

# Machine Learning in Practice I

Antal Jakovac, 2025



# About the course

- Course Title: "Machine Learning in Practice I"
- instructor: [antal.jakovac@uni-corvinus.hu](mailto:antal.jakovac@uni-corvinus.hu)
- course objectives
  - ➔ why and when to use Machine Learning (ML)
  - ➔ data handling methods
  - ➔ core ML models before deep learning
  - ➔ how to build an ML project



- about the course
  - ➔ evaluation: written exams (together with exercises)
  - ➔ learning material: course slides, books, podcasts, etc.
  - ➔ books:
    - An Introduction to Statistical Learning  
(G. James , D. Witten , T. Hastie , R. Tibshirani, <https://www.statlearning.com/>)
    - Designing machine learning systems (Chip Huyen)
  - ➔ github repository: <https://github.com/ajakovac/ML-in-Practice-I>



- about the course

- ➔ evaluation: written exams (together with exercises)
- ➔ learning material: course slides, books, podcasts, etc.

- ➔ books:

- An Introduction to Statistical Learning  
(G. James , D. Witten , T. Hastie , R. Tibshirani, <https://www.statlearning.com/>)

- Design *#git clone project from command line*

- ➔ github repository

```
git clone https://github.com/ajakovac/ML-in-Practice-I.git  
cd ML-in-Practice-I
```

*#read the README.md file and follow instructions*

# Introduction



- why do we need Artificial Intelligence (Machine Learning)?

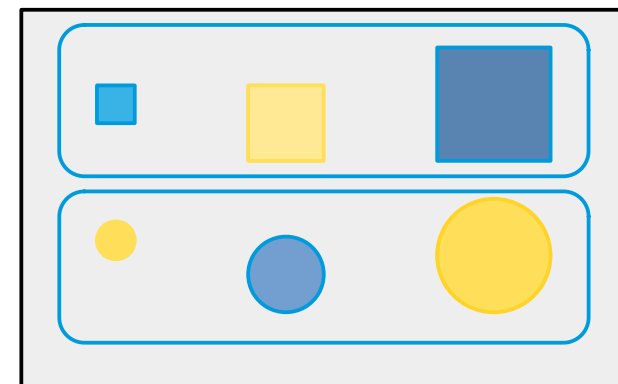
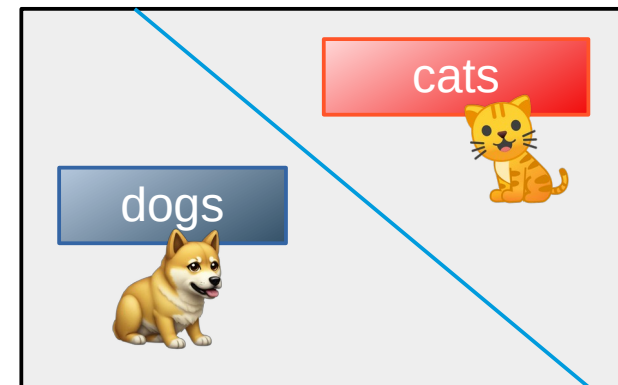
	observation	modelling	computation	conclude/action
historic times	human	human	human	human
machines	<b>machine</b>	human	human	human
computers	<b>machine</b>	human	<b>machine</b>	human
AI	<b>machine</b>	<b>machine</b>	<b>machine</b>	human

- task of AI is not to solve an actual problem, but solve modelling
- we do not know the model it uses → it can be smarter than us!



- Traditional Programming vs. ML
  - traditional: data + rules → output
  - ML: data + output → rules
- when to use ML?
  - Automation of complex tasks → heuristics
  - Adaptability
  - Performance on large data
  - Real-world examples (email spam, face detection, recommendation systems)

- Types of ML; traditionally
  - supervised:
    - show data + labels → learns the assignment
    - examples: classification, regression
  - unsupervised:
    - find patterns in unlabeled data
    - not unique → context, magnitude
    - examples: clustering, dimensionality reduction
  - reinforcement:
    - learn by interacting with the environment
    - not covered in this course







- Types of ML; other points of view
  - task-driven (System I)
    - define a task (classification, prediction, control)
    - show examples of good (eventually bad) solutions
    - train models to optimize for success on that task
    - includes most supervised/reinforcement learning examples
  - data-driven (System II)
    - reveal structures by showing similar data
    - does not need explicit labelling
    - LLM's, autoencoders, similarity learning
    - recently popular (self-supervised learning, foundation models, and retrieval-based systems)



- Typical ML tasks
  - classification
    - face, dog breeds, bird songs classification
    - spam detection, disease diagnosis, sentiment analysis
    - lot of labeled data
  - regression (price prediction, weather forecasting, energy demand)
  - clustering (customer segmentation, document grouping, biological cell types)
  - compression (PCA, visualization, autoencoders, embeddings)



- Typical ML tasks
  - ➔ decision making (robotics, game playing, resource allocation, monitoring)
  - ➔ generation (text, translation, image, data, molecular structure, code)
  - ➔ outlier analysis (email spam, suspicious customer, fraud detection, etc.)
  - ➔ recommendation systems (movies, jobs, code completion, etc.)
  - ➔ ranking/retrieval (search engines, document/data retrieval)



- somewhat overlapping areas; e.g. chemical plant monitoring system
  - ➔ anomaly detection
  - ➔ classification
  - ➔ time-series forecasting
  - ➔ reinforcement learning (control), recommendation systems



- ML algorithms (pre-deep neural networks)
  - Linear regression, logistic probability
  - Bayesian analysis, probability theory
  - Support Vector Machines
  - Decision Trees, ensemble methods
  - k-nearest neighbour method
  - Principal Component Analysis
  - Extreme Learning Machine
- majority of ML applications in production are not DNN!



## Tools

- language: Python
- packages:
  - numpy: numerical computing
  - scipy, scikit-learn: classical ML tools
  - pandas: data manipulation
  - matplotlib, seaborn: visualization



## Tools

- Python virtual environment:

```
> python -m venv .venv
```

- activate environment:

- Linux/Windows WSL:

```
> source .venv/bin/activate
```

- Windows PowerShell:

```
> .venv\Scripts\Activate
```

- Install necessary packages within the environment

```
> pip install matplotlib numpy scipy scikit-learn pandas seaborn ipykernel
```

- automation:

```
> pip freeze > requirements.txt
```

```
> pip install -r requirements.txt
```



## Tools for code development

- notebook services: Jupyter, Google Colab
- advanced editors: Visual Studio Code or similar
- Linux-like environment in Windows: WSL
- docker: containerize your work – like a standalone machine
- communication: API endpoints (Python fastAPI or Flask)



# Typical ML workflow



- Typical ML workflow
  - 1) Problem definition
  - 2) Data collection
  - 3) Data cleaning & preprocessing
  - 4) Feature engineering
  - 5) Model selection
  - 6) Training & validation
  - 7) Evaluation
  - 8) Deployment & monitoring



- Typical ML workflow


- 1) Problem definition
- 2) Data collection
- 3) Data cleaning & preprocessing
- 4) Feature engineering
- 5) Model selection
- 6) Training & validation
- 7) Evaluation
- 8) Deployment & monitoring



- what is the problem we want to solve? (classification, regression, clustering, ... )
- is ML an adequate tool?
- what does success look like? (accuracy? low error? business impact?)
- what are the domain constraints and goals (e.g. medicine data availability, privacy → practical, ethical or technical limits)
- example: predict customer churn or classify handwritten digits



- Typical ML workflow


- 1) Problem definition
- 2) Data collection 
- 3) Data cleaning & preprocessing
- 4) Feature engineering
- 5) Model selection
- 6) Training & validation
- 7) Evaluation
- 8) Deployment & monitoring

- enough data
- sampling methods → diverse data
- labelled data (if supervised)
- example: downloading a CSV of housing prices or collecting user logs



- Typical ML workflow


- 1) Problem definition
- 2) Data collection
- 3) Data cleaning & preprocessing
- 4) Feature engineering
- 5) Model selection
- 6) Training & validation
- 7) Evaluation
- 8) Deployment & monitoring

- 
- missing values, duplicates, outliers
  - normalize, scale, encode categorical features
  - convert raw inputs into usable formats
  - example: fill missing ages with median; scale prices to [0,1]



- Typical ML workflow

- 1) Problem definition
- 2) Data collection
- 3) Data cleaning & preprocessing
- 4) Feature engineering
- 5) Model selection
- 6) Training & validation
- 7) Evaluation
- 8) Deployment & monitoring

- 
- create meaningful input features
  - transform or combine raw attributes
  - select or construct new features based on domain knowledge
  - example: from date of birth → compute age; combine “city” and “job type”



- Typical ML workflow


- 1) Problem definition
- 2) Data collection
- 3) Data cleaning & preprocessing
- 4) Feature engineering
- 5) Model selection
- 6) Training & validation
- 7) Evaluation
- 8) Deployment & monitoring

- choose a suitable algorithm (e.g. linear regression; SVM; RF; k-NN)
- consider interpretability, complexity, training time
- example: try logistic regression first for classification, then maybe a random forest, or a deep neural network



- Typical ML workflow

- 1) Problem definition
- 2) Data collection
- 3) Data cleaning & preprocessing
- 4) Feature engineering
- 5) Model selection
- 6) Training & validation
- 7) Evaluation
- 8) Deployment & monitoring

- 
- split data into training and validation sets (and possibly test)
  - train on one part, validate on another
  - tune hyperparameters (e.g., via cross-validation)
  - example: train a model on 80% of the data, validate on 20%





- Typical ML workflow
  - 1) Problem definition
  - 2) Data collection
  - 3) Data cleaning & preprocessing
  - 4) Feature engineering
  - 5) Model selection
  - 6) Training & validation
  - 7) Evaluation
  - 8) Deployment & monitoring

Use metrics appropriate for the task:

- classification: accuracy, precision, recall, F1
- regression: MSE, MAE,  $R^2$
- plot confusion matrices, ROC curves, residuals
- example: check precision/recall on spam classification



- Typical ML workflow

- 1) Problem definition
- 2) Data collection
- 3) Data cleaning & preprocessing
- 4) Feature engineering
- 5) Model selection
- 6) Training & validation
- 7) Evaluation
- 8) Deployment & monitoring

- integrate the model into production (web app, API, etc.)
- monitor for drift, performance degradation
- retrain if needed (based on new data or feedback)
- example: host model on a server and track prediction accuracy over time

# Logics of Intelligence



- Consider a world  $W$  consisting of 2 pixels  
 $\{\square\square, \square\blacksquare, \blacksquare\square, \dots \blacksquare\blacksquare\}$
- $W$  also has two visible states  $A$  and  $B$ . We observe that
  - $A \rightarrow \{\square\square, \blacksquare\square, \dots \blacksquare\square\}$
  - $B \rightarrow \{\square\blacksquare, \blacksquare\blacksquare, \dots \blacksquare\blacksquare\}$
- **Step1:** problem definition: tell the state from the pixels  
→ classification task



- **Step2-3:** Data collection, cleaning, preprocessing
- in our case: observation → numerics
- the pixels are black-and-white → brightness is a number in  $[0,1]$ 
  - black: 0
  - white: 1
  - grey: somewhere in between
- 2 pixels mean 2 numbers  $(x_1, x_2)$
- *usually Step2-3 is rather tedious...*

- **Step4-5-6:** feature engineering, model selection, training (in practice these are separate steps)
- Let us plot the point pairs corresponding to state A and B

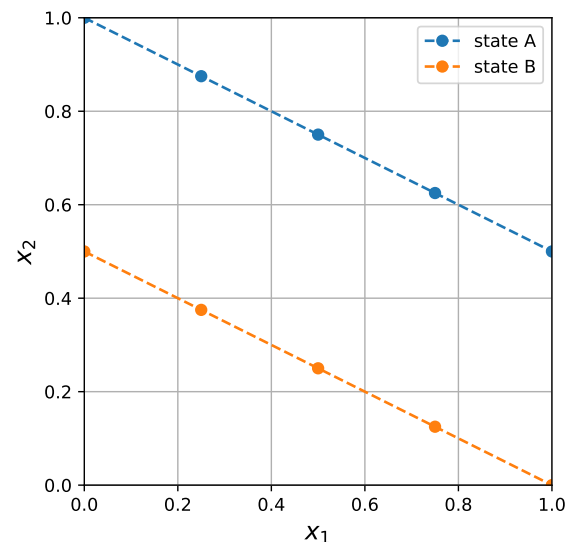
→ the points lie in subspaces (lines)

→ within state A and B

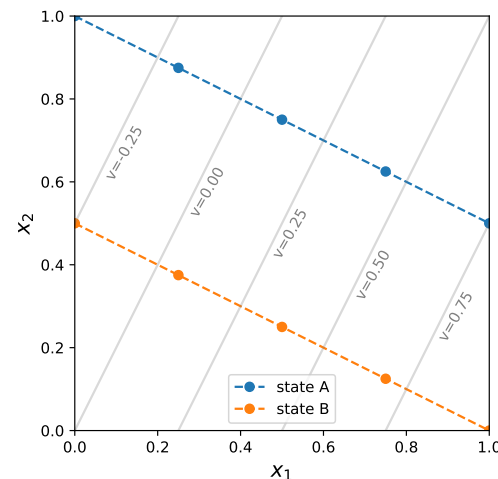
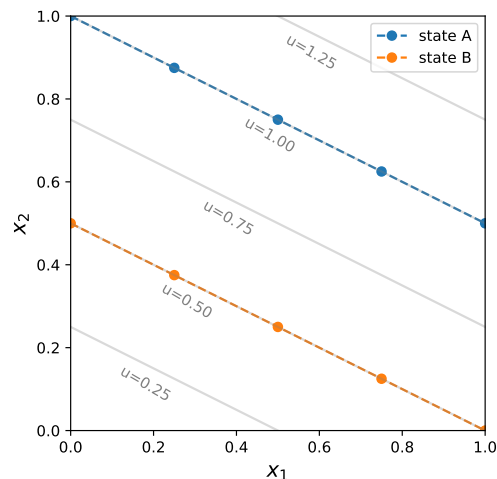
$$\frac{x_1}{2} + x_2 = \begin{cases} 1 & \text{for state A} \\ 0 & \text{for state B} \end{cases}$$

→ worth to introduce new coordinates (features)

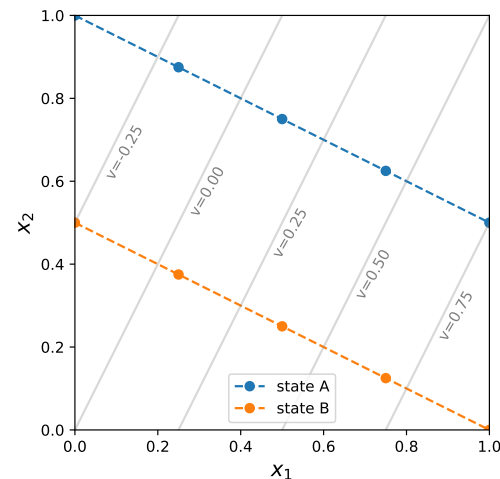
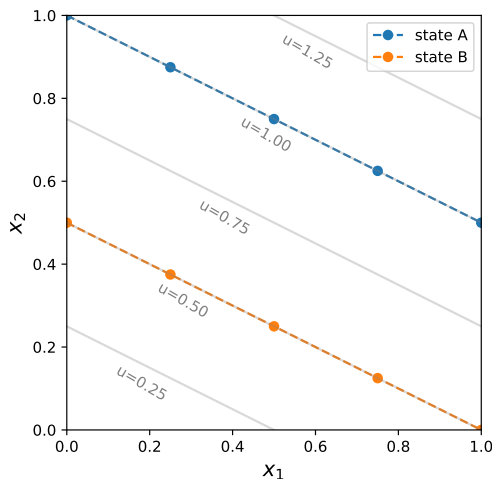
$$u = \frac{x_1}{2} + x_2, \quad v = x_1 - \frac{x_2}{2}$$



- **Step4-5-6:** feature engineering, model selection, training
- with the new coordinates  $x_1 = \frac{2u}{5} + \frac{4v}{5}$ ,  $x_2 = \frac{4u}{5} - \frac{2v}{5}$
- constant  $u$  – aligned with states; constant  $v$  – changes within a given state

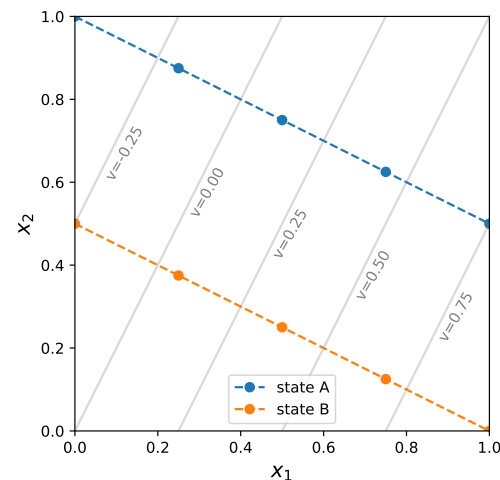
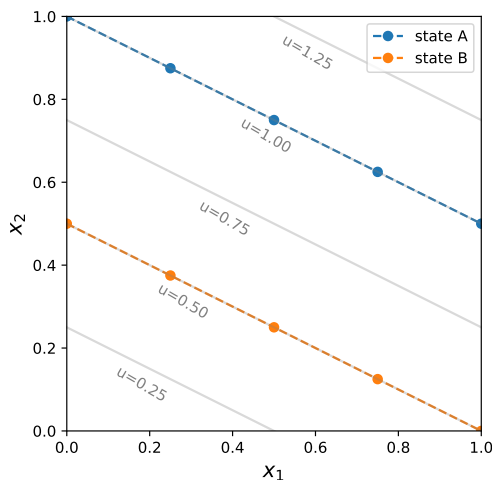


- **Step4-5-6:** feature engineering, model selection, training
- To tell apart state A and B we need only the value of  $u$ !
  - ➔  $u = 1.0$  state A
  - ➔  $u = 0.5$  state B
  - ➔ otherwise neither
- solves classification task





- **Step4-5-6:** feature engineering, model selection, training
- To describe a instance belonging to a state we need just  $v$ !
  - state A:  $v$  in  $[-0.5, 0.75]$
  - state B:  $v$  in  $[-0.25, 1]$
- solves compression,  
dimensional reduction





- **Step4-5-6:** feature engineering, model selection, training
- a perfect **feature selection** provides a **coordinate system** best aligned with the equivalence classes of the data
  - there are coordinates that are constant on each data class, and have different values on different classes  
(*relevant/selective coordinates* – good for classification)
  - there are coordinates that change within a given data class  
(*descriptive/irrelevant coordinates* – good for compression)
- **task of all model building is to find (approximately) these coordinates/features**



- **Step4-5-6:** feature engineering, model selection, training
- in practice we separate the task of coordination
  - ➔ **feature selection:** original features, simple combinations
  - ➔ **model selection:** single out a parametrizable functional space to combine the features (distance, linear- or nonlinear combinations)
  - ➔ **training:** determine the free parameters of the model that fits the data the best



- **Step4-5-6:** feature engineering, model selection, training
- **mind vs data?**
  - ➔ data/models importance ratio → data are central importance in real world applications
  - ➔ are data enough? do we possess the necessary knowledge?
  - ➔ present approach: data engineering is the most important, few fundamental research (cf. LeCun vs Wang in Meta)



- **Step7:** evaluation
- try to estimate how well the original task was solved
  - ➔ class reconstruction accuracy
  - ➔ robustness, proclivity for failures
  - ➔ treatment of outliers
  - ➔ business benefits



- **Step8:** deployment and monitoring
- in a real-life application the ML code is used as a product
- requires continuous monitoring
  - data/environment may change
  - new classes appear
  - errors may occur



**takehome message:** Task of the intelligence is to

find the **relevant features** and **adapt** their values to the task

# Organizing ML projects



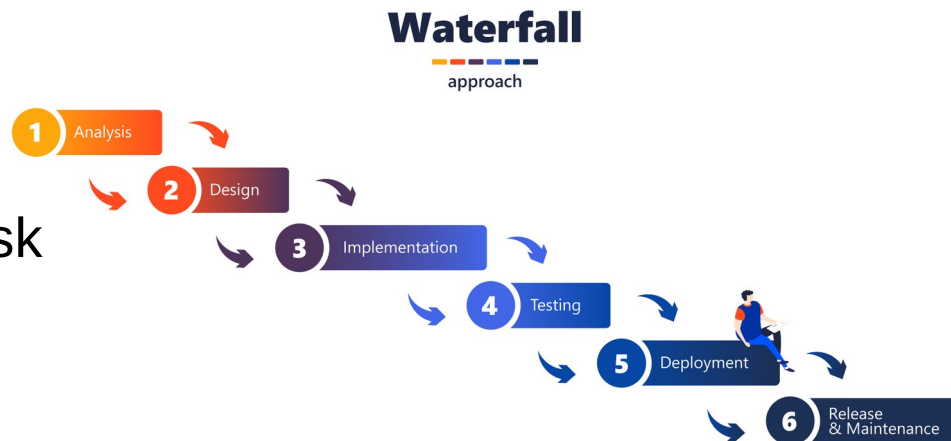
# Organizing ML projects

- two popular workflow schemes today: **waterfall** vs. **agile**

- waterfall** workflow:

- clear documentation of the task
- predictable timeline
- inflexible
- late discovery of errors

- good for well-defined projects** (like bridge building)

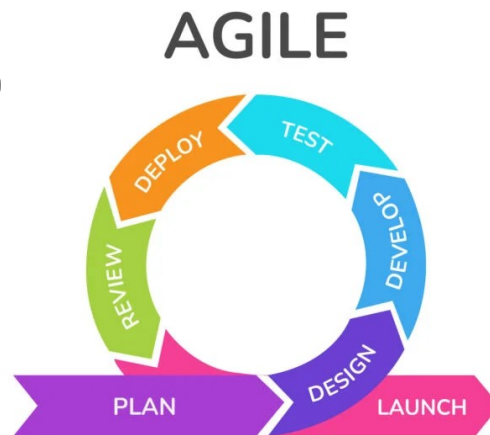


# Organizing ML projects

- **agile** workflow: (agile manifesto, description)
  - ➔ individuals > processes
    - collaborative vs rigid teams
  - ➔ working model > detailed documentation
    - Minimal Viable Product MVP
    - improvement

*"If you're not embarrassed by the first version of your product, you've launched too late."*

— Reid Hoffman, founder of LinkedIn

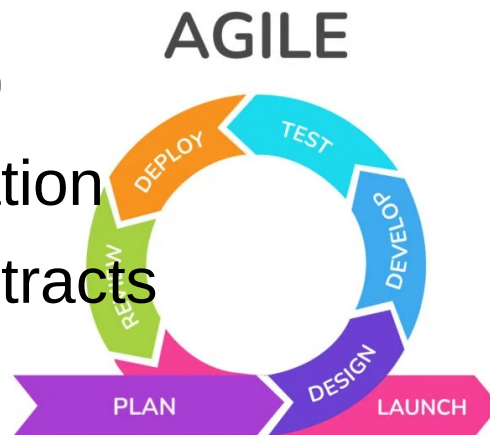


## Goal: **Solve transportation**

- MVP: Skateboard 🛹
- Next: Bicycle 🚲
- Then: Motorcycle 🏍️
- Finally: Car 🚗

# Organizing ML projects

- **agile** workflow: (**agile manifesto**, **description**)
  - ➔ customer collaboration > contract negotiation
    - customer needs even over signed contracts
  - ➔ responding to change > following a plan
    - original ideas/original task estimates may fail
    - continuous adaptation
  - ➔ launch a “good enough” product



# Organizing ML projects

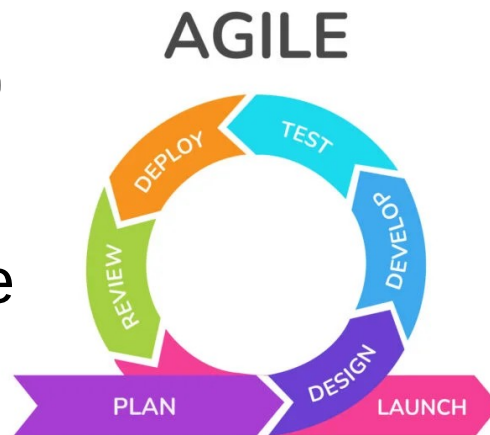
- **agile** workflow: (agile manifesto, description)

- ➔ pros:

- flexible, self-improving, communicative
- customer-centric

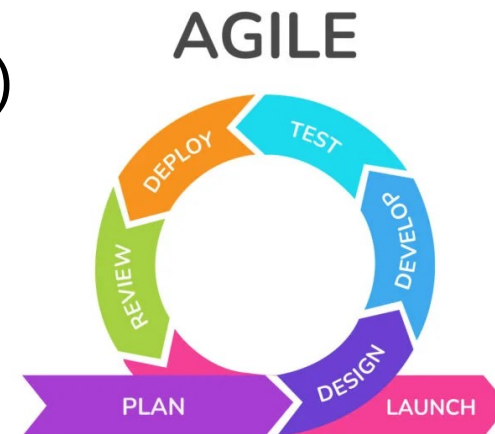
- ➔ cons:

- over-emphasized meetings (“agile theater”)
- over-controlled
- less predictable



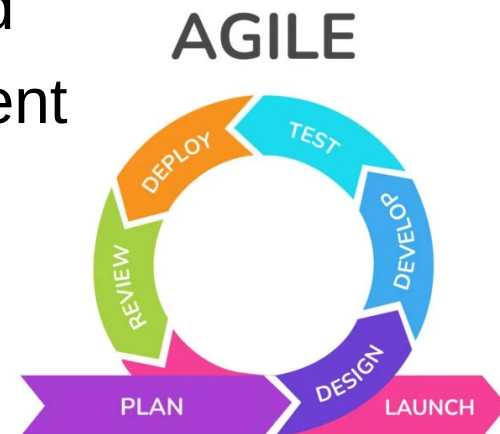
# Organizing ML projects

- **agile** workflow: (**agile manifesto**, **description**)
  - ➔ loops, sprints
  - ➔ user interface/user experience (UI/UX)
  - ➔ CI/CD, CI/CD pipelines  
continuous integration, continuous deployment
  - ➔ Kanban/Jira: project development visualization  
(backlog, to do, in progress, testing, done)




# Organizing ML projects

- typical small group agile project
  - ➔ define the goal (user stories), create a backlog (list of tasks)
  - ➔ make repo (github), branches
  - ➔ make MVP, backend API, frontend, testbed
  - ➔ get MVP working, deploy to test environment
  - ➔ collect feedback (team + users)
  - ➔ refine tasks, and directions to proceed






## Structure of a professional ML organization

-  Business side
  - ➔ client/stakeholder: business requirements (e.g., “reduce equipment downtime,” “recommend better products”)
  - ➔ Business Analyst / Product Manager (translates business goals into ML problems, acts as the bridge between client and technical teams)
  - ➔ Project/ Product Manager (budgeting, timelines, and resource allocation, Coordinates with business analysts and technical leads to estimate cost)

## Production level ML organization

-  Data/model experts
  - ➔ data engineer: build and maintain data pipelines → data collection, cleaning, validating, transformation (ETL/ELT), storing (database, data lake), monitoring
  - ➔ ML engineer/data scientist
    - feature generation
    - design, train, and fine-tune ML models
    - explores and experiments with data, build prototypes
    - performance evaluation




## Production level ML organization



### Software & Infrastructure

- ➔ software architect: designs overall system, service planning, integration with other services, APIs, databases
- ➔ software developer (coder): implements algorithms, frontend-backend
- ➔ DevOps/MLOps engineer:
  - automates deployment, monitoring, scaling of ML models
  - works with Docker, Kubernetes, CI/CD pipelines, cloud platforms.
  - ensures the ML system is reproducible, reliable, and easy to maintain.

## Production level ML organization

-  Operations, Monitoring & Safety
  - ➔ QA / Test Engineer: tests software and ML pipelines (unit tests, integration tests, stress tests), ensures reliability before release
  - ➔ Monitoring & Reliability Engineer: monitors system health after deployment.; tracks data drift, performance degradation, unusual patterns
  - ➔ Security Engineer: ensures compliance with data privacy, protects against adversarial attacks, model leaks.



## **The Full Pipeline in a Story (user story)**

- Client says: “I need early warnings for dangers in a chemical plant.”
- Business Analyst refines: “This means anomaly detection on sensor signals.”
- Data Engineer sets up pipelines to collect sensor data.
- Data Scientist explores, builds a prototype anomaly detector.
- ML Engineer optimizes the model for latency and accuracy.



## **The Full Pipeline in a Story (user story)**

- Software Architect decides how it fits into the production system.
- Developers implement the service around the model.
- DevOps/MLOps deploys it to the cloud with monitoring.
- QA & Monitoring check performance and flag issues.
- Security Engineer ensures compliance and robustness.

# Programming code structure

How to organize a (ML) code?

- points of view:
  - ➔ effective code development (also in teams)
  - ➔ scalability
  - ➔ extensibility
  - ➔ easy maintenance



## Original approach: monolith program structure

- UI, business logic, data access in one code
- coding logic: *input data* → *transformation* → *resulting data*
- best fits for imperative/procedural languages (like C, Fortran)
  - ➔ program state: mutable variables
  - ➔ program is a chain of commands, changing the state
- natural approach → original programming style



## problems:

- hard to trace bugs or change code:
  - variable change appear in different functions
  - frequently hidden state change in procedures
  - in a large code intractable structure
- functions are specific → not reusable
- experience: monolith codes become unmaintainable  
(*software complexity crisis in the 1970s–80s*)
- still used for starting a project (MVP)

```
y = 3 #global state variable
```

```
def function(x):  
    x += y  
    y += 1 #changes y tacitly  
    return x
```





different type of solutions:

- object oriented programming
  - ➔ data and the corresponding procedures are bundled: "objects" of the world
  - ➔ abstraction: hierarchy of classes
  - ➔ inheritance: classes are reusable
  - ➔ examples: C++, Java

```
class MyClass:
    def __init__(self, state):
        self.state = state

    #change of state is a class method
    def rotate(self, angle):
        self.state → self.state'(angle)
        return self
```



- functional programming

- stateless programs, immutable variables
- functions give back new objects (pure functions)
- based on mathematical logic ( $\lambda$ -calculus)  
e.g.  $\lambda x. x+1$  and their application, e.g.  $(\lambda x. x+1) 5 \rightarrow 6$
- basic elemental functions: see next page

```
counter=0
def impure():
    counter+=1 #side effect
    return counter
def pure(x):
    return x+1 #no side effect
```



- basic elemental functions

- map: `map ( $\lambda x. x*2$ ) [1,2,3]  $\rightarrow$  [2,4,6]`

- reduce/fold: `fold (+) 0 [1,2,3]  $\rightarrow$  6`

- filter: `filter even [1,2,3,4]  $\rightarrow$  [2,4]`

- zip: `zip ['a','b','c'] [1,2,3]  $\rightarrow$  [('a',1),('b',2),('c',3)]`

- compose: `(compose f g) x  $\rightarrow$  f(g(x))`

- examples: Haskell, Lisp, F#

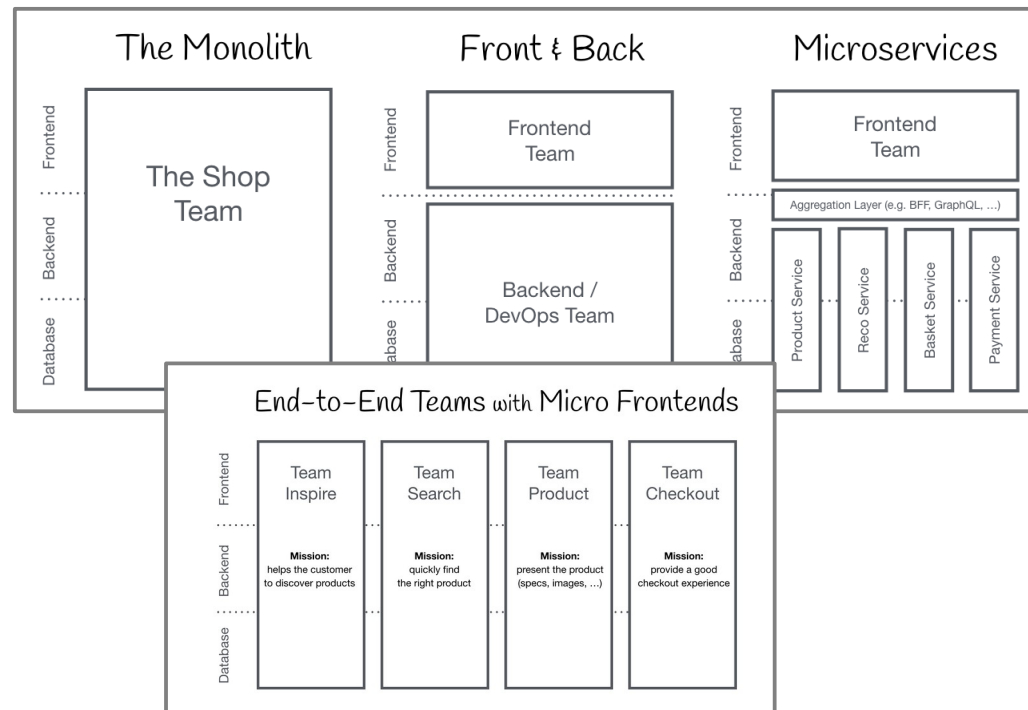
- not popular in the original form, but lot of elements are applied



- mixed (multi-paradigm) languages:
  - ➔ allow all kind of programming style
  - ➔ examples: Python, JavaScript, Rust, Swift,
  - ➔ e.g. Python has map, filter, reduce, based on lists.
  - ➔ Python supports OOP, classes, inheritance, etc.

## Other way of deviating from monolithic imperative coding

- monolithic structure
- frontend+backend
- microservices
- micro frontend structure  
(c.f. <https://micro-frontends.org/>)



## Backend + frontend

- rise of web (1990s) browsers became universal clients
- specialized tools for browser programming (like HTML, CSS, JavaScript)
- task solving (backend) + visualization (frontend) separately
  - ➔ develop separately, different languages, tools
  - ➔ scale BE and FE separately
  - ➔ needs interface management → see next page



- functions designed for intra-code communication, can not be applied any more
- inter-service communication standards:
  - ➔ APIs (application programming interface → synchronous)
  - ➔ queues (message buffer → asynchronous)



- APIs – call server functions (endpoints)
  - ➔ needs continuously running background processes (web server process), listening to input channels (ports)
  - ➔ endpoints: well defined input and output formats, usually in json
  - ➔ communication → REST API: get, put/patch, post, delete
  - ➔ in Python: Flask, fastAPI
  - ➔ can be asynchronous, too

*#fastAPI snippet*

```
from fastapi import FastAPI
app = FastAPI()
@app.get("/")
def read_root():
    return {"message": "Hello ✨"}
```

-----  
*#start service*

```
uvicorn main:app
```

-----  
*#call service*

```
open http://127.0.0.1:8000, or
curl http://127.0.0.1:8000/
results {"message": "Hello ✨"}
```



- queues: async send message
  - ➔ message broker runs in background, it stores messages
  - ➔ worker processes deliver messages for subscribers/consumers
  - ➔ data plain text or json
  - ➔ metadata + payload
  - ➔ in Python: queue + threading

*#queue application example*

```
import queue  
q = queue.Queue()
```

*#put data to queue*

```
q.put(item)
```

*#read data from queue*

```
item = q.get()
```

*#... process data ...*

```
q.task_done()
```



## Microservices + frontend

- good scaling properties (e.g. web users) → requiring independence of elementary tasks became popular
- lot independent, narrowly scoped (micro) services
  - separate development (languages, tools, teams), deployment
  - separate scaling
  - resilience: bugs are confined, not affect the whole system
- examples Netflix, Amazon → hype in 2010s



- new challenge: how to organize a lot of services
- microservice management: docker, Kubernetes

*#example Dockerfile*

```
FROM python:3.12-slim
```

```
WORKDIR /app
```

```
COPY main.py .
```

```
RUN pip install fastapi uvicorn
```

```
CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

*#example docker-compose.yml*

```
services:
```

```
  service1:
```

```
    build: ./service1
```

```
    container_name: service1
```

```
    ports:
```

```
      - "8001:8000"
```

```
  service2:
```

```
    build: ./service2
```

```
    container_name: service2
```

```
    ports:
```

```
      - "8002:8000"
```

-----  
docker-compose up --build



- new challenge: how to organize a lot of services
- CI/CD pipelines:
  - ➔ continuous improvement
  - ➔ continuous deployment
  - ➔ team communication
- example: github



- new challenge: how to organize a lot of services
- CI/CD pipelines:
  - ➔ continuous improvement
  - ➔ continuous deployment
  - ➔ team communication
- example: github

```
#example .github/workflows/ci.yml
name: CI
on:
  push: branches: ["main"]
  pull_request: branches: ["main"]
jobs:
  build-and-test:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v4
      - name: Set up Python
        uses: actions/setup-python@v5
        with:
          python-version: "3.12"
      - name: Install dependencies
        run: |
          python -m pip install --upgrade pip
          pip install -r requirements.txt
          pip install pytest
```



- new challenge: how to organize a lot of services

- CI/CD pipelines:

- continuous integration
- continuous deployment
- team communication

- example: github

*#practical example: git clone project*

```
git clone https://github.com/ajakovac/ML-in-Practice-I.git  
cd ML-in-Practice-I
```



- despite all advantages microservices have drawbacks
  - ➔ operational complexity → manage hundreds of services
  - ➔ network calls → latency, performance issues
  - ➔ interprocess debugging is hard



- Current trends (2020s → now)
  - ➔ modular monolith applications
  - ➔ microservices where they make sense
  - ➔ functions-as-a-service (FaaS)
    - in cloud services like AWS, Azure, Google cloud
    - locally they seem like functions
    - behave like services when deployed
    - run on demand





- Current trends

- ➔ modular n

- ➔ microserv

- ➔ functions-

- in clou

- locally

- behave

- run on

```
#example in AWS
```

```
#lambda_function.py
```

```
def lambda_handler(event, context):  
    name = event.get("name", "my_name")  
    return { "statusCode": 200,  
            "body": f"Hello, {name}! ✨"
```

```
-----  
zip function.zip lambda_function.py
```

```
aws lambda create-function \
```

```
--function-name helloLambda \
```

```
--runtime python3.12 \
```

```
--role arn:aws:iam::<YOUR_ACCOUNT_ID>:role/lambda-ex-role \
```

```
--handler lambda_function.lambda_handler \
```

```
--zip-file fileb://function.zip  
-----
```

```
curl -X POST \
```

```
-H "Content-Type: application/json" \
```

```
-d '{"name": "MLcourse"}' \
```

```
https://abcd1234.execute-api.us-east-1.amazonaws.com/default/hello
```

## **Code infrastructure → software architecture**

- use few necessary microservices → clear task definition
- use lambda for simple services
- containerize microservices → portability, environmental stability
- use container management → docker (Kubernetes)
- use CI/CD management → github or similar

# Data management

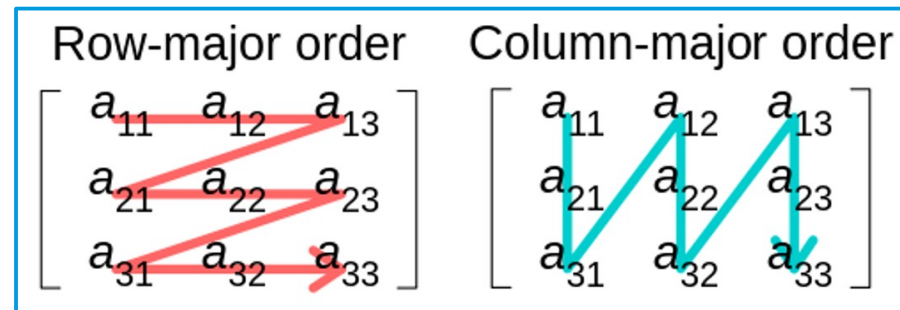


## data sources:

- user input (text, image, video, etc.) – often not well formed, errors
- system generated (logs, measurements, predictions) – well formed data, easy-to-use
- own data, own clients (first-party), other companies, their own clients (second-party), other companies not own clients (third-party – e.g. internet usage, traffic, etc)

## data formats

- CSV (comma separated values)
  - text, human readable
  - tabular, row-major
  - fast to get data belonging to the same example
- Parquet
  - binary – more compact, but not human readable
  - tabular, column major
  - fast to get data belonging to the same feature



## data formats

- CSV (comma-separated values)

- text, human-readable
- tabular, row-oriented
- fast to get data out

- Parquet

- binary – more compact
- tabular, column-oriented
- fast to get data out

*#example books.csv*

title,author,publisher,year

The Hobbit,J.R.R. Tolkien,George Allen & Unwin,1937

The Fellowship of the Ring,J.R.R. Tolkien,George Allen & Unwin,1954

Nineteen Eighty-Four,George Orwell,Secker & Warburg,1949

Brave New World,Aldous Huxley,Chatto & Windus,1932

The Catcher in the Rye,J.D. Salinger,"Little, Brown and Company",1951

*#read data using pandas*

```
import pandas as pd
```

```
df = pd.read_csv("books.csv")
```

```
print(df)
```

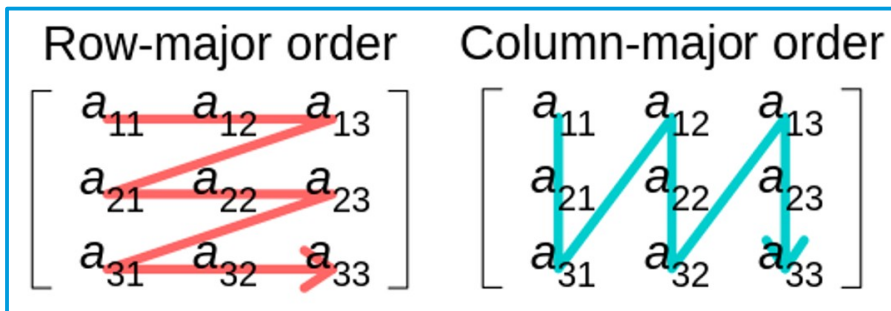
*#output*

	title	author	publisher	year
0	The Hobbit	J.R.R. Tolkien	George Allen & Unwin	1937
1	The Fellowship of the Ring	J.R.R. Tolkien	George Allen & Unwin	1954
2	Nineteen Eighty-Four	George Orwell	Secker & Warburg	1949
3	Brave New World	Aldous Huxley	Chatto & Windus	1932
4	The Catcher in the Rye	J.D. Salinger	Little, Brown and Company	1951

## data formats

- CSV (comma separated values)

- text, human readable
- tabular, row-major
- fast to get data belonging to the same example



- Parquet

- binary – more compact
- tabular, column-major
- fast to get data belonging to the same example

```
#store in parquet – needs fastparquet to be installed
df.to_parquet("books.parquet", engine="fastparquet", index=False)
-----
#store in parquet – needs fastparquet to be installed
df_parquet = pd.read_parquet("books.parquet", engine="fastparquet")
```



## data formats:

- JSON (JavaScript Object Notation) → today's standard
  - hierarchical structure
  - text, human readable
  - key-value pairs
  - in Python: dict
  - supported by all languages



## data formats:

- JSON (JavaScript Object Notation)

- hierarchical
- text, human-readable
- key-value pairs
- in Python: `dict`
- supported by many languages

```
#example books.json
```

```
[  
  {  
    "title": "The Hobbit",  
    "author": "J.R.R. Tolkien",  
    "publisher": "George Allen & Unwin",  
    "year": 1937  
  },  
  {  
    "title": "The Fellowship of the Ring",  
    "author": "J.R.R. Tolkien",  
    "publisher": "George Allen & Unwin",  
    "year": 1954  
  },  
  ...  
]
```

```
#read with pandas
```

```
df = pd.read_json("books.json")
```



## data models:

- for more complicated data, and if queries are required, storing data in simple files is not enough
- instead: databases with dedicated database handlers
  - ➔ relational (e.g. Sqlite)
  - ➔ nonrelational (e.g. Redis, Mongo)



## data models:

- relational
  - tabular logic: main tables, subtables for standardization, index tables for faster query
  - strict schemas
  - query language for data retrieval → SQL (structured query language): declarative, specify the result, not the algorithm
  - **pros**: simple logic, widely used
  - **cons**: abundant schema systems, lots of superficial data, tedious to introduce new features, complicated table structure

## data models:

- relational

- tabular logic  
faster query
- strict schema
- query language  
declarative,
- **pros:** simple
- **cons:** abundant  
introduce n

```
#example sqlite application - built in Python package
import sqlite3

#create/connect database
conn = sqlite3.connect("Data/books.db")
cur = conn.cursor()

#create tables
cur.execute("""
CREATE TABLE IF NOT EXISTS publishers (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL
)
""")

cur.execute("""
CREATE TABLE IF NOT EXISTS books (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    title TEXT NOT NULL,
    author TEXT NOT NULL,
    year INTEGER,
    publisher_id INTEGER,
    FOREIGN KEY (publisher_id) REFERENCES publishers(id)
)
""")
```

## data models:

- relational

- tabular logic  
faster query
- strict schema
- query language  
declarative
- **pros:** simple
- **cons:** abundant  
introduce n

```
#insert a publisher
```

```
cur.execute("INSERT INTO publishers (name) VALUES (?)", ("George Allen & Unwin",))
```

```
publisher_id = cur.lastrowid
```

```
# Insert a book linked to that publisher
```

```
cur.execute("INSERT INTO books (title, author, year, publisher_id) VALUES  
(?, ?, ?, ?)", ("The Hobbit", "J.R.R. Tolkien", 1937, publisher_id))
```

```
#commit changes
```

```
conn.commit()
```

```
-----
```

```
# Query join
```

```
cur.execute("""
```

```
SELECT b.title, b.author, p.name AS publisher
```

```
FROM books b
```

```
JOIN publishers p ON b.publisher_id = p.id
```

```
""")
```

```
print(cur.fetchall())
```

```
conn.close()
```

```
-----
```

```
#output
```

```
[('The Hobbit', 'J.R.R. Tolkien', 'George Allen & Unwin')]
```

## data models:

- NoSQL: no SQL → not only SQL
  - pros: no schemas → flexibility, scalability, treat of unstructured data
  - cons: storage capacity, performance trade-offs, data chaos
  - document stores: data are stored as a JSON (string)
    - query can be a more difficult task
    - examples: mongoDB, CouchDB – create own index tables for effectivity
  - key-value pairs: for fast query (example: RedisDB, DynamoDB)
  - graph databases: concentrate on relations (nodes and edges)  
(Examples: Neo4j, Amazon Neptune, ArangoDB)



## data models:

- NoSQL: no S

- ➔ pros: no sc

- ➔ cons: stora

- ➔ document s

- query ca

- examples

- ➔ key-value p

- ➔ graph data

(Examples: N

```
#install mongodb-image and pymongo
```

```
docker run -d --name mongodb -p 27017:27017 -v ~/my_data/mongo:/data/db mongo:7.0
```

```
pip install pymongo
```

```
-----
```

```
#runs a service in the background – start/stop it
```

```
docker start/stop mongodb
```

```
-----
```

```
#install mongodb and pymongo
```

```
from pymongo import MongoClient
```

```
# 1. Connect to local MongoDB
```

```
client = MongoClient("mongodb://localhost:27017/")
```

```
# 2. Create (or use existing) database
```

```
db = client["library"]
```

```
# 3. Create collections
```

```
publishers = db["publishers"]
```

```
books = db["books"]
```

```
# 4. Clear old data (for demo purposes)
```

```
publishers.delete_many({})
```

```
books.delete_many({})
```

## data

- No

*# 5. Insert publishers*

```
pub_allen = publishers.insert_one({"name": "George Allen & Unwin"}).inserted_id
pub_secker = publishers.insert_one({"name": "Secker & Warburg"}).inserted_id
pub_chatto = publishers.insert_one({"name": "Chatto & Windus"}).inserted_id
pub_little = publishers.insert_one({"name": "Little, Brown and Company"}).inserted_id
```

*# 6. Insert books with publisher references*

```
books.insert_many([
    {"title": "The Hobbit", "author": "J.R.R. Tolkien", "year": 1937, "publisher_id": pub_allen},
    {"title": "The Fellowship of the Ring", "author": "J.R.R. Tolkien", "year": 1954, "publisher_id": pub_allen},
    {"title": "Nineteen Eighty-Four", "author": "George Orwell", "year": 1949, "publisher_id": pub_secker},
    {"title": "Brave New World", "author": "Aldous Huxley", "year": 1932, "publisher_id": pub_chatto},
    {"title": "The Catcher in the Rye", "author": "J.D. Salinger", "year": 1951, "publisher_id": pub_little},
])
```

*# 7. Query: find all books and join publisher info manually*

```
for book in books.find():
    publisher = publishers.find_one({"_id": book["publisher_id"]})
    print(f"{book['title']} by {book['author']} "
          f"was published by {publisher['name']} in {book['year']}")
```





## data storage:

- data warehouse:
  - stores historical data → structured, cleaned, integrated
  - optimized for analysis, reporting → few but very large, complex queries
  - ETL (extract, transform, load)
  - OLAP (online analytic processing)
  - examples: historical sales data, stock price data
  - solutions: Amazon Redshift, Google BigQuery

## data storage:

- database:
  - stores current, operational data.
  - used for day-to-day transactions (insert, update, delete).
  - optimized for fast read/write, many small simple queries
  - example: an e-commerce database storing users, orders, payments
  - OLTP (Online Transaction Processing)
  - solutions: MySQL, PostgreSQL, Oracle DB, MongoDB, DynamoDB



## data storage:

- data lake:

- stores data in raw form → future-proof
- application goal not needed to be specify
- cheap and scalable storage (HDFS, Amazon S3, Azure Data Lake Storage, Google Cloud Storage)
- schema-on-read solutions (ELT, extract, load, transform) → processing engines (Kafka, TensorFlow, Athena)
- example: log storage, IoT data storage



**data sampling:** collecting all data usually not feasible

- nonprobability sampling → focus on data collection; prone to selection bias
  - ➔ convenience sampling: take all what is available
  - ➔ snowball sampling: start with a small set, and follow its descendants
  - ➔ judgement sampling: experts say what to collect
  - ➔ quota sampling: collect the same number of data from different areas (like age groups)
  - ➔ example of bias: psychology experiments (university students), sentiment analysis (wordy people), medical data (unsuccessful treatments)

## data sampling:

- probability sampling → needs a data model → biased?
  - stratified sampling: collect the same amount of data per class
  - weighted sampling: take into account data scaled with probability
    - in choice: class A 25%, class B 75% → consider A samples 3 times
    - in training: give weight for underrepresented classes (see later)
  - reservoir sampling: keep the first k sample, and replace it randomly
  - importance sampling

$$E_P[x] = \sum_x x P(x) = \sum_x x P \frac{P(x)}{Q(x)} Q(x) = E_Q \left[ x P \frac{P(x)}{Q(x)} Q(x) \right]$$



## data augmentation:

- if there are not enough data, or not representative enough, we can transform data to generate new ones (symmetries)
  - ➔ rotation, scaling
  - ➔ perturbation, noising
  - ➔ texture or style change
- data generation: simulations in artificial environment (e.g. self-driving cars)



## labelling:

- for classification we should collect examples in different classes
- problems:
  - ➔ how unambiguous are the labels? → multi-label classes
  - ➔ how precise are the labels? → no database is 100% correct!
  - ➔ what to do with missing labels? → label functions, semi-supervised, etc.
- hand labelled data, naturally labelled data (context is given), automatically generated data

# Monitoring and evaluation



## How can we assess the success of the ML results?

- not unique! depends on the problem on hand
  - ➔ example: percentage of correctly classified images
  - ➔ if the probability of belonging to class A is 99%, then the simple method saying that *all* elements belong to A is 99% accurate!
  - ➔ clearly not good, if the task is to find the 1% not-A class!



- baselines: how accurate are simple methods and other approaches?
  - random baseline: choose class randomly →  $1/N$  probability for balanced
  - zero rule: all belonging to one class → high probability for imbalanced
  - simple heuristic → use a simple proxy (feature)
  - human experts → how well do humans perform?
  - other ML methods

- confusion matrix

- correct classes vs. estimated classes
- works for multi-class evaluation
- in probabilistic sense joint probability:  $C_{ij} = P(\text{predicted}_i, \text{actual}_j)$

	Actual Yes	Actual No	Predicted Total
Predicted Yes	TP=0.510	FP=0.038	PP=0.548
Predicted No	FN=0.032	TN=0.420	PN=0.452
Actual Total	AP=0.542	AN=0.478	1.0

- correctness measures for 2-class evaluation
  - *accuracy*: prob. of choosing the correct class:  $(TP+TN)$
  - *precision*: conditional for Predicted Yes:  $P(\text{actual}_p | \text{predicted}_p) = \frac{TP}{PP}$
  - *recall*: conditional for Actual Yes:  $P(\text{predicted}_p | \text{actual}_p) = \frac{TP}{AP}$
  - *type I error*: probability of false alarm:  $P(\text{predicted}_p | \text{actual}_N) = \frac{FP}{AN}$
  - *type II error*: probability of missing signal:  $P(\text{predicted}_N | \text{actual}_p) = \frac{FN}{AP}$



- controlled case → measure does not matter
  - probability of class A or B is 50%, predict symmetrically
  - accuracy=98%
  - for class A: precision=98%, recall=98%, type II error=2%
  - for class B: precision=98%, recall=98%, type I error=2%

	Actual Yes	Actual No	Predicted Total
Predicted Yes	TP=0.49	FP=0.01	PP=0.5
Predicted No	FN=0.01	TN=0.49	PN=0.5
Actual Total	AP=0.5	AN=0.5	1.0



- extreme cases
  - classify everything as class A, but probability of class A is 99%
  - accuracy=99%
  - for class A: precision=99%, recall=100%, type II error=0%
  - for class B: precision=?, recall=0%, type I error=100%

	Actual Yes	Actual No	Predicted Total
Predicted Yes	TP=0.99	FP=0.01	PP=1.0
Predicted No	FN=0.0	TN=0.0	PN=0.0
Actual Total	AP=0.99	AN=0.01	1.0

- extreme cases
  - classify everything randomly, but probability of class A is 98%
  - accuracy=50%
  - for class A: precision=98%, recall=50%, type II error=50%
  - for class B: precision=2%, recall=50%, type I error=50%

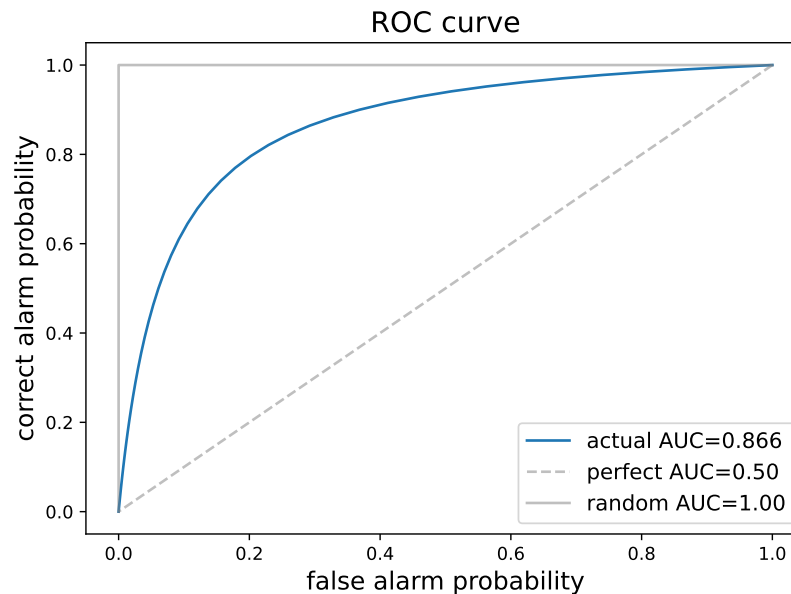
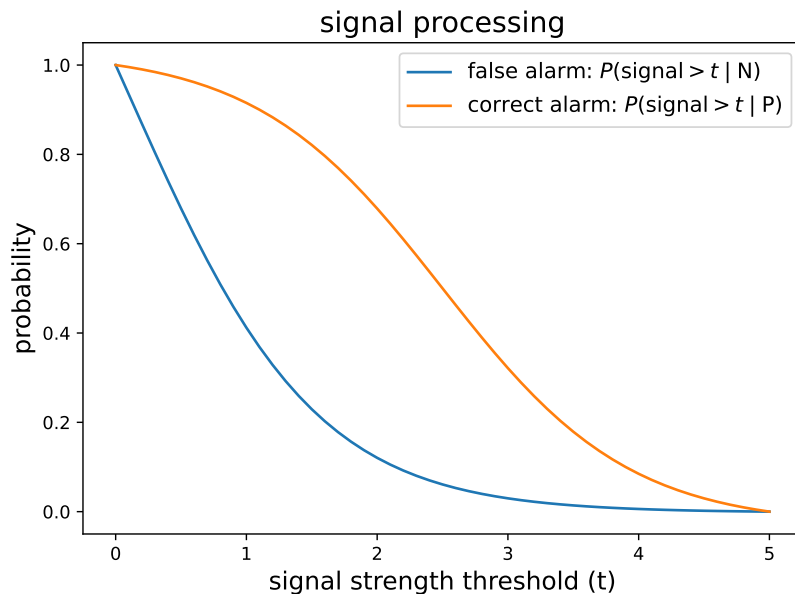
	Actual Yes	Actual No	Predicted Total
Predicted Yes	TP=0.49	FP=0.01	PP=0.5
Predicted No	FN=0.49	TN=0.01	PN=0.5
Actual Total	AP=0.98	AN=0.02	1.0



- effect of a threshold on the prediction
  - in lot of cases Yes/No decision comes from comparing a signal with a threshold → e.g. smoke detector signal vs. danger level
  - ROC (receiver operating characteristics)
    - $P(\text{predicted}_P | \text{actual}_P)$  vs.  $P(\text{predicted}_P | \text{actual}_N)$
    - recall vs. type I error
    - correct alarm probability vs. false alarm probability
  - AUC: area under curve



- effect of a threshold on the prediction

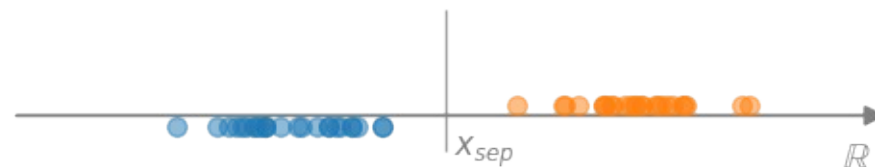


# Data modelling

- task: find out the selective/descriptive features
- depends both on the dataset and the context
- selective: constant on a class, descriptive: changes within a class  
class depends on the context
- finding the selective coordinates is the main challenge
- we start with simple 1D classification problems, and generalize to several dimensions

# One dimensional dataset

- Simplest basic case: characterization of one-dimensional data
  - data  $\rightarrow x \in \mathbb{R}$  (numbers for analytic characterization)
- Separation of two 1D sets
  - assume that in given sections belong (approximately) to a single class
  - basic case: one section  $\rightarrow A$ , other section  $\rightarrow B$
  - if the two sets are separated (hard margin)  $\left\{ \begin{array}{l} x \in A \Leftrightarrow x < x_{sep} \\ x \in B \Leftrightarrow x > x_{sep} \end{array} \right\}$
  - overlapping case?  $\rightarrow$  no perfect solution



# One dimensional dataset

- overlapping regions → no perfect solution (heuristics)

- choose  $x_{sep}$  somewhere in the overlapping region (eg. midpoint)

- define an error function in the overlapping region, e.g.

- $p=1$  (“hinge loss”) → median,  $p=2$  → mean

$$L_p(x_{sep}) = \sum_{x \in \text{overlap}} |x - x_{sep}|^p$$

- we can weight the points



$$0 = \frac{d}{dx_{sep}} \sum_{i=1}^N (x_i - x_{sep})^2 = 2 \sum_{i=1}^N (x_{sep} - x_i) \Rightarrow x_{sep} = \frac{1}{N} \sum_{i=1}^N x_i$$

$$\sum_{i=1}^N |x_i - x_{sep}| = \sum_{x_i > x_{sep}} (x_i - x_{sep}) + \sum_{x_i < x_{sep}} (x_{sep} - x_i) = (N_{<} - N_{>}) x_{sep} + \sum_{x_i > x_{sep}} x_i - \sum_{x_i < x_{sep}} x_i \Rightarrow N_{<} = N_{>}$$

- Another approach: treat the problem as a regression

→ define a function  $f: \mathbb{R} \rightarrow \mathbb{R}$ , that 
$$\begin{cases} f(x) < 0 \Leftrightarrow x \in A \\ f(x) > 0 \Leftrightarrow x \in B \end{cases}$$

→ a classification can always be translated to regression

→ simplest: linear relation  $f(x) = ax + b$

→ probabilistic interpretation:  $p(x) = \frac{e^{f(x)}}{1 + e^{f(x)}} \in [0, 1], \quad p(0) = 0.5$

logistic map, probability of belonging to class B

→ loss function: 
$$L(a, b) = \sum_x (p(x) - I_{x \in B})^2, \quad I_{x \in B} = 1 \text{ if } x \in B, \text{ else } 0$$

- Another approach: tree

- define a function  $f: \mathbb{R} \rightarrow \mathbb{R}$

- a classification can always be seen as a regression

- simplest: linear relation

- probabilistic interpretation

logistic map, probability of belonging to class B

- loss function: 
$$L(a, b) = \sum_x (p(x) - I_{x \in B})^2, \quad I_{x \in B} = 1 \text{ if } x \in B, \text{ else } 0$$

```
# Python function minimization
```

```
import numpy, matplotlib
```

```
def P(p, x):
```

```
    a, x0 = p
```

```
    return np.exp(a*(x-x0))/(1+np.exp(a*(x-x0)))
```

```
def loss(p, A, B):
```

```
    loss = np.sum([P(p, a)**2 for a in A])
```

```
    loss += np.sum([(P(p, b)-1)**2 for b in B])
```

```
    return loss
```

```
p0 = [1, 1] # initial guess
```

```
params = (A, B) # values of a, b
```

```
res = minimize(loss, p0, args=params)
```

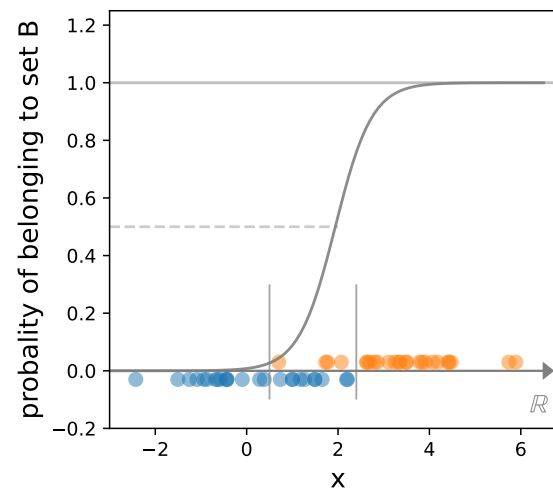
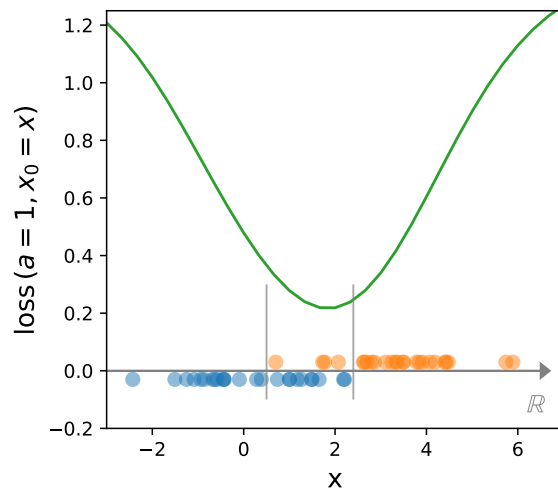
```
a, x0 = res.x
```

# One dimensional dataset

- Another approach: treat the problem as a regression

- ➔ logistic map, probability of belonging to class B

$$p(x) = \frac{e^{ax+b}}{1+e^{ax+b}}$$





# One dimensional dataset

## Statistical / information theory characterization

- take a sample where  $n_A \in A$ ,  $n_B \in B$ ,  $n = n_A + n_B$
- **entropy**:  $\sim$  log of how many ways we can order this set

$$S = \frac{1}{n} \ln \binom{n}{n_A} = \frac{1}{n} \ln \frac{n!}{n_A! n_B!}$$

- Stirling formula:  $\ln(n!) \approx n \ln n - n \dots$
- Shannon entropy:

$$S = \ln n - \frac{n_A}{n} \ln n_A - \frac{n_B}{n} \ln n_B = -n \sum_{i=1}^2 p_i \ln p_i$$

$$p_i = \frac{n_i}{n}$$

## Statistical / information theory characterization

- if we randomly choose A with probability  $q$ , and B with  $(1-q)$ , then what is the probability to get  $n_A \in A$ ,  $n_B \in B$  ?

→ choose the first  $n_A$  from A, the rest from B + ordering:  $P = \binom{n}{n_A} q^{n_A} (1-q)^{n_B}$

$$\ln P = -n p_A \ln p_A - n p_B \ln p_B + n_A \ln q + n_B \ln (1-q) = n \sum_{i=1}^2 p_i \ln \frac{q}{p_i}$$

- Kullback-Leibler (KL) divergence: probability to get the sample from a random choice with probability  $q$ :

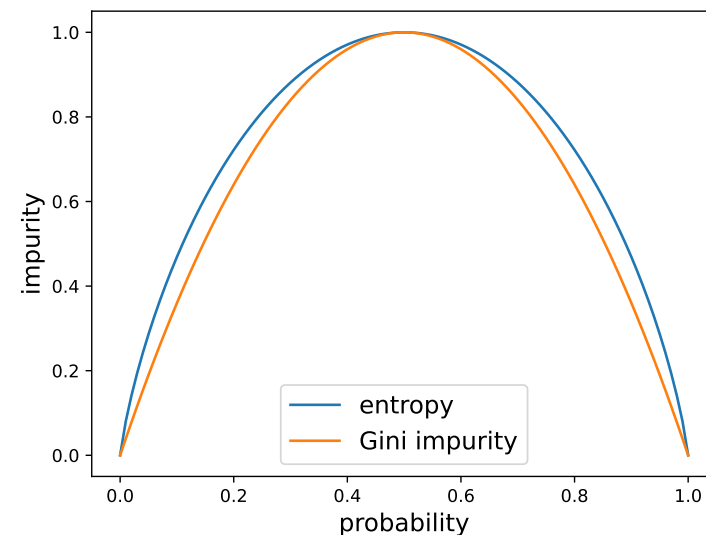
→ “distance” of distributions

$$D(p||q) = \frac{1}{n} \ln P = \sum_{i=1}^2 p_i \ln \frac{q}{p_i}$$

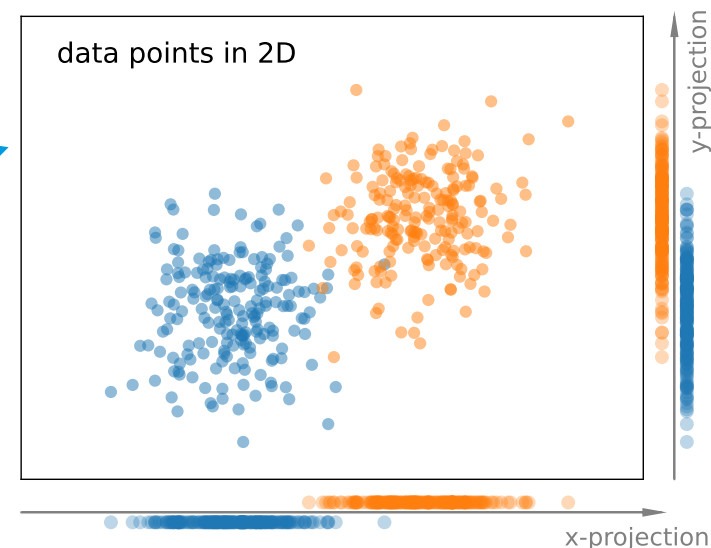
How successful is the separation?

- impurity of a sample:  $n_A \in A, n_B \in B \Rightarrow p = \frac{n_A}{n_A + n_B}$
- impurity measure:
  - entropy:  $I_H(p) = p \log(p) + (1-p) \log(1-p)$
  - Gini impurity:  $I_G(p) = p(1-p)$
- with separation total impurity change

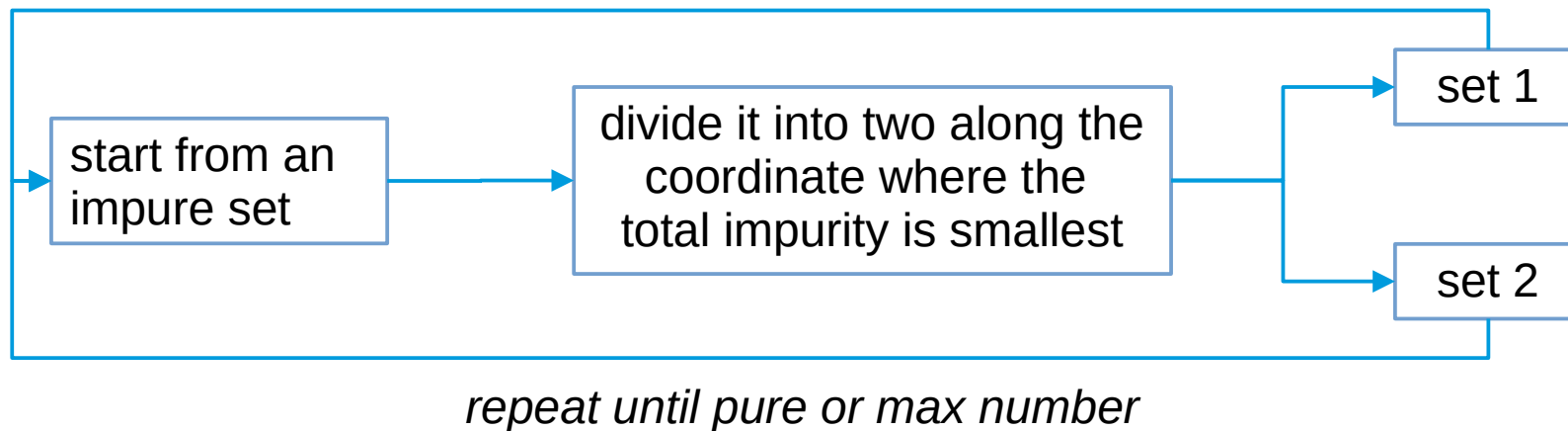
$$\Delta I = I_0 - \left( \frac{n_{\text{left}}}{n} I_{\text{left}} + \frac{n_{\text{right}}}{n} I_{\text{right}} \right)$$



- The **generalization to multiple dimensions** is not simple
  - in 1D the coordinates are unique
  - in multidimensional case we can optimize the coordination → features
  - simplest case: use original coordinates
  - usually the original coordinates are not equally useful  
(c.f. *image* → *x coord separates more*)
  - choose that coordinate that leads to the smallest total impurity!



- Decision tree (DT)



- always converges, but may result in fragmented divisions

- Decision tree (DT) example

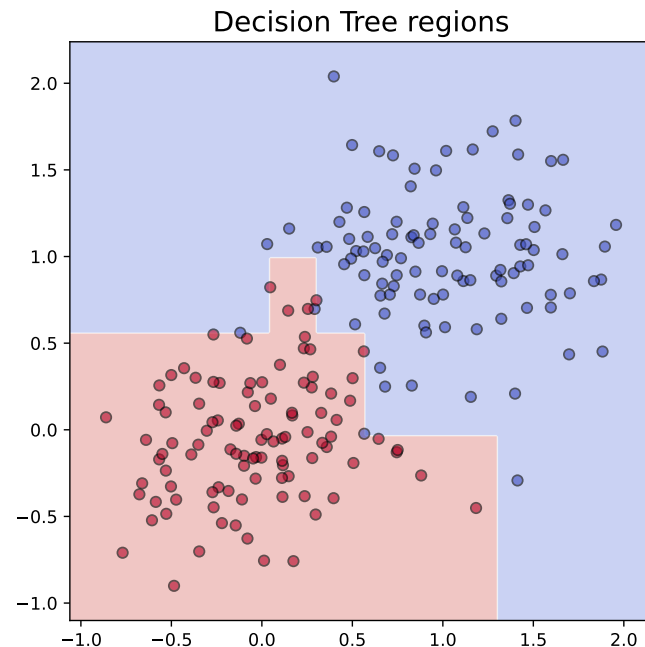
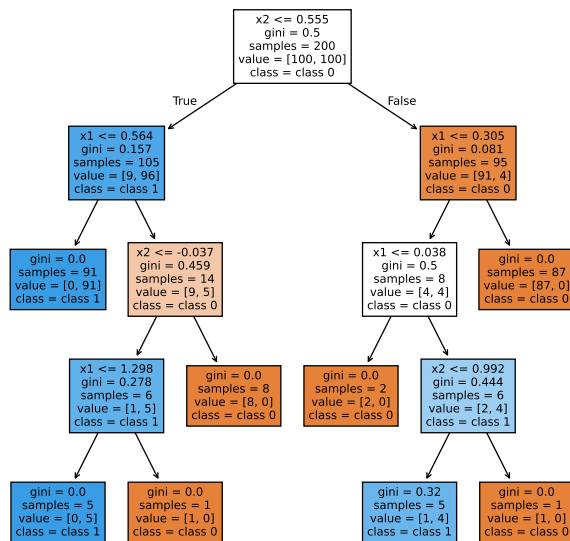
- two 2D sets

- pros:

- simple
  - interpretable

- cons:

- not robust enough  
( $\rightarrow$  *irregular shape*)



- Decision tree (DT) example

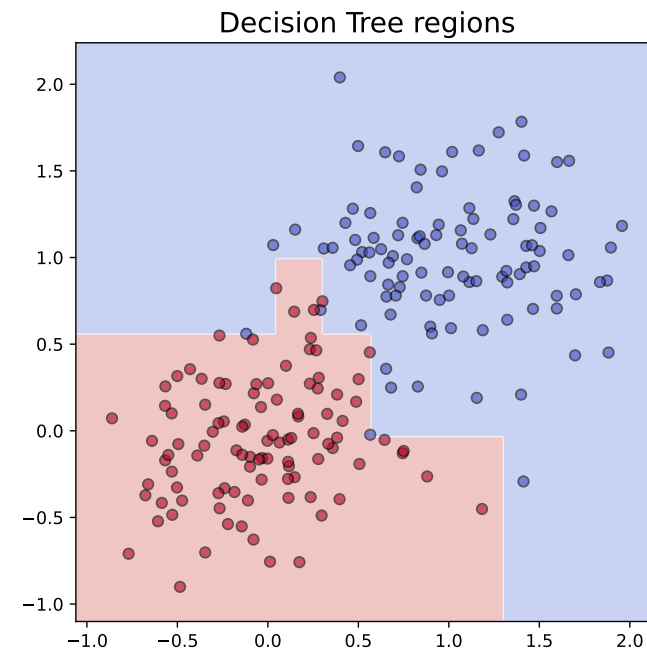
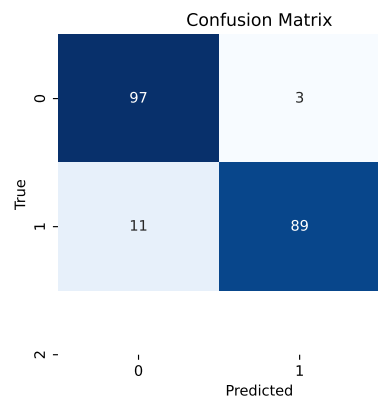
  - two 2D sets

- pros:

  - simple
  - interpretable

- cons:

  - not robust enough  
(→ *irregular shape*)  
*use fixed depth!*





- Decision tree (DT) technical solution in Python → [github](#)





- Decision tree → too specific for a sample
- Possible solution: use ensemble of methods
- ways of improvement
  - bagging (Bootstrap AGGregatING) → bagging, random forest
  - boosting → adaptive boosting, gradient boosting
  - other methods → extra trees, isolated forests, oblique trees



- Idea:
  - train several models in slightly different dataset
  - each dataset contains the same number of data, but some of them are missing, some appear multiple times
  - combine the results → regression: average, classification: majority vote

