networks (CNN)* och *recurrent neural networks (RNN)* samt utvecklingen av viktiga teoretiska resultat såsom träningsalgoritmen *backpropagation* – är avgörande faktorer bakom de artificiella neurala nätverkens framgångar i modern tid. År 2010 användes neurala nätverk inom bland annat bildenanalys och presterade då bättre än andra dominerande modeller. Detta brukar ofta anses vara startskottet för ”AI-våren” där utvecklingen, tillämpningarna, finansieringen, forskningen och intresset i största allmänhet för artificiella neurala nätverk åter var i blom. Utvecklingen av imponerande tillämpningar, exempelvis chattbottar såsom *ChatGPT* (som vi kommer lära oss om senare i boken, Kapitel 10) har gjort att neurala nätverk och AI i största allmänhet är ett mycket populärt ämne. Populariteten och intresset för neurala nätverk och AI rent allmänt kan vi också se genom det faktum att nobelpriset i fysik år 2024 tilldelades John J. Hopfield och Geoffrey Hinton med motiveringen ”*for foundational discoveries and inventions that enable machine learning with artificial neural networks*”. Sammanfattningsvis kan vi säga att de framgångar som har gjorts med hjälp av neurala nätverk har lett till att intresset för dem är stort.

## 7.2 Modellarkitektur

I presentationstalet för nobelpriset i fysik år 2024 sägs följande  
(källa: [www.nobelprize.org/prizes/physics/2024/225093-ceremony-speech-swedish/](http://www.nobelprize.org/prizes/physics/2024/225093-ceremony-speech-swedish/)):

(...) Årets Nobelpristagare i fysik, John Hopfield och Geoffrey Hinton, inspirerades av neuronnätverket i den mänskliga hjärnan. De har utvecklat artificiella neurala nätverk som är grundläggande för maskininlärning, som möjliggör att datorer lär sig utan explicit programmering. (...)

Precis som citatet ovan säger så har ANN-modellerna inspirerats av hur hjärnan fungerar med dess neuronnät.

**i** Liknelsen mellan artificiella neurala nätverk (ANN) och hjärnan är användbar för att skapa en intuition för hur nätverken fungerar. Samtidigt har mycket av utvecklingen inom ANN skett utan direkt inspiration från eller koppling till hur hjärnan faktiskt fungerar.

Kollar vi på Figur 7.1 ser vi hur ett ANN kan se ut. Vi ser att de huvudsakliga beståndsdelarna är: *Input layer*, *hidden layers* (dolda lager) samt *output layer* där varje lager består av noder/neuroner. Detta neurala nätverket har två dolda lager. Vi ser också att det finns två *outputs*,  $y_1$  och  $y_2$  innehållande att det neurala nätverket alltså kan utföra två prediktioner samtidigt, exempelvis kan vi tänka oss att det kan prediktera en persons ålder ( $y_1$ ) och lön ( $y_2$ ). Pilarna i figuren representerar hur vi går från *input* - *hidden layer 1* - *hidden layer 2* - *output*. När noderna i ett lager är kopplade till alla noder i föregående lager sägs det vara *dense* eller *fully connected layer*. En biologisk liknelse är:

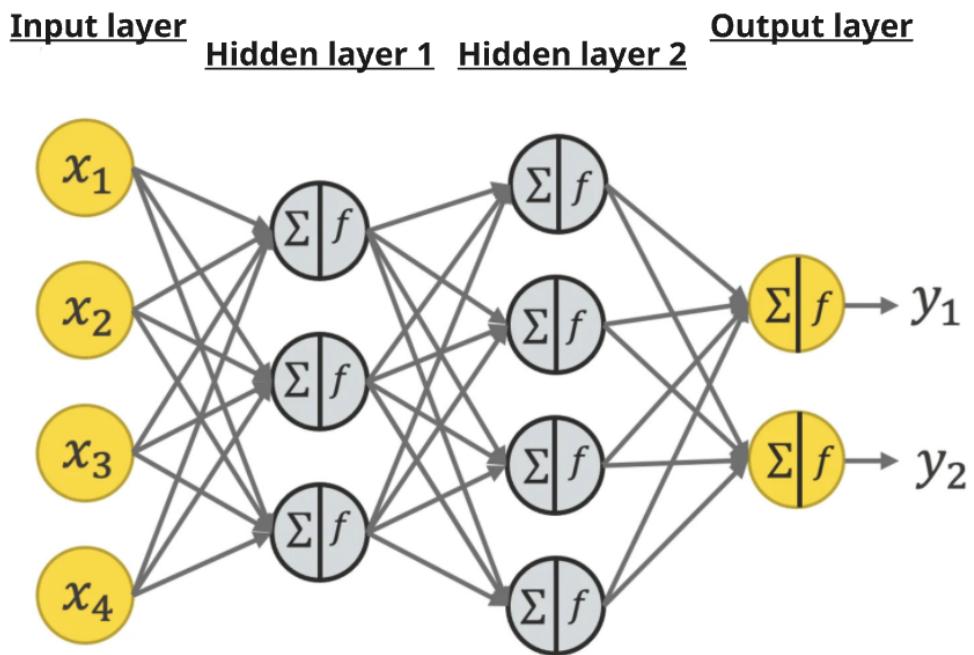
1. *Input layer*: Vi får ett sinnesintryck med hjälp av våra sinnen, det motsvarar att vi får in något. Exempelvis kan vi tänka oss att vi med våra ögon *ser* en farlig tiger.
2. *Hidden layer*: Det vi har fått in bearbetas i det "dolda", exempelvis diverse tankeprocesser. Exempelvis tänker vi att tigern är farlig och att vi bör avlägsna oss.
3. *Output layer*: Efter att sinnesintrycket bearbetats får vi ut något, exempelvis en slutsats. Exempelvis drar vi slutsatsen att vi bör springa så vi kommer bort ifrån tigern.

### **i Neurala nätverk och djupinlärning (DL)**

I Figur 7.1 hade det neurala nätverket två dolda lager och rent generellt kan vi själva välja hur många sådana som ska användas. Begreppet djupinlärning refererar till det faktum att neurala nätverk kan ha många dolda lager och blir då "djupa". I praktiken finns det ingen exakt definition på hur många dolda lager ett neutralt nätverk ska ha för att anses vara "djupt" och begreppen neurala nätverk och djupinlärning används ofta synonymt.

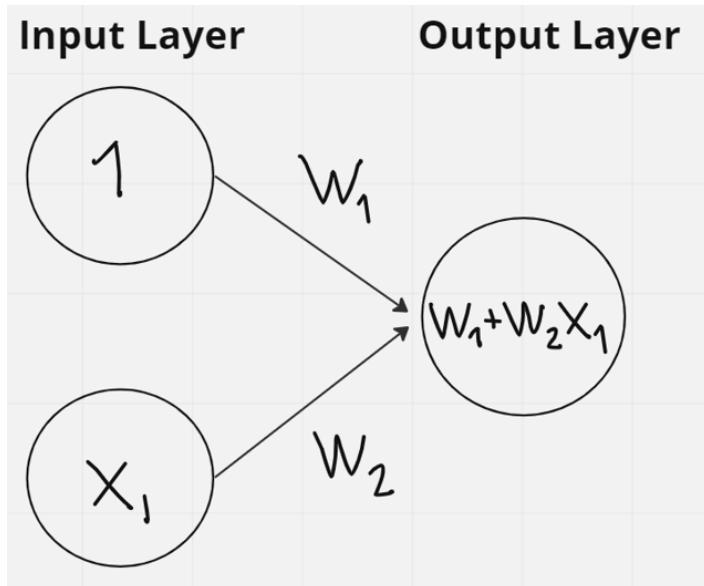
#### **7.2.1 Hur fungerar ett neutralt nätverk**

Vi kollar nu i mer detalj på hur ett neutralt nätverk faktiskt fungerar där vi kommer lära oss om den typ av neurala nätverk som benämns för *single-layer perceptron* och *multilayer perceptron (MLP)*.



Figur 7.1: Exempel på hur ett ANN kan se ut.

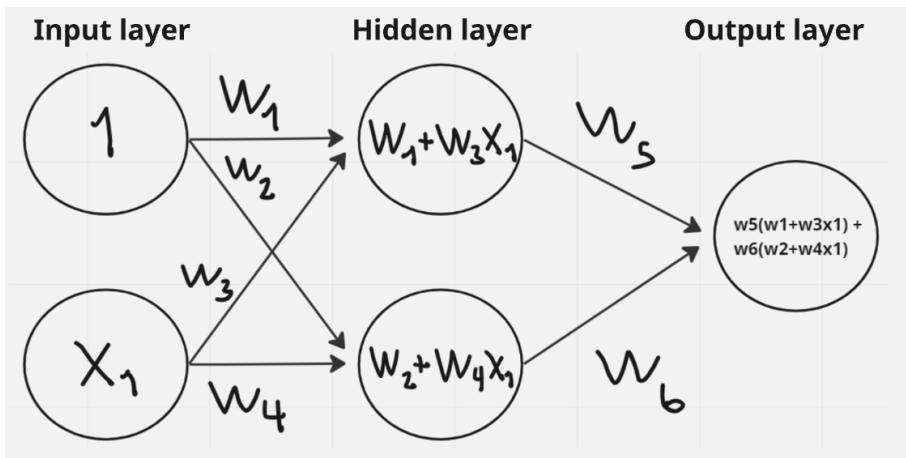
Vi börjar först med att visa något som *inte fungerar*. Se Figur 7.2. I figuren ser vi att vi har två *input*-noder som är 1 och  $x_1$ . Dessa noderna är kopplade till *output*-noden genom vikterna  $w_1$  och  $w_2$ . *Input*-noderna multipliceras med vikterna och adderas, vi får då att *outputen*,  $y$ , är  $y = w_1 + w_2x_1$ . Om vi kollar på uttrycket kan vi se att om vi hade kallat  $w_1$  för  $\theta_1$  och  $w_2$  för  $\theta_2$  är outputen exakt densamma som för *linjär regression*, nämligen  $y = \theta_1 + \theta_2x_1$ . Vi har alltså infört “en ny typ av modell” men får exakt samma resultat som för linjär regression. Låt oss därför testa något som är mer komplex, se Figur 7.3.



Figur 7.2: En enkel arkitektur för ett neuralt nätverk vars prediktion dock inte skiljer sig från vanlig enkel linjär regression.

### **i Bias-nod**

I exemplet kopplat till Figur 7.2 såg vi att den första *input*-noden var en 1. Vi kan fråga oss vad vi ska med den till och anledningen är att vi tack vare den fick ett intercept, nämligen  $w_1$  i  $y = w_1 + w_2x_1$ . Denna 1 kallas för *bias-nod* och är alltid inkluderad. Notera, *bias* ska i detta sammanhanget inte blandas ihop med *bias* i sammanhanget *bias variance trade-off*.



Figur 7.3: En mer komplex arkitektur för ett neuralt nätverk vars prediktion dock inte skiljer sig från vanlig enkel linjär regression.

Vi ser nu att vi har samma typ av *input-layer* och har lagt till ett dolt lager. Vår output blir i detta fall:

$$\begin{aligned}
 y &= w_5(w_1 + w_3 x_1) + w_6(w_2 + w_4 x_1) \\
 &= (w_1 w_5 + w_2 w_6) + (w_3 w_5 x_1 + w_4 w_6 x_1) \\
 &= (w_1 w_5 + w_2 w_6) + (w_3 w_5 + w_4 w_6) x_1 \\
 &= \text{konstant} + \text{konstant} \cdot x_1 \\
 &= \theta_1 + \theta_2 x_1
 \end{aligned}$$

I andra och tredje ledet har vi använt enkel algebra för att gruppera om termerna så att vi i fjärde ledet ser att vi har ett uttryck på formen  $\text{konstant} + \text{konstant} \cdot x_1$ . Notera att vikterna,  $w_i$  för godtyckligt index  $i$ , är konstanter som har konkreta siffror när modellerna tränas. I näst sista ledet har vi  $\text{konstant} + \text{konstant} \cdot x_1$  där konstanterna kan kallas för det vi vill, i sista ledet har vi alltså använt namnen  $\theta_1$  och  $\theta_2$  för de två konstanterna.

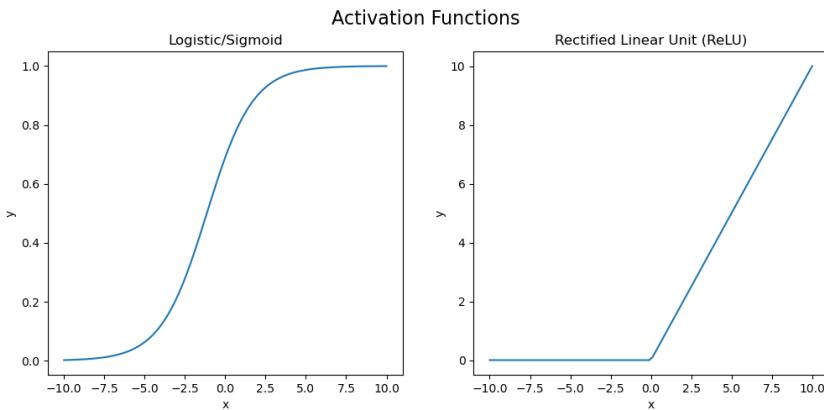
Vi ser att vi, trots en ökning i komplexitet, återigen hamnat på ett resultat som är exakt samma som för enkel linjär regression,  $y = \theta_1 + \theta_2 x_1$ .

Lösningen för att faktiskt få ett neuralt nätverk som ger något annat resultat än vad den linjära regressionsmodellen gör är att använda det som benämns för *aktiveringsfunktion*. Två exempel på aktiveringsfunktioner är *Logistic/Sigmoid* samt *Rectified Linear Unit*

(ReLU), se ekvationerna nedan och Figur 7.4 för hur de ser ut.

$$\text{Logistic / Sigmoid: } \sigma(z) = \frac{1}{1 + e^{-z}} \quad (7.1)$$

$$\text{Rectified Linear Unit (ReLU): } \text{ReLU}(z) = \max(0, z) \quad (7.2)$$



Figur 7.4: Två exempel på aktiveringsfunktioner.

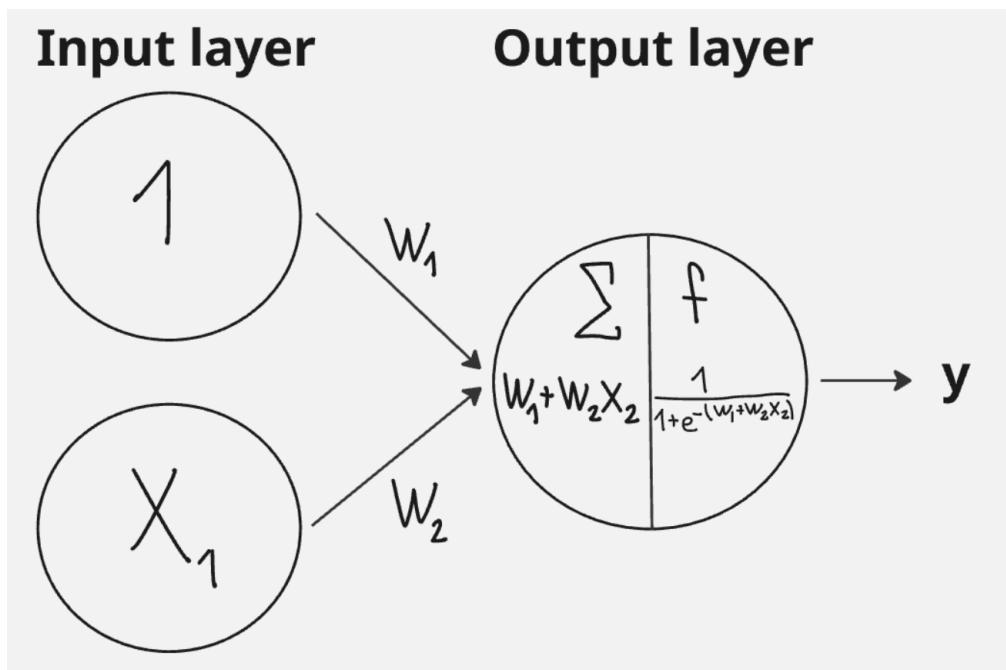
Andra exempel på aktiveringsfunktioner är de som benämns för *tanh*, *ELU* och *SELU* som den intresserade läsaren kan läsa in sig på.

När vi nu har aktiveringsfunktioner kan vi skapa ett neutralt nätverk som ser ut enligt Figur 7.5. Vi ser alltså att vi precis som tidigare beräknar viktade summor, som representeras av summa-symbolen  $\Sigma$ , men nu tar vi de viktade summorna som argument till aktiveringsfunktionen, detta representeras av funktions-symbolen  $f$ . I exemplet med Figur 7.5 har vi använt *sigmoid*, Ekvation 7.1, som aktiveringsfunktionen. Vi har nu fått något som skiljer sig från den linjära regressionsmodellen.

Det neurala nätverket vi såg i Figur 7.5 är det som benämns för *single-layer-perceptron*. Motsvarande arkitektur, fast med fler dolda lager som i Figur 7.1, benämns för *multi-layer perceptron*. Det som karakteriseras *perceptron*-arkitekturen är att de är av typen *feedforward* innehållande att datan går från input-till-output i framåt-riktning, är *fully connected/dense*, innehållande att alla noder i ett lager är kopplat till noderna i föregående lager, och har icke-linjära aktiveringsfunktioner. Aktiveringsfunktionerna i Ekvation 7.1 och Ekvation 7.2 samt *tanh*, *ELU* och *SELU* är alla icke-linjära.

### i Feedforward och Recurrent Neural Networks

*Feedforward* neurala nätverk innebär som vi såg att datan går framåt, från input-till-output. Senare i boken, Kapitel 9, kommer vi lära oss om *recurrent neural networks (RNN)* som har *feedback*-loopar där utdata från en neuron vid ett tidsteg matas tillbaka som indata till nätverket vid nästa tidsteg. *RNN*-modeller kan därför användas för att hantera sekventiell data såsom text, tal och tidsserier.



Figur 7.5: Arkitektur för en single-layer-perceptron. Det som karakteriseras är att de är av typen feedforward, är fully connected/dense och har icke-linjära aktiveringsfunktioner.

I koden nedan kommer vi demonstrera hur vi skapar en *Multilayer perceptron (MLP)* genom klassen `MLPRegressor` från *scikit-learn*. För resten av boken kommer vi använda *Keras* och *Tensorflow* för att skapa neurala nätverk eftersom det är mer utvecklat än vad *scikit-learn* är för neurala nätverk.

```

from sklearn.datasets import make_regression
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPRegressor

X, y = make_regression(n_samples=1000, n_features=20,
    ↵ random_state=1)
X_train, X_test, y_train, y_test = train_test_split(X, y,
    ↵ random_state=20)

mlp = MLPRegressor(random_state=5, hidden_layer_sizes=(200, 100,
    ↵ 10), activation='relu')                                ①
mlp.fit(X_train, y_train)

predictions = mlp.predict(X_test[:2])                      ②
print(y_test[:2], predictions)                            ③

# Calculating the number of weights for the model
total_weights = sum(w.size for w in mlp.coefs_)          ④
total_biases = sum(b.size for b in mlp.intercepts_)
total_params = total_weights + total_biases

print("Total weights:", total_weights)
print("Total biases:", total_biases)
print("Total parameters:", total_params)

```

- ① När neurala nätverk tränas så initialiseras vikterna slumpmässigt. Detta gäller generellt. För att få reproducerbara resultat specificerar vi `random_state=5`.
- ② Vi genomför prediktioner med vår tränade modell.
- ③ Vi skriver ut de sanna värdena som vi försökte prediktera bredvid våra faktiska prediktioner så vi kan få en konkret känsla för hur bra modellen presterar. Notera, vi gör det här för demonstrationssyfte, i praktiken hade vi kunnat använda något utvärderingsmått såsom *RMSE*.
- ④ Den efterföljande koden skrivs för att läsaren ska få en känsla för att neurala nätverk generellt sett har väldigt många parametrar vilket alltså innebär att modellerna är mer komplexa.

[214.58831339 367.6099045 ] [207.06313276 363.80322841]  
 Total weights: 25010  
 Total biases: 311

Total parameters: 25321

### i Neurala nätverk har väldigt många parametrar

Antalet parametrar (*weights* och *biases*) i modellen vi använde i koden ovan:

```
X, y = make_regression(n_samples=1000, n_features=20, random_state=1)
MLPRegressor(random_state=5, hidden_layer_sizes=(200, 100, 10),
activation='relu')
```

ligger på hela  $25010 + 311 = 25321$  stycken. Det kan exempelvis jämföras med en linjär regressionsmodell såsom  $y = \theta_1 + \theta_2 x_1$  vilken alltså endast har två parametrar. För att räkna ut hur vi fick 25321 gör vi följande uträkningar:  $(20 \times 200 + 200 \times 100 + 100 \times 10 + 10 \times 1) + (200 + 100 + 10 + 1) = 25010 + 311 = 25321$ . Den första parantesen beräknar antalet vikter mellan noderna och den andra parantesen är vikterna för *bias*-noderna. Notera, *bias*-noderna har inte skrivits ut i koden men de inkluderas alltid, det är alltså implicit att de är inkluderade. För att förstå uträkningen kan du prova räkna manuellt på exempelvis Figur 7.1, vi betonar att i varje lager förutom det sista lagret så ska en bias nod inkluderas även om det varken brukar ritas ut i figurer eller skrivas ut i kod. Det är alltså implicit att de är inkluderade.

Det gäller generellt att ju fler parametrar en modell har, desto mer data behövs för att modellträningen ska bli bra. Därför är det generellt sett så att neurala nätverk behöver mer data för att bli bra tränade jämfört med exempelvis en linjär regressionsmodell. Många parametrar ökar också risken att modellen blir överanpassad/*overfitted* eftersom modellen blir för komplex i förhållande till datans faktiska komplexitet.

## 7.2.2 Riktlinjer för modellarkitektur

När vi arbetar med neurala nätverk är Tabell 7.1 och Tabell 7.2 mycket hjälpsamma då de hjälper oss att skapa en grundmodell som vi sedan kan modifiera. När vi kommer till kodexemplen där vi demonstrerar *Keras*, Avsnitt 7.5, kommer vi även se att vi manuellt behöver specificera vissa val och då är det smidigt att använda tabellerna om vi behöver repetera något. Nedan skriver vi några kommentarer kopplat till tabellerna.

- Antal neuroner i *input/output* lagren styrs av problemet i sig självt. Exempelvis för

*MNIST*-datasetet så hade vi haft 784 *input* noder eftersom bilderna har  $28 \times 28 = 784$  pixlar. Vi hade haft 10 *output* noder eftersom det finns 10 olika klasser (siffror) som ska predikteras. Vi ser att detta står i Tabell 7.1.

- I Ekvation 7.1 såg vi ekvationen för den logistiska aktiveringsfunktionen och i Figur 7.4 såg vi hur funktionen ser ut. Från figuren ser vi att den är ett tal mellan 0 och 1 innebärande att det kan tolkas som en sannolikhet. Om vi exempelvis ska prediktera om någon har en hund eller inte så kanske den beroende variabeln är 1 om man har en hund och 0 om man inte har en hund, det är alltså ett binärt klassificeringsproblem. Om vi då använder den logistiska aktiveringsfunktionen för *output*-lagret och exempelvis får siffran 0.64 betyder det att modellen predikterar att det är 64% sannolikhet att personen har en hund och därmed 36% att personen inte har en hund. På motsvarande sätt, för multiklass-klassificeringsproblem används *softmax* som aktiveringsfunktion för *output*-lagret. Vi går inte in på matematiska detaljer, det räcker att veta att det är en generalisering av den logistiska aktiveringsfunktionen där siffran för varje output blir ett tal mellan 0 och 1 samt att summan av alla siffror blir 1. Det kan alltså återigen tolkas som en sannolikhet. Detta står i Tabell 7.2.
- I de fall vi exempelvis vill prediktera någons ålder så vet vi att man alltid måste vara  $\geq 0$  år gammal. Därför används *ReLU* som aktiveringsfunktion för *output*-lagret eftersom den alltid är större än eller lika med noll. Se Ekvation 7.2 och Figur 7.4. Detta står i Tabell 7.1.
- I Tabell 7.1 ser vi att vi för regressionsproblem kan använda *MSE* eller *MAE* som *loss*-funktion. I Tabell 7.2 ser vi att *cross entropy* används som *loss*-funktion för klassificeringsproblem. I korthet har *loss*-funktionen *cross entropy* egenskapen att den straffar felaktiga prediktioner mer om de görs med hög självskräckhet jämfört med låg självskräckhet. Vi går inte in på matematiska detaljer som den intresserade läsaren själv kan kolla upp.

Avslutningsvis, vill vi automatisera valet av hyperparametrar, exempelvis antalet *hidden layers* kommer vi senare, i Avsnitt 7.5.2, se ett kodexempel där **KerasTuner** används. Det är en motsvarighet till **GridSearch** i *scikit-learn* och används alltså för att optimera hyperparametrar.

Tabell 7.1: Typisk MLP-arkitektur för regressionsproblem.

Hyperparameter	Standardvärde
Antal <i>input</i> neuroner	En per <i>input feature</i> (t.ex. $28 \times 28 = 784$ för <i>MNIST</i> )
Antal <i>hidden layers</i>	Beror på problemet, men vanligtvis 1 till 5
Antal neuroner per <i>hidden layer</i>	Beror på problemet, men vanligtvis 10 till 100
Antal <i>output</i> neuroner	1 per prediktionsdimension
<i>Hidden activation</i>	<i>ReLU</i> (eller <i>SELU</i> )
<i>Output activation</i>	Ingen, eller <i>ReLU</i> (när $y$ är positivt) eller logistisk/tanh (när $y$ är i ett begränsat interval)
<i>Loss-funktion</i>	<i>MSE</i> eller <i>MAE</i> (om det finns outliers)

Tabell 7.2: Typisk MLP-arkitektur för klassificeringsproblem.

Hyperparameter	Binär klassificering	Multilabel binär klassificering	Multiklass klassificering
Antal <i>input neuroner</i> , antal <i>hidden layers</i> och antal neuroner i dessa	Samma som för regression	Samma som för regression	Samma som för regression
Antal <i>output</i> neuroner	1	1 per <i>label</i>	1 per klass
<i>Output layer activation</i>	<i>Logistic</i>	<i>Logistic</i>	<i>Softmax</i>
<i>Loss-funktion</i>	<i>Cross entropy</i>	<i>Cross entropy</i>	<i>Cross entropy</i>

### 7.2.3 Intuition för antal layers och noder

Generellt sett hade vi kunnat modellera komplexa samband med endast ett *hidden layer* som har väldigt många noder. Däremot har det visat sig att det ofta är effektivare med att ha flera dolda lager. Intuitionen är att de första lagren lär sig enklare strukturer medan påföljande lager högre upp kan kombinera dessa enklare strukturer för att modellera mer avancerade strukturer.

I de dolda lagren har man rent historiskt ofta använt en pyramidstruktur för lagren. I kodexemplet ovan såg vi exempelvis att vi körde följande kod:

```
mlp = MLPRegressor(random_state=5, hidden_layer_sizes=(200, 100, 10), activation='relu')
```

där vi alltså använde pyramidstrukturen 200-100-10. Tanken var att det finns färre avancerade strukturer än enkla strukturer och därfor användes fler noder i de lägre dolda lagren. I praktiken har det visat sig att det ofta är minst lika bra att ha lika många neuroner i varje lager vilket idag är vanligt förekommande. När vi arbetar med modellering kan det vara värt att prova båda tillvägagångssätten.

I praktiken är det ofta smidigt att ha många lager och noder för att sedan använda olika regulariseringstekniker som begränsar antalet noder och lager. Intuition är att ”*för stora byxor kan vi alltid sy in, men för små går inte att förstora*”. Vi fortsätter nu med att kolla på regularisering.

### i Tensorflow playground

På följande sida kan läsaren enkelt ”leka” och få en känsla för neurala nätverk. Kolla in den! <https://playground.tensorflow.org/>

## 7.3 Regularisering

I Avsnitt 3.3.3 gick vi igenom *the bias variance trade-off* som förklarade varför vi kan tänkas vilja regularisera/begränsa en modell.

Generellt sett har ANN väldigt många parametrar vilket gör att det ofta finns en risk för att de blir överanpassade. För att hantera detta kan vi tillämpa olika regulariseringstekniker. I detta avsnittet kommer vi kolla på;  $l_1$  och  $l_2$  regularisering, *dropout* samt *early stopping*.

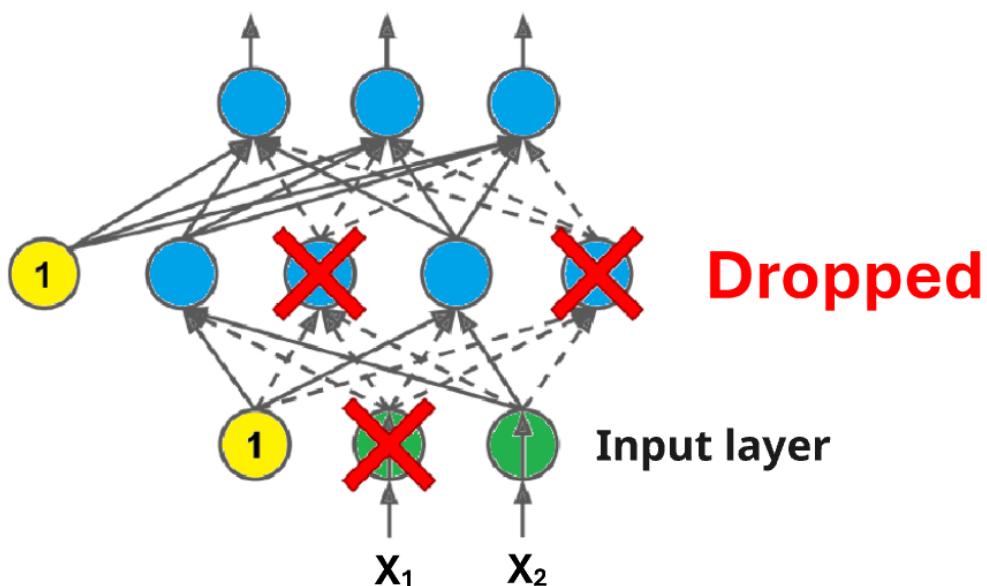
När vi arbetade med linjära regressionsmodeller så fick vi *lasso*-modellen när vi tillämpade  $l_1$  regularisering. Denna regulariseringen ledde till att parametrarna/vikterna ( $\theta$ ) drogs mot 0 där oviktiga vikter sattes till 0. *ridge*-modellen fick vi när vi tillämpade  $l_2$  regularisering. Denna regularisering ledde till att parametrarna/vikterna ( $\theta$ ) drogs mot 0 men blev sällan exakt 0. Samma logik gäller för neurala nätverk och även där kan vi alltså använda  $l_1$  och  $l_2$  regularisering. I kod kan det se ut enligt nedan när vi tillämpar det på ett lager.

```
layer = layers.Dense(units=64,
    ↳ kernel_regularizer=regularizers.L1(l1=0.01))
```

①

①  $l1=0.01$  styr styrkan på regulariseringen.

*Dropout* metodiken är en populär metod för att regularisera neurala nätverk. Metoden går ut på att vid varje träningsiteration så har varje neuron en sannolikhet  $p$  (vi väljer själva  $p$  men omkring 10% – 50% är vanligt) att bli *droppad*. Intuitivt så leder detta till att neuronerna tvingas till att ”lära sig själva” och inte samarbeta eller förlita sig på andra neuroner vilket brukar kallas för *co-adaptions*. Generellt sett gäller det att om vi har en modell som vi bedömer vara överanpassad så höjer vi alltså *dropout* graden. Se Figur 7.6 för en illustration av *dropout*.



Figur 7.6: Med dropout-regularisering så gäller det att för varje tränings-iteration kan en neuron (förutom de i output-lagret) med en sannolikhet  $p$ , som vi själva specificerar, bli ”droppad”. En ”droppad” neuron har en output på 0 och deltar inte i beräkningarna under just den iterationen. I figuren representeras dessa neuroner med streckade linjer.

I koden nedan demonstreras  $l1$  regularisering och *dropout*.

```

import numpy as np
from sklearn.datasets import make_regression
from sklearn.model_selection import train_test_split
from tensorflow import keras
from tensorflow.keras import layers, regularizers ①

# Data
X, y = make_regression(n_samples=1000, n_features=10, noise=0.1)
↳ ②

X_train_full, X_test, y_train_full, y_test = train_test_split(X,
    ↳ y, test_size=0.2, random_state=42)
X_train, X_val, y_train, y_val = train_test_split(X_train_full,
    ↳ y_train_full, test_size=0.25, random_state=42)

# Model with L1 and dropout regularization
model = keras.Sequential([
    keras.Input(shape=(10,)), ③
    layers.Dense(64, activation='relu'), ④
    ↳ kernel_regularizer=regularizers.L1(0.01)), ⑤
    layers.Dropout(rate=0.4), ⑥
    layers.Dense(1) ⑦
])

model.compile(optimizer='adam', loss='mse') ⑧
model.fit(X_train, y_train, epochs=15, batch_size=32,
    ↳ validation_data=(X_val, y_val), verbose=2) ⑨

# Evaluation
loss = model.evaluate(X_test, y_test, verbose=2) ⑩
print("Test MSE:", loss)

```

- ① Importerar de bibliotek vi använder.
- ② Skapar ett dataset som vi använder för demonstrationssyfte.
- ③ Vi använder `Sequential`-klassen som är det vi kommer använda när vi skapar neurala nätverk.
- ④ Vi specificerar att datan vi använder för modellen har tio kolumner. Det går att

köra koden utan denna rad men generellt så gäller det att “*explicit är bättre än implicit*” inom programmering.

- ⑤ Det första dolda lagret har 64 noder, använder `relu` som aktiveringsfunktion och har 11 regularisering.
- ⑥ Vi använder dropout-regularisering med `rate=0.4`.
- ⑦ Datan har en beroende variabel ( $y$ ) och därför har vi en *output*-nod.
- ⑧ Modellen kompileras där optimeringsalgoritmen *adam* (vi kommer komma in på detta i Avsnitt 7.4) och *loss*-funktion specificeras.
- ⑨ Modellen tränas. Senare i kapitlet, Avsnitt 7.4, kommer vi gå igenom vad `epoch` och `batch_size` innebär. `verbose=2` styr hur mycket information som skrivs ut. Prova att sätta `verbose=1` eller `verbose=0` och se vad som händer. Vi väljer `verbose=2` av formateringsskäl i samband med att boken trycks.
- ⑩ Modellen evalueras på testdatan.

```
Epoch 1/15
19/19 - 1s - 40ms/step - loss: 59463.0117 - val_loss: 50812.4258
Epoch 2/15
19/19 - 0s - 5ms/step - loss: 59325.3867 - val_loss: 50697.8516
Epoch 3/15
19/19 - 0s - 5ms/step - loss: 59188.2812 - val_loss: 50573.1602
Epoch 4/15
19/19 - 0s - 5ms/step - loss: 59021.9727 - val_loss: 50431.3633
Epoch 5/15
19/19 - 0s - 6ms/step - loss: 58816.5078 - val_loss: 50260.2812
Epoch 6/15
19/19 - 0s - 6ms/step - loss: 58575.3398 - val_loss: 50053.0508
Epoch 7/15
19/19 - 0s - 5ms/step - loss: 58303.8867 - val_loss: 49812.8008
Epoch 8/15
19/19 - 0s - 6ms/step - loss: 57995.8867 - val_loss: 49524.0195
Epoch 9/15
19/19 - 0s - 6ms/step - loss: 57593.5469 - val_loss: 49184.1953
Epoch 10/15
19/19 - 0s - 6ms/step - loss: 57138.1328 - val_loss: 48796.2461
Epoch 11/15
19/19 - 0s - 6ms/step - loss: 56652.2930 - val_loss: 48337.5703
Epoch 12/15
19/19 - 0s - 6ms/step - loss: 56057.2266 - val_loss: 47835.3047
Epoch 13/15
19/19 - 0s - 6ms/step - loss: 55339.3242 - val_loss: 47262.6094
```

```

Epoch 14/15
19/19 - 0s - 6ms/step - loss: 54608.8477 - val_loss: 46642.5234
Epoch 15/15
19/19 - 0s - 6ms/step - loss: 53861.4062 - val_loss: 45957.5586
7/7 - 0s - 7ms/step - loss: 45465.9258
Test MSE: 45465.92578125

```

*Early stopping* innebär att vi kollar på en modells valideringsfel, det vill säga felet den gör på valideringsdata, och när felet slutar minska i ett visst antal epoker (som vi specificerar med hyperparametern `patience`) så stoppas träningen. Det är en intuitivt rimlig metodik, när valideringsfelet inte förbättras efter ett visst antal epoker så avbryts träningen. En utmaning kan vara att effekten av träningen kommer in först efter ett visst antal epoker, om `patience` då är satt till ett lägre antal epoker kommer vi aldrig upptäcka den effekten. Sammanfattningsvis är det i praktiken en avvägning som måste göras kring hur stor `patience` som ska användas.

I kodexemplet ovan kan vi lägga till nedanstående kod för att implementera *early stopping*.

```

from tensorflow.keras.callbacks import EarlyStopping           ①

early_stop = EarlyStopping(monitor='val_loss', patience=3,
    ↵ restore_best_weights=True)                                ②

model.fit(X_train, y_train, epochs=1000, batch_size=32,
    ↵ validation_data=(X_val, y_val), callbacks=[early_stop],
    ↵ verbose=2)                                              ③

```

- ① Importera biblioteket.
- ② Efter kodraden `model.compile(optimizer='adam', loss='mse') # <8>` i kodexemplet ovan lägger vi till denna raden kod.
- ③ Koden `model.fit(X_train, y_train, epochs=15, batch_size=32, validation_data=(X_val, y_val), verbose=2) # <9>` ersätts med denna koden. Vi ändrade till `epochs=1000` så effekten av *early stopping* syns, kör koden så märker du varför. Vi lade även till `callbacks=[early_stop]`.

### **i** Batch Normalization layers

*Batch Normalization layers* är något som empiriskt visat sig ge goda resultat. Modellerna kan få bättre prediktionsförmåga, träningen kan gå snabbare (färre epoker kan behövas för att träna modellen) och bli mer stabil. Utan att gå in i detaljer så uppnås detta genom att man standardiseringar datan. Implementerat i kod kan det rent schematiskt se ut enligt nedan.

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(300, activation="relu"),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(100, activation="relu"),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(10, activation="softmax")
])
```

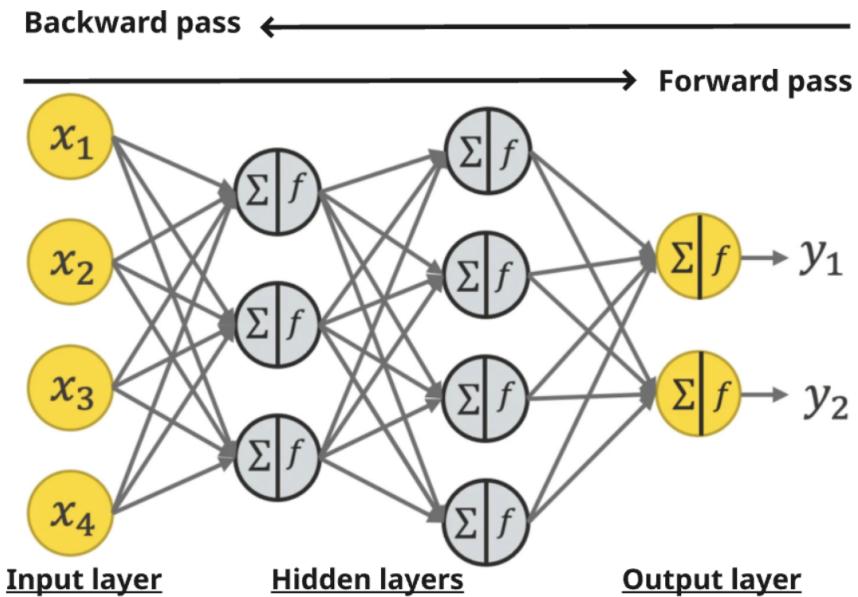
## 7.4 Träning av neurala nätverk - *Backpropagation*

*Backpropagation* är en algoritm som används för att träna neurala nätverk och den kan sägas bestå av följande fyra steg:

1. Algoritmen hanterar en *batch* av data (t.ex. 32 observationer) i taget och hela träningsdata (t.ex. 40000 observationer) går igenom flera gånger där varje genomgång av datan kallas för *epoch* på engelska).
2. Varje *batch* av *input*-datan flödar framåt genom nätverket och man gör slutligen en prediktion på datan som stoppades in. Detta kallas för *forward pass*.
3. Vi mäter felet (*loss*) genom att jämföra sanna värden med predikterade värden via vår *loss*-funktion (t.ex. *MSE* för regressionsproblem).
4. Därefter går algoritmen tillbaka genom nätverket och beräknar hur mycket varje vikt har bidragit till felet. Detta kallas för *backward pass*.
5. *Gradient Descent* tillämpas för att justera vikterna så felet minskar. *Gradient Descent* gicks igenom i Avsnitt 3.3.2.

Se Figur 7.7 för en visualisering av *Backpropagation*.

Exekverar vi koden nedan kommer vi få resultat som ser ut enligt Figur 7.8. Notera,



Figur 7.7: Backpropagation-algoritmen tar en batch av input-data och skjuter den framåt genom nätverket för att skapa prediktioner och beräkna loss. Detta benämns forward pass. Därefter går algoritmen tillbaka genom nätverket och mäter hur mycket varje vikt har bidragit till felet (loss). Detta benämns backward pass.

siffrorna blir annorlunda varje gång koden körs om. Kollar vi på figuren ser vi att totalt har 10 stycken epoker körts. Detta eftersom i koden `model.fit(X_train, y_train, epochs=10, batch_size=32, validation_data=(X_test, y_test), verbose=1)` specificerade vi hyperparametern `epochs=10`. Det innebär att datan går igenom 10 gånger. Vårt dataset har totalt 50000 stycken observationer men vi lade undan 20% i testdatan innebärande att träningsdelen har 40000 stycken observationer. I koden specificerade vi `batch_size=32` innebärande att 32 stycken observationer hanteras i taget. Det innebär att  $40000/32 = 1250$  stycken iterationer går igenom per epok. Det är vad 1250/1250 representerar i figuren.

```

import numpy as np
from sklearn.datasets import make_regression
from sklearn.model_selection import train_test_split
from tensorflow import keras
from tensorflow.keras import layers, regularizers

# Data
X, y = make_regression(n_samples=50000, n_features=10, noise=0.1)
X_train, X_test, y_train, y_test = train_test_split(X, y,
    ↵ test_size=0.2)

# Modell
model = keras.Sequential([
    keras.Input(shape=(10,)),
    layers.Dense(64, activation='relu'),
    layers.Dense(1)
])

model.compile(optimizer='adam', loss='mse')
model.fit(X_train, y_train, epochs=10, batch_size=32,
    ↵ validation_data=(X_test, y_test), verbose=1) ①

```

- ① Som tumregel används *batch sizes* på 32 stycken eller 64 stycken observationer. Har man tillgång till GPU kan *batch sizes* på 128 observationer eller 264 observationer skynda på träningen. Notera, detta är endast tumregler men är det som man oftast ser i kodexempel.

```
Epoch 1/10
1250/1250 - 4s - 3ms/step - loss: 13332.8877 - val_loss: 972.3654
Epoch 2/10
1250/1250 - 3s - 2ms/step - loss: 526.0670 - val_loss: 421.4492
Epoch 3/10
1250/1250 - 3s - 3ms/step - loss: 319.3844 - val_loss: 226.4935
Epoch 4/10
1250/1250 - 2s - 2ms/step - loss: 142.5712 - val_loss: 76.9230
Epoch 5/10
1250/1250 - 3s - 2ms/step - loss: 36.5343 - val_loss: 11.8799
Epoch 6/10
1250/1250 - 3s - 2ms/step - loss: 5.2890 - val_loss: 2.5650
Epoch 7/10
1250/1250 - 3s - 2ms/step - loss: 2.0627 - val_loss: 1.6714
Epoch 8/10
1250/1250 - 3s - 2ms/step - loss: 1.4446 - val_loss: 1.2140
Epoch 9/10
1250/1250 - 2s - 2ms/step - loss: 1.0167 - val_loss: 0.8485
Epoch 10/10
1250/1250 - 4s - 3ms/step - loss: 0.6909 - val_loss: 0.6165
```

Figur 7.8: Resultat som syns när vi exekverar koden `model.fit(X_train, y_train, epochs=10, batch_size=32, validation_data=(X_test, y_test), verbose=1)` i kodexemplet ovan.

Att träna neurala nätverk kan ta lång tid. Ett sätt att påskynda processen och möjligtvis få bättre resultat är att använda andra optimeringsalgoritmer än ”vanlig” *gradient descent* för att justera vikterna i *backpropagation*. När vi jobbar med neurala nätverk är det vanligt förekommande att *RMSprop*, *Adam* och *Nadam* används. Som standardvärde (*default*) används *RMSprop* i *Keras*, i kodexemplet ovan använde vi oss av `optimizer='adam'`. Hur dessa optimeringsalgoritmer fungerar rent matematiskt är inget vi går djupare in på i denna bok då vi inte behöver det när vi praktiskt arbetar med koden.

## 7.5 Två kodexempel

I detta avsnitt kommer vi kolla på två kodexempel. I det första genomförs en klassificering av bilder och i det andra demonstreras hur optimering av hyperparametrar kan göras med hjälp av *KerasTuner*.

### 7.5.1 Kodexempel 1 - Bildklassificering av *Fashion MNIST*

I detta kodexempel kommer vi arbeta med *Fashion MNIST*. Datasetet består av 70000 stycken bilder där varje bild är i gråskala och har  $28 \times 28$  pixlar. Totalt finns det tio stycken klasser i datan där varje klass representerar ett modeplagg. Eftersom bilderna representerar ett modeplagg snarare än en handskriven siffra är datan mer komplex och svårare att prediktera än vanliga *MNIST*.

Vi börjar exemplet med att importera biblioteken för att därefter ladda in datan.

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import keras
from keras.models import Sequential
from keras.layers import Dense, Flatten, Input
from tensorflow.keras.callbacks import EarlyStopping

fashion_mnist = keras.datasets.fashion_mnist
(X_train_full, y_train_full), (X_test, y_test) =
    fashion_mnist.load_data()
```

```

print("X_train_full: ", X_train_full.shape)
print("X_test: ", X_test.shape)

X_train, X_valid = X_train_full[5000:] / 255.0,
↪ X_train_full[:5000] / 255.0
y_train, y_valid = y_train_full[5000:], y_train_full[:5000]
X_test = X_test/255.0                                ①

print("X_train: ", X_train.shape)
print("X_valid: ", X_valid.shape)

```

- ① Träningsdatan kommer bestå av 55000 bilder och valideringsdatan kommer bestå av 5000 bilder. Vi standardiseringar även pixlarna till att vara i intervallet 0-1, detta uppnås genom att vi dividerar med 255.0.

```

X_train_full: (60000, 28, 28)
X_test: (10000, 28, 28)
X_train: (55000, 28, 28)
X_valid: (5000, 28, 28)

```

Från resultatet ser vi att vår data har tre dimensioner, exempelvis (5000, 28, 28) betyder att det finns 5000 bilder där varje bild har storleken  $(28 \times 28)$ .

Nedan gör vi en visualisering över hur bilderna kan se ut.

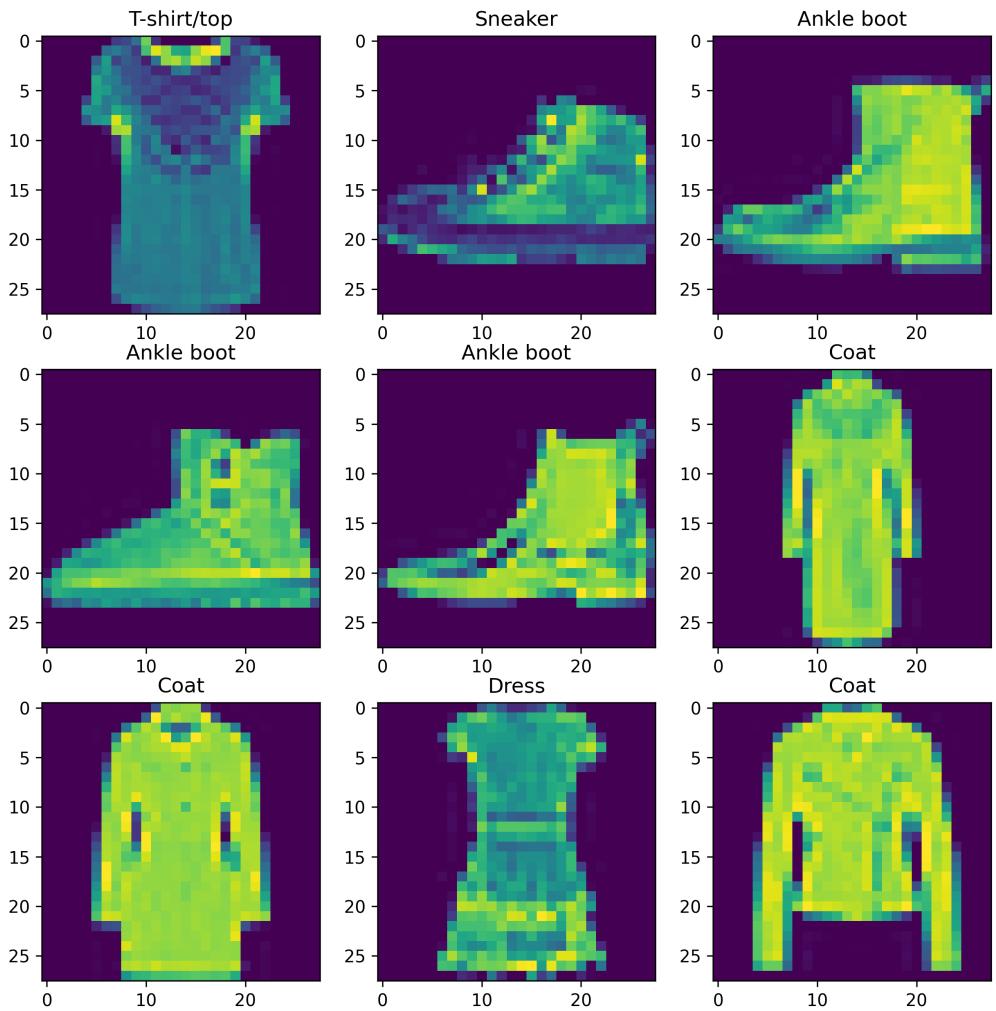
```

class_names = ["T-shirt/top", "Trouser", "Pullover", "Dress",
↪ "Coat", "Sandal", "Shirt", "Sneaker", "Bag", "Ankle boot"] ①

plt.figure(figsize=(10, 10))
for i in range(1, 10, 1):
    plt.subplot(3, 3, i)
    plt.imshow(X_train[i])
    plt.title(class_names[y_train[i]])
plt.show()

```

- ① Klassnamnen har vi hämtat från dokumentationen:  
[https://keras.io/api/datasets/fashion\\_mnist/](https://keras.io/api/datasets/fashion_mnist/). Dessa använder vi i vår visualisering.



Nedan specificerar vi modellen som vi kommer använda.

```
model = Sequential()  
model.add(Input(shape=(28, 28, 1)))  
model.add(Flatten())
```

①

```

model.add(Dense(300, activation="relu"))
model.add(Dense(100, activation="relu"))
model.add(Dense(10, activation="softmax"))          ②

model.summary()                                    ③

```

- ① Genom att använda `model.add(Flatten())` gör vi vår data till 1D vilket är vad modellen förväntar sig i detta fallet.
- ② Vi specificerar `activation="softmax"` innebärande att vår modell kommer prediktera sannolikheter för respektive klass.
- ③ Vi tar fram en summering för modellen. Från resultatet ser vi det faktum att neurala nätverk har väldigt många parametrar. Exempelvis, siffran 235500 får vi genom uträkningen:  $(784 \times 300) + 300 = 235500$ .  $(784 \times 300)$  representerar vikterna mellan noderna och vi får ytterligare 300 stycken eftersom *bias*-noderna alltid är med. Notera att vi börjar med 784 eftersom våra bilder hade storleken  $28 \times 28 = 784$  och vi använder `model.add(Flatten())`.

Model: "sequential\_1"

Layer (type)	Output Shape	Param #
flatten (Flatten)	(None, 784)	0
dense_2 (Dense)	(None, 300)	235,500
dense_3 (Dense)	(None, 100)	30,100
dense_4 (Dense)	(None, 10)	1,010

Total params: 266,610 (1.02 MB)

Trainable params: 266,610 (1.02 MB)

Non-trainable params: 0 (0.00 B)

```

model.compile(loss="sparse_categorical_crossentropy",
    ↵ metrics=["accuracy"])                                ①

# early_stop = EarlyStopping(monitor='val_loss', patience=5,
    ↵ restore_best_weights=True)                          ②
# history = model.fit(X_train, y_train, epochs=60,
    ↵ validation_data=(X_valid, y_valid), callbacks=[early_stop],
    ↵ verbose=2)                                     ③
history = model.fit(X_train, y_train, epochs=20,
    ↵ validation_data=(X_valid, y_valid), verbose=2)      ④

```

- ① Om våra  $y$  är *one-hot-encoded* så använder vi `loss="categorical_crossentropy"`, om de inte är det använder vi `loss="sparse_categorical_crossentropy"`. För att exemplifiera, om vi antar att vi har tre klasser och en observation tillhör den tredje klassen hade det i *one-hot-encoded* format sett ut enligt följande: `y_true = [0, 0, 1]`. Om det inte är *one-hot-encoded* så hade det helt enkelt sett ut enligt följande: `y_true = 3`.
- ② De kodraderna som är annoterade med “2” och “3” kan kommenteras in om kodraden med annoteringen “4” kommenteras bort. Vi har här i boken kört koden med annotering “4” av utrymmesskäl.

```

Epoch 1/20
1719/1719 - 6s - 3ms/step - accuracy: 0.8175 - loss: 0.5034 -
val_accuracy: 0.8584 - val_loss: 0.4107
Epoch 2/20
1719/1719 - 6s - 4ms/step - accuracy: 0.8626 - loss: 0.3827 -
val_accuracy: 0.8766 - val_loss: 0.3557
Epoch 3/20
1719/1719 - 6s - 3ms/step - accuracy: 0.8721 - loss: 0.3570 -
val_accuracy: 0.8736 - val_loss: 0.3965
Epoch 4/20
1719/1719 - 6s - 4ms/step - accuracy: 0.8795 - loss: 0.3445 -
val_accuracy: 0.8696 - val_loss: 0.4026
Epoch 5/20
1719/1719 - 6s - 3ms/step - accuracy: 0.8812 - loss: 0.3365 -
val_accuracy: 0.8764 - val_loss: 0.3611
Epoch 6/20
1719/1719 - 5s - 3ms/step - accuracy: 0.8856 - loss: 0.3264 -
val_accuracy: 0.8664 - val_loss: 0.5347

```

Epoch 7/20  
1719/1719 - 6s - 3ms/step - accuracy: 0.8890 - loss: 0.3178 -  
val\_accuracy: 0.8724 - val\_loss: 0.4268  
Epoch 8/20  
1719/1719 - 5s - 3ms/step - accuracy: 0.8910 - loss: 0.3149 -  
val\_accuracy: 0.8816 - val\_loss: 0.4173  
Epoch 9/20  
1719/1719 - 6s - 3ms/step - accuracy: 0.8938 - loss: 0.3109 -  
val\_accuracy: 0.8784 - val\_loss: 0.4332  
Epoch 10/20  
1719/1719 - 6s - 3ms/step - accuracy: 0.8953 - loss: 0.3038 -  
val\_accuracy: 0.8822 - val\_loss: 0.4468  
Epoch 11/20  
1719/1719 - 4s - 2ms/step - accuracy: 0.8969 - loss: 0.3024 -  
val\_accuracy: 0.8882 - val\_loss: 0.4164  
Epoch 12/20  
1719/1719 - 4s - 2ms/step - accuracy: 0.8999 - loss: 0.2953 -  
val\_accuracy: 0.8860 - val\_loss: 0.4292  
Epoch 13/20  
1719/1719 - 4s - 3ms/step - accuracy: 0.9013 - loss: 0.2911 -  
val\_accuracy: 0.8888 - val\_loss: 0.4765  
Epoch 14/20  
1719/1719 - 5s - 3ms/step - accuracy: 0.9035 - loss: 0.2845 -  
val\_accuracy: 0.8826 - val\_loss: 0.4926  
Epoch 15/20  
1719/1719 - 6s - 3ms/step - accuracy: 0.9058 - loss: 0.2835 -  
val\_accuracy: 0.8862 - val\_loss: 0.4548  
Epoch 16/20  
1719/1719 - 6s - 3ms/step - accuracy: 0.9047 - loss: 0.2833 -  
val\_accuracy: 0.8782 - val\_loss: 0.5253  
Epoch 17/20  
1719/1719 - 9s - 5ms/step - accuracy: 0.9061 - loss: 0.2758 -  
val\_accuracy: 0.8868 - val\_loss: 0.4878  
Epoch 18/20  
1719/1719 - 6s - 3ms/step - accuracy: 0.9079 - loss: 0.2720 -  
val\_accuracy: 0.8888 - val\_loss: 0.5110  
Epoch 19/20  
1719/1719 - 7s - 4ms/step - accuracy: 0.9103 - loss: 0.2726 -  
val\_accuracy: 0.8890 - val\_loss: 0.4690

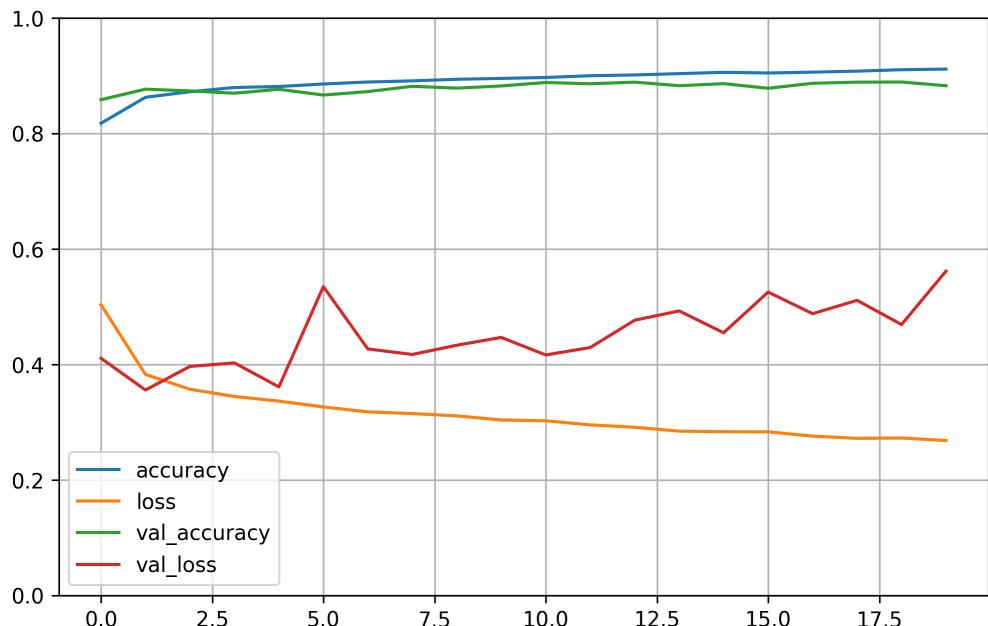
Epoch 20/20

1719/1719 - 6s - 3ms/step - accuracy: 0.9114 - loss: 0.2682 -  
val\_accuracy: 0.8826 - val\_loss: 0.5616

.fit() metoden returnerar ett History objekt som vi använder i koden nedan. I dokumentationen från Keras står följande:

A History object. Its History.history attribute is a record of training loss values and metrics values at successive epochs, as well as validation loss values and validation metrics values (if applicable).

```
pd.DataFrame(history.history).plot(figsize=(8, 5))
plt.grid(True)
plt.gca().set_ylim(0, 1)
plt.show()
```



I visualiseringen ser vi att *accuracy* för träningsdatan och valideringsdatan stadigt ökar

för varje epok under träningen samtidigt som *loss* på träningsdatan och valideringsdatan sjunker. Detta är bra. Vi ser även att kurvorna för träningsdatan och valideringsdatan är nära varandra vilket tyder på att modellen inte är överanpassad. Det är också bra!

Vi utvärderar/evaluerar vår modell på testdatan nedan.

```
model.evaluate(X_test, y_test, verbose=2)
```

```
313/313 - 1s - 2ms/step - accuracy: 0.8780 - loss: 0.5826  
[0.5825849175453186, 0.878000020980835]
```

Slutligen använder vi vår modell för att prediktera nya observationer. Eftersom vi inte har riktiga, nya observationer, tar vi bara några från testdatan och låtsas att de är nya. Läsaren uppmuntras till att försöka ta egna bilder, bearbeta dem så de har samma format som i *Fashion MNIST* och försöka prediktera dessa. Det är en mycket bra övning.

```
X_new = X_test[:3]  
y_proba = model.predict(X_new, verbose=2)  
y_proba.round(2)  
  
classes_x = np.argmax(y_proba, axis=1) ①  
print("Predicted class names: ", class_names[classes_x[0]], " ",  
     ↪ class_names[classes_x[1]], " ", class_names[classes_x[2]])  
y_new = y_test[:3]  
print("True class names: ", class_names[y_new[0]], " ",  
     ↪ class_names[y_new[1]], " ", class_names[y_new[2]])
```

- ① Vår modell predikterar sannolikheter och genom att använda `np.argmax` blir vår slutgiltiga prediktion den klass som har högst sannolikhet.

```
1/1 - 0s - 79ms/step  
Predicted class names: Ankle boot Pullover Trouser  
True class names: Ankle boot Pullover Trouser
```

Vi har nu kommit till slutet av kodexemplet. Det är en bra övning att också prova att skapa en modell från tidigare del i boken, exempelvis ett beslutsträd eller logistisk regression, för att se hur dessa modeller presterar på *Fashion MNIST*.

### 7.5.2 Kodexempel 2 - Optimering av hyperparametrar med *KerasTuner*

I detta kodexempel demonstreras hur optimering av hyperparametrar med *KerasTuner* kan se ut. Vi ger koden så att läsaren själv kan gå igenom den och läsa dokumentationen.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from sklearn.datasets import make_moons
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

import tensorflow as tf
from tensorflow import keras

from tensorflow.keras.layers import Dense, Input
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dropout
from tensorflow.keras.callbacks import EarlyStopping

from keras_tuner import RandomSearch

# Generating a dataset
X, y = make_moons(n_samples=10000, noise=0.4, random_state=42)

plt.scatter(X[:, 0], X[:, 1], c=y)
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.title('Data generated by make_moons() function')

# Split data into train, val and test
X_train_val, X_test, y_train_val, y_test = train_test_split(X, y,
    ↵ test_size=0.2, random_state=1)

X_train, X_val, y_train, y_val = train_test_split(X_train_val,
    ↵ y_train_val, test_size=0.2, random_state=1)
```

```

# Choosing hyperparameters with KerasTuner
def build_model(hp):
    n_cols = X_train.shape[1]
    nn_model = keras.Sequential()

    nn_model.add(Input(shape=(n_cols,)))

    # Tune the number of units in the first dense layer
    hp_units_1 = hp.Int('units_1', min_value=30, max_value=200,
    ↵ step=10)
    nn_model.add(Dense(units=hp_units_1, activation='relu'))

    # Tune whether to use dropout
    if hp.Boolean("dropout_1"):
        # Tune the dropout rate
        hp_dropout_rate = hp.Choice('dropout_rate_1', values=[i *
    ↵ 0.05 for i in range(12)])
        nn_model.add(Dropout(rate=hp_dropout_rate))

    # Tune the number of units in the second dense layer
    hp_units_2 = hp.Int('units_2', min_value=10, max_value=220,
    ↵ step=10)
    nn_model.add(Dense(units=hp_units_2, activation='relu'))

    # Tune whether to use dropout
    if hp.Boolean("dropout_2"):
        # Tune the dropout rate
        hp_dropout_rate = hp.Choice('dropout_rate_2', values=[0.0,
    ↵ 0.1, 0.2, 0.3, 0.4, 0.5])
        nn_model.add(Dropout(rate=hp_dropout_rate))

    # Tune number of layers
    for i in range(hp.Int("num_layers", 1, 3)):
        nn_model.add(Dense(
            # Tune number of units separately.
            units=hp.Int(f"units_{i+2}", min_value=10,
    ↵ max_value=150, step=5),

```

```

        activation='relu'))

nn_model.add(Dense(1, activation='sigmoid'))

nn_model.compile(optimizer='adam',
                  loss='binary_crossentropy',
                  metrics=['accuracy'])

return nn_model

tuner = RandomSearch(
    build_model,
    objective='val_accuracy',
    max_trials=10,
    overwrite=True,
)

early_stopping_monitor = EarlyStopping(patience = 2)

tuner.search(X_train, y_train, validation_split=0.2, epochs=3,
              callbacks=[early_stopping_monitor], verbose=2)

# tuner.results_summary() ①

# Get the top 2 models
tuned_nn = tuner.get_best_models(num_models=2)[0]
tuned_nn.build()

tuned_nn.fit(X_train, y_train, validation_split=0.2, epochs=100,
              callbacks=[early_stopping_monitor], verbose=2)

# Evaluating our tuned Neural Network
tuned_nn_pred_val = tuned_nn.predict(X_val, verbose=2)
print(tuned_nn_pred_val)

threshold = 0.5
tuned_nn_pred_val_label = np.where(tuned_nn_pred_val > threshold,
                                    1,0)

```

```
print(tuned_nn_pred_val_label)

accuracy_score(y_val, tuned_nn_pred_val_label)
```

- ① Koden kommenterades bort så det inte blir så många utskrifter i boken. När läsaren exekverar koden, ta bort kommentaren så kodraden exekveras.

```
Trial 10 Complete [00h 00m 02s]
val_accuracy: 0.8500000238418579
```

```
Best val_accuracy So Far: 0.850781261920929
Total elapsed time: 00h 00m 31s
```

```
Epoch 1/100
```

```
160/160 - 2s - 13ms/step - accuracy: 0.8693 - loss: 0.3211 -
val_accuracy: 0.8516 - val_loss: 0.3486
```

```
Epoch 2/100
```

```
160/160 - 0s - 2ms/step - accuracy: 0.8709 - loss: 0.3186 -
val_accuracy: 0.8492 - val_loss: 0.3428
```

```
Epoch 3/100
```

```
160/160 - 0s - 2ms/step - accuracy: 0.8699 - loss: 0.3168 -
val_accuracy: 0.8461 - val_loss: 0.3499
```

```
Epoch 4/100
```

```
160/160 - 0s - 2ms/step - accuracy: 0.8711 - loss: 0.3122 -
val_accuracy: 0.8469 - val_loss: 0.3455
```

```
50/50 - 0s - 2ms/step
```

```
[[0.24335721]
```

```
[0.09192536]
```

```
[0.6161302 ]
```

```
...
```

```
[0.02156992]
```

```
[0.9800086 ]
```

```
[0.642472 ]]
```

```
[[0]
```

```
[0]
```

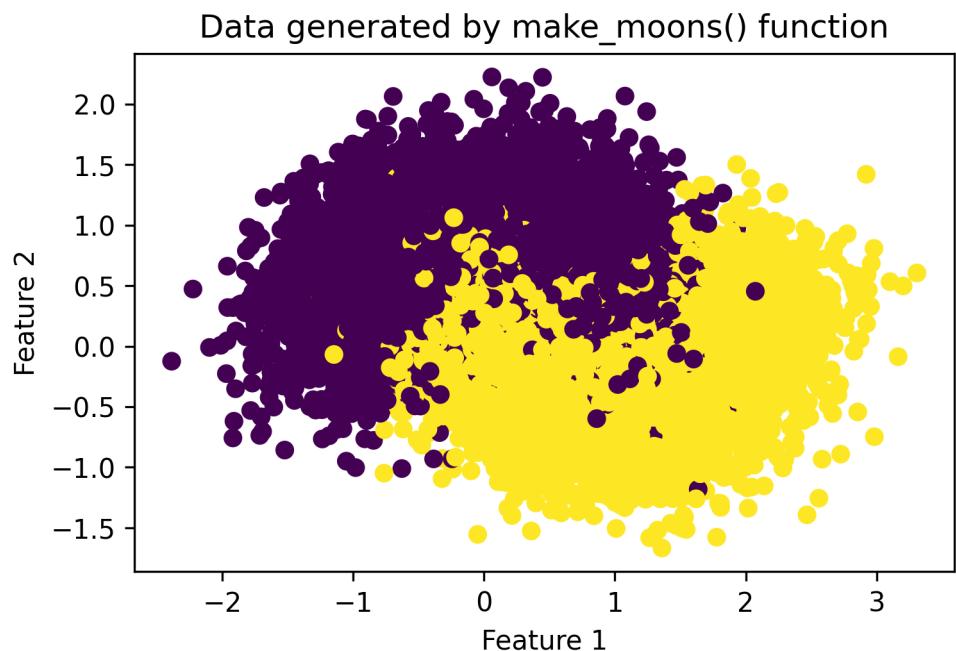
```
[1]
```

```
...
```

```
[0]
```

```
[1]
```

```
[1]  
0.84625
```



## 7.6 Övningsuppgifter

Övningsuppgifter för kapitlet finns på bokens hemsida:  
[https://github.com/AntonioPrgomet/ai\\_tillaempad\\_ml](https://github.com/AntonioPrgomet/ai_tillaempad_ml)

# Kapitel 8

## *Convolutional neural network (CNN)*

I detta kapitel kommer vi lära oss om *Convolutional Neural Networks (CNN)*. Kapitlet inleder med att ge en bakgrund följt av en genomgång för hur *CNN* fungerar. Därefter kollar vi på hur en modellarkitektur kan se ut och hur mer träningsdata kan skapas genom en teknik som kallas för *data augmentation*. Kapitlet avslutas med tre kodexempel där vi bland annat kommer lära oss om förtränaade modeller och *transfer learning*, något som kan vara mycket användbart.

### 8.1 Bakgrund

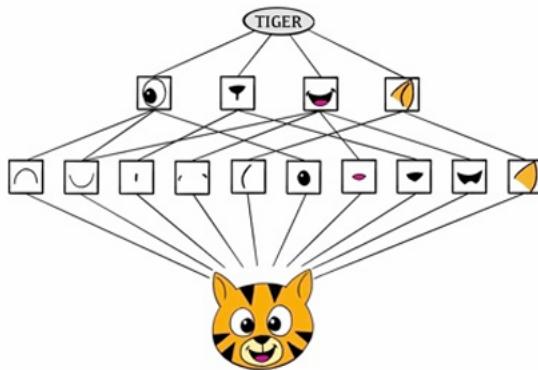
*CNN* är en klass av artificiella neurala nätverk (ANN) vilket innebär att allting som vi lärde oss i föregående kapitel har vi nyttta av även här. För att ett neutralt nätverk ska klassas som *CNN* behöver minst ett av de dolda lagren vara ett *convolution layer*. Vad det exakt innebär kommer vi gå igenom i Avsnitt 8.2.1.

*CNN* inspirerades av den del av hjärnan som hanterar visuell information. Rent generellt är *CNN* väldigt kraftfulla inom det område som kallas för *computer vision* eller datorseende på svenska. Exempel på tillämpningsområden inom datorseende är klassificering där man till exempel hade kunnat klassificera om det är en tiger på en bild. Som tumregel kan vi säga att i de fall vi arbetar med bilder eller videor så är *CNN* den modell vi börjar med att använda då de generellt sett presterar väldigt bra.

I avsnittet som följer ska vi nu gå in på hur *CNN* fungerar rent praktiskt.

## 8.2 Hur fungerar *CNN*?

I Figur 8.1 ser vi hur ett *CNN* fungerar. Vad modellen gör är att den först identifierar *low-level features* som enklare färger och former. Dessa mer enkla egenskaper kombineras därefter för att identifiera *high-level features* såsom ögon, mun, öron och dylikt.



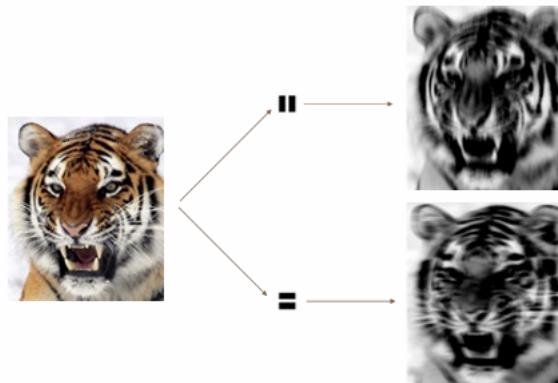
Figur 8.1: CNN identifierar först low-level features. Dessa används därefter för att identifiera high-level features. Bilden är tagen från An Introduction to Statistical Learning with Applications in R (2:a upplagan) av James et al. (2023). <https://www.statlearning.com/>.

Hur går detta till rent praktiskt? Modellen använder det som kallas för *convolutional layers*, ofta i kombination med *pooling layers*. Vi kollar på dessa två typer av lager härnäst.

### 8.2.1 *Convolution layers*

Ett *convolutional layer* består av flera filter där varje filter ”söker efter”/ betonar vissa lokala attribut/egenskaper. I Figur 8.2 ser vi till vänster originalbilden på en tiger. Den övre bilden till höger visar en tiger där de vertikala attributen betonas medan den nedre bilden till höger betonar de horisontella attributen.

I praktiken används flera filter för att hitta flera olika attribut och när det neurala nätverket tränas så lär det sig vilka filter som är bra att använda. När vi säger att



Figur 8.2: Bilden på tigern till vänster är ursprungsbilden. Den övre bilden till höger framhäver tigerns vertikala drag medan den nedre bilden till höger framhäver tigerns horisontella drag. Bilden är tagen från <https://www.statlearning.com/>.

det neurala nätverket lär sig vilka filter som ska användas menar vi egentligen att det neurala nätverket under träningen lär sig vikter som leder till att olika filter tillämpas.

Vi förklarar nu i mer tekniska termer hur filtreringen går till. Antag att vi har en  $(4 \times 3)$  bild enligt nedan, *original image*, där bokstäverna (a, b, c, ...) alltså representerar pixelvärdet.

$$\text{Original image} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \\ j & k & l \end{bmatrix}.$$

Antag därefter att vi har ett  $(2 \times 2)$  filter, *convolution filter* enligt nedan:

$$\text{Convolution filter} = \begin{bmatrix} \alpha & \beta \\ \gamma & \delta \end{bmatrix}.$$

Filtret tillämpas på varje delmatris av storlek  $(2 \times 2)$  i *original image* där den operation som genomförs benämns *convolution*. Vi visar nedan hur det går till för den första delmatrisen:

$$\begin{bmatrix} a & b \\ d & e \end{bmatrix} * \begin{bmatrix} \alpha & \beta \\ \gamma & \delta \end{bmatrix} = a\alpha + b\beta + d\gamma + e\delta.$$

För den andra delbilden hade vi fått:

$$\begin{bmatrix} d & e \\ g & h \end{bmatrix} * \begin{bmatrix} \alpha & \beta \\ \gamma & \delta \end{bmatrix} = d\alpha + e\beta + g\gamma + h\delta.$$

Så fortsätter det för varje delbild och slutligen får vi resultatet *convolved image* enligt nedan vilket har dimensionen  $3 \times 2$ . Notera, en summa av tal är fortfarande *ett* tal. Operationerna som genomfördes för att vi skulle få *convolved image* benämns, som tidigare nämnt, *convolution*.

$$\text{Convolved image} = \begin{bmatrix} a\alpha + b\beta + d\gamma + e\delta & b\alpha + c\beta + e\gamma + f\delta \\ d\alpha + e\beta + g\gamma + h\delta & e\alpha + f\beta + h\gamma + i\delta \\ g\alpha + h\beta + j\gamma + k\delta & h\alpha + i\beta + k\gamma + l\delta \end{bmatrix}.$$

Det går att matematiskt bevisa att om en delmatris i originalbilden liknar filtret så kommer vi få ett högt värde, annars ett litet värde i resultatet där vi med resultatet menar siffrorna i *convolved image*, exempelvis  $a\alpha + b\beta + d\gamma + e\delta$ . Således, *convolved image* fokuserar på de delar i en bild som liknar filtret som tillämpas.

För filtren i Figur 8.2 hade det vertikala filtret exempelvis kunnat se ut enligt följande:

$$\begin{bmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

och det horisontella filtret hade kunnat se ut enligt följande:

$$\begin{bmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix}.$$

Senare när vi applicerar *CNN*-modeller kommer vi ha kod som ser ut enligt nedan:

```
model.add(Conv2D(64, kernel_size=(2, 2), padding='same',
    activation='relu'))
```

①

- ① Vad detta lagret gör är att det specificerar att 64 olika filter ska användas där varje filter har storleken  $(2 \times 2)$ . I det tidigare exemplet med tigern, använde vi till exempel två filter, ett vertikalt och ett horisontellt som båda hade storleken  $(3 \times 3)$  i den exemplifiering vi gav. Vad `padding='same'` betyder går vi igenom i Avsnitt 8.2.3.

Härnäst kollar vi på *pooling layers* som är en typ av lager som ofta används i kombination med *convolutional layers*.

### 8.2.2 Pooling layers

*Pooling layers* fokuserar på de viktigaste delarna i en bild. Resultatet blir att bildens storlek minskar. Det finns olika varianter av *pooling layers* och den variant vi kommer använda heter *max pooling layer*.

*Max pooling layer* tar maximum värdet för varje icke-overlappande delmatris. Om vi använder ett  $(2 \times 2)$  *max pooling layer* tas alltså maximumvärdet för varje icke-overlappande  $(2 \times 2)$  delmatris. Detta exemplifieras nedan.

$$\text{Max pool} \quad \begin{bmatrix} 1 & 2 & 5 & 3 \\ 3 & 0 & 1 & 2 \\ 2 & 1 & 3 & 4 \\ 1 & 1 & 2 & 0 \end{bmatrix} \quad \longrightarrow \quad \begin{bmatrix} 3 & 5 \\ 2 & 4 \end{bmatrix}.$$

Senare när vi applicerar *CNN*-modeller kommer vi ha kod som ser ut enligt nedan:

```
model.add(MaxPooling2D(pool_size=(2, 2)))
```

### 8.2.3 Padding

När vi använder *convolution layers* eller *max pooling layers* så kallas delmatriser igenom och en viss operation utförs. I exemplet med *max pooling* gick det jämnt ut men det är alltså inte säkert att det går jämnt ut. Hade vi exempelvis haft det enligt nedan så går det inte jämnt ut.

$$\text{Max pool} \quad \begin{bmatrix} 1 & 2 & 5 & 3 \\ 3 & 0 & 1 & 2 \\ 2 & 1 & 3 & 4 \end{bmatrix}$$

För att lösa det hade vi helt enkelt kunnat lägga till nollor vilket då är en typ av *padding* som kallas för *zero-padding*. Se Figur 8.3 nedan.

Senare när vi applicerar *CNN*-modeller kommer vi ha kod som ser ut enligt nedan:

$$\text{Max pool} \quad \begin{array}{c|cc|cc} 1 & 2 & 5 & 3 \\ 3 & 0 & 1 & 2 \\ \hline 2 & 1 & 3 & 4 \\ 0 & 0 & 0 & 0 \end{array} \quad \rightarrow \quad \begin{bmatrix} 3 & 5 \\ 2 & 4 \end{bmatrix}$$

Figur 8.3: Exempel på zero-padding där fyra nollor lagts till i den nedersta raden.

```
model.add(Conv2D(64, kernel_size=(3, 3), padding='same',
    activation='relu'))
```

①

- ① Vi ser att vi specificerat `padding='same'`.

Läser vi dokumentationen:

[https://keras.io/api/layers/convolution\\_layers/convolution2d/](https://keras.io/api/layers/convolution_layers/convolution2d/)  
står det följande:

`padding`: string, either "valid" or "same" (case-insensitive). "valid" means no padding. "same" results in padding evenly to the left/right or up/down of the input. When `padding="same"` and `strides=1`, the output has the same size as the input.

Att vi valt `padding='same'` kommer alltså leda till att bilderna behåller samma storlek genom *zero-padding* eftersom `strides=1` är ett standardvärde, det vill säga det gäller om inget annat skrivs. Mer precist är standardvärdet `strides=(1, 1)` men det har samma effekt. Vad betyder *stride*? *Stride* är längden på förflyttningarna som görs när ett filter tillämpas. I exemplet från Avsnitt 8.2.1 var stride 1 medan i exemplet från Avsnitt 8.2.2 var stride 2.

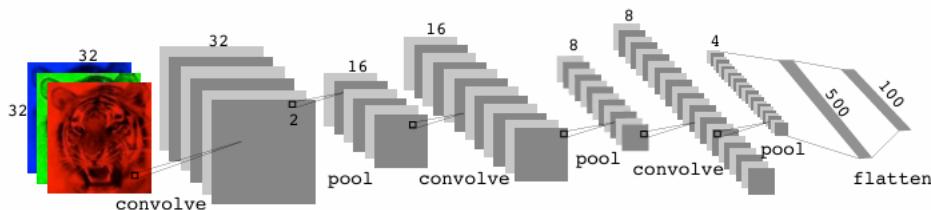
## 8.3 Exempel på en modellarkitektur för *CNN*

I detta avsnitt ska vi kolla på ett exempel där vi kommer se hur en modellarkitektur för ett *CNN* kan se ut där vi antar att datan som används består av 100 stycken klasser. Till vår hjälp har vi Figur 8.4 och vi förklarar vad vi ser i varje steg nedan:

1. För *input*-lagret ser vi en bild på en tiger med storleken  $(32 \times 32)$  som består av tre kanaler som representerar färgerna röd, grön och blå (förkortas ofta RGB). Generellt sett så används det som grundfärgar i additiv färgblandning med vars

hjälp man kan skapa andra färger. Se Figur 8.5 där RGB-färgerna kombineras för att skapa en bild.

2. För varje kanal tillämpar vi därefter ett *convolution layer* med två filter. Därför får vi nu sex kanaler totalt.
3. *Max pooling* tillämpas där storleken ( $2 \times 2$ ) används för filtreringen. Den nya bildstorleken blir därför ( $16 \times 16$ ).
4. *Convolution layer* med två filter tillämpas återigen vilket ger oss 12 stycken kanaler.
5. *Max pooling* tillämpas där storleken ( $2 \times 2$ ) används för filtreringen. Den nya bildstorleken blir därför ( $8 \times 8$ ).
6. *Convolution layer* med två filter tillämpas återigen vilket ger oss 24 stycken kanaler.
7. *Max pooling* tillämpas där storleken ( $2 \times 2$ ) används för filtreringen. Den nya bildstorleken blir därför ( $4 \times 4$ ) vilket gör att varje bild består av totalt 16 pixlar.
8. Ett *flatten layer* tillämpas så att pixlarna från de två dimensionella bilderna plattas ut till en dimension.
9. I sista steget har vi ett *output-lager* med 100 stycken noder eftersom vi antog att datan vi arbetar med består av 100 klasser. Vi använder oss generellt sett av *softmax* som aktiveringsfunktion när vi arbetar med multiklass klassificering. Se Tabell 7.2.



Figur 8.4: Arkitekturen för ett CNN där faltningsslagerna varvar med  $2 \times 2$  max-pool-lager. Storleken på bilden minskar med en faktor 2 i båda dimensionerna. Bilden är tagen från <https://www.statlearning.com/>.

Rent schematiskt hade en arkitektur enligt koden nedan kunnat användas. Notera, koden nedan följer inte exemplet kopplat till figuren vi precis gick igenom. I koden ser vi



Figur 8.5: Exempel på hur RGB-färgerna kan kombineras för att skapa en bild. Bilden är tagen från: <https://sv.wikipedia.org/wiki/RGB>.

hur vi använder fler filter för varje efterföljande Conv2D-lager (32, 64, 128, 256). Anledningen är att efter varje MaxPooling2D-lager så sjunker dimensionen på bilderna vilket alltså då kompenseras med fler kanaler. Vi nämner också att man till exempel hade kunnat använda flera Conv2D-lager i rad innan ett MaxPooling2D-lager används.

```

model = Sequential()
model.add(Input(shape=(32, 32, 3)))                                     ①
model.add(Conv2D(32, kernel_size=(3, 3), padding='same',
    ↴ activation='relu'))                                              ②
model.add(MaxPooling2D(pool_size=(2, 2)))                                ③
model.add(Conv2D(64, kernel_size=(3, 3), padding='same',
    ↴ activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(128, kernel_size=(3, 3), padding='same',
    ↴ activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(256, kernel_size=(3, 3), padding='same',
    ↴ activation='relu'))
model.add(Flatten())
model.add(Dropout(0.5))                                                 ④
model.add(Dense(512, activation='relu'))
model.add(Dense(100, activation='softmax'))

```

- ① Varje bild har storleken  $(32 \times 32)$  och tre kanaler som representerar färgerna röd, grön och blå.
- ② Ett *convolution layer* används med 32 stycken olika filter. Vilka filter som kommer användas lär modellen sig när den tränas på data.

- (3) Ett *max pooling layer* med storleken  $(2 \times 2)$  används.
- (4) *Dropout*, en typ av regularisering som vi gick igenom i Avsnitt 7.3.

Avslutningsvis, som läsaren säkert inser från exemplet ovan så kan modelleringen av neurala nätverk bli ganska komplex. Generellt sett provar vi oss därför fram och tar inspiration från dokumentation och hur andra har gjort.

## 8.4 *Data augmentation*

I samband med att vi arbetar med bild-data är en användbar teknik det som kallas för *data augmentation*. Det innebär att vi förvränger bilder på olika sätt, exempelvis roterar dem, ändrar ljusstyrkan eller zoomar in/ut, för att få fler bilder till vår träningsdata. Syftet med att öka storleken på träningsdata är att skydda modellerna mot att bli överanpassade vilket gör det till en form av regularisering. Generellt sett ska bilderna endast förvrängas så pass lite att en människas förmåga att tolka bilden inte ska påverkas. Se Figur 8.6.



Figur 8.6: Exempel på hur bilder kan förvrängas. Originalbilden ser vi längst till vänster och den har förvrängts på olika sätt för att vi ska kunna få fler bilder som har samma klass-etikett. Förvrängningarna är så pass små att vi i samtliga fall fortfarande ser att det är en tiger. Bilden är tagen från <https://www.statlearning.com/>.

## 8.5 Tre kodexempel

I detta avsnitt kommer vi kolla på tre kodexempel. I det första genomförs en klassificering av bilder, i det andra demonstreras förtränaade modeller vilket möjliggör oss att använda modeller som redan är tränade och slutligen i det tredje demonstreras *transfer learning* som möjliggör oss att återanvända lager från färdigtränaade modeller. Både förtränaade modeller och *transfer learning* kan vara mycket användbart vilket vi snart kommer se.

### 8.5.1 Kodexempel 1 - Bildklassificering av *CIFAR-100*

I detta kodexempel kommer vi arbeta med *CIFAR-100*. Datasetet består av 60000 stycken bilder i färgskala som har storleken  $(32 \times 32)$ . Det finns 20 superklasser där varje superklass har 5 underklasser. Från dokumentationen, se länkarna:

- <https://keras.io/api/datasets/cifar100/>
- <https://www.cs.toronto.edu/~kriz/cifar.html>

kan vi exempelvis se att superklassen *people* består av underklasserna [*baby*, *boy*, *girl*, *man*, *woman*] och att superklassen *fruit and vegetables* består av underklasserna [*apples*, *mushrooms*, *oranges*, *pears*, *sweet peppers*].

Vi börjar med att importera biblioteken för att därefter ladda in datan.

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

from tensorflow.keras.datasets import cifar100
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Input, Conv2D, MaxPooling2D,
    ↵ Flatten, Dropout, Dense
```

```
# Loading CIFAR-100 dataset
(x_train, y_train), (x_test, y_test) = cifar100.load_data()

# Limit the dataset so training goes faster for demonstration
    ↵ purposes
limit = 5000
(1)
x_train = x_train[:limit]
y_train = y_train[:limit]
x_test = x_test[:limit]
y_test = y_test[:limit]

# Normalize the input images
x_train = x_train / 255.0
x_test = x_test / 255.0

# Checking dimension
```

```
print(np.shape(x_train))
```

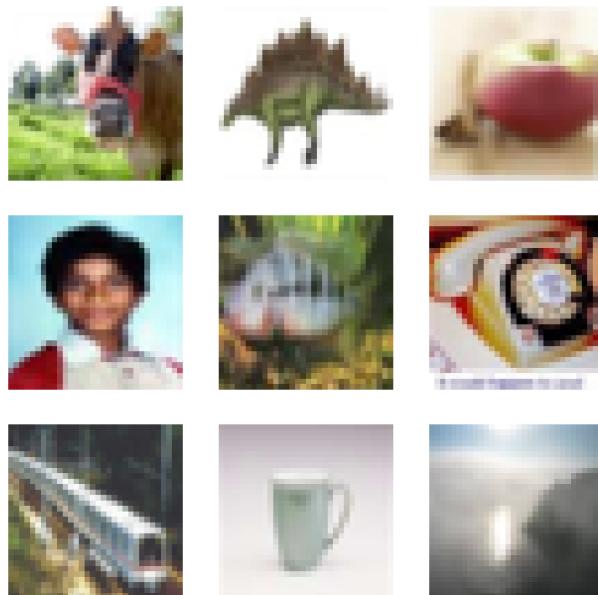
- ① Vi laddar endast in ett urval, `limit = 5000`, av datan för att träningen ska gå snabbare. Läsaren uppmanas att prova använda hela datasetet.

(5000, 32, 32, 3)

Från resultatet ser vi att vår träningsdata, `x_train` har formen (5000, 32, 32, 3). Det är alltså fyra dimensioner och representerar 5000 bilder där varje bild har storleken  $(32 \times 32)$  och tre stycken kanaler för färgerna röd, grön och blå.

I koden nedan gör vi en visualisering för att se hur bilderna från *CIFAR-100* kan se ut.

```
plt.figure(figsize=(4, 4))
for i in range(9):
    plt.subplot(3, 3, i + 1)
    plt.imshow(x_train[i])
    plt.axis('off')
```



Med koden nedan skapar vi en *CNN*-modell.

```
# Create the model
model = Sequential()
model.add(Input(shape=(32, 32, 3)))
model.add(Conv2D(32, kernel_size=(3, 3), padding='same',
    ↴ activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(64, kernel_size=(3, 3), padding='same',
    ↴ activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(128, kernel_size=(3, 3), padding='same',
    ↴ activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(256, kernel_size=(3, 3), padding='same',
    ↴ activation='relu'))
model.add(Flatten())
model.add(Dropout(0.5))
model.add(Dense(512, activation='relu'))
model.add(Dense(100, activation='softmax'))

# Compile the model
model.compile(loss='sparse_categorical_crossentropy',
    ↴ metrics=['accuracy'])

# Train the model
history = model.fit(x_train, y_train, epochs=10, batch_size=128,
    ↴ validation_split=0.2, verbose=2) ①
```

- ① I början av kodexemplet delade vi endast upp vår data i träning och test. Därför specificerar vi här `validation_split=0.2` vilket är smidigt.

```
Epoch 1/10
32/32 - 2s - 78ms/step - accuracy: 0.0127 - loss: 4.6039 - val_accuracy:
0.0200 - val_loss: 4.5905
Epoch 2/10
32/32 - 1s - 40ms/step - accuracy: 0.0230 - loss: 4.4958 - val_accuracy:
0.0220 - val_loss: 4.3912
```

```
Epoch 3/10
32/32 - 2s - 51ms/step - accuracy: 0.0258 - loss: 4.3880 - val_accuracy:
0.0370 - val_loss: 4.3294
Epoch 4/10
32/32 - 2s - 65ms/step - accuracy: 0.0420 - loss: 4.3086 - val_accuracy:
0.0340 - val_loss: 4.3400
Epoch 5/10
32/32 - 2s - 70ms/step - accuracy: 0.0463 - loss: 4.2231 - val_accuracy:
0.0270 - val_loss: 4.5428
Epoch 6/10
32/32 - 3s - 84ms/step - accuracy: 0.0505 - loss: 4.1475 - val_accuracy:
0.0470 - val_loss: 4.1340
Epoch 7/10
32/32 - 3s - 95ms/step - accuracy: 0.0593 - loss: 4.0646 - val_accuracy:
0.0610 - val_loss: 4.1545
Epoch 8/10
32/32 - 3s - 102ms/step - accuracy: 0.0705 - loss: 3.9951 -
val_accuracy: 0.0670 - val_loss: 4.0321
Epoch 9/10
32/32 - 3s - 81ms/step - accuracy: 0.0880 - loss: 3.9211 - val_accuracy:
0.0690 - val_loss: 4.0629
Epoch 10/10
32/32 - 3s - 83ms/step - accuracy: 0.0970 - loss: 3.8486 - val_accuracy:
0.0780 - val_loss: 3.9566
```

Slutligen evaluerar vi vår modell på testdata.

```
y_pred = model.predict(x_test, verbose=2)
y_pred_labels = np.argmax(y_pred, axis=1)
accuracy = np.mean(y_pred_labels == y_test.flatten())
print(f"Accuracy: {accuracy}")
```

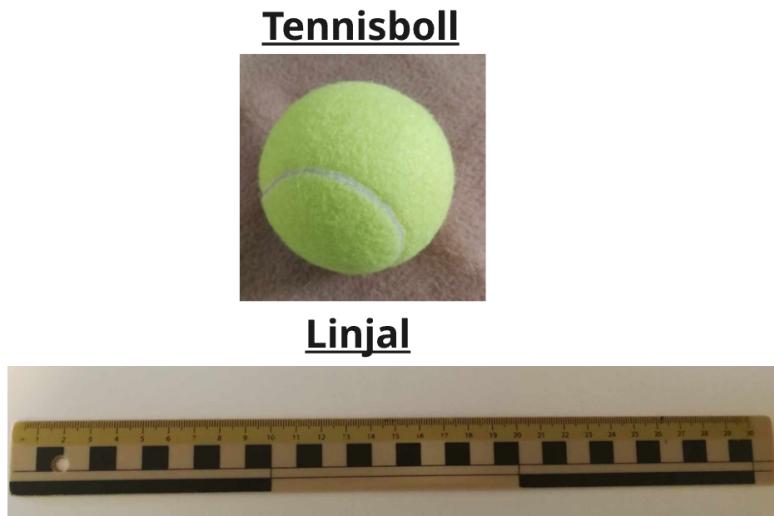
```
157/157 - 2s - 14ms/step
```

```
Accuracy: 0.0874
```

Vi ser att *accuracy* blev tämligen låg, läsaren uppmanas att försöka förbättra den vilket är en bra övning. Ett första steg kan vara att använda hela datasetet, därefter kan man prova andra modellarkitekturen genom att exempelvis justera antalet dolda lager och antalet noder i dessa. Även fler epoker under träningen av modellen kan provas.

### 8.5.2 Kodexempel 2 - Förtränaade modeller

I detta kodexempel kommer vi demonstrera hur vi kan använda en förtränaad modell för att prediktera två bilder som vi själva tagit med mobilen. Den ena bilden är på en tennisboll och den andra är på en linjal, se Figur 8.7.



Figur 8.7: Två bilder som vi själva tagit med mobilen där den ena är på en tennisboll och den andra på en linjal. En förtränaad modell kommer användas för att försöka klassificera bilderna.

Vi kommer ladda in en CNN-arkitektur som heter *ResNet50* och den har cirka 26 miljoner parametrar (kom ihåg att neurala nätverk generellt sett hade väldigt många parametrar). Dokumentationen är tillgänglig här: <https://keras.io/api/applications/>. Modellen har blivit tränad på ett dataset som heter *ImageNet* och det består av 1000 olika klasser, de olika klasserna som finns tillgängliga kan exempelvis ses här: <https://deeplearning.cms.waikato.ac.nz/user-guide/class-maps/IMAGENET/>.

```
from tensorflow.keras.applications.resnet50 import ResNet50
from tensorflow.keras.preprocessing import image
from tensorflow.keras.applications.resnet50 import
    preprocess_input, decode_predictions
import numpy as np
```

```

# Load pre-trained ResNet50 model
model = ResNet50(weights='imagenet')                                ①

# Paths to two images
img_paths = ['tennisboll.jpg', 'linjal.jpg']                         ②

# Load and preprocess both images
imgs = []
for path in img_paths:
    img = image.load_img(path, target_size=(224, 224))            ③
    x = image.img_to_array(img)                                     ④
    x = np.expand_dims(x, axis=0)                                    ⑤
    x = preprocess_input(x)                                         ⑥
    imgs.append(x)

# Stack into a single batch
batch = np.vstack(imgs)

# Predict
preds = model.predict(batch, verbose=2)                            ⑦

# Decode and print results
for i, pred in enumerate(preds):
    print(f'Predicted for image {i+1}:',
          decode_predictions(pred.reshape(1, 1000), top=3)[0])      ⑧

```

- ① ResNet50 modellen laddas in där vikterna för modellen blivit tränade på datasetet *ImageNet*.
- ② Två bilder som vi tagit med vår mobiltelefon. Andra bilder kan också användas. Prova gärna ta egna bilder.
- ③ Våra bilder laddas in och sätts till storleken  $(224 \times 224)$  vilket är storleken bilderna från *ImageNet* har.
- ④ Bilderna transformeras till *NumPy array*.
- ⑤ Vi gör att respektive bild får följande *shape*:  $(1, 224, 224, 3)$ .
- ⑥ Bilderna förbehandlas så de har samma format som de i *ImageNet*. Rent generellt, när vi laddar in färdiga modeller kan vi använda oss av respektive modells *preprocess\_input*, det är väldigt användbart.

- (7) Vi predikterar respektive bild.
- (8) Prediktionerna skrivs ut där vi ser de tre främsta resultaten vilket specificerats med `top=3`.

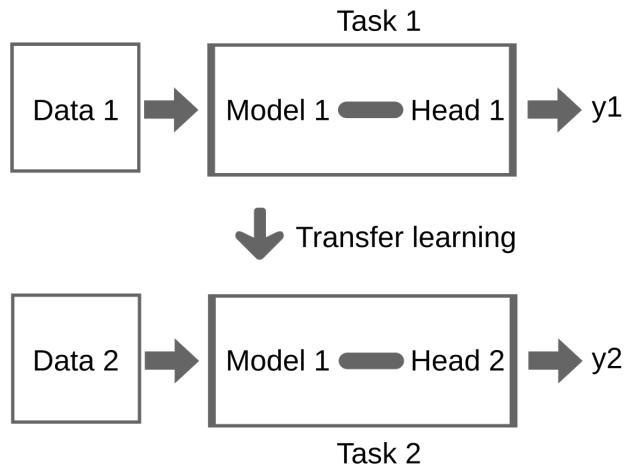
```
1/1 - 3s - 3s/step
Predicted for image 1: [('n04409515', 'tennis_ball',
np.float32(0.9999969)), ('n04039381', 'racket',
np.float32(2.7892713e-06)), ('n03942813', 'ping-pong_ball',
np.float32(5.5036654e-08))]
Predicted for image 2: [('n03777754', 'modem', np.float32(0.6229953)),
('n04118776', 'rule', np.float32(0.14351183)), ('n03494278',
'harmonica', np.float32(0.0778282))]
```

Från koden ser vi att den första bilden predikterades till en tennisboll med väldigt hög sannolikhet vilket är korrekt. Den andra bilden, som är svårare att prediktera, gör modellen fel på i den bemärkelse att den högsta sannolikheten är för kategorin *modem*. Däremot är linjal (*rule*) på andra plats.

### 8.5.3 Kodexempel 3 - *Transfer learning*

I det föregående kodexemplet använde vi *förträpnade* modeller som direkt kunde användas för att göra prediktioner. En praktisk begränsning med förträpnade modeller är att de endast kan prediktera de kategorierna som i fallet ovan finns på *ImageNet*. Önskar vi prediktera några kategorier som inte finns med där har vi alltså problem. För att lösa detta problem kan vi utnyttja de förträpnade modellerna men anpassa dem genom att justera de sista lagren. Denna metodik benämns *transfer learning*. På så sätt kan det neurala nätverket lära sig att känna igen, och därmed prediktera, de kategorier vi vill arbeta med. Denna metod är intuitivt rimlig. De nedre lagren i ett neutralt nätverk fångar generella låg-nivå mönster som kan återanvändas medan de övre lagren kombinerar dessa för att identifiera mer komplexa och specifika strukturer. Genom att behålla de nedre lagren och bara träna om de övre kan modellen anpassas till nya uppgifter utan att behöva tränas om från grunden. Se Figur 8.8 för en schematisk illustration av hur *transfer learning* fungerar.

För att illustrera hur kraftfullt *transfer learning* kan vara använder vi samma dataset som i Avsnitt 8.5.1, *CIFAR-100*. Koden i det första kodblocket nedan är samma som i Avsnitt 8.5.1 förutom att vi behöver några andra bibliotek som vi importerar.



Figur 8.8: En schematisk bild över hur transfer learning fungerar. Bilden är tagen från [https://en.wikipedia.org/wiki/Transfer\\_learning](https://en.wikipedia.org/wiki/Transfer_learning).

```

import numpy as np
from tensorflow.keras.datasets import cifar100
from tensorflow.keras.applications import MobileNetV2
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Flatten

# Same code as before
(x_train, y_train), (x_test, y_test) = cifar100.load_data()

limit = 5000
x_train = x_train[:limit]
y_train = y_train[:limit]
x_test = x_test[:limit]
y_test = y_test[:limit]

```

```
x_train = x_train / 255.0
x_test = x_test / 255.0
```

I koden nedan laddar vi in en modell som heter **MobileNetV2**, vi hade även kunnat använda **ResNet50** som i föregående avsnitt men vi vill demonstrera att det även finns andra modeller.

```
# Resize images to fit MobileNetV2 input requirements
x_train = tf.image.resize(x_train, (128, 128))
x_test = tf.image.resize(x_test, (128, 128))

# Load MobileNetV2 base model without top layers
base = MobileNetV2(weights='imagenet', include_top=False,
                     input_shape=(128, 128, 3))                                ①
base.trainable = False                                         ②

# for layer in base.layers[-10:]:
#     layer.trainable = True                                         ③

# Add two new top layers
model = Sequential([
    base,                                                       ④
    Flatten(),                                                 ⑤
    Dense(100, activation='softmax')                           ⑥
])

model.compile(loss='sparse_categorical_crossentropy',
               metrics=['accuracy'])
model.fit(x_train, y_train, epochs=5, validation_split=0.2,
          verbose=2)                                              ⑦

# Evaluate model test set
y_pred = model.predict(x_test, verbose=2)
y_pred_labels = np.argmax(y_pred, axis=1)
accuracy = np.mean(y_pred_labels == y_test.flatten())
print(f"Accuracy: {accuracy}")
```

- ① Vi specificerar `include_top=False`. Från dokumentationen kan vi utläsa att det betyder följande: *include\_top: Boolean, whether to include the fully-connected layer at the top of the network. Defaults to True.*
- ② `base.trainable = False` gör att vikterna i den förtränaade modellen vi laddar in (där det sista lagret alltså är exkluderat) frysas och inte uppdateras.
- ③ Hade vi exempelvis önskat uppdatera de tio sista lagren i den förtränaade modellen vi laddade in hade vi kunnat göra det med denna kod. Vi gör dock inte det i detta kodexempel, därför är koden är utkommenterad.
- ④ Vi specificerar att vi vill använda den förtränaade modellen som vi laddade in.
- ⑤ Vi lägger till ett `Flatten()`-lager.
- ⑥ *CIFAR-100* datan har 100 klasser och därför använder vi 100 noder i *output*-lagret med *softmax* som aktiveringsfunktion.
- ⑦ Vi använder endast fem epoker för att träningen ska ta mindre tid.

```

Epoch 1/5
125/125 - 31s - 251ms/step - accuracy: 0.1850 - loss: 11.9968 -
val_accuracy: 0.2760 - val_loss: 7.9463
Epoch 2/5
125/125 - 23s - 181ms/step - accuracy: 0.5312 - loss: 3.8684 -
val_accuracy: 0.3350 - val_loss: 7.9911
Epoch 3/5
125/125 - 20s - 163ms/step - accuracy: 0.7193 - loss: 1.8742 -
val_accuracy: 0.2760 - val_loss: 10.0518
Epoch 4/5
125/125 - 22s - 175ms/step - accuracy: 0.8210 - loss: 1.0611 -
val_accuracy: 0.3370 - val_loss: 8.5998
Epoch 5/5
125/125 - 22s - 178ms/step - accuracy: 0.8925 - loss: 0.5625 -
val_accuracy: 0.3110 - val_loss: 10.4563
157/157 - 21s - 132ms/step
Accuracy: 0.3344

```

Jämför vi resultatet vi fick för *accuracy* på testdatan här jämfört med Avsnitt 8.5.1 så ser vi att vi fick mycket bättre resultat.

Sammanfattningsvis, *transfer learning* kan vara mycket användbart och på följande källa kan den intresserade läsaren lära sig mer:  
[https://keras.io/guides/transfer\\_learning/](https://keras.io/guides/transfer_learning/).

## 8.6 Övningsuppgifter

Övningsuppgifter för kapitlet finns på bokens hemsida:

[https://github.com/AntonioPrgomet/ai\\_tillaempad\\_ml](https://github.com/AntonioPrgomet/ai_tillaempad_ml)

# Kapitel 9

## *Recurrent neural network (RNN)*

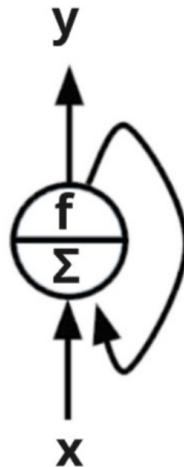
I detta kapitel kommer vi få en mycket översiktlig introduktion till *Recurrent neural network (RNN)*. Ämnet är stort och det finns många tillämpningsområden varför vi av omfattningsskäl inte kommer fördjupa oss i detaljer. Efter denna översiktliga introduktion hoppas vi att läsaren ska känna sig inspirerad och rustad för att själv kunna hitta mer information och fördjupa sig inom önskade delar.

Kapitlet inleds med en bakgrund, följt av en genomgång av två populära modellarkitekturen. Därefter introduceras några utvalda grundkoncept inom *natural language processing (NLP)* som handlar om hanteringen av naturligt språk med hjälp av datorer. Vi avslutar kapitlet med ett kodexempel kopplat till *NLP* där vi genomför en sentimentanalys.

### 9.1 Bakgrund

I föregående två kapitel var de neurala nätverken vi arbetade med, ANN och *CNN*, av typen *feedforward*. Det innebär att modellen tar *input data*, bearbetar den i de dolda lagren för att slutligen returnera en *output*. Daten flödade (*feed*) från *input*-lagret framåt (*forward*) till *output*-lagret. I detta kapitel kommer vi arbeta med *RNN* som alltså är av typen *recurrent*. Det innebär att *outputen* från en neuron vid ett tidssteg används

som *input* i det neurala nätverket för nästa tidssteg. Se Figur 9.1. Detta möjliggör *RNN*-modeller att hantera tidsmässiga beroenden och mönster i sekvenser såsom skriven text.



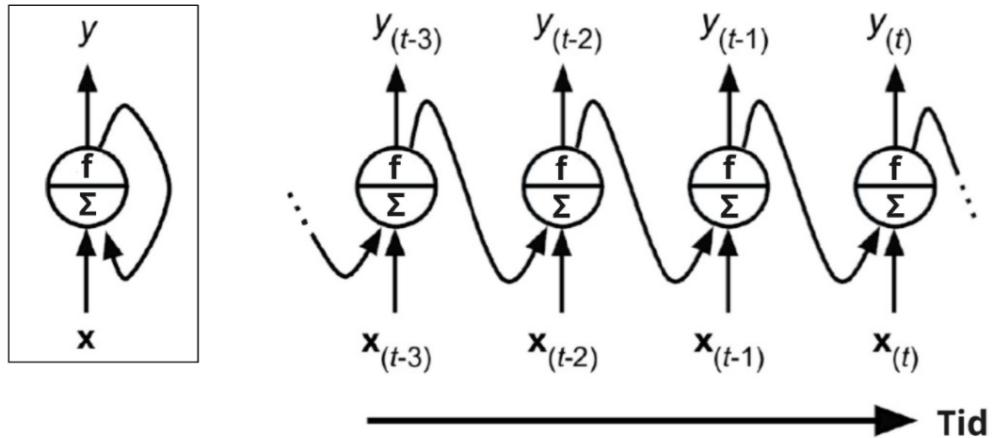
Figur 9.1: Outputen från en neuron vid ett tidssteg används som input i det neurala nätverket för nästa tidssteg inom RNN.

Rent generellt gäller det att *RNN* är designade för att hantera sekventiell data såsom text, tal och tidsserier. I alla dessa fall är ordningen av datan viktig. Om vi exempelvis har texten ”*jag gillar att lära mig nya saker*” så hade det fått en helt annan innehörd om vi skrev ”*mig saker jag nya att gillar lära*”.

I Figur 9.1 såg vi hur en neuron såg ut för en given tidpunkt. Om vi följer hur nätverket ser ut genom *tid* (på engelska brukar detta beskrivas som ”*unfolded in time*”), får vi Figur 9.2. Vi ser exempelvis att *outputen*  $y_{t-2}$  är resultatet av *inputen*  $x_{t-2}$  och  $y_{t-3}$ , på motsvarande sätt gäller det att *outputen*  $y_{t-1}$  är resultatet av *inputen*  $x_{t-1}$  och  $y_{t-2}$  och *outputen*  $y_t$  är resultatet av *inputen*  $x_t$  och  $y_{t-1}$ . Att *outputen* för ett tidssteg beror på *outputen* i föregående tidssteg, som i sin tur beror på *outputen* i föregående tidssteg och så vidare, medför att *RNN*-modeller har ett minne. Detta minne gör att sekventiell data där ordningen spelar roll kan hanteras.

När det gäller *input* och *output* för *RNN* kan det vara såväl vektorer som sekvenser. Vi går inte djupare in på det men följande är bra att känna till för att kunna identifiera olika typer av problemkategorier:

1. Sekvens-till-sekvens: Exempelvis kanske vi skapar en modell som kan ta en sekvens



Figur 9.2: Visualisering av hur en RNN-modell ser ut genom tid.

av historiska aktiekurser och prediktera aktiekurser som är förskjutna en dag framåt.

2. Sekvens-till-vektor: Exempelvis kan vi stoppa in orden från en filmrecension och prediktera sentimentet för recensionen, det vill säga om den upplevs som bra (1) eller dålig (0). Detta kommer att demonstreras i Avsnitt 9.4.
3. Vektor-till-sekvens: Exempelvis kan vi stoppa in en bild som representeras som en vektor och få en beskrivning av bilden vilket alltså då är en sekvens av ord.
4. Sekvens-till-vektor (*encoder*) åtföljt av vektor-till-sekvens (*decoder*): Exempelvis kan en mening skriven på svenska representeras som en vektor (*encoder*) som *decoder* sedan översätter till engelska.

Härnäst kollar vi på modellarkitekturen.

## 9.2 Modellarkitekturen

I detta avsnitt kommer vi kolla på två populära *RNN*-arkitekturen som heter *Long short-term memory (LSTM)* och *Gated recurrent unit (GRU)*.

### 9.2.1 Long short-term memory (*LSTM*)

Från *Keras* kan vi använda klassen `SimpleRNN` som enligt dokumentationen definieras som "*Fully-connected RNN where the output is to be fed back to input*". I kod hade det kunnat se ut enligt nedan. Notera, koden är inte fullständig utan syftet är att illustrera hur `SimpleRNN` klassen kan användas.

```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(1, input_shape=[None, 1])
])
```

Ett problem med den typen av modeller är att de glömmar viss information efter varje steg vilket leder till att de efter ett tag inte kommer ihåg den *input-data* som stoppades in i början vilket kan bli problematiskt för längre sekvenser av data. Detta fenomen benämns *Short Term Memory Problem*. För att hantera denna problematik kan *LSTM*-arkitekturen användas som har ett mer avancerat minne och kan lära sig vad som är viktigt att komma ihåg och vad som kan glömmas bort. Generellt sett är *LSTM* den mest använda *RNN*-arkitekturen.

I kod kan *LSTM* se ut enligt nedan.

```
model = keras.models.Sequential([
    keras.layers.LSTM(20, return_sequences=True,
        ↵ input_shape=[None, 1]),
    keras.layers.LSTM(20, return_sequences=True),
    keras.layers.TimeDistributed(keras.layers.Dense(10))
])
```

### 9.2.2 Gated recurrent unit (*GRU*)

*GRU* är en förenklad variant av *LSTM* men har i många sammanhang visat sig fungera lika bra som *LSTM*. I kod kan *GRU* se ut enligt nedan.

```
model = keras.models.Sequential([
    keras.layers.GRU(20, return_sequences=True, input_shape=[None,
        ↵ 1]),
    keras.layers.GRU(20, return_sequences=True),
    keras.layers.TimeDistributed(keras.layers.Dense(10))
```

①

] )

- ① I tidsseriesammanhang förtydligar *wrapper* `TimeDistributed` att det lagret ska användas för varje tidssteg. Vi går inte djupare in på detta.

## 9.3 *Natural language processing (NLP)*

*NLP* handlar om att hantera mänskligt språk med hjälp av datorer och det finns flera tillämpningsområden. Ett exempel är att klassificera om en film anses vara bra eller dålig beroende på dess omdöme. Detta område benämns för sentimentanalys och vi kommer kolla på ett kodexempel kopplat till detta i Avsnitt 9.4. Sentimentanalys kan exempelvis också användas för att analysera allmän opinion baserat på inlägg från sociala medier.

Inom *NLP* är följande terminologi vanligt förekommande:

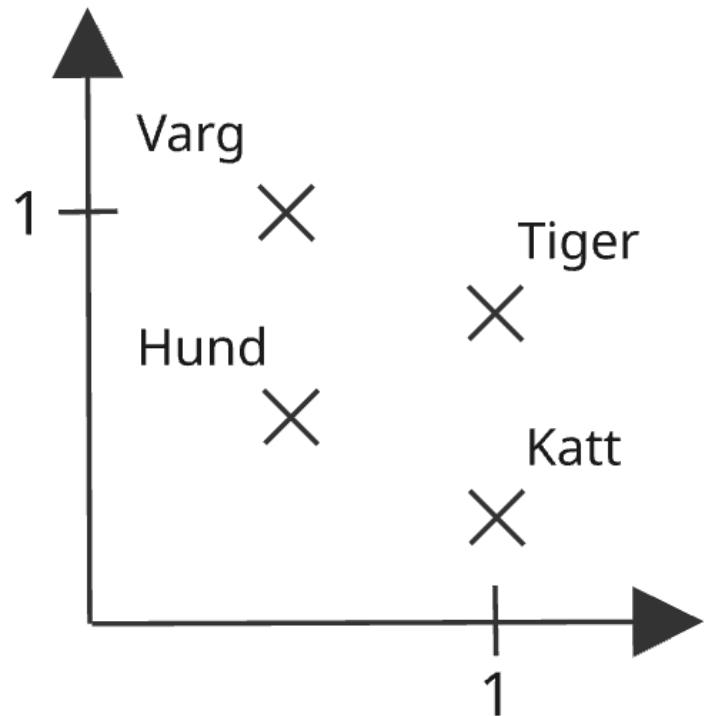
- Corpus: Samling av texter, t.ex. alla filmrecensioner för en specifik film.
- Dokument: En enhet i ett corpus, t.ex. en filmrecension för en film.
- Token: Enhets som text kan delas upp i, t.ex. ord eller bokstäver.

### 9.3.1 *Embeddings*

Antag att vi har en filmrecension där följande står: “*this film was just brilliant (...)*”. Då är det en rimlig gissning att filmen anses vara bra. För att kunna använda text-data i våra modeller måste vi först transformera texten till siffror.

Ett sätt att göra detta på är att använda oss av *one-hot-encoding* som gicks igenom i Avsnitt 1.3.6. Då hade vi fått en hög-dimensionell matris som hade bestått av mestadels nollar, matriser med mestadels nollar benämns *sparse*. Matrisen blir högdimensionell för det blir en ny kolumn för varje ord som används. Matrisen kommer bestå av mestadels nollar eftersom med *one-hot-encoding* blir endast den kolumn som representerar ordet 1 och övriga 0. Av praktiska skäl är alltså metoden med *one-hot-encoding* inte särskild smidig att använda. Istället använder vi oss av det som kallas för *word-embedding* vilket innebär att ord representeras som numeriska vektorer. Här vill man att geometriska förhållanden såsom avstånd mellan vektorerna ska reflektera ordens semantiska betydelse. Vi ser i Figur 9.3 att avståndet mellan “Varg” och “Hund” är lika stort som mellan “Tiger” och “Katt” eftersom vi går från husdjur till vilddjur i båda fallen.

Generellt sett kan vi använda ett *embedding layer* där vikterna tränas från datan eller så kan vi använda förträgnade *embeddings* såsom `Word2vec` eller `GloVe` som den intresserade



Figur 9.3: Exempel på word-embedding där varje ord representeras som en vektor (tvådimensionell koordinat i detta fallet). Exempelvis hade "Varg" kunnat representeras med vektorn  $(1, 0.5)$ .

läsaren kan läsa mer om på internet. I kodexemplet som följer kommer vi demonstrera användandet av *embeddings* i en sentimentanalys.

## 9.4 Kodexempel - Sentimentanalys

I detta kodexempel kommer vi genomföra en sentimentanalys på ett dataset som innehåller filmrecensioner från IMDB. Följande kan utläsas från dokumentationen för datasetet:

*This is a dataset of 25,000 movies reviews from IMDB, labeled by sentiment (positive/negative). Reviews have been preprocessed, and each review is encoded as a list of word indexes (integers). For convenience, words are indexed by overall frequency in the dataset, so that for instance the integer “3” encodes the 3rd most frequent word in the data.*

Vi börjar med att importera de bibliotek vi kommer använda.

```
import numpy as np
import matplotlib.pyplot as plt
from tensorflow import keras
from tensorflow.keras.datasets import imdb
from tensorflow.keras.preprocessing.sequence import pad_sequences
from sklearn.metrics import confusion_matrix,
    ConfusionMatrixDisplay
```

Vi fortsätter med att ladda in datan.

```
(x_train, y_train), (x_test, y_test) =
    ↵ imdb.load_data(num_words=10000)                                ①

limit = 12500
x_train = x_train[:limit]
y_train = y_train[:limit]
x_test = x_test[:limit]
y_test = y_test[:limit]

print(x_train.shape)
print(y_train.shape)
```

- ① num\_words=10000 gör att vi behåller de 10000 mest vanligt förekommande orden. Resterande ord representeras som oov\_char (*Out-Of-Vocabulary*). y-train och y-test består av värdena 1 och 0. Filmer som fått ett positivt omdöme representeras med 1 och filmer som fått ett negativt omdöme representeras med 0.
- ② Vi begränsar mängden data så koden ska gå snabbare att exekvera. Läsaren kan ta bort denna begränsning om hela datasetet vill användas.

```
(12500,)  
(12500,)
```

```
print(x_train[0][:12])
```

①

- ① Varje ord representeras med ett heltal som motsvarar den plats ordet har i vår ordlista som består av 10000 ord.

```
[1, 14, 22, 16, 43, 530, 973, 1622, 1385, 65, 458, 4468]
```

Med koden nedan skriver vi ut den faktiska texten istället för siffrorna ovan.

```
word_index = imdb.get_word_index()

def decode_review(text, word_index):
    reverse_word_index = dict([(value, key) for (key, value) in
    ↵ word_index.items()])
    decoded_review = ' '.join([reverse_word_index.get(i - 3, '?')
    ↵ for i in text])
    return decoded_review

print(decode_review(x_train[0][:12], word_index))
```

```
? this film was just brilliant casting location scenery story direction
everyone's
```

Med koden nedan tar vi fram lite statistik för dokumentens längd där vi beräknar medianlängden och hur stor andel av dokumenten som har mindre än eller lika med 500 ord.

```
wc = np.array([len(x) for x in x_train])
print(np.median(wc))
```

```
print(np.mean(wc <= 500))
```

```
178.0  
0.91536
```

Eftersom vi vet att cirka 91% av dokumenten har färre än 500 ord är det rimligt att ange att alla dokument ska vara 500 ord långa, det uppnår vi genom funktionen `pad_sequences` nedan. Dokument som är kortare fylls på med nollor och dokument som är längre trunkeras.

```
maxlen = 500  
x_train = pad_sequences(x_train, maxlen=maxlen)  
x_test = pad_sequences(x_test, maxlen=maxlen)  
print(np.shape(x_train))  
print(np.shape(x_test))
```

(1)

① Vi verifierar att längden på dokumenten är 500.

```
(12500, 500)  
(12500, 500)
```

Nedan skapar vi ett *RNN* där vi använder oss av *embeddings* och *LSTM*.

```
model = keras.models.Sequential()  
model.add(keras.layers.Embedding(input_dim=10000, output_dim=32))  
    ↪  
model.add(keras.layers.LSTM(units=32))  
    ↪  
model.add(keras.layers.Dense(units=1, activation='sigmoid'))  
    ↪  
  
model.compile(optimizer='rmsprop',  
              loss='binary_crossentropy',  
              metrics=['accuracy'])  
  
history = model.fit(x_train, y_train,  
                     epochs=5,  
                     batch_size=128,  
                     validation_data=(x_test, y_test),  
                     verbose=2)
```

(1)

(2)

(3)

- ① Tidigare specificerade vi `num_words=10000` innebärande att vår ordlista har 10000 ord. `output_dim=32` betyder att varje ord kommer representeras som en vektor bestående av 32 stycken element. I Figur 9.3 representerades varje ord av en vektor bestående av två element.
  - ② Vi använder ett *LSTM*-lager.
  - ③ I *output*-lagret använder vi *sigmoid* som aktiveringsfunktion, den aktiveringsfunktionen medför att vår *output* blir ett värde mellan 0 – 1 vilket vi tolkar som en sannolikhet att omdömet för filmen är positivt (1). Våra prediktioner blir alltså en sannolikhet, för att konvertera dem till värdet 1 och 0 kommer vi i kodblocket nedan att använda oss av koden:
- ```
predictions = (model.predict(x_test, verbose=2) > 0.5).astype(int)
```
- Vad den koden gör är att alla filmerna som vår modell predikterar har mer än 50% sannolikhet att ha fått ett positivt omdöme kommer vi klassificera som att de fått ett positivt omdöme och övriga negativt omdöme.

```
Epoch 1/5
98/98 - 40s - 409ms/step - accuracy: 0.6086 - loss: 0.6567 -
val_accuracy: 0.5922 - val_loss: 0.6746
Epoch 2/5
98/98 - 39s - 399ms/step - accuracy: 0.7917 - loss: 0.4584 -
val_accuracy: 0.8246 - val_loss: 0.4064
Epoch 3/5
98/98 - 43s - 440ms/step - accuracy: 0.8464 - loss: 0.3657 -
val_accuracy: 0.8452 - val_loss: 0.3565
Epoch 4/5
98/98 - 31s - 320ms/step - accuracy: 0.8639 - loss: 0.3355 -
val_accuracy: 0.8486 - val_loss: 0.3534
Epoch 5/5
98/98 - 31s - 316ms/step - accuracy: 0.8838 - loss: 0.2966 -
val_accuracy: 0.8383 - val_loss: 0.3695
```

Vi avslutar kodexemplet med att utvärdera modellen genom att ta fram en *confusion matrix* baserat på testdatan.

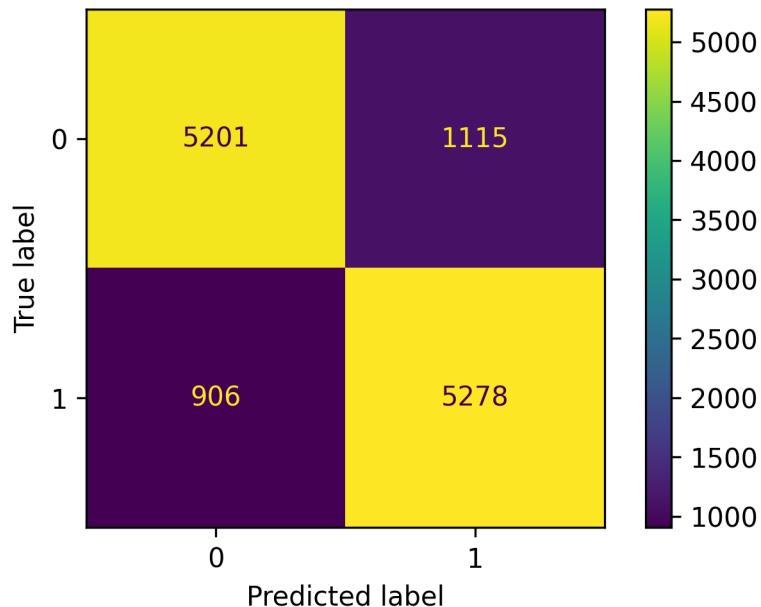
```
predictions = (model.predict(x_test, verbose=2) > 0.5).astype(int)

result = confusion_matrix(y_test, predictions)
disp = ConfusionMatrixDisplay(result)
disp.plot()
```

①

① Vi visualiseringar en *confusion matrix*.

391/391 - 21s - 54ms/step



## 9.5 Övningsuppgifter

Övningsuppgifter för kapitlet finns på bokens hemsida:  
[https://github.com/AntonioPrgomet/ai\\_tillaempad\\_ml](https://github.com/AntonioPrgomet/ai_tillaempad_ml)



# Kapitel 10

## Chattbottar

I detta kapitel kommer vi lära oss mer om hur chattbottar såsom ChatGPT fungerar och hur man ställer effektiva frågor till dem, vilket kallas *prompt engineering*. Vi kommer därefter lära oss hur vi bygger en lokal chattbot som vi kan använda på vår egna dator. Kapitlet avslutas med att gå igenom *Retrieval Augmented Generation (RAG)* vilket låter oss anpassa chattbottens svar utifrån en given kontext, exempelvis utifrån egna dokument.

### 10.1 ChatGPT och *prompt engineering*

Chattbottar, som till exempel ChatGPT, används av många människor för en rad olika syften. Alltifrån att skapa kreativt innehåll, få förslag på förbättringar av skriven text eller programmeringskod till att lösa mer komplexa problem.

I korthet har chattbottar tränats på enorma datamängder och har från detta lärt sig vad som är rimliga sekvenser av ord. Exempelvis vet vi att om någon säger “Hej, hur mår \_\_\_\_” så brukar det sista ordet vara “du” för att meningen ska bli “Hej, hur mår du”.

När vi ställer en fråga till ChatGPT, uppskattar den sannolikheter för nästkommande ord och ger alltså ett svar som är sannolikt utifrån den datan som den blivit tränad på. I korthet kan vi säga att det fungerar som en ”stokastisk papegoja”. Stokastisk för att den uppskattar sannolikheter och papegoja för att den upprepar vad den lärt sig baserat på datan den blivit tränad på.

Det är uppenbart att AI-modeller och svaren de ger är beroende av datan de tränats på och alltså inte “sanna” i en egentlig mening. Läsaren uppmanas till att tänka igenom vilka etiska implikationer och risker detta kan ha.

Den läsare som vill fördjupa sig i hur ChatGPT funkar kan läsa igenom följande relativt korta bok: “What Is ChatGPT Doing ... and Why Does It Work?” av Stephen Wolfram. Boken finns tillgänglig här: <https://writings.stephenwolfram.com/2023/02/what-is-chatgpt-doing-and-why-does-it-work/>.

Ett gammalt talesätt säger “som man frågar får man svar”. Detta gäller även chattbottar. Generellt sett kan vi ställa de frågor vi har och få rimliga svar utan att behöva tänka alltför mycket. Om vi däremot vill ha bättre svar kan vi försöka ställa effektiva frågor. Hur detta rent praktiskt går till är temat för det som kallas för *prompt engineering*. Olika chattbottar kan fungera på olika sätt men som tumregel är det bra att vara så specifik, deskriptiv och detaljerad som möjligt om önskad kontext, utfall, längd, format, stil med mera. Om vi jämför de två prompterna nedan,

“skriv en dikt om AI”.

och

“*Skriv en kort och inspirerande dikt om AI som fokuserar på dess möjligheter och risker inom utbildning. Stilen ska likna Shakespears.*”

är den andra mer effektiv.

Den läsare som vill lära sig mer om *prompt engineering* kan exempelvis kolla på följande guide kopplad till ChatGPT: <https://help.openai.com/en/articles/6654000-best-practices-for-prompt-engineering-with-the-openai-api>.

## 10.2 Molnbaserade och lokala språkmodeller

Att träna en *Large Language Model* (LLM) kostar både tid och pengar och kräver stora mängder energi. För de allra flesta är det därför bättre att använda en förtränad modell. Det kan vi åstadkomma antingen genom att koppla upp mot en modell i molnet via ett API, eller genom att ladda ned en förtränad modell och köra den lokalt på vår egen dator.

Nedan beskrivs några för- och nackdelar med respektive tillvägagångssätt och där efter demonstreras hur vi skapar en chattbot som kopplar upp mot en modell i molnet.

### **10.2.1 Molnbaserade modeller**

Att använda sig av molnbaserade modeller har sina för- och nackdelar.

#### **Fördelar**

- Enkla att implementera i ett befintligt system tack vare API
- Skalar automatiskt efter behov
- Uppdateras i takt med att teknologin utvecklas

#### **Nackdelar**

- Ökad användning kan leda till ökade kostnader
- Risker med hantering av känslig data
- Kräver extern anslutning till internet, hastigheten kan påverkas av problem med nätverket

### **10.2.2 Lokala modeller**

Även lokala modeller har sina för- och nackdelar.

#### **Fördelar**

- Större kontroll över känslig data
- Kan tränas ytterligare för att skapa specialiserade modeller
- Svarar snabbare och kräver ingen extern anslutning till internet

#### **Nackdelar**

- Investeringar i infrastruktur kan innehåra höga kostnader initialt
- Uppskalning kan innehåra ytterligare investeringar

En stor samling av modeller finns tillgängliga på Hugging Face: <https://huggingface.co>, som också fungerar som ett community för utveckling och implementering av AI.

## **10.3 Bygga en chattbot**

I följande avsnitt ska vi steg för steg bygga en enkel chattbot som körs i terminalen. Vi kommer att använda Googles Gemini-modell som LLM.

**i** För att kunna använda Gemini behöver vi en API-nyckel, vilket vi kan få genom att logga in på <https://aistudio.google.com/> och skapa en nyckel.

För att inte dela med sig av sin privata API-nyckel brukar man vanligtvis spara den som en miljövariabel och läsa in den med `os.getenv()`.

Vi installerar biblioteket `google-genai` via pip. Vi kan också använda exempelvis conda eller uv.

```
> pip install google-genai
```

Vi börjar med att kolla så att vi kan få Gemini att generera svar.

```
import os
from google import genai

client = genai.Client(api_key=os.getenv("API_KEY"))
```

① Skapa en instans av `Client`-klassen från `genai`-biblioteket.

```
response = client.models.generate_content(
    model="gemini-2.0-flash",
    contents="Hej där, vem pratar jag med?"
)
print(response.text)
```

① Generera ett svar med `generate_content()`-metoden från vår `client`-instans.

Hej! Du pratar med en stor språkmodell, tränad av Google. Du kan kalla mig för Google AI, eller bara en AI. Vad kan jag hjälpa dig med?

Med hjälp av en `while`-loop och Pythons `input()`-funktion kan vi nu skapa en enkel textbaserad chattbot som vi kan köra i terminalen. Koden nedan demonstrerar detta och Figur 10.1 visar hur det kan se ut.

```
# chatbot.py
```

```

import os
from google import genai

client = genai.Client(api_key=os.getenv("API_KEY"))

if __name__ == "__main__":
    print("*** Gemini chat ***")
    print("Type <q> to exit chat.")
    while True:
        prompt = input("User: ")
        if prompt == "q":
            break
        else:
            response = client.models.generate_content(
                model="gemini-2.0-flash",
                contents=prompt
            )
        print("Gemini: " + response.text)

```

```

(chatbot) linus@magnolia:~/documents/kapitel/chatbot$ python chatbot.py
*** Gemini chat ***
Type <q> to exit chat.
User: Hej, vem pratar jag med?
Gemini: Jag är en stor språkmodell, tränad av Google.

```

*Figur 10.1: En enkel chattbot i terminalen.*

## 10.4 *Retrieval Augmented Generation (RAG)*

Om vi vill att vår Gemini-chattbot ska hålla sig till ett visst ämne kan vi använda en teknik som kallas *RAG* (*Retrieval Augmented Generation*). Det går ut på att vi ger modellen en kontext, ofta ett eller flera dokument eller liknande, att förhålla sig till. I en prompt säger vi åt modellen att enbart svara utifrån den givna kontexten. Om modellen inte hittar svaret i kontexten ska den säga det istället för att försöka hitta svaren någon annanstans eller gissa.

En *RAG*-modell innehåller alltså två delar.

Den första delen är en *retriever*, som söker efter relevanta stycken i en större text. Dessa stycken skickas sedan vidare som kontext till den andra delen, en *generator* som genererar svaren utifrån den givna kontexten.

För att ge modellen en kontext behöver vi läsa in data (exempelvis ett eller flera PDF-dokument) och bearbeta den så att vi kan göra en *semantisk sökning* i datan, det vill säga leta upp de stycken i kontexten som verkar ha mest med själva frågan att göra. Dessa stycken skickar vi sedan med till språkmodellen när vi ställer vår fråga.

Det finns ett antal ramverk för att implementera *RAG*, bland annat LangChain. Vi kommer inte använda något ramverk i det här kapitlet utan istället använda oss av vanligt förekommande Python-bibliotek som pypdf för att läsa in text från ett PDF-dokument och numpy för beräkningar samt en enkel *vector store* för att hantera vår data.

I resten av det här kapitlet kommer vi att använda texten från det här kapitlet som kontext när vi ställer frågor till vår chattbot.

Kodexemplet som följer bygger på kod från ett Github-repository som är en mycket bra källa till vidare läsning och experimentering. Repositoryt kan hittas på <https://github.com/FareedKhan-dev/all-rag-techniques>.

#### 10.4.1 Läs in data

*RAG*-tekniken kan användas med många olika typer av data. I det här exemplet kommer vi arbeta med text-data.

Vi läser in ett PDF-dokument och extraherar all text i dokumentet.

```
from pypdf import PdfReader  
  
reader = PdfReader("chattbot.pdf")  
  
text = ""  
  
for page in reader.pages:  
    text += page.extract_text()
```

## 10.4.2 Chunking

Just nu är vår variabel `text` en enda lång textsträng som innehåller hela kapitlet. Vi behöver dela upp den i mindre delar, som brukar kallas *chunks*. Det är för att vi så småningom kommer att söka efter de delar av texten som innehåller den information vi är intresserade av, så kallad semantisk sökning.

Det finns ett antal olika strategier för *chunking*. Vi kommer inte gå igenom dem alla i det här kapitlet utan fokuserar på några av de vanligaste: *fixed-length chunking*, *sentence-based chunking*, och *semantic chunking*.

### *Fixed-length chunking*

En vanlig strategi är den som kallas *fixed-length chunking*. Vi delar då upp texten i ett antal lika stora delar baserat på *tokens*, ord eller tecken. Det är en effektiv och enkel strategi, men den görs utan hänsyn till innehållet i texten. Det kan leda till att vi tappar information och att våra semantiska sökningar blir sämre.

När vi använder *fixed-length chunking* låter vi vanligtvis delarna överlappa med ett antal tecken. Hur stora delarna ska vara och hur många tecken som ska överlappa beror på typen av text och är något vi kan behöva prova oss fram till när vi bygger en *RAG*-modell.

I kodexemplet nedan skapar vi *chunks* som innehåller 1000 tecken vardera, med 200 teckens överlappning mellan delarna.

```
chunks = []
n = 1000
overlap = 200

for i in range(0, len(text), n - overlap):
    chunks.append(text[i:i + n])

print(f"Antal chunks: {len(chunks)}")
```

Antal chunks: 30.

### *Sentence-based chunking*

Ett alternativ till *fixed-length chunking* är *sentence-based chunking*. Vi delar upp texten i meningar, till exempel genom Pythons `split()`-funktion.

```
sentences = text.split(". ")

print(f"Antal meningar: {len(sentences)}")
```

Antal meningar: 62.

Vi återkommer till *semantic chunking* längre fram.

I det här kapitlet kommer vi använda *fixed-length chunking*.

#### 10.4.3 *Embeddings*

Vi behöver dock köra vår textdata genom ytterligare ett steg innan den är redo att användas som kontext till vår *RAG*-modell. Vi behöver skapa *embeddings*, det vill säga numeriska representationer av texten. *Embeddings* fångar textens betydelse, vilket gör att *chunks* som ligger närmare varandra i betydelse också har *embeddings* som ligger närmare varandra rent matematiskt.

Vi definierar en funktion `create_embeddings()`, som använder `embed_content()`-metoden från vår `client`-instans.

```
from google.genai import types

def create_embeddings(text,
    model="text-embedding-004",
    task_type="SEMANTIC_SIMILARITY"):
    return client.models.embed_content(
        model=model,
        contents=text,
        config=types.EmbedContentConfig(task_type=task_type))
```

- ① Ange text, vilken modell vi vill använda samt i vilket syfte vi vill skapa *embeddings*.  
② Funktionen kör i sin tur metoden `embed_content()`.

```
embeddings = create_embeddings(chunks)
```

Våra *embeddings* lagras som vektorer. I Avsnitt 10.5 skapar vi en enkel *vector store* för att hantera våra *chunks* och *embeddings*.

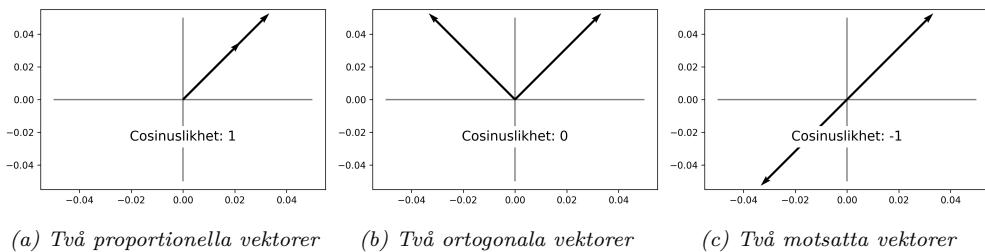
#### 10.4.4 Semantisk sökning

När vi har skapat våra *embeddings* är vi redo att göra semantiska sökningar för att hitta de *chunks* som verkar ligga närmast vår fråga i betydelse.

Det finns lite olika algoritmer vi kan använda oss av när vi jämför texter i en semantisk sökning. En av de vanligaste är cosinuslikheten (*cosine similarity*), som beskriver vinkeln mellan två vektorer. Cosinuslikheten är ett tal i intervallet  $[-1, 1]$ . Figur 10.2 visar exempel på tre olika par av vektorer med olika grad av cosinuslikhet:

- proportionella med cosinuslikheten 1
- ortogonala (vinkelräta) med cosinuslikheten 0
- motsatta med cosinuslikheten -1.

Notera att de vektorer som utgörs av våra *embeddings* har många fler dimensioner än de som visas i Figur 10.2.



Figur 10.2: Exempel på cosinuslikheter.

Vi kan skriva en funktion som räknar ut cosinuslikheten enligt exemplet nedan.

```
import numpy as np

def cosine_similarity(vec1, vec2):
    return np.dot(vec1, vec2) / (np.linalg.norm(vec1) *
                                 np.linalg.norm(vec2))
```

Den här funktionen kan vi nu använda för att utföra en semantisk sökning bland våra *embeddings*.

```

def semantic_search(query, chunks, embeddings, k=5):
    query_embedding =
        create_embeddings(query).embeddings[0].values
    similarity_scores = [] ①

    for i, chunk_embedding in enumerate(embeddings.embeddings):
        similarity_score = cosine_similarity(
            query_embedding,
            chunk_embedding.values
        )
        similarity_scores.append((i, similarity_score))

    similarity_scores.sort(key=lambda x: x[1], reverse=True)
    top_indices = [index for index, _ in similarity_scores[:k]]

    return [chunks[index] for index in top_indices]

```

- ① `embed_content()` returnerar ett `EmbedContentResponse`-objekt som innehåller olika data. Vi är bara intresserade av själva värdena på våra `embeddings`, vilka vi hittar i `embeddings`-attributet.

Vi får ut de fem `chunks` som har störst semantisk likhet med vår `query`. Vi kollar på det första resultatet.

Fråga: Vad innebär chunking?

Svar:

ka data. Vi är bara intresserade av själva värdena på våra `embeddings`, vilka vi hittar i `embeddings`-attributet.

Vi får ut de fem chunks som har störst semantisk likhet med vår query.

Vi kollar på det

första resultatet.

`query = "Vad innebär chunking?"`

`print(f"Fråga: {query}")`

`print("Svar:")`

`print(semantic_search(query, chunks, embeddings)[0])`

`query = "Vad innebär chunking?"`

`print(f"Fråga: {query}")`

`print("Svar:")`

`print(semantic_search(query, chunks, embeddings)[0])`

```
Fråga: Vad innebär chunking?  
Svar:  
vi  
hittar i embeddings-attributet.  
Vi får ut de fem chunks som har störst semantisk likhet med vår query.  
Vi kollar på det  
första resultatet.  
query = "Vad innebär chunking?"  
print(f"Fråga: {query}")  
print("Svar:")  
print(semantic_search(query, chunks, embeddings)[0])  
Fråga: Vad innebär chunking?  
Svar:  
12som innehåller olika data. Vi är bara intresserade av själva värdena  
på våra embeddings,  
vilka vi  
hittar i embeddings-attributet.  
Vi får ut de fem chunks som har störst semantisk likhet med vår query.  
Vi kollar på det  
första resultatet.  
query = "Vad innebär chunking?"  
print(f"Fråga: {query}")  
print("Svar:")  
print
```

### i *Semantic chunking*

Nu när vi känner till begreppet semantisk sökning kan vi nämna ytterligare en metod för *chunking*, nämligen *semantic chunking*.

*Semantic chunking* innebär att vi börjar med en *sentence based chunking*. Sedan skapar vi *embeddings* av våra meningar och jämför deras innehåll med en semantisk sökning. Utifrån resultaten skapar vi sedan *chunks* som innehåller meningar som ligger nära varandra i betydelse. Det är en bra metod om vi har flera olika dokument som tar upp samma saker.

#### 10.4.5 Generera svar

Vår semantiska sökning verkar fungera ganska bra, men den återger ju bara texten så som den står i kapitlet. Nu är vi redo att låta språkmodellen generera svar som låter mer naturliga.

Vi börjar med att definiera en *system prompt*, som talar om för språkmodellen att den bara får utgå från informationen i kontexten som vi kommer skicka med när vi ställer själva frågan i vår *user prompt*.

```
system_prompt = """Jag kommer ställa dig en fråga, och jag vill
    ↵ att du svarar
baserat bara på kontexten jag skickar med, och ingen annan
    ↵ information.

Om det inte finns nog med information i kontexten för att svara på
    ↵ frågan,
säg "Det vet jag inte". Försök inte att gissa.
Formulera dig enkelt och dela upp svaret i fina stycken. """
```

Nu kan vi definiera en funktion, `generate_user_prompt()`, som tar en fråga, skapar en kontext genom en semantisk sökning med vår fråga, och bakar ihop allting till en enda prompt som vi kan skicka till språkmodellen.

```
def generate_user_prompt(query):
    context = "\n".join(semantic_search(query, chunks,
        ↵ embeddings))

    user_prompt = f"Frågan är {query}. Här är kontexten:
    ↵ {context}."

    return user_prompt
```

Nu kan vi ställa en fråga till språkmodellen. Vi definierar ytterligare en funktion, `generate_response()`.

```
def generate_response(system_prompt, user_message,
    ↵ model="gemini-2.0-flash"):
    response = client.models.generate_content(
```

```
    model=model,
    config=types.GenerateContentConfig(
        system_instruction=system_prompt),
        contents=generate_user_prompt(user_message)
    )
return response
```

```
print(generate_response(system_prompt, "Vad är semantic
↪ chunking?").text)
```

Semantic chunking är en metod för chunking som börjar med sentence based chunking. Sedan skapas embeddings av meningarna och deras innehåll jämförs med en semantisk sökning. Utifrån resultaten skapas sedan chunks som innehåller meningar som ligger nära varandra i betydelse.

#### 10.4.6 Evaluering

Hur vet vi om vår *RAG*-modell fungerar som vi vill? Vi kan naturligtvis ställa den ett antal frågor och själva bilda oss en uppfattning, men det är ofta en bättre idé att skapa ett system för att utvärdera modellen med kod.

Vi skriver ett antal frågor, samt svar på frågorna som är i linje med vad vi önskar att modellen svarar. Sedan kan vi med hjälp av en annan *system prompt* låta modellen utvärdera om svaren på frågorna är i linje med det vi önskar, och sätta ”betyg” på svaren. Det ger oss data att jämföra olika modeller med.

Vi börjar med att skriva ett antal testfrågor, och de svar vi önskar att modellen ska ge oss. En testfråga kan se ut som i exemplet nedan.

```
validation_data = [
    {
        "question": "Vilka delar utgör en RAG-modell?",
        "ideal_answer": "En RAG-modell innehåller två delar: en
↪ retriever som söker efter relevanta stycken i en text,
↪ och en generator som genererar svar utifrån den givna
↪ kontexten."
    }
]
```

```
]  
  
validation_data[0]["question"]  
  
'Vilka delar utgör en RAG-modell?'  
  
validation_data[0]["ideal_answer"]  
  
'En RAG-modell innehåller två delar: en retriever som söker efter  
relevant stycken i en text, och en generator som genererar svar utifrån  
den givna kontexten.'
```

Nu skriver vi en ny *system prompt* som talar om för språkmodellen att vi vill att den ska utvärdera svaret.

```
evaluation_system_prompt = """Du är ett intelligent  
utvärderingssystem vars uppgift är att utvärdera en  
AI-assistents svar.  
Om svaret är väldigt nära det önskade svaret, sätt poängen 1. Om  
svaret är felaktigt eller inte bra nog, sätt poängen 0.  
Om svaret är delvis i linje med det önskade svaret, sätt poängen  
0.5. Motivera kort varför du sätter den poäng du gör.  
"""
```

```
query = validation_data[0]["question"]  
response = generate_response(system_prompt, query)  
  
evaluation_prompt = f"""Fråga: {query}  
AI-assistentens svar: {response.text}  
Önskat svar: {validation_data[0]['ideal_answer']}"""  
  
evaluation_response = generate_response(evaluation_system_prompt,  
   evaluation_prompt)  
  
print(evaluation_response.text)
```

Poäng: 1

Motivering: Svaret är korrekt och relevant i förhållande till frågan samt följer instruktionerna.

Vi kan också ge modellen en fråga som ligger utanför den givna kontexten och försäkra oss om att den inte försöker hitta på ett svar.

```
validation_data[2]
```

```
{'question': 'Hur många bultar finns det i Ölandsbron?',  
'ideal_answer': 'Det vet jag inte.'}
```

```
query = validation_data[2]["question"]  
response = generate_response(system_prompt, query)  
  
evaluation_prompt = f"""Fråga: {query}  
AI-assistentens svar: {response.text}  
Önskat svar: {validation_data[2]['ideal_answer']}"""  
  
evaluation_response = generate_response(evaluation_system_prompt,  
   evaluation_prompt)  
  
print(evaluation_response.text)
```

Poäng: 1

Motivering: AI-assistentens svar är korrekt och i linje med det önskade svaret, vilket indikerar att den följer instruktionerna om att svara "Det vet jag inte" när information saknas i kontexten.

## 10.5 Vector store

I det här kapitlet har vi arbetat med en liten mängd data. I fall där kontexten består av många dokument, bilder, filmer eller ljudfiler kan det ta lång tid att generera *embeddings*. Då är det en bra idé att spara dem till en fil. Det gör det också möjligt att dela våra *embeddings* med andra, eller att använda en molntjänst för att generera dem och sedan ladda ned dem och arbeta med dem lokalt på vår egen dator.

Det är vanligt att man använder en vektordatabas för att lagra *embeddings*. Exempel på vektordatabaser är faiss, qdrant och ChromaDB.

För enklare projekt är det dock inte nödvändigt att använda en vektordatabas. Vi kan implementera en enkel *vector store* med `numpy`, och använda datahanteringsbiblioteket `polars` för att spara våra *embeddings*, tillsammans med deras metadata, i Parquet-format. Vi använder `polars` istället för `pandas` på grund av att `polars` har bättre stöd för nästlad data, alltså kolumner som innehåller exempelvis listor.

```
import polars as pl

class VectorStore:
    def __init__(self):
        self.vectors = []
        self.texts = []
        self.metadata = []

    def add_item(self, text, embedding, metadata=None):
        self.vectors.append(np.array(embedding))
        self.texts.append(text)
        self.metadata.append(metadata or {})

    def semantic_search(self, query_embedding, k=5):
        if not self.vectors:
            return []

        query_vector = np.array(query_embedding)

        similarities = []
        for i, vector in enumerate(self.vectors):
            similarity = np.dot(query_vector, vector) /
        ↵ (np.linalg.norm(query_vector) * np.linalg.norm(vector))           ①
            similarities.append((i, similarity))

        similarities.sort(key=lambda x: x[1], reverse=True)

        results = []
        for i in range(min(k, len(similarities))):
            idx, score = similarities[i]
            results.append({
                "text": self.texts[idx],
```

```

        "metadata": self.metadata[idx],
        "similarity": score
    })
}

return results

def save(self):
    df = pl.DataFrame(
        dict(
            vectors=self.vectors,
            texts=self.texts,
            metadata=self.metadata)
    )
    df.write_parquet("embeddings.parquet")

def load(self, file):
    df = pl.read_parquet(file, columns=["vectors", "texts",
    "metadata"])
    self.vectors = df["vectors"].to_list()
    self.texts = df["texts"].to_list()
    self.metadata = df["metadata"].to_list()

```

① Det här är `cosine_similarity()`-funktionen från tidigare.

Nu kan vi lagra våra `embeddings` i en instans av vår `VectorStore`-klass.

```

vector_store = VectorStore()

for i, chunk in enumerate(chunks):
    chunk_embedding =
    create_embeddings(chunk).embeddings[0].values
    vector_store.add_item(
        text=chunk,
        embedding=chunk_embedding,
        metadata={"type": "chunk", "index": i}
    )

```

Vi använder metoden `semantic_search()` för att utföra semantiska sökningar bland våra `chunks`.

```
query = "Vad är en semantisk sökning?"  
query_embedding = create_embeddings(query).embeddings[0].values  
  
vector_store.semantic_search(query_embedding=query_embedding)
```

Med metoden `save()` kan vi lagra våra *embeddings*, tillsammans med texten och metadatan. Filen `embeddings.parquet` kan vi sedan till exempel dela med oss av till andra som vill använda våra *embeddings*.

```
vector_store.save()
```

Metoden `load()` låter oss läsa in en lagrad fil.

```
vector_store2 = VectorStore()  
vector_store2.load("embeddings.parquet")
```

Vår *vector store* kan vi nu använda istället för att generera kontext som vi kan skicka med till språkmodellen.

## 10.6 Vidare arbete med chattbottar

Det här kapitlet har gett en introduktion till hur vi bygger egna chattbottar. Den läsare som vill fördjupa sig och bland annat använda etablerade och välkända bibliotek kan undersöka till exempel *LangChain* för att hantera språkmodeller, och *DeepEval* för att evaluera dem.

## 10.7 Övningsuppgifter

Övningsuppgifter för kapitlet finns på bokens hemsida:  
[https://github.com/AntonioPrgomet/ai\\_tillaempad\\_ml](https://github.com/AntonioPrgomet/ai_tillaempad_ml)