ML - 25 Skriptum

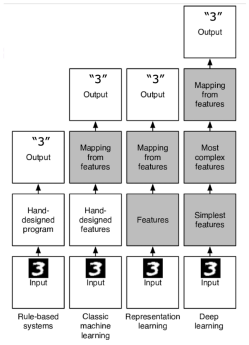
Philipp Schott - 12132552

June 24, 2025

Contents

1 What is Machine learning

There are three different sub-disciplines: Unspervised (data mining, cluster analysis), Supervised and Reinforcementlearning, we will fokus on the last two.



Learning Paradigms

vs. Rule-based systems
System that uses rules to make deductions or choices with two components:
Knowledge base: facts & rules (if → then style)
- Knowledge representation (language)

Inference engine: applies rules to deduce new facts
- Forward chaining: assert new facts
- Backward chaining: start with goal → determine which facts need to asserted

"A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E" (Tom Mitchell)

2 Data Preparation & Performance Evaluation

2.1 Data Preparation

General analytical (data science) process

- Identify problem / question
- Identify & capture the available data
- Prepare your data: clean & transform
- Analyse your data -> **actual machine learning**
- Create report with results, visualisation, insights
- Embed results into business / decision making
- Plan for better data capturing in the future

Generell Advice

- We decide between qualitative (nominal, ordinal) and quantitative (interval, ratio) Data
- **BUT:** Some ML algorithms rely on measuring the (numeric) distance between samples
- e.g. Scaling, Normalisation, Standardisation, One-Hot Encode (many impl. already have that)
- Other scalings:
  - Min-Max Scaling: $z_i = \frac{x_i - \min(X)}{\max(X) - \min(X)}$ (bounds input)
  - Z Scaling: $z_i = \frac{x_i - \mu}{\sigma}$
- **Missing values:** Bad for the algorithm? → deletion of row/column or substitution (e.g. NA-category, static mean etc. if else to small or difficult to identify)
- Attention: split first then clean to avoid **data leakage**

Data Augmentation

Def: Increases model invariance to irrelevant variations by transforming training data. Typical transformations include flipping, rotation, scaling, or color changes — so the model learns to focus on essential features and ignore superficial differences (e.g., in image classification). It also increases the apparent training set size, improving generalization and reducing overfitting.

Feature Selection

Def: The process of identifying and removing irrelevant or redundant input variables from the dataset by maximizing relevance to the target variable, in order to improve model efficiency, interpretability and perfomance.

An exhaustive search over all possible feature subsets is computationally infeasible for large numbers of features, as there are $\mathbf{2^n}$ possible combinations. Therefore, **heuristic methods** are used to identify useful subsets, reducing computational cost and improving learning performance.

Common methods include:

- **Filter:** evaluates features based on statistical properties (e.g., mutual information, correlation). *Simple and fast, but less accurate than wrappers:*
- **Wrapper:** searches for feature subsets by evaluating model performance (e.g. Tabu Search, Hill Climbing, Genetic Algorithms (metaheuristics), Branch and Bound, Best-first search).
  - **Greedy strategies:** (effiient and robust against overfitting)
    - **Forward Selection:** Iteratively starts with an empty feature set and adds the feature that most improves model performance (e.g. based on cross-validation error of models trained on the selected features) until a desired number of features is reached.
    - **Backward Elimination:** Starts with all features and iteratively removes the feature whose exclusion least harms model performance (i.e. that contributes the least), until a target subset size is reached.
- **Embedded approaches:** perform selection as part of model training (e.g., Lasso regression, which penalizes large coefficients; tree-based with built-in feature importance).

Feature Extraction

Def: The transformation of original input variables into a new set of features — often in a lower-dimensional space — that captures the most relevant structure in the data.

- **PCA (Principal Component Analysis):** reduces dimensionality by projecting data onto orthogonal components.
- **Autoencoders:** learn compressed representations of input data via neural networks.
- **Text features:** word embeddings (e.g., Word2Vec, BERT), bag-of-words, or TF-IDF vectors.
- **Images:** filters, edge detectors, and CNN feature maps.

2.2 Performance Evaluation

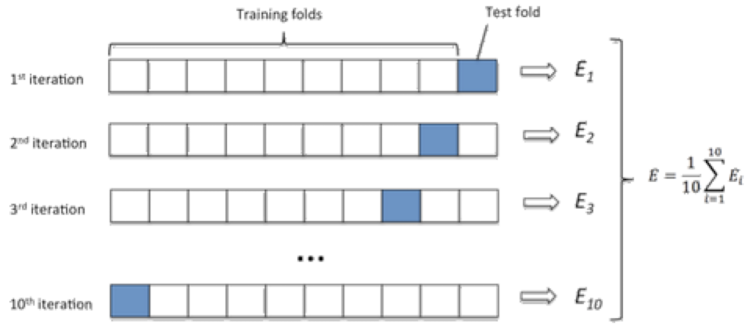
Effectiveness

- quality of classification
- **Table of Confusion** /Binary classification (also extendable to **confusion matrix**):

		Actual value (groundtruth)		
		true	false	
Prediction / Test outcome	true	True positive (TP)	False positive (FP, Type I error)	Precision $\frac{TP}{TP + FP}$
	false	False negative (FN, Type II error)	True negative (TN)	
		Recall Sensitivity $\frac{TP}{TP + FN}$	Specificity (True Negative Rate)	Accuracy $\frac{TP + TN}{\#\ samples}$

Actual / Predicted	Grey	Black	Red	Precision
Grey	5	3	0	0.625
Black	2	3	1	$\frac{3}{6} = 0.500$
Red	0	1	12	0.920
				0.740

- How will model perform on unseen data? ⇒ **Holdout method:** Training 80% vs. Test 20% (randomise before split & use **stratification** to equally train all classes but might not reflect the "real world" scenario)
- **k-fold Cross validation:** (also uses significance testing etc.)



  - vs. **Leave-p-out Cross validation:** uses $\binom{n}{p}$ of all combinations vs. **Bootstrapping:** sample of size N with replacement (arbituary times)
- **Other evaluation measures:**
  - **Micro average:** all classes together - not indicate issues with imbalanced classes (e.g. as above)
  - **Macro average:** e.g. Recall: $\frac{1}{|C|}\sum_{i=1}^{|C|}\frac{TP_i}{TP_i+FP_i}$ (with |C| = number of classes; so per class then average)
  - **Balanced accruacy:** $\frac{TPR+TNR}{2} = \frac{Recall+Specificity}{2} = \frac{\frac{TP}{TP+FN}+\frac{TN}{TN+FP}}{2}$ (important for imbalanced classes)
- **Cost of misclassification:**
  - Cost (loss) functions (matrix): Higher weight (e.g. by experts) to classes where errors are more severe
  - **Cost function:** Total Cost $= \sum_i \sum_j C(i,j) \cdot N(i,j)$ (elementwise product)
- **Model selection:** HP-tuning, alorithm, features etc. (test by e.g. 60-20-20 validation split or nested cross validation after training (randomness))
  ⇒ *Any classifier might outperform another on a specific training set*
- **Significance testing:** Null hypothesis – result of two classifiers are samples drawn from the same distribution with $\alpha = 0.05 - 0.01$
  - McNemar's test (based on χ^2-test: $\chi^2 = \sum_{i=1}^{n}\frac{(O_i - E_i - 1)^2}{E_i}$)
  - paired t-Test $t = \frac{\overline{X}_D\sqrt{n}}{s_D}$ assumes that the differences between paired observations are independent and normally distributed (e.g. two models on the same data splits)

Evaluating Regression Models

For evaluating numeric predictions (regression), standard performance metrics include:

- **Root Mean Squared Error (RMSE):** $\sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$ (e.g. MSE, MAE)
- **Root Relative Squared Error (RRSE):** $\sqrt{\frac{\sum_{i=1}^{n}(p_i - a_i)^2}{\sum_{i=1}^{n}(a_i - \bar{a})^2}} = \sqrt{\text{RSE}}$
- **Relative Absolute Error (RAE):** $\frac{\sum_{i=1}^{n}|p_i - a_i|}{\sum_{i=1}^{n}|a_i - \bar{a}|}$ (where p_i is the prediction, a_i is the true value)
- **R-squared (R^2):** proportion of variance explained by the model: $R^2 = 1 - \frac{\sum(y-\hat{y})^2}{\sum(y-\bar{y})^2}$
- **Pearson Correlation Coefficient:** $r = \frac{\sum_i (p_i - \bar{p})(a_i - \bar{a})}{\sqrt{\sum_i (p_i - \bar{p})^2 \sum_i (a_i - \bar{a})^2}} = \frac{S_{PA}}{\sqrt{S_P S_A}}$

Each metric captures different aspects of model performance (e.g., sensitivity to outliers, interpretability).

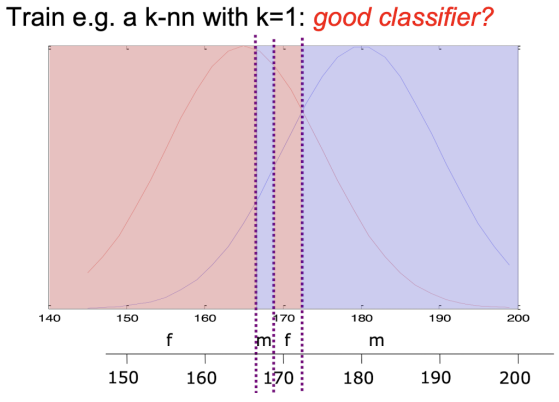
Efficiency

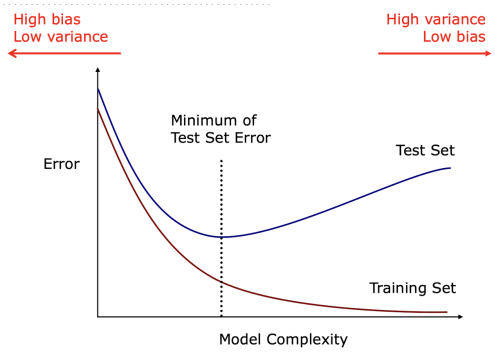
- computational efficiency (runtime, memory consumption) of a prediction model (always a Trade-off)
- Differentiate between efficiency on:
  - Training (learning) a model (usually a 1-time effort)
  - Classification (might require close-to real time speed / no time)
- becomes more relevant if model needs to be (re-)trained frequently

Overfitting & Generalisation

Definition:

- **Overfitting:** model is trained too specific to learning examples (i.e. to much variance learned)
- **Generalisation:** ability of model to perform well on the general problem (i.e. the real distribution that generated the training data)





Trade-off complexity vs. generalization

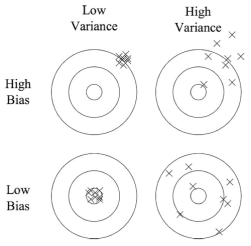
Bayes Optimal Classifier: Classification by taking the most likely output value for a given input (i.e. the highest probability -> Estimate probability densities based on samples (similar to what Naive Bayes does))

Bias & Variance

The actual error is influenced by **three parts**:

- **Irreducible error:** cannot be reduced regardless of algorithm used (e.g. introduced from the chosen framing of the problem or: caused by factors like unknown variables that influence the mapping of input to output, statistical noise, etc..)
- **Bias:**
  - errors from erroneous (simplifying) assumptions in learning algorithm
  - the algorithm's tendency to consistently learn the wrong thing, by **not taking into account all the information** in the data (∼ **"underfitting"**)
  - **High bias:**
    - more (simplifying) assumptions about the form of target function (e.g. linear models)
    - Fast to learn and easier to understand
    - Generally less flexible -> lower predictive performance on complex problems that fail to meet simplifying assumptions of algorithm's bias
    - May **underfit** - fail to capture important regularities
  - **Low bias:**
    - models are usually more complex but: less assumptions on target function
    - Represent the training set more accurately (Might also represent noise / **overfit**)
    - (e.g. Decision Trees, k-Nearest Neighbors, . . .)
- **Variance:**
  - error stems from sensitivity to small fluctuations in training set (e.g. 1-NN: "Models noise"; "instable"; "Measure for prediction consistency"; "memory capacity"; overfitting: 0-error)
  - the algorithm's tendency to learn random things irrespective of the real signal, by fitting highly flexible models that follow the error/noise in the data too closely (∼ **"overfitting"**)
  - **High variance:**
    - Different training sets lead to (very) different classifiers / decision boundaries
    - BUT: Classifier able to represent training set well (e.g. Decision Trees)

    Good variance: Algorithm good at discovering hidden underlying mapping between inputs and output, not specific instances



Bias-variance tradeoff:
- Accurately captures regularities in training data (low bias)
- Generalises well to unseen data (low variance)
- Typically impossible to do both simultaneously!
  → don't know true mapping function and its effects

3 Simple Networks

3.1 (single-layer) Perceptron

- Linear combination of inputs (contionous X), using weights W: $a = \sum_{i=1}^{n} w_i x_i$

- Pass through threshold activation function (Heaviside step function/ **activation function!**) with threshold θ (often: $\theta = 0$) where:
$$y = f(x) = \begin{cases} 1 & \text{if } a \geq \theta \\ 0 & \text{if } a < \theta \end{cases}$$
- **Training the (basic) model**: learning the weights from labelled samples (label: y) with initialise weights:
  - Repeat:
    - Present training sample $\mathbf{x}$
    - Predict sample label: $y' = f(x)$
    - Prediction correct? Compare y and y'
      - if $y' \neq y \Rightarrow$ Compute new weights w' as $w' = w + \alpha(y - y')\mathbf{x}$
  - Until prediction correct ($y' = y$) for all samples
- **Extensions**:
  - Can be extended to multi-class problems using multiple decision boundaries, but only works for linearly separable data (fails on problems like XOR)
  - different **stopping criteria** for non-ls: iteration or no more improvement
  - **pocket algorithm:** keeps the best solution found so far (e.g. by accuracy) and returns it
  - **Bias** can be introduced: $a = \sum_{i=1}^{n} w_i x_i + \mathbf{b}$
  - **More generel:**
    - In the **dual form**, the prediction is based on inner products between training examples and the input: $f(x) = \text{sign}\left(\sum_{i=1}^{N} \alpha_i y_i \langle x_i, x \rangle\right)$ (where α_i are coefficients updated during training)
    - Using a **kernel function** $K(x_i, x) = \langle \phi(x_i), \phi(x) \rangle$, we can generalize to the **kernel perceptron**: $f(x) = \text{sign}\left(\sum_{i=1}^{N} \alpha_i y_i K(x_i, x)\right)$ (allows the perceptron to learn **non-linear decision boundaries** without explicitly mapping to a higher-dimensional feature space)

3.2 Linear Regression

The **goal** is to learn a function that maps a feature vector $x \in \mathbb{R}^n$ to a **continuous target value** $y \in \mathbb{R}$ using a linear model $\hat{y} = w_0 + w^\top x$ where we want to find the **best** parameters (w_0, w) by minimizing the prediction error over a training set.

The most commonly used error function is the **Residual Sum of Squares:** $\text{RSS}(w_0, w) = \sum_{i=1}^{m} \left(y_i - (w_0 + w^\top x_i)\right)^2$

To minimize the residual sum of squares (RSS), two main approaches are commonly used:

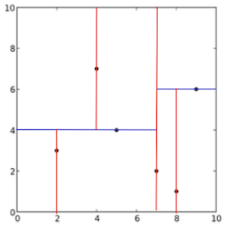
- **Gradient Descent (Iterative):** $w_j := w_j - \alpha \cdot \frac{\partial}{\partial w_j} \text{RSS}(w_0, w)$ (repeated until convergence)
- **Normal Equation:** $w = (X^\top X)^{-1} X^\top y$ (less efficient on large dataset)

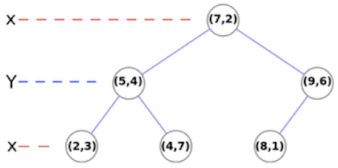
3.3 k-NN

- Short for k-nearest neighbour O(Nd) with two important **hyperparameters (HP)**: k and distance-funktion
- **BUT:** good values for k vary a lot by data caused by sensetivity to lokal noise
- Examples for distance funktion:
  - euklidean distance/ L2

- Minkowski for n=1/ L1/ Manhatten (outperformce L2 most of the time)
- exotic ones like: Linfinity (just y-distance) or Cosine

- is a **Lazy Learner**: no model built beforehand -> computation at classification step (Opposite is called "eager learning")
- **Extansions:**
  - majority voting vs. **weighted approaches**: ($(\frac{1}{x})$: rank vs. distance weighted) -> Are all equally important?
  - **K-d-Tree:** yields different predictions but doesn't always pay of for small sets (bad on high d; **weak** learner: slightly better than random guessing)





- **k-NN regression:** predicts the output of the k nearest neighbors, i.e., $\hat{y} = \frac{1}{k} \sum_{i=1}^{k} y_i$.

3.4 Bayesian Networks

Generating Rules

Covering algorithms ⇒ Problems: overlapping rules; default rule required

- Converting a decision tree into a rule set: easy but rule set gets overly complex (if → and etc.)
- Solution: generate rule set direktly
  - generates rules for each class separately (easier to understand, especially in multiclass settings), starting with a very general rule (e.g. "if true → class X") which covers all instances
  - makes the rule more specific by adding one condition at a time, always selecting the condition that maximizes the rule's accuracy = correct / total, until accuracy reaches 1 or no further improvement is possible (e.g. "if ... and ... → then ...")
  - after one rule is complete, removes the covered instances and repeats the process for the remaining instances of the same class
  - after all rules for one class are generated, the algorithm moves to the next class

PRSIM (concrete example of a covering algorithm in pseudo code; same as above but more generell)

For each class C:

- Initialize E to the instance set.
- While E contains instances in class C:
  - Create a rule R with empty left-hand side predicting class C.
  - While R is not perfect (or no attributes left to use):
    - For each attribute A not mentioned in R, and each value v:
      - Consider adding condition $A = v$ to the left-hand side of R.

· Select A and v to **maximize** accuracy p/t. (break ties by choosing largest p).

* Add $A = v$ to R.

– Remove instances covered by R from E.

3.4.1 Naive Bayes

- Opposite of (1R) rule learning: use all attributes at once
- Two assumptions:
  - attributes are equally important
  - attributes are conditionally independent given the class: $P(A \wedge B \mid C) = P(A \mid C) \cdot P(B \mid C)$

Computation:

- Let C = class label of prediction and $E = (E_1, E_2, \ldots, E_n)$ = observed attributes (the evidence)
- **We want to compute:** $P(C \mid E) = \frac{P(E|C) \cdot P(C)}{P(E)}$ where $P(C) = \frac{\text{count}(C)}{\text{total}}$
- Since $P(E)$ is the same for all classes (constant), we only need to calculate: $P(E \mid C) \cdot P(C)$
- With the independence assumption: $P(E \mid C) = \prod_i P(E_i \mid C)$ ("Yes" for this evidence / total)
- Thus, for classification, we select the class that maximizes: $P(C) \cdot \prod_i P(E_i \mid C)$

What if $P(E_i \mid C) = 0$? (NA in training not included, in classification obtained by probability)
$\Rightarrow$ **Generalized Laplace correction:** $P(E_i \mid C) = \frac{\text{count}(E_i, C) + \mu \cdot p_i}{\text{count}(C) + \mu}$ (or simply "add 1" ($\mu = k$))

For numeric attributes: we assume there density function: $P(E_i = x \mid C) = \frac{1}{\sqrt{2\pi}\sigma_C} \exp\left(-\frac{(x-\mu_C)^2}{2\sigma_C^2}\right)$
with $\mu_C = \frac{1}{N_C} \sum_{i=1}^{N_C} x_i$ and $\sigma_C = \sqrt{\frac{1}{N_C - 1} \sum_{i=1}^{N_C} (x_i - \mu_C)^2}$ and $P\left(x - \frac{\varepsilon}{2} \leq X \leq x + \frac{\varepsilon}{2}\right) \approx \varepsilon \cdot f(x)$

Notes on Naive Bayes:

- may not produce accurate probabilities, but correct class ranking is sufficient for classification
- adding redundant or highly correlated attributes can harm performance

3.4.2 Learning Bayesian Networks

Def: BN is probabilistic graphical models that captures dependencies via structure of a directed acyclic graph (DAG) where Nodes = 'random variables' and Edges = 'dependencies' and each node has a conditional probability table (CPT) that specifies the probability of the variable given its parents: $P(\text{Variable} \mid \text{Parents})$

Where the joint probability distribution is: $P(X_1, \ldots, X_n) = \prod_{i=1}^{n} P(X_i \mid \text{Parents}(X_i))$ ("Chain rule")

Learning a Bayesian Network: Two tasks:

- **Structure learning:** learn the graph structure (NP-complete, if not given)
  - trade-off between fit to data and model complexity (maximize: $\log P(D \mid M) - \alpha \cdot \#M$)
  - Initialize a BN and do **Structure search** to explore the space of possible networks by generating neighborhood solutions (e.g. Tabu Search, Simulated Annealing etc.)
- **Parameter learning:** estimate conditional probabilities (CPTs) from data, usually by counting and applying smoothing ("easy" e.g. Laplace, like in Naive Bayes)
  - **Goodness of fit:** product over all data points and all variables
  - **Log-likelihood:** sum instead of product (log of goodness of fit)

Inference: How do we compute probabilities from a Bayesian Network once it is built?

- **Maximum a posteriori probability** (MAP): $\arg\max_q \; P(Q = q \mid E = e)$
  - is the most likely assignment for the query variables given the evidence and solves the general inference task by maximizing the posterior probability.
- **Inference via marginalization:** summing out variables that you are not interested in, to get a distribution over the variables you care about (determined by the BN structure)
  - We want to compute: $P(Q \mid E) = \frac{P(Q,E)}{P(E)}$
  - with $P(Q, E) = \sum_H \prod_i P(X_i \mid \text{Parents}(X_i))$ and $P(E) = \sum_Q \sum_H \prod_i P(X_i \mid \text{Parents}(X_i))$
  - **Enumeration:** sum over all possible combinations of hidden variables (simple but computationally expensive)
  - **Variable Elimination:** smarter summing by eliminating hidden variables one by one and combining intermediate factors to avoid redundant computations

D (dependency)-Separation: Tells whether two variables are conditionally independent given a set of observed variables. A path is **blocked** (i.e., no information flow) if any of the following conditions apply:

- **Chain:** $A \to B \to C$ or $A \leftarrow B \leftarrow C$ — the path is blocked if B is observed.
- **Fork:** $A \leftarrow B \to C$ — the path is blocked if B is observed.
- **Collider:** $A \to B \leftarrow C$ — the path is blocked if B is not observed, and no descendant of B is observed. Observing B or its descendants opens the path.

If **all** paths between two nodes are blocked, they are conditionally independent (d-separated) given the observed variables.

4 From Decision Trees to Random Forest

4.1 Decision Tree

- **Simplest versions**: 1R (One Rule; Decision Stump): just one (root) node or: ZeroR, Zero Rule: Always return the majority class (both build baseline)
- **Training**: (Rules by experts or) splits data consecutively into (two or more) sub-spaces until stopping criterion (e.g (number of unique values) – 1)
- **Classification**: traverse through tree from root node (by majority vote/ binning) - can also do regression by average in the leaf nodes
- some difficulties e.g. French, Italian, Thai vs. French, others (binary vs. n-ary)
- **Regression Trees:** Tree-based models that recursively split the feature space and predict a continuous value (e.g., mean of training samples) in each leaf node which results in a piecewise constant function. Splits are chosen to minimize variance or squared error within nodes.
- **Model Trees:** An extension of regression trees where each leaf contains a linear regression model instead of a constant value.

4.1.1 Popular measures to compute best split:

- **Generell:** evaluate each attribute and possible split (numerical vs. categorical) then: chooses best split
- **Error rate**:
  - Absolute error rate: number of classification errors for each split (vs. relative error: in relation to total number of samples (0..1); (usage not further explained))

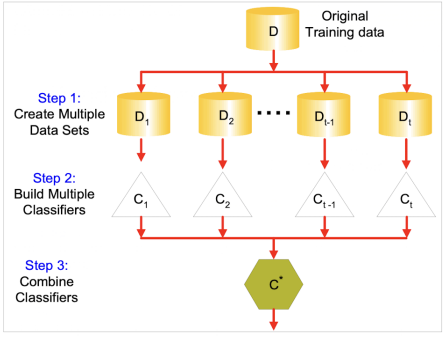
- **Information Gain** (best):
  - measure (lokal) for the "impurity"/ uncertainty of a set (High Entropy -> bad for prediction)
  - **Entropy:** $H(X) = \mathbb{E}(I(X)) = \sum_{i=1}^{n} p(x_i) I(x_i) = -\sum_{i=1}^{n} p(x_i) \log_2 p(x_i)$
  - (with I(X): information content of X; for H(X): Only relative frequencies matter!; if equal H(X)=1; $\log_2$ for yes/no)
  - **Information Gain:** $IG(X_1, \ldots, X_m) = H(X) - \sum_{j=1}^{m} p(x_j) H(X_j)$ (want to maximize)
  - **Information Gain Ratio:** $V(X) = \sum_{i=1}^{N} \frac{|T_i|}{|T|} \cdot \log\left(\frac{|T_i|}{|T|}\right)$
  - (IG favours features with many possible outcomes -> puts this into perspective)
  - "Normalises" Information gain V: $R(X) = \frac{G(X)}{V(X)}$
- **Gini impurity (Gini index)**:
  - Inequality among values of a distribution (Value range: 0 (only one class) to 1 (total inequality))
  - $I_G(p) = \sum_{i=1}^{|C|} p_i(1 - p_i) = \sum_{i=1}^{|C|} (p_i - p_i^2) = \sum_{i=1}^{|C|} p_i - \sum_{i=1}^{|C|} p_i^2 = 1 - \sum_{i=1}^{|C|} p_i^2$

4.1.2 (Pre-)pruning:

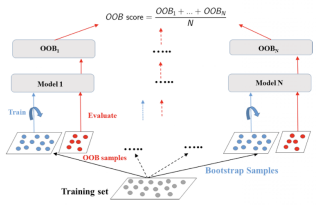
- To overcome overfitting:
- **Prepruning:** Stop splitting a node (e.g. max depth, min IG/Entropy etc. -> HP)
- **Pruning:** 'Cut back' complicated trees (more effort but mostly better)
  - Least contributing nodes are removed - sometimes remodelled (or selection out of candidates)
  - **Simple bottom-up approach:** reduced error pruning
    - remove sub-tree and eplace it with the majority class then evaluate performance → keep tree if effectiveness is not decreased (too much) and repeat
  - **Cost complexity pruning:** (Bottom up) generates a list of canidate trees and takes best (vs. Top down: evaluate relevance of node/subtree)

4.2 Random Forest

- Combination of Decision Tree and Bootstrapping concepts (robust; less overfitting)
- For each tree: use bootstrap sample → grow → repeat tens to hundreds times → aggregate predictions by majority voting (bagging)
- **Spacial evaluation:** out-of-bag (OOB) error / estimate (OOB): holdout of bootstrap sample



1. For each training sample x_i
   1. Find trees that were not using x_i in the tree's bootstrap sample
   2. Take majority vote prediction on those trees



5 Neural Networks / MLP

5.1 Forward pass

- **Multi-Layer Perceptron:** (feed-forward neural networks/ fully connected (directed) graph)

$$h^{(0)} = x$$

For all layers $l = 1, 2, \ldots, L$:

$$z^{(l)} = W^{(l)} h^{(l-1)} + b^{(l)}$$
$$h^{(l)} = \sigma^{(l)}(z^{(l)}) \quad \text{(different for l=L)}$$

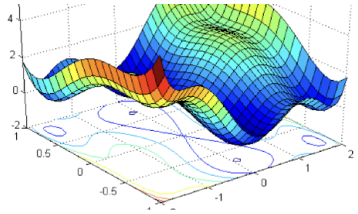
where $\mathbb{R}^{n_{l-1}} \xrightarrow{W^{(l)}} \mathbb{R}^{n_l}$ (with rules of thumb: $n_l \approx (0.7 - 0.9) \cdot n_{l-1}$; $n_l < 2 \cdot n_{l-1}$; $n_l \gtrsim \log_2(\#\text{classes})$)

The final output is $\hat{y} = h^{(L)}$.

- **Activation function:** (for piece-wise combination of decision boundaries)
  - **non-linear:**
    - sigmoid: $\sigma(x) = \frac{1}{1+e^{-x}}$ with $\sigma'(x) = \sigma(x) \cdot (1 - \sigma(x))$ (small slope → wg)
    - $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ with $\tanh'(x) = 1 - \tanh^2(x)$ (wg → LReLU)
    - $\text{ReLU}(x) = \max(0, x)$ (avoids vanishing gradient)
  - **probabilistic:** (maps the output scores (logits) into a probability distribution over C classes)
    - Softmax: $\text{softmax}(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{C} e^{z_j}}$ with $\frac{\partial \mathcal{L}}{\partial z_i} = \text{softmax}(z)_i - y_i$ (simplified)
    - **One-vs-All (OvA):** Train C independent binary classifiers (e.g. logistic regression), each distinguishing one class from the rest (vs. all others). Final prediction is the class whose classifier gives the highest score. Does not jointly normalize over all classes (unlike softmax).

5.2 Training

Main Idea: Start at a random solution. Compute the gradients of the loss function with respect to the network's weights and biases by recursively applying the chain rule from the output layer back to the input layer, in order to determine how each parameter influences the loss and to update both weights and biases in the direction that minimizes the error.



» x-axis: weight w_1
» y-axis: bias b
» z-axis: ?
» The error of the model with given w_1/b

5.2.1 Actual Training

- **Compute the Loss:** how well the neural network's predictions $\hat{\mathbf{y}}$ match the true target values $\mathbf{y}$ for a batch of size N (Batch GD)? (for N=1: SGD; N-random: Mini-Batch GD)
  - **MSE:** (reg) $\mathcal{L}(\hat{y}, y) = \frac{1}{N} \sum_{i=1}^{N} (\hat{y}_i - y_i)^2$ with $\frac{\partial \mathcal{L}}{\partial \hat{y}_i} = \frac{2}{N}(\hat{y}_i - y_i)$
  - **MAE:** (reg) $\mathcal{L}(\hat{y}, y) = \frac{1}{N} \sum_{i=1}^{N} |\hat{y}_i - y_i|$ with $\frac{\partial \mathcal{L}}{\partial \hat{y}_i} = \frac{1}{N} \cdot \text{sign}(\hat{y}_i - y_i)$
  - **Binary Cross-Entropy:** $\mathcal{L}(\hat{y}, y) = -\frac{1}{N} \sum_{i=1}^{N} [y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)]$ with $\frac{\partial \mathcal{L}}{\partial \hat{y}_i} = \frac{1}{N} \left(-\frac{y_i}{\hat{y}_i} + \frac{1-y_i}{1-\hat{y}_i} \right)$

- **Categorical Cross-Entropy:** $\mathcal{L}(\hat{y}, y) = -\sum_{i=1}^{C} y_i \log(\hat{y}_i)$
  - with $\frac{\partial \mathcal{L}}{\partial \hat{y}_i} = -\frac{y_i}{\hat{y}_i}$
  - Note: In practice, for numerical stability, the combination of Binary Cross-Entropy with Sigmoid activation often simplifies the gradient to $\frac{\partial \mathcal{L}}{\partial \hat{y}_i} = \hat{y}_i - y_i$
- often uses $+\alpha\|W\|^2$ as **regularization**/ penalty
  - with Ridge (L2): $\|W\|_2 = \sqrt{\sum_{j=1}^{p} W_j^2}$
  - with Lasso (L1): $\|W\|_1 = \sum_{j=1}^{p} |W_j|$
  - High α: reduces variance (risk of underfitting), Low α: reduces bias (risk of overfitting)

- **Backpropagation:**
  - **1. Compute error at output layer:**

$$\delta^{(L)} = \frac{\partial \mathcal{L}}{\partial h^{(L)}} \circ {\sigma^{(L)}}'(z^{(L)})$$

Using chain rule explicitly:

$$\frac{\partial \mathcal{L}}{\partial W^{(L)}} = \frac{\partial \mathcal{L}}{\partial h^{(L)}} \cdot \frac{\partial h^{(L)}}{\partial z^{(L)}} \cdot \frac{\partial z^{(L)}}{\partial W^{(L)}}$$

where:

$$\frac{\partial h^{(L)}}{\partial z^{(L)}} = {\sigma^{(L)}}'(z^{(L)}), \quad \frac{\partial z^{(L)}}{\partial W^{(L)}} = h^{(L-1)}$$

so that:

$$\frac{\partial \mathcal{L}}{\partial W^{(L)}} = \delta^{(L)} \cdot (h^{(L-1)})^T$$

(to prevent vanishing or blow up: $W_{ij} \sim \mathcal{N}\left(0, \frac{2}{n_{\text{in}}+n_{\text{out}}}\right)$ (Xavier (Glorot) for tanh and sigmoid) or $W_{ij} \sim \mathcal{N}\left(0, \frac{2}{n_{\text{in}}}\right)$ (for ReLU)

  - **2. Compute error for hidden layers:**
For all $l = L-1, L-2, \ldots, 1$:

$$\delta^{(l)} = \left(W^{(l+1)\,T} \delta^{(l+1)}\right) \circ {\sigma^{(l)}}'(z^{(l)})$$

Using chain rule:

$$\frac{\partial \mathcal{L}}{\partial W^{(l)}} = \frac{\partial \mathcal{L}}{\partial h^{(l)}} \cdot \frac{\partial h^{(l)}}{\partial z^{(l)}} \cdot \frac{\partial z^{(l)}}{\partial W^{(l)}}$$

with:

$$\frac{\partial h^{(l)}}{\partial z^{(l)}} = {\sigma^{(l)}}'(z^{(l)}), \quad \frac{\partial z^{(l)}}{\partial W^{(l)}} = h^{(l-1)}$$

so again:

$$\frac{\partial \mathcal{L}}{\partial W^{(l)}} = \delta^{(l)} \cdot (h^{(l-1)})^T$$

  - **3. Compute gradients:**

$$\frac{\partial \mathcal{L}}{\partial W^{(l)}} = \frac{1}{N} \sum_{i=1}^{N} \delta_i^{(l)} \cdot (h_i^{(l-1)})^T \quad (maybe: +\alpha W^{(l)})$$

$$\frac{\partial \mathcal{L}}{\partial b^{(l)}} = \frac{1}{N} \sum_{i=1}^{N} \delta_i^{(l)}$$

– **4. Parameter update:**

$$W^{(l)} \leftarrow W^{(l)} - \eta \frac{\partial \mathcal{L}}{\partial W^{(l)}}$$
$$b^{(l)} \leftarrow b^{(l)} - \eta \frac{\partial \mathcal{L}}{\partial b^{(l)}}$$

(where Time decay: $\eta \doteq \alpha_t = \frac{\alpha_0}{t}$ or Exponential decay: $\eta \doteq \alpha_t = \alpha_0 \cdot \exp(-t \cdot k)$ can be used)

- **Repeated** until loss is minimised / other stopping criterion

5.2.2 Other approaches of Gradient Descent:

- **Stochastic Gradient Descent (SGD) with Momentum:**
  - $\mathbf{v} = \beta \mathbf{v} - \alpha \frac{\partial \mathcal{L}}{\partial \theta}$ where $\theta_{t+1} = \theta_t - \mathbf{v}$
  - **Intuition:** Ball rolling downhill (along the cost function)
  - Issues: Often results in oscillations and instability in high-curvature regions
- **Adaptive Moment Estimation (Adam):**
  - Optimization methods that adapt the learning rate for each parameter
  - as above + second moment estimate: exponential moving average over past gradient magnitudes
  - **Intuition:** counter notoriously small gradients by upscaling, and large gradients by downscaling – Separately for each weight

6 Deep Learning

Definition: NN with with at least two hidden layers (2 ≤ L-1) ⇒ DNN
→ (with enough units DNN can approximate any continuous function arbitrarily well)

6.1 Convolutional neural network (CNN)

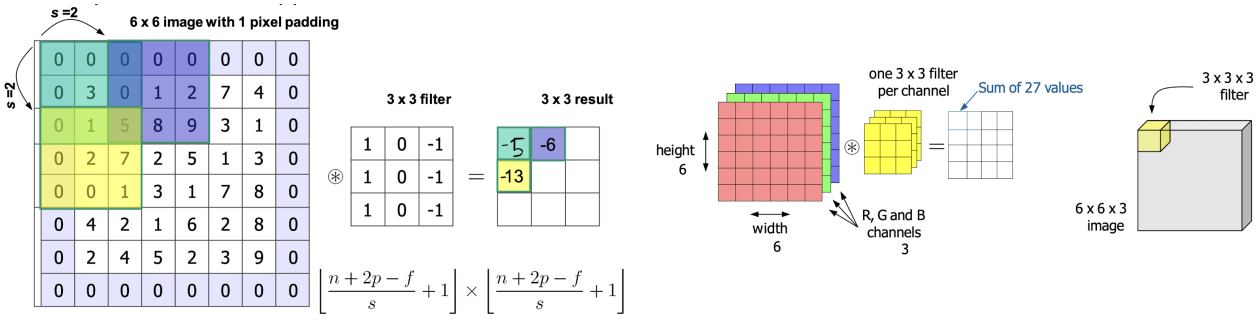
Combines **three types of layers**:

- **Convolutional layer:** performs 2D convolution of 2D input with multiple learned 2D kernels/ filters to extract spatial features
- **Subsampling layer:** reduces spatial dimensions by aggregating values within local regions
- **Fully-connected layer:** computes weighted sums of its input with learned coefficients (MLP)

6.1.1 Convolutions in detail

Instead of $f(x;\theta) = W^T x$ as above we are using: (vs. Image processing were W is defined:)

- $f(x;\theta) = x * W = \sum_{u=0}^{k-1} \sum_{v=0}^{k-1} x(i+u, j+v) \cdot W(u,v)$
- where $W \in \mathbb{R}^{f \times f}$ is the filter/ kernel so W: $\mathbb{R}^{n \times n} \rightarrow \mathbb{R}^{\lfloor \frac{n+2p-f}{s}+1 \rfloor \times \lfloor \frac{n+2p-f}{s}+1 \rfloor}$
- where p is the padding (adding zeros) with $p = \frac{f-1}{2}$ and f odd by convention and s the stride/ step size



Interpretation of the learned rules:

Mostly hard because layers show intermediate result **vs. Fully-connected layer:**

- as output is a **local feature extraction** by convolved sum of weights (cause filter is applied across the entire input image by sliding (shifting) the filter over different regions) ⇒ **order (spatial arrangement)** does not matter!
- But we want the machine to learn the same concepts that we know **Solution:**
  - Connect each neuron to small part of the input image only so share weights between neurons applied to different parts of the input image (e.g. three 3x3 have less weights then 7x7 or different sizes)
- **BUT:** to provide local translation invariance (still needed to learn)
  ⇒ **Subsampling** (e.g. max pooling (max across a region), average-, or other pooling methods)

Dropout: (e.g. 10-50 %) prevents co-adaptation / overfitting of feature detectors (they cannot rely on all inputs and neighbours being present)

Batch Normalisation: stabilizes and accelerates training by normalizing layer activations to *zero mean and unit variance* within each mini-batch, followed by a learnable scaling and shifting (⇒ reduces sensitivity to weight initialization, mitigates exploding or vanishing gradients and adds a slight regularization effect by introducing noise)

6.2 Transfer & Ensemble Learning

Transfer Learning

Main idea: Pre-trained models are available → Can transfer learned representations from a related task → Same domain, different task – Different domain, same task

- **"Off-the-shelf" Transfer Learning:** Network trained on different (but similar) task → Use output of one/more layers as generic feature detectors → train shallow model on these features → Fine-tune network with (small set of) labels for target domain
- or **freeze:** not update - target task labels are scarce, want to avoid overfitting or mix up (e.g. less fine-tuning for earlier layers as higher layers are more task specific; less transferable)

Ensemble Learning

Main idea: combine several classifier predictions (no model selection) – Goal: improved performance (e.g. Random Forest is Homogenous (vs. Heterogeneous))

Different voting types:

- **Majority voting:**
  - estimated accuracy computed by binomial model (mostly increased)

  - in extreme cases: can decrease correctness compared to best single one cause no identical accuracy
- ⇒ **Weighted majority voting:**
  - $\sum_{i=1}^{L} b_i d_{i,k} = \max_{j=1}^{c} \sum_{i=1}^{L} b_i d_{i,j}$
  - The predicted class is selected by summing the weighted votes for each class and choosing the class with the highest total weighted vote. Each classifier contributes to the vote according to its weight b_i, which reflects its reliability or accuracy.

Important: Types of classifier outputs: More informative outputs → more flexible ensemble methods (Type 1: label < Type 2: rank < Type 3: probability)

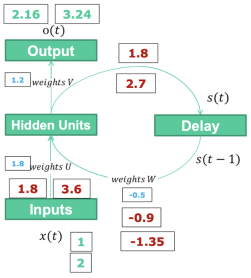
- **Bagging (Bootstrap AGGregatING):** builds ensembles by training classifiers on different bootstrap samples of the data to reduce variance by averaging predictions from independent models. It works best for unstable, high-variance classifiers and aims to create independent models (classifiers are learned in parallel)
- **Boosting:** combines weak learners sequentially, where each learner focuses on the errors of its predecessors, reducing bias and improving overall performance (e.g. **AdaBoost:**)
  - initializes all training samples with uniform weight $w_i = \frac{1}{N}$ and iteratively increases the weights of misclassified samples to focus on harder cases
  - final classifier after T iterations: $H(x) = \text{sign}\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right)$
  - vs. **Gradient Boosting:**
    - starts with a simple initial prediction $F_0(x)$ (e.g. 'zero-rule' e.g. mean for regression/ log odds), then iteratively fits weak learners to the residuals of the current model and adds them to improve predictions
    - in each iteration, compute residuals $r_i = y_i - F_{t-1}(x_i)$ ((pseudo) residual)
    - fit weak learner $h_t(x)$ to residuals, add correction: $F_t(x) = F_{t-1}(x) + \rho_t h_t(x)$
    - residuals correspond to negative gradients of the loss function $\mathcal{L}$; hence: gradient boosting
    - final model after T iterations: $F(x) = \sum_{t=1}^{T} \rho_t h_t(x)$

6.3 Challenges

Generell problems of AI

- Minimal perturbation t of input x leads to misclassification ⇒ Greedy search for decision boundary by changing pixels (etc.)
- Model transparency vs. algorithmic transparency (Ex-ante, Intrinsic, Post-hoc etc.)
- Data Privacy: quasi-identifiers (QI) → Input & Output perturbation etc.
- **White box attack:**
  - uses loss function to maximize model prediction error
  - Compute $g = \nabla_x \mathcal{L}(f(x), y)$ and applies $x_{\text{adv}} = x + \epsilon \cdot \text{sign}(g)$
- **Backdoor Attacks:** attacks embed hidden patterns (triggers) into the training data so that the model behaves normally on clean inputs but misclassifies inputs containing the trigger

6.4 Recurrent Networks



Definition: process sequential data by passing information through feedback connections, allowing them to maintain a state (memory) of previous inputs and handle variable-length input sequences (e.g. sequential / time series data)
Other Architectures: **Auto-Encoder:** Learn a lower dimentional encoding for a data set

Architecture

- **Forward Pass:**
  - Compute network input at time t: $a_h(t) = Ux(t) + Ws(t-1)$
    (combines current input $x(t)$ with the previous hidden state $s(t-1)$ (this introduces the recurrence))
  - Activation of input at time t: $s(t) = f_h(a_h(t))$
  - Compute pre-activation of output at time t: $A_0(t) = Vs(t)$
    (where V is the output weight matrix that maps the hidden state into the output space to obtain output pre-activation)
  - final output at time t: $O(t) = f_0(a_0(t))$
- for **Backpropagation:**
  - Cyclic graph (can be more complex) unfold into an acyclic one (BPT(hrough)T(ime))
  - as consists of long sequences ⇒ exploding/vanishing gradients → gradient clipping (e.g. norm of max 1) or:

LSTM (Long Short-Term Memory) → units / blocks (or: GRU: Gated Recurrent Units)

- Long-term memory (LTM): scalar, vector updated element-wise (not transformed by weights directly)
- Short-term memory (STM): / hidden state: vector passed to next step and output, multiplied by weight
- **Steps:** (different weights for each gate)
  - **Forget gate:** computes how much of LTM to keep.
  - **Input gate:** decides how to update the memory (Computes new potential LTM - multiplies by percentage of potential to keep (control))
  - **Output gate:** decides on new STM and output.

7 Automated Machine Learning (AutoML)

Motivation: Manual selection and configuration of machine learning algorithms is time-consuming and inflexible, leading to the algorithm selection and configuration problem.

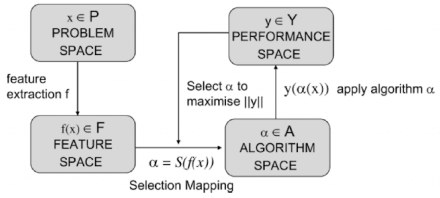
7.1 Metalearning

Def: means learning about learning by accumulating experience on the performance of machine learning algorithms across multiple applications, enabling dynamic model selection and method combination. vs. **Learning:** accumulates experience on a specific learning task.

NFL Theorem:

- "...for any algorithm, any elevated performance over one class of problems is offset by performance over another class."
- "any two algorithms are equivalent when their performance is averaged across all possible problems."
- **Implications:**
  - **Closed Classification World Assumption (CCWA)**
    - assumes that *classification tasks in real applications form a structured subset of all possible problems*, allowing algorithms to be selected based on good performance within that subset
  - **Open Classification World Assumption (OCWA)**
    - assumes there is *no inherent structure in real-world classification tasks*, so the applicability of an algorithm is (only after observing its performance) well characterized
  - **In practice (CCWA is favored):**
    - Algorithms are compared on benchmark datasets (e.g., UCI), assuming they represent relevant real-world tasks (which is hard to characterize)
    - New algorithms are proposed to address known limitations and evaluated empirically (Generalization to unseen tasks often remains unclear)

Rice Framework for Algorithm Selection **(vs. running parallel)**



Formal Problem Definition:

- **Problem space** P: set of classification tasks; if too small, it can be augmented with artificially generated problems
- **Feature space** F: space of meta-feature vectors $f(x) \in \mathbb{R}^d \subseteq F$, where f is a feature extraction function applied to classification tasks $x \in P$; each vector describes dataset-level characteristics
- **Algorithm space** A: set of candidate machine learning algorithms
- **Performance space** Y: performance values $y(a, x)$ for applying algorithm $a \in A$ to task $x \in P$

- **Meta-training set:** $\{\langle f(x), t(x)\rangle : x \in P' \subseteq P\}$ with $t(x) = \arg\max_{a \in A} y(a, x)$
- **Selection model** S: a meta-learner trained on the meta-training set, which predicts the best algorithm $a \in A$ for a task based on its meta-features $f(x)$, i.e., $S(f(x)) = a$

Goal: Given a classification task $x \in P$, with meta-features $f(x) \in F$, find a selection mapping $S(f(x)) \to a \in A$ that maximizes performance $y(a, x) \in Y$ to select the best algorithm for each individual task (local selection) or to find a single algorithm that performs well across tasks (global selection).

- AS is inherently incremental, as (single) new tasks continuously extend the meta-knowledge base.
- In practice, the success depends on how f, S, and y are selected — S is itself a learning algorithm, and practical success depends on training data quality, algorithm set A, and computational cost.
  - **Feature space** F: meta-features should provide informative and predictive signals for selecting appropriate algorithms; should be inexpensive to compute, i.e., $\text{cost}(f(x)) \ll \text{cost}(t(x))$.
    - **Statistical / information-theoretic:** global dataset properties (e.g., number of features or classes, class entropy, feature-label correlation), based on the assumption that learning algorithms are sensitive to the structure of the dataset.

    - **Model-based:** properties of hypotheses induced on a particular problem are used as an indirect form of characterization (e.g., number of nodes per feature, tree depth, imbalance).
    - **Landmarking:** characterizes tasks using the performance of simple learners (landmarkers), locating them in an expertise space and inferring task similarity based on shared areas of learner competence; landmarkers serve as efficient indicators for task similarity and learner suitability and should be computationally efficient.
  - **Algorithm space** A: should consist of the smallest set of complementary base learners with diverse inductive biases, ideally covering different model classes and their hyperparameter configurations.
  - **Performance space** Y: typically defined via accuracy, but may include other measures (e.g., complexity, compactness); algorithms can also be ranked per task.
  - **Selection model** S: Induction cost: depends on the chosen meta-learning regime;
    Prediction cost: typically negligible and not problematic

vs. Model Combination: (e.g., ensemble methods) merges multiple algorithms into a single system to reduce misclassification, metalearning for algorithm selection aims to choose the best algorithm for a given problem based on meta-features.

7.2 AutoML

Def: The process of automating the configuration steps in supervised machine learning tasks on a dataset – including hyperparameter optimization, algorithm selection, and feature selection or preprocessing. Algorithm selection and preprocessing steps (e.g., data normalization) can be treated as nominal hyperparameters.

Hyperparameter Optimization: (special case of model selection)

- Given a machine learning algorithm A with hyperparameters $\boldsymbol{\lambda} = (\lambda_1, \ldots, \lambda_n) \in \Lambda = \Lambda_1 \times \cdots \times \Lambda_n$, and k training/validation splits $D^{(i)}_{\text{train}}, D^{(i)}_{\text{valid}}$, the goal is to minimize the average validation loss over k-fold cross-validation:
$$f(\boldsymbol{\lambda}) = \frac{1}{k} \sum_{i=1}^{k} L(A_{\boldsymbol{\lambda}}, D^{(i)}_{\text{train}}, D^{(i)}_{\text{valid}})$$
- black-box optimization problem with expensive evaluations and no gradient information
  → **Search problem:**
  - **Grid search:** search over a discretized parameter grid; simple but inefficient in high dimensions
  - **Randomized search:** randomly samples configurations from the search space according to a distribution; can be more effective when only few parameters strongly affect performance as it produces diversity fastly
  - **Bayesian Optimization:** sequential model-based optimization using a surrogate model (e.g., regression) to approximate the objective function; selects new configurations by maximizing an acquisition function:
    - Initialize observation set $\mathcal{H} \leftarrow \emptyset$ and surrogate model $\mathcal{M}$ trained on $(\lambda, \mathcal{L}(A_\lambda))$ to approx. f
    - While time budget not exhausted:
      - Select a candidate configuration $\lambda \in \Lambda$ by maximizing the *acquisition function* $a_{\mathcal{M}}(\lambda)$ (e.g., expected improvement), which uses $\mathcal{M}$ to identify promising inputs for evaluation.
      - Evaluate the selected configuration λ by computing $f(\lambda)$. Since this evaluation is expensive, $\mathcal{M}$ focuses the search on informative regions, i.e., areas with high predicted performance, high uncertainty, or both.
      - Add the new observation $(\lambda, \mathcal{L})$ to $\mathcal{H}$ and update $\mathcal{M}$ accordingly.

    - Return the configuration $\lambda^* \in \mathcal{H}$ with the lowest observed validation loss.
  - **SMAC:** practical Bayesian optimization framework using random forests as surrogate model; handles categorical and conditional hyperparameters and is applicable beyond ML to hard combinatorial problems.

8 Reinforcement Learning

Def: A machine learning paradigm where the agent observes the state of the environment, selects an action based on its policy, receives a reward signal as feedback, and updates its behavior using a value function — optionally relying on a model of the environment to predict future states and rewards.

- **Agent:** Learns through trial-and-error which actions to take by interacting with the environment based on the consequences of *its own* actions (rather than from supervised data).
  It must:
  - sense the current **state** of the environment and take actions that influence the state
  - balance **exploration** (trying new actions) and **exploitation** (choosing actions that yielded high rewards)
- **Policy:** The agent's strategy: a mapping from states to actions. Determines which action is taken in each state, try to find hidden structure.
- **Reward signal:** A scalar feedback from the environment after taking an action. Defines the goal: maximize the cumulative reward (possibly delayed over time).
- **Value function:** Estimates how good a state (or state–action pair) is in terms of expected future rewards. Helps the agent evaluate long-term outcomes beyond immediate rewards.
- **Model of the environment:** Approximates how the uncertain environment behaves; used for planning and simulating outcomes of actions.

Tabular Solution Methods: apply to small state and action spaces and can compute optimal value functions and policies exactly (unlike approximate methods)

8.1 k-armed Bandit Problem

A class of reinforcement learning problems with no state transitions, where the agent repeatedly selects one of k actions (arms), each with an unknown reward distribution, aiming to maximize expected cumulative reward.

- The true value of an action a at time t is $q_*(a) = \mathbb{E}[R_t \mid A_t = a]$. (initial $Q_1(a) = 0 \quad \forall a$)
- Agent estimates $q_*(a)$ using the sample average: $Q_t(a) \doteq \frac{\text{sum of rewards when } a \text{ taken prior to } t}{\text{number of times } a \text{ taken prior to } t} = \frac{\sum_{i=1}^{t-1} R_i \cdot \mathbf{1}_{A_i=a}}{\sum_{i=1}^{t-1} \mathbf{1}_{A_i=a}}$
  - **Incremental Implementation:** $Q_{n+1} = Q_n + \frac{1}{n}(R_n - Q_n)$ (as stationary)
  - **non-stationary:** $Q_{n+1} = (1-\alpha)^n Q_1 + \sum_{i=1}^{n} \alpha(1-\alpha)^{n-i} R_i$ (recency-weighted)
- **Optimistic initial values:** encourage exploration by assigning high initial estimates, causing a greedy agent to try all actions as observed rewards fall short. (doesn't converge in non-stationary)
- Greedy action selection: $A_t = \arg\max_a Q_t(a)$
- ε-greedy: with probability ε (e.g. 5–10%), select an action uniformly at random (exploration)
- **Upper-Confidence-Bound Action Selection:** Select according to how close their estimates are to being maximal and the uncertainties in those estimates: $A_t = \arg\max_a \left[Q_t(a) + c \cdot \sqrt{\frac{\ln t}{N_t(a)}} \right]$
  This encourages trying actions that are promising but not yet sufficiently explored.

10-armed Testbed: A benchmark consisting of 2000 randomly generated 10-armed bandit problems. Used to evaluate and compare action selection strategies under controlled randomness.
For each problem:

- True action values $q_*(a)$ are sampled from $\mathcal{N}(0,1)$
- Each observed reward R_t is drawn from $\mathcal{N}(q_*(A_t),1)$

8.2 Markov Decision Processes

Context: MDPs formalize sequential decision-making problems, where the agent interacts with an environment over discrete time steps. Unlike the k-armed bandit problem, the outcome of an action depends not only on the action itself but also on the current state and influences future states and rewards.

To define this interaction probabilistically, we use the transition-reward function

$$p(s',r \mid s,a) \doteq \Pr(S_t=s', R_t=r \mid S_{t-1}=s, A_{t-1}=a), \quad where \sum_{s'\in\mathcal{S}}\sum_{r\in\mathcal{R}} p(s',r \mid s,a)=1 \quad \text{for all } s\in\mathcal{S}, a\in\mathcal{A}(s)$$

which gives the likelihood of arriving in state s' and receiving reward r after taking action a in state s. From this function, we can derive useful quantities like:

- $p(s' \mid s,a) \doteq \Pr\{S_t=s' \mid S_{t-1}=s, A_{t-1}=a\} = \sum_{r\in\mathcal{R}} p(s',r \mid s,a)$
  (transition probability of reaching state s'),
- $r(s,a) \doteq \mathbb{E}[R_t \mid S_{t-1}=s, A_{t-1}=a] = \sum_{r\in\mathcal{R}} r \cdot \sum_{s'\in\mathcal{S}} p(s',r \mid s,a)$ (expected reward value),
- $r(s,a,s') = \mathbb{E}[R_t \mid S_{t-1}=s, A_{t-1}=a, S_t=s'] = \sum_r r \cdot \frac{p(s',r|s,a)}{p(s'|s,a)}$ (as $\Pr(A \mid B,C) = \frac{\Pr(A,C|B)}{\Pr(C|B)}$).
  (expected reward for transitioning to state s' after taking action a in state s)

Def (Finite MDP): is defined by a tuple $(\mathcal{S},\mathcal{A},p,R,\gamma)$, where

- $\mathcal{S}$ is a finite set of states, and $\mathcal{A}$ is a finite set of actions,
- $p:\mathcal{S}\times\mathcal{R}\times\mathcal{S}\times\mathcal{A}\to[0,1]$ defined as above
- $R(s,a)=\mathbb{E}[R_t \mid S_{t-1}=s, A_{t-1}=a]$
  (expected reward at time t after taking action a in state s at time $t-1$)
- $\gamma\in[0,1]$ is the discount factor for future rewards.

The goal is to estimate the **optimal action-value function:**

$$q_*(s,a)=\mathbb{E}[G_t = R_t+\gamma R_{t+1}+\gamma^2 R_{t+2}+\ldots (\approx R_{t+1}+\gamma G_{t+1}) \mid S_t=s, A_t=a]$$

which gives the expected return when starting in state s, taking action a, and thereafter following the optimal policy.

A **policy** π is a mapping from states to probabilities over actions: $\pi(a \mid s)=\Pr(A_t=a \mid S_t=s)$
It defines the agent's behavior: in each state s, it gives the probability of taking action a.

Before estimating the optimal value function, define **value functions with respect to a given policy π:**

- **State-value function:** $v_\pi(s) \doteq \mathbb{E}_\pi[G_t \mid S_t=s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k R_{t+k+1}\middle| S_t=s\right]$, for all $s\in\mathcal{S}$
  expected return when starting in state s and following policy π.
- **Action-value function:** $q_\pi(s,a) \doteq \mathbb{E}_\pi[G_t \mid S_t=s, A_t=a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k R_{t+k+1}\middle| S_t=s, A_t=a\right]$
  expected return when starting in state s, taking action a, and thereafter following policy π.

The Bellman equation expresses a recursive relationship between the value of a state and the values of its successor states under a policy π:

$$v_\pi(s)=\mathbb{E}_\pi[G_t \mid S_t=s]=\mathbb{E}_\pi[R_{t+1}+\gamma G_{t+1} \mid S_t=s]=\sum_a \pi(a \mid s)\sum_{s',r} p(s',r \mid s,a)\left[r+\gamma v_\pi(s')\right], \quad \text{for all } s\in\mathcal{S}$$

This equation averages over all possible actions and successor states, weighting each by its probability of occurring under policy π.

Goal: Find a policy π_* that *maximizes long-term expected reward*. We do this via the optimal value functions:

- **Optimal state-value function:** $v_*(s) \doteq \max_\pi v_\pi(s)$ gives the best expected return from state s.
- **Optimal action-value function:** $q_*(s,a) = \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] = \max_\pi q_\pi(s,a)$ gives the expected return for taking action a in s and following π_* thereafter.

These functions can be computed using **Dynamic Programming**, which solves the Bellman equations exactly — but it requires full knowledge of the environment dynamics and is often computationally expensive.

8.3 Monte Carlo Methods

Def: Estimate value functions based on complete episodes of experience, without requiring knowledge of the environment's dynamics (model-free / no model of the environment). Updates are made only at the end of each episode.

The idea is to average observed returns following visits to a state (or state-action pair) under a fixed policy π, which approximates the expected return. Each state can be seen as a separate bandit problem, though they are interrelated since actions in one state influence future states and rewards within the same episode.

Monte Carlo Prediction: Given a policy π, estimate the state-value function $v_\pi(s)$ by averaging returns:

$$v_\pi(s) \approx \frac{1}{N} \sum_{i=1}^{N} G_t^{(i)} \quad \text{where each } G_t^{(i)} \text{ is a return following a visit to } s$$

- **First-visit MC:** only the return following the *first* visit to s in each episode is used.
- **Every-visit MC:** averages all returns following *every occurrence* of state s in an episode.

Monte Carlo Estimation of Action Values: Estimates the action-value function $q_\pi(s,a)$ by averaging returns after visiting state–action pairs (s,a) under a fixed policy π. A pair (s,a) is visited if state s is encountered and action a is taken. Without a model, state values alone are not sufficient to derive a policy — we need $q_\pi(s,a)$.

Challenge: Some pairs may **never be visited** under a deterministic policy. **Solutions:**

- **Exploring starts:** start episodes from randomly chosen (s,a).
- **Stochastic policies:** ensure every action has non-zero probability in every state.

Monte Carlo Control: Iteratively improve the policy using sample-based estimates of $q_\pi(s,a)$.
Goal: Find an optimal policy π^* that maximizes expected return.

- Alternate between:
  - **Policy evaluation:** estimate $q_\pi(s,a)$ from episodes.
  - **Policy improvement:** update $\pi(s) \leftarrow \arg\max_a q(s,a)$.
- This is **sample-based policy iteration**.
- Greedy improvement guarantees:
$$q_{\pi_k}(s, \pi_{k+1}(s)) \geq v_{\pi_k}(s)$$
- Converges to an optimal policy π^* under sufficient exploration.