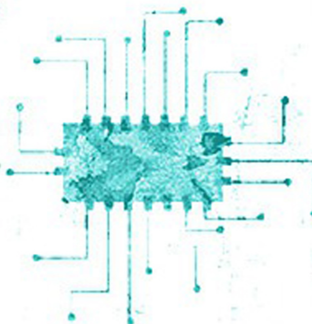
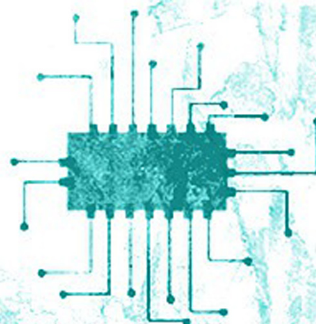
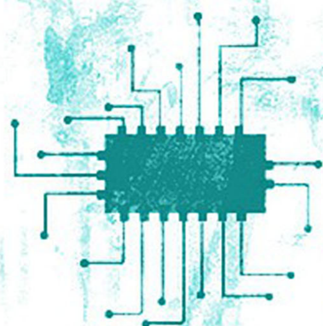
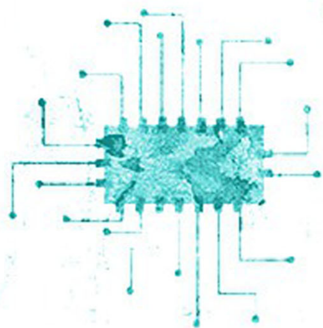
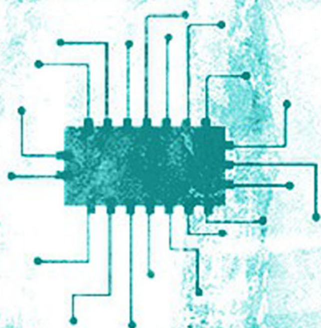
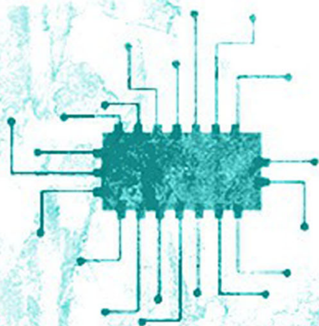
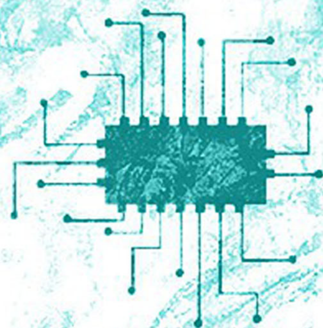
Sašo Karakatič
Iztok Fister Ml.



STROJNO UČENJE



S Pythonom do prvega klasifikatorja

Univerza v Mariboru

Fakulteta za elektrotehniko, računalništvo in informatiko

Strojno učenje

S Pythonom do prvega klasifikatorja

Avtorja

Sašo Karakatič

Iztok Fister Ml.

Januar 2022

Naslov *Title*
Strojno učenje
Machine Learning

Podnaslov *Subtitle*
S Pythonom do prvega klasifikatorja
Classification in Python

Avtorja *Authors*
Sašo Karakatič
(Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko)

Iztok Fister Ml.
(Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko)

Recenzija *Review*
Niko Lukač
(Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko)

Branko Kavšek
(Univerza na Primorskem, Fakulteta za matematiko, naravoslovje in informacijske tehnologije)

Lektoriranje *Language editing*
Nuša Grah

Tehnična urednika *Technical editors*
Sašo Karakatič
(Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko)

Jan Perša
(Univerza v Mariboru, Univerzitetna založba)

Oblikovanje ovitka *Cover designer*
Jan Perša
(Univerza v Mariboru, Univerzitetna založba)

Grafika na ovitku *Cover graphic*
Chenspec s Pixabay.com, CC0

Založnik *Published by*
Univerza v Mariboru, Univerzitetna založba
Slomškov trg 15, 2000 Maribor, Slovenija
https://press.um.si, zalozba@um.si

Izdajatelj *Issued by*
Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko
Koroška cesta 46, 2000 Maribor, Slovenija
http://feri.um.si, feri@um.si

Izdaja *Edition*
Prva

Vrsta publikacije *Publication type*
E-knjiga

Izdano *Published*
Maribor, Slovenija, januar 2022

Dostopno na *Available at*
http://press.um.si/index.php/ump/catalog/book/643

© **Univerza v Mariboru, Univerzitetna založba**
/ *University of Maribor, University Press*

Besedilo/*Text* © Karakatič, Fister Ml., 2022

Delo je pod licenco Creative Commons Priznanje avtorstva 4.0 Mednarodna.
/ *Work is licensed under the Creative Commons Attribution 4.0 International License.*

Uporabnikom je dovoljeno tako nekomercialno kot tudi komercialno reproduciranje, distribuiranje, dajanje v najem, javna priobčitev in predelava avtorskega dela, pod pogojem, da navedejo avtorja izvirnega dela.

Vsa gradiva tretjih oseb v tej knjigi so objavljena pod licenco Creative Commons, razen če to ni navedeno drugače. Če želite ponovno uporabiti gradivo tretjih oseb, ki ni zajeto v licenci Creative Commons, boste morali pridobiti dovoljenje neposredno od imetnika avtorskih pravic.

https://creativecommons.org/licenses/by/4.0/

CIP - Katadložni zapis o publikaciji
Univerzitetna knjižnica Maribor

004.85(075.8)(0.034.2)

KARAKATIČ, Sašo
Strojno učenje [Elektronski vir] : s Pythonom do prvega klasifikatorja / avtorja Sašo Karakatič, Iztok Fister Ml. - 1. izd. - E-publikacija. - Maribor : Univerza v Mariboru, Univerzitetna založba, 2022

Način dostopa (URL): https://press.um.si/index.php/ump/catalog/book/643

ISBN 978-961-286-560-3

doi: 10.18690/um.feri.1.2022

COBISS.SI-ID 94628867

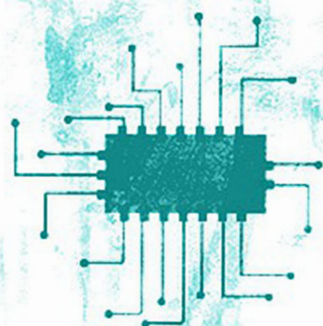
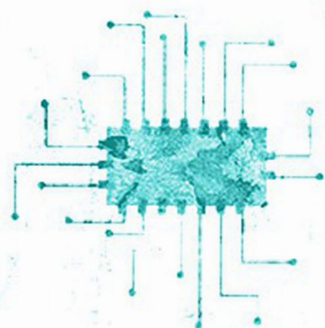
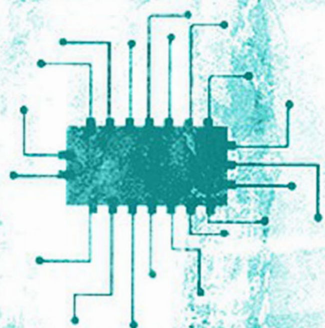
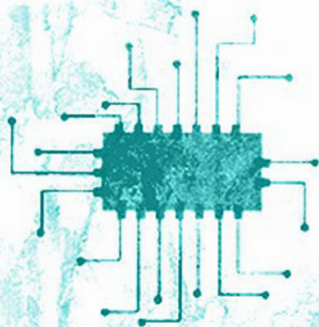
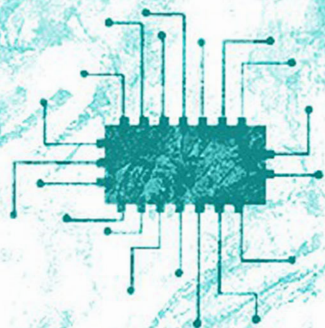
ISBN 978-961-286-560-3 (pdf)

DOI https://doi.org/10.18690/um.feri.1.2022

Cena / *Price* Brezplačni izvod

Odgovorna oseba založnika / *For publisher* prof. dr. Zdravko Kačič, rektor Univerze v Mariboru

Citiranje / *Attribution* Karakatič, S. in Fister Ml., I. (2022). *Strojno učenje: s Pythonom do prvega klasifikatorja.* Maribor: Univerzitetna založba. doi: 10.18690/um.feri.1.2022

Kazalo

1 Uvod **1**

2 Vzpostavitev okolja **7**
2.1 Razvojno okolje Jupyter Notebook 7
2.2 Uporaba oblačnega okolja 9
2.3 Namestitev in uporaba lokalnega okolja Anaconda . . 10
2.4 Nalaganje podatkov 24

3 Klasifikacija **27**
3.1 Model znanja 27
3.2 Ekspertni sistem 30
3.3 Inteligentni sistem 33
3.4 Podatki 35
3.5 Strojno učenje 35
3.6 Klasifikacija 38
3.7 Matematična definicija pojmov 39

4 Prvi klasifikator **41**
4.1 Računanje razdalj 42
4.1.1 Kreacija indikacijskih atributov v Pythonu . . . 44
4.2 Skupna razdalja 48
4.2.1 Evklidska razdalja 48
4.2.2 Mahattanska razdalja 49

4.2.3 Kosinusna razdalja 50
4.3 Klasifikator k najbližjih sosedov 51
4.3.1 Delovanje k najbližjih sosedov 51
4.3.2 Standardizacija podatkov 54
4.3.3 Določitev razreda podane instance 59
4.4 Uporaba k najbližjih sosedov v Pythonu 61
4.4.1 Vizualizacija podatkov 64
4.4.2 Učenje in uporaba modela znanja 65
4.4.3 Vpliv nastavitev na rezultate 67
4.4.4 Vpliv standardizacije podatkov na rezultate . . 72
4.4.5 Enovit primer klasifikacije več instanc 73
4.5 Utrjevanje znanja 75

5 Kakovost klasifikacije **85**
5.1 Delitev podatkov 87
5.1.1 Učna in testna množica 87
5.1.2 Validacijska množica 89
5.1.3 Navzkrižna validacija 92
5.2 Metrike klasifikacije 95
5.2.1 Binarna klasifikacija 96
5.2.2 Klasifikacija v več razredov 104
5.3 Utrjevanje znanja 109

6 Rešitve nalog **113**
6.1 Poglavje Prvi klasifikator 113
6.2 Poglavje Kakovost klasifikacije 128

7 Zaključek in kako naprej **145**

Literatura **149**

Stvarno kazalo **157**

1 | Uvod

Strojno učenje, umetna inteligenca, podatkovna znanost, podatkovno rudarjenje in velepodatki (ali kar veliki podatki). V zadnjih letih so to precej uporabljeni izrazi, ki jih zasledimo v novicah o revolucionarnih premikih na področjih obdelave podatkov, razpoznave slik ter videoposnetkov in obdelave besedila. S temi izrazi podjetniki zvito opišejo svoje storitve in produkte, da jim dvignejo vrednost ali jih lažje prodajo. S temi izrazi razvijalci programske opreme opišejo svoje izdelke, ko so se ti zmožni prilagajati na različne situacije. Pa gre res za revolucionarne metode ali le za trenutno modo oziroma modne besede (angl. *buzzwords*)? So algoritmi strojnega učenja res nekaj, kar izraža inteligentnost računalnika, ali pa gre le za kombinacijo `if` stavkov in statističnih metod?

Žal ni enovitega odgovora. Inteligentnega in samostojnega stroja, kot so prikazani v nekaterih filmskih uspešnicah, ne moremo pričakovati še nekaj časa, če sploh kdaj. Vsekakor pa imajo pristopi strojnega učenja vrednost, kar kaže tudi vztrajnost zanimanja in njihove uporabe tako v industriji kakor tudi raziskovalni sferi. Strojno učenje je danes ključen sestavni del številnih komercialnih informacijskih sistemov in raziskovalnih projektov na različnih področjih — od medicinske diagnoze in zdravljenja [1], avtomatskega trgovanja z vrednostnimi papirji [2], sa-

movozečih avtomobilov [3], pa do priporočil na družbenih omrežjih in spletnih trgovinah [4]. Mnogi menijo, da je uporaba strojnega učenja primerna za uporabo le v velikih podjetjih z obsežnimi raziskovalnimi skupinami in globokimi žepi. S to knjigo ti avtorja želiva pokazati, kako enostavna je uporaba strojnega učenja, če le že poznaš osnove programiranja.

Kot na drugih področjih, tudi pri strojnem učenju obstajajo tako kompleksni in napredni pristopi, kakor tudi enostavni. Za razumevanje najbolj naprednih pristopov (na primer globokega učenja (angl. *deep learning*)) res potrebujemo že kar nekaj semestrov matematike. Po drugi strani pa razumevanje najenostavnejših pristopov zahteva le izkušnje iz programiranja.

Seveda se največ govori o najbolj naprednih metodah, ki terjajo ekipo dvestotih podatkovnih znanstvenikov (kot se imenujemo) in za dve košarkarski dvorani velik računalnik. V resnici pa večji del informacijskih sistemov, ki so podprti s strojnim učenjem, uporablja zelo elementarne tehnike. Vsakdo pa že ne potrebuje naprednega algoritma, ki identificira človeka iz treh pikslov. V večji meri so za obogatitev obstoječih sistemov dovolj enostavni pristopi, kar je razvidno iz uporabe umetne inteligence v praksi. Nova storitev, katere razvijalci poudarjajo, da je podprta z naprednimi pristopi umetne inteligence, res nima le nekaj pogojnih `if`-ov in osnovnih opisno statističnih izračunov, kljub temu pa navadno ni tako kompleksna, kot se mogoče zdi.

Komu je ta knjiga namenjena?

Ta knjiga služi kot uvod v področje strojnega učenja. Namenjena je tistim, ki že znajo programirati -– idealno v programskem jeziku Python, saj so primeri prikazani z uporabo tega jezika. Zakaj Python? Ker je to eden izmed najbolj uporabljenih programskih jezikov za namen uporabe strojnega učenja. Knjiga se osredotoča na uporabo Pythona in knjižnice `scikit-learn` [5], ki je osnovna knjižnica za uporabo strojnega učenja. Seveda se ne gre izogniti uporabi knjižnic

`NumPy` in `Matplotlib` oz. `pandas` in `seaborn` [6] -– ampak poudarek vsekakor ni na teh.

Zavestno je bila narejena odločitev, da se knjiga preveč ne osredotoča na matematični vidik strojnega učenja. Drži, matematične formule so še vedno prisotne, ampak le do te mere, da pomagajo (ne pa ovirajo) pri razumevanju posameznih pristopov.

Knjiga se osredotoči le na eno tehniko strojnega učenja — na klasifikacijo. Klasifikacija se predstavi na le enem, po mnenju avtorjev, najbolj intuitivnem algoritmu klasifikacije, imenovanemu k najbližjih sosedov. Ta algoritem je predstavljen nekoliko bolj podrobno, s čimer se ponazori, da ima vsak pristop strojnega učenja svoje posebnosti, ki ga naredijo bodisi primernega ali neprimernega za dan problem.

Komu ta knjiga ni namenjena?

V knjigi avtorja okvirno predstaviva različne tehnike in področja strojnega učenja -– podrobnosti pa izpustiva. Če te zanima regresija, katera izmed tehnik nenadzorovanega učenja ali delo z globokimi nevronskimi mrežami, ta knjiga ni zate. Vsak dober podatkovni znanstvenik je najprej spoznaval temelje strojnega učenja ter se šele potem posvetil naprednejšim tehnikam. Čemu nam bo napredna globoka nevronska mreža, če pa je ne znamo pravilno ovrednotiti? Knjiga je napisana tako, da bo samostojno nadaljevanje učenja kar se da enostavno.

V knjigi niso omenjene vse možne knjižnice strojnega učenja v Pythonu ali možnosti uporabe strojnega učenja v drugih programskih jezikih. Četudi te delo na strojnem učenju v Pythonu zanese k uporabi *PyTorcha* ali *TensorFlowa*, je *scikit-learn* osnova, brez katere ne gre. Četudi te bosta delodajalec ali lastna radovednost v prihodnosti usmerila v uporabo strojnega učenja v drugih jezikih, so tudi ostale knjižnice strojnega učenja spisane po zgledu knjižnice *scikit-learn*.

Po površnem pregledu knjige bi kdo morda prišel do zaključka, da knjiga pokrije snov, ki bi jo lahko predstavili v eni objavi na blogu. Drži – če bi snov, predstavljeno v tej knjigi, strnili v programsko kodo,

to ne bi bil kompleksen informacijski sistem, temveč le daljša Python skripta. Namen knjige ni, da predstavi čim več različnih načinov uporabe knjižnice `scikit-learn` in predela vse možne algoritme v tej – temu namreč služi dokumentacija te knjižnice. Skozi branje se od bralca te knjige pričakuje, da vsak korak (oziroma vsako vrstico kode) razume – kaj se zgodi in čemu je namenjena. Skozi poglobljeno razumevanje ti avtorja želiva predati sposobnost nadaljnjega samostojnega učenja – kaj je pomembno pri določenem pristopu, kako se uporabi ta pristop, kako nastavitve vplivajo na rezultate in tako naprej. Povedano drugače, s knjigo te avtorja želiva naučiti samostojnega učenja snovi strojnega učenja.

Struktura knjige in kje začeti

Uvodu sledijo štiri poglavja.

Poglavje 2 govori o vzpostavitvi okolja, ki bo primerno za uporabo knjižnic, ki jih knjiga vključuje, za zagon primerov iz knjige in za reševanje praktičnih nalog. Če imaš Python okolje že vzpostavljeno na svojem računalniku, lahko to poglavje preskočiš.

Poglavje 3 začne s splošno razlago definicije strojnega učenja in vzpostavi vzporednice tega z našim (človeškim) učenjem novega znanja. Večji del poglavja je namenjen predstavitvi glavne tehnike strojnega učenja v tej knjigi — klasifikaciji podatkov. Opisi in definicije gredo od najenostavnejše razlage na primerih pa do formalne matematične definicije pojmov. To poglavje preskoči, če si z osnovnimi tehnikami že seznanjen/-a in bi se želel/-a hitro posvetiti programiranju.

Poglavje 4 obravnava izbran algoritem klasifikacije v tej knjigi — k najbližjih sosedov. Algoritem je najprej predstavljen neodvisno od programiranja na način, ki ti bo omogočal, da ga znaš rešiti tudi na roko. Predstavljene so nekatere nastavitve tega algoritma, s čimer se predstavi pomembnost poznavanja podrobnosti algoritma, da je uporaba tega kar se da učinkovita. Opisu in

definicijam pa sledijo primeri v programski kodi. Ti primeri so napisani tako, da se lahko neposredno uporabijo pri testiranju in spoznavanju. Za utrjevanje znanja iz tega poglavja so priložene tudi štiri naloge -– dve preverjata teoretično razumevanje in se rešita na roko, dve pa sta programerskega tipa. Tega poglavja nikar ne preskoči, saj predstavlja osnovo, brez katere ne gre.

Poglavje 5 pa se dotakne pomembnega področja ovrednotenja kakovosti uporabljenih tehnik strojnega učenja. Strojno učenje nam prinese dodatno vrednost le, če deluje pravilno. Kako pa izmerimo, če deluje pravilno? V tem poglavju so predstavljene različne metrike kakovosti klasifikacijskih modelov in pristopi, k pravilni vzpostavi testnega okolja (in podatkov) za vrednotenje modelov. Ker je ovrednotenje kakovosti klasifikacijskih modelov ključnega pomena za razvoj uporabnih modelov, je tudi to poglavje priporočljivo za začetnike.

Dobrodošel oziroma dobrodošla v svet strojnega učenja.

2 | Vzpostavitev okolja

Za namen prikaza praktičnih primerov bo v tej knjigi uporabljen programski jezik Python, saj je en izmed bolj priljubljenih programskih jezikov z odličnim naborom knjižnic strojnega učenja. Za razumevanje primerov je vsekakor priporočljivo poznavanje tega programskega jezika ali vsaj splošno poznavanje enega izmed modernih programskih jezikov (Java, Perl, C#, R, C, C++). Sledijo navodila vzpostavitve okolja, primernega za izvedbo praktičnih primerov na svojem računalniku.

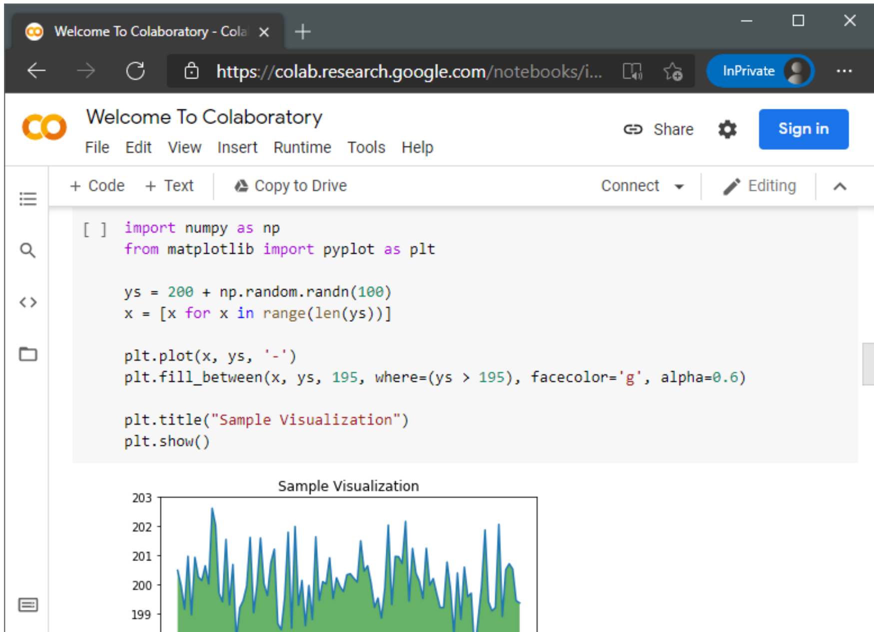
2.1 Razvojno okolje Jupyter Notebook

Vsi praktični primeri v knjigi so prikazani tako, da se brez težave izvedejo v JupyterLab [1] okolju. To razvojno okolje omogoča pisanje Python programske kode v tako imenovanih Jupyter zvezkih (angl. *Jupyter notebooks*).

Ti zvezki se urejajo v poljubnem spletnem brskalniku in zato za urejanje ne potrebujejo namenskega orodja. Posebnost kode, zapisane v Jupyter zvezkih, je, da je ta v eni datoteki (zvezku) mešana s ta-

[1] `http://jupyterlab.io/`

kojšnjimi izpisi kode in navodili, ki jih razvijalci zapišemo sproti za razlago delovanja kode. Zvezki so razdeljeni na tako imenovane celice, kjer je celica bodisi navodilo ali pa programska koda z izpisom rezultata, kot je prikazano na sliki 2.1.



Slika 2.1: Jupyter zvezek na spletni storitvi Google Colab.

Namen takega razvoja je, da je programska koda z navodili enostavno deljiva in razumljiva. Prav tako razvoj v zvezku omogoča inkrementalno zaganjanje kode, celico za celico – tako se zaženejo le želeni deli kode in ne celotna datoteka. To je idealen način razvoja za učenje in za raziskovanje. Jupyter zvezki so postali privzeti način razvoja na področju podatkovne znanosti, saj zadostijo primarnemu namenu podatkovne znanosti – odkrivanju vzorcev in pregledu podatkov skozi zgodbo. Tako je vsaka celica s kodo in priloženo celico navodil kar en del zgodbe, kjer celica z navodili opiše, kaj bo celica s kodo naredila

ter povzame njene rezultate.

Alternativnih razvojnih okolij je mnogo. Od popolnoma namenskih za programiranje v Pythonu, kot sta PyCharm [2] in Spyder [3], pa do splošnih razvojnih okolij, kot sta Visual Studio Code [4] in Atom [5].

2.2 Uporaba oblačnega okolja

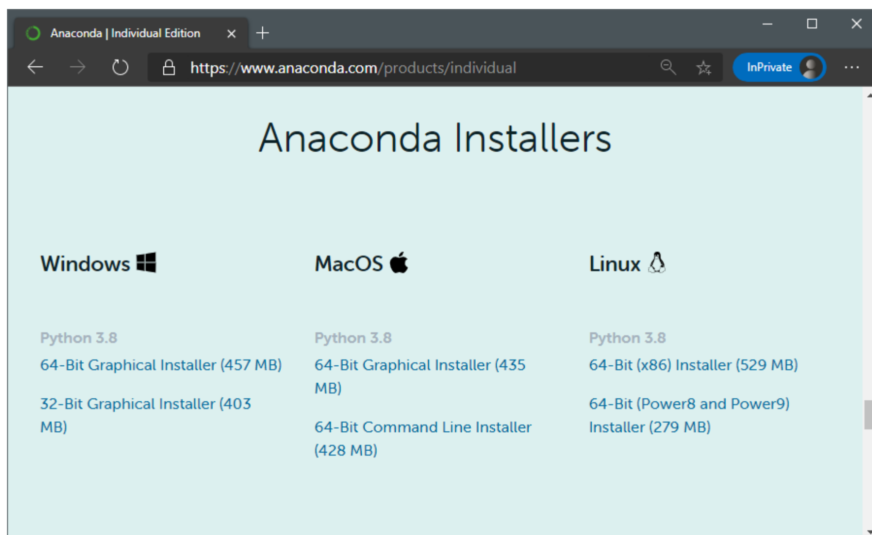
Prvi način uporabe Jupyter zvezkov za namen strojnega učenja je uporaba ene izmed ponujenih storitev. V ta namen so na voljo številni ponudniki z bodisi zastonjskimi ali plačljivimi storitvami izvajanja Python kode v oblaku. Dober primer take storitve je Google Colab[6], ki vsem svojim uporabnikom ponuja kreacijo Jupyter zvezkov in zagon teh na Googlovih strežnikih, do določene mere zastonj – v času pisanja knjige je uporaba Google Colab zvezkov za namen zagona primerov in nalog bila brezplačna. Google Colab je prikazan prav na sliki 2.1.

Konkurence na področju oblačnih zvezkov je ogromno in uporabnost teh se mesečno spreminja ter je vezana na ceno storitve, dostopnost slovenskim razvijalcem in nabor funkcionalnosti. Preprosto spletno iskanje "Jupyter Notebook service" ali "Google Colab alternative" vrne številne rezultate. V času pisanja knjige so med omembe vrednimi bile storitve drugih gigantov strojnega učenja Microsoft Azure Machine Learning[7] in Amazon SageMaker[8] ter s strojno opremo radodarna Paperspace[9] in Deepnote[10].

[2] `https://www.jetbrains.com/pycharm/`
[3] `https://www.spyder-ide.org/`
[4] `https://code.visualstudio.com/`
[5] `https://atom.io/`
[6] `https://colab.research.google.com/`
[7] `https://ml.azure.com/`
[8] `https://aws.amazon.com/sagemaker/`
[9] `https://www.paperspace.com/`
[10] `https://deepnote.com/`

2.3 Namestitev in uporaba lokalnega okolja Anaconda

V tej sekciji bo predstavljeno, kako vzpostavimo primerno okolje na svojem računalniku. Da namestimo Python in vse potrebne knjižnice, se bomo zatekli k okolju Anaconda, ki vsebuje tako Python kot skupek številnih knjižnic, ki se uporabljajo za namen analize podatkov. Na uradni spletni strani okolja Anaconda poiščemo predel za prenos različice, namenjene za posameznike. V času pisanja knjige se ta različica imenuje *Individual Edition* in je dosegljiva na spletni strani `https://www.anaconda.com/products/individual` ter prikazana na sliki 2.2.



Slika 2.2: Spletna stran Python okolja Anaconda.

Okolje Anaconda poleg programskega jezika Python namesti tudi številne knjižnice, namenjene za podatkovno analizo v tem programskem jeziku. Skozi knjigo bomo uporabili številne izmed teh:

- `NumPy`, ki se uporablja za upravljanje s podatki v matričnih in več-dimenzionalnih poljih. Podrobnejšega dela s to knjižnico ne bo, je pa vseeno dobro, da se zavedamo, da je to ena izmed ključnih knjižnic, ko imamo opravek s podatki in analizo teh. Ta knjižnica je tudi sestavni del sledeče knjižnice.
- `pandas` je knjižnica, ki je namenjena za manipulacijo in analizo podatkov v tabelarični obliki. Čeprav v zaledju uporablja knjižnico `NumPy`, so določene operacije nad podatki poenostavljene.
- `scikit-learn` je knjižnica, ki vsebuje implementacije različnih algoritmov, metrik in pristopov pred-procesiranja za strojno učenje. Algoritem klasifikacije, ki bo predstavljen v tej knjigi, je povzet po implementaciji iz te knjižnice.
- `Matplotlib`, ki se uporablja za prikaz grafov različnih vrst. Je zelo prilagodljiva, saj omogoča velik nabor različnih tipov vizualizacij podatkov in prilagoditve teh (tako stilno, kot vsebinsko). Je pa zaradi svoje prilagodljivost delo s to knjižnico nekoliko težje in zaradi tega uporabljamo ...
- `seaborn`, ki poenostavi prikaz grafov. V zaledju uporablja knjižnico `Matplotlib`, ampak preko svojih vmesnikov določene pogoste operacije pri risanju grafov poenostavi.

Uporaba Jupyter zvezkov

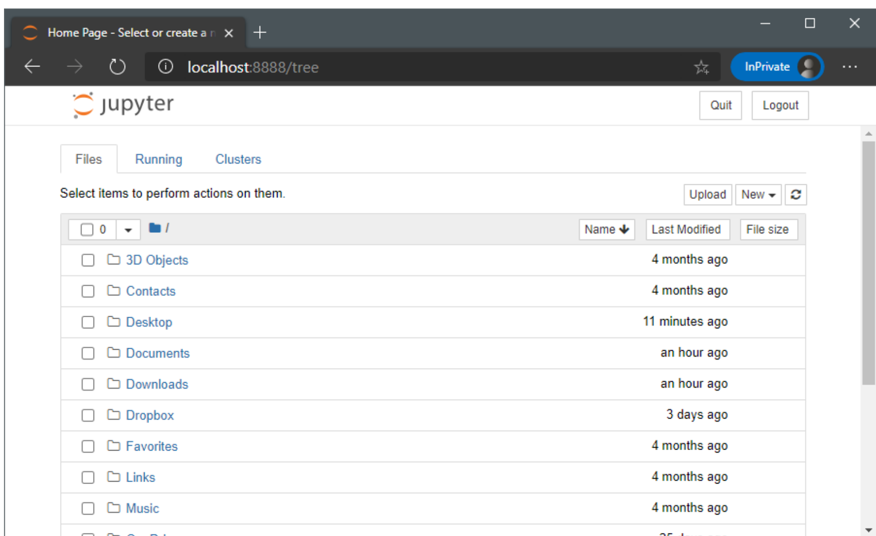
Za pisanje Jupyter zvezkov na lastnem računalniku je najprej potreben zagon okolja JupyterLab, za kar lahko uporabimo enega izmed dveh načinov. Prvi način je zagon JupyterLaba ročno iz komandne vrstice:

1. Zaženemo komandno vrstico. V Windowsih sta to bodisi Command Prompt ali PowerShell. V Linux in MacOS operacijskih sistemih pa je komanda vrstica največkrat pod imenom Terminal.
2. Zaženemo ukaz `jupyter lab`.

3. Odpre se okno brskalnika in pokaže se domača stran JupyterLab, kot je prikazano na sliki 2.3.

Drugi način je preko uporabniškega vmesnika okolja Anaconda.

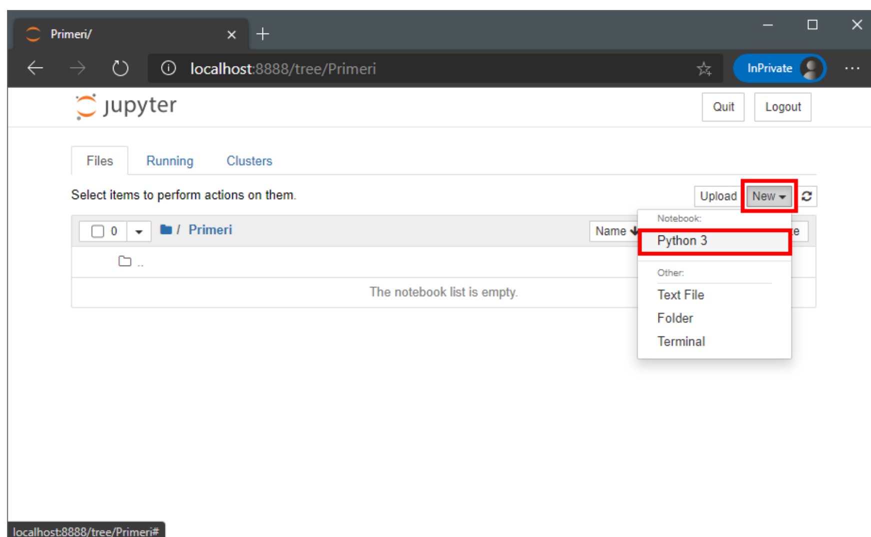
1. Zaženemo Anaconda Navigator.
2. V seznamu ponujenih aplikacij najdimo JupyterLab in ga s pritiskom možnosti *Launch* zaženemo.
3. Odpre se okno brskalnika in pokaže se domača stran JupyterLab, kot je prikazano na sliki 2.3.



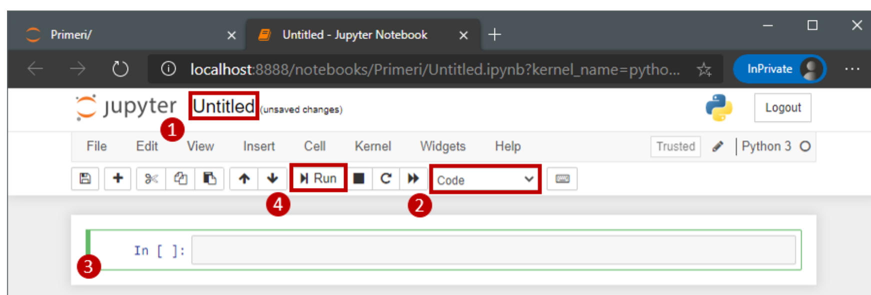
Slika 2.3: Domača stran JupyterLab okolja.

Po želji naredimo novo mapo, kamor shranimo naše zvezke in druge podporne datoteke. V tej mapi ustvarimo novi zvezek, kot prikazuje slika 2.4.

Odpre se novo okno s praznim zvezkom, kot je prikazano na sliki 2.5. Jupyter zvezke razvijamo preko spletnega vmesnika, kar je tudi razlog za prikaz spletnega brskalnika.



Slika 2.4: Kreacija novega zvezka v JupyterLab okolju.



Slika 2.5: Nov prazen Jupyter zvezek.

Slika 2.5 ima označene pomembne dele uporabniškega vmesnika. Oznaka **1** prikazuje naslov zvezka, kar s klikom na napis lahko spremenimo. Oznaka **2** prikazuje tip celice, ki je trenutno izbrana, oznaka **3** pa prikazuje trenutno izbrano in hkrati zaenkrat še edino celico.

Markdown v Jupyter zvezkih

Za začetek spremenimo tip prve celice na *Markdown*, ki je stil zapisa navodil in druge vsebine v celico. V celico zapišemo sledeč Markdown zapis.

```
# Prvi primer
V __prvem__ primeru bomo:
- Naredili prvi izpis `Python` kode.
- Naredili prvi graf s pomočjo _seaborn_ knjižnice.
```

S pritiskom na gumb zagona, ki je na sliki 2.5 označen s **4** (ali s pritiskom `Shift + Enter`), se izbrana celica izvede in izbere se (ter če ne obstaja se tudi ustvari) naslednja celica. Ker je izbrana celica bila tipa Markdown, se je zapisana koda oblikovala po stilu Markdown zapisa. Osnovni ukazi za zapis Markdowna so sledeči.

Z znakom `#`, čemur sledi presledek definiramo naslov.

- `#` za glavne naslove,
- `##` za podnaslove,
- `###` za tretji nivo naslovov, pa vse do šestega nivoja naslovov z `######`.

Besedilo lahko tudi poudarimo.

- *Poševno* zapisano besedilo: `_besedilo_` ali `*besedilo*` ter
- **krepko** zapisano besedilo: `__besedilo__` ali `**besedilo**`.

Dodamo lahko tudi neurejen seznam, tako da vsak element seznama zapišemo v svoji vrstici, začnemo pa ga bodisi z znakom `*` ali z `-`. Sezname lahko gnezdimo – ugnezdene elemente zamaknemo s tabulatorjem.

```
- Prvi element.
* Drugi element.
    - Prvi ugnezden element.
    - Pa še drugi.
```

Urejene seznami, kjer si elementi sledijo v vrstnem redu, pa ustvarimo tako, da pred elemente dodamo številko ali oznako vrstnega reda. Tudi take sezname lahko gnezdimo. Številko vrstnega reda lahko podamo poljubno, saj ni potrebno, da so te v pravem vrstnem redu.

```
1. Prvi element.
2. Drugi element.
    11. Prvi ugnezden element.
    12. Pa še drugi.
```

Besedilo v stilu kode pa v besedilo zapišemo med znakoma `‘`, ki ga na slovenski tipkovnici izberemo z `AltGr + 7`.

```
Spremenljivka `vrednost` in razred `podatki`.
```

Tabelo vstavimo vrstico po vrstico, vsaka celica v vrstici se začne in konča z znakom `|` (na slovenski tipkovnici `AltGr + W`). Tako je ena vrstica s tremi celicami zapisana na sledeč način.

```
|Prva celica|Druga celica|Tretja celica|
```

Črte med vrstice pa dodamo kar z znakom `-` namesto vsebine celice. Primer tabele s tremi stolpci in tremi vrsticami ter črto med prvo in drugo vrstico izgleda sledeče.

```
|Prva vrstica in prvi stolpec|Drugi stolpec|Tretji stolpec|
|-|-|-|
|Druga vrstica|12|-42|
|Tretja vrstica|-7|65|
```

Za lažjo razumevanje kode lahko to nekoliko oblikujemo s presledki.

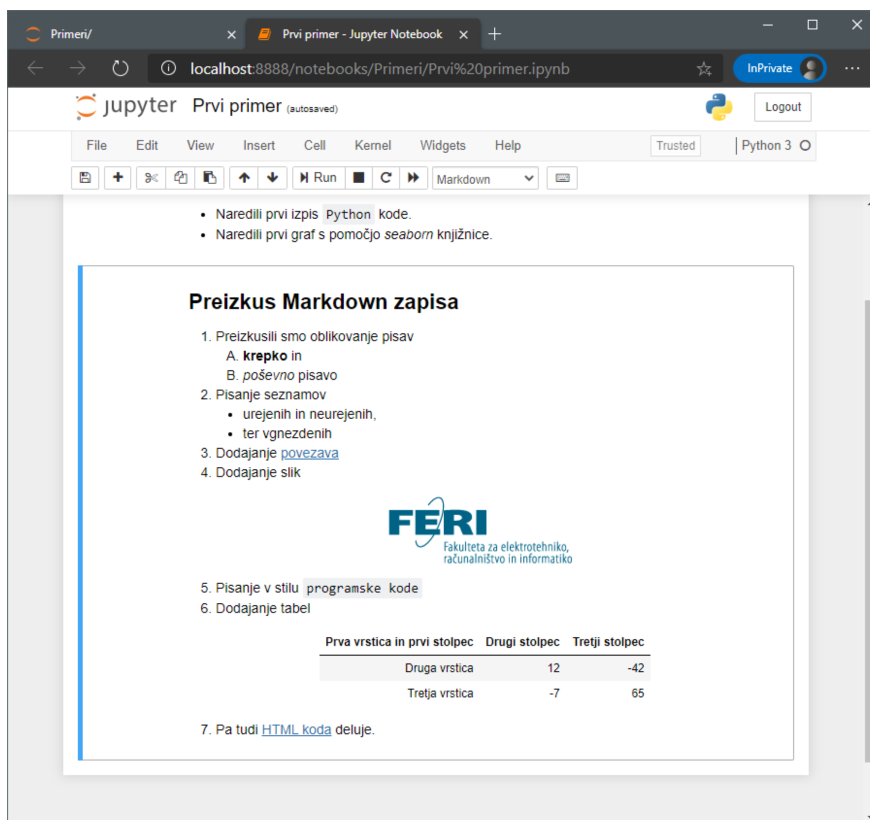
```
|Prva vrstica in prvi stolpec|Drugi stolpec|Tretji stolpec|
|----------------------------|-------------|--------------|
|Druga vrstica               |12           |-42           |
|Tretja vrstica              |-7           |65            |
```

Hiperpovezave vstavljamo s sledečo kodo `[Napis povezave](URL)`, slike pa z zapisom `![Naziv slike](URL do datoteke slike)` – razlika je v klicaju.

```
[Povezava do iskalnika Google](http://www.google.com/)
![Primer slike](/imgs/slika.png)
```

Če ne gre drugače, pa lahko vsebino celic navodil oblikujemo tudi s preprosto HTML kodo. HTML in Markdown zapis lahko po želji mešamo. Končen primer vsega skupaj bi zapisali s sledečo kodo, rezultat pa je prikazan na sliki 2.6.

```
## Preizkus Markdown zapisa

1. Preizkusili smo oblikovanje pisav
    1. __krepko__ in
    2. _poševno_ pisavo
2. Pisanje seznamov
    - urejenih in neurejenih,
    - ter ugnezdenih
3. Dodajanje [povezava](http://www.um.si/)
4. Dodajanje slik

![FERI](https://feri.um.si/site/images/logo-feri.png)

5. Pisanje v stilu `programske kode`
6. Dodajanje tabel

|Prva vrstica in prvi stolpec|Drugi stolpec|Tretji stolpec|
|----------------------------|-------------|--------------|
|Druga vrstica               |12           |-42           |
|Tretja vrstica              |-7           |65            |

7. Tudi <a href="https://feri.um.si/">HTML koda</a> deluje.
```



Slika 2.6: Rezultat Markdown zapisa.

Python koda v Jupyter zvezkih

Če ima celica vsebino tipa *Code*, pa se vsebina smatra kot Python koda – primerno je tudi obarvana, deluje avtomatsko dopolnjevanje kode (angl. *code auto-completion*) in rezultati kode se izpišejo kar takoj pod celico. Sledi primer zapisa programske kode v celico.

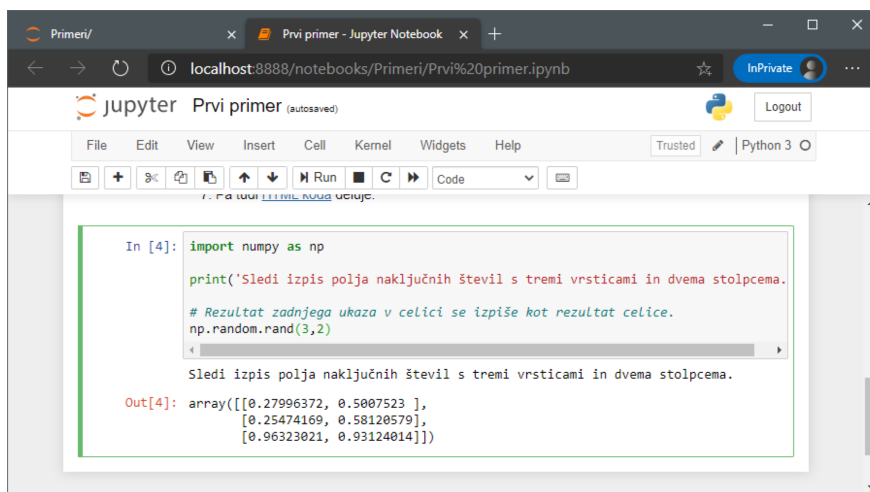
V prvi vrstici se uvozi knjižnica `NumPy` ter se poimenuje kot `np` za nadaljnjo uporabo. Sledi izpis niza znakov z ukazom `print()` – izpis

sledi kar pod celico, kot je prikazano na sliki 2.7. Tretja vrstica vsebuje komentar v kodi, ki opisuje zadnjo vrstico. Zadnja vrstica pa vsebuje klic metode `rand()` knjižnice `NumPy`, ki vrne polje naključnih števil po podanih velikostih (v primeru je podana velikost polja s tremi vrsticami in dvema stolpcema). Ker ta vrstica vsebuje tudi zadnji ukaz, ki vrača rezultat, se rezultat tega izpiše tudi pod celico.

```
import numpy as np

print('Sledi izpis polja naključnih števil '
      's tremi vrsticami in dvema stolpcema.')

# Rezultat zadnjega ukaza v celici se
# izpiše kot rezultat celice.
np.random.rand(3,2)
```

```
Sledi izpis polja naključnih števil s tremi vrsticami in dvema
stolpcema.
array(0.27687911, 0.19824952,
0.06910974, 0.7548379 ,
0.64441325, 0.11951306)
```

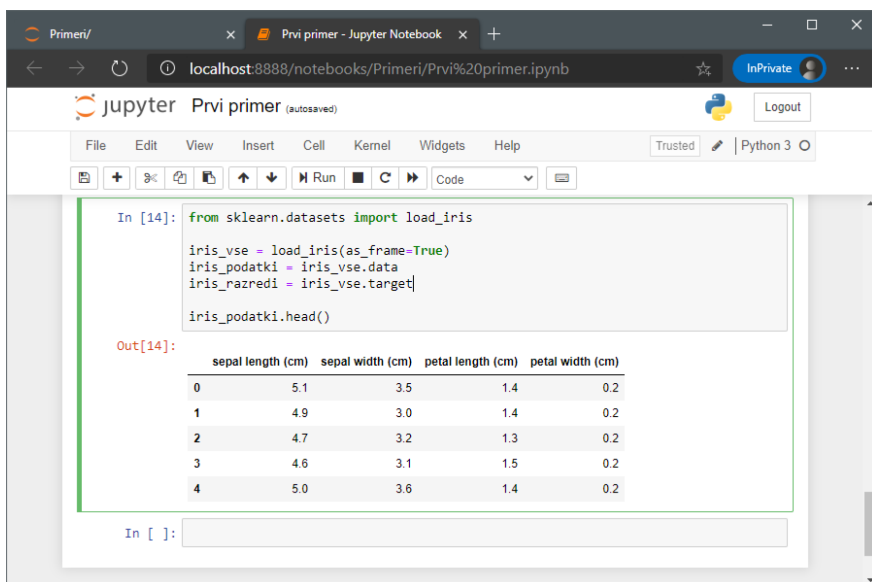
Preizkusimo še uporabo knjižnice `pandas`, ki se uvozi v prvi vrstici kode. Kot že omenjeno pri prvi predstavitvi knjižnice, ta v zaledju uporablja podatke v obliki `NumPy` polj. To dejstvo lahko izkoristimo pri kreaciji nove strukture podatkov – v `pandas`-u je to razpredelnica, poimenovana `DataFrame`. Sledeča koda ustvari polje naključnih števil s 100 vrsticami in dvema stolpcema, ki pa služi pri kreaciji razpredelnice `DataFrame`. V `DataFrame` razpredelnici lahko stolpce in vrstice poimenujemo, kot je prikazano s podanim parametrom `columns`. Zadnja vrstica kliče metodo `head()` naše instance `data_df` razreda `DataFrame`, ki vrne prvih pet vrstic te razpredelnice, kot tudi kaže slika 2.8.



Slika 2.7: Prvi preizkus zapisa Python kode v Jupyter zvezek.

```
import numpy as np
import pandas as pd

data_np = np.random.rand(100, 2)
data_df = pd.DataFrame(data_np,
                       columns=['Prvi stolpec',
                                'Drugi stolpec'])

data_df.head()
```

```
  Prvi stolpec Drugi stolpec
0 0.753358     0.029272
1 0.901894     0.846576
2 0.492876     0.533067
3 0.943245     0.538149
4 0.567161     0.537613
```



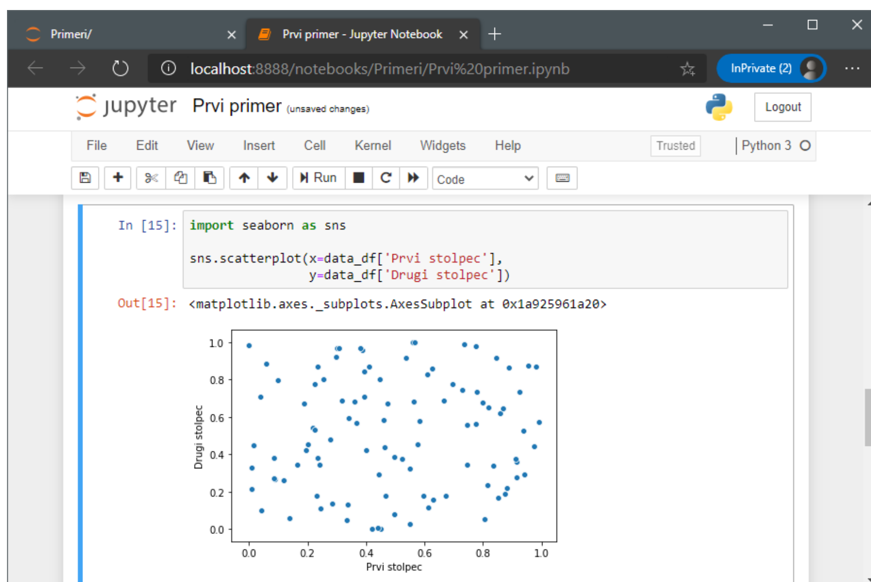
Slika 2.8: Preizkus delovanja knjižnice `pandas`.

Sledi preizkus izrisa grafov s knjižnicama `seaborn` in `Matplotlib`. Sledeča koda prikazuje izris grafa raztrosa (angl. *scatter plot*) za prej ustvarjeno razpredelnico `data_df`.

```
import seaborn as sns

sns.scatterplot(x=data_df['Prvi stolpec'],
                y=data_df['Drugi stolpec'])
```

Če za izris grafa uporabimo `Matplotlib`, pa bo koda sledeča.

```
import matplotlib.pyplot as plt

plt.scatter(x=data_df['Prvi stolpec'],
            y=data_df['Drugi stolpec'])
```



Slika 2.9: Preizkus izrisa grafa raztrosa s knjižnico `seaborn`.

Na prvi pogled se zdi uporaba `seaborn` in `Matplotlib` knjižnic podobna, kar tudi v resnici je. Nekatere tehnike grafične predstavitve podatkov je s knjižnico `seaborn` lažje narediti, medtem, ko je pri izrisu grafov `Matplotlib` mnogo bolj prilagodljiv. Prikaz uporabe knjižnice `seaborn` je na sliki 2.9.

Sedaj bomo preizkusili še knjižnico `scikit-learn`, ki vsebuje različne pristope, metode ter tehnike strojnega učenja in predprocesiranja podatkov. Preden se lotimo uporabe metod strojnega učenja, bomo tokrat preizkusili nalaganje priloženih podatkov v knjižnici `scikit-learn`. Ena izmed priloženih je podatkovna zbirka *Iris*, ki vsebuje podatke o cvetočih rastlinah perunikah. V prvi vrstici se najprej naloži modul knjižnice `scikit-learn`, v drugi vrstici pa se naloži podatkovna zbirka *Iris* ter se shrani v spremenljivko `iris_vse`. Ker smo pri klicu metode podali `as_frame=True`, bodo rezultati v obliki

pandas razpredelnice DataFrame. Če tega ne bi podali, pa bi rezultat podatkov bil v obliki NumPy polja. Shranjeni podatki v spremenljivki iris_vse imajo več vrednosti: vrednost data nam vrne podatke o rožah (višine in širine čašnih in cvetnih listov). Vrednost target pa nam vrne vrednosti razredov te podatkovne zbirke – za kateri tip perunike gre. Zadnja vrstica kode izpiše prvih pet vrstic podatkov, kot je tudi prikazano na sliki 2.8.

```
from sklearn.datasets import load_iris

iris_vse = load_iris(as_frame=True)
iris_podatki = iris_vse.data
iris_razredi = iris_vse.target

iris_podatki.head()
```

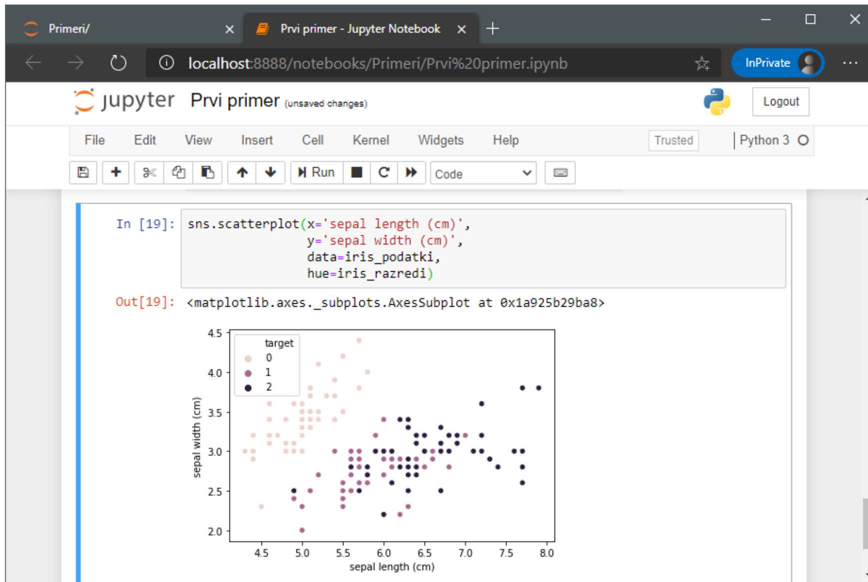
```
 sepal length (cm) sepal width (cm) petal length (cm) petal width
(cm)
0 5.1               3.5              1.4               0.2
1 4.9               3.0              1.4               0.2
2 4.7               3.2              1.3               0.2
3 4.6               3.1              1.5               0.2
4 5.0               3.6              1.4               0.2
```

Za konec preizkusimo z izrisom grafa raztrosa, kjer bodo pike vsake izmed instanc pobarvane glede na podan razred te instance.

```
sns.scatterplot(x='sepal length (cm)',
                y='sepal width (cm)',
                data=iris_podatki,
                hue=iris_razredi)
```

Tokrat smo za parameter data podali kar razpredelnico iris_podatki, ki je pandas DataFrame. Parametra x in y smo tokrat zapisali kot niz znakov – imena stolpcev iz podane razpredelnice. S parametrom hue določimo, kako se oznake na grafu pobarvajo. Lahko bi podali tudi

ime stolpca (ki pa ga tokrat v razpredelnici `data` nimamo), ali pa kot ločeno spremenljivko razpredelnice s temi podatki (kot smo naredili tokrat z `iris_razredi`). Rezultat je prikazan na sliki 2.10.



Slika 2.10: Graf raztrosa podatkovne zbirke *Iris*.

2.4 Nalaganje podatkov

Da lahko izvedemo analize in uporabimo algoritme strojnega učenja, pa potrebujemo podatke. Sledeča sekcija pregleda, kako naložimo podatke v naš Python program oziroma Jupyter zvezek.

Nalaganje prostodostopnih podatkov

Že v prejšnji sekciji smo pokazali, kako lahko dostopamo do standardnih podatkovnih zbirk, ki se mnogokrat uporabljajo v procesu učenja. To smo storili s pomočjo knjižnice `scikit-learn`, saj so preko te knjižnice že nameščene številne podatkovne zbirke. Skozi celotno knjigo bomo največkrat uporabili podatkovno zbirko *Iris*, v nalogah pa bodo uporabljene tudi *Wine* in *Breast cancer*. V vseh primerih gre za podatkovne zbirke, namenjene klasifikaciji. Nabor podatkovnih zbirk, ki so priložene tej knjižnici, je dostopen na `https://scikit-learn.org/stable/datasets/toy_dataset.html` za enostavne in majhne zbirke ter `https://scikit-learn.org/stable/datasets/real_world.html` za zbirke iz realnega sveta.

Podatkovne zbirke iz `scikit-learn` vrnejo objekt s sledečimi vrednostmi:

- `data` je razpredelnica s podatki vseh instanc,
- `target` je polje z rešitvami vseh instanc,
- `feature_names` je seznam imen vseh atributov, ki so v `data` ter
- `target_names` je seznam imen rešitev (če gre za razrede).

Če nam priložene prostodostopne podatkovne zbirke niso dovolj, lahko uporabimo poljubno podatkovno zbirko iz portala *OpenML.org*[11]. Ta portal igra vlogo repozitorija za podatkovne zbirke. Te lahko uporabniki portala prosto nalagajo na portal in tudi do njih dostopajo. Knjižnica `scikit-learn` ponuja vmesnik za neposredno nalaganje teh

[11] `https://www.openml.org/`

datotek s pomočjo metode `fetch_openml`, kateri podamo ime zbirke, kot kaže spodnja koda.

```
from sklearn.datasets import fetch_openml

podatki = fetch_openml(name='ecoli')

print(podatki.feature_names)
```

```
['mcg', 'gvh', 'lip', 'chg', 'aac', 'alm1', 'alm2']
```

Tudi pri nalaganju podatkovnih zbirk iz *OpenML.org* lahko `fetch_openml` metodi povemo, da želimo razpredelnico `DataFrame`. To storimo enako kot pri nalaganju priloženih podatkovnih zbirk, s podajo `as_frame=True`.

```
from sklearn.datasets import fetch_openml

podatki = fetch_openml(name='nursery', as_frame=True)

print(podatki.target.head())
```

```
0 recommend
1 priority
2 not_recom
3 recommend
4 priority
```

Nalaganje lastnih podatkov

Podatke lahko v program oziroma zvezek naložimo s pomočjo knjižnice `pandas`, ki prepozna podatke v številnih različnih formatih[12]:

- tekstovne datoteke, kjer so podatki ločeni z znaki (npr. z vejico v CSV, tabulatorjem ali drugim znakom),

[12] https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html

- JSON format tekstovne datoteke,
- HTML spletne strani, iz katerih se podatki preberejo iz elementa `<table>`,
- Excel ali OpenDocument razpredelnice,
- Lastniške SAS ali SPSS datoteke podatkov,
- podatkovne baze s SQL klicem ter
- lokalne odložišča (angl. *clipboard*).

Primer nalaganja iz CSV datoteke, kjer so vrednosti ločene s podpičjem ; prikazuje spodnja koda. Vsak format ima svojo metodo za prebiranje datotek, saj so določene nastavitve formatu specifične. Pri prebiranju iz tekstovnih datotek tako lahko podamo znak, ki ločuje vrednosti s `sep`, znak ločevanja decimalnih mest s `dec`, vrstico, ki predstavlja imena atributov, s `header` ali pa podamo ta imena kar v `names`.

```
import pandas as pd

podatki = pd.read_csv('datoteka.txt', sep=';', dec=',')
```

S pomočjo tega je po prebiranju knjige mogoče osvojeno znanje aplicirati na lastne podatke.

3 | Klasifikacija

Kam gremo po nasvet, ko se počutimo bolni? Se zatečemo k sosedu, sodelavki, prijatelju ali pa raje obiščemo zdravnika? Vsekakor je najbolje, da obiščemo zdravnika. Premislimo, zakaj. Kaj je prednost zdravnika v primerjavi z ostalimi naštetimi v danem problemu? Zdravnik ima najverjetneje mnogo več znanja in izkušenj pri diagnozi bolezni, saj se je kot študent več let učil prav o tej tematiki in v svoji delovni karieri že pridobil mnogotero izkušenj. Tekom tega procesa pridobivanja znanja in izkušenj tako predpostavimo, da se je zdravnik že usposobil za kakovostno spopadanje z danim problemom diagnoze bolezni.

3.1 Model znanja

Naslednjič, ko srečate zdravnika, ga kar povprašajte o tem, kako se spopade z izzivom diagnoze pacienta na podlagi simptomov obolenj. Verjetno bo začel s pregledom trenutnega in preteklega stanja pacienta. Naredil bo različne meritve pacientovega telesa (telesna temperatura, krvni pritisk, hormonska slika, krvna slika, rentgen ...) in dodatno pregledal pacientovo kartoteko.

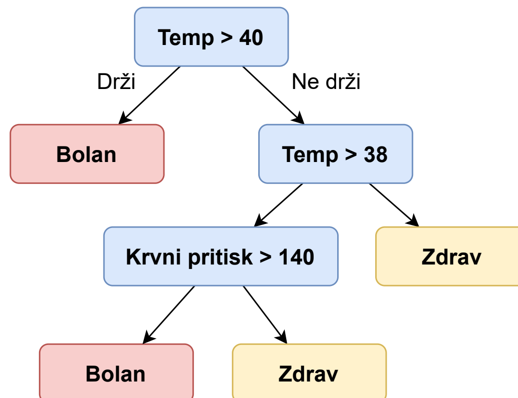
Ob preučevanju novih in preteklih meritev pacienta bo zdravnik naj-

verjetneje že imel ustaljen postopek razmišljanja pri postavitvi diagnoze. Recimo, da želimo analizirati delo našega zdravnika in ga prosimo, da na kar se da razumljiv način opiše svoj proces odločanja. Najverjetneje zdravnik (ali strokovnjak drugih področij) nima formalno določenega procesa odločanja, ampak se pri tem zateka tudi k svoji intuiciji, ki jo je razvil po vseh letih šolanja in izkušenj. Pa vendar, če bi se zdravnik odločil zapisati svoj proces odločanja na papir, bi ta verjetno izgledal nekako tako:

- Če je telesna temperature pacienta nad 40 °C, je pacient **bolan**.
- Če je telesna temperature nad 38 °C in je sistolični krvni pritisk nad 140 mHg, je pacient **bolan**.
- Če je telesna temperature pod 38 °C, je pacient **zdrav**.

Svoj proces diagnoze bi zdravnik zapisal v obliki pravil, katerim bi sledil bodisi po vrsti ali pa bi jih upošteval vse hkrati. Tem pravilom pravimo *model znanja* (angl. *knowledge model*).

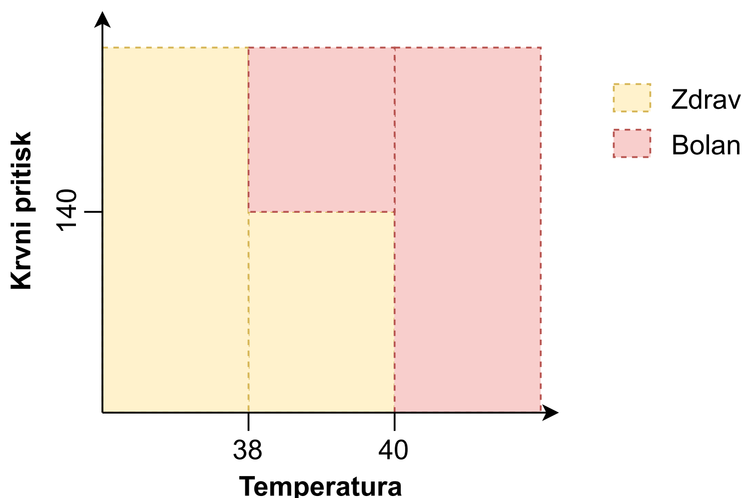
Kateri drug zdravnik pa bi svoj proces odločanja mogoče lažje zapisal v obliki odločitvenega drevesa (slika 3.1).



Slika 3.1: Model znanja v obliki odločitvenega drevesa.

Pri interpretaciji takega drevesa bi zdravnik začel na vrhu (korenu) drevesa in se z odgovarjanjem na vprašanja v posameznem vozlišču drevesa pri pregledu posameznega pacienta premikal vse do listov drevesa. Listi pa mu povedo odgovor glede diagnoze pacienta.

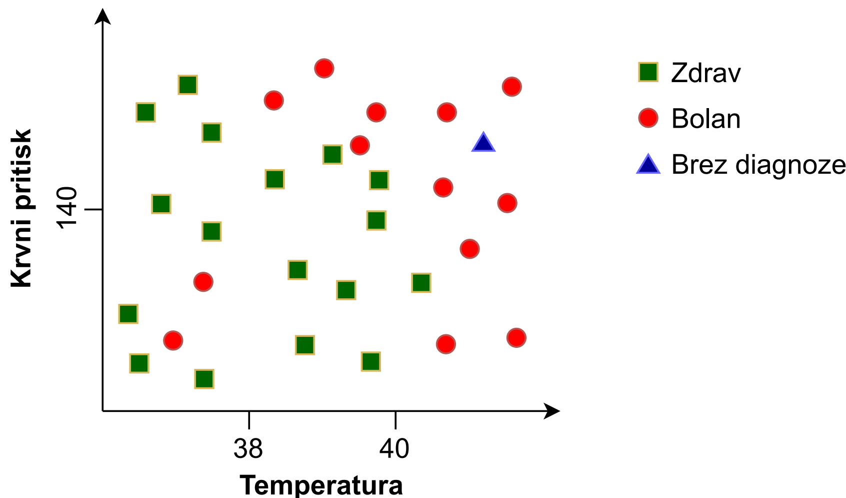
Nek tretji zdravnik pa boljše razmišlja, ko svoj model znanja predstavi v obliki matematičnih formul. Slika 3.2 prikazuje graf, kjer je na horizontalni X osi telesna temperatura pacienta, na vertikalni Y osi pa je krvni pritisk pacienta. Matematična formula deli prostor na dva dela – na prostor, kjer se nahajajo zdravi in na prostor, kjer se nahajajo bolni pacienti.



Slika 3.2: Model znanja v obliki matematične funkcije.

Zadnji zdravnik pa bi deloval popolnoma drugače kot prejšnji trije. Ta ne bi znal na papir zapisati svojega modela znanja, saj se odloča po drugačnem postopku – pregleda svoje prejšnje paciente iz bogate dvajsetletne kariere in novega pacienta primerja z njimi. Ko najde primere že diagnosticiranih pacientov, ki so novemu pacientu najbolj podobni, preprosto pogleda, kakšna je bila njihova diagnoza in temu

novemu poda mnenje o njegovem stanju bolezni, ki je skladna s prejšnjimi (podobnimi) pacienti. Ta proces je prikazan na sliki 3.3.



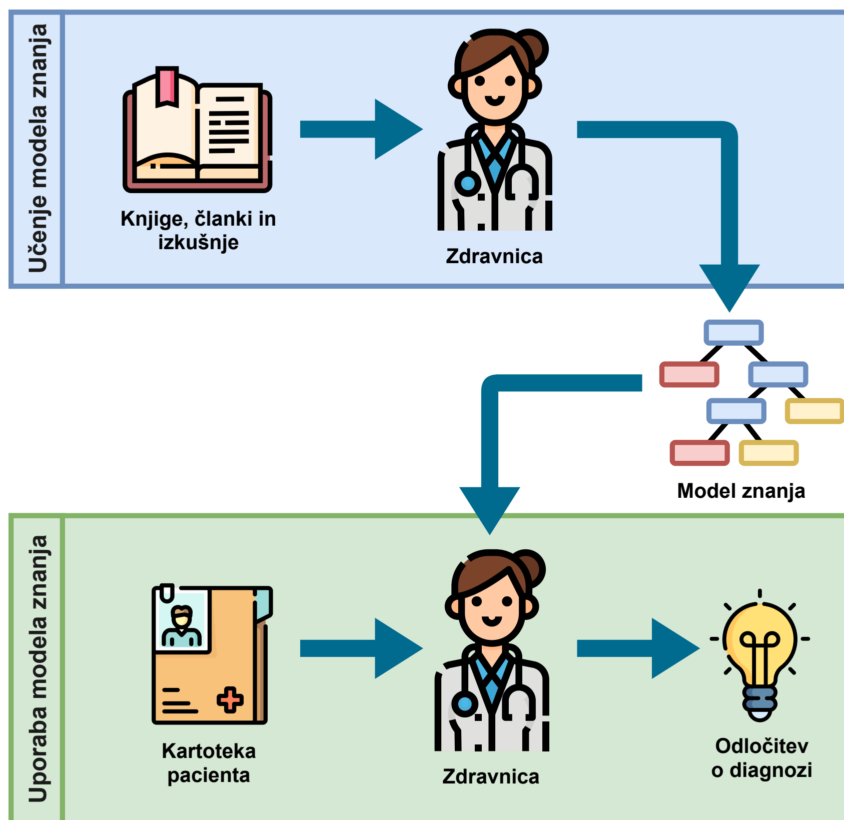
Slika 3.3: Model znanja v obliki sprotne primerjave z že rešenimi primeri.

Vsi štirje opisani načini odločanja zdravnikov v resnici prikazujejo različne načine predstavitve modela znanja. Teh je v realnosti še mnogo več, tekom te knjige pa bomo podrobneje spoznali zadnji način zapisa modela znanja – primerjavo novih primerov s starimi, že rešenimi.

3.2 Ekspertni sistem

Gradnjo modela znanja zdravnika vizualno prikazuje slika 3.4. Najverjetneje preberejo zdravniki tekom študija medicine in delovne kariere mnogo knjig, znanstvenih in strokovnih člankov ter pridobijo mnogotere izkušnje. To kaže zgornji del slike. Rezultat takega procesa je model znanja (seveda ne formalno zapisan). Ob pregledu novega

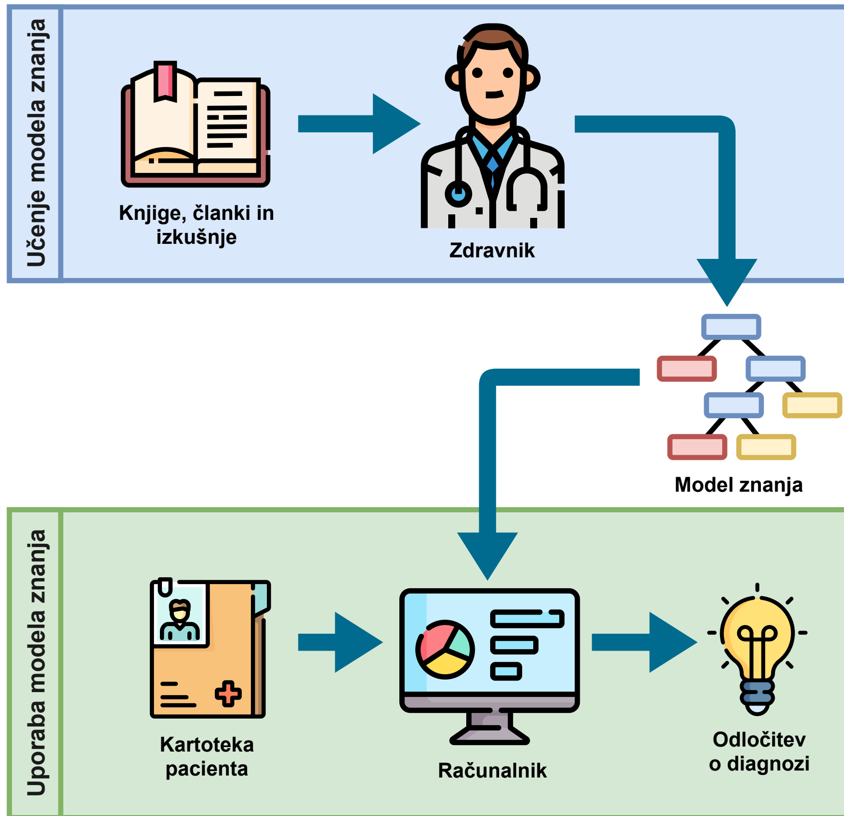
pacienta uporabijo ta model znanja, da pridejo do končne diagnoze pacienta, kar je ponazorjeno s spodnjim delom slike.



Slika 3.4: Poenostavljen prikaz procesa gradnje in uporabe modela znanja.

Seveda lahko zdravnikov model znanja prepišemo v programsko kodo in tako uporabimo njegov model znanja v vsakdanjih informacijskih sistemih, kot kaže slika 3.5. To je standardni postopek pri gradnji *ekspertnih sistemov* (angl. *expert systems*) tj. informacijskih sistemov, ki uporabijo model znanja, ki so jih zgradili strokovnjaki (na primer

zdravniki). Z digitalizacijo modela znanja pridobimo zmožnost hitrejše uporabe modela znanja in posledičnega odločanja, saj računalnik lahko tak model znanja uporabi tudi milijonkrat v sekundi.



Slika 3.5: Proces uporabe modela znanja v ekspertnem sistemu.

Pri ekspertnem sistemu še vedno ne govorimo o umetni inteligenci ali strojnem učenju, saj računalnik ni sodeloval pri procesu učenja (gradnje modela znanja), ampak je le *neumen* stroj, ki sledi modelu znanja, zapisanem v obliki *if* in *for* stavkov.

3.3 Inteligentni sistem

Premislimo, kako bi lahko v prejšnjem procesu vključili računalnik v proces gradnje modela znanja. Namesto da človek postane strokovnjak s pomočjo preučevanja literature in nabiranja izkušenj, bomo tokrat uporabili kar računalnik za gradnjo modela. Ampak kako? Računalnik ne razume prebranih knjig in ne more pridobivati izkušenj. Tukaj pa uporabimo enak pristop učenja, kot ga uporabijo ljudje v situacijah, kjer se učijo s pomočjo opazovanja – pri delu opazujejo strokovnjake in skušajo ugotoviti, kako strokovnjaki delajo, da lahko to kasneje posnemajo.

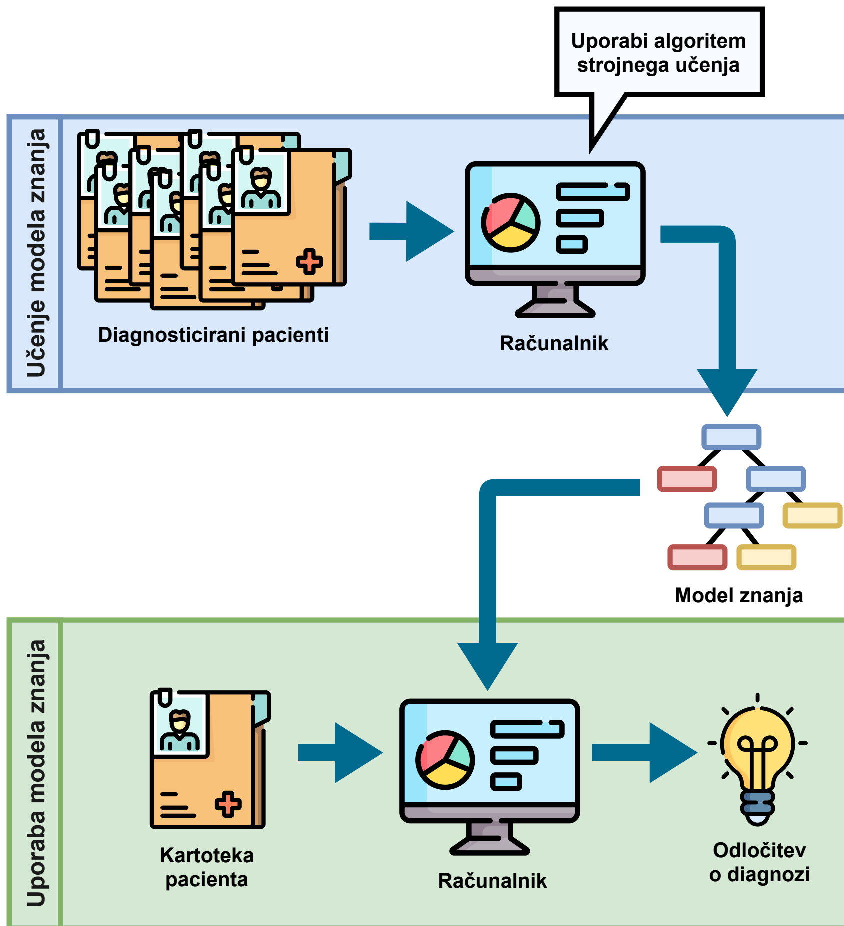
Podobno pot uberemo, ko želimo pri gradnji modela uporabiti računalnik kar kaže slika 3.6. Namesto iz knjig in izkušenj, računalnik razbere vzorce iz dela strokovnjakov, ki pa je v tem primeru sestavljeno iz že rešenih problemov. Če računalniku podamo kartoteke že diagnosticiranih pacientov, bo ta lahko iz teh kartotek razbral vzorce, ki so značilni za bolne paciente in vzorce, ki so značilni za zdrave paciente.

Seveda, enako kot vajenec ne bo postal mojster pri enournem opazovanju strokovnjaka pri delu, tako tudi računalnik ne uspe najti vzorcev iz le nekaj primerov že diagnosticiranih pacientov. Koliko podatkov pa bo dovolj? Če je delo mojstra zelo zapleteno, bo vajenec potreboval nekaj let opazovanja, da bo ugotovil pravi način dela tega strokovnjaka. Če pa vajencu kažemo, kako nalepiti obliž na rano, pa bo vajenec vzorec za reprodukcijo videnega odkril že po nekaj minutah. Podobno je tudi pri računalniku.

Količina potrebnih rešenih primerov je odvisna od količine in kompleksnosti vzorcev – več kot je vzorcev in bolj so ti kompleksni, več podatkov bo računalnik potreboval, da bo vzorce prepoznal.

Računalniški algoritem, ki iz podatkov razbira vzorce, imenujemo *algoritem strojnega učenja* (angl. *machine learning algorithm*). Ta algoritem je zadolžen, da namesto človeka ustvari model znanja, ki ga

kasneje lahko pri svojem delu uporabita tako človek, kakor tudi računalnik. Sistem, ki vključuje algoritem strojnega učenja za gradnjo modela znanja, pa imenujemo *inteligentni sistem* (angl. *intelligent system*).



Slika 3.6: Proces učenja in uporabe modela znanja v inteligentnem sistemu.

3.4 Podatki

Podatki, uporabljeni v algoritmu strojnega učenja za namen kreacije modela znanja, so združeni v *podatkovne množice* (angl. *datasets*) in so lahko v obliki preproste strukturirane tekstovne datoteke ali podatkovne baze poljubne strukture (relacijske, objektne ali dokumentne podatkovne baze).

Najenostavnejša struktura podatkov, namenjenih za strojno učenje ima obliko, ki je prikazana na sliki 3.7. Vsako vrstico v tekstovnem dokumentu ali vsak primerek podatkovne baze imenujemo *učna instanca* ali *učni primerek* (angl. *learning instance* ali *learning example*). Vsaka instanca je opisana z množico karakteristik imenovanih *atributi* (angl. *features*) ali tudi *neodvisne spremenljivke* oziroma *značilnice*. Prostor atributov (angl. *feature space*) F je vektor vseh atributov.

Atributi so v stolpcih ↓

Instance so v vrsticah →

Višina	*Teža*	*Starost*	...
181	92	45	
178	71	27	
168	73	65	
	...		

Slika 3.7: Struktura podatkov, primernih za strojno učenje.

3.5 Strojno učenje

Strojno učenje (angl. *machine learning*) zajema tehnike, kjer se računalnik nauči reševanja specifičnih in ozko usmerjenih nalog iz podatkov – pravimo, da odločitve strojnega učenja temeljijo na podatkih. Tehnike strojnega učenja uvrščamo v krovno področje *umetne inteligence* (angl. *artificial intelligence*), ki pa pokriva mnogo širše raziskovalno področje. Področje, povezano s strojnim učenjem, je tako

imenovano *podatkovno rudarjenje* (angl. *data mining*), kjer s pomočjo različnih tehnik, med drugimi tudi s strojnim učenjem, obdelujemo in preučujemo podatke ter poskušamo iz njih razbrati vzorce in posledično novo znanje. Izraz podatkovno rudarjenje uporabljamo, ko se srečamo s problemom uporabe samih metod strojnega učenja kot orodij za reševanje drugih problemov in ne s samo implementacijo teh.

Strojno učenje delimo na štiri področja glede na stopnjo nadzora nad učenjem [7]:

Nadzorovano učenje (angl. *supervised learning*) se uporablja, ko želimo, da se računalnik nauči klasificirati (razvrščati) podatke v vnaprej določene razrede ali jim pripisovati številske vrednosti. Temu rečemo nadzorovano učenje, ker se stroj uči na rešenih podatkih (z znanimi razredi ali vrednostmi) in ker lahko nadzorujemo kakovost dobljenih modelov znanja. Pri nadzorovanem učenju imamo dve nalogi, ki ju mora stroj opravljati: regresijo in klasifikacijo.

Regresija (angl. *regression*) se uporablja, ko se računalnik na podlagi podanih karakteristik (neodvisnih spremenljivk) nauči napovedovanja numeričnih vrednosti (odvisne spremenljivke). Primer takega problema bi bil napovedovanje cene delnice ali količine dežja glede na znane podatke. Z regresijo se v tej knjigi ne bomo ukvarjali, zato se v podrobnejše razlage ne bomo spuščali.

Klasifikacija (angl. *classification*) pa se uporablja, ko se računalnik iz rešenih podatkov nauči te razvrščati v vnaprej določene razrede. Primer smo že omenjali, ko smo govorili o diagnozi pacientov in ga bomo podrobneje opisali v nadaljevanju.

Nenadzorovano učenje (angl. *unsupervised learning*) uporabimo, ko želimo odkriti še neznane povezave med podatki in strukturo teh podatkov. V tem primeru naši podatki ne vsebujejo rešitve,

saj rešitev še ne poznamo, in posledično kakovosti takih modelov ne moremo nadzirati. Nenadzorovano učenje ima več nalog, ki se jih stroj nauči: gručenje in sprememba strukture podatkov.

Gručenje (angl. *clustering*) je tehnika, kjer računalnik najde vzorce, ki povezujejo podatke v gruče, in tako najde doslej neznane povezave med podatki. Definicija teh gruč ni vnaprej znana, a v vsakem primeru računalnik uporabi eno izmed tehnik, da gruče združujejo podobne in povezane podatke skupaj. Število gruč je lahko vnaprej določeno ali pa se odločitev o številu gruč prepusti računalniku. Primer gručenja je iskanje profilov strank v trgovini, kjer imajo stranke v isti gruči podobne nakupovalne navade.

Spreminjanje in preučevanje strukture podatkov pa združuje tehnike, ki se ukvarjajo s transformacijo, preslikavo, združevanjem in selekcijo posameznih karakteristik iz podatkov. Podrobneje teh tehnik ne bomo obravnavali v okviru te knjige.

Delno nadzorovano učenje (angl. *semi-supervised learning*) je srednja pot med nadzorovanim in nenadzorovanim učenjem. Pri tej tehniki še vedno klasificiramo podatke v vnaprej podane razrede, pri čemer si pomagamo z novimi karakteristikami podatkov, ki pa so rezultat nenadzorovanega učenja, ali pa uporabljamo le delno označene podatke (na primer, če imamo v učni množici le bolne paciente). Tipična uporaba delno nadzorovanega učenja je iskanje anomalij, kjer poznamo samo lastnosti normalnih podatkov in iz tega sklepamo, kaj je normalno – vse, kar je drugačno, pa je anomalija.

Okrepitveno učenje (angl. *reinforcement learning*) je razširjeno nadzorovano ali nenadzorovano učenje, kjer se računalnik uči na podlagi nagrad ali kazni glede na izide učenja. Pri nagrajevanju in kaznovanju lahko sodeluje človek ali pa je nagrada podeljena računsko. V primeru sodelovanja človeka, ta poda dodatne in-

formacije o samih podatkih, ali pa v iterativnem postopku poda mnenje o kakovosti modela. Tega pristopa strojnega učenja v okviru te knjige ne bomo obravnavali.

3.6 Klasifikacija

Osrednja metoda strojnega učenja te knjige je metoda *klasifikacije*, kjer se računalnik nauči klasificirati instance v vnaprej določene razrede. Z regresijo napovemo številske vrednosti in so odločitve zvezne, pri klasifikaciji pa napovedujemo nominalne vrednosti – diskretne razrede. Klasifikacija se uporablja v primerih, kot so razpoznava vzorcev na slikah, preprečevanje prevar, zaznavanje nezaželene pošte in diagnosticiranje bolezni. Če računalnik podatke deli v dva razreda, govorimo o *binarni klasifikaciji*, če pa računalnik klasificira v več razredov, pa imamo opravka z *večrazredno klasifikacijo*. [8]

Obstaja na tisoče različnih algoritmov klasifikacije, ki jih v grobem delimo glede na to, v kakšni obliki shranijo model znanja [8], [9]:

- matematične formule in porazdelitve (logistična regresija, naivni Bayesov klasifikator, metoda podpornih vektorjev),
- odločitvena drevesa (CART, C4.5, ID3, evolucijska drevesa),
- odločitvena pravila (RIPPER, PART, evolucijska pravila),
- umetne nevronske mreže (konvolucijske, rekurzivne) in
- klasifikatorji na podlagi podobnosti (k najbližjih sosedov).

Nekatere metode zgradijo model znanja, ki se uporablja pri klasifikaciji novih instanc, ne da bi bil potreben vpogled v prej podane podatke. Takim metodam pravimo *metode takojšnjega učenja* (angl. *eager learning*), saj zgradijo model znanja takoj in ga kasneje ne prilagajajo.

Nasprotno, pa nekatere metode, kot na primer k najbližjih sosedov, ne ustvarijo učnega modela, ampak za vsako klasifikacijo nove instance

ponovno naredijo pregled že prej podanih podatkov. Te metode uporabljajo *leno učenje* (angl. *lazy learning*), saj se učijo sproti po potrebi. Prav ta pristop bomo kasneje pregledali pri preizkusu našega klasifikatorja k najbližjih sosedov.

3.7 Matematična definicija pojmov

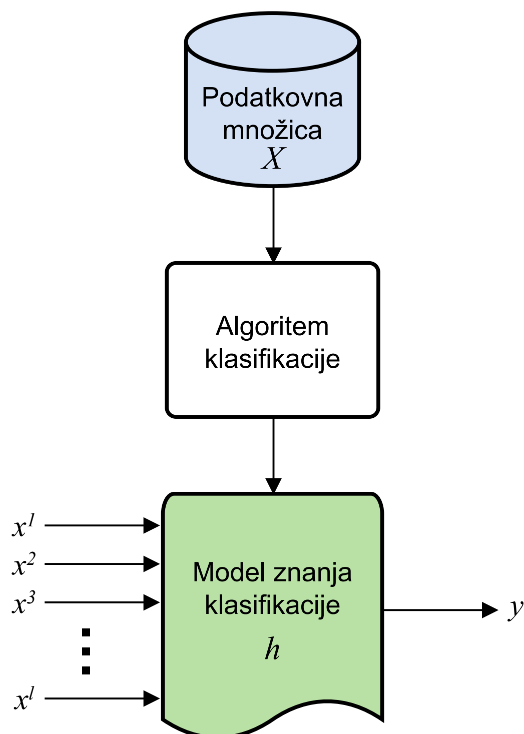
V nadaljevanju bomo za metodo klasifikacije uporabljali sledečo definicijo instanc. Ena instanca je par (x_i,y_i), kjer je x_i vektor vrednosti (atributov) te instance, y_i pa je dejanski razred instance (skalarna vrednost). Učna množica X je definirana kot množica vseh instanc, na katerih se algoritem uči, sestavljena je iz n instanc in je definirana, kot prikazuje spodnja enačba.

$$\begin{aligned}
X &= \{(x_1,y_1),(x_2,y_2),\ldots,(x_n,y_n)\} \\
x_i &= (x_i^1,x_i^2,\ldots,x_i^l) \\
y_i &\in \{razred_1, razred_2, \ldots, razred_k\} \\
n &= \text{število instanc} \\
l &= \text{število atributov} \\
k &= \text{število razredov}
\end{aligned}$$

Vrednost x_i je vektor atributov $(x_i^1,x_i^2,\ldots,x_i^l)$ velikosti l, ki je definiran v prostoru atributov F, razred instance y_i pa je vrednost iz nabora vseh možnih razredov v velikosti k. Cilj algoritmov klasifikacije je, da ob dani učni množici X najdejo sledečo funkcijo:

$$h : x_i \to y_i$$

za vsak $i \in [1,n]$, tako da bo model znanja klasifikacije $h(x_i)$ dovolj dober napovedovalec razreda y_i. Proces klasifikacije prikazuje slika 3.8.



Slika 3.8: Splošni proces klasifikacije, kjer iz podatkovne množice X algoritem klasifikacije ustvari model znanja klasifikacije h. Ta model preslika vhode instance $x^1, x^2, \ldots, x^l$ v razred y.

Ob pregledu splošnega področja strojnega učenja in klasifikacije pa zdaj sledi preizkus prvega klasifikatorja. Naredili bomo teoretični pregled klasifikatorja k najbližjih sosedov in prikazali praktični primer njegove uporabe.

4 | Prvi klasifikator

Začeli bomo s pregledom delovanja klasifikacije na podlagi podobnosti med instancami. Prvo vprašanje, ki se nam poraja, je, kako sploh določimo podobnost med instancami? Oziroma povedano drugače, kdaj sta si dve instanci podobni in kdaj ne? Intuitivno si ljudje naredimo prvi vtis glede na podobnost z že poznanimi koncepti. Vidimo nov avto? Ta je podoben našemu avtu doma – torej je najverjetneje hiter, porabi veliko goriva in ni primeren za vožnjo po slabi cesti.

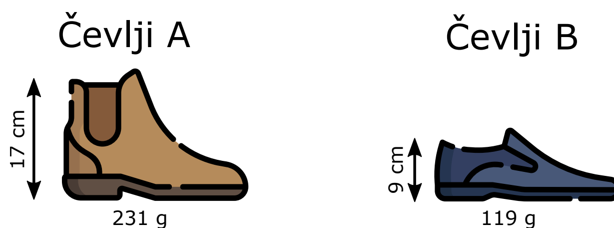
Kaj pa, če izbiramo nove zimske čevlje izmed nabora čevljev, ki jih trgovina ponuja? Vse čevlje v ponudbi primerjamo na podlagi izkušenj z zimskimi čevlji, ki smo si jih lastili v preteklosti. Želimo si nove čevlje, ki so čim bolj podobni prejšnjim, na žalost obrabljenim. Všeč nam je bila njihova barva, prav tako nas niso žulili pri daljši hoji, na ledu nam nikoli ni drselo pa tudi za na fakulteto so bili primerni. Izmed vseh ponujenih čevljev najdemo najbližje našim prejšnjim – to ljudje naredimo intuitivno, hitro in brez nepotrebnih kalkulacij.

Kako pa pripravimo računalnik, da bo videl podobnosti med predstavljenimi koncepti? Z izračunom razdalje med dvema konceptoma. Bolj sta si dve stvari podobni, manjša je razdalja med njima, ter obratno – manj sta si dve stvari podobni, večja je razdalja med njima.

Slika 4.1: Izbira novih čevljev s primerjavo.

4.1 Računanje razdalj

Poglejmo si primer primerjave dveh čevljev prikazan na sliki 4.2.



Slika 4.2: Primerjava dveh čevljev.

Ta primer preslikajmo v tabelo, kjer je prva vrstica namenjena atributom prvih čevljev, druga vrstica je namenjena atributom drugih čevljev, v tretji vrstici pa so razlike med njima.

Tabela 4.1: Primer računanja razdalje med dvema čevljema.

	Višina	*Teža*	*Vodoodporni*	*Barva*
Čevlji A	17	231	da	rjavi
Čevlji B	9	119	ne	modri
Razlika	8	112	?	?

Razlike med številskimi atributi je enostavno izračunati; preprosto vzamemo vrednost številskih atributov prve instance in odštejemo vrednost atributov druge instance. V primeru kategoričnih atributov pa ni možno določiti, kateri kategoriji sta si bližji in kateri sta si dlje. Ko moramo izračunati razdaljo med dvema kategorijama, preprosto uporabimo naslednje pravilo:

- Če sta kategoriji enaki, je razdalja med njima 0.
- Če sta kategoriji različni, je razdalja med njima 1.

Knjižnica `scikit-learn` pa ne razlikuje med tipi spremenljivk – vse spremenljivke obravnava kot številske oziroma razmernostne. Za naši dve kategorični spremenljivki vodoodpornosti in barve čevljev to predstavlja težavo, saj v trenutni obliki nista zapisani v obliki števil.

Indikacijski atributi

Za uporabo podatkov s knjižnico `scikit-learn` je podatke potrebno prilagoditi tako, da kategorične atribute spremenimo v številske.

Napačen pristop bi bil določitev števila vsaki kategoriji. Pri barvi čevljev bi tako barva *črna* postala število 0, barva *modra* število 1, *barva* rjava število 2, in tako naprej. Ta pristop še vedno ni pravilen, saj algoritmi strojnega učenja največkrat operirajo s takimi vrednostmi – naključna določitev števil kategorijam pa bi izračune pokvarila. Pri računanju razdalj bi tako razdalja med črnimi in rjavimi čevlji bila 2, med črnimi in modrimi pa le 1. To je nesmiselno, saj barv ne moremo

postaviti v vrstni red.

Posledično se transformacije kategoričnih atributov v številske atribute lotimo s pomočjo kreacije *indikacijskih atributov* (angl. *dummy attribute*). Iz enega kategoričnega atributa tako nastane več novih številskih atributov. Za vsako kategorijo enega kategoričnega atributa tako nastane svoj številski atribut. Če so čevlji lahko treh različnih barv (črni, modri, rjavi), potem nastanejo trije novi atributi: *Barva (črni)*, *Barva (modri)* in *Barva (rjavi)*. Indikacijski atribut ima lahko le dve vrednosti:

- vrednost 0, če kategorija **ne drži** za instanco ter
- vrednost 1, če kategorija **drži** za instanco.

Poglejmo si tabelo 4.2 podatkov obeh čevljev po transformaciji kategoričnih atributov v indikacijske atribute. Zdaj je primerjava mnogo bolj smiselna. Prav tako je opazno, da obstaja razlika tako v vodoodpornosti čevljev, kakor v barvi.

Tabela 4.2: Primer računanja razdalje z indikacijskimi atributi.

	Višina	*Teža*	*Vod. (da)*	*Vod. (ne)*	*Barva (črni)*	*Barva (modri)*	*Barva (rjavi)*
Čevlji A	17	231	1	0	0	0	1
Čevlji B	9	119	0	1	0	1	0
Razlika	8	112	1	-1	0	-1	1

4.1.1 Kreacija indikacijskih atributov v Pythonu

Najenostavnejši postopek kreacije indikacijskih atributov je s pomočjo `pandas` knjižnice. Struktura podatkov `DataFrame` ima že zabeležen tip vsakega stolpca, kar poenostavi avtomatsko transformacijo kategoričnih atributov v indikacijske. Sledeča koda prikazuje kreacijo podatkov, iz katerih se bodo kreirali indikacijski atributi.

```python
# Definiramo vsako instanco kot seznam atributov
cevljiA = [17, 231, 'da', 'rjavi']
cevljiB = [9, 119, 'ne', 'modri']
cevljiC = [12, 143, 'ne', 'črni']
cevljiD = [8, 112, 'ne', 'rjavi']
cevljiE = [11, 198, 'da', 'modri']
cevljiF = [15, 245, 'da', 'črni']
```

Sledi še kreacija `DataFrame` strukture s podatki.

```python
import pandas as pd

# Z združitvijo instanc ustvarimo DataFrame
podatki = pd.DataFrame([cevljiA, cevljiB, cevljiC,
                        cevljiD, cevljiE, cevljiF],
                       columns=['Višina', 'Teža',
                                'Vodood', 'Barva'],
                       index=['cevlji A', 'cevlji B',
                              'cevlji C', 'cevlji D',
                              'cevlji E', 'cevlji F'])
print(podatki)
```

```
          Višina  Teža Vodood  Barva
cevlji A      17   231     da  rjavi
cevlji B       9   119     ne  modri
cevlji C      12   143     ne   črni
cevlji D       8   112     ne  rjavi
cevlji E      11   198     da  modri
cevlji F      15   245     da   črni
```

S pregledom vrednosti `dtypes` podatkov lahko ugotovimo kakšnega tipa je posamezen atribut (stolpec).

```
podatki.dtypes
```

```
Višina     int64
Teža       int64
Vodood    object
Barva     object
dtype: object
```

Atributa *Vodood* in `Barva` sta tipa `object`, kar pomeni, da sta obravnavana kot kategorični spremenljivki. Pred uporabo teh podatkov v procesu strojnega učenja s knjižnico `scikit-learn` je potrebno spremeniti vse atribute `object` v številske. Knjižnica `pandas` ponuja metodo `get_dummies()`, ki pregleda podane podatke tipa `DataFrame` in vse atribute tipa `object` spremeni v indikacijske atribute. Izvorna spremenljivka se ne spremeni, temveč se podatki z indikacijskimi atributi vrnejo kot rezultat metode.

Pregled tipa atributov pokaže, da so zdaj vsi atributi številskega tipa, s čimer zadostimo uporabi v kombinaciji s `scikit-learn` knjižnico.

```
# Vse kategorične (object) atribute spremenimo v
                                 indikacijske
podatki_z_indikacijskimi = pd.get_dummies(podatki)

podatki_z_indikacijskimi.dtypes
```

```
Višina         int64
Teža           int64
Vodood_da      uint8
Vodood_ne      uint8
Barva_modri    uint8
Barva_rjavi    uint8
Barva_črni     uint8
```

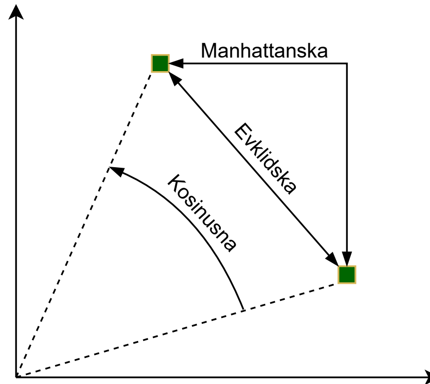
Pregled vsebine novih podatkov prikaže novonastale indikacijske atribute.

```
print(podatki_z_indikacijskimi)
```

```
           Višina  Teža  Vodood_da  Vodood_ne  Barva_modri  Barva_rjavi
cevlji A       17   231          1          0            0            1
cevlji B        9   119          0          1            1            0
cevlji C       12   143          0          1            0            0
cevlji D        8   112          0          1            0            1
cevlji E       11   198          1          0            1            0
cevlji F       15   245          1          0            0            0

.           Barva_črni
cevlji A             0
cevlji B             0
cevlji C             1
cevlji D             0
cevlji E             0
cevlji F             1
```

4.2 Skupna razdalja

Pri vmesnih izračunih razdalj še vedno ne pridemo do končne skupne razdalje med dvema instancama. Za agregacijo vmesnih razdalj v eno mero razdalje pa lahko izberemo več različnih pristopov. Slika 4.3 prikazuje več vrst razdalj, ki bodo opisane v sledečih sekcijah.



Slika 4.3: Različne razdalje med dvema točkama.

4.2.1 Evklidska razdalja

Evklidsko razdaljo med dvema točkama v ravnini je definiral matematik Evklid. Deluje po principu Pitagorovega izreka, kjer se razdalja med dvema točkama oz. instancama (x_1 in x_2) izračuna kot koren vsote kvadratov vseh dimenzij.

$$\begin{aligned} d_E\left(x_1,x_2\right) &= \sqrt{\left(x_1^{(1)} - x_2^{(1)}\right)^2 + \left(x_1^{(2)} - x_2^{(2)}\right)^2 + \ldots \left(x_1^{(l)} - x_2^{(l)}\right)^2} \\ &= \sqrt{\sum\nolimits_{i=1}^{l} \left(x_1^{(i)} - x_2^{(i)}\right)^2} \end{aligned}$$

kjer je l število atributov posamezne instance.

4.2.2 Mahattanska razdalja

Zelo pogosto pa nas zanima manhattanska razdalja, ki si zgled za računanje razdalje med dvema točkama vzame po postavitvi cest na otoku Manhattan, kot kaže slika 4.4. V večjem delu otoka so ceste postavljene vzdolž otoka (avenije) in prečno po otoku (ulice). Pri računanju razdalje od ene točke do druge je tako potrebno v obzir vzeti dolžine vseh cest med dvema točkama, pri tem pa ni možno krajšati poti z diagonalami.

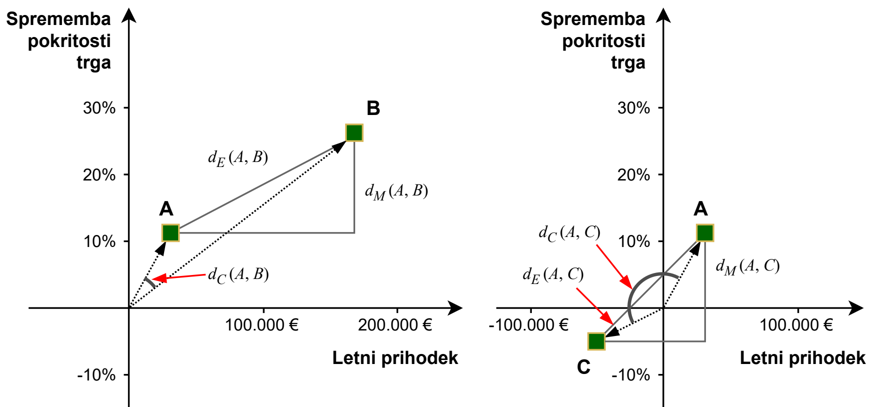
Slika 4.4: Razdalje na Manhattnu.

Pri izračunu manhattanske razdalje tako ni najkrajša pot predstavljena kot diagonalna in najkrajša daljica med dvema instancama, ampak kot vsota vseh daljic, ki poteka vzdolž vseh osi.

$$
\begin{aligned}
d_M(x_1,x_2) &= \left|x_1^{(1)} - x_2^{(1)}\right| + \left|x_1^{(2)} - x_2^{(2)}\right| + \ldots \left|x_1^{(l)} - x_2^{(l)}\right| \\
&= \sum\nolimits_{i=1}^{l} \left|x_1^{(i)} - x_2^{(i)}\right|
\end{aligned}
$$

4.2.3 Kosinusna razdalja

Z manhattansko in evklidsko razdaljo pa pridemo do težav, ko imamo meritve, ki lahko imajo tako negativne kot pozitivne vrednosti. Primer take meritve bi bil letni zaslužek podjetja, saj je ta lahko tudi negativen (podjetje je na letni ravni imelo izgubo). Za primer vzemimo tri podjetja in njihove letne zaslužke ter spremembe deleža pokritega trga od prejšnjega leta, kot je na sliki 4.5.



Slika 4.5: Primerjava evklidske, manhattanske in kosinusne razdalje.

Tako evklidska kakor tudi manhattanska razdalja pravita, da sta podjetji A in C bližje kot pa podjetji A in B.

$$d_E(A,B) > d_E(A,C)$$
$$d_M(A,B) > d_M(A,C)$$
$$d_C(A,B) < d_M(A,C)$$

To je v nasprotju z našo intuicijo: podjetje C je namreč imelo izgubo v letnem prihodku in negativno spremembo pokritosti trga. Podjetji

A in B pa sta imeli tako dobiček, kakor tudi se je njun delež pokritega trga povečal v primerjavi z lanskim letom.

V takih situacijah sta evklidska in manhattanska razdalja neprimerni ter se uporabi kosinusna razdalja. Pri kosinusni razdalji je pomembna (1) smer vektorja posamezne instance ter (2) njegova dolžina. Razliko med smerjo dveh vektorjev merimo s kotom α med vektorjema (instancama), ta pa je proporcionalna kosinusu kotov. Za instanci x_1 in x_2 velja sledeč izračun kosinusne razdalje.

$$
\begin{aligned}
d_C\left(x_1,x_2\right) &= 1 - \frac{x_1 \cdot x_2}{||x_1|| \cdot ||x_2||} \\
&= 1 - \frac{\sum_{i=1}^{l} x_1^{(i)} * x_2^{(i)}}{\sqrt{\sum_{i=1}^{l} x_1^{i\;2}} * \sqrt{\sum_{i=1}^{l} x_2^{i\;2}}}
\end{aligned}
$$

Izbira načina izračuna razdalje je odvisna od problema. Najbolj intuitivna je vsekakor evklidska razdalja. Večji je nabor atributov l, bolj je primerna manhattanska razdalja v primerjavi z evklidsko [10]. Kosinusna razdalja pa je primerna, ko nas bolj kot razdalje zanimajo podobnosti med instancami.

4.3 Klasifikator *k* najbližjih sosedov

V sekciji 3.6 je bilo predstavljeno, da pri nekaterih algoritmih klasifikacije ne nastane učni model, ampak se klasifikator odloča vsakokrat ob pogledu v že klasificirane instance. Taka vrsta učenja se imenuje *leno učenje* in algoritem klasifikacije *k najbližjih sosedov* (angl. *k nearest neighbors*) je tipični primer *lenega* algoritma učenja. Sledi pregled delovanja tega algoritma k najbližjih sosedov.

4.3.1 Delovanje *k* najbližjih sosedov

Algoritem k najbližjih sosedov steče po sledečem postopku [11], [12].

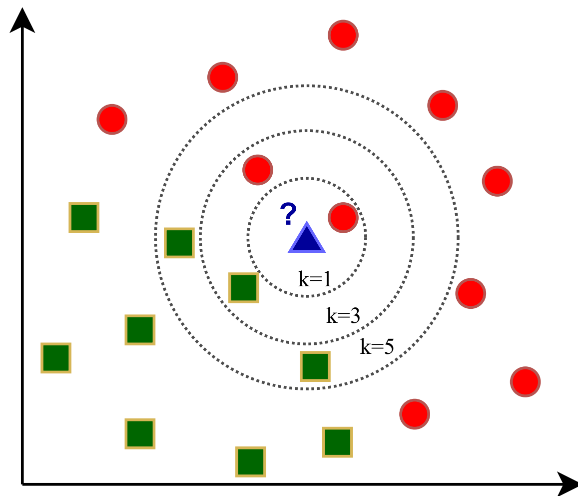
1. Za podano instanco x_N, ki jo želimo klasificirati, algoritem izračuna razdalje do vseh že klasificiranih instanc.

2. Že klasificirane instance razvrsti naraščajoče glede na razdaljo do instance x_N.

3. V nadaljnji obravnavi upošteva le k prvih instanc v naraščajočem seznamu – k najbližjih sosedov.

4. Izračuna pogostosti razredov iz nabora k najbližjih instanc kot je na sliki 4.6.

5. Najpogostejši razred (modus) vrne kot rezultat klasifikacije instance x_N. Če se več razredov pojavlja z največjo pogostostjo, se izbere naključen razred izmed vseh najpogostejših.



Slika 4.6: Klasifikacija instance (moder trikotnik) s pomočjo k najbližjih sosedov ob različnih nastavitvah parametra k. Pri računanju razdalj je uporabljena evklidska razdalja.

Tabela 4.3: Primer k najbližjih sosedov.

Učne instance

Sist. krvni tlak (mmHg)	*Tel. temp. (°C)*	*Bolan*	*Evklidska*		*Manhattanska*	
			d	*Rang*	*d*	*Rang*
82	38,0	Da	28,0	17	28,7	17
85	36,4	Da	25,1	16	27,3	16
89	36,7	Da	21,1	14	23,0	14
94	38,2	Da	16,0	11	16,5	11
95	38,2	Da	15,0	9	15,5	9
95	37,6	Ne	15,0	10	16,1	10
100	36,6	Ne	10,2	7	12,1	7
104	35,5	Ne	6,8	5	9,2	6
108	35,7	Ne	3,6	3	5,0	3
108	38,4	Da	2,0	2	2,3	2
109	39,4	Da	1,2	1	1,7	1
113	36,4	Ne	3,8	4	5,3	4
119	38,7	Da	9,0	6	9,0	5
124	37,5	Ne	14,1	8	15,2	8
128	37,8	Ne	18,0	12	18,9	12
129	38,8	Da	19,0	13	19,1	13
135	39,4	Da	25,0	15	25,7	15
140	36,6	Ne	30,1	18	32,1	18
145	36,0	Da	35,1	19	37,7	19
147	36,5	Da	37,1	20	39,2	20

Nova instanca

Sist. krvni tlak (mmHg)	*Tel. temp. (°C)*	*Bolan*
110	38,7	?

Primer klasifikacije pacientov

Sledi primer izračunov algoritma k najbližjih sosedov po korakih na primeru diagnoze bolezni pacientov. Imamo učno množico, kjer merimo sistolični krvni tlak pacientov in njihovo telesno temperaturo. Med kartotekami imamo že zabeležene podatke 20 prejšnjih pacientov in njihove diagnoze, ki so jih postavili zdravniki. Instance in nov pacient so prikazani v tabeli 4.3.

V tabeli je prav tako podan izračun razdalj, evklidske in manhattanske. Izračun obeh razdalj med prvo instanco množice in novo instanco pacienta poteka po sledečem postopku.

$$\begin{aligned} d_E\left(x_N, x_1\right) &= \sqrt{(110-82)^2 + (38{,}7-38{,}0)^2} \\ &= \sqrt{(28)^2 + (0{,}7)^2} \\ &= \sqrt{784 + 0{,}49} = \sqrt{784{,}49} = 28{,}0 \end{aligned}$$

$$\begin{aligned} d_M\left(x_N, x_1\right) &= |110-82| + |38{,}7-38{,}0| \\ &= |28| + |0{,}7| \\ &= 28 + 0{,}7 = 28{,}7 \end{aligned}$$

Razvidno je, da razlika v krvnem tlaku prevladuje pri izračunu razdalje. Do tega pride zaradi različnih enot, posledica pa je, da razdalje atributov z večjimi številskimi vrednostmi prevladujejo nad atributi manjših številskih vrednosti. Če želimo prispevek atributov pri računanju skupne razdalje poenotiti, se je potrebno lotiti standardizacije podatkov, s čimer postavimo vse meritve na enako zalogo vrednosti.

4.3.2 Standardizacija podatkov

Standardizacija (angl. *standardization*) je postopek transformacije podatkov na tak način, da bodo ti po transformaciji imeli povprečje

enako 0 in standardni odklon enak 1. Za standardizacijo podatkov x rabimo njihovo povprečje μ in standardni odklon ρ, ki je definiran sledeče za vseh n instanc.

$$\rho = \sqrt{\frac{(x_1 - \mu)^2 + (x_1 - \mu)^2 + \cdots + (x_n - \mu)^2}{n}}$$

Standardizirane vrednosti z izračunamo iz izvornih podatkov x sledeče.

$$z = \frac{x - \mu}{\rho}$$

Vsak atribut x^1, x^2, …, x^l standardiziramo ločeno – transformirano z njegovim povprečjem in standardnim odklonom. Po standardizaciji vseh atributov (tudi indikacijskih) imajo vsi atributi povprečje v vrednosti 0 in standardni odklon 1. Razdalje posameznih atributov med instancami bodo tako v podobnem intervalu.

Za standardizacijo podatkov pacientov tako potrebujemo podatke povprečja in standardnega odklona instanc, kar prikazuje spodnja enačba. Tabela 4.4 prikazuje vrednosti atributov po standardizaciji.

$$\begin{aligned}
\mu\,(\text{Sistolični krvni pritisk}) &= 112{,}45 \\
\rho\,(\text{Sistolični krvni pritisk}) &= 19{,}58 \\
\mu\,(\text{Telesna temperatura}) &= 37{,}42 \\
\rho\,(\text{Telesna temperatura}) &= 1{,}17
\end{aligned}$$

Podatki po standardizaciji kažejo drugačno sliko. Vrstni red najbližjih instanc podani instanci se spremeni.

Tabela 4.4: Primer k najbližjih sosedov s standardiziranimi podatki.

Standardizirane učne instance

Sist. krvni tlak (mmHg)	*Tel. temp. (°C)*	*Bolan*	*Evklidska*		*Manhattanska*	
			d	*Rang*	*d*	*Rang*
-1,55	0,49	Da	1,55	11	2,03	11
-1,40	-0,87	Da	2,34	15	3,24	17
-1,20	-0,61	Da	2,01	14	2,78	15
-0,94	0,66	Da	0,92	5	1,24	6
-0,89	0,66	Da	0,88	4	1,19	5
-0,89	0,15	Ne	1,21	8	1,70	8
-0,64	-0,70	Ne	1,86	12	2,30	13
-0,43	-1,64	Ne	2,74	19	3,03	16
-0,23	-1,47	Ne	2,56	17	2,66	14
-0,23	0,84	Da	0,28	1	0,36	1
-0,18	1,69	Da	0,60	3	0,65	3
0,03	-0,87	Ne	1,97	13	2,11	12
0,33	1,09	Da	0,46	2	0,46	2
0,59	0,07	Ne	1,25	9	1,74	9
0,79	0,32	Ne	1,20	7	1,69	7
0,85	1,18	Da	0,97	6	1,06	4
1,15	1,69	Da	1,41	10	1,87	10
1,41	-0,70	Ne	2,36	16	3,32	18
1,66	-1,21	Da	2,91	20	4,09	20
1,76	-0,78	Da	2,66	18	3,76	19

Standardizirana nova instanca

Sist. krvni tlak (mmHg)	*Tel. temp. (°C)*	*Bolan*
-0,13	1,09	?

Instanca, ki je v tabeli 4.4 označena s sivo barvo, je pred standardizacijo bila četrta najbližja podani instanci. Pred standardizacijo je namreč bila razlika v temperaturi majhna v primerjavi z razliko v krvnem tlaku. Po standardizaciji pa je razlika v temperaturi mnogo večja, kot razlika pri krvnem tlaku. To je prav, saj sta bili pred standardizacijo zalogi vrednosti obeh meritev drugačni. Realni obseg telesne temperature človeka je približno med 35 in 41 °C, kar predstavlja maksimalno razliko 6 enot. Šest enot razlike pri krvnem tlaku pa ni niti približno realnemu obsegu sistoličnega krvnega tlaka, ki se lahko giblje v intervalu med 80 in 180 mmHg z maksimalno razliko kar 100 enot. Po standardizaciji se krvni tlak giblje med −1,55 ter 1,76 z maksimalno razliko 3,31. To je v skladu z obsegom telesne temperature, ki se po standardizaciji giblje med −1,64 ter 1,69 z maksimalno razliko 3,33.

Standardizacija je le en postopek tehnike, ki jo imenujemo *normalizacija podatkov*. Alternative standardizaciji so *min-max skaliranje*, *centriranje*, *rangiranje* in drugi. Vsak pristop strojnega učenja ni občutljiv na intervale oz. skalo atributov. Med take štejemo algoritme kreacije odločitvenih dreves in naivnega Bayesa. Dobra praksa je, da pri postopku podatkovnega rudarjenja podatke pred obdelavo vedno normaliziramo, saj se s tem izognemo morebitnemu zavajanju algoritmov.

Negativna plat normalizacije podatkov pa je, da ti niso največkrat enostavno interpretabilni. Kakor kažejo standardizirane vrednosti v tabeli 4.4, so vrednosti telesne temperature nesmiselne. To postane še posebej problematično v primeru, ko gradimo model znanja, ki temelji na pravilih (odločitvena pravila ali odločitvena drevesa), saj bodo vrednosti v pravilih standardizirane.

Standardizacija v Pythonu

Sledi primer, kjer za začetek najprej pregledamo povprečne vrednosti in standardne odklone atributov pred standardizacijo.

```
from sklearn.datasets import load_iris

# Naložimo podatke
podatki = load_iris(as_frame=True)

print('Povprečja pred standardizacijo:')
print(podatki.data.mean(axis=0))
print('Standardni odkloni pred standardizacijo:')
print(podatki.data.std(axis=0))
```

```
Povprečja pred standardizacijo:
sepal length (cm) 5.843333
sepal width (cm) 3.057333
petal length (cm) 3.758000
petal width (cm) 1.199333
dtype:  float64
Standardni odkloni pred standardizacijo:
sepal length (cm) 0.828066
sepal width (cm) 0.435866
petal length (cm) 1.765298
petal width (cm) 0.762238
dtype:  float64
```

Sedaj pa te podatke standardiziramo in izpišemo povprečja ter standardne odklone novih vrednosti. Knjižnica `scikit-learn` ponuja kar nekaj načinov transformacije podatkov. Za standardizacijo se uporablja razred `StandardScaler`. S klicem metode `fit` se izračunata povprečje in standardni odklon iz podanih podatkov. Za transformacijo podatkov, pa se uporabi klic metode `transform`, kateri podamo podatke, ki jih želimo transformirati.

```
from sklearn.preprocessing import StandardScaler

# Inicializacija standardizatorja
std = StandardScaler()
# Izračunamo povprečja in standardne odklone atributov
std.fit(podatki.data)

# Standardizacija podatkov
stand_podatki = std.transform(podatki.data)

print('Povprečja po standardizaciji:')
print(stand_podatki.mean(axis=0))
print('Standardni odkloni po standardizaciji:')
print(stand_podatki.std(axis=0))
```

```
Povprečja po standardizaciji:
-4.73695157e-16 -7.81597009e-16 -4.26325641e-16 -4.73695157e-16
Standardni odkloni po standardizaciji:
1.  1.  1.  1.
```

Pregled rezultatov povprečij kaže, da so te zelo blizu vrednosti 0 (10^{-15}), standardni odkloni pa so enaki 1. Če želimo proces izračuna povprečij in standardnih odklonov ter proces transformacije podatkov združiti, lahko uporabimo metodo `fit_transform`, ki na podanih podatkih izračuna vmesne vrednosti in jih vrne transformirane.

```
stand_podatki = std.fit_transform(podatki.data)
```

4.3.3 Določitev razreda podane instance

Po izračunu razdalj in določitvi rangov glede na bližino do nove instance sledi določitev razreda (klasifikacija) nove instance. Pri tem procesu igra pomembno vlogo določitev vrednosti parametra k, ki nam pove, koliko najbližjih instanc upoštevamo pri klasifikaciji. Vseh k najbližjih instanc bo namreč glasovalo za razred nove instance – vsaka instanca bo dala glas za razred, kateremu le-ta pripada.

Če je parameter $k = 1$, potem vzamemo le najbližjo instanco in bo novi instanci dodeljen razred te instance. Če se navežemo na primer iz tabele 4.4, je pri izračunu obeh razdalj najbližja ista instanca – ta je razreda "Da" kar pomeni, da tudi novo instanco klasificiramo v razred "Da". Tabela 4.5 prikazuje rezultate klasifikacije pri različnih nastavitvah – pri standardiziranih in nestandardiziranih podatkih, pri različnih vrednostih k in pri evklidski ter manhattanski razdalji.

Tabela 4.5: Klasifikacija s k najbližjih sosedov pri različnih nastavitvah.

		Evklidska			*Manhattanska*		
Podatki	*k*	*Da*	*Ne*	*Rezultat*	*Da*	*Ne*	*Rezultat*
Nestandardizirani	1	1	0	Da	1	0	Da
	2	2	0	Da	2	0	Da
	3	2	1	Da	2	1	Da
	4	2	2	Da/Ne	2	2	Da/Ne
	5	2	3	Ne	3	2	Da
Standardizirani	1	1	0	Da	1	0	Da
	2	2	0	Da	2	0	Da
	3	3	0	Da	3	0	Da
	4	4	0	Da	4	0	Da
	5	5	0	Da	5	0	Da

Tabela 4.5 kaže zanimive rezultate. Najprej poglejmo nestandardizirane podatke. Tako pri evklidski, kot tudi pri manhattanski razdalji, se z večanjem števila najbližjih instanc, ki se upoštevajo pri klasifikaciji, veča tudi negotovost, saj iz razreda "Da" pri upoštevanju le ene najbližje instance ($k = 1$) preidemo do negotovosti, ko upoštevamo štiri najbližje instance ($k = 4$), pa vse do spremembe odločitve v razred "Ne", ko upoštevamo pet najbližjih instanc ($k = 5$) pri evklidski razdalji. Ti rezultati nam kažejo tudi razliko med evklidsko in manhattansko razdaljo, saj se odločitve končnega razreda instance ne skladajo pri $k = 5$. Hkrati pa se moramo soočiti še z negotovostjo

pri $k = 4$. Ko pridemo do neodločenega izida, se algoritem odloči za en naključen razred v vodstvu (tisti, ki ima manjši indeks), kar pa ni vedno najboljša odločitev. Če bi ročno želeli izničiti možnosti za neodločene izide in naključne odločitve med njimi, bi algoritem preprosto zagnali še na drugih nastavitvah vrednosti k in pogledali, kakšen je končen razred tedaj. Ko imamo opravek z binarno klasifikacijo (delitev instanc v dva razreda), pa se neodločenih izidov lahko znebimo z liho vrednostjo k.

Po drugi strani pa ob pregledu rezultatov po standardizaciji vidimo večjo stabilnost, saj obstaja konsenz pri vseh nastavitvah k in pri obeh tipih razdalje. Ta pristop se tako izkaže kot bolj robusten na manjše spremembe, pa tudi konceptualno je primernejši, saj vsi atributi instanc enakovredno vplivajo na odločitev klasifikacije.

4.4 Uporaba *k* najbližjih sosedov v Pythonu

Algoritem k najbližjih sosedov je v knjižnici `scikit-learn` implementiran z razredom `KNeighborsClassifier`. Že pri inicializaciji primerka tega razreda določimo k število najbližjih sosedov s parametrom `n_neighbors` in način računanja razdalje s parametrom `metric`.

```
from sklearn.neighbors import KNeighborsClassifier
klasif = KNeighborsClassifier(n_neighbors=3,
                              metric='manhattan')
```

Pri računanju razdalje lahko uporabimo evklidsko razdaljo z `euclidean`, mahattansko z `manhattan` in kosinusno razdaljo s `cosine`.

S klicem metode `fit` in podajo podatkov instanc `X_u` in njihovih razredov `y_u` te shranimo in bodo služili za izračun najbližjih sosedov. Metodo `predict` pa kličemo s podajo množice instanc `X_t`, ki jih želimo klasificirati.

```
klasif.fit(X_u, y_u)
napovedi = klasif.predict(X_t)
```

To je tudi standardni postopek uporabe drugih algoritmov klasifikacije, regresije in gručenja v knjižnici `scikit-learn`:

1. Nastavitve definiramo v konstruktorju.
2. Model znanja zgradimo s `fit(podatki_instanc, resitve_instanc)`.
3. Model znanja uporabimo s `predict(podatki_novih_instanc)`.

Algoritmi pa lahko imajo tudi sebi specifične metode. V primeru k najbližjih sosedov je njemu posebna metoda `kneighbors`, kateri podamo instance, za katere iščemo najbližje sosede, ter parameter `n_neighbors`, s katerim povemo, koliko najbližjih sosedov iščemo iz nabora vseh instanc podanih že prej v metodi `fit`. Rezultat sta dva seznama: (1) seznam razdalj od izbranih instanc do najbližjih sosedov v naraščajočem vrstnem redu glede na razdaljo, ter (2) seznam indeksov najbližjih instanc, ponovno od najbližjega naprej.

```
klasif.fit(X_u, y_u)
razdalje, sosedi = klasif.kneighbors(nove_instance,
                                     n_neighbors=3)
```

Sledi pregled uporabe klasifikacijskega algoritma k najbližjih sosedov za namen iskanja petih najbližjih sosedov eni instanci. Najprej s sledečo kodo naložimo instance podatkovne množice *Iris*, iz nje izberemo instanco v vrstici z indeksom 133 ter jo odstranimo iz podatkovne množice.

```
from sklearn.datasets import load_iris

# Naložimo podatke
podatki = load_iris(as_frame=True)

# Izberemo eno instanco
izbrana = 133
X_izbrana = podatki.data.iloc[izbrana,:]
y_izbrana = podatki.target.iloc[izbrana]

# Izbrano instanco izločimo iz ostalih podatkov
X_ostali = podatki.data.drop(izbrana, axis=0)
y_ostali = podatki.target.drop(izbrana)
```

Izbira vrednosti iz polja poteka s pomočjo klica `iloc[vrstica, stolpec]`, kamor podamo indeks vrstice in indeks stolpca. Če želimo izbrati celotno vrsto, podamo namesto indeksa kar dvopičje `:`. Tako s `podatki.data.iloc[:, 12]` izberemo stolpec z indeksom 12, s klicem `podatki.data.iloc[8, :]` pa izberemo vrstico z indeksom 8.

Če želimo izbrati več vrednosti, pa namesto števila na mestu vrstice in stolpca podamo polje indeksov. S klicem `podatki.data.iloc[[2, 5, 8], :]` izberemo vrstice s temi indeksi. In obratno, s klicem `podatki.data.iloc[:, [3, 6, 9]]` se vrnejo stolpci z indeksi 3, 6 in 9.

Pri izbiri vrednosti iz vektorja podamo le eno vrednost, saj ima ta le eno dimenzijo. Tako nam klic `podatki.target.iloc[[2, 5, 8]]` vrne podatke z indeksi 2, 5 in 8.

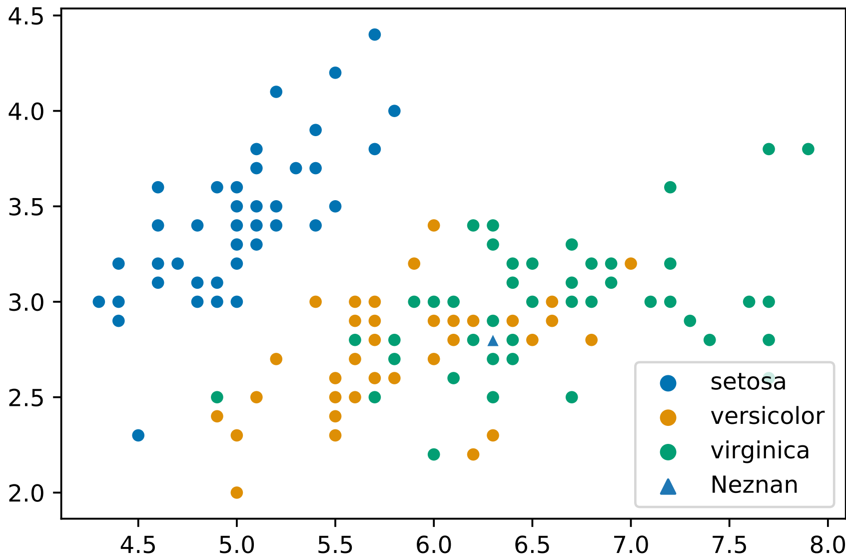
Z metodo `podatki.data.drop(izbrana, axis=0)` odstranimo instanco z indeksom `izbrana` iz podatkov `podatki.data`. Parameter `axis` določa, po kateri osi izbrišemo podatke – če je podana 0, izbrišemo vrstico, če pa 1, pa izbrišemo stolpec. Pri izbrisu iz `podatki.target` parametra `axis` ni potrebno podati, saj so podatki v obliki vektorja, ki ima le eno dimenzijo.

4.4.1 Vizualizacija podatkov

S pomočjo knjižnice `seaborn` lahko enostavno vizualiziramo instance. Z dvema klicema metode `scatterplot` se en na drugega izrišeta dva grafa raztrosa. S parametroma `x` in `y` podamo podatke, ki naj so izrisani na teh oseh.

```
import seaborn as sns

x_os, y_os = 0, 1

# Izrišemo ostale instance
sns.scatterplot(x=X_ostali.iloc[:,x_os],
                y=X_ostali.iloc[:,y_os],
                hue=podatki.target_names[y_ostali],
                palette='colorblind')

# Izrišemo eno izbrano instanco
sns.scatterplot(x=[X_izbrana.iloc[x_os]],
                y=[X_izbrana.iloc[y_os]],
                hue=['Neznan'], style=['Neznan'],
                markers={'Neznan': '^'})
```

Pomanjkljivost grafične predstavitve podatkov je, da lahko izrišemo instance glede na omejeno število njihovih atributov. V našem primeru imamo dvodimenzionalen graf, kjer smo posamezne osi določili s spremenljivkama `x_os` in `y_os`. Parameter `hue` prejme podatke, ki bodo narisane označbe delili glede na barvo – v našem primeru so to podatki o razredu. Parameter `style` pa prejme podatke, ki določajo stil označbe, ki jih definiramo s parametrom `markers`. Barve označb določamo s podajanjem teme v parameter `palette`. Slika, ki nastane ob tem klicu, je sledeča.



Slika 4.7: Instance podatkovne zbirke *Iris*. Horizontalna x os predstavlja prvi atribut, vertikalna y os pa drugi atribut podatkovne množice. Barve ločijo razrede instanc, s trikotnikom pa je označena instanca, ki jo želimo klasificirati.

4.4.2 Učenje in uporaba modela znanja

Učenje modela in napoved razreda instance na indeksu 133 poteka na sledeč način. Najprej s konstruktorjem določimo vrednost k na pet najbližjih sosedov po izračunu manhattanske razdalje. Temu sledi klic metode `fit`, kateri podamo podatke instanc `X_ostali` in njihove razrede `y_ostali`. Ker metoda `predict` pričakuje več instanc, dodamo instanco `X_izbrana` najprej v seznam in ta seznam podamo v klic metode `predict([X_izbrana])`. Če `X_izbrana` ne bi bil vektor, ampak polje, bi klic metode potekal brez ovijanja v seznam `predict(X_izbrana)`.

```
from sklearn.neighbors import KNeighborsClassifier

# Inicializiramo klasifikator
knn = KNeighborsClassifier(n_neighbors=5,
                           metric='manhattan')
# Shranimo instance za primerjavo
knn.fit(X_ostali, y_ostali)

# Napovemo razred izbrane instance
napoved = knn.predict([X_izbrana])

print(f'KNN je napovedal, da je instanca razreda
        {podatki.target_names[napoved]}.')
print(f'Ta instanca je dejansko razreda
        {podatki.target_names[y_izbrana]}.')
```

```
KNN je napovedal, da je instanca razreda ['versicolor'].
Ta instanca je dejansko razreda virginica.
```

Klasifikator je napovedal napačen razred za to instanco. Preglejmo katerih pet instanc je po izračunu manhattanske razdalje najbližje naši izbrani instanci iz vrstice z indeksom 133.

```
# Izbrani instanci najdemo pet najbližjih sosedov
razdalje, sosedi = knn.kneighbors([X_izbrana],
                           n_neighbors=5)
print(f'Pet najbližjih: {sosedi}')
print(f'Razdalje od najbližjih do izbrane: {razdalje}')
```

```
Pet najbližjih:  [[ 72 83 123 126 54]]
Razdalje od najbližjih do izbrane:  [[0.5 0.5 0.6 0.7 0.7]]
```

Pet najbližjih instanc po manhattanski razdalji so instance z indeksi 72, 83, 123, 126 in 54. Rezultati razdalj pa so kar manhattanske razdalje teh sosedov do podane instance.

4.4.3 Vpliv nastavitev na rezultate

Poglejmo, kako se spremeni seznam petih najbližjih instanc, ko spreminjamo način izračuna razdalje med instancami.

```
for razdalja in ['euclidean', 'manhattan', 'cosine']:
    knn = KNeighborsClassifier(n_neighbors=5,
                               metric=razdalja)
    knn.fit(X_ostali, y_ostali)
    razdalje, najblizje = knn.kneighbors([X_izbrana],
                                          n_neighbors=5)

    print(f'Najbližje instance po {razdalja} so {
                               najblizje}')
    print(f'Razdalje so {razdalje}')
```

```
Najbližje instance po euclidean so [[ 83 72 123 126 127]]
Razdalje so [[0.33166248 0.36055513 0.37416574 0.43588989
0.45825757]]
Najbližje instance po manhattan so [[ 72 83 123 126 54]]
Razdalje so [[0.5 0.5 0.6 0.7 0.7]]
Najbližje instance po cosine so [[125 129 90 131 83]]
Razdalje so [[0.0001158 0.00022532 0.00033682 0.00034546
0.00039085]]
```

Izračun petih najbližjih sosedov po evklidski in manhattanski razdalji vrne podobne rezultate. Instanca z indeksom 83 je najbližja izbrani po izračunu evklidske razdalje in je druga najbližja po manhattanski razdalji – mesto si izmenja z instanco 72. Tretje in četrto mesto sta v obeh razdaljah zasedli instanci 123 in 125. Peto mesto ima v primeru evklidske razdalje instanca 127, v primeru manhattanske pa instanca 54. Če sta instanci 127 in 54 drugačnega razreda, lahko ta razlika vpliva na razred izbrane instance.

Seznam najbližjih instanc po izračunu kosinusne razdalje pa je skorajda popolnoma drugačen od ostalih dveh – le instanca 83 je v vseh treh seznamih. Vse ostale štiri instance so v seznamu kosinusne razdalje druge v primerjavi s seznamoma evklidske in manhattanske razda-

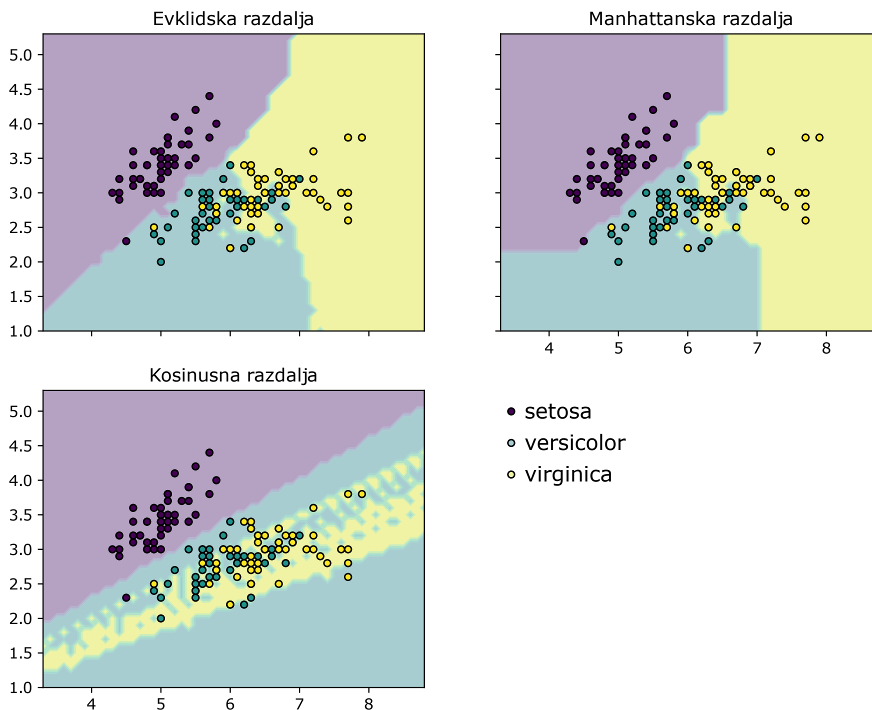
lje. Ti rezultati nam kažejo, kako pomembna je odločitev glede načina izračuna razdalje.

Slika 4.8 kaže delitev območja razredov glede na način računanja razdalje. Pri kreaciji slike sta bila upoštevana prva dva atributa in vrednost števila sosedov k je bila nastavljena na 5. Slika kaže, da večjih razlik med evklidsko in manhattansko razdaljo pri tej podatkovni množici in nastavitvi k ni. Delitev območij je relativno jasna, z nekoliko večjim prekrivanjem v sredini, kjer so si instance razredov *versicolor* in *virginica* zelo podobne. Po drugi strani je razvidno, da delitev po izračunu glede na kosinusno razdaljo ni primerna za podano podatkovno množico, saj sta območji *versicolor* in *virginica* preveč prepleteni in je klasifikacija instanc v tem območju nestabilna.

Poglejmo si še, kako vrednost k vpliva na končno klasifikacijo izbrane instance.

```
for k in [1, 3, 5, 8, 10]:
    knn = KNeighborsClassifier(n_neighbors=k,
                               metric='euclidean')
    knn.fit(X_ostali, y_ostali)
    napoved = knn.predict([X_izbrana])

    print(f'KNN z k={k} je napovedal, da je izbrana
      instanca razreda {podatki.target_names[napoved]}')
```

```
KNN z k=1 je napovedal, da je izbrana instanca razreda
['versicolor']
KNN z k=3 je napovedal, da je izbrana instanca razreda
['versicolor']
KNN z k=5 je napovedal, da je izbrana instanca razreda ['virginica']
KNN z k=8 je napovedal, da je izbrana instanca razreda
['versicolor']
KNN z k=10 je napovedal, da je izbrana instanca razreda
['virginica']
```



Slika 4.8: Delitev na območja razredov glede na način računanja razdalj med instancami.

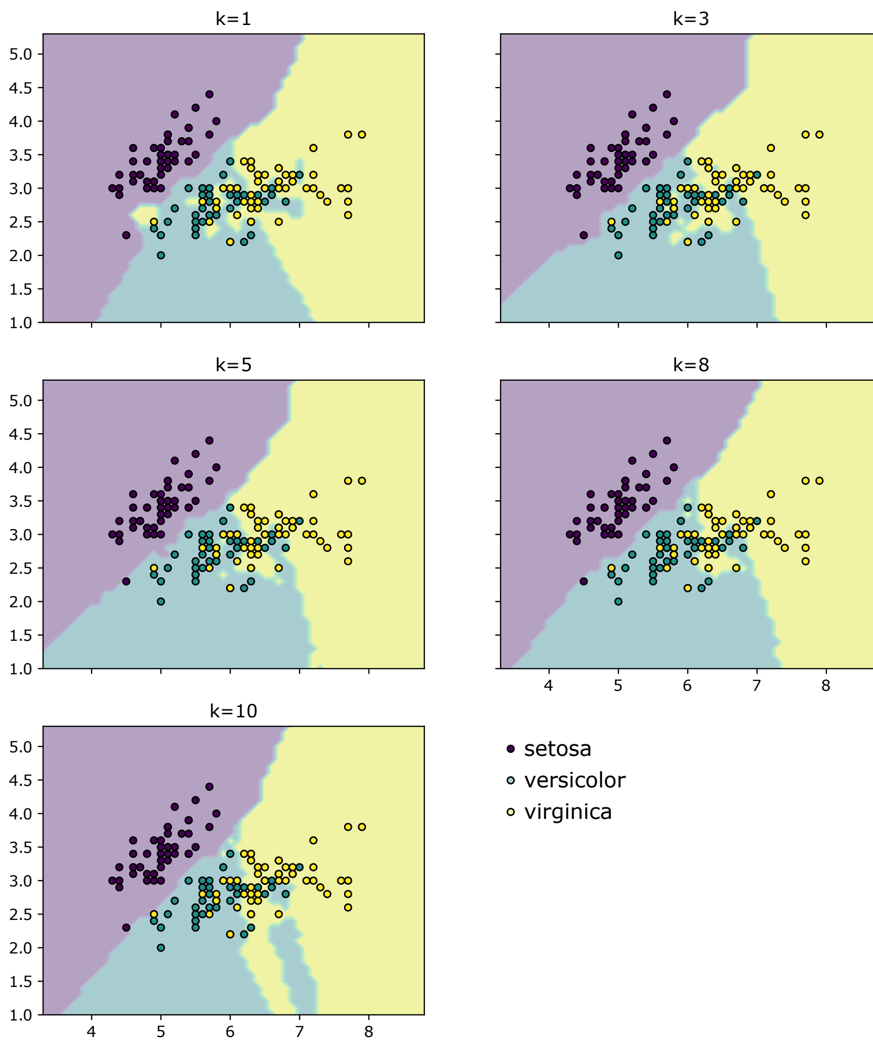
Pri spreminjanju števila najbližjih sosedov ne vidimo konsenza. Razred izbrane instance z indeksom 133 se namreč spreminja iz (napačnega) razreda *versicolor* ob glasovanju enega, treh in osmih najbližjih sosedov, v (pravilen) razred *virginica* ob glasovanju petih ali desetih najbližjih sosedov.

Ponovno je razvidno, da nastavitev igra vlogo pri napovedih algoritma. Vrednost k se največkrat določi po preizkušanju, vsekakor pa mora biti vrednost smiselna (če je k prevelik, bo v resnici algoritem vrnil najpogostejši razred).

Preveč optimiziranja nastavitev za namen boljše klasifikacije ene instance je nesmiselno. Instanca indeksa 133 je bila izbrana namenoma, saj se napovedi njenega razreda zelo spreminjajo ob drugačnih nastavitvah. Večji del instanc dobi enako napoved razreda, ne glede na tip razdalje in k.

To nam pove več o tej instanci kot pa o samem algoritmu. Mogoče je ta nekoliko nenavadna, ali pa je bila že v osnovi (s strani ekspertov) klasificirana napačno. Iz tega sledi, da je smiselno gledati rezultate in kakovost klasifikacije na več instancah, ne le na eni – kaj pa če je ta izbrana nenavadno. V naslednjem poglavju bomo spoznali načine ovrednotenja kakovosti klasifikacije in proces pravilne izbire instanc, s katerimi testiramo izbrani algoritem klasifikacije in njegovih nastavitev.

Slika 4.9 prikazuje različna območja razredov glede na različne vrednosti števila sosedov k. Pri kreaciji slike sta bila ponovno upoštevana le prva dva atributa podatkov in računanje evklidskih razdalj. Iz slike je razvidno, da majhne vrednosti k prinesejo kar nekaj majhnih področij klasifikacije. Po drugi strani pa je razvidno, da nastavitev števila sosedov na 10 nekoliko pokvari delitev območja.



Slika 4.9: Delitev na območja razredov glede na število najbližjih sosedov k: 1, 3, 5, 8 in 10.

4.4.4 Vpliv standardizacije podatkov na rezultate

Poglejmo si, če se klasifikacija kaj spremeni, če uporabimo standardizirane podatke.

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import StandardScaler

# Standardiziramo podatke
standardizator = StandardScaler()
standardizator.fit(X_ostali)
X_ostali_s = standardizator.transform(X_ostali)
X_izbrana_s = standardizator.transform([X_izbrana])

for razdalja in ['euclidean', 'manhattan', 'cosine']:
    for k in [1, 3, 5]:
        knn = KNeighborsClassifier(n_neighbors=k,
                                   metric=razdalja)
        knn.fit(X_ostali_s, y_ostali)
        nap = knn.predict(X_izbrana_s)
        print(f'KNN metric={razdalja}, k={k} je
          napovedal razred {podatki.target_names[nap]}')
```

```
KNN metric=euclidean, k=1 je napovedal razred ['versicolor']
KNN metric=euclidean, k=3 je napovedal razred ['versicolor']
KNN metric=euclidean, k=5 je napovedal razred ['versicolor']
KNN metric=manhattan, k=1 je napovedal razred ['versicolor']
KNN metric=manhattan, k=3 je napovedal razred ['versicolor']
KNN metric=manhattan, k=5 je napovedal razred ['versicolor']
KNN metric=cosine, k=1 je napovedal razred ['versicolor']
KNN metric=cosine, k=3 je napovedal razred ['virginica']
KNN metric=cosine, k=5 je napovedal razred ['versicolor']
```

Napovedi so mnogo bolj robustne na spreminjanje nastavitev klasifikacijskega algoritma, ko imamo opravek s standardiziranimi podatki. Namreč, najpogosteje napovedan razred je bil *versicolor*. Čeprav je napoved razreda napačna, imamo raje robustne napovedi, kot pa take, ki so preveč odvisne od nastavitev algoritma.

4.4.5 Enovit primer klasifikacije več instanc

Sledi enovit primer klasifikacije več instanc iz *Iris* podatkovne zbirke, kjer so podatki tudi standardizirani.

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.datasets import load_iris
import numpy as np
from sklearn.preprocessing import StandardScaler

# Naložimo podatke
podatki = load_iris(as_frame=True)

# Izberemo več instanc
izbrane = [1, 31, 61, 91, 121]
X_izbrane = podatki.data.iloc[izbrane,:]
y_izbrane = podatki.target.iloc[izbrane]

# Izbrano instanco izločimo iz ostalih podatkov
X_ostali = podatki.data.drop(izbrane, axis=0)
y_ostali = podatki.target.drop(izbrane)

# Standardiziramo podatke
standardizator = StandardScaler()
standardizator.fit(X_ostali)
X_ostali_s = standardizator.transform(X_ostali)
X_izbrane_s = standardizator.transform(X_izbrane)

# Zgradimo klasifikator in napovedmo razrede
knn = KNeighborsClassifier(n_neighbors=3, metric='
                           euclidean')
knn.fit(X_ostali_s, y_ostali)
napovedi = knn.predict(X_izbrane_s)

for i, napoved, dejansko in zip(izbrane, napovedi,
                                y_izbrane):
    print(f'Instanca {i} je klasificirana kot
            {podatki.target_names[napoved]} dejansko pa
            je {podatki.target_names[dejansko]}.')
```

Izpis zgornje kode je sledeč.

```
Instanca 1 je klasificirana kot setosa dejansko pa je
                            setosa.
Instanca 31 je klasificirana kot setosa dejansko pa je
                            setosa.
Instanca 61 je klasificirana kot versicolor dejansko pa
                            je versicolor.
Instanca 91 je klasificirana kot versicolor dejansko pa
                            je versicolor.
Instanca 121 je klasificirana kot virginica dejansko pa
                            je virginica.
```

4.5 Utrjevanje znanja

V tej sekciji sledijo naloge uporabe k najbližjih sosedov z ročnim izračunom in s programiranjem.

Naloga 4-1

Za lokalnega preprodajalca avtomobilov ustvarjamo model znanja, ki mu bo poenostavil ugotoviti, kateri rabljeni avtomobili so primerni za nakup in kateri niso. Podana je tabela preteklih nakupov tega preprodajalca avtomobilov.

a) Tabelo dopolni z izračuni evklidskih in manhattanskih razdalj izbranega avtomobila do preteklih nakupov. Zapiši range razdalj – od najbližje instance (rang 1) do najbolj oddaljene instance.

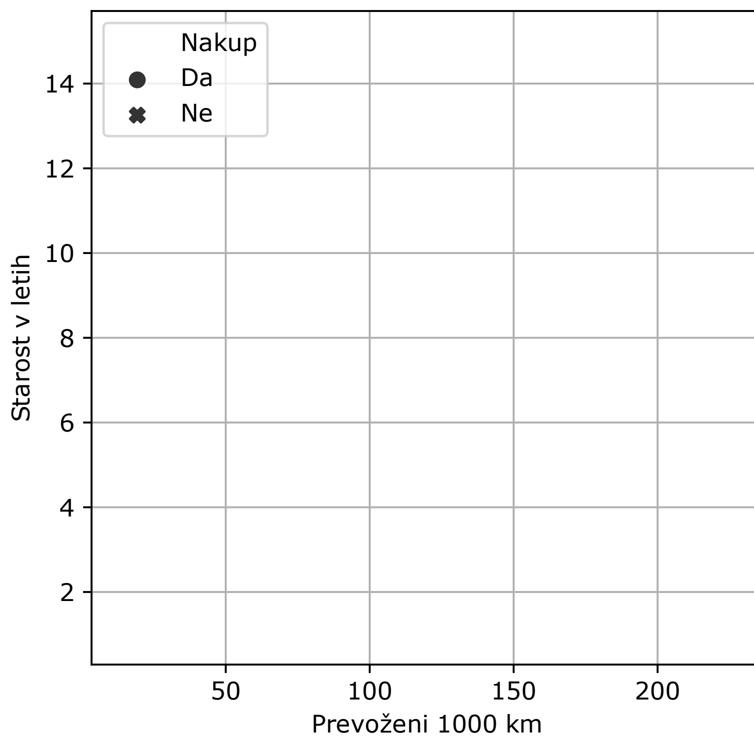
Pretekli nakupi rabljenih avtomobilov

Indeks	*Prevoženi (1000 km)*	*Starost (leta)*	*Nakup*	*Evklidska d*	*Rang*	*Manhattanksa d*	*Rang*
0	14	2	Da				
1	29	1	Ne				
2	38	4	Da				
3	54	5	Da				
4	71	4	Ne				
5	80	6	Da				
6	88	7	Da				
7	103	7	Da				
8	122	8	Ne				
9	127	9	Da				
10	143	9	Ne				
11	148	9	Ne				
12	153	9	Ne				
13	170	11	Ne				
14	183	11	Ne				
15	193	12	Ne				
16	194	14	Da				
17	200	14	Da				
18	214	14	Ne				
29	226	15	Ne				

Izbran avtomobil

Prevoženi (1000 km)	*Starost (leta)*
100	5

b) Na roko izriši graf raztrosa za celotno množico. Označbe instanc prilagodi razredu posamezne instance.



c) V spodnjo tabelo dopiši indekse najbližjih petih instanc za posamezno mero razdalje.

Rangi	*1*	*2*	*3*	*4*	*5*
Evklidska					
Manhattanska					

d) V tabelo dopiši rezultate klasifikacije za posamezno nastavitev k in posamezno mero razdalje.

	Evklidska			*Manhattanska*		
k	*Da*	*Ne*	*Rezultat*	*Da*	*Ne*	*Rezultat*

Naloga 4-2

Uporabi podatke prejšnje naloge o nakupih rabljenih avtomobilov.

a) Izračunaj povprečje in standardni odklon za vsak atribut podatkov.

$\mu\,(\text{Prevoženi km}) =$ $\rho\,(\text{Prevoženi km}) =$

$\mu\,(\text{Starost}) =$ $\rho\,(\text{Starost}) =$

b) V sledečo tabelo vnesi standardizirane podatke.

Standardizirani podatki rabljenih avtomobilov

Indeks	*Prevoženi (1000 km)*	*Starost (leta)*	*Nakup*	*Evklidska*		*Manhattanska*	
				d	*Rang*	*d*	*Rang*
0			Da				
1			Ne				
2			Da				
3			Da				
4			Ne				
5			Da				
6			Da				
7			Da				
8			Ne				
9			Da				
10			Ne				
11			Ne				
12			Ne				
13			Ne				
14			Ne				
15			Ne				
16			Da				
17			Da				
18			Ne				
29			Ne				

Izbran avtomobil

Prevoženi (1000 km)	*Starost (leta)*

c) Tabelo dopolni z evklidskimi in manhattanskimi razdaljami na standardiziranih podatkih.

d) V spodnjo tabelo dopiši rezultate klasifikacije za posamezno nastavitev k in posamezno mero razdalje.

	Evklidska			*Manhattanska*		
k	*Da*	*Ne*	*Rezultat*	*Da*	*Ne*	*Rezultat*

Naloga 4-3

V Pythonu napiši sledečo kodo.

a) S pomočjo knjižnice `scikit-learn` preberi podatke priložene množice *Wine*. Za namene te naloge uporabiš le dva atributa množice: 'magnesium' in 'flavanoids'. Obdrži le ta dva atributa, ostale zavrzi. Pazi, da ne izgubiš podatka o razredu.

b) Zadnjo instanco te množice shrani kot testno instanco, ves ostali del množice pa shrani kot učne podatke. Izpiši prvih pet vrstic učnih podatkov in testno instanco.

c) Izriši graf raztrosa učnih podatkov, kjer je 'magnesium' na x osi ter 'flavanoids' na y osi. Označbe instanc naj so obarvane glede na razred posamezne instance. Pri tem uporabi bodisi knjižnico `seaborn` ali `Matplotlib`.

d) Učne podatke uporabi za učenje sledečih klasifikatorjev k najbližjih sosedov:

- $k = 1$ in evklidska razdalja,
- $k = 1$ in manhattanska razdalja,
- $k = 3$ in evklidska razdalja,
- $k = 3$ in manhattanska razdalja,
- $k = 5$ in evklidska razdalja in
- $k = 5$ in manhattanska razdalja.

Napovedi za testno instanco vsakega klasifikatorja izpiši.

Naloga 4-4

V Pythonu napiši sledečo kodo.

a) S pomočjo knjižnice `scikit-learn` preberi podatke priložene množice *Breast cancer*. Zadnjih deset instanc te množice shrani ločeno kot testno množico, ves ostali del množice pa shrani kot učne podatke. Izpiši prvih pet vrstic učnih podatkov in prvih pet instanc testne množice.

b) Standardiziraj podatke na podlagi učne množice. Standardiziraj tako učne kot testne podatke. Pri tem pazi, da ne izgubiš nestandardiziranih učnih in testnih podatkov. Izpiši prvih pet vrstic standardiziranih učnih podatkov in prvih pet instanc standardizirane testne množice.

c) Nauči dva k najbližjih sosedov klasifikatorja ($k = 3$, evklidska razdalja), enega na nestandardiziranih podatkih, drugega na standardiziranih podatkih. Za vsako instanco v testni množici izpiši v eni vrstici tri vrednosti:

1. dejanski razred iz množice,
2. napoved klasifikatorja naučenega na nestandardiziranih podatkih in
3. napoved klasifikatorja naučenega na nestandardiziranih podatkih.

5 | Kakovost klasifikacije

Pri iskanju najprimernejšega klasifikatorja je potrebno evalvirati vsako izmed metod. Množica podatkov, na podlagi katere smo model znanja zgradili, mora vsebovati tudi rešitve (dejanske razrede) instanc, zato lahko pravilnost klasifikacije modela znanja preverimo kar s primerjavo napovedi in dejanskih razredov. Tak pristop ni optimalen, saj lahko vodi v *prenasičenje* (angl. *overfitting*). Pri prenasičenju algoritem vrne model, ki uspešno klasificira podane podatke, vendar je kakovost klasificiranja na novih podatkih tipično zelo slaba. Pravimo, da se je model preveč prilegel učnim podatkom in ni dovolj splošen za dano problematiko. Prenasičeni modeli imajo visoko stopnjo *variance* (angl. *variance*), kar pomeni, da se preveč prilagajajo majhnemu nihanju in naključnemu šumu v podatkih.

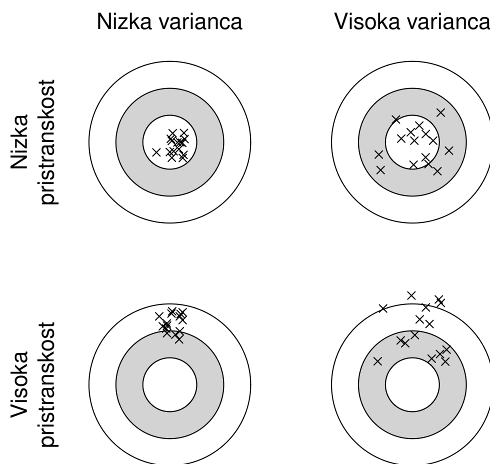
Če primerjamo prenasičene modele z učenjem ljudi, bi lahko rekli, da pride do prenasičenja, če se ljudje naučimo snov dobesedno, ampak brez pravega razumevanja konceptov. Če bi se učili množenja števil, bi se s prenasičenjem naučili na pamet le rezultate množenja števil do 10 (ker so le ti v učbeniku). Ker pa nismo dojeli koncepta množenja,

ne bi poznali rezultata množenja dveh še prej nevidenih števil (primer: $12 * 27$).

Nasprotje prenasičenju pa je *nenasičenost* (angl. *underfitting*). Nenasičeni modeli imajo visoko *pristranskost* (angl. *bias*). To pomeni, da je model preveč splošen, saj ne zazna pomembnih relacij ali zakonitosti (glej sliko 5.1).

Če se ponovno vrnemo na analogijo učenja množenja dveh števil, bi prišli do nenasičenosti, ko ne bi v celoti dojeli koncepta množenja dveh števil – znali bi množiti dve celoštevilski pozitivni števili, ne pa realnih števil (primer: $6{,}4 * 1{,}34$) ali celo negativnih celih števil (primer: $-7 * 21$).

Praviloma imajo klasifikatorji z visoko pristranskostjo nizko stopnjo variance in obratno. Vedno želimo najti klasifikator s pravim ravnovesjem med pristranskostjo in varianco, a ta meja je arbitrarna in jo določimo sami.



Slika 5.1: Prikaz variance in pristranskosti.

5.1 Delitev podatkov

Da se izognemo prekomernemu prileganju podatkom, je potrebno podatke razdeliti na več delov. Na enem izmed delov se klasifikator uči (klasifikacijski algoritem zgradi klasifikacijski model), na drugih podatkih (na takih, ki niso sodelovali pri procesu učenja) pa se preveri kakovost dobljenega modela.

Nadaljujmo z analogijo učenja množenja dveh števil. Učno množico dojemimo kot učno gradivo, ki ga uporabljamo za učenje na izpit. Če bi na izpitu dobili enake primere množenja dveh števil, kot smo jih že videli v učnem gradivu, bi preverjali le, če imamo dober spomin, ne pa tudi, če smo dejansko dojeli postopek množenja dveh števil. Tako naj bo izpit sestavljen iz popolnoma novih primerov množenja dveh števil, kar pa v našem postopku ponazorimo z ločeno testno množico. Če je klasifikator dejansko izluščil vzorce iz podatkov, ne bi smel imeti težav pri klasifikaciji novih podatkov. Če se je učne podatke naučil le na pamet, potem pa ne bo dobro klasificiral testnih podatkov (bo padel na izpitu).

5.1.1 Učna in testna množica

Osnovna razdelitev izvorne množice je z delitvijo na dve podmnožici: *učno množico* (angl. *learning dataset* ali *train dataset*) in *testno množico* (angl. *test dataset*) [13]. Na učni množici le učimo klasifikatorje, ki pa jih nato testiramo in evalviramo le na testni množici – podatki iz testne množice ne smejo sodelovati pri gradnji klasifikacijskih modelov. Kljub deljenju na učno in testno množico lahko dobimo prenasičen model, a je možnost za to mnogo manjša. Učno množico X smo definirali že prej, testna množica X' pa je definirana spodaj.

$$X' = \left\{ \left(x'_1, y'_1 \right), \left(x'_2, y'_2 \right), \ldots, \left(x'_m, y'_m \right) \right\}$$
$$y'_i \in \{razred_1, razred_2, \ldots, razred_k\}$$
$$m = \text{število instanc v testni množici}$$

Klasifikacijski model vsaki testni instanci določi razred $\hat{y}$, ki pa ni nujno enak dejanskemu razredu testne instance y' – v tem primeru naredi napako pri klasifikaciji instance. Cilj je najti tak klasifikacijski model, ki pravilno klasificira čim več testnih instanc (idealno vse).

Delitev na učno in testno množico v Pythonu

V kodi pa razdelimo podatke na učno in testno množico na sledeč način.

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

# Naložimo podatke
podatki = load_iris(as_frame=True)

# Razdelimo podatke na učne in testne
X_u, X_t, y_u, y_t = train_test_split(podatki.data,
                                      podatki.target,
                                      train_size=0.7,
                                      random_state=42)

print(f'Velikost učne množice: {X_u.shape}')
print(f'Velikost učnih razredov: {y_u.shape}')
print(f'Velikost testne množice: {X_t.shape}')
print(f'Velikost testnih razredov: {y_t.shape}')
```

```
Velikost učne množice:  (105, 4)
Velikost učnih razredov:  (105,)
Velikost testne množice:  (45, 4)
Velikost testnih razredov:  (45,)
```

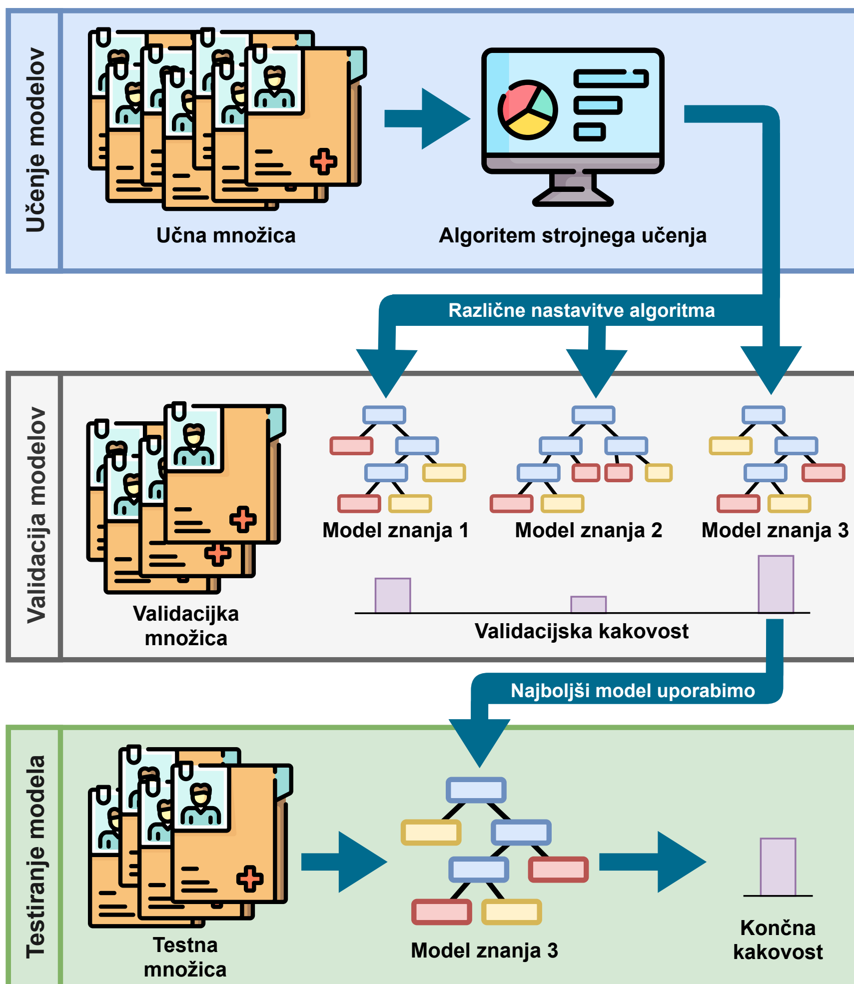
S klicem `train_test_split` podane podatke razdelimo na učno in testno množico na tak način, da zadostimo sledečim kriterijem.

- Če podamo parameter `train_size` in je vrednost tega celo število, bo učna množica štela prav toliko instanc. Primer: `train_size=10` pove, da bo učna množica štela točno 10 instanc.
- Če podamo parameter `train_size` in je vrednost tega realno število med 0 in 1, bo učna množica v velikosti deleža izvorne podane množice. Primer: `train_size=0.8` pove, da bo učna množica 80 % velikosti izvorne množice.
- Če podamo parameter `test_size` in je vrednost tega celo število, bo testna množica štela prav toliko instanc. Primer: `test_size=10` pove, da bo testna množica štela 10 instanc.
- Če podamo parameter `test_size` in je vrednost tega realno število med 0 in 1, bo testna množica v velikosti deleža izvorne podane množice. Primer: `test_size=0.2` pove, da bo testna množica 20 % velikosti izvorne množice.

Klicu `train_test_split` smo podali tako izvorno množico atributov `podatki.data`, kakor tudi izvorne razrede `podatki.target`. S tem zagotovimo, da dobimo vrnjene razdeljene tako atribute instanc (učne `X_u` in testne `X_t`), kakor tudi rešitve (učne `y_u` in testne `y_t`).

5.1.2 Validacijska množica

Če se ukvarjamo s specifičnim problemom podatkovnega rudarjenja, se pri gradnji klasifikacijskih modelov srečamo z optimizacijo parametrov klasifikatorja (na primer definiranja števila sosedov k pri klasifikatorju k najbližjih sosedov). Če parametre prilagajamo glede na rezultate klasifikatorjev na testni množici, delamo napako, saj tako testni podatki sodelujejo pri gradnji optimalnega modela. V tem primeru potrebujemo še *validacijsko množico* (angl. *validation dataset*), na podlagi katere preizkušamo in optimiziramo parametre, šele na koncu pa zgrajen model testiramo na testni množici [14].



Slika 5.2: Delitev množice na učno, validacijsko in testno ter njihova uporaba.

Slika 5.2 prikazuje razdelitev izvorne podatkovne množice na te tri dele: učno množico, validacijsko množico in testno množico. Iz procesa je razvidno, da se nastali modeli znanja najprej preizkusijo na validacijski množici ter se njihova kakovost ovrednoti na tej množici. Šele izbrani (po navadi najboljši) model se uporabi za končno ovrednotenje na testni množici.

Delitev na učno, validacijsko in testno množico v Pythonu

V paketu `scikit-learn` ne obstaja klic, s katerim bi dosegli enostavno delitev na tri dele, zato je potrebno, da smo nekoliko zviti.

V prvem klicu `train_test_split` smo pridobili testno množico `X_t, y_t` ter *začasno učno* množico `X_u1, y_u1`. Razmerje med njima je 4:1 v prid začasne učne množice. V drugem klicu `train_test_split` pa začasno učno množico razdelimo na dejansko učno `X_u, y_u` ter validacijsko množico `X_v, y_v` v razmerju 3:1. Končno razmerje med učno množico, validacijsko množico in testno množico je tako 3:1:1.

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

# Naložimo podatke
podatki = load_iris(as_frame=True)

# Razdelimo podatke na 'začasno učne' in testne
X_u1, X_t, y_u1, y_t = train_test_split(podatki.data,
                                        podatki.target,
                                        train_size=0.8,
                                        random_state=42)

# 'Začasno učne' podatke razdelimo na 'dejanske učne' in
                              validacijske
X_u, X_v, y_u, y_v = train_test_split(X_u1,
                                      y_u1,
                                      train_size=0.75,
                                      random_state=42)
```

Poglejmo kakšne so množice po razdelitvi.

```
print(f'Velikost učne množice: {X_u.shape}')
print(f'Velikost učnih razredov: {y_u.shape}')
print(f'Velikost validacijske množice: {X_v.shape}')
print(f'Velikost validacijskih razredov: {y_v.shape}')
print(f'Velikost testne množice: {X_t.shape}')
print(f'Velikost testnih razredov: {y_t.shape}')
```

```
Velikost učne množice:  (90, 4)
Velikost učnih razredov:  (90,)
Velikost validacijske množice:  (30, 4)
Velikost validacijskih razredov:  (30,)
Velikost testne množice:  (30, 4)
Velikost testnih razredov:  (30,)
```

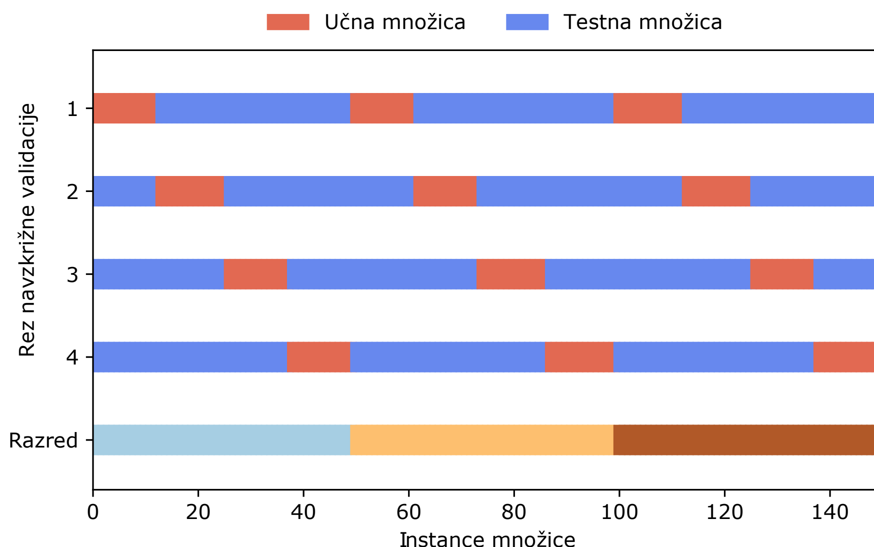
5.1.3 Navzkrižna validacija

Pri deljenju podatkov na učno, testno in validacijsko množico pa lahko zgolj po naključju razdelimo podatke v učno, validacijsko in testno množico, tako da se zgodi ena izmed sledečih stvari:

- kakšen vzorec ni prisoten (v celoti ali v dovolj veliki meri) v učni množici in pri učenju klasifikatorja ta ni zajet v modelu;
- se kakšen, v realnosti neprisoten, vzorec pokaže v učni množici in se ta zapiše v model klasifikacije, v realnosti pa ni uporaben.

Nadaljujemo z analogijo učenja množenja števil. Pri učenju matematične naloge (z rešitvami) razdelimo na dva dela: tiste primere, na katerih se učimo (učno množico) in tiste, na katerih se pred izpitom preverimo, če vemo dovolj (validacijska množica; izpit bo pa testna množica). Če smo po naključju razdelili naloge množenja tako, da se učimo le iz najlažjih (recimo le množenje celih pozitivnih števil), določenega naprednega znanja ne bomo osvojili (množenja negativnih ali realnih števil).

Da zmanjšamo možnost za ponesrečeno razdelitev množice, to razdelimo na več načinov – na n načinov. Tako iz ene množice dobimo n učnih množic in n testnih množic. Ta proces imenujemo *navzkrižna validacija* (angl. *cross-validation*) in steče tako, da celotno množico razdelimo na n delov – na n *rezov* (angl. *folds*) [15]. Vsak rez je enkrat v vlogi testne množice, v vseh ostalih primerih pa je v kombinaciji z ostalimi rezi del učne množice. Posledično je tudi vsaka instanca enkrat v testni množici, v ostalih primerih pa je vedno v učni množici. Slika 5.3 prikazuje razdelitev celotne množice na štiri reze za namen navzkrižne validacije.



Slika 5.3: Stratificirana navzkrižna validacija. Vertikalna Y os prikazuje pripadnost instanc bodisi v učno (modra) ali testno (rdeča) množico pri posameznem rezu.

Pri naključni delitvi na reze se zna zgoditi, da bodo razmerja med razredi različna med rezi. Tega se želimo izogniti, saj lahko nastane situacija, ko so vse instance enega razreda le v enem rezu. Ko bo ta rez v vlogi testnih instanc, klasifikacijski algoritem nima možnosti, da

pravilno klasificira instance tega manjšinskega razreda – saj jih nikoli ne vidi v času učenja. S postopkom *stratifikacije* (angl. *stratification*) poskrbimo, da se množica premeša na tak način, da so razmerja razredov v vseh rezih kar se da enaka.

V Pythonu razdelimo množico s stratificirano navzkrižno validacijo sledeče.

```
from sklearn.datasets import load_iris
from sklearn.model_selection import StratifiedKFold

# Naložimo podatke
podatki = load_iris(as_frame=True)

# Stratificirana navzkrižna validacija s štirimi rezi
skf = StratifiedKFold(n_splits=4)

rez = 1 # S tem bomo šteli reze
instanca = 13 # Pregledovali bomo instanco na mestu 13
print(f'Zanima nas instanca {instanca} s podatki:')
print(podatki.data.iloc[instanca,:])

for ucne, testne in skf.split(podatki.data,
                              podatki.target):
    if instanca in ucne:
        print(f'Instanca {instanca} je v {rez}. delitvi
                v učni množici.')
    elif instanca in testne:
        print(f'Instanca {instanca} je v {rez}. delitvi
                v testni množici.')
    rez = rez + 1
```

Izpis zgornje kode je sledeč.

```
Zanima nas instanca 13 s podatki:
 sepal length (cm)    4.3
 sepal width (cm)     3.0
 petal length (cm)    1.1
 petal width (cm)     0.1
Name: 13, dtype: float64
 Instanca 13 je v 1. delitvi v učni množici.
 Instanca 13 je v 2. delitvi v testni množici.
 Instanca 13 je v 3. delitvi v učni množici.
 Instanca 13 je v 4. delitvi v učni množici.
```

Z uporabo razreda `StratifiedKFold` in njegove metode `split` razdelimo podane podatke na reze. Z vrednostjo `n_splits` ob inicializaciji podamo, na koliko rezov delimo množico. V `for` zanki tako na vsaki učni množici naučimo svoj model klasifikacije, ki ga evalviramo na testni množici tistega reza. Tako dobimo n klasifikacijskih modelov, vsak se je naučil na nekoliko drugačni učni množici in vsak se je evalviral na popolnoma drugačni testni množici. Iz tega sledi, da dobimo tudi n rezultatov klasifikacije na testnih množicah. Da pa ocenimo celokupno kakovost podanega klasifikacijskega algoritma, pa vse rezultate preprosto povprečimo. Sledi poglavje, ki pregleda, kako sploh merimo kakovost klasifikacije, razdeljeno glede na število razredov, v katere model klasifikacije razvršča instance.

5.2 Metrike klasifikacije

Ko imamo napovedi modela, lahko ovrednotimo oz. evalviramo kakovost te napovedi. To naredimo s pomočjo *klasifikacijskih metrik* (angl. *classification metrics*) [16]–[19]. Sledi pregled različnih klasifikacijskih metrik, najprej, ko imamo opravka s klasifikacijo v dva razreda (dvorazredna ali binarna klasifikacija) in nato, ko imamo opravka z več kot dvema razredoma (večrazredna klasifikacija).

5.2.1 Binarna klasifikacija

Pri binarnih klasifikatorjih gradimo model za klasifikacijo instanc v dva razreda. Ko imamo model zgrajen, ga ocenimo s pomočjo testnih podatkov in iz rezultatov zgradimo *matriko zmede* ali *kontingenčno tabelo* (angl. *confusion matrix* ali *contingency table*), ki je prikazana v tabeli 5.1.

Tabela 5.1: Matrika zmede za dva razreda: A in B.

		Napovedano	
		A	B
Dejansko	A	TP	FN
	B	FP	TN

V danem primeru iz tabele 5.1 klasificiramo podatke v dva razreda: A in B. Vsaka celica v tabeli prikazuje število instanc. Vrstice matrike kažejo dejansko pripadnost instanc v razrede, stolpci pa povedo razporeditev instanc v razrede s pomočjo klasifikatorja. Enemu razredu pripišemo lastnost pozitivnega razreda, drugemu pa lastnost negativnega razreda. Ta določitev je arbitrarna in je odvisna od problematike reševanja – pravilne določitve ni in bi tudi druga delitev v pozitivne in negativne vrnila prave rezultate.

```
from sklearn.metrics import confusion_matrix

# Podamo dejanske razrede in napovedi klasifikatorja
y_dejanski = ['A', 'A', 'B', 'B', 'A', 'B']
y_napovedan = ['A', 'A', 'B', 'A', 'A', 'B']

# Izračunamo matriko zmede
mat_zmede = confusion_matrix(y_dejanski, y_napovedan,
                             labels=['A', 'B'])
print(mat_zmede)
```

```
[[3 0]
[1 2]]
```

Klicu `confusion_matrix` podamo vsaj dve vrednosti: polje dejanskih razredov in polje napovedi. Pomembno je, da so elementi v obeh poljih v enakem vrstnem redu (prvi element v obeh poljih kaže tako dejanski razred kot napoved prvega elementa). S parametrom `labels` pa podamo vrstni red razredov pri kreaciji matrike zmede – če tega ne podamo, bo vrstni red razredov po abecednem vrstnem redu.

V primeru iz kode vidimo, da je klasifikacijski model za tri instance razreda A povedal, da so res razreda A; za dve instanci razreda B je povedal, da sta res razreda B; ter za eno instanco razreda B je napačno povedal, da je ta razreda A. Če privzamemo, da je razred A pozitiven, matrika zmede binarne klasifikacije vsebuje štiri vrednosti, ki kažejo količino instanc glede na rezultate klasifikacije:

- **TP** – *pravilno* klasificirane kot *pozitivne* (angl. *true positives*),
- **FP** – *napačno* klasificirane kot *pozitivne* (angl. *false positives*),
- **TN** – *pravilno* klasificirane kot *negativne* (angl. *true negatives*),
- **FN** – *napačno* klasificirane kot *negativne* (angl. *false negatives*).

Razširimo prejšnji primer z naslednjo kodo, ki nam vrne te štiri vrednosti (deluje le pri binarni klasifikaciji). Klicu metode `confusion_matrix` dodamo še klic metode `ravel`, ki dobljeno metriko

zloži v enodimenzionalno polje (najprej vse iz prve vrstice, nato vse iz druge vrstice ...).

```
# Izračunamo vrednosti napovedi
tp, fn, fp, tn = confusion_matrix(y_dejanski,
                                  y_napovedan,
                                  labels=['A', 'B']).ravel()

print(f'TP: {tp}')
print(f'TN: {tn}')
print(f'FP: {fp}')
print(f'FN: {fn}')
```

```
TP: 3
TN: 2
FP: 1
FN: 0
```

S pomočjo teh štirih vrednosti lahko izračunamo različne *metrike klasifikacije* (angl. *classification metrics*) oziroma vrednosti, ki nam povedo, kako kakovosten je klasifikacijski model. Teh metrik je več in vsaka služi svojemu namenu – obstajajo tudi različni pogledi, kaj sploh pomeni *kakovosten* model znanja.

Točnost klasifikacije in delež napake

Prva taka metrika je *točnost* klasifikacije (angl. *accuracy*), ki prikazuje, kolikšen deležen instanc je klasificiran v dejanski razred – delež pravilno klasificiranih instanc. Formula točnosti je prikazana v sledeči enačbi in se izračuna kot ulomek števila pravilno klasificiranih instanc ($TP + TN$) deljeno s številom vseh klasificiranih instanc ($TP+FP+TN+FN$). Metriko točnost želimo maksimirati in težimo k temu, da je čim bližje vrednosti 1 [20].

$$točnost = \frac{TP + TN}{TP + FP + TN + FN}$$

Metrika, ki kaže nasprotno vrednost točnosti, je *delež napake* (angl. *error rate*); pravimo tudi, da je obratno sorazmerna kot točnost. Predstavlja delež nepravilno klasificiranih instanc, kar je prikazano v spodnji enačbi. Lahko pa delež napake izračunamo tudi preprosto kot $1 - točnost$. Delež napake je metrika, ki jo minimiziramo in katere idealna vrednost je 0.

$$delež\ napake = \frac{FP + FN}{TP + FP + TN + FN}$$

Prikažimo še izračun točnosti, deleža napake in števila pravilno klasificiranih instanc v Pythonu s klicem `accuracy_score`.

```
from sklearn.metrics import accuracy_score

# Podamo dejanske razrede in napovedi klasifikatorja
y_dejanski = ['A', 'A', 'B', 'B', 'A', 'B']
y_napovedan = ['A', 'A', 'B', 'A', 'A', 'B']

# Izračunamo točnost
tocnost = accuracy_score(y_dejanski, y_napovedan)
delez_napake = 1 - tocnost

# Izračunamo število pravilno klasificiranih (TP+TN)
st_pravilnih = accuracy_score(y_dejanski, y_napovedan,
                              normalize=False)

print(f'Točnost: {tocnost}')
print(f'Delež napake: {delez_napake}')
print(f'Število pravilnih: {st_pravilnih}')
```

```
Točnost: 0.8333333333333334
Delež napake: 0.16666666666666663
Število pravilnih: 5
```

Priklic in preciznost

Naslednji dve metriki, ki ju bomo obravnavali skupaj, sta priklic in preciznost. Obe metriki imata vrednost v intervalu [0,1], kjer je 0 najslabša vrednost, 1 pa najboljša.

Metrika *priklic* (angl. *recall*) nam pove delež pozitivnih instanc, ki so pravilno klasificirane v pozitivni razred. Včasih priklic imenujemo tudi *senzitivnost* (angl. *sensitivity*) ali *delež pravilno klasificiranih pozitivnih instanc* (angl. *true positive rate*). Izračun priklica je prikazan v sledeči enačbi.

$$priklic = \frac{TP}{TP + FN}$$

Poglejmo si uporabo priklica v Pythonu.

```
from sklearn.metrics import recall_score

# Podamo dejanske razrede in napovedi klasifikatorja
y_dejanski = ['A', 'A', 'B', 'B', 'A', 'B']
y_napovedan = ['A', 'A', 'B', 'A', 'A', 'B']

# Izračunamo priklic obeh razredov
priklic_A = recall_score(y_dejanski, y_napovedan,
                         pos_label='A')

priklic_B = recall_score(y_dejanski, y_napovedan,
                         pos_label='B')

print(f'Priklic A: {priklic_A}')
print(f'Priklic B: {priklic_B}')
```

```
Priklic A: 1.0
Priklic B: 0.6666666666666666
```

Podobno kot pri izračunu točnosti, tudi pri klicu metode `recall_score` podamo najprej polje z dejanskimi razredi, čemur pa sledi polje z napovedanimi razredi. Pri binarni klasifikaciji moramo podati tudi pa-

rameter pos_label, s katerim določimo razred, za katerega računamo priklic.

O senzitivnosti (ali občutljivosti) testa v praksi zelo pogosto govorimo, ko imamo opravka s testom, ki ugotavlja prisotnost ali odsotnost določenega obolenja (npr. bolezni, ki jo povzroča virus). Senzitivnost takega testa nam pove kakšen je delež bolnih, ki jih je test našel. Predstavljajmo si primer, ko imamo 100 pacientov, ki so dejansko bolni. Test je za dejansko bolne pokazal, da jih je 80 bolnih, za ostalih 20 pa je test (napačno) trdil, da niso bolni. Tak test ima 80% senzitivnost oz. priklic.

Druga metrika je *preciznost* (angl. *precision*) in nam pove, kolikšen delež instanc, klasificiranih v pozitivni razred, je dejansko pripadnikov pozitivnega razreda. Včasih metriko priklic imenujemo *zaupanje* (angl. *confidence*) ali *točnost pozitivno klasificiranih pozitivnih instanc* (angl. *true positive accuracy*).

$$preciznost = \frac{TP}{TP + FP}$$

Sledeča koda prikazuje uporabo klica precision_score.

```
from sklearn.metrics import precision_score

# Izračunamo preciznost obeh razredov
preciznost_A = precision_score(y_dejanski, y_napovedan,
                               pos_label='A')

preciznost_B = precision_score(y_dejanski, y_napovedan,
                               pos_label='B')

print(f'Preciznost A: {preciznost_A}')
print(f'Preciznost B: {preciznost_B}')
```

```
Preciznost A: 0.75
Preciznost B: 1.0
```

Ponovno se vrnimo na primer testa za ugotavljanje določenega obolenja. Imamo 100 pacientov, ki so dejansko bolni. Test je za 80 izmed teh pokazal, da so bolni, za še 10 dodatnih (dejansko zdravih) pacientov pa je prav tako (napačno) pokazal, da so bolni. Tak test ima 88,89% preciznosti oz. zaupanja.

F-mera

Iz prejšnjih sekcij je razvidno, da obstaja nabor metrik za merjenje kakovosti binarne klasifikacije, od katerih ima vsaka svoj namen. Ko želimo meriti kakovost klasifikacijskega modela v splošnem, pa uporaba večjega števila metrik ni praktična. V ta namen uporabljamo metriko, imenovano *F-mera* (angl. *F-measure* ali *F-score*), ki združi metriki priklic in preciznost v harmonični sredini, kot je prikazano v enačbi spodaj. Metriki preciznost in priklic nista neposredno povezani in prikazujeta različne informacije. Visoka vrednost ene ne pomeni nujno nizke vrednosti druge, zato stremimo k temu, da bi obe metriki bili dovolj visoki. Prednost F-mere je, da združi obe omenjeni metriki skupaj v eno število, kjer se takoj opazi nizka vrednost katerekoli izmed njiju.

$$\begin{aligned} F\text{-}mera_\beta &= (1+\beta^2) \cdot \frac{preciznost \cdot priklic}{(\beta^2 \cdot preciznost) + priklic} \\ &= \frac{(1+\beta^2) \cdot TP}{(1+\beta^2) \cdot TP + \beta^2 \cdot FN + FP} \end{aligned}$$

Ta enačba prikazuje splošno obliko F-mere, kjer lahko posamezno metriko (preciznost ali priklic) utežimo pri končnem izračunu. Vlogo uteži igra vrednost β, ki je višja pri večji pomembnosti metrike preciznost in nižja, ko damo večji poudarek metriki priklica. Tradicionalna oblika metrike F-mera je, ko sta obe vrednosti enako uteženi in imata uravnoteženo vlogo pri izračunu (ko je $\beta = 1$). Največkrat se upo-

rablja prav ta, ki se včasih imenuje tudi F_1-mera in je prikazana s sledečo enačbo.

$$F_1\text{-}mera = 2 \cdot \frac{preciznost \cdot priklic}{preciznost + priklic}$$

Če uporabljamo pri izračunu F-mere vrednost β, ki ni enaka 1, se to zapiše kot podpisana vrednost (F_2-mera ali $F_{0,5}$-mera). Če pa je β enak vrednosti 1, lahko podpisano vrednost izpustimo in zapišemo le F-mera.

Sledeča Python koda prikazuje izračun F-mere z in brez podane β vrednosti. Koda razširja prejšnje primere, kjer so dejanski in napovedani razredi že definirani.

```
from sklearn.metrics import f1_score, fbeta_score

# Izračunamo F-mero in F-meta (s podano beto) razreda A
f1_score_A = f1_score(y_dejanski, y_napovedan,
                      pos_label='A')

f_beta_score_A = fbeta_score(y_dejanski, y_napovedan,
                             pos_label='A', beta=2)

print(f'F1-mera A: {f1_score_A}')
print(f'F-mera (beta=2) A: {f_beta_score_A}')
```

```
F1-mera A: 0.8571428571428571
F-mera (beta=2) B: 0.9375
```

5.2.2 Klasifikacija v več razredov

V prejšnji sekciji smo naredili pregled metrik za ocenjevanje kakovosti modelov klasifikacije v dva razreda. Mnogokrat pa imajo podatki več možnih razredov, zato evalvacija teh modelov z metrikami, neprilagojenimi za več razredov, postane težavna. Kreacija matrike zmede poteka po enakem postopku kot pri binarni klasifikaciji, kjer v vrstice beležimo dejanske, v stolpce pa napovedane razrede instanc, kot kaže tabela 5.2.

Tabela 5.2: Matrika zmede za več razredov: A, B in C.

		Napovedano		
		A	B	C
Dejansko	A	50	1	2
	B	4	20	0
	C	5	1	10

Najpreprostejša metrika za izračun ne glede na število razredov je točnost, saj še vedno velja, da število pravilno klasificiranih instanc delimo s številom vseh instanc. Podobno velja tudi za delež napake, le da število pravilno klasificiranih zamenjamo s številom nepravilno klasificiranih primerkov.

Težava nastopi pri poimenovanju pozitivnih in negativnih razredov, saj smo prisiljeni oznako pozitivni ali negativni dodeliti več razredom. Vse metrike, ki vsebujejo število pozitivnih ali negativnih instanc, se morajo prilagoditi uporabi klasifikacije v več razredov na sledeč način. Te metrike računamo v več korakih, kjer v vsakem koraku dobi naziv pozitivnega razreda drug razred, ostali pa predstavljajo negativni razred. Končno vrednost metrik dobimo na več načinov: z *makro* agregacijo, z *uteženim povprečenjem*, ali z *mikro* agregacijo [21].

Makro agregacija metrik

Pri *makro agregaciji metrik* klasifikacije izdelamo več matrik zmede (m matrik zmede, če imamo m razredov), in sicer tako, da je v vsaki metriki zmede drug razred označen kot pozitivni razred. Za vsako matriko zmede izračunamo izbrano metriko (dobimo m metrik klasifikacije). Končno vrednost metrike klasifikacije pa na koncu izračunamo tako, da povprečimo vseh m metrik. Sledeče enačbe prikazujejo makro (Ma) izračun nekaterih metrik klasifikacije.

$$priklic_{Ma} = \frac{\sum_{i=1}^{m} \frac{TP_i}{TP_i + FN_i}}{m} = \frac{\sum_{i=1}^{m} priklic_i}{m}$$

$$preciznost_{Ma} = \frac{\sum_{i=1}^{m} \frac{TP_i}{TP_i + FP_i}}{m} = \frac{\sum_{i=1}^{m} preciznost_i}{m}$$

$$F_1\text{-}mera_{Ma} = \frac{\sum_{i=1}^{m} F_1\text{-}mera_i}{m}$$

Čeprav se metrika točnost izračuna za m razredov enako kot za binarni problem klasifikacije, pa lahko izračunamo metriko *povprečna točnost* (angl. *average class accuracy*). Ta metrika izračuna najprej točnost za vsak posamezni razred, nato pa povpreči izračunane vrednosti. Po enakem postopku izračunamo tudi *povprečno stopnjo napake* (angl. *average class error rate*), le da povprečimo deleže napak razredov.

$$povprečna\ točnost = \frac{\sum_{i=1}^{m} \frac{TP_i + TN_i}{TP_i + FP_i + TN_i + FN_i}}{m}$$

$$povprečen\ delež\ napake = \frac{\sum_{i=1}^{m} \frac{FP_i + FN_i}{TP_i + FP_i + TN_i + FN_i}}{m}$$

Utežena agregacija metrik

Preprosto povprečenje metrik vsakega razreda največkrat ni smiselno. Ko imamo neuravnoteženo množico (tj. množico, ko je razmerje med razredi zelo neenakomerno – čez 8:1), povprečenje kakovosti klasifikacije instanc pogostega razreda in kakovosti instanc redkih razredov ni smiselno. Namreč, več instanc imamo, več možnosti ima klasifikacijski algoritem, da izlušči vzorce, ki so značilni za instance tega razreda. To privede do klasifikacijskih modelov, ki delujejo mnogo boljše pri klasifikaciji instanc bolj pogostega razreda v primerjavi s klasifikacijo instanc redkih razredov. Če kakovost klasifikacije povprečimo (kot je to pri makro agregaciji), kakovost klasifikacije vseh razredov enakomerno prispeva h končni metriki. To je odlično, če želimo večji poudarek na kakovosti manjšinskih instanc. Če pa ne želimo, da instance v manjšini prevzamejo neenakomerno večjo vlogo pri ocenjevanju kakovosti, pa lahko posamezne metrike pred povprečenjem utežimo.

Pri uteževanju metriko posameznega razreda i zmnožimo z deležem instanc u_i razreda i, zmnožene metrike pa nato seštejemo v skupno mero. Delež razreda u_i je preprost odstotek instanc tega razreda i v testni množici.

$$priklic_{Ua} = \sum_{i=1}^{m} u_i \cdot \frac{TP_i}{TP_i + FN_i} = \sum_{i=1}^{m} u_i \cdot priklic_i$$

$$preciznost_{Ua} = \sum_{i=1}^{m} u_i \cdot \frac{TP_i}{TP_i + FP_i} = \sum_{i=1}^{m} u_i \cdot preciznost_i$$

$$F_1\text{-}mera_{Ua} = \sum_{i=1}^{m} u_i \cdot F_1\text{-}mera_i$$

Mikro agregacija metrik

Pri *mikro agregaciji metrik* začnemo podobno kot pri makro agregaciji; zgradimo m matrik zmede, nadaljnji postopek pa je drugačen. Namesto povprečenja različnih metrik seštejemo posamezne vrednosti iz matrik zmede. Primer: vse TP_i seštejemo skupaj, da dobimo TP_{Mi}, in tega vstavimo v enačbo za izračun mikro agregiranih metrik. Enačbe spodaj prikazujejo način izračuna posameznih metrik na mikro (Mi) način.

$$priklic_{Mi} = \frac{\sum_{i=1}^{m} TP_i}{\sum_{i=1}^{m}(TP_i + FN_i)}$$

$$preciznost_{Mi} = \frac{\sum_{i=1}^{m} TP_i}{\sum_{i=1}^{m}(TP_i + FP_i)}$$

$$F_1\text{-}mera_{Mi} = 2 \cdot \frac{preciznost_{Mi} \cdot priklic_{Mi}}{preciznost_{Mi} + priklic_{Mi}}$$

Posebnih mikro agregacij točnosti in deleža napake ni, saj v tem primeru dobimo že prej definirano točnost in delež napake.

Agregacija metrik v Pythonu

Z uporabo knjižnice `scikit-learn` in že do sedaj predstavljenih metod `accuracy_score`, `precision_score`, `recall_score` in `f1_score` enostavno agregiramo metrike vseh razredov v eno vrednost. Pri agregaciji nam služi parameter teh metod `average`, s katerim podamo način agregacije.

```
from sklearn.metrics import precision_score,
                              recall_score, f1_score

# Izračunamo preciznost z makro agregacijo
preciznost_makro = precision_score(y_dejanski,
                              y_napovedan,
                              average='macro')

# Izračunamo utežen priklic
priklic_utezen = recall_score(y_dejanski, y_napovedan,
                              average='weighted')

# Izračunamo mikro F-mero
f_mera_mikro = f1_score(y_dejanski, y_napovedan,
                              average='micro')

# Izračunamo ločeno preciznost za vsak razred
preciznost_vseh = precision_score(y_dejanski,
                              y_napovedan, average=None)

print(f'Makro preciznost: {preciznost_makro}')
print(f'Utežen priklic: {priklic_utezen}')
print(f'Mikro F-mera: {f_mera_mikro}')
print(f'Preciznost vseh razredov: {preciznost_vseh}')
```

```
Makro preciznost: 0.875
Utežen priklic: 0.8333333333333334
Mikro F-mera: 0.8333333333333334
Preciznost vseh razredov: [0.75 1.  ]
```

Vrednost parametra `average` določa način agregacije:

- `'macro'` se uporabi za makro agregacijo.
- `'weighted'` se uporabi za uteženo agregacijo, kjer so uteži sestavljene iz razmerja razredov v podani testni množici.
- `'micro'` se uporabi za mikro agregacijo.
- `None` se uporabi, ko želimo ločen izpis metrike za vsak razred posebej, brez agregacije.

5.3 Utrjevanje znanja

Preizkusi se na nekaj nalogah izračuna kakovosti klasifikatorjev, ki jih rešiš tako na roko kakor s programiranjem v Pythonu.

Naloga 5-1

Podani sta matriki zmede dveh različnih klasifikatorjev za klasifikacijo v dva razreda.

Klasifikator 1

Dejansko \ Napovedano	A	B
A	62	8
B	11	59

Klasifikator 2

Dejansko \ Napovedano	A	B
A	50	20
B	30	40

a) Za vsak klasifikator izračunaj točnost klasifikacije in delež napake.

b) Obravnavaj razred A kot pozitivni razred. Za vsak klasifikator izračunaj priklic, preciznost in F-mero.

Naloga 5-2

Podani sta matriki zmede dveh različnih klasifikatorjev za klasifikacijo v več razredov.

Klasifikator 1

Dejansko \ Napovedano	A	B	C
A	500	10	20
B	40	20	0
C	50	10	10

Klasifikator 2

Dejansko \ Napovedano	A	B	C
A	430	60	40
B	10	50	0
C	0	10	60

a) Za vsak klasifikator izračunaj točnost klasifikacije in delež napake.

b) Za vsak klasifikator in za vsak razred posebej izračunaj priklic, preciznost in F-mero.

c) Izračunaj skupen priklic, preciznost in F-mero po načinu makro agregacije metrik.

d) Izračunaj skupen priklic, preciznost in F-mero po načinu mikro agregacije metrik.

e) Izračunaj skupen priklic, preciznost in F-mero po načinu uteženega povprečenja metrik.

Naloga 5-3

a) S pomočjo knjižnice `scikit-learn` preberi podatke priložene množice *Iris*. Množico razdeli na učno, validacijsko in testno množico v razmerju 60:20:20.

b) Na učni množici nauči tri klasifikatorje k najbližjih sosedov z Evklidsko razdaljo. Prvi naj ima $k = 1$, drugi $k = 5$ in tretji $k = 10$.

c) Vsak klasifikator uporabi za klasifikacijo validacijske množice. Za dobljene rezultate na validacijski množici za vsak klasifikator izračunaj točnost. Tistega z najboljšo točnostjo obravnavaj kot najboljšega in z njim nadaljuj.

d) Za najboljši klasifikator na testni množici izračunaj sledeče:

1. preciznost, priklic in F-mero za vsak razred,
2. makro agregirane preciznost, priklic in F-mero,
3. mikro agregirane preciznost, priklic in F-mero ter
4. uteženo agregirane preciznost, priklic in F-mero.

Naloga 5-4

a) S pomočjo knjižnice `scikit-learn` preberi podatke priložene množice *Iris*. Množico s stratificirano navzkrižno validacijo razdeli na pet rezov.

b) Za vsako delitev na reze naredi sledeče:

1. nauči k najbližjih sosedov ($k = 3$, Evklidska razdalja) na učnih podatkih;
2. izpiši točnost klasifikacije na testnih podatkih in
3. izpiši uteženo agregirano F-mero na testnih podatkih.

c) Povpreči metriki točnosti in F-mere na vseh rezih. Izpiši dobljeno povprečno točnost in povprečno F-mero vseh rezov.

6 | Rešitve nalog

6.1 Poglavje Prvi klasifikator

Naloga 4-1

Za lokalnega preprodajalca avtomobilov ustvarjamo model znanja, ki mu bo poenostavil ugotoviti, kateri rabljeni avtomobili so primerni za nakup in kateri niso. Podana je tabela preteklih nakupov tega preprodajalca avtomobilov.

a) Tabelo dopolni z izračuni evklidskih in manhattanskih razdalj izbranega avtomobila do preteklih nakupov. Zapiši range razdalj – od najbližje instance (rang 1) do najbolj oddaljene instance.

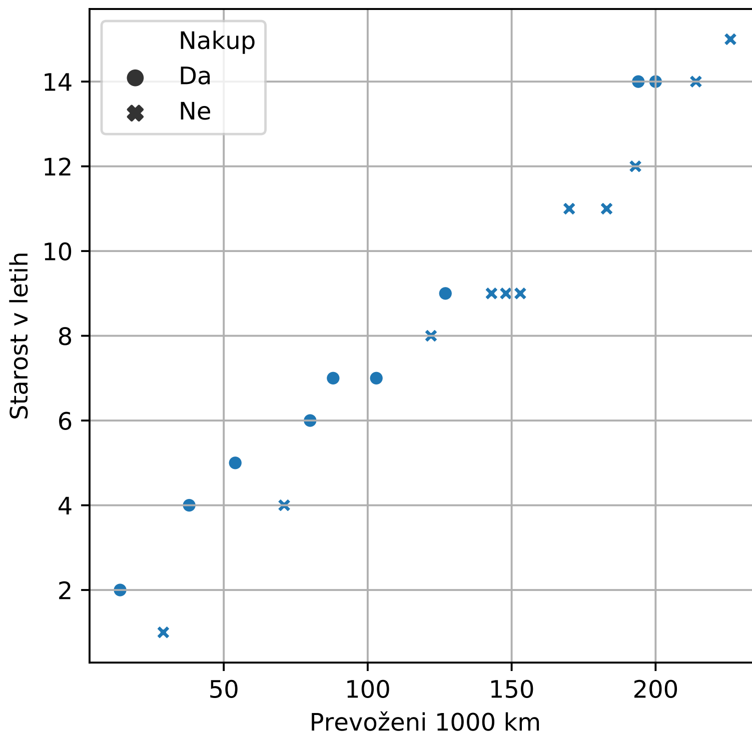
Pretekli nakupi rabljenih avtomobilov

Indeks	*Prevoženi (1000 km)*	*Starost (leta)*	*Nakup*	*Evklidska d*	*Evklidska Rang*	*Manhattanska d*	*Manhattanska Rang*
0	14	2	Da	86,1	15	89,0	14
1	29	1	Ne	71,1	13	75,0	12
2	38	4	Da	62,0	11	63,0	11
3	54	5	Da	46,0	8	46,0	7
4	71	4	Ne	29,0	6	30,0	5
5	80	6	Da	20,0	3	21,0	3
6	88	7	Da	12,2	2	14,0	2
7	103	7	Da	3,6	1	5,0	1
8	122	8	Ne	22,2	4	25,0	4
9	127	9	Da	27,3	5	31,0	6
10	143	9	Ne	43,2	7	47,0	8
11	148	9	Ne	48,2	9	52,0	9
12	153	9	Ne	53,2	10	57,0	10
13	170	11	Ne	70,3	12	76,0	13
14	183	11	Ne	83,2	14	89,0	14
15	193	12	Ne	93,3	16	100,0	16
16	194	14	Da	94,4	17	103,0	17
17	200	14	Da	100,4	18	109,0	18
18	214	14	Ne	114,4	19	123,0	19
29	226	15	Ne	126,4	20	136,0	20

Izbran avtomobil

Prevoženi (1000 km)	*Starost (leta)*
100	5

b) Na roko izriši graf raztrosa za celotno množico. Označbe instanc prilagodi razredu posamezne instance.



c) V spodnjo tabelo dopiši indekse najbližjih petih instanc za posamezno mero razdalje.

Rangi	*1*	*2*	*3*	*4*	*5*
Evklidska	7	6	5	8	9
Manhattanska	7	6	5	8	4

d) V tabelo dopiši rezultate klasifikacije za posamezno nastavitev k in posamezno mero razdalje.

	Evklidska			*Manhattanska*		
k	*Da*	*Ne*	*Rezultat*	*Da*	*Ne*	*Rezultat*
1	1	0	Da	1	0	Da
2	2	0	Da	2	0	Da
3	3	0	Da	3	0	Da
4	3	1	Da	3	1	Da
5	4	1	Da	3	2	Da

Naloga 4-2

Uporabi podatke prejšnje naloge o nakupih rabljenih avtomobilov.

a) Izračunaj povprečje in standardni odklon za vsak atribut podatkov.

$$\mu\,(\text{Prevoženi km}) = 126{,}2 \qquad \rho\,(\text{Prevoženi km}) = 63{,}6$$

$$\mu\,(\text{Starost}) = 8{,}4 \qquad \rho\,(\text{Starost}) = 4{,}0$$

b) V sledečo tabelo vnesi standardizirane podatke.

Standardizirani podatki rabljenih avtomobilov

Indeks	*Prevoženi (1000 km)*	*Starost (leta)*	*Nakup*	*Evklidska*		*Manhattanska*	
				d	*Rang*	*d*	*Rang*
0	-1,8	-1,6	Da	1,5	13	2,1	12
1	-1,5	-1,8	Ne	1,5	12	2,1	13
2	-1,4	-1,1	Da	1,0	7	1,2	7
3	-1,1	-0,8	Da	0,7	5	0,7	5
4	-0,9	-1,1	Ne	0,5	3	0,7	4
5	-0,7	-0,6	Da	0,4	1	0,6	2
6	-0,6	-0,3	Da	0,5	4	0,7	3
7	-0,4	-0,3	Da	0,5	2	0,5	1
8	-0,1	-0,1	Ne	0,8	6	1,1	6
9	0,0	0,2	Da	1,1	8	1,4	8
10	0,3	0,2	Ne	1,2	9	1,7	9
11	0,3	0,2	Ne	1,3	10	1,8	10
12	0,4	0,2	Ne	1,3	11	1,8	11
13	0,7	0,7	Ne	1,9	14	2,6	14
14	0,9	0,7	Ne	2,0	15	2,8	15
15	1,0	0,9	Ne	2,3	16	3,2	16
16	1,1	1,4	Da	2,7	17	3,7	17
17	1,2	1,4	Da	2,7	18	3,8	18
18	1,4	1,4	Ne	2,9	19	4,0	19
29	1,6	1,7	Ne	3,2	20	4,5	20

Izbran avtomobil

Prevoženi (1000 km)	*Starost (leta)*
-0,4	-0,8

c) Tabelo dopolni z evklidskimi in manhattanskimi razdaljami na standardiziranih podatkih.

d) V spodnjo tabelo dopiši rezultate klasifikacije za posamezno nastavitev k in posamezno mero razdalje.

k	*Evklidska* Da	*Evklidska* Ne	*Evklidska* Rezultat	*Manhattanska* Da	*Manhattanska* Ne	*Manhattanska* Rezultat
1	1	0	Da	1	0	Da
2	2	0	Da	2	0	Da
3	2	1	Da	3	0	Da
4	3	1	Da	3	1	Da
5	4	1	Da	4	1	Da

Naloga 4-3

V Pythonu napiši sledečo kodo.

a) S pomočjo knjižnice `scikit-learn` preberi podatke priložene množice *Wine*. Za namene te naloge uporabiš le dva atributa množice: 'magnesium' in 'flavanoids'. Obdrži le ta dva atributa, ostale zavrzi. Pazi, da ne izgubiš podatka o razredu.

```
from pandas import Index
from sklearn.datasets import load_wine

podatki = load_wine(as_frame=True)

mag = Index(podatki.feature_names).get_loc('magnesium')
fla = Index(podatki.feature_names).get_loc('flavanoids')

podatki_ostali = podatki.data.iloc[:,[mag, fla]]
```

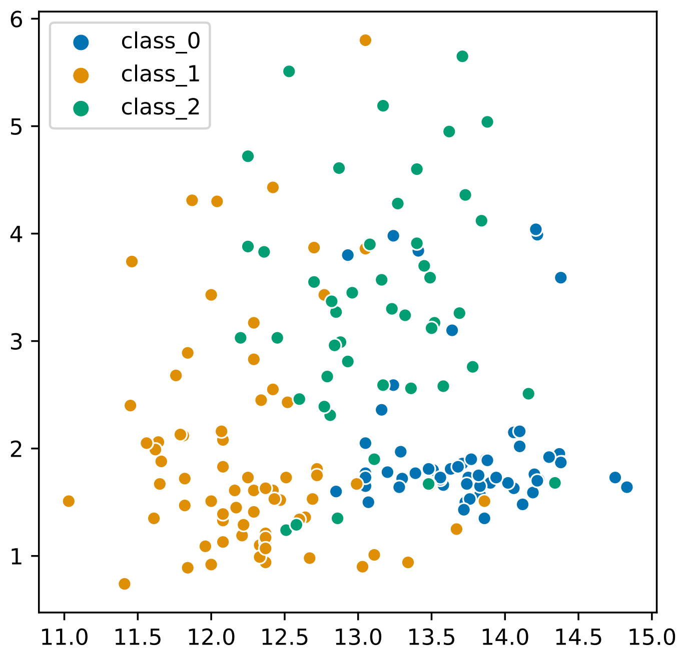
b) Zadnjo instanco te množice shrani kot testno instanco, ves ostali del množice pa shrani kot učne podatke. Izpiši prvih pet vrstic učnih podatkov in testno instanco.

```
izbrana = -1
X_izbrana = podatki.data.iloc[izbrana,:]
y_izbrana = podatki.target.iloc[izbrana]

X_ostali = podatki.data.drop(podatki.data.tail(1).index,
                             axis=0)
y_ostali = podatki.target.drop(podatki.target.tail(1).
                            index)

print(X_ostali.iloc[0:5,:])
print(X_izbrana)
```

```
 alcohol malic_acid   ash alcalinity_of_ash magnesium total_phenols
...
0   14.23      1.71   2.43             15.6      127.0      2.80
1   13.20      1.78   2.14             11.2      100.0      2.65
2   13.16      2.36   2.67             18.6      101.0      2.80
3   14.37      1.95   2.50             16.8      113.0      3.85
4   13.24      2.59   2.87             21.0      118.0      2.80
alcohol 14.13
malic_acid 4.10
ash 2.74
alcalinity_of_ash 24.50
magnesium 96.00
total_phenols 2.05
flavanoids 0.76
nonflavanoid_phenols 0.56
...
```

c) Izriši graf raztrosa učnih podatkov, kjer je 'magnesium' na x osi ter 'flavanoids' na y osi. Označbe instanc naj so obarvane glede na razred posamezne instance. Pri tem uporabi bodisi knjižnico `seaborn` ali `Matplotlib`.

```
import seaborn as sns

sns.scatterplot(x=X_ostali.iloc[:,0],
                y=X_ostali.iloc[:,1],
                hue=podatki.target_names[y_ostali],
                palette='colorblind')
```



d) Učne podatke uporabi za učenje sledečih klasifikatorjev k najbližjih sosedov:

- $k = 1$ in evklidska razdalja,
- $k = 1$ in manhattanska razdalja,
- $k = 3$ in evklidska razdalja,
- $k = 3$ in manhattanska razdalja,
- $k = 5$ in evklidska razdalja in
- $k = 5$ in manhattanska razdalja.

Napovedi za testno instanco vsakega klasifikatorja izpiši.

```
from sklearn.neighbors import KNeighborsClassifier

for k in (1, 3, 5):
    for razdalja in ('euclidean', 'manhattan'):
        knn = KNeighborsClassifier(n_neighbors=5,
                                   metric=razdalja)
        knn.fit(X_ostali, y_ostali)
        napoved = knn.predict([X_izbrana])

        print(f'KNN metric={razdalja}, '
              f'k={k} napove {podatki.target_names[
                                          napoved]}')
```

```
KNN metric=euclidean, k=1 napove 'class_1'
KNN metric=manhattan, k=1 napove 'class_2'
KNN metric=euclidean, k=3 napove 'class_1'
KNN metric=manhattan, k=3 napove 'class_2'
KNN metric=euclidean, k=5 napove 'class_1'
KNN metric=manhattan, k=5 napove 'class_2'
```

Naloga 4-4

V Pythonu napiši sledečo kodo.

a) S pomočjo knjižnice `scikit-learn` preberi podatke priložene množice *Breast cancer*. Zadnjih deset instanc te množice shrani ločeno kot testno množico, ves ostali del množice pa shrani kot učne podatke. Izpiši prvih pet vrstic učnih podatkov in prvih pet instanc testne množice.

```
from sklearn.datasets import load_breast_cancer

podatki = load_breast_cancer(as_frame=True)

izbrani = list(range(podatki.data.shape[0]-10,
                     podatki.data.shape[0]))

X_izbrani = podatki.data.iloc[izbrani,:]
y_izbrani = podatki.target.iloc[izbrani]

X_ostali = podatki.data.drop(izbrani, axis=0)
y_ostali = podatki.target.drop(izbrani)

print(X_izbrani.head(5))
print(X_ostali.head(5))
```

```
   mean radius   mean texture   mean perimeter   mean area ...
559     11.51        23.93            74.52        403.5
560     14.05        27.15            91.38        600.4
561     11.20        29.37            70.67        386.0
562     15.22        30.62            103.40       716.9
563     20.92        25.09            143.00       1347.0
mean radius   mean texture   mean perimeter   mean area ...
0     17.99        10.38            122.80       1001.0
1     20.57        17.77            132.90       1326.0
2     19.69        21.25            130.00       1203.0
3     11.42        20.38            77.58        386.1
4     20.29        14.34            135.10       1297.0
```

b) Standardiziraj podatke na podlagi učne množice. Standardiziraj tako učne kot testne podatke. Pri tem pazi, da ne izgubiš nestandardiziranih učnih in testnih podatkov. Izpiši prvih pet vrstic standardiziranih učnih podatkov in prvih pet instanc standardizirane testne množice.

```
from sklearn.preprocessing import StandardScaler

standardizator = StandardScaler()
standardizator.fit(X_ostali)
X_ostali_s = standardizator.transform(X_ostali)
X_izbrani_s = standardizator.transform(X_izbrani)

print(X_izbrani_s[0:5,:])
print(X_ostali_s[0:5,:])
```

```
[[-7.40458594e-01 ...  -3.96577043e-01]]
```

c) Nauči dva k najbližjih sosedov klasifikatorja ($k = 3$, evklidska razdalja), enega na nestandardiziranih podatkih, drugega na standardiziranih podatkih. Za vsako instanco v testni množici izpiši v eni vrstici tri vrednosti:

1. dejanski razred iz množice,
2. napoved klasifikatorja naučenega na nestandardiziranih podatkih in
3. napoved klasifikatorja naučenega na nestandardiziranih podatkih.

```python
from sklearn.neighbors import KNeighborsClassifier

knn1 = KNeighborsClassifier(n_neighbors=3,
                            metric='euclidean')
knn1.fit(X_ostali, y_ostali)
napoved1 = knn1.predict(X_izbrani)

knn2 = KNeighborsClassifier(n_neighbors=3,
                            metric='euclidean')
knn2.fit(X_ostali_s, y_ostali)
napoved2 = knn2.predict(X_izbrani_s)

for i, d, n1, n2 in zip(izbrani, y_ostali,
                        napoved1, napoved2):

    print(f'{i} je: {podatki.target_names[d]}, '
          f'knn(izvorni): {podatki.target_names[n1]}, '
          f'knn(stand): {podatki.target_names[n2]}')
```

```
559 je: malignant, knn(izvorni): benign, knn(stand): benign
560 je: malignant, knn(izvorni): benign, knn(stand): malignant
561 je: malignant, knn(izvorni): benign, knn(stand): benign
562 je: malignant, knn(izvorni): malignant, knn(stand): malignant
563 je: malignant, knn(izvorni): malignant, knn(stand): malignant
564 je: malignant, knn(izvorni): malignant, knn(stand): malignant
565 je: malignant, knn(izvorni): malignant, knn(stand): malignant
566 je: malignant, knn(izvorni): malignant, knn(stand): malignant
567 je: malignant, knn(izvorni): malignant, knn(stand): malignant
568 je: malignant, knn(izvorni): benign, knn(stand): benign
```

6.2 Poglavje Kakovost klasifikacije

Naloga 5-1

Podani sta matriki zmede dveh različnih klasifikatorjev za klasifikacijo v dva razreda.

Klasifikator 1

		Napovedano	
		A	B
Dejansko	A	62	8
	B	11	59

Klasifikator 2

		Napovedano	
		A	B
Dejansko	A	50	20
	B	30	40

a) Za vsak klasifikator izračunaj točnost klasifikacije in delež napake.

$$točnost_{Klasifikator1} = \frac{62 + 59}{62 + 8 + 11 + 59} = 0{,}8643$$
$$točnost_{Klasifikator2} = \frac{50 + 40}{50 + 20 + 30 + 40} = 0{,}6429$$
$$delež\ napake_{Klasifikator1} = 1{,}0 - 0{,}8643 = 0{,}1357$$
$$delež\ napake_{Klasifikator2} = 1{,}0 - 0{,}6429 = 0{,}3571$$

b) Obravnavaj razred A kot pozitivni razred. Za vsak klasifikator izračunaj priklic, preciznost in F-mero.

$$
\begin{aligned}
priklic_{Klasifikator1} &= \frac{62}{62+8} = 0{,}8857 \\
priklic_{Klasifikator2} &= \frac{50}{50+20} = 0{,}7143 \\
preciznost_{Klasifikator1} &= \frac{62}{62+11} = 0{,}8493 \\
preciznost_{Klasifikator2} &= \frac{50}{50+30} = 0{,}6250 \\
F-mera_{Klasifikator1} &= 2 \cdot \frac{0{,}8493 * 0{,}8857}{0{,}8493 + 0{,}8857} = 0{,}8671 \\
F-mera_{Klasifikator2} &= 2 \cdot \frac{0{,}6250 * 0{,}7143}{0{,}6250 + 0{,}7143} = 0{,}6667
\end{aligned}
$$

Naloga 5-2

Podani sta matriki zmede dveh različnih klasifikatorjev za klasifikacijo v več razredov.

Klasifikator 1

		Napovedano		
		A	B	C
Dejansko	A	500	10	20
	B	40	20	0
	C	50	10	10

Klasifikator 2

	Napovedano		
	A	B	C
A	430	60	40
B	10	50	0
C	0	10	60

a) Za vsak klasifikator izračunaj točnost klasifikacije in delež napake.

$$točnost_{K1} = \frac{500 + 20 + 10}{500 + 10 + 20 + 40 + 20 + 0 + 50 + 10 + 10} = 0{,}8030$$

$$točnost_{K2} = \frac{430 + 50 + 60}{430 + 60 + 40 + 10 + 50 + 0 + 0 + 10 + 60} = 0{,}8181$$

$$delež\ napake_{K1} = 1{,}0 - 0{,}8030 = 0{,}1970$$

$$delež\ napake_{K2} = 1{,}0 - 0{,}8181 = 0{,}1819$$

b) Za vsak klasifikator in za vsak razred posebej izračunaj priklic, preciznost in F-mero.

Klasifikator 1:

$$
\begin{aligned}
priklic_A &= \frac{500}{500+10+20} = 0{,}9434 \\
priklic_B &= \frac{20}{40+20+0} = 0{,}3333 \\
priklic_C &= \frac{10}{50+10+10} = 0{,}1429 \\
preciznost_A &= \frac{500}{500+40+50} = 0{,}8475 \\
preciznost_B &= \frac{20}{10+20+10} = 0{,}5000 \\
preciznost_C &= \frac{10}{20+0+10} = 0{,}3333 \\
F-mera_A &= 2\cdot\frac{0{,}9434 * 0{,}8475}{0{,}9434 + 0{,}8475} = 0{,}8929 \\
F-mera_B &= 2\cdot\frac{0{,}3333 * 0{,}5000}{0{,}3333 + 0{,}5000} = 0{,}4000 \\
F-mera_C &= 2\cdot\frac{0{,}1429 * 0{,}3333}{0{,}1429 + 0{,}3333} = 0{,}2000
\end{aligned}
$$

Klasifikator 2:

$$priklic_A = \frac{430}{430 + 60 + 40} = 0{,}8113$$
$$priklic_B = \frac{50}{10 + 50 + 0} = 0{,}8333$$
$$priklic_C = \frac{60}{0 + 10 + 60} = 0{,}8571$$
$$preciznost_A = \frac{430}{430 + 10 + 0} = 0{,}9773$$
$$preciznost_B = \frac{50}{60 + 50 + 10} = 0{,}4167$$
$$preciznost_C = \frac{60}{40 + 0 + 60} = 0{,}6000$$
$$F - mera_A = 2 \cdot \frac{0{,}8113 * 0{,}9773}{0{,}8113 + 0{,}9773} = 0{,}8866$$
$$F - mera_B = 2 \cdot \frac{0{,}8333 * 0{,}4167}{0{,}8333 + 0{,}4167} = 0{,}5556$$
$$F - mera_C = 2 \cdot \frac{0{,}8571 * 0{,}6000}{0{,}8571 + 0{,}6000} = 0{,}7059$$

c) Izračunaj skupen priklic, preciznost in F-mero s po načinu makro agregacije metrik.

$$\begin{aligned} priklic_{Ma} &= \frac{0{,}9434 + 0{,}3333 + 0{,}1429}{3} = 0{,}4732 \\ preciznost_{Ma} &= \frac{0{,}8475 + 0{,}5000 + 0{,}3333}{3} = 0{,}5603 \\ F - mera_{Ma} &= \frac{0{,}8929 + 0{,}4000 + 0{,}2000}{3} = 0{,}4976 \end{aligned}$$

d) Izračunaj skupen priklic, preciznost in F-mero s po načinu mikro agregacije metrik.

$$\begin{aligned} priklic_{Mi} &= \frac{500 + 20 + 10}{500 + 10 + 20 + 40 + 20 + 0 + 50 + 10 + 10} = 0{,}8030 \\ preciznost_{Mi} &= \frac{500 + 20 + 10}{500 + 40 + 50 + 10 + 20 + 10 + 20 + 0 + 10} = 0{,}8030 \\ F - mera_{Mi} &= 2 \cdot \frac{0{,}8030 * 0{,}8030}{0{,}8030 + 0{,}8030} = 0{,}8030 \end{aligned}$$

e) Izračunaj skupen priklic, preciznost in F-mero s po načinu uteženega povprečenja metrik.

$$u_A = \frac{500 + 10 + 20}{500 + 10 + 20 + 40 + 20 + 50 + 10 + 10} = 0{,}8030$$

$$u_B = \frac{40 + 20 + 0}{500 + 10 + 20 + 40 + 20 + 50 + 10 + 10} = 0{,}0909$$

$$u_C = \frac{50 + 10 + 10}{500 + 10 + 20 + 40 + 20 + 50 + 10 + 10} = 0{,}1061$$

$$\begin{aligned} priklic_{Ua} &= 0{,}8030 * 0{,}9434 + 0{,}0909 * 0{,}3333 + 0{,}1061 * 0{,}1429 \\ &= 0{,}8031 \\ preciznost_{Ua} &= 0{,}8030 * 0{,}8475 + 0{,}0909 * 0{,}5000 + 0{,}1061 * 0{,}3333 \\ &= 0{,}7614 \\ F - mera_{Ua} &= 0{,}8030 * 0{,}8929 + 0{,}0909 * 0{,}4000 + 0{,}1061 * 0{,}2000 \\ &= 0{,}7746 \end{aligned}$$

Naloga 5-3

a) S pomočjo knjižnice `scikit-learn` preberi podatke priložene množice *Iris*. Množico razdeli na učno, validacijsko in testno množico v razmerju 60:20:20.

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

# Naložimo podatke
podatki = load_iris(as_frame=True)

# Razdelimo podatke na učne/validacijske in testne
X_ucna_validacijska, X_testna, y_ucna_validacijska, \
y_testna = train_test_split(podatki.data,
                            podatki.target,
                            train_size=0.8)

# Razdelimo podatke na učne in validacijske
X_ucna, X_validacijska, y_ucna, \
y_validacijska = train_test_split(X_ucna_validacijska,
                                  y_ucna_validacijska,
                                  train_size=0.75)

print(X_ucna.shape[0]/podatki.data.shape[0],
      X_validacijska.shape[0]/podatki.data.shape[0],
      X_testna.shape[0]/podatki.data.shape[0])
```

```
0.6 0.2 0.2
```

b) Na učni množici nauči tri klasifikatorje k najbližjih sosedov z Evklidsko razdaljo. Prvi naj ima $k = 1$, drugi $k = 5$ in tretji $k = 10$.

```
from sklearn.neighbors import KNeighborsClassifier

knn1 = KNeighborsClassifier(n_neighbors=1, metric='
                            euclidean')
knn5 = KNeighborsClassifier(n_neighbors=5, metric='
                            euclidean')
knn10 = KNeighborsClassifier(n_neighbors=10, metric='
                            euclidean')

knn1.fit(X_ucna, y_ucna)
knn5.fit(X_ucna, y_ucna)
knn10.fit(X_ucna, y_ucna)
```

c) Vsak klasifikator uporabi za klasifikacijo validacijske množice. Za dobljene rezultate na validacijski množici za vsak klasifikator izračunaj točnost. Tistega z najboljšo točnostjo obravnavaj kot najboljšega in z njim nadaljuj.

```
from sklearn.metrics import accuracy_score

knn1_napovedi = knn1.predict(X_validacijska)
knn5_napovedi = knn5.predict(X_validacijska)
knn10_napovedi = knn10.predict(X_validacijska)

knn1_tocnost = accuracy_score(y_validacijska,
                              knn1_napovedi)
knn5_tocnost = accuracy_score(y_validacijska,
                              knn5_napovedi)
knn10_tocnost = accuracy_score(y_validacijska,
                               knn10_napovedi)

print(f'Knn(k=1) točnost je {knn1_tocnost}.')
print(f'Knn(k=5) točnost je {knn5_tocnost}.')
print(f'Knn(k=10) točnost je {knn10_tocnost}.')
```

```
Knn(k=1) točnost je 0.9.
Knn(k=5) točnost je 0.9333333333333333.
Knn(k=10) točnost je 1.0.
```

d) Za najboljši klasifikator na testni množici izračunaj sledeče:

1. preciznost, priklic in F-mero za vsak razred,

```
# Za najboljšega vzamemo knn10

from sklearn.metrics import precision_score,
                            recall_score
from sklearn.metrics import f1_score

print(f'Preciznost: {precision_score(y_testna,
                                     knn10_napovedi,
                                     average=None)}')
print(f'Priklic: {recall_score(y_testna,
                               knn10_napovedi,
                               average=None)}')
print(f'F-mera: {f1_score(y_testna,
                          knn10_napovedi,
                          average=None)}')
```

```
Preciznost:  [0.125 0.46666667 0.57142857]
Priklic:  [0.125 0.7 0.33333333]
F-mera:  [0.125 0.56 0.42105263]
```

2. makro agregirane preciznost, priklic in F-mero,

```
print(f'Preciznost: {precision_score(y_testna,
                             knn10_napovedi,
                             average="macro")}')
print(f'Priklic: {recall_score(y_testna, knn10_napovedi,
                              average="macro")}')
print(f'F-mera: {f1_score(y_testna, knn10_napovedi,
                             average="macro")}')
```

```
Preciznost:  0.38769841269841265
Priklic:  0.38611111111111107
F-mera:  0.3686842105263158
```

3. mikro agregirane preciznost, priklic in F-mero ter

```
print(f'Preciznost: {precision_score(y_testna,
                             knn10_napovedi,
                             average="weighted")}')
print(f'Priklic: {recall_score(y_testna, knn10_napovedi,
                              average="weighted")}')
print(f'F-mera: {f1_score(y_testna, knn10_napovedi,
                             average="weighted")}')
```

```
Preciznost:  0.4174603174603175
Priklic:  0.4
F-mera:  0.388421052631579
```

4. uteženo agregirane preciznost, priklic in F-mero.

```
print(f'Preciznost: {precision_score(y_testna,
                            knn10_napovedi,
                            average="micro")}')
print(f'Priklic: {recall_score(y_testna, knn10_napovedi,
                            average="micro")}')
print(f'F-mera: {f1_score(y_testna, knn10_napovedi,
                            average="micro")}')
```

```
Preciznost:  0.4
Priklic:  0.4
F-mera:  0.4000000000000001
```

Naloga 5-4

a) S pomočjo knjižnice `scikit-learn` preberi podatke priložene množice *Iris*. Množico s stratificirano navzkrižno validacijo razdeli na pet rezov.

```
from sklearn.datasets import load_iris
from sklearn.model_selection import StratifiedKFold

# Naložimo podatke
podatki = load_iris(as_frame=True)

# Naložimo stratificirani navzkrižno validacijo s 4 rezi
skf = StratifiedKFold(n_splits=5)
```

b) Za vsako delitev na reze naredi sledeče:

1. nauči k najbližjih sosedov ($k = 3$, Evklidska razdalja) na učnih podatkih;
2. izpiši točnost klasifikacije na testnih podatkih in
3. izpiši uteženo agregirano F-mero na testnih podatkih.

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

# Gremo skozi vse razdelitve rezov
for ucni, testni in skf.split(podatki.data,
                              podatki.target):
    X_ucna = podatki.data.iloc[ucni,:]
    X_testna = podatki.data.iloc[testni,:]

    y_ucna = podatki.target.iloc[ucni]
    y_testna = podatki.target.iloc[testni]

    knn = KNeighborsClassifier(n_neighbors=3,
                               metric='euclidean')
    knn.fit(X_ucna, y_ucna)
    knn_napovedi = knn.predict(X_testna)

    tocnost_reza = accuracy_score(y_testna,
                                  knn_napovedi)

    print(f'Točnost klasifikacije na tem '
          f'rezu je {tocnost_reza}.')
```

```
Točnost klasifikacije na tem rezu je 0.9666666666666667.
Točnost klasifikacije na tem rezu je 0.9666666666666667.
Točnost klasifikacije na tem rezu je 0.9333333333333333.
Točnost klasifikacije na tem rezu je 0.9666666666666667.
Točnost klasifikacije na tem rezu je 1.0.
```

c) Povpreči metriki točnosti in F-mere na vseh rezih. Izpiši dobljeno povprečno točnost in povprečno F-mero vseh rezov.

```
import numpy as np

# Sem bomo shranjevali vse točnosti
tocnosti = []

# Gremo skozi vse razdelitve rezov
for ucni, testni in skf.split(podatki.data,
                              podatki.target):
    X_ucna = podatki.data.iloc[ucni,:]
    X_testna = podatki.data.iloc[testni,:]

    y_ucna = podatki.target.iloc[ucni]
    y_testna = podatki.target.iloc[testni]

    knn = KNeighborsClassifier(n_neighbors=3,
                                metric='euclidean')

    knn.fit(X_ucna, y_ucna)
    knn_napovedi = knn.predict(X_testna)

    tocnost_reza = accuracy_score(y_testna,
                                  knn_napovedi)

    # Zdaj pa še shranimo posamezne točnosti v seznam
                                  vseh točnosti
    tocnosti.append(tocnost_reza)

print(f'Vse točnosti so {tocnosti}.')
print(f'Povprečna točnost je {np.mean(tocnosti)}.')
```

```
Vse točnosti so [0.9666666666666667, 0.9666666666666667,
0.9333333333333333, 0.9666666666666667, 1.0].
Povprečna točnost je 0.9666666666666668.
```

7 | Zaključek in kako naprej

S pomočjo te knjige si naredil/-a prve korake spoznavanja s strojnim učenjem. Pri tem smo pregledali eno tehniko nekoliko bolj poglobljeno – tehniko nadzorovanega učenja, klasifikacijo. Čeprav je klasifikacija ena izmed poglavitnih tehnik strojnega učenja, pa vsekakor ni edina. Za učinkovito in polno rabo strojnega učenja v raziskovalne ali poslovne namene je vsekakor potrebno spoznati tudi ostale (regresija, gručenje ...). Snov, ki smo jo predelali, pa služi kot osnova za lažje spoznavanje ostalih tehnik, bodisi samostojno ali ob pomoči druge literature.

Vsekakor obstajajo področja klasifikacije, ki so vredna dodatnega preučevanja – od tisoče drugih klasifikatorjev pa do posebnosti, kot so klasifikacija z neuravnoteženimi podatki in klasifikacija v več razredov. V knjigi smo se posvetili enemu klasifikacijskemu algoritmu, tj. k najbližjih sosedov. Kljub temu, da ta algoritem v praksi ni vedno primeren za uporabo, smo se osredotočili nanj, ker je razumevanje njegovega delovanja relativno enostavno. V praksi se večkrat uporabijo algoritmi takojšnjega učenja, saj je uporaba (ne pa tudi gradnja) mo-

dela znanja pri takih algoritmih računsko manj zahteva in posledično bolj praktična. Podrobno poznavanje vsakega klasifikacijskega algoritma pa ni ne praktično kakor tudi izvedljivo zaradi nešteto različnih pristopov. Izkušen uporabnik tehnik strojnega učenja ima v repertoarju nabor uporabnih in znanstveno dokazano učinkovitih algoritmov. Poseg v eksotične algoritme je smiseln le ob prisotnosti specifičnih zahtev podatkov in analiz.

Prav tako pa je nesmiselno prehitevati z uporabo algoritmov za kreacijo kompleksnih modelov znanja – na primer globokih nevronskih mrež. Obstajajo primeri, kjer je uporaba teh smiselna tudi že v samem začetku (klasifikacija slik), ampak je osnovno poznavanje procesa strojnega učenja še vedno potrebno. Ob uporabi nevronskih mrež je vsekakor večji poudarek na sami optimizaciji modela znanja kot pa na samem procesu ekstrakcije vzorcev. Posledica tega se izrazi predvsem v potrebi po teoretičnem poznavanju podrobnosti samega algoritma kreacije modela znanja v obliki nevronske mreže in optimizacije mnogoterih posameznih nastavitev. Nepremišljena zagnanost v uporabo kompleksnejših algoritmov največkrat ne prinese boljših rezultatov strojnega učenja, temveč le zafrustriranost nad samim procesom optimizacije takega modela in potrebo po neprimerno močnejši ter dražji strojni opremi.

Primeri v knjigi so prikazani v okolju Jupyter Notebook, ki je idealno za učenje, preizkušanje in raziskovanje s pomočjo tehnik strojnega učenja. Vsekakor pa uporaba tega okolja ni nujna, saj se lahko prav vsi primeri knjige uporabijo tudi v navadnih Python programih – datotekah s končnico *.py*. Prav implementacija in uporaba metod strojnega učenja v Python programih (in ne zvezkih) je pogosta pri implementaciji storitev, ki se vključujejo v druge procese poslovnega ali informacijskega sistema.

V knjigi so vsi praktični primeri narejeni s pomočjo `scikit-learn` knjižnice, ampak dobro se je zavedati, da obstajajo še druge knjižnice in ogrodja strojnega učenja za delo v Pythonu. Če izvajamo strojno učenje na neuravnoteženih podatkih, bi uporabili knjižnico

`imbalanced-learn`, za iskanje vzorcev v časovnih vrstah imamo knjižnico `sktime`, pri obdelavi besedil nam prav pridejo `huggingface`, `NLTK` ter `spaCy`, če pa se spuščamo v vode globokih nevronskih mrež, pa bi uporabili `PyTorch` ali `TensorFlow` in `Keras`. Je pa za vse navedene knjižnice značilno, da so kompatibilne ali celo v zaledju uporabljajo `scikit-learn`.

Prav tako se je dobro zavedati, da strojno učenje lahko uporabimo tudi v drugih programskih jezikih. Poleg Pythona je strojno učenje dobro podprto tudi v jezikih R, Java, C#, C, C++, JavaScript, Julia in mnogih drugih. V knjigi smo se osredotočili na delo v Pythonu, ker je to eden izmed najbolj pogosto uporabljenih jezikov v splošnem, kakor tudi za namen implementacije inteligentnih sistemov, ki uporabljajo strojno učenje. Vsekakor pa programski jezik ne bi smel biti ovira pri uporabi strojnega učenja, če le ne posegamo v kakšne bolj eksotične ali najnovejše pristope.

Avtorja tega dela srčno upava, da ti je knjiga podala primerno količino teoretičnega znanja o osnovah strojnega učenja ter praktičnih primerov klasifikacije. Upava, da se lahko zdaj samostojno spopadeš s prvimi izzivi na tem področju tudi v praksi. Predvsem pa je najina želja, da te je knjiga dovolj navdušila nad področjem strojnega učenja in so to bili tvoji prvi koraki v procesu spoznavanja in uporabe tehnik strojnega učenja.

Vso srečo!

LITERATURA

[1] A. Holzinger, G. Langs, H. Denk, K. Zatloukal in H. Müller, "Causability and explainability of artificial intelligence in medicine," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, let. 9, št. 4, e1312, 2019.

[2] D Fister, J. Mun, V Jagrič in T Jagrič, "Deep Learning for Stock Market Trading: A Superior Trading Strategy?" *Neural Network World*, let. 29, št. 3, str. 151–171, 2019.

[3] J. Del Ser, E. Osaba, J. J. Sanchez-Medina in I. Fister, "Bioinspired computational intelligence and transportation systems: a long road ahead," *IEEE Transactions on Intelligent Transportation Systems*, let. 21, št. 2, str. 466–495, 2019.

[4] G. Bello-Orgaz, J. J. Jung in D. Camacho, "Social big data: Recent achievements and new challenges," *Information Fusion*, let. 28, str. 45–59, 2016.

[5] F. Pedregosa, G. Varoquaux, A. Gramfort in sod., "Scikit-learn: Machine learning in Python," *the Journal of machine Learning research*, let. 12, str. 2825–2830, 2011.

[6] M. Waskom in the seaborn development team, *mwaskom/seaborn*, ver. latest, sep. 2020. DOI: `10.5281/zenodo.592845`. spletni naslov: `https://doi.org/10.5281/zenodo.592845`.

[7] J. Han, M. Kamber in J. Pei, *Data mining: Concepts and techniques: concepts and techniques*, 3. izd. Amsterdam, Netherlands: Elsevier, 2011, ISBN: 9789380931913.

[8] T. Hastie, R. Tibshirani in J. Friedman, *The elements of statistical learning: Data mining, inference, and prediction*, 2. izd., zbirka Springer Series in Statistics. Berlin, Heidelberg, Germany: Springer, 2009, ISBN: 9780387848587.

[9] S. B. Kotsiantis, I Zaharakis in P Pintelas, "Supervised machine learning: A review of classification techniques," *Emerging artificial intelligence applications in computer engineering*, let. 160, št. 1, str. 3–24, 2007.

[10] C. C. Aggarwal, A. Hinneburg in D. A. Keim, "On the surprising behavior of distance metrics in high dimensional space," v *International conference on database theory*, Springer, 2001, str. 420–434.

[11] D. W. Aha, D. Kibler in M. K. Albert, "Instance-based learning algorithms," *Machine learning*, let. 6, št. 1, str. 37–66, 1991.

[12] J. Goldberger, G. E. Hinton, S. Roweis in R. R. Salakhutdinov, "Neighbourhood components analysis," *Advances in neural information processing systems*, let. 17, str. 513–520, 2004.

[13] J. Lever, M. Krzywinski in N. Altman, *Classification evaluation*, 2016.

[14] I. Guyon in sod., "A scaling law for the validation-set training-set size ratio," *AT&T Bell Laboratories*, let. 1, št. 11, 1997.

[15] C. Schaffer, "Selecting a classification method by cross-validation," *Machine Learning*, let. 13, št. 1, str. 135–143, 1993.

[16] M. Hossin in M. Sulaiman, "A review on evaluation metrics for data classification evaluations," *International Journal of Data Mining & Knowledge Management Process*, let. 5, št. 2, str. 1, 2015.

[17] D. M. Powers, “Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation,” *Journal of Machine Learning Technologies*, let. 2, št. 1, str. 37–63, 2011.

[18] M. Sokolova, N. Japkowicz in S. Szpakowicz, “Beyond accuracy, F-score and ROC: a family of discriminant measures for performance evaluation,” v *AI 2006: Advances in Artificial Intelligence*, Berlin, Heidelberg, Germany: Springer, 2006, str. 1015–1021, ISBN: 9783540497875.

[19] M. Sokolova in G. Lapalme, “A systematic analysis of performance measures for classification tasks,” *Information Processing & Management*, let. 45, št. 4, str. 427–437, 2009.

[20] J. Demšar, “Statistical comparisons of classifiers over multiple data sets,” *The Journal of Machine Learning Research*, let. 7, št. Jan, str. 1–30, 2006.

[21] G. Jurman in C. Furlanello, “A unifying view for performance measures in multi-class prediction,” *ArXiv e-prints*, avg. 2010. arXiv: `1008.2908 [stat.ML]`.

Seznam slik

2.1 Jupyter zvezek na spletni storitvi Google Colab. 8
2.2 Spletna stran Python okolja Anaconda. 10
2.3 Domača stran JupyterLab okolja. 12
2.4 Kreacija novega zvezka v JupyterLab okolju. 13
2.5 Nov prazen Jupyter zvezek. 13
2.6 Rezultat Markdown zapisa. 17
2.7 Prvi preizkus zapisa Python kode v Jupyter zvezek. . . 19
2.8 Preizkus delovanja knjižnice `pandas`. 20
2.9 Preizkus izrisa grafa raztrosa s knjižnico `seaborn`. . . . 21
2.10 Graf raztrosa podatkovne zbirke *Iris*. 23

3.1 Model znanja v obliki odločitvenega drevesa. 28
3.2 Model znanja v obliki matematične funkcije. 29
3.3 Model znanja v obliki sprotne primerjave z že rešenimi primeri. 30
3.4 Poenostavljen prikaz procesa gradnje in uporabe modela znanja. 31
3.5 Proces uporabe modela znanja v ekspertnem sistemu. . 32
3.6 Proces učenja in uporabe modela znanja v inteligentnem sistemu. 34
3.7 Struktura podatkov, primernih za strojno učenje. . . . 35
3.8 Splošni proces klasifikacije. 40

4.1 Izbira novih čevljev s primerjavo. 42
4.2 Primerjava dveh čevljev. 42
4.3 Različne razdalje med dvema točkama. 48
4.4 Razdalje na Manhattnu. 49
4.5 Primerjava evklidske, manhattanske in kosinusne razdalje. 50
4.6 Klasifikacija instance s pomočjo k najbližjih sosedov ob različnih nastavitvah parametra k. 52
4.7 Instance podatkovne zbirke *Iris*. 65
4.8 Delitev na območja razredov glede na način računanja razdalj med instancami. 69
4.9 Delitev na območja razredov glede na število najbližjih sosedov. 71

5.1 Prikaz variance in pristranskosti. 86
5.2 Delitev množice na učno, validacijsko in testno ter njihova uporaba. 90
5.3 Stratificirana navzkrižna validacija. 93

Seznam tabel

4.1 Primer računanja razdalje med dvema čevljema. 43
4.2 Primer računanja razdalje z indikacijskimi atributi. . . 44
4.3 Primer k najbližjih sosedov. 53
4.4 Primer k najbližjih sosedov s standardiziranimi podatki. 56
4.5 Klasifikacija s k najbližjih sosedov pri različnih nastavitvah. 60

5.1 Matrika zmede za dva razreda. 96
5.2 Matrika zmede za več razredov. 104

Stvarno kazalo

algoritem strojnega učenja, 33
Anaconda, 10
atribut, 35

binarna klasifikacija, 38

delež napake, 99
delno nadzorovano učenje, 37

ekspertni sistem, 31
Evklidska razdalja, 48

F-mera, 102

gručenje, 37

indikacijski atribut, 44
instanca, 35
inteligentni sistem, 34

Jupyter Notebook, 7
JupyterLab, 7

k najbližjih sosedov, 51
klasifikacija, 38
kontingenčna tabela, *glej* matrika zmede
kosinusna razdalja, 50

leno učenje, 39, 51

Mahattanska razdalja, 49
makro agregacija metrik, 105
Markdown, 14
Matplotlib, 11
matrika zmede, 96
metrike klasifikacije, 98
mikro agregacija metrik, 107
model znanja, 28

nadzorovano učenje, 36
navzkrižna validacija, 93
nenadzorovano učenje, 36
nenasičenost, 86

normalizcija, 57
NumPy, 11

okrepitveno učenje, 37

pandas, 11
podatkovna množica, 35
podatkovno rudarjenje, 36
povprečen delež napake, 105
povprečna točnost, 105
preciznost, 101
prenasičenje, 85
priklic, 100
primerek, *glej* instanca
pristranskost, 86

regresija, 36
rez, 93

scikit-learn, 11
seaborn, 11
senzitivnost, *glej* priklic
standardizacija, 54
stratifikacija, 94
strojno učenje, 35

takojšnje učenje, 38
testna množica, 87
točnost, 98

umetna inteligenca, 35
učno množico, 87

validacijska množica, 89
varianca, 85
večrazredna klasifikacija, 38

zaupanje, *glej* preciznost
značilnica, *glej* atribut

STROJNO UČENJE
S PYTHONOM DO PRVEGA KLASIFIKATORJA

SAŠO KARAKATIČ IN IZTOK FISTER ML.
Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko, Maribor, Slovenija.
E-pošta: saso.karaktic@um.si, iztok.fister1@um.si

Povzetek
Knjiga služi kot uvod v področje strojnega učenja za vse, ki imajo vsaj osnovne izkušnje s programiranjem. Pregledajo se pomembni pojmi strojnega učenja (model znanja, učna in testna množica, algoritem učenja), natančneje pa se predstavi tehnika klasifikacije in način ovrednotenja kvalitete modelov znanja klasifikacije. Spozna se algoritem klasifikacije k najbližjih sosedov in predstavi se uporaba tega algoritma -– tako konceptualno kakor v programski kodi. Knjiga poda številne primere v programskem jeziku Python in okolju Jupyter Notebooks. Za namen utrjevanja znanja pa so ponujene naloge (tako računske, kot programerske) s podanimi rešitvami.

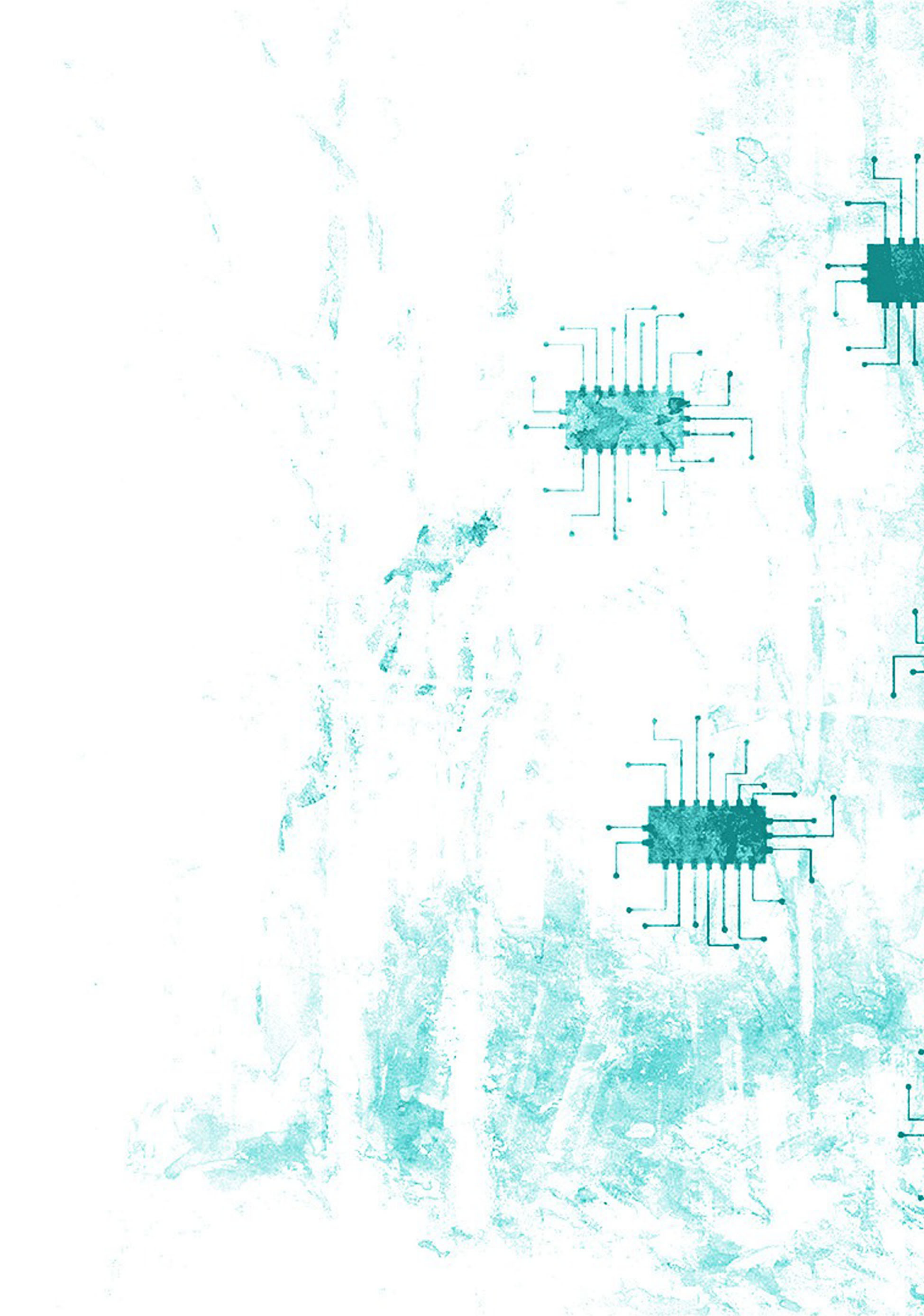
Ključne besede
strojno učenje, umetna inteligenca, klasifikacija, k najbližjih sosedov, Python

MACHINE LEARNING
CLASSIFICATION IN PYTHON

SAŠO KARAKATIČ & IZTOK FISTER JR.
University of Maribor, Faculty of Electrical Engineering and Computer Science, Maribor, Slovenia.
E-mail: saso.karaktic@um.si, iztok.fister1@um.si

Abstract
The book serves as an introduction to the field of machine learning for anyone with basic programming experience. Important concepts of machine learning (knowledge model, learning and test set, learning algorithm) are reviewed. More details are given for the classification technique and quality evaluating procedures of classification knowledge models. The classification algorithm k nearest neighbors is presented – both conceptually and in program code. The book provides many examples in the Python programming language and the Jupyter Notebooks environment. For the purpose of consolidating knowledge, several computational and programming exercises with the given solutions are offered.

Keywords
machine learning, artificial intelligence, classification, k nearest neighbors, Python



Fakulteta za elektrotehniko, računalništvo in informatiko