

---

# 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 klustering	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).

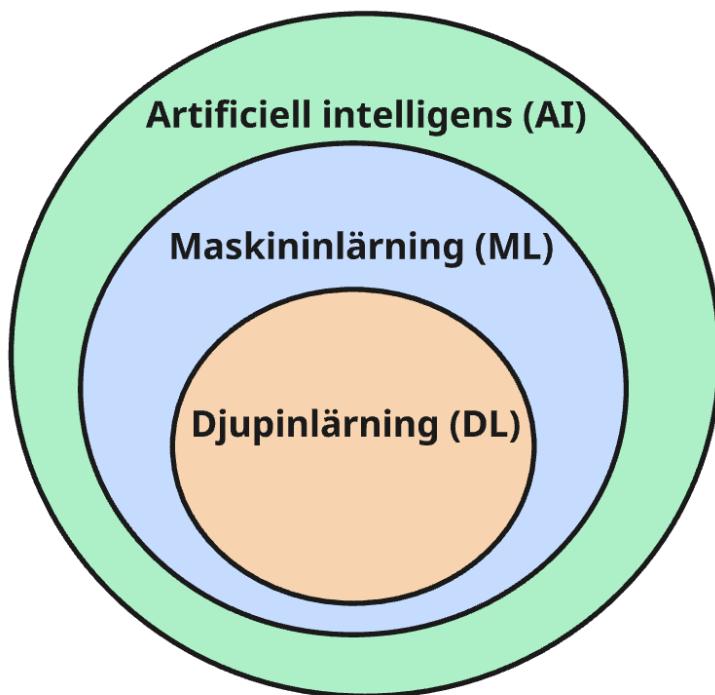
*“The science and art of programming computers so they can learn from data”*  
(Tom Mitchell, 1997).

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.

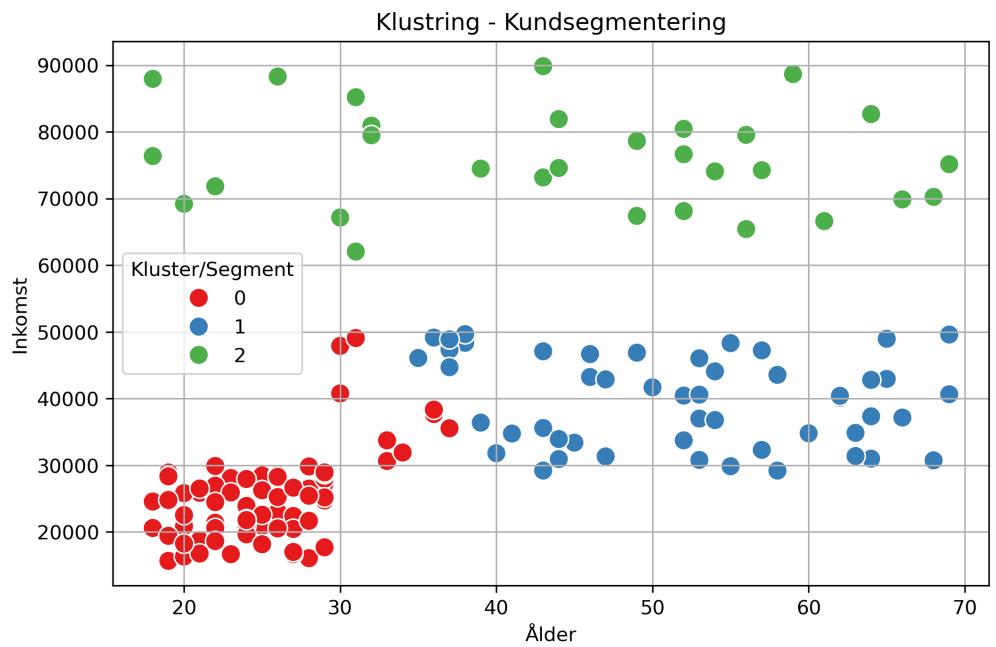
Inom *klustering* har vi endast indata och vill dela in denna i olika grupper/kluster. Ett tillämpningsområde inom detta är kundsegmenteringsmodeller. Exempelvis i Figur 1.2 kan vi tänka oss att kunder från de olika klustren/segmenten kan ha olika preferenser och köpkraft. Vilka preferenser de har kan 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.

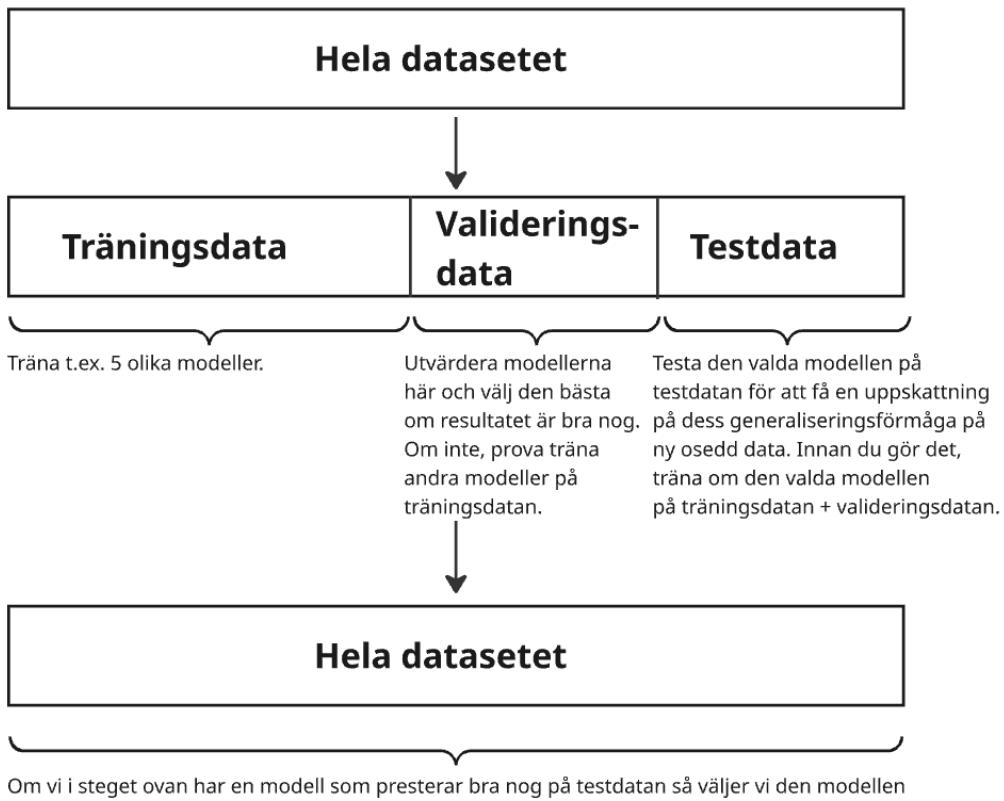
- Om modellen som testats på testdatan bedöms vara tillräckligt bra för att börja användas/produktionssättas så ska modellen tränas om på hela datasetet, det vill säga träningsdelen, valideringsdelen och testdelen ihopslagen. Det är också anledningen till att vi ska träna om modellen på träningsdelen 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



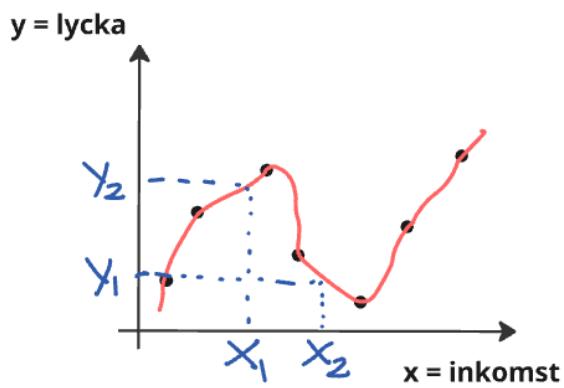
Figur 1.3: En schematisk illustration av hur träningsdata, valideringsdata och testdata används.

(*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.

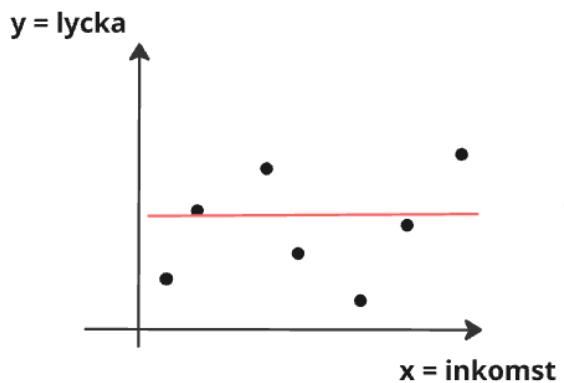
### **i En modell är en förenkling av verkligheten**

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.

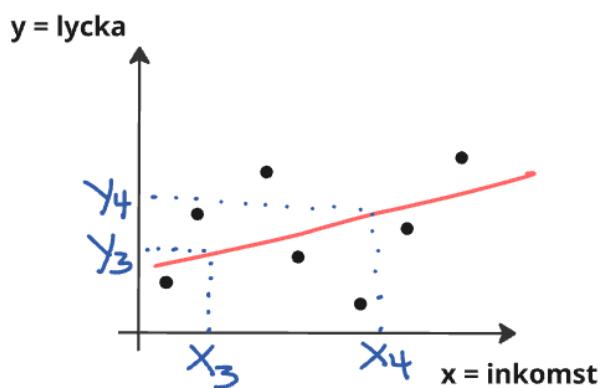
Varför testar vi en modell på testdatan? Räcker det inte att modellen utvärderas på valideringsdatan? Det faktum att man inom ML ofta skapar flera olika modeller och väljer den modell som presterar bäst på valideringsdatan gör att det av ren slump kan hända att modellen råkar prestera bra på valideringsdatan.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

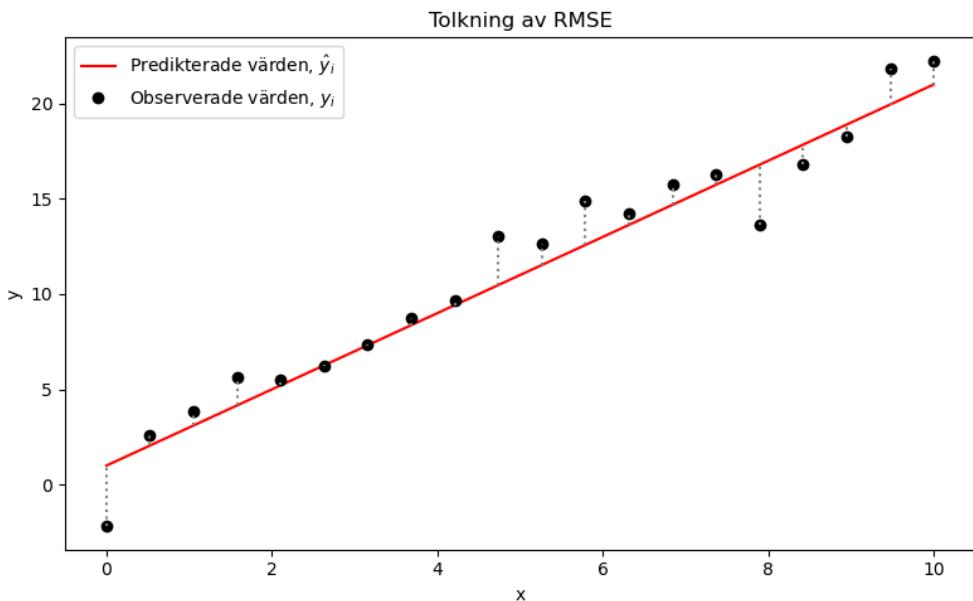
Vi går nu igenom formeln steg för steg.

- 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 användare 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

```

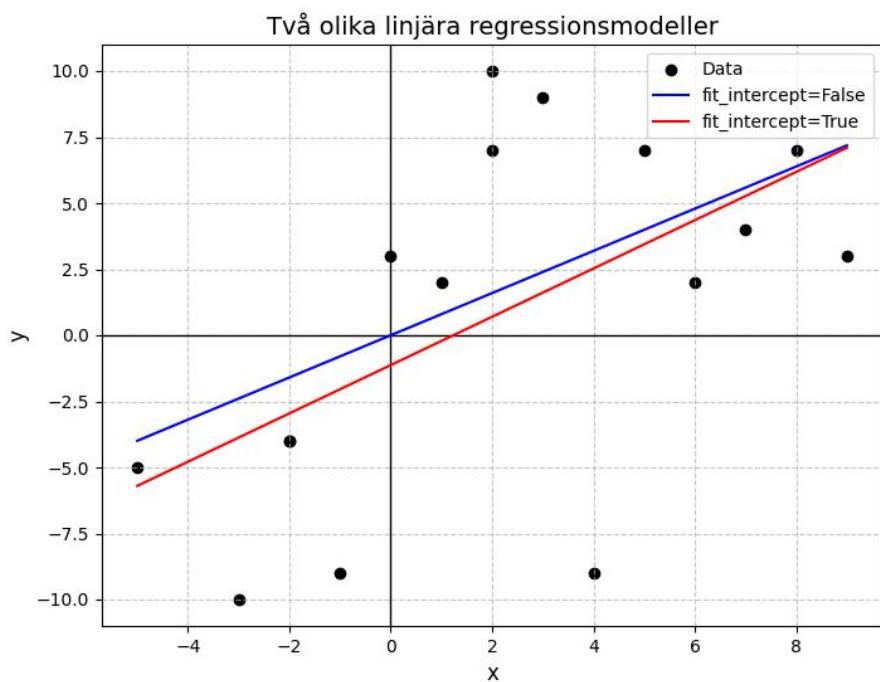
- ① Importera den linjära regressionsmodellen från *scikit-learn* biblioteket.
- ② Skapa ett dataset som vi kommer använda för demonstrationssyfte.
- ③ Instantiera en linjär regressionsmodell där hyperparametern `fit_intercept` är satt till `False`.
- ④ Träna modellen på den skapade exempeldatan.
- ⑤ 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.]

```

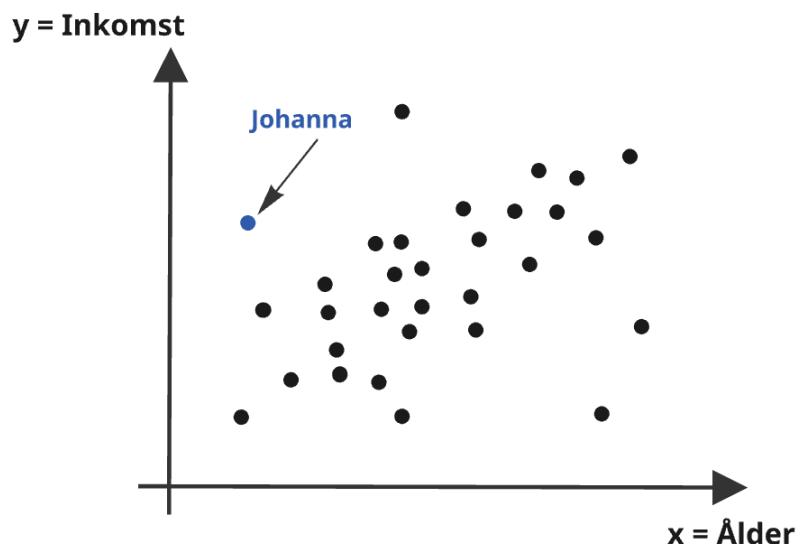
Det går att standardisera variabler på fler sätt, exempelvis kan den intresserade läsaren läsa *scikit-learn* dokumentationen för *MinMaxScaler* som gör att variablerna befinner sig i ett visst intervall, exempelvis mellan 0 – 1. Inom ML förekommer det också att variabler logaritmeras (log-transformeras), 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

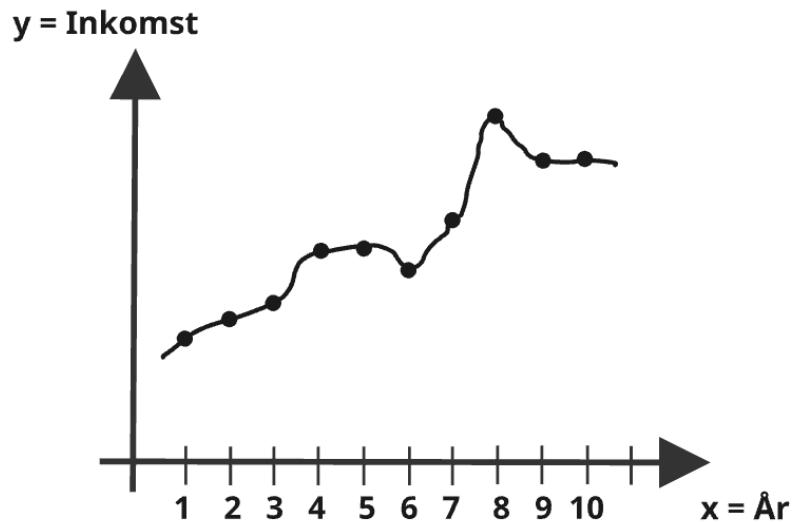
Data kan kategoriseras på olika sätt. Bra kategorier som är bra att känna till är *tvärnittsdata*, *tidsseriedata* och *paneldata*.

- *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.

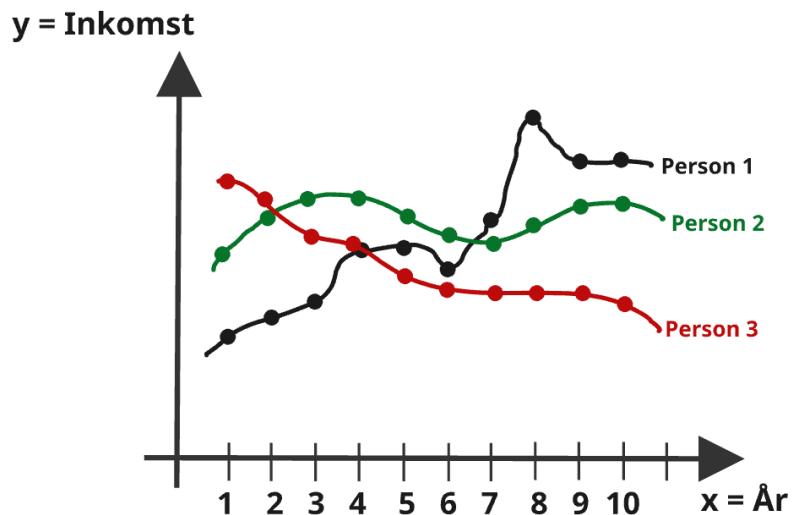
I boken kommer vi mestadels arbeta med tvärnittsdata, bland annat när vi tillämpar regressionsmodeller och klassificeringsmodeller. Tidsseriedata kommer vi mycket översiktligt nämna i Kapitel 9. Paneldata kommer vi inte gå in på i denna bok. Det finns mycket information på bland annat internet där analysmetoder 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.

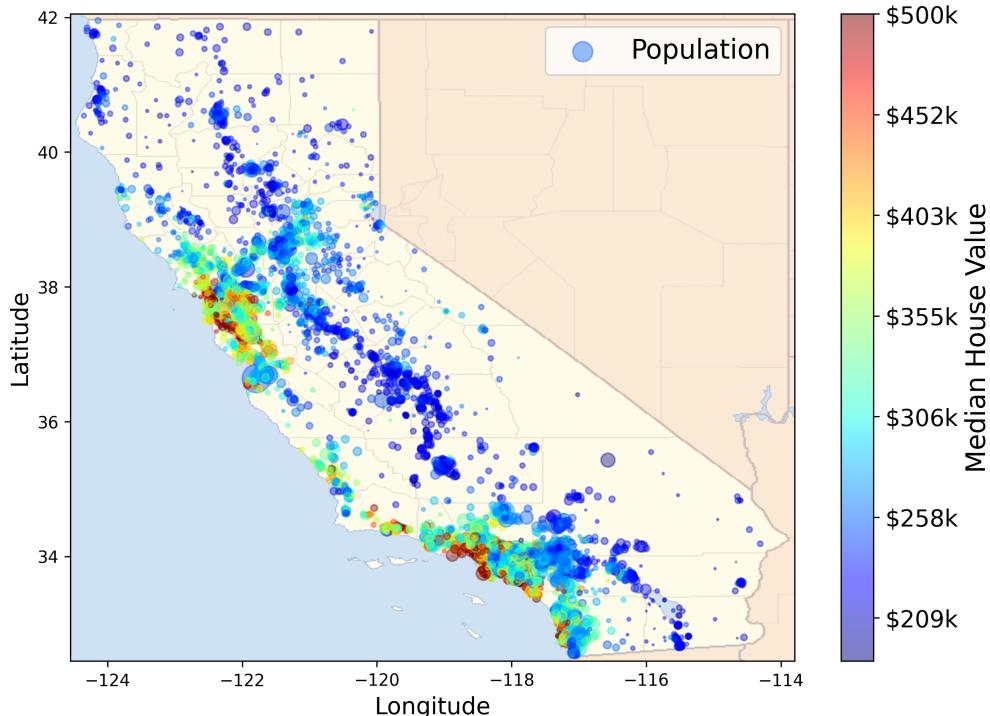
```
import matplotlib.image as mpimg
california_img=mpimg.imread('california.png')                                     ①
ax = train.plot(kind="scatter", x="longitude", y="latitude",
                 figsize=(10,7),
                 s=train['population']/100, label="Population",
                 c="median_house_value",
                 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(["${:dk}{}".format(v/1000) 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*”

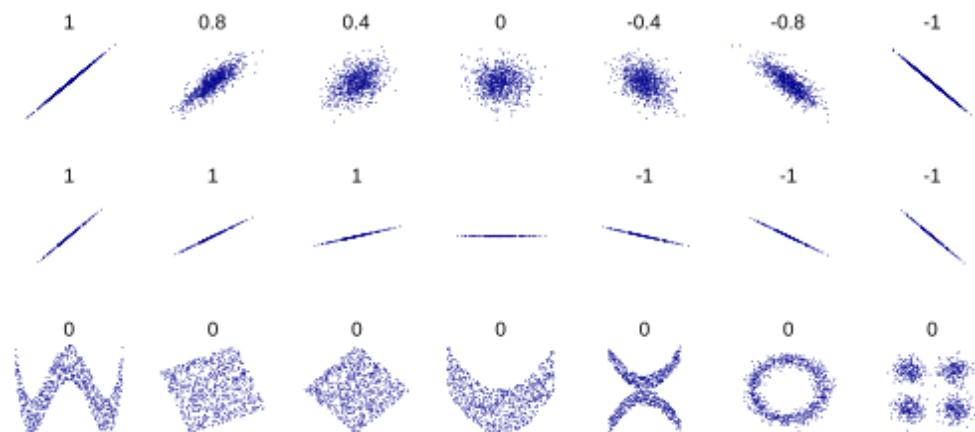
som finns på bokens hemsida.



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

```

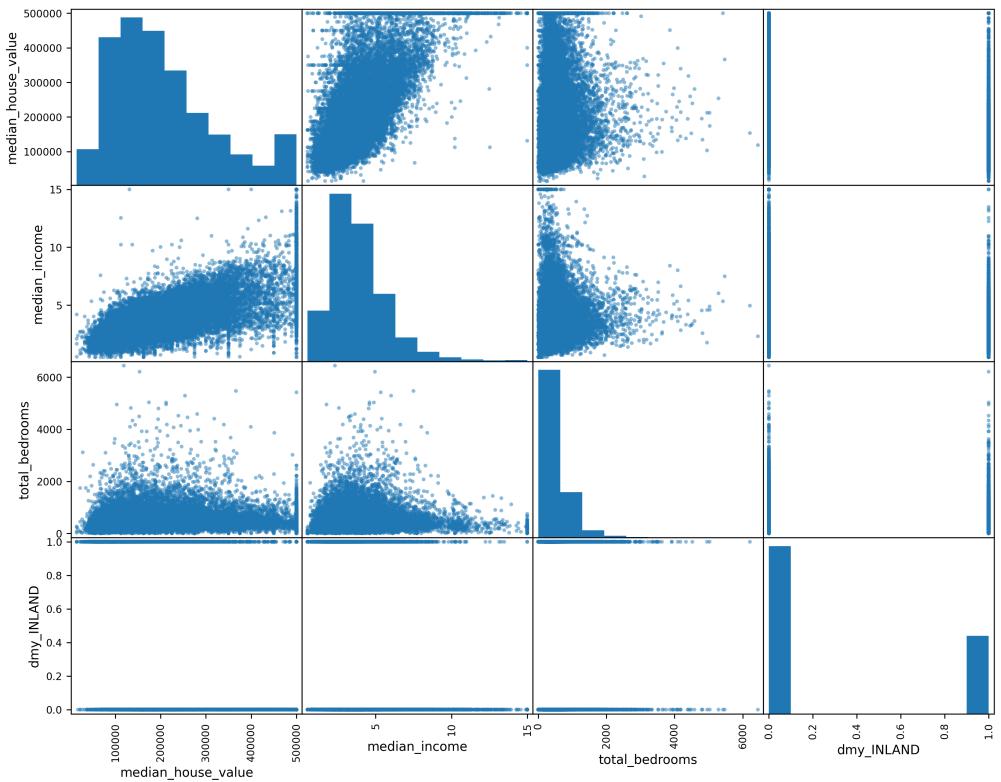
I tabellen ovan har vi beräknat korrelationskoefficienten mellan `median_house_value` och övriga kolumner. Föga förvånande ser vi exempelvis att korrelationskoefficienten mellan `median_house_value` och `median_income` är positiv och relativt nära +1. Det är rimligt att ökad inkomst leder till dyrare 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()

```

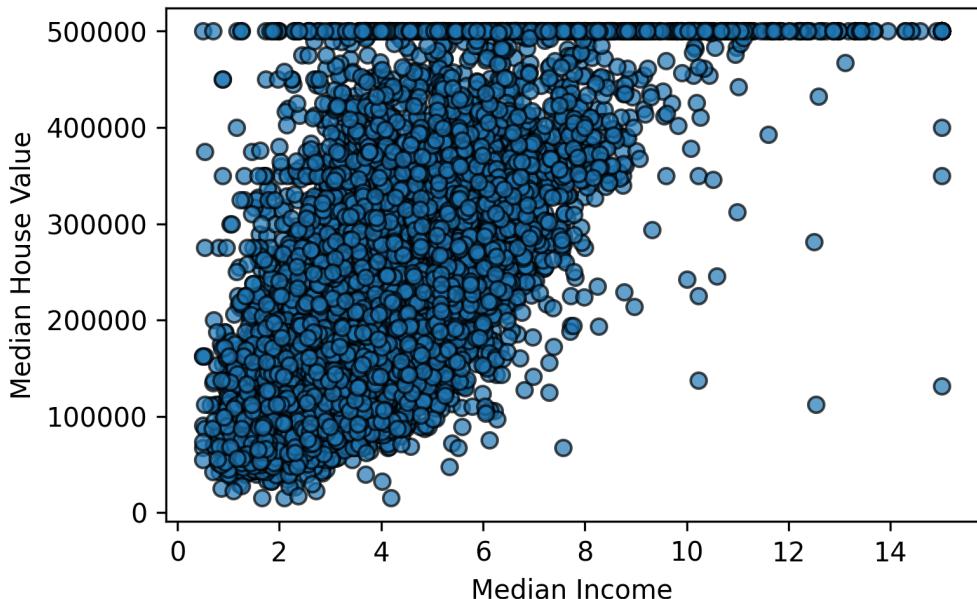


Är vi nyfikna på sambandet mellan inkomst och medianvärden på husen så kan vi göra en separat visualisering enligt nedan för att få en ännu tydligare bild. Från figuren ser vi tydligt att det är ett positivt samband mellan inkomst och husvärde. Men vi ser också något mer som är intressant. Medianhuspriserna 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 distrikten 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])

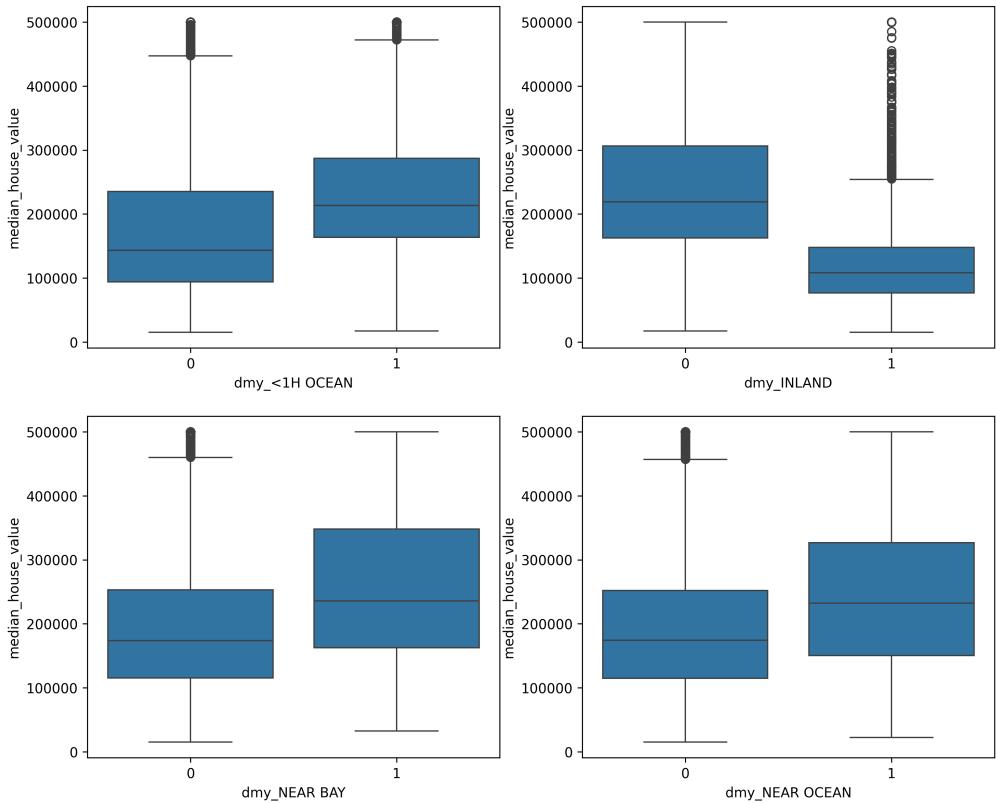
```

```

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”.

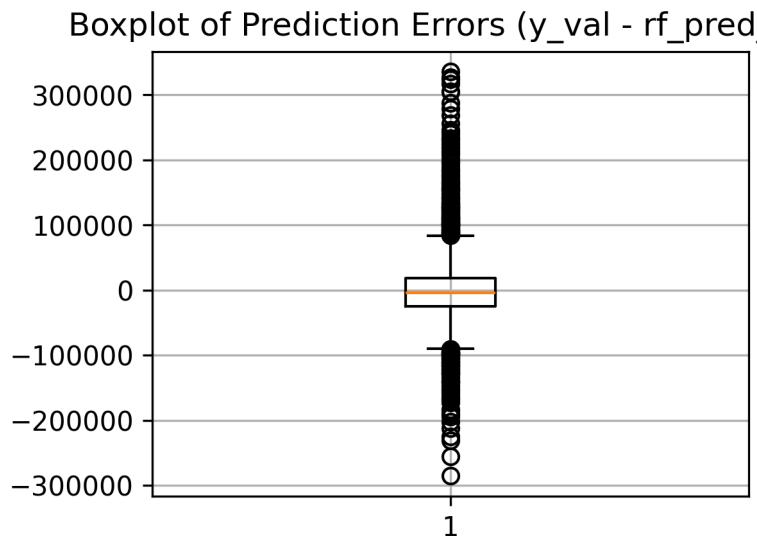
207517.1138589618

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.

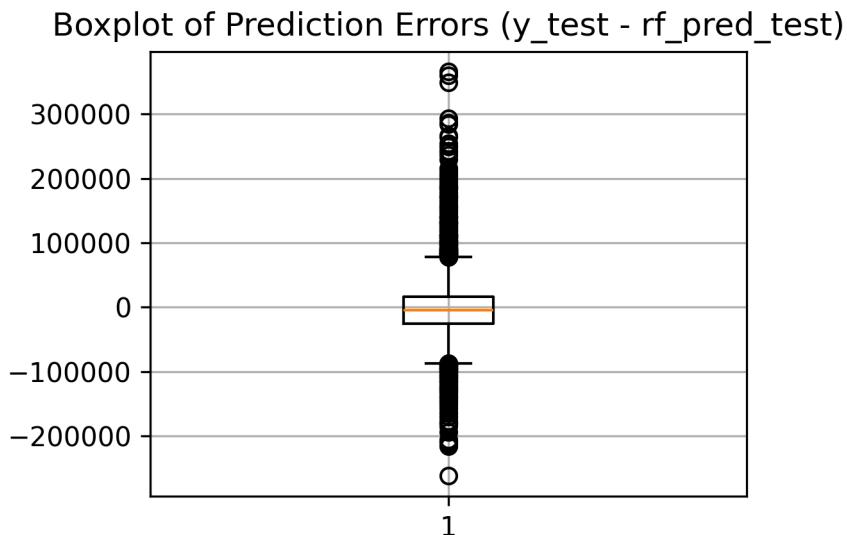
```
rf_pred_test = best_rf.predict(X_test)
rmse_test = root_mean_squared_error(y_test, rf_pred_test)
print('RMSE Random Forest on Test data:', rmse_test)
print(rmse_test/np.mean(y_test))
```

```
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']②

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]
})①

predicted_values = saved_model.predict(new_districts)②

for i, value in enumerate(predicted_values, start=1):
    print(f"Predicted median house value for district {i}:
        ${value:.2f}")③

```

- 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.

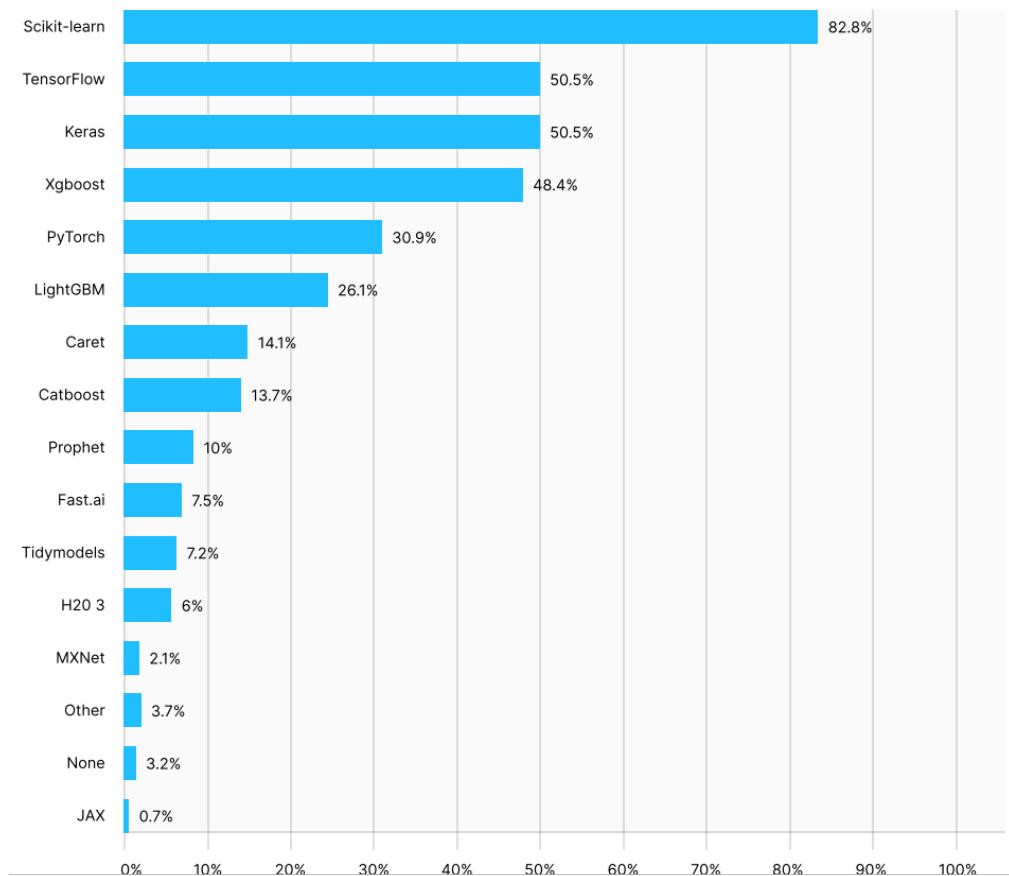
- För lite data. Många tillämpningsområden inom maskininlärning kräver tusentals observationer för att en modell ska kunna tränas till att prestera tillräckligt bra. I vissa fall, exempelvis inom bildanalys, kan det krävas miljontals bilder för att modellen ska bli praktiskt användbar. Oavsett hur skickliga 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 tränar 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 polynomkombinationer 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()

```

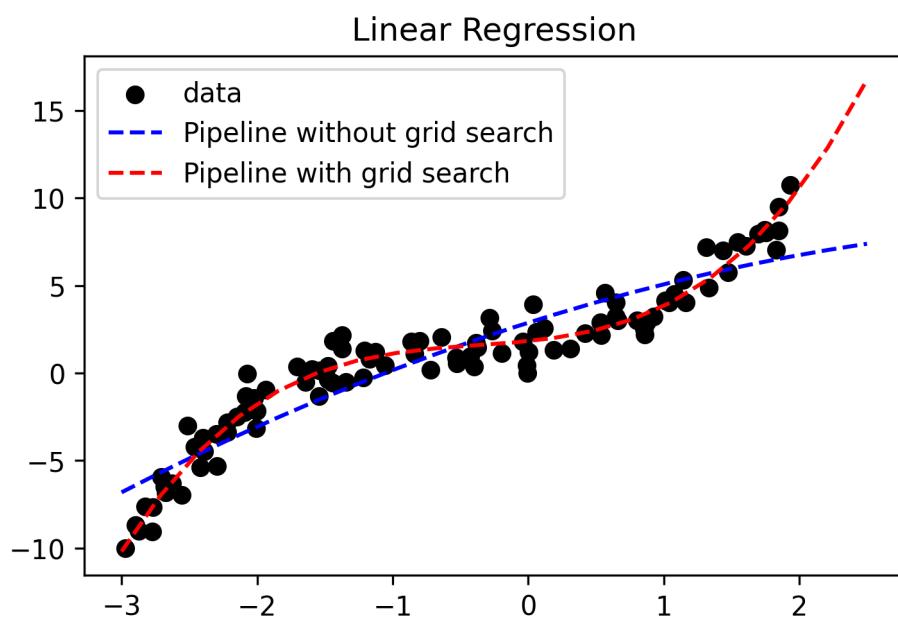
- ① Vi importerar de bibliotek som används.
- ② *Ridge* är en regulariserad regressionsmodell där regulariseringssstyrkan styrs av hyperparametern *alpha*. Vi kommer lära oss om denna modell i Avsnitt 3.3.4.
- ③ Vi skapar en *pipeline* där den sista *estimatore*n är en *predictor*.
- ④ Vi kollar vilka 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**