Machine Learning

Passerini Andrea
a.y. 2025/2026
University of Trento

Davide Donà

January 2026

Notes Repository

Contents

1 Introduction **1**
1.1 Machine Learning Systems 1
1.2 Learning settings 3
1.3 Supervised Learning Task 4
1.4 Unsupervised Learning tasks 4
1.5 Probabilistic reasoning 4
1.6 Choice of learning algorithms 4

2 Decision Tree Learning **5**
2.1 Appropriate problems 6
2.2 Learning decision tree 6
2.2.1 Choosing the best attribute 6
2.3 Issues in decision tree learning 7
2.3.1 Overfitting avoidance 7
2.3.2 Dealing with continuous-valued attributes: 8
2.3.3 Alternative attributes test measure 8
2.3.4 Handling Attributes with missing values 9
2.4 Random Forests 9

3 K-Nearest Neighbors **10**
3.1 Metric function 10
3.2 Classification 11
3.3 Regression 11
3.4 Characteristics 12
3.5 Distance-weighted K-Nearest Neighbor 12

4 Evaluation **13**
4.1 Binary classification 13
4.1.1 Accuracy 14
4.1.2 Precision and Recall 15
4.1.3 Precision-Recall curve 15
4.2 Multi-class classification 16
4.3 Regression 17
4.4 Performance estimation 17
4.4.1 Hold-out procedure 17
4.4.2 K-Fold Cross-validation 18

5 Parameter Estimation **18**
5.1 Maximum Likelihood Estimation 20
5.1.1 Maximizing the log-likelihood 20
5.1.2 Example: MLE for Gaussian distribution 20
5.2 Bayesian Estimation 22
5.3 Sufficient statistics 24

5.4 Conjugate priors 24
5.4.1 Example: Bernoulli distribution 25

6 Bayesian Networks **25**

7 Learning Bayesian Networks **25**

8 Linear Discriminant Functions **25**
8.1 Advantages and disadvantages of discriminative models 26
8.2 Linear Discriminant Functions 26
8.3 Linear Binary Classifier 27
8.4 Multiclass Classification 28
8.4.1 One-vs-All (OvA) 28
8.4.2 One-vs-One (OvO) 28
8.4.3 Onve-vs-One (OvO) vs One-vs-All (OvA) 29

9 Perceptron **29**
9.1 Biological Motivation 29
9.2 Single Neuron Architecture 29
9.2.1 Augmented feature / weight vector 29
9.2.2 Representational power 29
9.3 Parameter learning 30
9.3.1 Gradient descent 30
9.3.2 Stochastic Gradient Descent 31
9.4 Perceptron Regression 31
9.4.1 Error function 32
9.4.2 Closed form solution 32
9.4.3 Gradient descent 33

10 Support Vector Machines **33**
10.1 Maximum Margin Classifier 34
10.2 Hard Margin SVM 34
10.2.1 Learning problem 35
10.2.2 Karush-Kuhn-Tucker (KKT) Approach 35
10.2.3 Lagrangian of the SVM problem 36
10.2.4 Observations 37
10.3 Soft Margin SVM 38
10.3.1 Learning problem 38
10.3.2 Regularization theory 38
10.3.3 Lagrangian of the Soft Margin SVM 39
10.3.4 Observations 40
10.4 Large Scale SVM 40
10.5 Margin Error Buond 41

11 Kernel Machines **41**
11.1 Feature Map . . . 42
11.2 Linear separation in feature space . . . 43
11.3 Kernel trick . . . 43
11.3.1 Examples of kernels . . . 43
11.3.2 Valid Kernels . . . 44
11.4 Basic Kernels . . . 45
11.5 Kernel combinations . . . 45
11.5.1 Kernel Sum . . . 46
11.5.2 Kernel Product . . . 46
11.5.3 Linear combination of kernels . . . 46
11.5.4 Kernel normalization . . . 47
11.6 Kernel on Graphs . . . 47
11.6.1 Graph Isomorphism . . . 47
11.6.2 Weisfeiler-Lehman Test . . . 47
11.6.3 Weisfeiler-Lehman Kernel . . . 48

12 Neural Networks **48**
12.1 Why Neural Networks? . . . 48
12.2 Multi Layer Perceptron . . . 49
12.2.1 Activation Functions . . . 50
12.2.2 Output Layer . . . 50
12.2.3 representational power of MLPs . . . 51
12.3 Training Multi Layer Perceptrons . . . 51
12.3.1 Loss Functions . . . 52
12.3.2 Stochastic Gradient Descent . . . 52
12.3.3 Backpropagation . . . 52
12.4 Modular Structure of Neural Networks . . . 54
12.5 Remarks on Training Neural Networks . . . 54
12.5.1 Stopping Criteria . . . 55
12.5.2 Vanishing gradients . . . 55
12.6 Popular architectures . . . 58

13 Unsupervised Learning - Clustering **58**
13.1 K-Means Clustering . . . 59
13.2 Quality of Clustering . . . 60
13.3 Gaussian Mixture Models (GMM) . . . 60
13.4 Choose the number of clusters k . . . 61
13.4.1 Elbow Method . . . 61
13.4.2 Silhouette Method . . . 62
13.4.3 Hierarchical Clustering . . . 62

14 Reinforcement Learning **63**
14.1 Learning Settings 64
14.2 Markov Decision Process (MDP) 64
14.3 Utilities 64
14.4 Policy 65
14.4.1 Bellman Equations 66
14.4.2 Value Iteration 66
14.4.3 Policy Iteration 66
14.5 Dealing with Partial Knowledge 67
14.5.1 Policy Evaluation 67
14.5.2 Policy Improvement 69
14.6 Scaling to Large Problems 71
14.6.1 TD learning: State Utilities 72
14.6.2 TD learning: Action Utilities 72

15 Ensemble methods **73**
15.1 Bagging 73
15.1.1 Bootstrap resampling 73
15.1.2 Combining methods 74
15.2 Stacking 75
15.3 Boosting 75
15.3.1 AdaBoost 76

1 Introduction

What is Machine Learning: a computer **program** is said to **learn** from **experience** E, with respect to a **class of task** T and a **performance measure** P, if its performance at the task T, as measured by P, increases with the experience E.

- **Task** T: the problem to be addressed (resolved) by the computer;
- **Performance measure** P: evaluate the learned system. Can be tricky to design in cases of data generation;
- **Training experience** E: data, used to train the system.

1.1 Machine Learning Systems

To design a **Machine Learning System**, we have to follow these steps:

1. **Formalizing** the **learning task**;
2. **Collecting** the **data**;
3. **Features extraction**;
4. **Choose** the class of the **learning model**;
5. **Train** the model;
6. **Evaluate** the model;

Going more in depth for each step, we can say:

Formalizing the learning task: we are going to define the task that should be addressed by the learning system. A learning problem is often composed of many related tasks. During this phase, also an appropriate performance measure for the learning system should be defined.

Collecting the data: a set of training examples must be collected in a machine-readable format. The data could be of two types:

- **Labeled data**, used for supervised learning, requires manual intervention for its generation;
- **Unlabeled data**: used in unsupervised learning or in semi-supervised learning.

This is one of the most critical phases of the whole process.

Extracting Features: from the raw data, a relevant set of features (a subset of all the features of the data, only containing relevant information) must be extracted before feeding it as input to the learning system.

- Too **few features** can miss relevant information contained in the data, preventing the system from learning the task with reasonable performance.
- Too **many features** (usually done in deep learning) requires more training data to achieve the same levels of generalization.

Choosing learning model: Based on the task:

- **Simple linear model** is easy to train, but might not be enough for non linearly separable data.

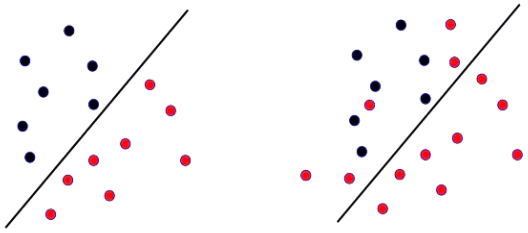


Figure 1: Linear model applied to Linear and Non Linear data

- **Complex model** can learn even non linearly separable data. A too complex model could learn from noise in the data, failing to generalize on new unseen data.

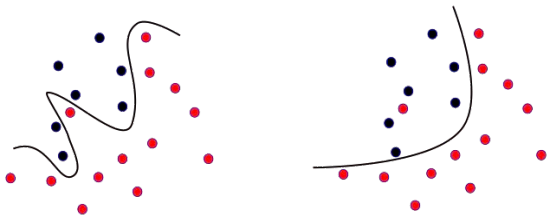


Figure 2: Complex model learning noise on the left, and generalizing well on the right

It's important to note that, when a linear model can be applied, it's usually preferred due to its simplicity and efficiency. In such cases, there is no need to resort to more complex models.

Training the model: this implies searching through the space of possible models, trying to fit the available training examples, according to the chosen performance measure. The learned model should generalize well on unseen data, not just memorize the training examples (overfitting).

Evaluating the model : applying **performance measures** on the **learned model**, using unseen data. This phase can provide insights into the model's weaknesses, giving suggestions for refining it.

1.2 Learning settings

In this section we will analyze the learning setting for different types of learning.

Supervised Learning In supervised learning the learner is provided with a set of input/output pairs $(x_i, y_i) \in X \times Y$, where:

- X: input space (set of possible feature vectors);
- Y: output space (set of possible labels or target values);
- x_i: the i-th input feature vector, with dimension X;
- y_i: the corresponding output (label or target), with dimension Y.

The learned model $f : X \to Y$ should map input into their outputs. Typically, a domain expert is involved in labeling the training data.

Unsupervised Learning The learner is provided with a set of input examples $x_i \in X$, with no labeling information. The task is to model training examples, for example by grouping them into clusters according to their similarity.

Semi-supervised Learning As in the Supervised Learning, the learner is provided with a set of input/output pairs. Also, a much larger group of unlabeled data $x_i \in X$ is provided. Like in supervised learning, the learned model $f : X \to Y$ should map input examples into their outputs.
The additional unlabeled data can be exploited to improve performances of the model (forcing the model to produce similar outputs for similar inputs or to learn the structure of the input data).

Reinforcement Learning The learner is provided with a set of possible **STATES** S, and for each state, a set of possible **actions** A, moving it to a next state. While performing an action a, the learner is provided with an immediate reward $r(s, a)$. The task is to learn a **POLICY**, that given a state s, selects an action a that maximize the overall reward (including future moves).

1.3 Supervised Learning Task

: We can have different types of tasks in supervised learning:

- **CLASSIFICATION**: learning a discrete label (finite number of options). Could be:
  - **Binary**: assign one of two possible classes;
  - **Multiclass**: assign one of $n > 2$ possible classes;
  - **Multi-Label**: assign a *subsets* of size $m \leq n$ of all the possible labels. Each data point can be associated with multiple labels simultaneously.
- **REGRESSION**: assigning a *real* value to an example;
- **RANKING**: ordering a set of examples, according to their importance with the task.

1.4 Unsupervised Learning tasks

:

- **Dimensionality reduction**: reduce the dimensionality of the data, maintaining as much information as possible. A common example is **Principal Component Analysis** (PCA);
- **Clustering**: clustering the data into **homogeneous groups**, according to their similarity;
- **Anomaly detection**:finding instancies in the given examples, that contains errors. This can be used in example for the recognition of anomalous network traffic;

1.5 Probabilistic reasoning

: reasoning in presence of uncertainty. It can be used for:

- evaluating the effect of a certain piece of evidence on other related variables;
- Estimate probability and relations between variables from a set of observation.

1.6 Choice of learning algorithms

Based on the information about the task available to us, we can choose the learning algorithms accordingly:

- **Full knowledge** of distribution of data: **Bayesian decision theory**;

- Form of **distribution known**, **parameters unknown**: parameter estimation from training data. Once learned, we can use **generative methods** to model the distribution;
- **Distribution unknown**, **training examples available**: **discriminative methods**, learn a function predicting the output, given the input.
- **Distribution unknown**, **training examples unavailable**: **unsupervised methods**.

All of these methods will be analyzed in the following sections.

2 Decision Tree Learning

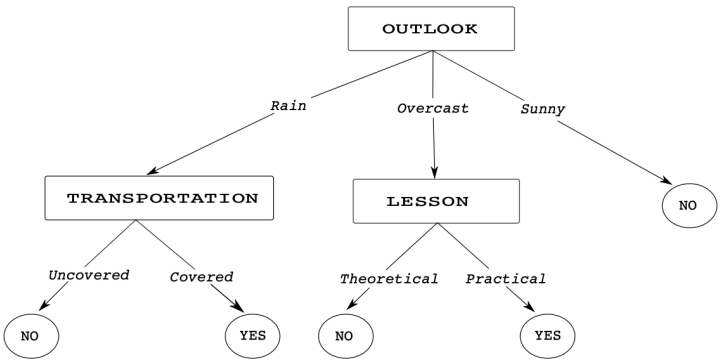


Figure 3: An example of a **learned decision tree**, *Go to lesson*

A **decision Tree** represent a *disjunction* ($\vee$: represent the different path to reach the same outcome) of *conjunction* ($\wedge$: represent the **and** between the conditions on the path to reach the leafs) of constraints over attribute values. As just described, it's used to encode DNF logical formulas:

- Each path from the root to a leaf is a **conjunction** of the **constraints** specified on the node along it. For example, in **fig**3, the leftmost path can be read as:

$$\text{OUTLOOK} = \text{RAIN} \wedge \text{TRANSPORTATION} = \text{Uncovered}$$

- The leaf contains the **label** to be assigned to istances reaching it;
- The **disjunction** of all paths is the logical formula represented by the tree.

2.1 Appropriate problems

There are several tasks where decision trees can be effectively applied. In particular, the task must have these features:

- Binary or multi-class **classification**;
- Instancies are represented as **attribute value pair** (tabular data);
- Different explanation for the same class are possible (disjunction of different paths);
- Some instancies have missing attributes;
- There is a **need** for an **interpretable explanation** for the output. In fact, decision trees can be easily visualized and understood by humans.

2.2 Learning decision tree

To learn **decision trees**, there is a simple **greedy top-down** strategy.
Starting from the **root node** with the full training set, the algorithm follow these steps:

1. Choose best attribute to be evaluated (**fig**3: **outlook** is the first chosen attribute);
2. Add a child node for each attribute value (**fig**3: rain, overcast and sunny are **all possible values** for outlook);
3. Split the node training set into children nodes, according to the attribute values;
4. **Stop splitting** a node if all it's training set belongs to a single class (**homogeneous leaf**), or there are no more attributes to test

2.2.1 Choosing the best attribute

To understand how we can choose the best attribute to be evaluated, we first have to define **entropy**.

Entropy: a measure of the **amount of information** contained in a **collection of instances** S which can take a number of c possible values (labels in our case). In simpler words, entropy quantifies the amount of uncertainty (or impurity) in a dataset:

- If all examples belong to the same class (homogeneous group), then entropy = 0
- If examples are evenly split among classes (uncertainty is maximal), then entropy is maximal.

It's defined as follows:

$$H(S) = -\sum_{i=1}^{c} p_i \log_2 p_i \tag{2.1}$$

Where:

- S is the collection of training examples, the dataset or a subset of it;
- p_i is the fraction of S, with label i. It represents the probability of finding the label i, inside S.

Information Gain: represents the **expected reduction of entropy**, obtained by partitioning S over an attribute A. It's defined as follows:

$$IG(S, A) = H(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} H(S_v) \tag{2.2}$$

Where:

- $H(S)$: entropy of the original set S (before the split);
- $Values(A)$: set of possible values, taken by the attribute A;
- S_v: the subset of S, taking value v at attribute A;
- The second term represents the **sum of entropies** of all the subsets S_v, obtained by partitioning over the attribute A, weighted by their respective sizes.

We can derive that an attribute with **high Information Gain** IG tends to produce homogeneous groups in terms of labels, thus favoring their classification.

2.3 Issues in decision tree learning

2.3.1 Overfitting avoidance

Requiring that each leaf contains only examples of a certain class can lead to very complex trees. A complex tree can easily overfit the training set, incorporating noise in the data.
To solve this problem, it's possible to accept impure leaves, assigning them the label of the majority of their training examples. This process is called **pruning**. There are two possible strategies to prune a decision tree:

- **pre-pruning**: decide whether to stop splitting a node, even if it contains training examples with different labels.
- **post-pruning**: learn the full tree and successively prune it, removing sub-trees.

Reduces Error Pruning: a **post-pruning** strategy, using the validation set. The procedure follows these next steps:

1. for each node in the tree:
   - evaluate the performance on the validation set, after removing the subtree rooted at it;
2. If all node removals made the performance worse, STOP;
3. Choose the node that gave the best performance improvement;
4. Replace the subtree rooted at it with a leaf;
5. Assign to the lead the majority label of all the example in the subtree;
6. Return to 1.

2.3.2 Dealing with continuous-valued attributes:

Continuous valued attributes must be discretized in order to be used as internal node tests, or otherwise, we could end up with an infinite decision tree. The procedure follows these next steps:

1. Examples are **sorted** according to their continuous value;
2. For **each pair of successive examples**, having **different labels**, a **candidate treshold** is placed at the average of the two values;
3. For **each candidate treshold**, the **IG** obtained by splitting the examples, at the given treshold, is computed;
4. The treshold with the **highest IG** is used to discretize the attribute;

2.3.3 Alternative attributes test measure

The previously defined **Information Gain** criterion tends to **prefer attributes** with a **larger number of possible values**, since it will cause data to be split into very small subsets.
As an extreme case, if we consider a unique attribute in our data, such as an ID, it will cause the data to be **perfectly split into singletons**. This may look like a perfect split, but offers no generalization on new examples.

Attribute Value Entropy: To resolve this issue, a measure of the entropy of the dataset S, with respect to the values of an attribute A is defined:

$$H_A(S) = -\sum_{v \in Values(A)} \frac{|S_v|}{S} \log_2 \frac{|S_v|}{S} \tag{2.3}$$

- If, for a certain attribute (ex: ID), every value is unique, then $H_A(S)$ is very high;

- If a certain attribute has only a few values (ex: binary attribute), then $H_A(S)$ is very low;

Information gain Ratio: We can then define an **Information Gain Ratio** measure, that **down weights** the **IG** by the **Attribute Value Entropy**

$$IGR(S, A) = \frac{IG(S, A)}{H_A(S)} \tag{2.4}$$

Obviously, we have that:

- If the attribute A has many unique values, then $H_A(S)$ is high, resulting in a lower **IGR**, for the attribute A;
- If the attribute A has few unique values, then $H_A(S)$ is low, resulting in a higher **IGR**.

2.3.4 Handling Attributes with missing values

Assuming an **example** x, belonging to **class** $c(x)$ has a missing value for the attribute A. When the attribute A is to be tested at node n, we can apply two solutions:

- **SIMPLE SOLUTION**: Assign, to the example x, the most common attribute A values among the training examples at node n;
- **COMPLEX SOLUTION**: propagate x to each children of n, with a fractional value equal to the proportion of examples with the corresponding attribute value.
  At test times for each candidate class (since more leaf nodes are reached), all fraction belonging to the same class are summed. The example is assigned the class with the highest overall value.

2.4 Random Forests

Random Forests are an *ensemble learning method*, that combines multiple decision trees to improve classification or regression performance.

Training To train a Random Forest, the following steps are performed:

1. Given a training set of N examples, sample N examples with replacement (each example can be selected multiple times);
2. Train a decision tree on the sampled data, but at each split, choose the best split among a random subset of features, rather than all features. This approach allows to introduce more diversity among the trees;
3. Repeat steps 1 and 2 to create a forest of M trees.

Prediction To make predictions with a Random Forest, the following steps are performed:

- For classification tasks, each tree in the forest makes a prediction for the input example;
- The final prediction is made by majority voting among all trees.

A more in depth explanation of ensemble methods can be found in **section 15**.

3 K-Nearest Neighbors

To better understand this **non-parametric** machine learning algorithm, we will first analyze its **1-Nearest Neighbor** variant.

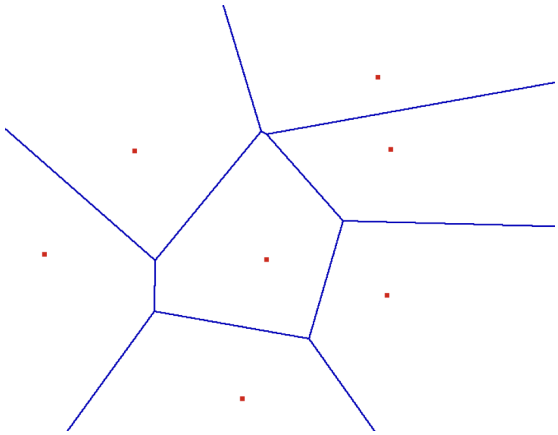


Figure 4: Example of a 1-Nearest Neighbor

- Each **red dot** represents one instance of the **training set**. Each instance has two features, and every label is unique;
- The **blue lines** represent the **decision boundaries**. Each region contains the set of points that are **closer** to the **red dot inside that region** than to any other red dot;
- For every instance of the test set, we can determine in **which region** it falls, and the predicted label will be the one associated with the red dot of that region.

3.1 Metric function

To implement this algorithm, we first have to define a **metric** to measure the **distance between instances**.

Given a set $\mathcal{X}$, a function $d : \mathcal{X} \times \mathcal{X} \to R_0^+$ is a **metric** iff for any $x, y, z \in \mathcal{X}$ the following properties are satisfied:

- **reflexivity**: $d(x, y) = 0 \;\; iff \;\; x = y$
- **symmetry**: $d(x, y) = d(y, x)$
- **triangle inequality**: $d(x, y) + d(y, z) \geq d(x, z)$

An example of **metric** is the **Euclidean distance in** R^n. It's defined as follows:

$$d(x, y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2} \tag{3.1}$$

3.2 Classification

Algorithm 1 k-Nearest Neighbor Classification

1: **for all** test examples x **do**
2: **for all** training examples (x_i, y_i) **do**
3: compute distance $d(x, x_i)$
4: **end for**
5: select the k-nearest neighbors of x
6: return class of x as majority class among neighbors:

$$\arg\max_y \sum_{i=1}^{k} \delta(y, y_i)$$

7: **end for**

Where:

$$\delta(x, y) = \begin{cases} 1 & \text{if } x = y \\ 0 & \text{otherwise} \end{cases}$$

Assigns a class to a test example by looking at the k training examples closest to it. Given a text example x, the predicted class of x is the majority class y among its K-nearest neighbors.

3.3 Regression

Here, given a text example x, the predicted value of x is the mean of the values among its K-nearest neighbors.

Algorithm 2 k-Nearest Neighbor Regression

for all test examples x **do**
  for all training examples (x_i, y_i) **do**
    compute distance $d(x, x_i)$
  end for
  select the k-nearest neighbors of x
  return the average output value among neighbors:

$$\frac{1}{K}\sum_{i=1}^{k} y_i$$

end for

3.4 Characteristics

- **Instance-based learning**: the model used for prediction is calibrated for test example to be processed. This model doesn't build a real general representation for data, but it's limited to the example on the training set.
- **Lazy learning**: computation is mostly deferred to the classification phase. In fact for every new example, we compare it with every other example in the training set. There is no real training phase.
- **Local learner**: assumes prediction should be mainly influenced by nearby instances.
- **Uniform feature weighting**: every feature contribute at the same way on computing distance. So there are not parameters to be learned.

3.5 Distance-weighted K-Nearest Neighbor

The just presented algorithm can be enhanced with its **Distance-Weighted K-Nearest Neighbor** variation.
Take as an example this case here, using 3-Nearest Neighbors:

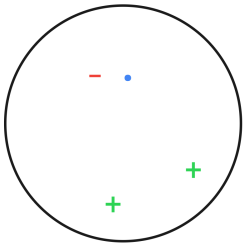


Figure 5: Example of issue with K-Nearest Neighbors

Even though the **blue dot**, representing the testing instance, is closer to the **- label**, the algorithm will still assign the **+ label**, since it represents the majority label among its neighbors.

We can weight the **influence** that each neighbor has on the label, based on its distance from the testing instance, giving closer neighbors a higher impact on the final classification.
We first define a new function:

$$w_i = \frac{1}{d(x, x_i)}$$

The algorithms now present as follows:

Classification

$$\arg\max_y \sum_{i=1}^{k} w_i \delta(y, y_i)$$

Regression

$$\frac{\sum_{i=1}^{k} w_i y_i}{\sum_{i=1}^{w} y_i}$$

4 Evaluation

In this section we will discuss about *Evaluation*, the process of assessing a model's performance, reliability, and generalizability using specific quantitative metrics and qualitative techniques.

- Requires to define a **performance measures** to be optimized. These vary based on the task we are addressing;
- Performance of learning algorithms cannot be evaluated on the entire domain. We have to introduce some approximations;
- Performance evaluation is needed for
  - **Tune** the hyper-parameters of learning methods;
  - **Evaluating** the performance of the learned model;
  - **Computing statistical significance** of results.

4.1 Binary classification

The typical way to evaluate a binary classifier is by means of a **confusion matrix**: Where:

- The rows represent the **actual** classes (ground truth);

		prediction	
		P	N
ground truth	P	TP	FN
	N	FP	TN

Table 1: Confusion matrix for binary classification

- The columns represent the **predicted** classes;
- Each entry in the matrix represents the number of examples that fall into that category:
  - **TP** (True Positive): number of positive examples correctly classified as positive;
  - **TN** (True Negative): number of negative examples correctly classified as negative;
  - **FN** (False Negative): number of positive examples incorrectly classified as negative;
  - **FP** (False Positive): number of negative examples incorrectly classified as positive;

From the confusion matrix we can derive several performance measures.

4.1.1 Accuracy

Accuracy

Accuracy is the fraction of correctly labelled example among all predictions.

$$Acc = \frac{TP + TN}{TP + TN + FP + FN} \tag{4.1}$$

Accuracy metric is not very reliable when the classes are imbalanced. For example, if in a dataset the positive class represents a small fraction of the examples, a classifier that always predicts negative will achieve a high accuracy, but it wouldn't be able to predict any positive example.
A possible solution consists in **rebalancing costs**, making each positive example worth more than a negative one (e.g., each positive counts $\frac{N}{P}$, where $N =$ number of negative examples, and $P =$ number of positive examples).

4.1.2 Precision and Recall

Precision

Precision is the fraction of correctly predicted positive examples among all predicted positive examples.

$$Prec = \frac{TP}{TP + FP} \tag{4.2}$$

This metric measures the precision of the learner in predicting positive examples. A problem with precision is that the learner can achieve a high precision by predicting only a few positive examples, only when 100% sure.

Recall

Recall (or sensitivity) is the fraction of correctly predicted positive examples among all actual positive examples.

$$Rec = \frac{TP}{TP + FN} \tag{4.3}$$

It measures the ability of the learner to find all positive examples.

Since these last two metrics are complementary, we can combine them into a single one, called **F-measure**.

F-measure

F-measure combines **precision** and **recall**, balancing their trade-off.

$$F_\beta = \frac{(1 + \beta^2) \cdot Prec \cdot Rec}{\beta^2 \cdot Prec + Rec} \tag{4.4}$$

F_1 is usually a good performance measure in the case of unbalanced datasets.

4.1.3 Precision-Recall curve

The **Precision-Recall curve** is a graphical representation that illustrates the trade-off between precision and recall for different threshold values.

Many classifiers output a confidence (probability score) for each prediction. All the evaluation metrics seen so far measure the performance of the classifier at a specific threshold. We can vary this threshold from min to max to obtain different pairs of precision and recall values, which can be plotted to create the Precision-Recall curve. Also, a single value can be computed by computing the

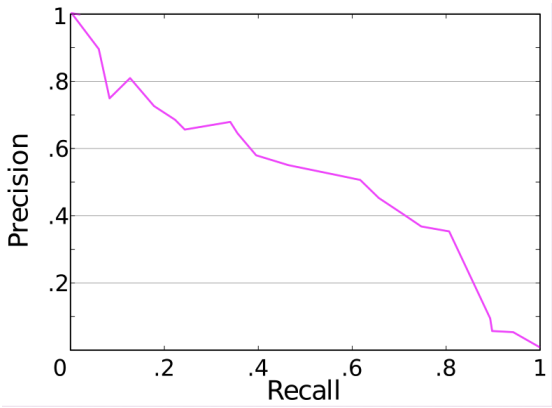


Figure 6: Precision-Recall curve example

area under the curve. It combines the performance of the classifier across all thresholds into a single metric.

4.2 Multi-class classification

In the case of multi-class classification, we can simply extend the concepts of precision, recall, and F-measure to multiple classes. We first redefine the **confusion matrix** as a generalization of the binary case:

		prediction		
		y_1	y_2	y_3
ground truth	y_1	n_{11}	n_{12}	n_{13}
	y_2	n_{21}	n_{22}	n_{23}
	y_3	n_{31}	n_{32}	n_{33}

Table 2: Confusion matrix for multi-class classification

Where:

- n_{ij} is the number of examples of class y_i predicted as class y_j;
- The **main diagonal** elements n_{ii} represent the number of correctly classified examples for each class;
- The **sum** of off-diagonal elements in a **column** j ($\sum_{i \neq j} n_{ij}$) represents the number of **false positives** for class y_j;
- The **sum** of off-diagonal elements in a **row** i ($\sum_{j \neq i} n_{ij}$) represents the number of **false negatives** for class y_i;

From this confusion matrix, we can compute precision and recall for each class y_i as follows:

- **Precision** for class y_i:
$$Prec_i = \frac{n_{ii}}{\sum_j n_{ji}}$$
- **Recall** for class y_i:
$$Rec_i = \frac{n_{ii}}{\sum_j n_{ij}}$$

In addition, we can extend the definition of accuracy to the multiclass setting.

Multiclass accuracy

Multiclass accuracy is the fraction of correctly predicted examples among all examples.

$$MAcc = \frac{\sum_i n_{ii}}{\sum_i \sum_j n_{ij}} \tag{4.5}$$

4.3 Regression

The typical performance measures in the regression domain is the **Root Mean Squared Error** (RMSE). It represents the **distance** between the predicted values $f(x_i)$ and the actual values y_i.

Root Mean Squared Error

Root Mean Squared Error is defined as:

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(f(x_i) - y_i)^2} \tag{4.6}$$

For a dataset $\mathcal{D}$ with $|\mathcal{D}| = N$ examples.

4.4 Performance estimation

To estimate the performance of a learning algorithm, we need to evaluate it on data that has not been used for training. To ensure this, we can use two main strategies.

4.4.1 Hold-out procedure

Given a dataset $\mathcal{D}$, we split it into three disjoint subsets:

- **Training set** $\mathcal{D}_{train}$: used to train the model;
- **Validation set** $\mathcal{D}_{val}$: used to tune hyper-parameters and select the best model;

- **Test set** $\mathcal{D}_{test}$: used to evaluate the final performance of the selected model.

This procedure is simple and effective, but can be applied only when we have a sufficiently large dataset are available.

4.4.2 K-Fold Cross-validation

The **k-fold** cross validation approach **splits** the dataset $\mathcal{D}$ into k disjoint subsets (folds) of equal size. Then, for each $i \in [1, k]$:

- **Train** a predictor $\mathcal{L}$ using dataset $T_i = \mathcal{D} \setminus \mathcal{D}_i$ (all folds except the i-th one);
- **Compute score** S of the **predictor** $\mathcal{L}(T_i)$:

$$S_i = S_{\mathcal{D}_i}[\mathcal{L}(T_i)]$$

After this iterative procedure, we can compute the **average score** over all folds:

$$\bar{S} = \frac{1}{k} \sum_{i=1}^{k} S_i$$

5 Parameter Estimation

Setting: In order to understand what we will be doing in this section, we need to first introduce the setting of the environment we are working in. We have:

- A **collection of data** sampled from a **probability distribution** $p(x, y)$;
- The **probability distribution** $p(x, y)$ is **known**, but the parameters θ of the distribution are **unknown**;
- There is a **training set** $\mathcal{D} = \{(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)\}$ of N examples, sampled **i.i.d.** from the distribution $p(x, y)$;

I.I.D.

Independent and Identically Distributed (i.i.d.) refers to a set of random variables that are all drawn from the same probability distribution and are mutually independent.

Multiclass classification setting:

- The training set $\mathcal{D}$ can be divided into $\mathcal{D}_1, \ldots, \mathcal{D}_c$, where each $\mathcal{D}_i = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$ contains examples of class y_i;

- For any new example $\boldsymbol{x}$, we can compute the posterior probability for each class y_i (what's the probability of y_i given $\boldsymbol{x}$) using Bayes' theorem:

$$p(y_i|\boldsymbol{x}, \mathcal{D}) = \frac{p(\boldsymbol{x}|y_i, \mathcal{D})p(y_i|\mathcal{D})}{p(\boldsymbol{x}|\mathcal{D})} \tag{5.1}$$

A posteriori probability

The probability of a class y_i given the features $\boldsymbol{x}$ and the training set $\mathcal{D}$, denoted as $p(y_i|\boldsymbol{x}, \mathcal{D})$, is called the posterior probability. It represents the probability of x belonging to class y_i after observing the features $\boldsymbol{x}$.

Simplification: Starting from the equation 5.1, we can simplify it by making some assumptions:

- We assume $\boldsymbol{x}$ is independent of $\mathcal{D}_j (j \neq i)$, so:

$$P(y_i|\boldsymbol{x}, \mathcal{D}) = \frac{p(\boldsymbol{x}|y_i, \mathcal{D})p(y_i|\mathcal{D})}{p(\boldsymbol{x}|\mathcal{D})}$$

- Without additional knowledge, $P(y_i|\mathcal{D})$ can be computed as the **relative frequency** of class y_i in the training set:

$$P(y_i|\mathcal{D}) = \frac{|\mathcal{D}_i|}{|\mathcal{D}|}$$

- The denominator $p(\boldsymbol{x}|\mathcal{D})$ can be computed using the **law of total probability**:

$$p(\boldsymbol{x}|\mathcal{D}) = \sum_{i=1}^{c} p(\boldsymbol{x}|y_i, \mathcal{D})p(y_i|\mathcal{D})$$

At this point the only factor of equation 5.1 that remains unknown is $p(\boldsymbol{x}|y_i, \mathcal{D})$. In order to estimate it, we need to estimate the parameter θ of the distribution. This can be done in two ways:

1. **Maximum Likelihood Estimation** (MLE): we assume that the parameter θ_i is a fixed but unknown value, and we estimate it by maximizing the likelihood of the observed data $\mathcal{D}_i$. Obtained values are used to compute probabilities of new examples:

$$p(\boldsymbol{x}|y_i, \mathcal{D}_i) \approx p(\boldsymbol{x}|\theta_i)$$

2. **Bayesian estimation**: we assume that the parameter θ_i is a random variable with a prior distribution $p(\theta_i)$. Observed data turns the prior into a posterior distribution. The prediction for new examples is obtained by integrating over all possible values of θ_i:

$$p(\boldsymbol{x}|y_i, \mathcal{D}_i) = \int_{\theta_i} p(\boldsymbol{x}, \theta_i|y_i, \mathcal{D}_i) d\theta_i$$

5.1 Maximum Likelihood Estimation

Maximum Likelihood Estimation (**MLE**) is a method used to estimate the parameters of a statistical model. Given a set of training data $\mathcal{D} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$, the estimation of the parameter θ is done by finding the combination of parameters that fits at best the distribution of the observed data, without any prior knowledge about the parameters.

$$\theta_i^* = \text{argmax}_\theta p(\theta_i|\mathcal{D}_i, y_i) = \text{argmax}_{\theta_i} p(\mathcal{D}_i, y_i|\theta_i)p(\theta_i) \tag{5.2}$$

From this, assuming that $p(\theta_i)$ (prior probability) is given, we can derive the most common form of MLE, which doesn't take into account the prior:

$$\theta^* = \text{argmax}_\theta p(\mathcal{D}, y|\theta) \tag{5.3}$$

Since the examples are **i.i.d.**, we can rewrite the equation as:

$$\theta^* = \text{argmax}_\theta \prod_{j=1}^{n} p(\boldsymbol{x}_j|\theta) \tag{5.4}$$

To find the value of θ we would have to compute the derivative of the likelihood function and set it to zero. This is often difficult, so instead we work with the **log-likelihood**.

5.1.1 Maximizing the log-likelihood

The **log-likelihood** is defined as the logarithm of the likelihood function (equation 5.4):

$$\theta^* = \text{argmax}_\theta \sum_{j=1}^{n} \ln p(\boldsymbol{x}_j|\theta) \tag{5.5}$$

In order to **find the maximum**, we compute the derivative of the log-likelihood and set it to zero:

$$\nabla_\theta \sum_{j=1}^{n} \ln p(\boldsymbol{x}_j|\theta) = 0 \tag{5.6}$$

5.1.2 Example: MLE for Gaussian distribution

Assuming that the data is generated from a Gaussian distribution, we can estimate the parameters μ and σ^2 using MLE.
First, the probability density function of a Gaussian distribution is given by:

$$p(x|\mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

Considering its logarithm, we have:

$$\sum_{j=1}^{n} \ln p(\boldsymbol{x}_j|\theta) = \sum_{j=1}^{n} \ln\left[\frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(\boldsymbol{x}_j-\mu)^2}{2\sigma^2}\right)\right] = \sum_{j=1}^{n} \ln(\frac{1}{\sqrt{2\pi\sigma^2}}) - \frac{(\boldsymbol{x}_j-\mu)^2}{2\sigma^2}$$

Estimating μ:

$$\frac{\partial p(\boldsymbol{x}_j|\theta)}{\partial \mu} = \frac{\partial}{\partial \mu}\sum_{j=1}^{n}\ln(\frac{1}{\sqrt{2\pi\sigma^2}}) - \frac{(\boldsymbol{x}_j-\mu)^2}{2\sigma^2}$$

The derivative of the first term is zero, so we only need to compute the derivative of the second term, which is:

$$\sum_{j=1}^{n}\frac{(\boldsymbol{x}_j-\mu)}{\sigma^2}$$

Setting this quantity to zero, we have:

$$\sum_{j=1}^{n}\frac{(\boldsymbol{x}_j-\mu)}{\sigma^2} = 0$$

$$\sum_{j=1}^{n}(\boldsymbol{x}_j-\mu) = 0$$

$$\sum_{j=1}^{n}\boldsymbol{x}_j - \sum_{j=1}^{n}\mu = 0 \implies \sum_{j=1}^{n}\boldsymbol{x}_j - n\mu = 0$$

$$\mu = \frac{1}{n}\sum_{j=1}^{n}\boldsymbol{x}_j$$

Which is the **sample mean** of the data.

Estimating σ: Repating the same process for σ, we have:

$$\frac{\partial p(\boldsymbol{x}_j|\theta)}{\partial \sigma} = \frac{\partial}{\partial \sigma}\sum_{j=1}^{n}\ln(\frac{1}{\sqrt{2\pi\sigma^2}}) - \frac{(\boldsymbol{x}_j-\mu)^2}{2\sigma^2}$$

$$= \sum_{j=1}^{n}\sqrt{2\pi}\sigma\frac{1}{\sqrt{2\pi}}(-\sigma^{-2}) - \frac{(\boldsymbol{x}_j-\mu)^2}{2}(-2\sigma^{-3})$$

$$= \sum_{j=1}^{n} -\frac{1}{\sigma} + \frac{(\boldsymbol{x}_j-\mu)^2}{\sigma^3}$$

if we set the quantity to zero, we have:

$$\sum_{j=1}^{n} -\frac{1}{\sigma} + \frac{(\boldsymbol{x}_j-\mu)^2}{\sigma^3} = 0$$

Multiplying by σ^3, we have:

$$\sum_{j=1}^{n} -\sigma^2 + (\mathbf{x}_j-\mu)^2 = 0$$

$$\sigma^2 = \frac{1}{n}\sum_{j=1}^{n}(\mathbf{x}_j-\mu)^2$$

Conclusion: By this two derivations, we have found that the parameters that maximize the likelihood of the data are the **sample mean** and the **sample variance** of the data.

5.2 Bayesian Estimation

With this approach we focus on the estimation of the parameters of our distribution $p(\boldsymbol{x}, y)$ a priori, meaning that we update our knowledge about the parameters θ based on the observed data.
We start by assuming that the parameters θ_i are **random variables** with a known **prior distribution**.
Predictions for new examples are obtained by integrating over all possible values of θ_i:

$$p(\boldsymbol{x}|y_i, \mathcal{D}_i) = \int_{\theta_i} p(\boldsymbol{x}, \theta_i|y_i, \mathcal{D}_i) d\theta_i$$

Since probability of $\boldsymbol{x}$ given each class y_i is indipendent of the other classes, we can simplify the equation as:

$$p(\boldsymbol{x}|\mathcal{D}) = \int_{\theta} p(\boldsymbol{x}, \theta|\mathcal{D}) d\theta$$

Using the definition of conditional probability, we can rewrite the equation as:

$$p(\boldsymbol{x}|\mathcal{D}) = \int_{\theta} p(\boldsymbol{x}|\theta, \mathcal{D}) p(\theta|\mathcal{D}) d\theta$$

In computing $p(\boldsymbol{x}|\theta, \mathcal{D})$ we can assume that $\boldsymbol{x}$ is independent of $\mathcal{D}$ given θ, since the parameter θ captures all the information about the distribution. Thus, we have:

$$p(\boldsymbol{x}|\mathcal{D}) = \int_{\theta} p(\boldsymbol{x}|\theta) p(\theta|\mathcal{D}) d\theta$$

- The term $p(\boldsymbol{x}|\theta)$ can be easily computed since θ is known;
- The term $p(\theta|\mathcal{D})$ is the **posterior distribution** of the parameter θ given the data $\mathcal{D}$.

Using Bayes' theorem, we can express the posterior distribution as:

$$p(\theta|\mathcal{D}) = \frac{p(\mathcal{D}|\theta) p(\theta)}{p(\mathcal{D})} \tag{5.7}$$

Where $p(\mathcal{D})$ is a constant independent of θ, so we can ignore it for the purpose of estimation.

Example: Bayesian estimation for Gaussian distribution

Unknown μ, known σ: Assuming that the data is drawn from a Gaussian distribution $p(\boldsymbol{x}|\mu) \sim \mathcal{N}(\mu, \sigma^2)$.
The prior distribution for μ is also a Gaussian distribution: $p(\mu) \sim \mathcal{N}(\mu_0, \sigma_0^2)$.
Where:

- μ_0 is the prior mean (our initial guess for the mean of the data);
- σ_0^2 is the prior variance (our initial uncertainty about the mean of the data).

Using Bayes' theorem, we can compute the posterior distribution for μ given the data $\mathcal{D}$. Starting from the equation 5.7 we have:

$$p(\mu|\mathcal{D}) = \frac{p(\mathcal{D}|\mu)p(\mu)}{p(\mathcal{D})}$$

Since we have already seen that $\frac{1}{p(\mathcal{D})}$ is indipendent of μ and that the examples are i.i.d we can derive:

$$p(\mu|\mathcal{D}) = \alpha p(\mathcal{D}|\mu)p(\mu) = \alpha \prod_{j=1}^{n} p(\boldsymbol{x}_j|\mu)p(\mu)$$

From here we can substitute the two Gaussian distributions:
First, the distribution of the data given μ:

$$p(\boldsymbol{x}_j|\mu) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(\boldsymbol{x}_j - \mu)^2}{2\sigma^2}\right)$$

Then the prior distribution of μ:

$$p(\mu) = \frac{1}{\sqrt{2\pi}\sigma_0} \exp\left(-\frac{(\mu - \mu_0)^2}{2\sigma_0^2}\right)$$

Substituting these into our equation for the posterior distribution, we get:

$$p(\mu|\mathcal{D}) = \alpha \prod_{j=1}^{n} \left(\frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(\boldsymbol{x}_j - \mu)^2}{2\sigma^2}\right)\right)\left(\frac{1}{\sqrt{2\pi}\sigma_0} \exp\left(-\frac{(\mu - \mu_0)^2}{2\sigma_0^2}\right)\right)$$

Which simplifies to:

$$p(\mu|\mathcal{D}) = \alpha \prod_{j=1}^{n} \frac{1}{\sqrt{2\pi}\sigma} \exp\left[-\frac{1}{2}\frac{(\boldsymbol{x}_j - \mu)^2}{\sigma^2}\right] \frac{1}{\sqrt{2\pi}\sigma_0} \exp\left[-\frac{1}{2}\frac{(\mu - \mu_0)^2}{\sigma_0^2}\right]$$

$$= \alpha' \exp\left[-\frac{1}{2}\left(\sum_{j=1}^{n}\left(\frac{\mu - \boldsymbol{x}_j}{\sigma}\right)^2 + \left(\frac{\mu - \mu_0}{\sigma_0}\right)^2\right)\right]$$

$$= \alpha'' \exp\left[-\frac{1}{2}\left[\left(\frac{n}{\sigma^2} + \frac{1}{\sigma_0^2}\right)\mu^2 - 2\left(\frac{1}{\sigma^2}\sum_{j=1}^{n}\boldsymbol{x}_j + \frac{\mu_0}{\sigma_0^2}\right)\mu\right]\right]$$

We can in fact see that this composition still behaves like a normal distribution itself.
Solving for μ_n and σ_n^2 we have:

$$\mu_n = \frac{n\sigma_0^2}{n\sigma_0^2 + \sigma^2}\hat{\mu}_n + \frac{n\sigma_0^2}{n\sigma_0^2 + \sigma^2}\mu_0$$

$$\sigma_n^2 = \frac{\sigma_0^2\sigma^2}{n\sigma_0^2 + \sigma^2}$$

Where $\hat{\mu}_n$ is the sample mean of the data.

Interpretation: The Interpretation of the results are the following:

- the **mean** of the posterior distribution μ_n is a **linear combination** of the prior mean μ_0 and the sample mean $\hat{\mu}_n$;
- the **more training examples** (n) are seen, the more the sample mean $\hat{\mu}_n$ influences the posterior mean μ_n;
- the more training examples are seen, the **smaller** the variance σ_n^2 of the posterior distribution becomes, making the distribution sharper around the mean.

5.3 Sufficient statistics

Statistic

Any function on a set of examples $\mathcal{D}$ is called a statistic.

Sufficient statistic

A statistic $\boldsymbol{s} = \phi(\mathcal{D})$ is said to be sufficient for a parameter θ if:

$$P(\mathcal{D}|\boldsymbol{s}, \theta) = P(\mathcal{D}|\boldsymbol{s})$$

If θ is a random variable, then a **sufficient statistic** contains all the information about θ that is present in the data $\mathcal{D}$.

A sufficient statistic allows to summarize a sample $\mathcal{D}$ into a smaller set of values $\boldsymbol{s}$, without losing information. For example **sample mean** and **covariance** are sufficient statistics for the parameters of a Gaussian distribution.

5.4 Conjugate priors

Conjugate prior

Given:

- A likelihood function $p(x|\theta)$;
- A prior distribution $p(\theta)$;

The prior distribution $p(\theta)$ is said to be a **conjugate prior** for the likelihood function $p(x|\theta)$ if the posterior distribution $p(\theta|x)$ is in the same family as the prior distribution $p(\theta)$.

5.4.1 Example: Bernoulli distribution

Consider a **Bernoulli distribution** with events $x = 1$ for success and $x = 0$ for failure. Parameters θ is the probability of success. The **probability mass function** is given by:

$$P(x|\theta) = \theta^x(1-\theta)^{1-x}$$

The the **conjugate prior** is a **Beta distribution** that depends on α_t (number of successes) and α_h (number of failures):

$$P(\theta|\alpha_t, \alpha_h) = \frac{\Gamma(\alpha)}{\Gamma(\alpha_h)\Gamma(\alpha_t)}\theta^{\alpha_t-1}(1-\theta)^{\alpha_h-1}$$

Maximum likelihood estimation: Suppose that a dataset $\mathcal{D} = \{H, H, T, T, T, H, H\}$ of n realizations is given.
The likelihood function is given by:

$$P(\mathcal{D}|\theta) = \theta \cdot \theta \cdot (1-\theta) \cdot (1-\theta) \cdot (1-\theta) \cdot \theta \cdot \theta = \theta^h(1-\theta)^t)$$

The maximum likelihood estimation of θ is given by:

$$\frac{\partial}{\partial\theta}\ln p(\mathcal{D}|\theta) = 0 \implies \frac{\partial}{\partial\theta}\left(h\ln\theta + t\ln(1-\theta)\right) = 0$$

$$\implies \frac{h}{\theta} - \frac{t}{1-\theta} = 0 \implies h(1-\theta) = t\theta \implies \theta = \frac{h}{h+t}$$

t, h are sufficient statistics for the Bernoulli distribution.

6 Bayesian Networks

7 Learning Bayesian Networks

8 Linear Discriminant Functions

We can distinguish **learning models** between **generative models** and **discriminative models**.

- **Generative models** aim to model the **distribution** governing the data. They learn the joint probability distribution $P(x, y)$ and can generate new data points, according to the learned distribution.
- **Discriminative models** aim to model the **decision boundary** between classes, rather than the underlying data distribution.

8.1 Advantages and disadvantages of discriminative models

Pros:

- When data is complex, modelling the underlying distribution can be very challenging (E.g., high resolution images);
- If **data discrimination** is the final goal, then there is no need to model the entire data distribution;
- Discriminative models can focus on learning the parameters that are most relevant for learning the decision boundary, potentially leading to better performance.

Cons:

- The learned model is less flexible in its usage;
- It doesn't allow to perform arbitrary inference tasks on the data. In example, it is not possible to generate new data points from the learned model.

8.2 Linear Discriminant Functions

A **linear discriminant model** uses a **linear function** to make predictions. The **decision boundary** (discriminant function) is a linear combination of the input features:

$$f(\boldsymbol{x}) = \boldsymbol{w}^T \boldsymbol{x} + w_0 \tag{8.1}$$

where:

- $\boldsymbol{x}$ is the input feature vector;
- $\boldsymbol{w}$ is the weight vector, representing the importance of each feature;
- w_0 is the bias term, which allows to shift the decision boundary.

8.3 Linear Binary Classifier

In a **binary classification** problem, the linear discriminant function can be used to classify data points into two classes. This is usually done by taking the sign of the discriminant function:

$$f(\boldsymbol{x}) = \text{sign}(\boldsymbol{w}^T\boldsymbol{x} + w_0) = \begin{cases} +1 & \text{if } \boldsymbol{w}^T\boldsymbol{x} + w_0 \geq 0 \\ -1 & \text{if } \boldsymbol{w}^T\boldsymbol{x} + w_0 < 0 \end{cases} \tag{8.2}$$

The **decision boundary**, which is a **hyperplane** H in the feature space, is defined by the equation:

$$f(\boldsymbol{x}) = 0 \tag{8.3}$$

The weight vector $\boldsymbol{w}$ is **orthogonal** to the decision boundary H.

Proof. We take two points on the decision boundary, $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$, such that:

$$f(\boldsymbol{x}_1) = f(\boldsymbol{x}_2) = 0$$

Based on equation (8.1), we can rewrite this as:

$$\boldsymbol{w}^T\boldsymbol{x}_1 + w_0 = \boldsymbol{w}^T\boldsymbol{x}_2 + w_0 = 0$$

$$\boldsymbol{w}^T\boldsymbol{x}_1 - \boldsymbol{w}^T\boldsymbol{x}_2 = 0$$

$$\boldsymbol{w}^T(\boldsymbol{x}_1 - \boldsymbol{x}_2) = 0$$

This shows that the vector $\boldsymbol{w}^T$ is orthogonal to the vector $(\boldsymbol{x}_1 - \boldsymbol{x}_2)$, which lies on the decision boundary. □

Functional margin

The value $f(\boldsymbol{x})$ of the discriminative function for a certain point $\boldsymbol{x}$ is called **functional margin**. It can be seen as the **confidence** of the model in the prediction for that point. A larger absolute value of the functional margin indicates a higher confidence in the prediction.

Geometric margin

The distance from a point $\boldsymbol{x}$ to the decision boundary H is called **geometric margin**.

$$r^x = \frac{f(\boldsymbol{x})}{||\boldsymbol{w}||} \tag{8.4}$$

The distance from the origin to the decision boundary is given by:

$$r_0 = \frac{|w_0|}{||\boldsymbol{w}||} \tag{8.5}$$

Proof. A generic point $\boldsymbol{x}$ can be expressed by its **projection** on H plus its **distance** from H times the vector in that direction:

$$\boldsymbol{x} = \boldsymbol{x}^P + r^x \frac{\boldsymbol{w}}{||\boldsymbol{w}||}$$

where:

- $\boldsymbol{x}^P$ is the projection of $\boldsymbol{x}$ on H;
- r^x is the distance from $\boldsymbol{x}$ to H;
- $\frac{\boldsymbol{w}}{||\boldsymbol{w}||}$ is the unit vector in the direction of $\boldsymbol{w}$.

Substituting this into the discriminant function:

$$f(\boldsymbol{x}) = \boldsymbol{w}^T \left(\boldsymbol{x}^P + r^x \frac{\boldsymbol{w}}{||\boldsymbol{w}||} \right) + w_0$$

$$= \boldsymbol{w}^T \boldsymbol{x}^P + w_0 + r^x w_t \frac{\boldsymbol{w}}{||\boldsymbol{w}||}$$

We can notice that $\boldsymbol{w}^T \boldsymbol{x}^P + w_0 = 0$ since $\boldsymbol{x}^P$ lies on the decision boundary H. Therefore:

$$f(\boldsymbol{x}) = r^x ||\boldsymbol{w}||$$

Rearranging this gives us the formula for the geometric margin as in equation (8.4). □

8.4 Multiclass Classification

In a **multiclass classification** problem with K classes, we can extend the linear binary classifier by using **one-vs-all** (OvA) or **one-vs-one** (OvO) strategies.

8.4.1 One-vs-All (OvA)

The idea is to train K binary classifiers, one for each class. Each classifier $f_k(\boldsymbol{x})$ is trained to distinguish class k from all other classes. This way, we have a model with k **hyperplanes**, each separating one class from the rest. At prediction time, we evaluate all K classifiers and assign the input $\boldsymbol{x}$ to the class with the maximum **functional margin**.

8.4.2 One-vs-One (OvO)

The idea is to train a binary classifier for every pair of classes. For K classes, we need to train $\frac{K(K-1)}{2}$ classifiers. The model predicts a a new example $\boldsymbol{x}$ in the class winning the most pairwise comparisons, as in a tournament.

8.4.3 Onve-vs-One (OvO) vs One-vs-All (OvA)

- OvA requires to train K classifiers, each on the entire dataset;
- OvO requires to train $\frac{K(K-1)}{2}$ classifiers, each on a smaller subset of the data (only the samples belonging to the two classes being compared);

If the number of examples is high, OvO can be more efficient, since each classifier is trained on a smaller dataset.

9 Perceptron

9.1 Biological Motivation

9.2 Single Neuron Architecture

A single neuron perceptron is a specific learning model that implements a linear binary classifier. It can be represented as:

$$f(\boldsymbol{x}) = sign(\boldsymbol{w}^T \boldsymbol{x} + w_0) \tag{9.1}$$

The function is a **linear combination** of the input features $\boldsymbol{x}$, where:

- $\boldsymbol{w}$ is the weight vector, can be seen as the **synapses** of the neuron;
- w_0 is the bias term, which allows to shift the decision boundary;
- The weighted signals are summed up and passed through an **activation function**.

9.2.1 Augmented feature / weight vector

To simplify the notation, we can introduce an **augmented feature vector**. We can rewrite the equation (9.1) as:

$$f(\boldsymbol{x}) = sign(\hat{\boldsymbol{w}}^T \hat{\boldsymbol{x}}) \tag{9.2}$$

where the **bias term** is included in the weight vector and the feature vector as follows:

$$\hat{\boldsymbol{w}} = \begin{bmatrix} w_0 \\ \boldsymbol{W} \end{bmatrix}, \quad \hat{\boldsymbol{x}} = \begin{bmatrix} 1 \\ \boldsymbol{x} \end{bmatrix}$$

In the following sections, we will skip the hat notation for simplicity.

9.2.2 Representational power

The single neuron perceptron can only represent **linearly separable** functions. Examples of this are the **primitive boolean functions** such as **AND**, **OR**, and **NOT**. This implies that any logic formula can be represented by a network of a 2 level perceptron in **disjunctive normal form** (DNF) or **conjunctive normal form** (CNF).

9.3 Parameter learning

Similar to what we saw for **maximum likelihood estimation**, we need to find a **function** of the parameters to be optimized.

In this case, a reasonable choice is to minimize the **measure of error** on the training set $\mathcal{D}$. This is called **loss function** ℓ and it compares the predicted output $f(\boldsymbol{x}_i)$ with the ground truth y_i.
We can define **training error** E as:

$$E(\boldsymbol{w}, D) = \sum_{(\boldsymbol{x},y)\in\mathcal{D}} \ell(f(\boldsymbol{x}), y) \tag{9.3}$$

9.3.1 Gradient descent

This is an optimization problem where we want to find the weights $\boldsymbol{w}$ that minimize the error function E. There is no closed-form solution for this problem, so we need to use an iterative optimization algorithm.
The common approach is to minimize the error function using a **gradient descent** approach:

1. Initialize the weights $\boldsymbol{w}$ randomly or with zeros.
2. Iterate until gradient is approximately zero, updating the weights as:

$$\boldsymbol{w} = \boldsymbol{w} - \eta\nabla E(\boldsymbol{w}, \mathcal{D}) \tag{9.4}$$

   where η is the learning rate and controls the amount of movement at each step.

This approach is guaranteed to converge to a local optimum of the error function $E(\boldsymbol{w}; D)$ for small enough learning rates η.

- too **low** learning rate: slow convergence;
- too **high** learning rate: the algorithm may **diverge** (oscillate around the minimum).

Perceptron training rule

In order to apply a **gradient descent** approach, we need to define a **loss function** which is **differentiable** (smooth). As a consequence, we adopt a different training rule, called **perceptron training rule**. It's defined as follows:

$$E(\boldsymbol{w}; \mathcal{D}) = -\sum_{(\boldsymbol{x},y)\in\mathcal{D}_E} -yf(\boldsymbol{x}) \tag{9.5}$$

where $\mathcal{D}_E$ is the set of **misclassified** samples in the training set $\mathcal{D}$, for which:

$$yf(\boldsymbol{x}) \leq 0$$

The error is the **sum** of the **functional margins** of the misclassified samples. Applying gradient descent to the error function in (9.5), we obtain:

$$\begin{aligned}\nabla E(\boldsymbol{w};\mathcal{D}) &= \nabla \sum_{(\boldsymbol{x},y)\in\mathcal{D}_E} -yf(\boldsymbol{x}) \\ &= \nabla \sum_{(\boldsymbol{x},y)\in\mathcal{D}_E} -y(\boldsymbol{w}^T\boldsymbol{x}) \\ &= \sum_{(\boldsymbol{x},y)\in\mathcal{D}_E} -y\boldsymbol{x}\end{aligned}$$

Thus, the amount of change in the weights is:

$$-\eta\nabla E(\boldsymbol{w};\mathcal{D}) = \eta \sum_{(\boldsymbol{x},y)\in\mathcal{D}_E} y\boldsymbol{x}$$

9.3.2 Stochastic Gradient Descent

In practice, the **batch gradient descent** approach is rarely used, since it requires to compute the gradient over the entire training set $\mathcal{D}$ at each iteration. A more common approach is to use **stochastic gradient descent** (SGD). It works as follows:

1. Initialize the weights $\boldsymbol{w}$ randomly;
2. Iterate until all examples are correctly classified:
   (a) For each incorrectly classified example $(\boldsymbol{x}, y)$ in the training set $\mathcal{D}$, update the weights as:
   $$\boldsymbol{w} = \boldsymbol{w} + \eta y\boldsymbol{x} \tag{9.6}$$

With this approach, the weights are updated after each misclassified example, rather than after processing the entire training set. Each gradient step is **very fast** and can sometimes avoid local minima.

9.4 Perceptron Regression

Linear models can also be used for **regression** tasks, where the goal is to predict a continuous output variable y given an input feature vector $\boldsymbol{x}$. The problem can be modeled as:

- Let $X \in \mathbb{R}^{n\times d}$ be the input training matrix (i.e. $X = [\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n]^T$), where each row is a d-dimensional feature vector and $n = |\mathcal{D}|$;
- Let $\boldsymbol{y} \in \mathbb{R}^n$ be the output training Matrix (i.e. $\boldsymbol{y}_i$ is the target value for the i-th training example);

- **Regression learning** can be stated as a set of **linear equations**:

$$X\boldsymbol{w} = \boldsymbol{y}$$

Giving as a solution:

$$\boldsymbol{w} = X^{-1}\boldsymbol{y} \tag{9.7}$$

However this approach has a few problems:

- The matrix X is usually not square, so the inverse X^{-1} is not defined;
- System of equations is overdetermined (more equations than unknowns) $\implies$ no exact solution exists.

9.4.1 Error function

Our goal is not to solve the system exactly, but to find the weights $\boldsymbol{w}$ that minimize the **mean squared error** (MSE) on the training set:

$$E(\boldsymbol{w}; \mathcal{D}) = \sum_{(\boldsymbol{x},y)\in\mathcal{D}} (y - f(\boldsymbol{x}))^2 = (\boldsymbol{y} - X\boldsymbol{w})^T(\boldsymbol{y} - X\boldsymbol{w}) \tag{9.8}$$

This is a **convex** function, so it has a single global minimum with a closed-form solution. **Gradient descent** can still be faster than computing the closed-form solution, especially for large datasets.

9.4.2 Closed form solution

To find the weights $\boldsymbol{w}$ that minimize the mean squared error, we can set the gradient of the error function to zero and solve for $\boldsymbol{w}$:

$$\nabla E(\boldsymbol{w}; \mathcal{D}) = \nabla(\boldsymbol{y} - X\boldsymbol{w})^T(\boldsymbol{y} - X\boldsymbol{w})$$

$$2(\boldsymbol{y} - X\boldsymbol{w})^T(-X) = 0$$

$$-2\boldsymbol{y}^T X + 2\boldsymbol{w}^T X^T X = 0$$

Rearranging the terms, we get:

$$\boldsymbol{w}^T X^T X = \boldsymbol{y}^T X$$

Taking the transpose of both sides:

$$X^T X\boldsymbol{w} = X^T\boldsymbol{y}$$

Assuming that $X^T X$ is invertible, we can solve for $\boldsymbol{w}$:

$$\boldsymbol{w} = (X^T X)^{-1} X^T\boldsymbol{y} \tag{9.9}$$

NOTE:

- $(X^T X)^{-1} X^T$ is a pseudoinverse called the **left inverse** of X;
- If X is square, than the left inverse is equal to the regular inverse X^{-1}, and this correspomonds to the exact solution of the system of equations;
- The left inverse exists only if the columns of X are linearly independent. If this is not the case, we can use **regularization** techniques to make the matrix invertible.

9.4.3 Gradient descent

In case of large datasets, computing the closed-form solution can be computationally expensive. In such cases, we can use **gradient descent** to iteratively update the weights until convergence.

$$\begin{aligned}
\frac{\partial E}{\partial \boldsymbol{w}_i} &= \frac{\partial}{\partial \boldsymbol{w}_i} \frac{1}{2} \sum_{(\boldsymbol{X}, y) \in \mathcal{D}} (y - f(\boldsymbol{x}))^2 \\
&= \frac{1}{2} \sum_{(\boldsymbol{X}, y) \in \mathcal{D}} \frac{\partial}{\partial \boldsymbol{w}_i} (y - f(\boldsymbol{x}))^2 \\
&= \frac{1}{2} \sum_{(\boldsymbol{X}, y) \in \mathcal{D}} 2(y - f(\boldsymbol{x})) \frac{\partial}{\partial \boldsymbol{w}_i} (y - \boldsymbol{w}^T \boldsymbol{x}) \\
&= \sum_{(\boldsymbol{X}, y) \in \mathcal{D}} (y - f(\boldsymbol{x}))(-x_i)
\end{aligned}$$

10 Support Vector Machines

Support Vector Machines (SVMs) are a very popular method for linear (and non-linear) classification and regression tasks.

- **Large Margin Classifier:** SVMs aim to find the hyperplane that maximizes the margin between different classes in the feature space;
- **Support Vectors:** the subset of data points that influence the position of the decision boundary. We will see how they are determined during the training process;
- **Sound Theoretical Foundation:** the **large margins** help to improve generalization performance;
- **Kernel machines:** SVMs can be extended to non-linear decision boundaries using kernel functions.

10.1 Maximum Margin Classifier

SVMs learns the **central hyperplane** that separates the classes with the **largest margin**. Intuitively, the margin is the distance between the hyperplane and the closest data points from each class (the **support vectors**). To formally define this concept, we first need to introduce the notion of **classifier margin**.

Classifier Margin

Given a training set $\mathcal{D}$, a classifier confidence margin is:

$$\rho = \min_{(\boldsymbol{X},y)\in\mathcal{D}} yf(\boldsymbol{x}) \tag{10.1}$$

This is the minimal confidence of the classifier ($yf(\boldsymbol{x})$) in a correct prediction. It has to be positive for all training points to be correctly classified. The geometric margin is the same value rescaled by the norm of the weight vector:

$$\frac{\rho}{||\boldsymbol{w}||} = \min_{(\boldsymbol{X},y)\in\mathcal{D}} \frac{yf(\boldsymbol{x})}{\|\boldsymbol{w}\|}. \tag{10.2}$$

Intuitively, we want to learn a classifier that learns correct predictions with high confidence (large margin). To do this we need to remove the ambiguity in the definition of the hyperplane.

Canonical Hyperplane

There are infinitely many equivalent formulation for the same hyperplane:

$$\boldsymbol{w}^T\boldsymbol{x} + w_0 = 0$$

$$\alpha(\boldsymbol{w}^T\boldsymbol{x} + w_0) = 0 \; \forall\alpha \neq 0$$

To remove this ambiguity, we can define the **canonical hyperplane** by imposing the constraint that the closest points to the hyperplane satisfy:

$$\rho = \min_{(\boldsymbol{X},y)\in\mathcal{D}} yf(\boldsymbol{x}) = 1$$

Its geometric margin then becomes:

$$\frac{\rho}{||\boldsymbol{w}||} = \frac{1}{||\boldsymbol{w}||}$$

10.2 Hard Margin SVM

Hard Margin SVMs assume that the training data is linearly separable and that we can find a hyperplane that classifies **all training points correctly**. Let's see how we can formulate this as a learning problem.

10.2.1 Learning problem

The learning objective of SVMs is to find the canonical hyperplane that maximizes the margin, which is equivalent to minimizing $||\boldsymbol{w}||$ since the margin is inversely proportional to it. This leads to the following optimization problem:

$$\min_{\boldsymbol{w},w_0} \frac{1}{2}||\boldsymbol{w}||^2 \tag{10.3}$$

subject to the constraints:

$$y_i(\boldsymbol{w}^T\boldsymbol{x}_i + w_0) \geq 1, \quad \forall(\boldsymbol{x}_i, y_i) \in \mathcal{D}$$

The constraint guarantees that all points are correctly classified with a margin of at least 1. This is a convex **quadratic optimization**, with linear constraints and can be solved efficiently using Lagrange multipliers. Let's first see how to handle this kind of constrained optimization problems in general.

10.2.2 Karush-Kuhn-Tucker (KKT) Approach

A constrained optimization problem can be addressed by converting it into an unconstrained one with the same solution.
Let's take in example the following general optimization problem:

$$\min_z f(z)$$

subject to the constraint:

$$g_i(z) \geq 0, \quad \forall i$$

We can introduce a non-negative Lagrange multiplier α_i for each constraint $g_i(z)$, and define the **Lagrangian** function:

$$L(z, \boldsymbol{\alpha}) = f(z) - \sum_i \alpha_i g_i(z)$$

The solution of the original constrained problem can be found by solving the following **saddle point** problem:

$$min_z max_{\boldsymbol{\alpha}\geq 0} L(z, \boldsymbol{\alpha})$$

The optimal solutions z^* for this problem are the same as the one from the **original constrained problem**.

- If, for a given $\hat{z}$, a constraint $g_i(\hat{z})$ is **not satisfied** (i.e., $g_i(\hat{z}) < 0$), maximizing over α_i will lead to an **infinite value**. In fact, we are subtracting a negative value multiplied by an unbounded positive value, which can grow indefinitely;
- If all constraints are satisfied, maximizing over $\boldsymbol{\alpha}$ will set all the elements of the sum to zero, so that $\hat{z}$ is a valid solution of the original problem.

10.2.3 Lagrangian of the SVM problem

Applying this to the SVM optimization problem (10.3), we can define the relative Lagrangian:

$$L(\boldsymbol{w}, w_0, \boldsymbol{\alpha}) = \frac{1}{2}||\boldsymbol{w}||^2 - \sum_{i=1}^{m} \alpha_i (y_i(\boldsymbol{w}^T \boldsymbol{x}_i + w_0) - 1) \tag{10.4}$$

where:

- $\alpha_i \geq 0$ are the Lagrange multipliers associated to each constraint;
- $m = \mathcal{D}$ is the number of training examples. We have one constraint per training example.

The lagrangian is minimized with respect to $\boldsymbol{w}$ and w_0, and maximized with respect to the non-negative multipliers α_i.
We now try to solve the **Lagrangian equivalent form**. To do this, we first compute the partial derivatives of L with respect to $\boldsymbol{w}$ and w_0, and set them to zero.
Starting with $\boldsymbol{w}$:

$$\frac{\partial L}{\partial \boldsymbol{w}} = \frac{\partial}{\partial \boldsymbol{w}} \left(\frac{\boldsymbol{W}^T \boldsymbol{w}}{2} \right) - \frac{\partial}{\partial \boldsymbol{w}} \sum_{i=1}^{m} \alpha_i y_i \boldsymbol{w}^T \boldsymbol{x}_i$$

$$= \boldsymbol{w} - \sum_{i=1}^{m} \alpha_i y_i \boldsymbol{x}_i$$

Setting this to zero gives:

$$\boldsymbol{w} = \sum_{i=1}^{m} \alpha_i y_i \boldsymbol{x}_i \tag{10.5}$$

Similarly, for w_0:

$$\frac{\partial L}{\partial w_0} = -\sum_{i=1}^{m} \alpha_i y_i$$

Setting this to zero gives:

$$\sum_{i=1}^{m} \alpha_i y_i = 0 \tag{10.6}$$

We can now substitute these expressions back into the Lagrangian (10.4) to eliminate the parameters $\boldsymbol{w}$ and w_0. This gives us the **dual formulation** of the SVM optimization problem:

$$\frac{1}{2} \sum_{i,j=1}^{m} \alpha_i \alpha_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j - \sum_{i,j=1}^{m} \alpha_i \alpha_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j - \underbrace{\sum_{i=1}^{m} \alpha_i y_i}_{=0} w_0 + \sum_{i=1}^{m} \alpha_i =$$

Applying the simplifications, our objective becomes:

$$L(\alpha) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{m} \alpha_i \alpha_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j \tag{10.7}$$

We did get rid of the primal parameters $\boldsymbol{w}$ and w_0, and now we have to maximize $L(\alpha)$ with respect to the multipliers α_i.
Our new optimization problem is:

$$\begin{aligned} & max_{\alpha \in \mathbb{R}^m} \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{m} \alpha_i \alpha_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j \\ & \text{subject to} \begin{cases} \sum_{i=1}^{m} \alpha_i y_i = 0 \\ \alpha_i \geq 0, \quad \forall i \end{cases} \end{aligned} \tag{10.8}$$

This is a **dual formulation** of the original SVM optimization problem, since it is expressed in terms of the Lagrange multipliers α_i instead of the primal parameters $\boldsymbol{w}$ and w_0. It is also a convex quadratic optimization problem, but the constraints are now much simpler than in the primal formulation.

We can now rewrite the **decision function** of the SVM in terms of the Lagrange multipliers:

$$f(\boldsymbol{x}) = \boldsymbol{w}^T \boldsymbol{x} + w_0 = \sum_{i=1}^{m} \alpha_i y_i \mathbf{x}_i^T \mathbf{x} + w_0 \tag{10.9}$$

10.2.4 Observations

At this point, we can make some important observations. At the **optimal point** each **constraint** should be equal to zero. This can happen in two cases:

1. If $\alpha_i = 0$, then the corresponding data point does not contribute to the decision function;
2. If $\alpha_i > 0 \implies y_i(\boldsymbol{w}^T \boldsymbol{x}_i + w_0) - 1 = 0 \implies y_i(\boldsymbol{w}^T \boldsymbol{x}_i + w_0) = 1$, meaning that the data point lies exactly on the margin boundary.

The points that satisfy the second condition are called **support vectors**, as they are the only points that influence the position of the decision boundary. All the other points have $\alpha_i = 0$ and do not affect the decision function.

The bias term w_0 can be computed using any of the support vectors, using the condition that they lie on the margin boundary. This is usually done by averaging the values obtained from all support vectors to improve numerical stability.

10.3 Soft Margin SVM

In practice, there is one big problem with the assumption made by hard margin SVMs: **assuming that all data is correctly classified** is often unrealistic and can lead to overfitting.
To address this issue, we introduce the concept of **soft margin SVMs**, which allow for some misclassifications in the training data. This is done by introducing **slack variables** to the optimization problem.

Slack variables

Slack variables $\xi_i \geq 0$ are introduced for each training point to allow for some misclassification.

They represent the **penalty** for the example i being on the wrong side of the margin.

The **sum of the slack variables** $\sum_{i=1}^{m} \xi_i$ has to be minimized along with the norm of the weight vector to find a balance between maximizing the margin and minimizing the misclassification error.

10.3.1 Learning problem

In the soft margin SVM, we still want to maximize the margin (minimize the norm of $\boldsymbol{w}$), but we also want to penalize the misclassifications. This can be done by adding **slack variables** to the constraints, leading to the following optimization problem:

$$\begin{aligned} &min_{\boldsymbol{w}\in X, w_0\in\mathbb{R}, \boldsymbol{\xi}\in\mathbb{R}^m} \frac{||\boldsymbol{w}||^2}{2} + C\sum_{i=1}^{m}\xi_i \\ &\text{subject to} \begin{cases} y_i(\boldsymbol{w}^T\boldsymbol{x}_i + w_0) \geq 1 - \xi_i, & \forall i \\ \xi_i \geq 0, & \forall i \end{cases} \end{aligned} \tag{10.10}$$

where $C > 0$ is a regularization parameter that controls the **trade-off** between **maximizing the margin** and **minimizing the misclassification error**.

10.3.2 Regularization theory

The problem in (10.10) can be interpreted from a regularization perspective. The objective function can be seen as a combination of two terms:

- The first term, $\frac{||\boldsymbol{w}||^2}{2}$, encourages the model to have a small norm, which corresponds to a large margin;
- The second term, $C\sum_{i=1}^{m}\xi_i$, penalizes the misclassifications, with C controlling the strength of this penalty.

This can be seen a a **Regularized loss minimization**:

$$\min_{\boldsymbol{w}\in X, w_0\in\mathbb{R}, \boldsymbol{\xi}\in\mathbb{R}^m} \frac{||\boldsymbol{w}||^2}{2} + C\sum_{i=1}^{m} L(y_i, f(\boldsymbol{x}_i)) \tag{10.11}$$

We need to define a suitable loss function $L(y_i, f(\boldsymbol{x}_i))$ that captures the misclassification error. Considering the constraints in (10.10), we can define the **hinge loss**.

Hinge Loss

The hinge loss is defined as:

$$\begin{aligned} L(y_i, f(\boldsymbol{x}_i)) &= |1 - y_i f(\boldsymbol{x}_i)|_+ \\ &= |1 - y_i(\boldsymbol{w}^T\boldsymbol{x}_i + w_0)|_+ \\ &= \max(0, 1 - y_i(\boldsymbol{w}^T\boldsymbol{x}_i + w_0)) \end{aligned} \tag{10.12}$$

10.3.3 Lagrangian of the Soft Margin SVM

We can now derive the Lagrangian for the soft margin SVM optimization problem (10.10). The Lagrangian is defined as:

$$L = \frac{1}{2}||\boldsymbol{w}||^2 + C\sum_{i=1}^{m}\xi_i - \sum_{i=1}^{m}\alpha_i(y_i(\boldsymbol{w}^T\boldsymbol{x}_i + w_0) - 1 + \xi_i) - \sum_{i=1}^{m}\beta_i\xi_i \tag{10.13}$$

In this case the primal variables are $\boldsymbol{w}$, w_0 and $\boldsymbol{\xi}$, while the dual variables are the Lagrange multipliers $\alpha_i \geq 0$ and $\beta_i \geq 0$.
We can now compute the partial derivatives of L with respect to the primal variables and set them to zero.

- Starting with $\boldsymbol{w}$:

  $$\frac{\partial L}{\partial \boldsymbol{w}} = \boldsymbol{w} - \sum_{i=1}^{m}\alpha_i y_i \boldsymbol{x}_i$$

  Setting this to zero, we get the same expression as in the hard margin case:

  $$\boldsymbol{w} = \sum_{i=1}^{m}\alpha_i y_i \boldsymbol{x}_i$$

- For w_0:

  $$\frac{\partial L}{\partial w_0} = -\sum_{i=1}^{m}\alpha_i y_i$$

  Setting this to zero, we again get:

  $$\sum_{i=1}^{m}\alpha_i y_i = 0$$

- Finally, for ξ_i:

$$\frac{\partial L}{\partial \xi_i} = C - \alpha_i - \beta_i$$

  Setting this to zero, we get:

$$\alpha_i + \beta_i = C$$

Substituting these expressions back into the Lagrangian (10.13), we obtain the dual formulation of the soft margin SVM optimization problem:

$$\begin{aligned} &max_{\alpha \in \mathbb{R}^m} \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{m} \alpha_i \alpha_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j \\ &\text{subject to} \begin{cases} \sum_{i=1}^{m} \alpha_i y_i = 0 \\ 0 \leq \alpha_i \leq C, \quad \forall i \end{cases} \end{aligned} \tag{10.14}$$

10.3.4 Observations

At the saddle point, it must hold that:

$$\begin{cases} \alpha_i (y_i (\boldsymbol{w}^T \boldsymbol{x}_i + w_0) - 1 + \xi_i) = 0 \quad \forall i \\ \beta_i \xi_i = 0 \quad \forall i \end{cases} \tag{10.15}$$

Thus, **support vectors** ($\alpha_i > 0$) are examples for which:

$$y_i (\boldsymbol{w}^T \boldsymbol{x}_i + w_0) \leq 1 \tag{10.16}$$

We can distinguish some cases:

- If $0 < \alpha_i < C$, then $C - \alpha_i - \beta_i = 0$ and $\beta_i \xi_i = 0$ imply that $\xi_i = 0$ and the point lies exactly on the margin boundary: $y_i (\boldsymbol{w}^T \boldsymbol{x}_i + w_0) = 1$;
- If $\alpha_i = C$, then $\xi_i > 0$ and the point is either inside the margin or misclassified: $y_i (\boldsymbol{w}^T \boldsymbol{x}_i + w_0) < 1$;
- If $\alpha_i = 0$, then $\xi_i = 0$ and the point is correctly classified and outside the margin. It doesn't count

10.4 Large Scale SVM

When considering large datasets, the standard SVM training algorithms can become computationally expensive due to the quadratic programming involved. To address this, several large-scale SVM training methods have been developed, such as **stochastic gradient descent (SGD)**.
When using SGD, we can reformulate the SVM optimization problem as:

$$\min_{\boldsymbol{w} \in X} \frac{1}{2} ||\boldsymbol{w}||^2 + \frac{1}{m} \sum_{i=1}^{m} |1 - y_i \langle \boldsymbol{w}, \boldsymbol{x}_i \rangle|_+ \tag{10.17}$$

Using SGD, we can iteratively update the weight vector $\boldsymbol{w}$ based on the gradient of the loss function computed on a single training example:

$$E(\boldsymbol{w}; \boldsymbol{x}_i, y_i) = \frac{\lambda}{2}||\boldsymbol{w}||^2 + |1 - y_i\langle \boldsymbol{w}, \boldsymbol{x}_i\rangle|_+ \tag{10.18}$$

There is a problem with this approach: the hinge loss is not differentiable at the point where $1 - y_i\langle \boldsymbol{w}, \boldsymbol{x}_i\rangle = 0$. To handle this, we can use the concept of **subgradients**, which generalizes the notion of gradients to non-differentiable functions. The gradients then become:

$$\nabla_{\boldsymbol{w}} E(\boldsymbol{w}; \mathbf{x}_i, y_i) = \lambda \boldsymbol{w} - \mathbb{I}[y_i\langle \boldsymbol{w}, \mathbf{x}_i\rangle l1] y_i \mathbf{x}_i \tag{10.19}$$

where $\mathbb{I}[\cdot]$ is the indicator function which can be written as:

$$\mathbb{I}[y_i\langle \boldsymbol{w}, \boldsymbol{x}_i\rangle l1] = \begin{cases} 1 & \text{if } y_i\langle \boldsymbol{w}, \boldsymbol{x}_i\rangle l1 \\ 0 & \text{otherwise} \end{cases}$$

10.5 Margin Error Buond

The performance of SVMs can be theoretically analyzed using the concept of **margin error bounds**. The probability of misclassification on unseen data is bound from above by:

$$v + \sqrt{\frac{c}{m}\left(\frac{R^2 \bigwedge^2}{\rho^2} \ln^2 m + \ln(\frac{1}{\delta})\right)} \tag{10.20}$$

Analyzing the equetion, we can see that the **probability of error** depends on:

- **number of margin errors** v on the training set (example inside the margin or misclassified);
- **number of training examples** m (error depends on $\sqrt{\frac{\ln^2 m}{m}}$);
- **size of the margin** ρ (error depends on $\frac{1}{\rho^2}$);

11 Kernel Machines

With what we have seen so far about SVMs, we find out that:

- **Hard Margin SVMs** can address linearly separable data only;
- **Soft Margin SVMs** can address linearly separable problems with some degree of misclassification;
- **Non-linearly separable problems** need higher expressive power to be addressed.

Kernel Machines are an extension of SVMs that can address non-linearly separable problems, mantaining the same underlying principles of SVMs (large margin and theoretical guarantees).
The key idea is to *map* the input data into a higher dimensional space, where it is more likely to be linearly separable and perform a linear separation in that space.

11.1 Feature Map

Feature Map

$$\phi : \mathcal{X} \to \mathcal{H}$$

A feature map ϕ is a function that maps the input space $\mathcal{X}$ to a higher dimensional (possibly infinite-dimensional) feature space $\mathcal{H}$.

All the examples $\boldsymbol{x}_i$ are replaced by their images $\phi(\boldsymbol{x}_i)$ in the feature space. This should increase the expressive power of the representation, introducing features that are combinations of the original inputs.

An example of feature map is the following:

Polynomial Feature Map

- **Homogeneous Polynomial Feature Map** of degree 2:

$$\phi\left(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}\right) = \begin{pmatrix} x_1^2 \\ x_1 x_2 \\ x_2^2 \end{pmatrix} \tag{11.1}$$

  This feature map takes a 2D input vector $\boldsymbol{x} = (x_1, x_2)$ and maps it to a 3D feature space by including polynomial terms of degree 2.

- **Non-Homogeneous Polynomial Feature Map** of degree 2:

$$\phi\left(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}\right) = \begin{pmatrix} x_1 \\ x_2 \\ x_1^2 \\ x_1 x_2 \\ x_2^2 \end{pmatrix} \tag{11.2}$$

This feature map is similar to the previous one but includes a bias term (constant 1) to account for non-homogeneous polynomials.

11.2 Linear separation in feature space

Once the data is mapped into the feature space using the feature map ϕ, we can apply SVM algorithms, replacing $\boldsymbol{x}$ with $\phi(\boldsymbol{x})$:

$$f(x) = \boldsymbol{w}^T \phi(\boldsymbol{x}) + w_0 \tag{11.3}$$

A linear separation in the feature space corresponds to a non-linear separation in the original input space.
E.g. using the homogeneous polynomial feature map of degree 2, the decision function becomes:

$$\phi\left(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}\right) = sgn(w_1 x_1^2 + w_2 x_1 x_2 + w_3 x_2^2 + w_0)$$

The linear separator in the $3D$ feature space corresponds to a ellipse in the original $2D$ input space.

11.3 Kernel trick

Computing the feature map $\phi(\boldsymbol{x})$ explicitly can be computationally expensive. In the **dual formulation** of SVMs, the data points appear only in the form of **dot products** $\langle \phi(\boldsymbol{x}), \phi(\boldsymbol{x}') \rangle$.

Kernel trick

The **kernel trick** consists in replacing the dot product in the feature space with an equivalent **kernel function**

$$\langle \phi(\boldsymbol{x}), \phi(\boldsymbol{x}') \rangle = \phi(\boldsymbol{x})^T \phi(\boldsymbol{x}') = K(\boldsymbol{x}, \boldsymbol{x}') \tag{11.4}$$

The **kernel functions** uses examples in the **original input space** $\mathcal{X}$ to compute the dot product in the **feature space** $\mathcal{H}$, without explicitly computing the feature map ϕ.

11.3.1 Examples of kernels

Homogeneous Polynomial Kernel : taking in consideration the equation 11.1, we can derive the following kernel:

$$K(\boldsymbol{x}, \boldsymbol{x}') = (\boldsymbol{x}^T \boldsymbol{x}')^d \tag{11.5}$$

Setting $d = 2$, we have:

$$K(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}, \begin{pmatrix} x'_1 \\ x'_2 \end{pmatrix}) = (x_1 x'_1 + x_2 x'_2)^2$$

$$= (x_1 x'_1)^2 + 2 x_1 x_2 x'_1 x'_2 + (x_2 x'_2)^2$$

$$= \begin{pmatrix} x_1^2 \\ \sqrt{2}x_1x_2 \\ x_2^2 \end{pmatrix}^T \begin{pmatrix} x_1'^2 \\ \sqrt{2}x_1'x_2' \\ x_2'^2 \end{pmatrix}$$

This is equivalent to the dot product in the feature space defined by the homogeneous polynomial feature map of degree 2, up to a scaling factor $\sqrt{2}$.

Non-Homogeneous Polynomial Kernel : taking in consideration the equation 11.2, we can derive the following kernel:

$$K(\boldsymbol{x}, \boldsymbol{x}') = (\boldsymbol{x}^T\boldsymbol{x}' + 1)^d \tag{11.6}$$

Setting $d = 2$, we have:

$$K(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}, \begin{pmatrix} x_1' \\ x_2' \end{pmatrix}) = (x_1x_1' + x_2x_2' + 1)^2$$

$$= (x_1x_1')^2 + 2x_1x_2x_1'x_2' + (x_2x_2')^2 + 2x_1x_1' + 2x_2x_2D + 1$$

$$= \begin{pmatrix} x_1^2 \\ \sqrt{2}x_1x_2 \\ x_2^2 \\ \sqrt{2}x_1 \\ \sqrt{2}x_2 \\ 1 \end{pmatrix}^T \begin{pmatrix} x_1'^2 \\ \sqrt{2}x_1'x_2' \\ x_2'^2 \\ \sqrt{2}x_1' \\ \sqrt{2}x_2' \\ 1 \end{pmatrix}$$

This is equivalent to the dot product in the feature space defined by the non-homogeneous polynomial feature map of degree 2, up to a scaling factor $\sqrt{2}$.

11.3.2 Valid Kernels

Valid Kernel

A kernel function $K : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ is valid if it corresponds to a dot product in some feature space $\mathcal{H}$:

$$K(\boldsymbol{x}, \boldsymbol{x}') = \langle \phi(\boldsymbol{x}), \phi(\boldsymbol{x}') \rangle \tag{11.7}$$

The kernel generalizes the concept of similarity between data points in arbitrary feature spaces.

Gram Matrix

Given examples $\{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_m\}$, and a kernel function k, the **Gram matrix** K is defined as:

$$K_{ij} = k(\boldsymbol{x}_i, \boldsymbol{x}_j) \forall i, j \tag{11.8}$$

Positive definite matrix

A matrix $M \in \mathbb{R}^{m \times m}$ is positive definite if:

$$\sum_{i,j=1}^{m} c_i c_j K_{ij} \geq 0 \quad \forall \boldsymbol{c} \in \mathbb{R}^m \tag{11.9}$$

A sufficient and necessary condition for a kernel K to be valid is that the corresponding Gram matrix is positive definite for any choice of examples $\{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_m\}$. There are several ways to check if a kernel is valid, e.g.:

- Prove its positive definiteness;
- Find out a corresponding feature map ϕ (as we did before);
- Use kernel combination properties (e.g. sum, product yield valid kernels).

11.4 Basic Kernels

Basic kernels can be useful as building blocks for more complex kernels. Some common examples include:

- **Linear Kernel**:
$$K(\boldsymbol{x}, \boldsymbol{x}') = \boldsymbol{x}^T \boldsymbol{x}' \tag{11.10}$$
- **Polynomial Kernel**:
$$K_{d,c}(\boldsymbol{x}, \boldsymbol{x}') = (\boldsymbol{x}^T \boldsymbol{x}' + c)^d \tag{11.11}$$
- **Gaussian Kernel**:
$$k_\sigma(\boldsymbol{x}, \boldsymbol{x}') = \exp\left(-\frac{\|\boldsymbol{x} - \boldsymbol{x}'\|^2}{2\sigma^2}\right) = \exp\left(-\frac{\boldsymbol{x}^T \boldsymbol{x} + \boldsymbol{x}'^T \boldsymbol{x}' - 2\boldsymbol{x}^T \boldsymbol{x}'}{2\sigma^2}\right) \tag{11.12}$$
The kernel depends on a parameter σ that controls the width of the Gaussian function. The smaller the σ, the more localized the influence of each training example.

The Gaussian kernel is an example of **universal kernel**, which means that it can approximate any continuous function on a compact domain given enough data.

11.5 Kernel combinations

Basic kernels can be **combined** using various operations to create more complex kernels from simpler ones. Correctly using combination operators guarantees that the resulting kernel is still positive definite.
We will now analyze some common kernel combination methods.

11.5.1 Kernel Sum

The sum of two kernels of two kernels corresponds to the concatenation of their feature spaces:

$$K(\boldsymbol{x}, \boldsymbol{x}') = K_1(\boldsymbol{x}, \boldsymbol{x}') + K_2(\boldsymbol{x}, \boldsymbol{x}') \tag{11.13}$$

Applying the definition of kernel:

$$= \phi_1(\boldsymbol{x})^T \phi_1(\boldsymbol{x}') + \phi_2(\boldsymbol{x})^T \phi_2(\boldsymbol{x}')$$

$$= (\phi_1(\boldsymbol{x}) + \phi_2(\boldsymbol{x}))^T (\phi_1(\boldsymbol{x}') + \phi_2(\boldsymbol{x}'))$$

11.5.2 Kernel Product

The product of two kernels corresponds to the **cartesian product** of their features:

$$(k_1 \times k_2)(\boldsymbol{x}, \boldsymbol{x}') = K_1(\boldsymbol{x}, \boldsymbol{x}') K_2(\boldsymbol{x}, \boldsymbol{x}') \tag{11.14}$$

$$= \sum_{i=1}^{n} \phi_{1i}(\boldsymbol{x}) \phi_{1i}(\boldsymbol{x}') \sum_{j=1}^{m} \phi_{2j}(\boldsymbol{x}) \phi_{2j}(\boldsymbol{x}')$$

$$= \sum_{i=1}^{n} \sum_{j=1}^{m} (\phi_{1i}(\boldsymbol{x}) \phi_{2j}(\boldsymbol{x}))(\phi_{1i}(\boldsymbol{x}') \phi_{2j}(\boldsymbol{x}'))$$

$$= \sum_{k=1}^{nm} \psi_{12k}(\boldsymbol{x}) \psi_{12k}(\boldsymbol{x}') = \psi_{12}(\boldsymbol{x})^T \psi_{12}(\boldsymbol{x}')$$

where $\psi_{12}(\boldsymbol{x}) = \phi_1(\boldsymbol{x}) \times \phi_2(\boldsymbol{x})$ is the cartesian product of the two feature maps. The product can also be between kernel in different spaces.

11.5.3 Linear combination of kernels

A kernel can be rescaled by a positive constant:

$$k_\beta(\boldsymbol{x}, \boldsymbol{x}') = \beta k(\boldsymbol{x}, \boldsymbol{x}')$$

We can define a linear combination of kernels as:

$$k_{sum}(\boldsymbol{x}, \boldsymbol{x}') = \sum_{k=1}^{K} \beta_k k_k(\boldsymbol{x}, \boldsymbol{x}')$$

Since understanding what kernel works best for a given problem can be challenging, combining multiple kernels allows us to leverage the strengths of each individual kernel. With **kernel learning** the weights β_k of the combination can be learned from data, allowing the model to adaptively combine multiple kernels.

11.5.4 Kernel normalization

Kernel values can be influenced by the dimension of objects. I.E., in text classification, longer documents tend to have higher dot products simply because they contain more words.
To mitigate this effect, we can normalize the kernel:

$$\hat{K}(\boldsymbol{x}, \boldsymbol{x}') = \frac{K(\boldsymbol{x}, \boldsymbol{x}')}{\sqrt{K(\boldsymbol{x}, \boldsymbol{x})K(\boldsymbol{x}', \boldsymbol{x}')}} \tag{11.15}$$

This is a **cosine normalization**, which takes into account the angle between the feature vectors rather than their magnitudes.

11.6 Kernel on Graphs

Kernels on graphs heavily rely on the Weistfeiler-Lehman (WL) test for graph isomorphism.

11.6.1 Graph Isomorphism

To understand this algorithm, we need to define the concept of **graph isomorphism**.

Graph Isomorphism

Two graphs $G = (V, E)$ and $G' = (V', E')$ are isomorphic if there exists a bijection $f : V \to V'$ such that:

$$(u, v) \in E \iff (f(u), f(v)) \in E' \tag{11.16}$$

In a nutshell, two graphs are isomorphic if they have the same structure, but possibly different node labels.

11.6.2 Weisfeiler-Lehman Test

If isomorphism is easy to check on vectorial data, it is much more challenging on graph data. This is in fact an NP-hard problem.
The WL test is an approximate algorithm that can efficiently determine if two graphs are non-isomorphic, but it may fail to distinguish some non-isomorphic graphs (false positives).

Algorithm: We are now going to present the WL algorithm on two graphs with same node labels:

- Given two graphs $G = (V, E)$ and $G' = (V', E')$ with $|V| = |V'| = n$;
- Let $L(G) = l(v)|v \in V$ be the set of node labels in G. $l(v)$ is a function returning the label of node v;

- Let $L(G') = L(G)$;
- Let $label(s)$ be a function assigning a unique label to a string.

The algorithm is defined as follows:

Algorithm 3 Weisfeiler-Lehman Test

```
Set l_0(v) ← l(v), ∀v ∈ V
for i = 1 to n − 1 do
    for each node v ∈ G do
        Let M_i(v) = {l_{i−1}(u) | u ∈ neigh(v)}
        Concatenate the sorted labels of M_i(v) into s_i(v)
        Let l_i(v) = label(l_{i−1}(v) ⊕ s_i(v))
    end for
    If L_i(G) ≠ L_i(G'), return not isomorphic
end for
Return possibly isomorphic
```

Where:

- $M_i(v)$: the multiset of labels of the neighbors of node v at iteration i;
- $l_i(v)$: the new label assigned to node v at iteration i. It is computed by assigning a unique label to the concatenation of the previous label $l_{i-1}(v)$ and the sorted labels of its neighbors $s_i(v)$.

11.6.3 Weisfeiler-Lehman Kernel

- Let $G_1, G_2, \ldots, G_h = (V, E, l_0), (V, E, l_1), \ldots, (V, E, l_h)$ be a set of resulting graphs after applying h iterations of the WL algorithm on graph G.
- Let $k : G \times G \to \mathbb{R}$ be any kernel on graphs;

The **Weisfeiler-Lehman kernel** between two graphs G and G' is defined as:

$$K_{WL}^{h}(G, G') = \sum_{i=0}^{h} k(G_i, G_i') \tag{11.17}$$

12 Neural Networks

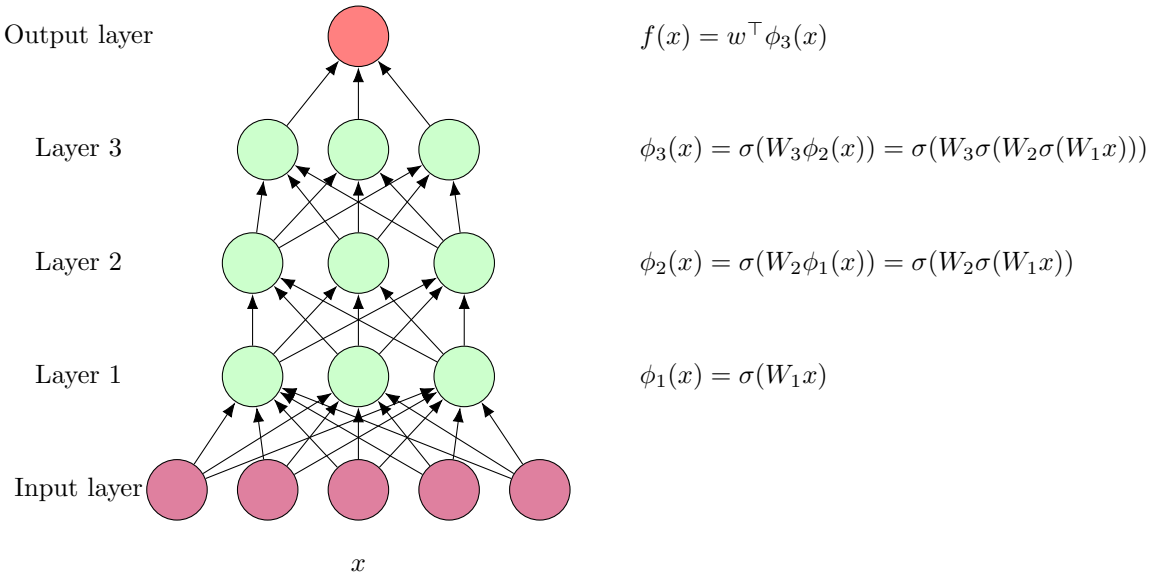
12.1 Why Neural Networks?

Neural networks are needed to resolve the limitations of other seen models. More in depth:

- **Perceptron:** can only learn linearly separable functions;
- **Kernel Machines:** add the concept of non-linearity, but the choice of the appropriate kernel is not trivial;

12.2 Multi Layer Perceptron

A Multi Layer Perceptron (MLP) is a network of interconnected neurons, organized in layers. Neuorns from one layer sends their output to the neurons of the next layer. The first layer is called *input layer*, one or mode *hidden layers* are in the middle, and the last layer is called *output layer*.



Where:

- $\boldsymbol{x}$ is the input vector;
- $\phi_i(x)$ is the output of layer i, learned from the data during training.

$$\phi_i(x) = \sigma(W_i\boldsymbol{x})$$

- W_i is the weight matrix of layer i. It's dimensions depend on the number of neurons in layer i. Each row contains the weights associated to a neuron. Computing $W_i\boldsymbol{x}$ will return a vector which contains the weighted sum of the inputs for each neuron in layer i.
- $\sigma(\cdot)$ is a non-linear activation function, applied element-wise.

The final output is computed as a linear combination in the new feature space created by the hidden layers. Differently from SVMs, where the kernel is chosen *a priori*, in MLPs the transformation is learned from data. The only downside is that training is requires more data and more computational power.

12.2.1 Activation Functions

Activation functions introduce non-linearity in the model. We will analyze some proposed functions:

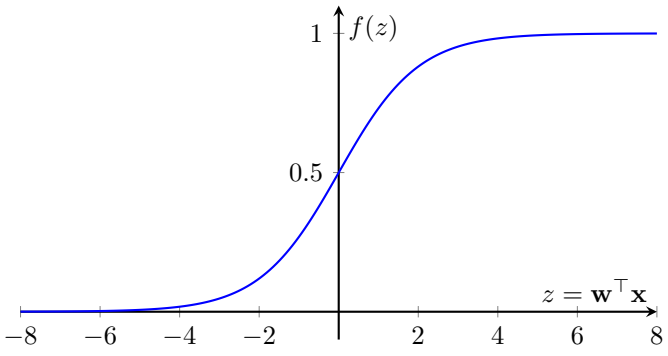
Threshold activation: the treshold activation, already seen in the perceptron model, is defined as:

$$f(\boldsymbol{x}) = sign(\boldsymbol{w}^T \boldsymbol{x})$$

It's cannot be used in MLPs, since it's derivative is zero almost everywhere, apart from the discontinuity in zero (not differentiable). This makes impossible to use gradient-based optimization methods for training.

Sigmoid activation: The sigmoid activation function is defined as:

$$f(\boldsymbol{x} = \sigma(\boldsymbol{w}^T \boldsymbol{x}) = \frac{1}{1 + e^{-\boldsymbol{w}^T \boldsymbol{x}}}$$



It represents a smooth approximation of the threshold function. It is approximately linear around zero, which helps in gradient-based optimization. However, for large positive or negative values of z, the function saturates (approaches 0 or 1), causing the gradient to vanish.

12.2.2 Output Layer

The output layer of a neural network produces the final predictions. The function used can be different from the one used in hidden layers and heavily depends on the task at hand.
We will now analyze the activation functions used in the output layer for different tasks.

Binary Classification: for binary classification tasks, only one output neuron $o(\boldsymbol{x})$ is needed. The output activation function is usually the sigmoid activation function, which maps the output to the range (0, 1). It is defined as:

$$f(\boldsymbol{x}) = \sigma(o(\boldsymbol{x})) = \frac{1}{1 + e^{-o(\boldsymbol{x})}}$$

This output can be interpreted as the probability of belonging to the positive class. To obtain the final class label, a threshold (usually 0.5) is applied:

$$y^* = sign(f(\boldsymbol{x}) - 0.5)$$

Multi-class Classification: for multi-class classification, similar to what was done with single layer perceptrons, a one vs all approach can be used. This requires K output neurons, one for each class. The softmax activation function is commonly used in this case, defined as:

$$f_i(\boldsymbol{x}) = \frac{e^{o_i(\boldsymbol{x})}}{\sum_{j=1}^{K} e^{o_j(\boldsymbol{x})}} \quad \text{for } i = 1, 2, \ldots, K$$

This function converts the raw output scores into probabilities that sum to 1 across all classes. This is not activated element-wise, but it is a layer-wise activation function. The predicted class is the one with the highest probability:

$$y^* = \arg\max_i f_i(\boldsymbol{x})$$

Regression: for regression tasks, where the output is a continuous value, the decision is the value of the output neuron itself.

12.2.3 representational power of MLPs

In this section we will analyze the representational power of MLPs on different tasks.

- **Boolean functions:** any boolean function can be represented by a MLP with two layers of units. This can be done using CNF or DNF, but it would require an exponential number of hidden units in the worst case;
- **Continuous functions:** any bounded continuous function can be approximated arbitrarily well by a MLP with two layers of units (using sigmoid activations).
- **Arbitrary functions:** any function can be approximated arbitrarily well by a MLP with three layers of units (using sigmoid activations).

12.3 Training Multi Layer Perceptrons

Similarly to other models, MLPs are trained by minimizing a loss function on the training data. Different loss functions can be used depending on the task at hand.

12.3.1 Loss Functions

Binary Classification: For binary classification tasks, the binary cross-entropy loss is commonly used. It is defined as:

$$L(y, f(\boldsymbol{x})) = -(y \log(f(\boldsymbol{x})) + (1-y)\log(1-f(\boldsymbol{x})))$$

This can be interpreted as a switch function between two log-likelihoods, depending on the true label y. By putting the minus sign, we convert the maximization problem into a minimization one (from a likelihood to a loss).

Multi-class Classification: For multi-class classification tasks, the cross-entropy loss is used. It is defined as:

$$L(y, \boldsymbol{f}(\boldsymbol{x})) = -\log f_y(\boldsymbol{x})$$

Where $f_y(\boldsymbol{x})$ is the predicted probability for the true class y.

Regression: For regression tasks, the mean squared error (MSE) loss is commonly used. It is defined as:

$$L(y, f(\boldsymbol{x})) = (y - f(\boldsymbol{x}))^2$$

12.3.2 Stochastic Gradient Descent

In order to train MLPs, we use Stochastic Gradient Descent (SGD) to minimize the chosen loss function. As example, chossing mean squared error (MSE) as loss function, we would have:

$$E(W) = \frac{1}{2}(y - f(\boldsymbol{x}))^2$$

Where W represents all the weights in the network. The relative gradient update is given by:

$$w_{lj} = w_{lj} - \eta \frac{\partial E(W)}{\partial w_{lj}}$$

Where η is the learning rate, w_{lj} is the weight connecting neuron j (source node) in the previous layer, to neuron l (target node) in the current layer. To compute the partial derivative can be quite complex when the weights are really far from the output layer, where we actually compute the loss.
To solve this problem, we use the Backpropagation algorithm, which efficiently computes the gradients using the chain rule of differentiation.

12.3.3 Backpropagation

The idea behind backpropagation is to compute the gradients layer by layer, starting from the output layer and moving backwards to the input layer. Since each neuron's output depends on the outputs of the previous layer, which in

turn depend on their own weights, we can apply the chain rule to compute the gradients. We can define the derivative as follows:

$$\frac{\partial E(W)}{\partial w_{lj}} = \underbrace{\frac{\partial E(W)}{\partial a_l}}_{\delta_l} \frac{\partial a_l}{\partial w_{lj}} = \delta_l \, \phi_j$$

where:

- a_l is the weighted sum of inputs to neuron l. It is passed to the activation function to produce the output;
- ϕ_j is the output of neuron j in the previous layer. It is one of the inputs of the node l;
- $\frac{\partial a_l}{\partial w_{lj}}$ is the input ϕ_j, since $a_l = \sum_i w_{li}\phi_i$.
- δ_l represents the error term for neuron l. Its computation depends on whether neuron l is in the output layer or in a hidden layer.

The gradient of a weight w_{lj} is given by the product of the error term

Output layers: Since neuron l is in the output layer, we can directly compute the derivative of the loss function with respect to the output of the network. Considering the MSE loss function, we have:

$$\delta_o = \frac{\partial E(W)}{\partial a_o} = \frac{\partial \frac{1}{2}(y - f(\boldsymbol{x}))^2}{\partial a_o}$$

Since we do not apply a non-linear activation function in the output layer for regression tasks, we have that $f(\boldsymbol{x}) = a_o$, so:

$$= \frac{\partial \frac{1}{2}(y - a_0)^2}{\partial a_o} = -(y - a_o)$$

Hidden layers: Here the node l is in a hidden layer. Since we compute δ_l using the chain rule, we need to consider the contributions from the neurons in the next layer.

$$\delta_l = \frac{\partial E(W)}{\partial a_l} = \sum_{k \in \text{children}(l)} \frac{\partial E(W)}{\partial a_k} \frac{\partial a_k}{\partial a_l}$$

Where k are the neurons in the next layer connected to neuron l. Intuitively, we are computing the contribution of neuron l to the final error, by considering all the paths from l to the output layer.
The term $\frac{\partial E(W)}{\partial a_k}$ is simply δ_k, already computed in the next layer. We derive that:

$$\delta_l = \sum_{k \in \text{children}(l)} \delta_k \frac{\partial a_k}{\partial a_l}$$

Remembering that $a_k = \sum_m w_{km}\phi_m$, where ϕ_m is the output of neuron m in the previous layer, we have that:

$$\frac{\partial a_k}{\partial a_l} = \frac{\partial a_k}{\partial \phi_l}\frac{\partial \phi_l}{\partial a_l}$$

The first term is simply the weight w_{kl}. The second term is the derivative of the activation function $\sigma(\cdot)$ used in neuron l. Putting everything together, we have:

$$\delta_l = \sum_{k \in \text{children}(l)} \delta_k w_{kl}\sigma(a_l)(1-\sigma(a_l))$$

Where we used the derivative of the sigmoid activation function ($\frac{\partial \sigma(x)}{\partial x} = \sigma(x)(1-\sigma(x))$).

12.4 Modular Structure of Neural Networks

Neural networks can be seen as a combination of different modules, each with a specific function. Each of these modules has an interface to communicate with other modules. Each layer j takes as input ϕ_{j-1} and it's weights W_j, and produces as output $\phi_j = F(\phi_{j-1}, W_j)$. Once we get to the output layer, we can compute the loss function and the gradients using backpropagation.

$$\frac{\partial E(W)}{\partial W_j} = \frac{\partial E(W)}{\partial \partial \phi_j}\frac{\partial \phi_j}{\partial W_j} = \frac{\partial E(W)}{\partial \phi_j}\frac{\partial F_j(\phi_{j-1}, W_j)}{\partial W_j}$$

Each layer computes:

$$\frac{\partial E(W)}{\partial \phi_{j-1}} = \frac{\partial E(W)}{\partial \phi_j}\frac{\partial \phi_j}{\partial \phi_{j-1}} = \frac{\partial E(W)}{\partial \phi_j}\frac{\partial F_j(\phi_{j-1}, W_j)}{\partial \phi_{j-1}}$$

Each module must be able to compute:

- the derivative of its output with respect to its weights:

$$\frac{\partial \phi_j}{\partial W_j} = \frac{\partial F_j(\phi_{j-1}, W_j)}{\partial W_j}$$

- the derivative of its output with respect to its input:

$$\frac{\partial \phi_j}{\partial \phi_{j-1}} = \frac{\partial F_j(\phi_{j-1}, W_j)}{\partial \phi_{j-1}}$$

12.5 Remarks on Training Neural Networks

The error surface of neural networks is non-convex. This implies that there can be multiple local minima and backpropagation is only guaranteed to find a local minimum. There are many techniques to improve the training of neural networks, that we will cover later in this section.

12.5.1 Stopping Criteria

Overtraining the network can increase the possibility of overfitting the training data. When the network is initiallized, the weights are set to small random values. In this phase, we have a very simple surface, which is easy to optimize. As the weights are updated, the surface becomes more complex, and the risk of overfitting increases. To avoid this, a separate validation set is used to monitor the performance of the network during training. When the performance on the validation set starts to degrade, training is stopped.

12.5.2 Vanishing gradients

A common problem when training deep neural networks is the vanishing gradient problem. During backpropagation, at each step the gradient is multiplied by the derivative of the activation function (sigmoid in our case). Since the saturated regions of the sigmoid have very small derivatives, the gradients can become very small as they are propagated back through the layers. A few simple suggestions to mitigate this problem are:

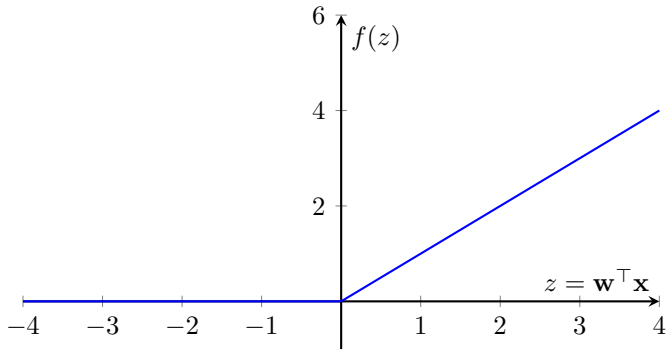
- **Weight initialization:** initialize the weights to small random values;
- **Normalization:** normalize the input data to have zero mean and unit variance ($X' = \frac{X-\mu_x}{\sigma_x}$)

Other tricks of the trade will be covered in the next paragraph.

Activations functions: Since the main problem generating vanishing gradients is the use of the sigmoid activation function, a new activation function has been proposed: the ReLU (Rectified Linear Unit). It is defined as:

$$f(x) = \max(0, \boldsymbol{w}^T \boldsymbol{x})$$

Where $\boldsymbol{w}^T \boldsymbol{x}$ represents the weighted sum of inputs to the neuron.



The ReLU activation function does not saturate for positive values, and its derivative is constant (1) in that region. This helps to mitigate the vanishing gradient problem.

Regularization: Two common regularization techniques used in training neural networks are:

- **2-norm regularization:** adding a penalty term to the loss function proportional to the euclidean norm of the weights.

  $$J(W) = E(W) + \lambda ||W||_2$$

  In essence, this encourages less relevant weights to be small, reducing overfitting.

- **1-norm regularization:** adding a penalty term to the loss function proportional to the absolute value of the weights. This encourages less relevant weights to be exactly zero.

  $$J(W) = E(W) + \lambda |W|$$

Initialization: Random initialization of weights is crucial for braking symmetry between neurons. If all weights are initialized to the same value, all neurons in a layer will learn the same features. A common approach is to initialize weights randomly, from a uniform distribution in the range:

$$W_{ij} \sim U\left(-\frac{\sqrt{6}}{\sqrt{n+m}}, \frac{\sqrt{6}}{\sqrt{n+m}}\right)$$

Where n is the number of input units and m is the number of output units.

Gradient descent:

- **Batch Gradient Descent:** updates weights after seeing the entire training set. This can be computationally expensive for large datasets;
- **Stochastic Gradient Descent:** updates weights after seeing each training example. The final objective function is noisy and may be different from the true objective;
- **Mini-batch Gradient Descent:** updates weights after seeing a small batch of training examples. The size of the batch is usually associated with the hardware capabilities (e.g., GPU memory).

Momentum: Momentum is a technique used to accelerate gradient descent and escape local minima. The idea is to update the weights using a combination of the current gradient and the previous updates, similar to a ball rolling down a hill. The update rule with momentum is given by:

$$v_{ji}^{(\text{new})} = \alpha v_{ji}^{(\text{old})} - \eta \frac{\partial E(W)}{\partial w_{ji}}$$

$$w_{ji}^{(\text{new})} = w_{ji}^{(\text{old})} + v_{ji}^{(\text{new})}$$

Where:

- α is the momentum coefficient. $0 \leq \alpha < 1$
- v_{ji} is the velocity term for weight w_{ji};

Adaptive Learning Rates: Adaptive learning rate methods adjust the learning rate for each weight individually. Some popular methods include:

- **Decreasing learning rate:** gradually decrease the learning rate over time.
$$\eta_t = \begin{cases} (1 - \frac{t}{\tau})\eta_0 + \frac{t}{\tau}\eta_t & \text{if } t \leq \tau \\ \eta_{\min} & \text{if } t > \tau \end{cases}$$
The idea is to start with a high learning rate to explore the error surface, and then decrease to avoid oscillations around minima. The learning rate is decreased linearly from η_0 to $\eta_{\min}$ over τ iterations. After that, it remains constant at $\eta_{\min}$;
- **AdaGrad:** adapts the learning rate for each weight based on the historical gradients.
$$r_{ji}^{(\text{new})} = r_{ji}^{(\text{old})} + \left(\frac{\partial E(W)}{\partial w_{ji}}\right)^2$$
$$w_{ji}^{(\text{new})} = w_{ji}^{(\text{old})} - \frac{\eta}{\sqrt{r_{ji}^{(\text{new})}}} \frac{\partial E(W)}{\partial w_{ji}}$$
Where r_{ji} accumulates the squared gradients for weight w_{ji}. This results in smaller updates for weights with large gradients (steep directions) and larger updates for weights with small gradients (flat directions); This has a drawback: the accumulated squared gradients can grow indefinitely.
- **RMSProp:** addresses the drawback of AdaGrad by using an exponentially decaying average of squared gradients.
$$r_{ji}^{(\text{new})} = \rho r_{ji}^{(\text{old})} + (1 - \rho)\left(\frac{\partial E(W)}{\partial w_{ji}}\right)^2$$
$$w_{ji}^{(\text{new})} = w_{ji}^{(\text{old})} - \frac{\eta}{\sqrt{r_{ji}^{(\text{new})} + \epsilon}} \frac{\partial E(W)}{\partial w_{ji}}$$
Where ρ allows to exponentially decay the influence of past gradients.

Batch Normalization: To mitigate the problem of internal covariate shift, batch normalization is used. It normalizes the input of each activation function on its mini-batch.
$$\hat{x_i} = \frac{x_i - \mu_B}{\sigma_B}$$
Where μ_B and σ_B are the mean and standard deviation of the mini-batch B, x is the activation of an arbitrary layer. This normalization helps to stabilize the learning process and allows for higher learning rates.

12.6 Popular architectures

Different architectures of neural networks have been proposed to tackle specific problems. In this section we will briefly describe some of the most popular ones.

Autoencoders

Autoencoders are a type of neural network used for unsupervised learning (learning with no labelled data). They consist of an encoder, that maps the input data to a lower-dimensional latent space, and a decoder, that reconstructs the input data from the latent representation. This is useful for dimensionality reduction, feature learning, and data compression.

Convolutional Neural Networks (CNNs)

Convolutional Neural Networks (CNNs) are specialized neural networks designed for processing grid-like data, such as images. They use convolutional filters to extract local features from the input data. Pooling layers are used to compose higher-level features from lower-level ones. After several convolutional and pooling layers, fully connected layers are used to perform the final classification.

Generativr Adversarial Networks (GANs)

Generative Adversarial Networks (GANs) learns to generate items from random noise. They consist of two neural networks: a generator, that generates new data, and a discriminator, that tries to distinguish between real and generated data. The two networks are trained simultaneously, with no supervision needed.

Transformers

Use attention mechanisms to learn input words encodings, that depends on the context of the sentence. They can also learn output words encodings, that depends on the context of the already generated words and on the input sentence.

Graph Neural Networks (GNNs)

Allow to learn from graph-structured data. Allow to propagate informations from a node to its neighbors, learning a representation that takes into account the graph structure.

13 Unsupervised Learning - Clustering

What we have seen so far are all examples of **supervised learning**, which required a labeled dataset to learn from. This process is usually extremely expensive and it could happen that we don't have access to labeled data at all.

In such cases, we can resort to **unsupervised learning** techniques, which can be used to group data points into **clusters**.

13.1 K-Means Clustering

One of the most popular clustering algorithms is **K-Means Clustering**. The setting is the following:

- We have a dataset of n points $\{x_1, x_2, \ldots, x_n\}$ in $\mathbb{R}^d$.
- We assume that examples should be grouped into k clusters, we will see later how to choose k;
- Each cluster i is represented by its **centroid** (mean) μ_i;

Algorithm

The K-Means algorithm works as follows:

1. Initialize k cluster means $\mu_1, \mu_2, \ldots, \mu_k$ (randomly or using some heuristic);
2. Repeat until convergence:
   - **Assignment step**: Assign each point x_j to the nearest cluster mean;
   - **Update step**: Update cluster means according to the assigned points;

The algorithm converges when the assignments no longer change or the cluster means stabilize.

Distance Metrics

To compute the distance between points and cluster means, we have to choose a distance metric, the most common choices are:

- Euclidean distance in $\mathbb{R}^d$;

$$d(\boldsymbol{x}, \boldsymbol{x}') = \sqrt{\sum_{i=1}^{d} (x_i - x_i')^2}$$

- Manhattan distance:

$$d(\boldsymbol{x}, \boldsymbol{x}') = \sum_{i=1}^{d} |x_i - x_i'|$$

- Cosine similarity:

$$d(\boldsymbol{x}, \boldsymbol{x}') = 1 - \frac{\boldsymbol{x}^T \boldsymbol{x}'}{||\boldsymbol{x}||\,||\boldsymbol{x}'||}$$

13.2 Quality of Clustering

Since we are working in an unsupervised setting, we don't have labels to evaluate the quality of our clustering. However, we can define the following metrics:

Sum-of-Squared error criterion

The sum-of-squared error criterion is defined as:

- let n_i be the number of points assigned to cluster i (D_i);
- let μ_i be the centroid of cluster i. It can be computed as:

$$\mu_i = \frac{1}{n_i} \sum_{\boldsymbol{x} \in D_i} \boldsymbol{x}$$

- the sum-of-squared error criterion is defined as:

$$E = \sum_{i=1}^{k} \sum_{\boldsymbol{x} \in D_i} ||\boldsymbol{x} - \mu_i||^2$$

It measures the squared error incurred in representing each point by its cluster centroid.

13.3 Gaussian Mixture Models (GMM)

The idea behind Gaussian Mixture Models is to model the data as a mixture of several Gaussian distributions. Each cluster is represented by a Gaussian distribution with its own parameters (mean and covariance). It assumes that the number of clusters k is given and that each data point is generated from one of the k Gaussian distributions.

Parameter Estimation

Maximum Likelihood Estimation (MLE) cannot be used directly, since the cluster assignments are unknown. Instead, we can apply an iterative approach called the **Expectation-Maximization (EM)**.

Expectation-Maximization (EM) Algorithm

The EM algorithm consists of two main steps:

1. **E-step (Expectation step)**: Compute the expected cluster assignments given the current parameters;
2. **M-step (Maximization step)**: Update the parameters of the Gaussians given the expected cluster assignments;
3. Repeat until convergence.

Example: estimating the mean of k univariate Gaussians

Here, we assume to have a dataset of $x_1, x_2, \ldots, x_n$ examples. For each example x_i, the cluster assignment is modelled as $z_{i1}, z_{i2}, \ldots, z_{ik}$ binary variables, in a one-hot encoding fashion ($z_{ij} = 1$ if x_i belongs to cluster j, and 0 otherwise). Parameters to estimate are the means of each Gaussian: $\mu_1, \mu_2, \ldots, \mu_k$, assuming that the variances are known and fixed.
The algorithm works as follows:

1. **Initialization**: randomly initialize the means $h = \langle \mu_1, \mu_2, \ldots, \mu_k \rangle$;
2. **Iterate until convergence**: convergence is checked by measuring the change in Maximum Log-Likelihood:
   - **E-step**: compute the expected value $E[z_{ij}]$ of each latent variable, given the current hypotesis h:

     $$E[z_{ij}] = \frac{p(x_i|\mu_j)}{\sum_{l=1}^{k} p(x_i|\mu_l)}$$

     We are basically computing the probability that x_i belongs to cluster j and normalizing it over all clusters. This way, summing over all clusters gives 1.
     Since we are working with Gaussians, we use the Gaussian probability density function to compute $p(x_i|\mu_j)$.

     $$= \frac{\exp -\frac{1}{2\sigma^2}(x_i - \mu_j)^2}{\sum_{l=1}^{k} \exp -\frac{1}{2\sigma^2}(x_i - \mu_l)^2}$$

   - **M-step**: calculate a new hypotesis h' assuming values of latent variables are their expected values. The maximum likelihood estimate of the mean μ_j is given by the weighted sample mean, where each istance is weighted by its probability of belonging to cluster j:

     $$\mu'_j = \frac{\sum_{i=1}^{n} E[z_{ij}] x_i}{\sum_{i=1}^{n} E[z_{ij}]}$$

13.4 Choose the number of clusters k

Choosing the right number of clusters k is a crucial step in clustering. Increasing the number of clusters will always improve the fit to the data, but it may lead to overfitting. We will now introduce different methods to choose k.

13.4.1 Elbow Method

The Elbow Method tries to trade-off the quality of the clustering with quantity. The number of clusters k should stop increasing when the advantage of adding another cluster is limited. The approach is composed of the following steps:

- Run the clustering algorithm for increasing values of k;
- For each k, compute the sum-of-squared error criterion $E(k)$;
- Plot $E(k)$ against k;
- Choose k at the "elbow" point, where the decrease in $E(k)$ starts to slow down.

This method is really intuitive, but can be ambiguous, since the elbow point is not always clear.

13.4.2 Silhouette Method

Increasing the number of clusters k will always decrease the sum-of-squared error criterion, but it will also increase the similarity between different clusters. The Silhouette Method tries to balance these two aspects. The silhouette score for a single point x_i is defined as:

- $a(i)$: average distance between x_i and all other points in the same cluster C:
$$a(i) = \frac{1}{|C|} \sum_{j \in C} d(i, j)$$
- $b(i)$: minimum average distance between x_i and all points in any other cluster C':
$$b(i) = \min_{C' \neq C} \frac{1}{|C'|} d(i, C')$$
- The silhouette score $s(i)$ is then defined as:
$$s(i) = \frac{b(i) - a(i)}{\max(a(i), b(i))}$$

To choose the number of clusters k, we compute the average silhouette score over all points for different values of k. As before, we plot the average silhouette score against k and choose the value of k that maximizes the score.

13.4.3 Hierarchical Clustering

Hierarchical Clustering is an alternative approach, that assumes that the data is organized in a hierarchy of clusters. This structure can be built from examples in two ways:

- **Top-down approach**: start with all data points in a single cluster and recursively split clusters into smaller clusters;
- **Bottom-up approach**: start with each data point in its own cluster and recursively merge clusters into larger clusters.

The algorithm for the Bottom-up approach is defined as follows:

- Initialize:
  - Set the target number of clusters k;
  - Set the initial number $\hat{k}$ of clusters to n;
  - Initialize clusters as individual data points: $D_i = \{x_i\}, \forall i \in [1, n]$
- Repeat until the number of clusters $\hat{k}$ is equal to k:
  - Find the two closest clusters D_i and D_j;
  - Merge the clusters: $D_i \leftarrow D_i \cup D_j$;
  - Update the number of clusters: $\hat{k} \leftarrow \hat{k} - 1$;

The similarity between clusters can be computed in different ways:

- **Nearest neighbor**: distance between the two closest points in the clusters:
$$d_{min}(D_i, D_j) = \min_{\boldsymbol{x} \in D_i, \boldsymbol{x'} \in D_j} ||\boldsymbol{x} - \boldsymbol{x'}||$$
- **Farthest neighbor**: distance between the two farthest points in the clusters:
$$d_{max}(D_i, D_j) = \max_{\boldsymbol{x} \in D_i, \boldsymbol{x'} \in D_j} ||\mathbf{x} - \boldsymbol{x'}||$$
- **Average linkage**: average distance between all points in the clusters:
$$d_{avg}(D_i, D_j) = \frac{1}{|D_i||D_j|} \sum_{\boldsymbol{x} \in D_i} \sum_{\boldsymbol{x'} \in D_j} ||\boldsymbol{x} - \boldsymbol{x'}||$$

The first two methods are sensitive to outliers, while the average linkage is more robust. All result complex to compute. A more efficient method is the **centroid method**, which computes the distance between the centroids of the clusters:

$$d_{centroid}(D_i, D_j) = ||\mu_i - \mu_j||$$

where μ_i and μ_j are the centroids of clusters D_i and D_j respectively.

14 Reinforcement Learning

Reinforcement Learning (RL) is a type of machine learning where an agent learns to make decisions by taking actions in an environment to maximize cumulative rewards.

14.1 Learning Settings

- The learner (agent) is provided with a set of possible states S, and for each state, a set of possible actions A, moving it to a new state.
- In performing action a, from state s, the agent receives an immediate reward r.
- The goal is to learn a policy $\pi : S \to A$ that maximizes the overall reward over time.
- The learner has to deal with problems of dealayed reward, where the consequences of an action may not be immediately apparent and trade-offs between exploration (trying new actions) and exploitation (choosing actions known to yield high rewards).

More formally, the setting of reinforcement learning can be modeled as a Markov Decision Process (MDP).

14.2 Markov Decision Process (MDP)

An MDP is defined by:

- A set of **states** S, in which the agent can be;
- A set of **terminal states** $S_G \subseteq S$, where the process ends. Might also be empty;
- A set of **actions** A available to the agent;
- A **transition model**, providing the probability of going to state s' when taking action a in state s:

$$P(s'|s,a); s, s' \in S, a \in A$$

- A **reward function** $R(s, a, s')$, providing the immediate reward received after transitioning from state s to state s' due to action a.

The agent's objective is to find a policy $\pi(s)$ that maximizes an utility function, taking into account the uncertainty in state transitions and rewards.

14.3 Utilities

Utilities are defined over environment histories. An environment history is a sequence of states $[s_0, s_1, s_2, ...]$, passed through by the agent. When calculating the utility, we assume an infinite horizon, meaning the agent will continue to interact with the environment indefinitely. We also assume stationary preferences, meaning the agent's preferences do not change over time.
We can define the utility in two ways:

- **Additive rewards**: values immediate and future rewards equally.

$$U([s_0, s_1, s_2, \dots]) = R(s_0) + R(s_1) + R(s_2) + \dots$$

- **Discounted utility**: prefers immediate rewards over future rewards.

$$U([s_0, s_1, s_2, \dots]) = R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \dots$$

  where $0 \leq \gamma < 1$ is the discount factor.

Here we are assuming that rewards only depend on the current state, not the action taken or the next state.

14.4 Policy

A policy π is a mapping from states to actions: $\pi : S \to A$. It defines the action that the agent should take when in a given state.

Expected Utility of a policy Given a policy π, we define the expected utility of π as the utility of an environment, taken in expectation over all possible state sequences generated by following policy π. In other words, we are multiplying the utility of that history, by the probability of that history occurring under policy π.

Optimal Policy

An optimal policy π^* is a policy that maximizes the expected utility. In other words, it is the best policy the agent can follow to achieve the highest cumulative reward over time.

Utility of states

Given a policy π, the utility of a state s is defined as:

$$U^\pi(s) = \mathbb{E}_\pi \left[\sum_{t=0}^{\inf} \gamma^t R(S_t) | S_0 = s \right]$$

Where S_t is the state reached after t steps, using policy π, starting from state $S_0 = s$. This is the expected utility of the environment history, starting from state s and following policy π.

We can define the **true utility** of a state s, under the optimal policy π^*, as:

$$U^*(s) = U^{\pi^*}(s) \tag{14.1}$$

Given the true utility of each state, we can easily derive the optimal policy by choosing the action that maximizes the expected utility:

$$\pi^*(s) = \arg\max_{a \in A} \sum_{s' \in S} P(s'|s, a) U^*(s') \tag{14.2}$$

This definition creates a circular dependency between the optimal policy and the true utility, which can be resolved using the Bellman equations.

14.4.1 Bellman Equations

The Bellman equation for state s is defined as:

$$U(s) = R(s) + \gamma \max_{a \in A} \sum_{s' \in S} P(s'|s,a)U(s')$$

This equation states that the utility of a state s is equal to the immediate reward $R(s)$ plus the discounted expected utility of the next state, assuming the agent takes the optimal action. We can define a system of Bellman equations, one for each state in the MDP. Solving this system will yield the true utilities for all states, which can then be used to derive the optimal policy. Directly solving this system can be computationally expensive, since they are non-linear equations. Instead, we can use iterative methods such as value iteration or policy iteration to approximate the solution.

14.4.2 Value Iteration

Value iteration, also known as Utility Iteration, works by initializing the utility of all states to arbitrary values (for example zero) and then repeatedly updating the utilities using the Bellman equation until convergence.

Algorithm 4 Value Iteration

1: Initialize $U_0(s) \forall s \in S$ to 0
2: **repeat**
3: Update $U_{i+1}(s)$ using the Bellman equation:

$$U_{i+1}(s) \leftarrow R(s) + \gamma \max_{a \in A} \sum_{s' \in S} P(s'|s,a)U_i(s')$$

4: $i \leftarrow i + 1$
5: **until** convergence
6: **return** $U(s)$

At each iteration, we update the utility of each state based on the current estimates of the utilities of the successor states. This process continues until the utilities converge to stable values. The policy can then be derived from the converged utilities using the formula for the optimal policy 14.4.

14.4.3 Policy Iteration

Contrary to what happens in value iteration, in policy iteration we start with an arbitrary policy and then iteratively improve it.

Algorithm 5 Policy Iteration

1: Initialize an arbitrary policy π_0 randomly
2: **repeat**
3: **Policy Evaluation**: solve the set of linear equations:

$$U_i(s) = R(s) + \gamma \sum_{s' \in S} P(s'|s, \pi_i(s)) U_i(s') \ \ \forall s \in S$$

where $\pi_i(s)$ is the action taken in state s under policy π_i.
4: **Policy Improvement**: update the policy using the new utilities:

$$\pi_{i+1}(s) = \arg\max_{a \in A} \sum_{s' \in S} P(s'|s, a) U_i(s') \ \ \forall s \in S$$

5: **until** convergence
6: **return** $\pi(s)$

In the policy evaluation step, we compute the utilities of the states under the current policy by solving a system of linear equations. Using the current policy, instead of the max operator from the Bellman equation, we made the equations linear, making it easier to solve. In the policy improvement step, we update the policy using equation 14.4, based on the newly computed utilities.

14.5 Dealing with Partial Knowledge

Value iteration and policy iteration both assume that the agent has complete knowledge of the MDP. In most real-world scenarios, the agent does not have access to the full transition model or reward function, but it aims to learn them through interaction with the environment.
There are two main approaches to deal with this partial knowledge:

- **Policy Evaluation**: the policy is given, environment is learned through experience (passive agent). The environment simply learns the transition model while following a fixed policy.
- **Policy Improvement**: both the policy and the environment are learned through experience (active agent).

14.5.1 Policy Evaluation

Adaptive Dynamic Programming (ADP)

In ADP, the agent maintains counts of useful transitions to estimate the transition model:

- $N(s, a)$: number of times action a has been taken in state s;
- $N(s'|s, a)$: number of times action a has led to state s' from state s.

Algorithm 6 Adaptive Dynamic Programming

repeat
  Initialize s
  repeat
    receive reward r, set R(s) = r
    choose action $a \leftarrow \pi(s)$
    execute action a, observe next state s'
    update counts:

$$N(s,a) \leftarrow N(s,a) + 1$$

$$N(s'|s,a) \leftarrow N(s'|s,a) + 1$$

    update transition model:

$$P(s''|s,a) = \frac{N(s''|s,a)}{N(s,a)} \quad \forall s'', s \in S, a \in A$$

    update utilities using policy evaluation
    $U \leftarrow \text{PolicyEvaluation}(\pi, U, p, R, \gamma)$
  until s is terminal
until convergence

Remark:

the outer loop is needed to ensure any significant results. Each iteration of the outer loop is called an **Episode**.

The algorithm performs maximum likelihood estimation of the transition model based on observed counts. It then updates the utilities of each state, based on the current learned transition model, using standard policy Evaluation. This approach is really expensive, since it requires computing the policy evaluation at every step.

Temporal-Difference Learning (TD Learning)

TD Learning tries to approximate the utility of states, without needing to run full policy evaluation at every step. This is done by assuming a deterministic transition model (no real transition model is learned). If the agent is in state s and takes action a, arriving in state s', we assume that s' is the only possible successor of s:

- If s' was always the successor of s, then we can update the utility of s as:

$$U(s) = R(s) + \gamma U(s')$$

- The temporal-difference updates the utility of s incrementally towards this target at each step:

$$U(s) \leftarrow U(s) + \alpha \left(R(s) + \gamma U(s') - U(s)\right)$$

Where α is the learning rate, controlling how much new information overrides old information.

Algorithm 7 TD Learning

repeat
  Initialize s
  repeat
    Receive reward r
    Choose next action $a \leftarrow \pi(s)$
    Execute action a, observe next state s'
    update local utility estimates:

$$U(s) \leftarrow U(s) + \alpha\left(r + \gamma U(s') - U(s)\right)$$

    $s \leftarrow s'$
  until s is terminal
until convergence

Differences

- TD Learning does not learn a transition model, while ADP does;
- Each step of TD Learning is computationally cheaper than ADP;
- TD Learning takes longer to converge than ADP.

We can say that TD Learning is a rough approximation of ADP.

14.5.2 Policy Improvement

As we have said before, in policy learning both the optimal policy and the environment are learned through experience.

A simple solution

A simple solution to this problem consists in using a variation of ADP, where the policy evaluation is replaced with the computation of the optimal policy (both value and policy iteration can be used).

$$U(s) = R(s) + \gamma \max_{a \in A} \sum_{s' \in S} P(s'|s,a)U(s')$$

This approach theoretically works, but it has two main issues:

- It is computationally expensive, since it requires computing the optimal policy at every step;

- This is a greedy approach, which may lead to suboptimal policies. The space is not explored enough, and the agent tend to overfit to the current knowledge of the environment.

To resolve the exploration vs exploitation trade-off, non entirely greedy algorithms should be defined, to encourage exploration of the state space.

Trade off strategies

There are two main strategies to deal with the exploration vs exploitation trade-off:

- **ϵ-greedy**: with probability $1 - \epsilon$, the agent chooses the action that maximizes the expected utility (the greedy action). With probability ϵ, the agent chooses a random action (exploration).
- **Define a new Bellman equation**: assign **higher utilities estimates** to less explored states, to encourage the agent to visit them.

  $$U^+(s) = R(s) + \gamma \max_{a \in A} f(\sum_{s' \in S} P(s'|s,a)U^+(s'), N_{sa})$$

  Where the function $f(u, n)$ returns a lower number for higher n (the state has been explored more often).

Both these strategies help with the suboptimal policies issue, but still require computing the optimal policy at every step, which is computationally expensive.

Utilities of An action:

To resolve the computational issue we could follow the idea used in section 14.5.1 and use TD Learning to approximate the utilities of states. However, TD Learning cannot be directly applied in this context. Here, we are learning both the policy and the environment.

As we have seen in equation 14.4, the optimal policy depends on the transition model, which is not computed in TD Learning.

$$\pi^*(s) = \arg\max_{a \in A} \underbrace{\sum_{s' \in S} \overbrace{P(s' \mid s, a)}^{\text{Not computed}} \; U^*(s')}_{Q(s,a)} \tag{14.3}$$

Where $Q(s, a)$ represents the utilities of taking action a in state s. Given the utilities of actions $Q(s, a)$, we can derive the optimal policy as:

$$\pi^*(s) = \arg\max_{a \in A} Q(s, a)$$

There are 2 algorithms that use this idea to compute the optimal policy: SARSA (On-policy) and Q-Learning (Off-policy).

SARSA (On-policy)

SARSA (State-Action-Reward-State-Action) is an on-policy TD learning algorithm. It chooses the next action based on a ϵ-greedy policy derived from the current utility Q. It then takes that action, to reach state s'. It uses again the ϵ-greedy policy to choose the next action a' in state s'. Since it is an on-policy

Algorithm 8 SARSA

repeat
  Initialize s
  repeat
    Receive reward r
    Choose action $a \leftarrow \pi^\epsilon(s)$
    Execute action a, observe reward r and next state s'
    Choose next action $a' \leftarrow \pi^\epsilon(s')$
    Update utility estimates:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left(r + \gamma Q(s',a') - Q(s,a)\right)$$

    $s \leftarrow s'$
    $a \leftarrow a'$
  until s is terminal
until convergence

algorithm, SARSA updates the utility action $Q(s,a)$ using the current policy π^ϵ. This means that the action a' taken in state s' is also influenced by the exploration strategy, which can lead to more conservative updates.

Q-Learning (Off-policy)

Q-Learning is an off-policy TD learning algorithm. It uses different policies for exploration and for learning the utilities of actions (the greedy policy).

14.6 Scaling to Large Problems

All the techniques we have seen so far assume a tabular representation of the utility functions. This is not feasible for large state or action spaces, since the tables would become too large to store and update efficiently.
The solution is to use function approximation techniques: we can approximate the utility functions, using parametric functions (such as neural networks). The problem now shifts to learning the parameters of these functions, instead of maintaining large tables. This can be done using gradient descent methods.

Algorithm 9 Q-Learning

repeat
  Initialize s
  repeat
    Receive reward r
    Choose action $a \leftarrow \pi^{\epsilon}(s)$
    Execute action a, observe reward r and next state s'
    Update utility estimates:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left(r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right)$$

    $s \leftarrow s'$
  until s is terminal
until convergence

14.6.1 TD learning: State Utilities

First, to do gradient descent, we need to define an error function we need to minimize.

$$E(s,s') = \frac{1}{2} \left(R(s) + \gamma U_\theta(s') - U_\theta(s) \right)^2 \tag{14.4}$$

Where $U_\theta(s)$ is the approximated utility function, parameterized by θ. This error function measures the distance between the current value function $U_\theta(s)$ and the target value $R(s) + \gamma U_\theta(s')$.
We can then compute the gradient of this error function with respect to the parameters θ:

$$\nabla_\theta E(s,s') = -\left(R(s) + \gamma U_\theta(s') - U_\theta(s) \right) \nabla_\theta U_\theta(s) \tag{14.5}$$

The stochastic gradient descent update rule for the parameters θ is then given by:

$$\theta \leftarrow \theta + \alpha \nabla_\theta E(s,s') \tag{14.6}$$

Where α is the learning rate.

14.6.2 TD learning: Action Utilities

Similarly, we can define an error function for the action utilities:

$$E(s,a,s',a') = \frac{1}{2} \left(R(s) + \gamma Q_\theta(s',a') - Q_\theta(s,a) \right)^2 \tag{14.7}$$

This error function measures the distance between the current action-value function $Q_\theta(s,a)$ and the target value $R(s) + \gamma Q_\theta(s',a')$.
The gradient of this error function with respect to the parameters θ is:

$$\nabla_\theta E(s,a,s',a') = -\left(R(s) + \gamma Q_\theta(s',a') - Q_\theta(s,a) \right) \nabla_\theta Q_\theta(s,a) \tag{14.8}$$

The stochastic gradient descent update rule for the parameters θ is then given by:

$$\theta \leftarrow \theta + \alpha \nabla_\theta E(s, a, s', a') \tag{14.9}$$

Where α is the learning rate.

15 Ensemble methods

Ensemble methods are techniques that combine multiple ML models to improve the overall performance compared to individual models. The idea is based on the principle that group of individuals can make better decisions than a single individual, if they have diverse perspectives. Ensemble strategies can be categorized into three main types:

1. Bagging: diversifies the training sets;
2. Stacking: diversifies the model architectures being trained;
3. Boosting: diversifies the weights of the single training example, focusing on the "hard" ones.

15.1 Bagging

In a nutshell, Bagging is based on this simple tasks:

- Take a learning algorithm A;
- Extract from it m different dataset $D^{(i)}$ from the original training set D, using **bootstrap resampling**;
- Train one base model on each dataset:

$$f^{(i)} = A(D^{(i)})$$

- Combine predictions of different base models:

$$\hat{y} = \text{COMBINE}(f^{(1)}(x), \ldots, f^{(m)}(x))$$

Where $\hat{y}$ is the final prediction. The various methods to combine will be discussed later.

15.1.1 Bootstrap resampling

The easiest way to create different datasets from the original would be to split it into m disjoint subsets. This would however reduce the amount of data available for training each model, which is not ideal.
Bootstrap resampling instead extracts $N = |D|$ samples from D **with replacement**, meaning that the same sample can be extracted multiple times.

This procedure is then repeated m times to create m different datasets, of the same size as D.
We can show that each bootstrap sample contains on average 63% of the unique samples from the original dataset.

Proof. Consider a single sample from the original dataset. The probability that it is selected in one draw is:

$$P(\text{selected}) = \frac{1}{N}$$

We derive that the probability that it is not selected in one draw is:

$$P(\text{not selected}) = 1 - \frac{1}{N}$$

Since we draw N times (with replacement), the probability that it is never selected in the bootstrap sample is:

$$P(\text{never selected}) = \left(1 - \frac{1}{N}\right)^N$$

As N approaches infinity, this expression converges to $e^{-1} \approx 0.3679$. □

The instances that are not selected in a bootstrap sample are called **out-of-bag** (OOB) instances, and can be used to estimate the test performance of the base model.

15.1.2 Combining methods

There are 3 main methods to combine the predictions of the base models:

- **Majority voting**: for classification tasks, the final prediction is the class with the most votes:

  $$\hat{y} = \arg\max_y \sum_{i=1}^{m} \delta(y, \hat{y}^{(i)})$$

  where $\delta(a, b)$ is 1 if $a = b$, 0 otherwise, and $\hat{y}^{(i)}$ is the prediction of the i-th base model;

- **Soft voting**: for multiclass classification tasks, the final prediction is the class with the highest average predicted probability. This assumes that the base models can output class probabilities.

  $$\hat{y} = \arg\max_y \frac{1}{m} \sum_{i=1}^{m} f_y^{(i)}(x)$$

  where $f_y^{(i)}(x)$ is the predicted probability of class y for the i-th base model;

- **Mean**: for regression tasks, predict the mean of the based models' predictions:

$$\hat{y} = \frac{1}{m} \sum_{i=1}^{m} \hat{y}^{(i)}$$

Given this, we can now formally define **Random Forests**. A Random Forest is an ensemble of decision trees trained using bagging, which introduce stochasticity not only by bootstrap resampling the data, but also by randomly selecting a subset of features at each split in the tree.

15.2 Stacking

Stacking is an ensemble method that combines multiple different base models by training a meta-model to make final predictions based on the outputs of the base models. The process can be summarized in the following steps:

- Train multiple base models $f^{(i)}$ on the original training set D, using m different algorithms $A^{(i)}$:

$$f^{(i)} = A^{(i)}(D)$$

- Use a **meta-learner** to learn a combination of the base models' predictions (e.g., a linear combination):

$$g = A_{\text{META}}([f^{(1)}, \ldots, f^{(m)}], D')$$

- Use the meta-model to make final predictions:

$$\hat{y} = g([f^{(1)}(x), \ldots, f^{(m)}(x)])$$

The **meta-model** should be trained on a separate dataset D' or it will simply focus on learning the best performing model on D.

15.3 Boosting

Boosting is an ensemble method that combines multiple weak learners to create a strong learner. The key idea is to train models sequentially, with each new model focusing on the errors made by the previous ones. The process can be summarized in the following steps:

- Take a learner A and train it on D;
- Reweight examples in D based on their accuracy according to the trained model (incorrectly classified examples get higher weights);
- Train A again on the reweighted dataset;
- Repeat the procedure for m times;
- Combine the learned models into the final model.

This procedure is particullarly effective with weak learners.

Weak learner

A weak learner is a model that performs slightly better than random guessing. A weak learner is easy to implement and train.
Applying boosting with weak learners as the base models, allows to turn them into a strong learner. This may come out easyer than training a single complex model.

One of the most popular boosting algorithms is **AdaBoost** (Adaptive Boosting).

15.3.1 AdaBoost

Algorithm 10 AdaBoost

$d^{(0)} = \left(\frac{1}{N}, \dots, \frac{1}{N}\right)$ ▷ initialize uniform important weights
for $i = 1, \dots, m$ **do**
  $f^{(i)} \leftarrow \mathcal{A}(D, d^{(i-1)})$ ▷ train i^{th} model on weighted data
  $\hat{y}_n \leftarrow f^{(i)}(x_n),\ \forall n$ ▷ collect model predictions
  $\hat{\varepsilon}^{(i)} \leftarrow \sum_n d_n^{(i-1)} \mathbb{I}[y_n \neq \hat{y}_n]$ ▷ compute weighted training error
  $\alpha^{(i)} \leftarrow \frac{1}{2} \log\left(\frac{1-\hat{\varepsilon}^{(i)}}{\hat{\varepsilon}^{(i)}}\right)$ ▷ compute adaptive parameter
  $d_n^{(i)} \leftarrow \frac{1}{Z} d_n^{(i-1)} \exp\left(-\alpha^{(i)} y_n \hat{y}_n\right),\ \forall n$ ▷ re-weight examples
end for
return $f(x) = \text{sgn}\left(\sum_i \alpha^{(i)} f^{(i)}(x)\right)$

Where $d^{(i)}$ are the importance weights of the training examples at iteration i:

$$d_n^{(i)} = \frac{1}{Z} d_n^{(i-1)} \exp\left(-\alpha^{(i)} y_n \hat{y}_n\right)$$

- Correctly classified examples ($y_n * \hat{y}_n = 1$) will have their weight decreased;
- Incorrectly classified examples ($y_n * \hat{y}_n = -1$) will have their weight increased;
- Z is a normalization factor to ensure that the weights sum to 1.