

Introduction to Machine Learning: Theory and Application

Chapter 1

Alexander Szimayer

Universität Hamburg

Summer Term 2024

1. Introduction and Model Selection and Evaluation

In this chapter we

- motivate the idea of machine learning,
- introduce structures of machine learning, and
- provide an overview of real life applications.
- define the error rate as well as overfitting and underfitting.
- discuss how different evaluation methods mitigate overfitting.
- show how performance measures are used to monitor and measure the model predictions.
- learn about the inductive bias and the bias-variance trade-off.

Symbols

x	Scalar	$\ \cdot\ _p$	L_p norm; L_2 when p is absent.
x	Vector	$P(\cdot), P(\cdot \cdot)$	Probability mass function conditional probability mass function.
x	Variable set	$p(\cdot), p(\cdot \cdot)$	Probability density function, conditional probability density function.
A	Matrix	$E_{\cdot \sim \mathcal{D}}[f(\cdot)]$	Expectation of function $f(\cdot)$ with respect to \cdot over distribution \mathcal{D} ; \mathcal{D} and/or \cdot are omitted when context is clear.
I	Identity Matrix	I	Indicator function, returns 1 if \cdot is true, and 0 if \cdot is false.
\mathcal{X}	Sample space or state space	$\text{sign}(\cdot)$	Sign function, returns -1 if $\cdot < 0$, 0 if $\cdot = 0$, and 1 if $\cdot > 0$
\mathcal{D}	Probability Distribution	D	Data set
H	Hypothesis set	\mathcal{H}	Hypothesis space
(\cdot, \cdot, \cdot)	Row vector	$(\cdot; \cdot; \cdot)$	Column vector
$\{\dots\}$	Set	$\{ \cdot \}$	Number of elements in set $\{\dots\}$

1.1 Introduction: Real Life Example

- Imagine, observing a gentle breeze and sunset glow in the evening.
You think about the weather for the next day and expect it to be beautiful. Why?
- From our **experience** the weather on the next day is often beautiful after such a scene in the present day.
- With expecting the weather to be beautiful we have made an **experience-based prediction**.

Can computers also learn from experience?

- Yes, computers can learn from experience in form of data.
- Machine Learning improves system performance by learning from experience via computational methods.
- The main task of Machine Learning is to create learning algorithms that build models from data.
- By giving the learning algorithm **experience data**, we obtain a model that can make predictions on new observations.

1.2 Terminology

Example: Predicting the ripeness of uncut watermelons

- Suppose we want to determine the ripeness of uncut watermelons and have collected the following set of watermelons:

ID	color	root	sound
1	green	curly	muffled
2	dark	curly	muffled
3	green	straight	crisp
4	dark	slightly curly	dull

Fundamental Terms

- **Model:** Outcome learned from data.
- **Data set:** A collection of samples where each sample describes an event or object.
- **Sample/Instance:** An example in the data set (*a specific watermelon*).
- **Attribute/Feature:** Is an aspect or description of the sample (*the color, root or sound of a watermelon*).
- **Sample space/attribute space/input space:** The space spanned by the attributes (*color, root and sound span a three-dimensional space describing watermelons, where we can position every watermelon in this space*).

Mathematical Terminology

- The data set $D = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_m\}$ contains m samples, where each sample is described by d attributes.
- Each sample $\mathbf{x}_i = (x_{i1}, x_{i2}, \dots, x_{id}) \in \mathcal{X}$ is a vector in the d -dimensional sample space \mathcal{X} , where d is the dimensionality of the sample \mathbf{x}_i , and x_{ij} is the value of the j th attribute of the sample \mathbf{x}_i .

Can we already predict the ripeness of uncut watermelons?

- In our watermelon example, we don't know which of the four watermelons are ripe or unripe. Therefore, the samples are not sufficient for determining the ripeness of an uncut watermelon.
- To train an effective prediction model, the **outcome** information must be available.

ID	color	root	sound	ripe
1	green	curly	muffled	true
2	dark	curly	muffled	true
3	green	straight	crisp	false
4	dark	slightly curly	dull	false

Label

- The outcome of a sample (ripe or unripe) is called **label**, and a sample with a label is called an **example**.
- Generally:
We can write the i th sample as (x_i, y_i) , where $y_i \in \mathcal{Y}$ is the label of the sample x_i , and \mathcal{Y} is the set of all labels, also called **label space** or **output space**.

Training vs. Testing (1/2)

- Building models from given data by using Machine Learning algorithms is called **training**.
- Making predictions with a learned model is called **testing**.
- The data used in the training phase is called **training data**. Each sample in the training data is called **training example**, and the set of all training examples is a **training set**.
- The training set is a proportion of the whole sample space.

Training vs. Testing (2/2)

- In Machine Learning we want to learn models that can work well on new samples. Therefore, we want the training set to reflect the characteristics of the whole sample space.
- The ability to work on the new samples is called **the generalization ability**.
- A well-generalized model should work well on the whole sample space.
- In general, we **assume** that all samples are independent and identically distributed (i.i.d.):
 - All samples in the sample space follow a distribution \mathcal{D} .
 - All samples are independently sampled from \mathcal{D} .

Classification vs. Regression

- **Regression problem:** the prediction output is continuous (the degree of ripeness).
- **Classification problem:** the prediction output is discrete (ripe or unripe).
 - **Binary classification problem:** the prediction output has only two possible classes.
 - **Multiclass classification problem:** the prediction output has more than two classes.

Mathematical Explanation

- A prediction problem is nothing less than establishing a mapping

$$f : \mathcal{X} \mapsto \mathcal{Y}$$

from the input space \mathcal{X} to the output space \mathcal{Y} by learning from a training set $\{(x_1, y_1), (x_2, y_2), \dots, (x_m, y_m)\}$.

- Depending on the specific problem, the output space follows:
 - For regression problems: $\mathcal{Y} = \mathbb{R}$.
 - For binary classification problems: $\mathcal{Y} = \{-1, +1\}$, or $\{0, 1\}$.
 - For multiclass classification problems: $|\mathcal{Y}| > 2$.

Supervised vs. Unsupervised learning

- Depending on the training data we can roughly divide learning problems in two classes:
 - Supervised learning:** training data is labeled (e.g. Regression and Classification).
 - Unsupervised learning:** training data is unlabeled (e.g. Clustering).

1.3 Empirical Error and Overfitting (1/2)

- The **error rate E** is the proportion of incorrectly classified samples to the total number of samples.
 - $E = \frac{a}{m}$, where a is the number of misclassified samples and m the number of all samples.
 - Accuracy = $1 - E$.
- The **error** is the difference between the output predicted by the learner and the ground-truth output.
 - The **empirical error** is calculated on the training set.
 - The **generalization error** is calculated on the test set.
- The aim is to minimize the generalization error, however, this is not possible during the testing phase in practice such that the empirical error is minimized instead.

1.3 Empirical Error and Overfitting (2/2)

- Good learners should learn general rules from the training process which apply also to unseen new data.
 - **Overfitting:** learners learn too well on the training set.
 - **Underfitting:** learners learn too less on the training set.
- Both types lead to poor generalization performance on the test set.
- In practice, there are often multiple candidate learning algorithms such that evaluation methods are needed.

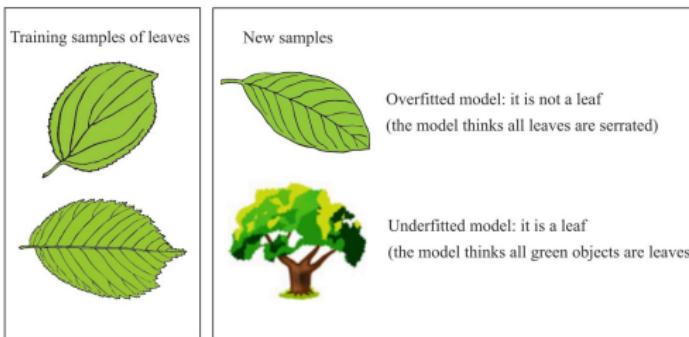


Figure 1.1: Analogy of overfitting and underfitting.

1.4 Evaluation Methods

- To avoid overfitting, machine learning experiments create **training data** to learn model parameters and **testing data** to evaluate the learning parameters.
- The data partitioning should ensure the preservation of several characteristics:
 - Testing and training set should be **mutually exclusive** as much as possible.
 - The partitioned data should have comparable distribution characteristics of crucial attribute characteristics (**stratified sampling**).

Hold-Out

- The **hold-out** method splits the data set D into two disjoint subsets. One, as the training set S and the other as the testing set T :

$$D = S \cup T \text{ and } S \cap T = \emptyset.$$

- Different splitting strategies result in different training and test sets which has a large influence on the model outcomes.
- Way out: **stratified sampling** ensures comparable distribution of attributes on the training and test set.

Cross-Validation (1/2)

- **Cross-validation** splits the data set D into k disjoint subsets with similar sizes:

$$D = D_1 \cup D_2 \cup \dots \cup D_k, D_i \cap D_j = \emptyset (i \neq j).$$

- Each subset D_i tries to maintain the original data distribution via stratified sampling.

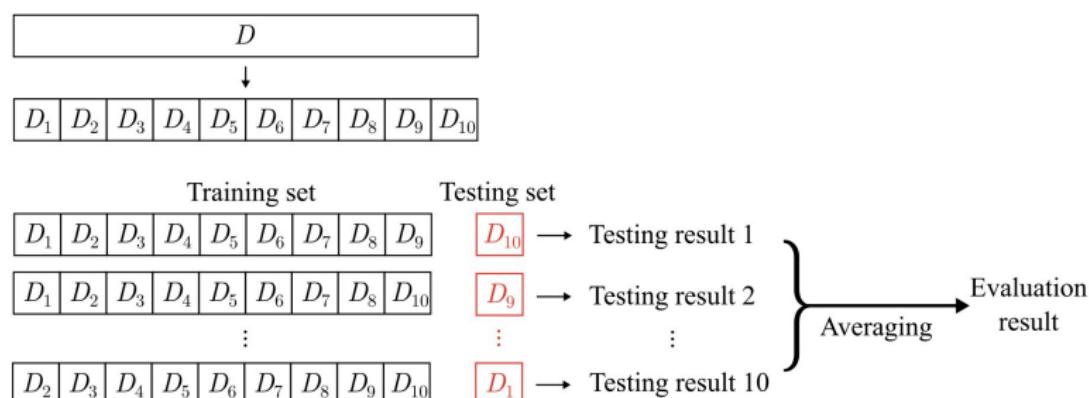


Figure 1.2: 10-fold cross-validation.

Cross-Validation (2/2)

- The stability and fidelity of cross-validation largely depend on the value of k .
- To decrease the error introduced by splitting, we often repeat the random splitting p times and average the evaluation results of p times of k -fold cross-validation.
- Special case of cross-validation: **leave-one-out (LOO)**
 - Data set D with m samples and $k = m$.
 - In most cases, the evaluation from LOO is very close to the ideal evaluation of training the model on D .
 - The approach is computational costs prohibitive.

Bootstrapping

- **Bootstrapping** samples a data set D' by randomly picking one sample from D from m samples, copying it to D' , and then placing it back to D so that it still has a chance to be picked next time.
- Repeating this process m times results in the bootstrap sampling data set D' containing m samples.
- Due to **replacement**, some samples in D may appear more than once, while others may not appear, where the chance of not appearing is:

$$\lim_{n \rightarrow \infty} \left(1 - \frac{1}{m}\right)^m = \frac{1}{e} \approx 0.368.$$

- Use D' as training set and $D \setminus D'$ as testing set.
- The evaluation result obtained via this approach is called **out-of-bag estimate**.

Parameter Tuning and Final Model

- Most learning algorithms have parameters to set, and different parameter settings often lead to models with significantly different performance.
- The process of finding the right parameters is called **parameter tuning**.
- Machine learning involves two types of parameters:
 - **Hyper-parameters** are configured manually to find the model parameters.
 - **Model parameters** are generated by learning.
- Since parameters are often real-valued, it is impossible to try all parameter settings (in practice, set a range and a step size for each parameter).
- We often call the data set used in the model selection a **validation set**.

1.5 Performance Measure

- Performance measures are used to monitor and measure the model predictions on the training and test set.
- In prediction problems, we are given a data set $D = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_n, y_n)\}$, where y_i is the ground-truth label of the sample \mathbf{x}_i .
- To evaluate the performance of the learner f , compare its prediction $f(\mathbf{x})$ to the ground-truth label y .
- For regression problems, the most common used performance measure is the **mean squared error**:

$$E(f; D) = \frac{1}{m} \sum_{i=1}^m (f(\mathbf{x}_i) - y_i)^2.$$

- Or more, general for a data distribution \mathcal{D} and a probability density function $p(\mathbf{x})$ the MSE can be written as:

$$E(f; \mathcal{D}) = \int_{\mathbf{x} \sim \mathcal{D}} (f(\mathbf{x}) - y)^2 p(\mathbf{x}) d\mathbf{x}.$$

Error Rate and Accuracy

- For classification problems, the model performance is evaluated by:
 - Error rate**, which is the proportion of misclassified samples relative to all samples, where I is the indicator function and D a data set:

$$E(f; D) = \frac{1}{m} \sum_{i=1}^m I(f(\mathbf{x}_i) \neq y_i).$$

- Accuracy**, determined by the proportion of correctly classified samples:

$$\begin{aligned} \text{acc}(f; D) &= \frac{1}{m} \sum_{i=1}^m I(f(\mathbf{x}_i) = y_i) \\ &= 1 - E(f; D). \end{aligned}$$

- Analogous to the MSE, both measures can be expressed in more general terms, assuming a given data distribution \mathcal{D} and a probability distribution $p(\mathbf{x})$.

Precision, Recall and Confusion Matrix (1/2)

- To evaluate a binary classifier quality, there are four combinations of the ground-truth class and the predicted class:
 - True positives (TP)**: learner correctly predicts the presence of a characteristic.
 - True negative (TN)**: learner correctly predicts the absence of a characteristic.
 - False positive (FP)**: learner wrongly predicts the presence of a characteristic.
 - False negative (FN)**: learner wrongly predicts the absence of a characteristic.

$\Rightarrow TP + TN + FP + FN = \text{total number of samples.}$

Precision, Recall and Confusion Matrix (2/2)

- The confusion matrix is a special kind of contingency table, visualizing the classification performance.
- There are two other accuracy measures:
 - Precision (P)** = $\frac{TP}{TP+FP}$ → which is the fraction of relevant samples among retrieved samples.
 - Recall (R)** = $\frac{TP}{TP+FN}$ → which is the fraction of relevant samples that were retrieved.

Ground-truth class	Predicted class	
	Positive	Negative
Positive	TP	FN
Negative	FP	TN

Figure 1.3: Confusion Matrix.

Precision-Recall Curve (1/2)

- Precision and recall are contradictory measures, such that in general both measures can not be high at the same time which results in the **precision-recall trade-off**.
- Varying the learner's threshold, which forces each sample prediction to belong to a 'true' or 'false' prediction, leads to different outcomes for the precision and recall measures.
- Plotting the recall (x-axis) against precision (y-axis) results in the **precision-recall curve (PR curve)**.

Precision-Recall Curve (2/2)

- If the PR curve of one learner entirely encloses the curve of another learner, the performance of the first learner is superior.
- When the PR curves intersect, the learners can only be compared at a specific precision recall.
- The **break-even point (BEP)** is the point at which precision equals recall.
- As a solution, one can calculate the area under the curve (AUC).

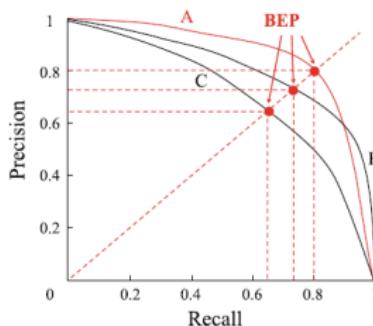


Figure 1.4: P-R curve and break-even points.

F_1 Score

- The F_1 score is the harmonic mean of the precision and recall, where the best score is a value of 1 and the worst score is a value of 0.
- The relative contributions of precision and recall are equal in the standard F_1 score:

$$F_1 = \frac{2 \times P \times R}{P + R} = \frac{2 \times TP}{\text{Total number of samples} + TP - TN}.$$

- The general form is defined as:

$$F_\beta = \frac{(1 + \beta^2) \times P \times R}{(\beta^2 \times P) + R}.$$

- $\beta > 1$ gives relative more importance of recall to precision.
- $\beta = 1$ reduces to F_1 .
- $\beta < 1$ gives relative more importance of precision to recall.

Multiple Confusion Matrices

- Sometimes multiple confusion matrices can occur such that the overall confusion matrix needs to be determined.
- One approach is to calculate precision and recall for each confusion matrix, denoted by $(P_1, R_1), \dots, (P_n, R_n)$ by taking the averages:

$$\text{macro-}P = \frac{1}{n} \sum_{i=1}^n P_i,$$

$$\text{macro-}R = \frac{1}{n} \sum_{i=1}^n R_i,$$

$$\text{macro-}F_1 = \frac{2 \cdot \text{macro-}P \cdot \text{macro-}R}{\text{macro-}P + \text{macro-}R}.$$

- Element-wise averages across the confusion matrices to get $\overline{TP}, \overline{FP}, \overline{TN}, \overline{FN}$ and then taking averages to obtain the micro- P , the micro- R and the micro- F_1 is also possible.

ROC and AUC

- Learner predictions are often in the form of real values or probabilities, e.g. according to classification certainty.
- Changing the classification threshold (**cut point**) determining how the learner prediction is classified has an influence on the generalization ability.
- The position of the cut point influences whether the learner puts more weight on high precision or high recall.

Receiver Operating Characteristics Curve (ROC)

- Sort samples by the predictions and then obtain two measures by gradually moving the cut point from the top towards the bottom of the ranked list in descend order.
- Using those two measures as x-axis and y-axis results in the **receiver operating curve (ROC)**:

True Positive Rate (TPR) = $\frac{TP}{TP + FN}$ on the y-axis and

False Positive Rate (FPR) = $\frac{FP}{TN + FP}$ on the x-axis.

- Plotting the ROC curves is called the ROC plot.

Constructing the ROC Curve

- ① Given m^+ positive samples and m^- negative samples, all samples are sorted by learner's prediction and the cut point is set to maximum, meaning all samples are classified as negative.
- ② Gradually decrease the cut point to the predicted value of each sample of the sorted list:
 - Let (x, y) denote the previous candidate, put a mark at $(x, y + \frac{1}{m^+})$ if the current samples are true positive and
 - put a mark at $(x + \frac{1}{m^-}, y)$ if the current samples are false positive.
- ③ Connecting all adjacent marked points results in the ROC curve.

Interpreting the ROC Curve

- The diagonal corresponds to the 'random guessing' model.
- The coordinate point $(0,0)$ indicates both, TPR and FPR as zero and the coordinate point $(0,1)$ corresponds to the 'ideal model'.
- If the ROC curve of learner A entirely encloses the ROC curve of learner B , it follows that learner A is superior to learner B .
- If the ROC curves of learner A and B intersect at some point, no conclusion about the superiority can be made by simply comparing the ROC curves.
- Another possibility to compare the ROC curves the **Area under the ROC Curve (AUC)**.

Area Under the ROC Curve

- AUC is calculated by integrating the areas under the steps of the ROC curve.
- Suppose that the ROC curve is obtained sequentially connecting the points $(x_1, y_1), (x_2, y_2), \dots, (x_m, y_m)$, where $x_1 = 0$ and $x_m = 1$, AUC is estimated as:

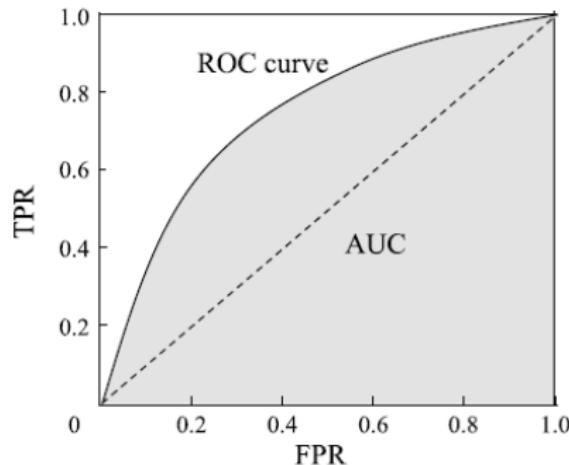
$$\text{AUC} = \frac{1}{2} \sum_{i=1}^{m-1} (x_{i+1} - x_i) \cdot (y_i + y_{i+1}).$$

- AUC is closely related to ranking errors since it considers the ranking quality of predictions,

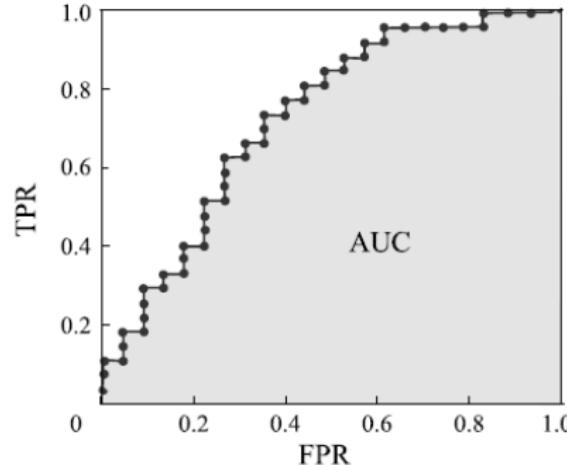
$$\ell_{rank} = \frac{1}{m^+ m^-} \sum_{\mathbf{x}^+ \in D^+} \sum_{\mathbf{x}^- \in D^-} \left(I(f(\mathbf{x}^+) < f(\mathbf{x}^-)) + \frac{1}{2} I(f(\mathbf{x}^+) = f(\mathbf{x}^-)) \right),$$

where D^+ denotes the positive samples and D^- the negative samples.

Drawing the ROC Curve and the AUC



(a) ROC curve and AUC.



(b) ROC curve and AUC with finite samples.

Figure 1.5: An illustration of ROC curve and AUC.

1.6 Inductive Bias

- Since each learned model corresponds to one hypothesis in the version space, a potential problem arises:
 - Different models may predict the new samples differently, though all hypotheses are consistent with the training examples.
 - In this case, which model (or hypothesis) should we use?
- A learning algorithm must make a choice and produce a model.
- The **inductive bias** of the learning algorithm plays a decisive role.
- The bias of a learning algorithm toward a particular class of hypotheses is called the **inductive bias** or simply **bias**.

Consistent Curve

- Each training example is shown as a point (x, y) , and the objective is to learn a curve passing through all training examples:

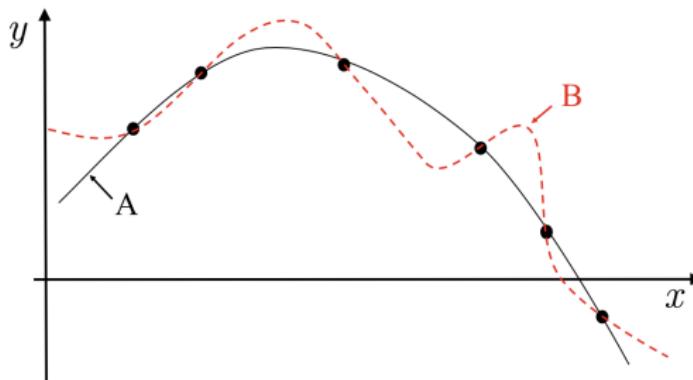


Figure 1.6: Curves consistent with finite training set.

- Since there are infinite qualified curves for the finite training set, a learning algorithm must have its inductive bias to learn the “correct” model.

1.7 Bias-Variance Trade-off

- If the learning algorithm believes that similar samples should have similar labels, then it is likely to prefer the smooth curve A over the oscillating curve B.
- We can regard inductive bias as the value philosophy of learning algorithms for search in potentially huge hypothesis spaces.
- There is always a “trade-off” between reducing the bias or variance of the sample distribution.
- An essential tool for understanding the generalization performance of algorithms is the **bias-variance decomposition**, which decomposes the expected generalization error of learning algorithms.

Bias-Variance Decomposition (1/4)

- Let x be a testing sample, y_D be the label of x in the data set D , y be the ground-truth label of x , and $f(x; D)$ be the output of x predicted by the model f trained on D .
- Then, in regression problems, the **expected prediction** of a learning algorithm is: $\bar{f}(x) = E_D[f(x; D)]$.
- The **variance** of using different equal-sized training sets is:
$$\text{var}(x) = E_D[f(x; D) - \bar{f}(x)]^2.$$
- The **noise** is:
$$\varepsilon^2 = E_D[(y_D - y)^2].$$
- The difference between the expected output and the ground-truth label is called **bias**, that is:
$$\text{bias}^2(x) = (\bar{f}(x) - y)^2.$$

Bias-Variance Decomposition (2/4)

We assume the expectation of noise is zero, i.e., $E_D[y_D - y] = 0$ and since the noise is independent of f , the last term in the third row is zero.

$$\begin{aligned} E(f; D) &= E_D[(f(\mathbf{x}; D) - y_D)^2] \\ &= E_D[(f(\mathbf{x}; D) - \bar{f}(\mathbf{x}) + \bar{f}(\mathbf{x}) - y_D)^2] \\ &= E_D[(f(\mathbf{x}; D) - \bar{f}(\mathbf{x}))^2] + E_D[(\bar{f}(\mathbf{x}) - y_D)^2] \\ &\quad + E_D[2(f(\mathbf{x}; D) - \bar{f}(\mathbf{x}))(\bar{f}(\mathbf{x}) - y_D)] \\ &= E_D[(f(\mathbf{x}; D) - \bar{f}(\mathbf{x}))^2] + E_D[(\bar{f}(\mathbf{x}) - y + y - y_D)^2] \\ &= E_D[(f(\mathbf{x}; D) - \bar{f}(\mathbf{x}))^2] + E_D[(\bar{f}(\mathbf{x}) - y)^2] \\ &\quad + E_D[(y - y_D)^2] + 2E_D[(\bar{f}(\mathbf{x}) - y)(y - y_D)] \\ &= E_D[(f(\mathbf{x}; D) - \bar{f}(\mathbf{x}))^2] + (\bar{f}(\mathbf{x}) - y)^2 + E_D[(y_D - y)^2]. \end{aligned}$$

Bias-Variance Decomposition (3/4)

The generalization error can be decomposed into the sum of bias, variance and noise

$$E(f; D) = \text{bias}^2(\mathbf{x}) + \text{var}(\mathbf{x}) + \varepsilon^2,$$

- The **bias** measures the difference between the learning algorithm's expected prediction and the ground-truth label, expressing the fitting ability of the learning algorithm.
- The **variance** measures the change of learning performance caused by changes to the equal-sized training set, expressing the impact of data disturbance on the learning outcome.
- The **noise** represents the lower bound of the expected generalization error that can be achieved by the learning algorithm.

Bias-Variance Decomposition (4/4)

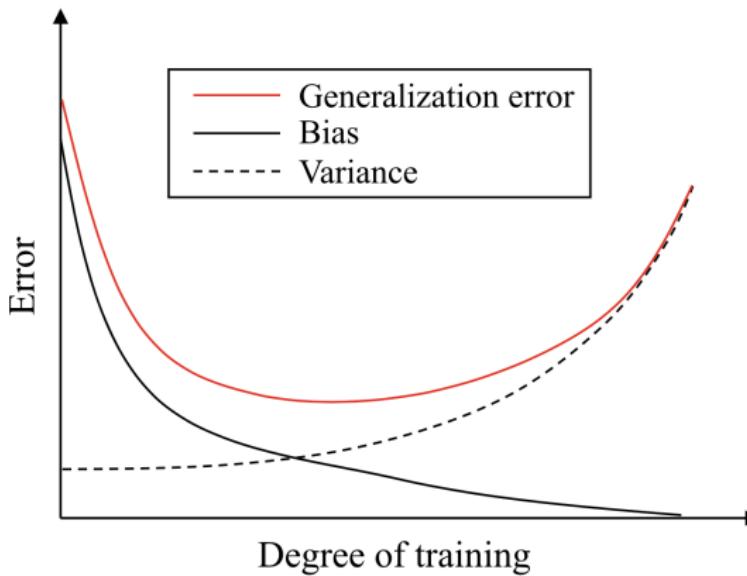


Figure 1.7: Relationships between generalization error, bias and variance.

Summary

In this chapter we have learned

- how the error rate is defined and how it is related to overfitting and underfitting.
- how to create different data sets to avoid overfitting.
- tools to evaluate the performance of machine learning techniques.
- about inductive bias as well as the bias-variance trade-off.

Introduction to Machine Learning: Theory and Application

Chapter 2

Alexander Szimayer

Universität Hamburg

Summer Term 2024

2. Linear Models

In this chapter we

- recap the linear regression,
- introduce the logistic regression and the linear discriminant analysis, and
- provide an insight into two further classification problems (the multiclass classification and the class imbalance problem).

2.1 Linear Regression: Basic Form

- A sample $\mathbf{x} = (x_1; x_2; \dots; x_d)$ is described by d variables, where x takes the value x_i in the i th variable.
- The objective of a linear model is to learn a function that makes predictions by a linear combination of the input variables,

$$f(\mathbf{x}) = w_1x_1 + w_2x_2 + \dots + w_dx_d + b,$$

or written in vector form,

$$f(\mathbf{x}) = \mathbf{w}^\top \mathbf{x} + b,$$

where $\mathbf{w} = (w_1; w_2; \dots; w_d)$.

- Once \mathbf{w} and b are learned, the model is determined.

Discrete Variables

- When dealing with discrete variables it is important to transform them. Here we have two cases:
 - Variables which can be converted into real-valued variables when an ordinal relationship naturally exists between values.
 - E.g. the values tall, medium and short of the ordinal variable height can be converted into $\{1.0, 0.5, 0.0\}$.
 - Variables, with no ordinal relationship, often can be converted with k possible values will be converted into a k -dimensional vector.
 - E.g. the values watermelon, pumpkin and cucumber of the ordinal variable cucurbits can be converted into $(0,0,1)$, $(0,1,0)$ and $(1,0,0)$.

Univariate Linear Regression

- The objective of a linear regression is to learn a linear model, such as,

$$f(x) = wx + b,$$

where $f(w_i) \simeq y_i$, for $i = 1, \dots, m$. So we can predict the real-valued output labels.

The Mean Squared Error (MSE)

- To compute w and b , we measure the difference between $f(x)$ and y by minimizing the MSE:

$$\begin{aligned}(w^*, b^*) &= \arg \min_{(x,y)} \sum_{i=1}^m (f(x_i) - y_i)^2 \\ &= \arg \min_{(x,y)} \sum_{i=1}^m (y_i - wx_i - b)^2.\end{aligned}$$

- The process of minimizing $E_{(w,b)} = \sum_{i=1}^m (y_i - wx_i - b)^2$ is called **least squares parameter estimation** of linear regression.

Least Squares Parameter Estimation of Linear Regression

- Setting the derivations of $E_{(w,b)}$ equal to zero, we can compute w and b :

$$\frac{\partial E_{(w,b)}}{\partial w} = 2 \left(w \sum_{i=1}^m x_i^2 - \sum_{i=1}^m (y_i - b)x_i \right) = 0,$$

$$\frac{\partial E_{(w,b)}}{\partial b} = 2 \left(mb - \sum_{i=1}^m (y_i - wx_i) \right) = 0.$$

- Which results in the closed-form solutions:

$$w = \frac{\sum_{i=1}^m y_i(x_i - \bar{x})}{\sum_{i=1}^m x_i^2 - \frac{1}{m}(\sum_{i=1}^m x_i)^2},$$

$$b = \frac{1}{m} \sum_{i=1}^m (y_i - wx_i).$$

where $\bar{x} = \frac{1}{m} \sum_{i=1}^m x_i$ is the arithmetic mean of x .

Multivariate Linear Regression

- If we have a data set $D = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_m, y_m)\}$, where $\mathbf{x}_i = (x_{i1}; x_{i2}; \dots; x_{id})$ and $y_i \in \mathbb{R}$, our linear model becomes,

$$f(\mathbf{x}) = \mathbf{w}^\top \mathbf{x} + b,$$

which is known as **multivariate linear regression**.

- To compute \mathbf{w} and b , we use **the least squares method**.
- For simplification, we rewrite \mathbf{w} and b as $\hat{\mathbf{w}} = (\mathbf{w}; b)$.

Least Squared Method for Multivariate Linear Regression (1/2)

- The data set D is represented as an m by $(d + 1)$ matrix \mathbf{X} , that is:

$$\mathbf{X} = \begin{pmatrix} x_{11} & x_{12} & \dots & x_{1d} & 1 \\ x_{21} & x_{22} & \dots & x_{2d} & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ x_{m1} & x_{m2} & \dots & x_{md} & 1 \end{pmatrix} = \begin{pmatrix} \mathbf{x}_1^\top & 1 \\ \mathbf{x}_2^\top & 1 \\ \vdots & \vdots \\ \mathbf{x}_m^\top & 1 \end{pmatrix}$$

- By vectorizing the labels to $\mathbf{y} = (y_1; y_2; \dots; y_m)$, we get:

$$\hat{\mathbf{w}}^* = \arg \min_{\hat{\mathbf{w}}} (\mathbf{y} - \mathbf{X}\hat{\mathbf{w}})^\top (\mathbf{y} - \mathbf{X}\hat{\mathbf{w}}).$$

- When we use $E_{(\hat{\mathbf{w}})} = (\mathbf{y} - \mathbf{X}\hat{\mathbf{w}})^\top (\mathbf{y} - \mathbf{X}\hat{\mathbf{w}})$ and find the derivative with respect to $(\mathbf{y} - \mathbf{X}\hat{\mathbf{w}})^\top (\mathbf{y} - \mathbf{X}\hat{\mathbf{w}})\hat{\mathbf{w}}$, we have:

$$\frac{\partial E_{\hat{\mathbf{w}}}}{\partial \hat{\mathbf{w}}} = 2\mathbf{X}^\top (\mathbf{X}\hat{\mathbf{w}} - \mathbf{y}) = 0. \quad (2.1)$$

Least Squared Method for Multivariate Linear Regression (2/2)

- Setting (2.1) equal to zero, we obtain the close-form solution of \hat{w} . However the calculation is more complicated, due to matrix inverse operation.
- If the matrix $\mathbf{X}^\top \mathbf{X}$ is **full-ranked**, we receive,

$$\hat{w}^* = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y},$$

with the inverse $(\mathbf{X}^\top \mathbf{X})^{-1}$ of $\mathbf{X}^\top \mathbf{X}$.

- Now with $\hat{x}_i = (x_i; 1)$, we then get the **multivariate linear regression model**:

$$f(\hat{x}_i) = \hat{x}_i^\top (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}.$$

Large Number of Variables

- In reality, we often have a large number of variables, or even more variables than samples (more columns than rows in \mathbf{X}). In this case $\mathbf{X}^\top \mathbf{X}$ is not full-ranked.
- If $\mathbf{X}^\top \mathbf{X}$ is not full-ranked there is more than one $\hat{\mathbf{w}}$ that can minimize the MSE.
- In this case, the choice on $\hat{\mathbf{w}}$ depends on the inductive bias of the learning algorithm.

Log-Linear Regression and Generalized Linear Model

- A **log-linear regression** is a form of linear regression which is searching for a nonlinear mapping from the input space to the output space. That is:

$$\ln y = \mathbf{w}^\top \mathbf{x} + b.$$

- Log-linear regression approximates y with $e^{\mathbf{w}^\top \mathbf{x} + b}$, where the predictions of linear regression are linked to the ground-truth labels.
- We can generalize it even more by using a monotonic differentiable function $g(\cdot)$ as the link function,

$$y = g^{-1}(\mathbf{w}^\top \mathbf{x} + b),$$

which is called the **generalized linear model**.

2.2 Logistic Regression

- For binary classification we have to convert the real-valued predictions of the linear regression model $z = \mathbf{w}^\top \mathbf{x} + b$ into 0/1. Ideally, getting the following unit-step function:

$$y = \begin{cases} 0, & z < 0; \\ 0.5, & z = 0; \\ 1, & z > 0. \end{cases}$$

The output of z is positive for z greater than zero, negative for z smaller than zero and arbitrary when z equals zero.

- Since the unit-step function is not continuous we use a monotonic differentiable logistic function to approximate the unit-step function. Often, this is the **logistic function**:

$$y = \frac{1}{1 + e^{-z}}. \quad (2.2)$$

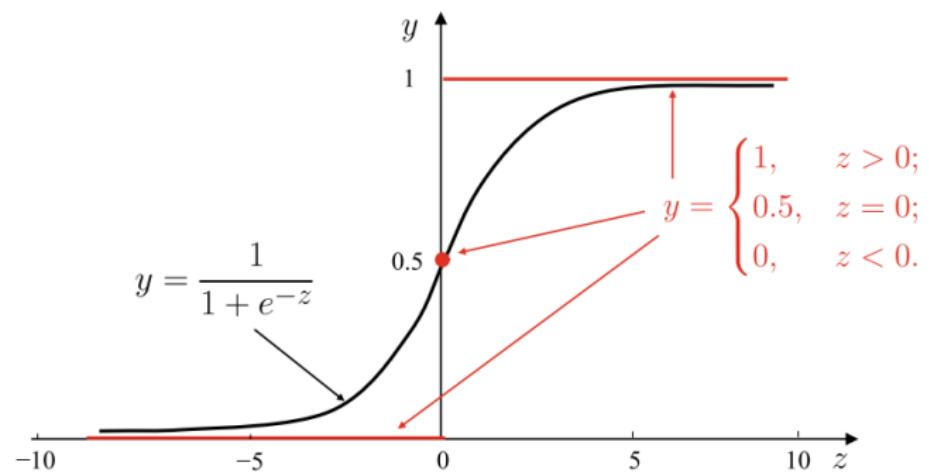


Figure 2.1: Unit-step function and logistic function. The logistic function is a type of sigmoid function which converts z into y that is either close to 0 or 1.

Logistic Regression Model (1/2)

- If we substitute the logistic function into equation (2.2), we have:

$$y = \frac{1}{1 + e^{-(w^\top x + b)}}. \quad (2.3)$$

Which is called the **logistic regression** or **logit regression**.

- Suppose y is the likelihood of x being a positive sample and $1 - y$ is the likelihood of x being a negative sample, then we call the following ratio **the odds**:

$$\frac{y}{1 - y}.$$

- The logarithmic form of the odds is called **log odds**:

$$\ln \frac{y}{1 - y}.$$

Logistic Regression Model (2/2)

- The equation (2.3) can be transformed into:

$$\ln \frac{y}{1 - y} = \mathbf{w}^\top \mathbf{x} + b. \quad (2.4)$$

- It should be noted that the logistic regression is indeed a **classification** model despite the term 'regression' in its name.

Estimation of w and b (1/3)

- Consider y in equation (2.3) as the **posterior probability** $p(y = 1|x)$, then equation (2.4) can be rewritten as:

$$\ln \frac{p(y = 1|x)}{p(y = 0|x)} = \mathbf{w}^\top \mathbf{x} + b. \quad (2.5)$$

- From this we can derive:

$$p(y = 1|\mathbf{x}) = \frac{e^{\mathbf{w}^\top \mathbf{x} + b}}{1 + e^{\mathbf{w}^\top \mathbf{x} + b}}, \quad (2.6)$$

$$p(y = 0|\mathbf{x}) = \frac{1}{1 + e^{\mathbf{w}^\top \mathbf{x} + b}}. \quad (2.7)$$

Estimation of w and b (2/3)

- We can use the Maximum-Likelihood method to estimate w and b .

Given the data set $\{(x_i, y_i)\}_{i=1}^m$ the log-likelihood function is:

$$\ell(w, b) = - \sum_{i=1}^m \ln p(y_i | x_i; w, b). \quad (2.8)$$

- We define $\beta := (w, b)$ and $\hat{x} := (x; 1)$ to simplify $w^\top x + b$ as $\beta^\top \hat{x}$.

Further define $p_1(\hat{x}; \beta) := p(y = 1 | \hat{x}; \beta)$ and
 $p_0(\hat{x}; \beta) := p(y = 0 | \hat{x}; \beta)$. Then (2.8) becomes:

$$p(y_i | x_i; w, b) = y_i p_1(\hat{x}_i; \beta) + (1 - y_i) p_0(\hat{x}_i; \beta). \quad (2.9)$$

- Now substitute equation (2.9) in equation (2.8), use the equations (2.6) and (2.7) and note that $y_i \in \{0, 1\}$ to get:

$$\ell(\beta) = \sum_{i=1}^m \left(-y_i \beta^\top \hat{x}_i + \ln \left(1 + e^{\beta^\top \hat{x}_i} \right) \right). \quad (2.10)$$

Estimation of w and b (3/3)

- The function (2.10) is a higher order differential convex function with respect to β . Therefore, it can be solved via classical numerical optimization methods (e.g. Newton's method). Hence, we have:

$$\beta^* = \arg \min_{\beta} \ell(\beta). \quad (2.11)$$

- With Newton's method we get:

$$\ell'(\beta) = - \sum_{i=1}^m \hat{x}_i (y_i - p_1(\hat{x}_i; \beta)),$$

$$\ell''(\beta) = \sum_{i=1}^m \hat{x}_i \hat{x}_i^\top p_1(\hat{x}_i; \beta)(1 - p_1(\hat{x}_i; \beta)).$$

Benefits of Logistic Regression

- The logistic regression does not require any prior assumptions on the data distribution to directly model the label probability. Issues such as inappropriate hypothetical data distributions can therefore be avoided.
- The logistic regression predicts labels together with associated probabilities which is beneficial when probability is used for decision making.
- The objective function of logistic regression is convex, which makes it solvable with numerical optimization properties.

2.3 Linear Discriminant Analysis

- We want to find another method for solving binary classification problems.
- The idea is to project the training samples onto a line in a way that:
 - Samples of the same class are near to each other, and
 - Samples of different classes are far away from each others location, after projecting them onto the same line.
- Classes of new samples can then be determined by their projected locations.
- This method is called **Linear Discriminant Analysis (LDA)**, also known as **Fischer's Linear Discriminant**.

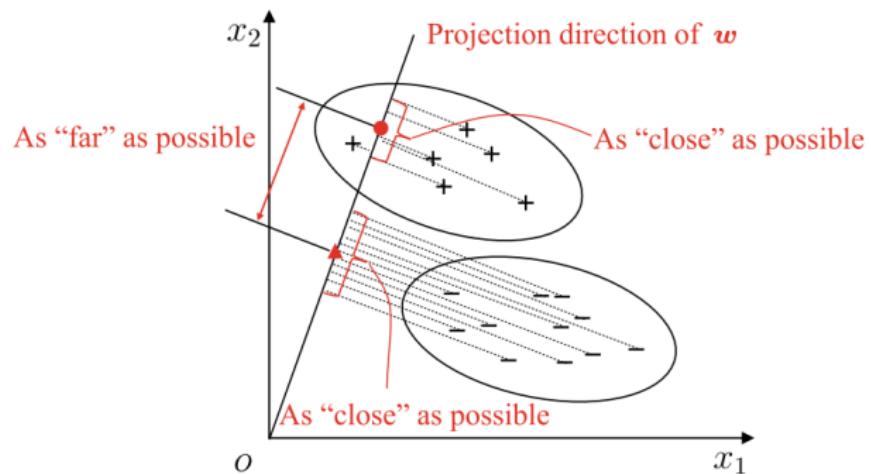


Figure 2.2: A two-dimensional illustration of LDA, where '+' and '-' denote positive and negative samples and ' O ' the origin. The ellipses are the boundaries of clusters; the dashed lines represent projections; the solid red dot and triangle are the centers of the projections.

Linear Discriminant Analysis Model (1/3)

- Given the data set $D = \{(\mathbf{x}_i, y_i)\}_{i=1}^m$, $y_i \in \{0, 1\}$ and X_i denotes the sample set, μ_i denotes the mean vector and Σ_i denotes the covariance matrix of the i th class ($i \in \{0, 1\}$).
- We project data onto the line w , where:
 - The centers of the two classes samples are $w^\top \mu_0$ and $w^\top \mu_1$.
 - The covariance of the two classes samples are $w^\top \Sigma_0 w$ and $w^\top \Sigma_1 w$.

Linear Discriminant Analysis Model (2/3)

- By **minimizing the covariance** of the projection points,
 $w^\top \Sigma_0 w + w^\top \Sigma_1 w$, we can get the projection points of similar
points as close as possible.
- By **maximizing the distance** between the class centers,
 $\|w^\top \mu_0 - w^\top \mu_1\|$, we can get the projection points of examples from
different samples as far away as possible.
- Therefore, the objective is to maximize:

$$\begin{aligned} J &= \frac{\|w^\top \mu_0 - w^\top \mu_1\|_2^2}{w^\top \Sigma_0 w + w^\top \Sigma_1 w} \\ &= \frac{w^\top (\mu_0 - \mu_1)(\mu_0 - \mu_1)^\top w}{w^\top (\Sigma_0 + \Sigma_1) w}. \end{aligned} \tag{2.12}$$

Linear Discriminant Analysis Model (3/3)

- Defining the **within-class scatter matrix** as,

$$\begin{aligned}\mathbf{S}_w &= \sum_0 + \sum_1 \\ &= \sum_{x \in X_0} (x - \mu_0)(x - \mu_0)^\top + \sum_{x \in X_1} (x - \mu_1)(x - \mu_1)^\top\end{aligned}\quad (2.13)$$

and the **between-class scatter matrix** as,

$$\mathbf{S}_b = (\mu_0 - \mu_1)(\mu_0 - \mu_1)^\top \quad (2.14)$$

- We can rewrite equation (2.12) as,

$$J = \frac{\mathbf{w}^\top \mathbf{S}_b \mathbf{w}}{\mathbf{w}^\top \mathbf{S}_w \mathbf{w}}, \quad (2.15)$$

which gives us the **generalized Rayleigh quotient** of \mathbf{S}_b and \mathbf{S}_w .
Which is the objective of LDA to be maximized.

How to Compute w

- Without loss of generality, letting $w^\top S_w w = 1$, then maximizing (2.15) is equivalent to:

$$\begin{aligned} & \min_w -w^\top S_b w, \\ & \text{s.t. } w^\top S_w w = 1. \end{aligned} \tag{2.16}$$

- Using the method of Lagrange multipliers and knowing that the direction of $S_b w$ is always $\mu_0 - \mu_1$, the equation is equivalent to:

$$\begin{aligned} S_b w &= \lambda S_w w \\ &= \lambda(\mu_0 - \mu_1). \end{aligned} \tag{2.17}$$

- Which can be rearranged to:

$$w = S_w^{-1}(\mu_0 - \mu_1). \tag{2.18}$$

Further Remarks

- Numerical solution:

In order to achieve the stability of numerical solutions, singular value decomposition is often applied to \mathbf{S}_w .

- Bayesian solution:

From the aspect of the Bayesian decision theory, the optimal solution of LDA can be found when both classes follow the Gaussian distribution with the same prior and covariance.

Multiclass Case (1/2)

- Suppose we have N classes, where the i th class has m_i samples.
- We define the **global scatter matrix** (with μ being the mean vector of all samples) as:

$$\begin{aligned}\mathbf{S}_t &= \mathbf{S}_b + \mathbf{S}_w \\ &= \sum_{i=1}^m (x_i - \mu)(x_i - \mu)^\top.\end{aligned}\tag{2.19}$$

- We define the **within-class-scatter matrix** \mathbf{S}_w as the sum of scatter matrices of each class,

$$\mathbf{S}_w = \sum_{i=1}^N \mathbf{S}_{w_i}\tag{2.20}$$

where,

$$\mathbf{S}_{w_i} = \sum_{x \in X} (x - \mu_i)(x - \mu_i)^\top.\tag{2.21}$$

Multiclass Case (2/2)

- From the equations (2.19), (2.20) and (2.21), we get:

$$\begin{aligned}\mathbf{S}_b &= \mathbf{S}_t - \mathbf{S}_w \\ &= \sum_{i=1}^N m_i (\boldsymbol{\mu}_i - \boldsymbol{\mu})(\boldsymbol{\mu}_i - \boldsymbol{\mu})^\top\end{aligned}$$

- The multiclass LDA can be implemented in different ways by choosing any two from \mathbf{S}_b , \mathbf{S}_w , and \mathbf{S}_t . A common implementation is to optimize

$$\max_{\mathbf{W}} \frac{\text{tr}(\mathbf{W}^\top \mathbf{S}_b \mathbf{W})}{\text{tr}(\mathbf{W}^\top \mathbf{S}_w \mathbf{W})},$$

where $\mathbf{W} \in \mathbf{R}^{d \times (N-1)}$, and $\text{tr}(\cdot)$ is the trace of matrix.

- LDA reduces the data dimension while considering the class information, therefore LDA is considered a classic supervised dimensionality reduction technique.

2.4 Further Classification

2.4.1 Multiclass Classification

- To solve multiclass classification problems with any existing binary classification method, some strategies have to be applied.
- Given N classes, C_1, C_2, \dots, C_n , the basic idea of multiclass learning is **decomposition**, which means, dividing the multiclass classification problem into several binary classification problems.
- After decomposing the problem, we train a binary classifier for each divided binary classification problem.
- We then ensemble the outputs collected from all binary classifiers into the final multiclass predictions (testing phase).
- The key question is how to divide multiclass classification problems and how to ensemble multiple classifiers. There are mainly three classic dividing strategies, **One versus One (OvO)**, **One versus Rest (OvR)** and **Many versus Many (MvM)**.

One Versus One and One Versus Rest

- Let $D = \{(x_1, y_1), (x_2, y_2), \dots, (x_m, y_m)\}$, where $y_i \in \{C_1, C_2, \dots, C_N\}$.
- One versus One:**
 - N classes are divided into pairs, leading to $N(N - 1)/2$ binary classification problems.
 - Testing phase: A new sample is classified by all classifiers, resulting in $N(N - 1)/2$ classification outputs.
 - The final decision is made by voting: The class with the most votes will be the predicted class.
- One versus Rest:**
 - N classifiers are trained by considering each class as positive in turn, and the rest classes as negative.
 - Testing phase: If the new sample is predicted as positive by only one classifier, this is the final classification result.
 - If the new sample is predicted as positive by multiple classifiers, then the class with the highest prediction confidence is used as the classification result.

Many Versus Many

- Multiple trials, where each trial puts several classes as positive and several classes as negative, are conducted.
- A commonly used technique: **Error Correcting Output Codes (ECOC)**
 - Encoding: N classes will be split M times, where each time some classes are split as positive and some as negative (M training sets are generated and M classifiers are trained).
 - Decoding: The M classifiers are used to predict a testing sample and combine the predicted labels into a codeword. Then the distances between the codeword and the base codeword of each class are calculated. The final prediction will be the class with the shortest distance.

Advantages and Disadvantages

- The memory and testing time costs of OvO are often higher compared to that of OvR.
- The computational costs of training OvO is lower compared to that of OvR, especially when there are many classes.
- OvO and OvR are special cases of MvM.

2.4.2 Class Imbalance

- So far we assumed that there is no significant difference in the number of samples in each class.
- For the learning process, small differences in the number of samples are not as bad as large differences.
- A scenario with classification problem with a significantly different number of samples for each class, is called a **class imbalance**.
- A way to solve the class imbalance problem is **rescaling**.
- In the following we assume: The positive class is the minority, the negative class is the majority.

Rescaling (1/2)

- Using $y = \mathbf{w}^\top \mathbf{x} + b$ to classify a new sample \mathbf{x} , the predicted value is compared with a threshold value, e.g. positive if $y > 0.5$.
- The value y represents the likelihood of being positive.
- The odds $\frac{y}{1-y}$ represent the ratio of likelihoods for being positive or negative.
- Set the decision rule as following:
If the odds are greater than one ($\frac{y}{1-y} > 1$), predicted as positive.

Rescaling (2/2)

- To solve Class Imbalance let m^+ and m^- denote the number of positive or negative samples.
- The observed class ratio $\frac{m^+}{m^-}$ represents the ground-truth ratio.
- If the predicted odds of a new sample are higher than the observed odds ($\frac{y}{1-y} > \frac{m^+}{m^-}$) the new sample is classified as positive.
- Therefore, we have to adjust the prediction value of our class, as:

$$\frac{y'}{1 - y'} = \frac{y}{1 - y} \times \frac{m^+}{m^-}. \quad (2.22)$$

Implementation of Rescaling

- Though the idea of rescaling is simple, the implementation is non-trivial. Overall, there are three major rescaling approaches:
 - ① Perform **undersampling** on the negative class: To balance the classes, negative samples are selectively dropped.
 - ② Perform **oversampling** on the positive class: To balance the classes, the number of positive samples will be increased.
 - ③ **Threshold moving:** The original training set is used for learning, but equation (2.22) is used as the decision process.

Advantages and Disadvantages

- If we perform undersampling on the negative class, valuable information can get lost if the negative samples are discarded randomly.
- Looking at the computational costs, the costs for undersampling are much lower than the costs for oversampling.
- Rescaling is also used for the basis of **cost-sensitive learning**, if $\frac{m^-}{m^+}$ is replaced with $\frac{\text{cost}^+}{\text{cost}^-}$ in (2.22). Then cost^+ is the cost of misclassifying positive as negative, cost^- is the cost of misclassifying negative as positive.

Summary

In this chapter we have learned

- how linear regressions work in the machine learning context,
- how to apply the logistic regression and the linear discriminant analysis, and
- about two further classification problems (the multiclass classification and the class imbalance problem).

Introduction to Machine Learning: Theory and Application

Chapter 3

Prof. Dr. Alexander Szimayer

Universität Hamburg

Summer Term 2024

3. Decision Trees

In this chapter we

- introduce the concept of decision trees.
- discuss the splitting process of decision trees and how they are pruned.
- thematize how decision trees handle continues and missing values.
- discuss multivariate decision trees.

3.1 Basic Process (1/2)

- Decision trees are a popular class of machine learning methods.
- We usually go through a **series of judgments or sub-decisions**.
- Every question asked in the decision process is a test on one feature.
- Example: decide a binary classification task
Q: Is this instance positive ?
- Every test leads to either the conclusion or an additional test.
- The conclusions at the end of the decision process correspond to the possible classifications.

3.1 Basic Process (2/2)

- A decision tree consists of one **root node**, multiple **internal nodes**, and multiple **leaf nodes**.
 - The **root node** is the start node.
 - Each **internal node** denotes a test on an feature.
 - Leaf nodes, also called **terminal nodes**, are nodes that don't split into more nodes and hold a class label.
- Each path from root node to leaf node is a **decision sequence**.
- When there is only one splitting in the entire tree, the decision tree is called a **decision stump**.
- The construction of decision trees follows the **divide-and-conquer strategy**.

Example: Decision Tree

- Decide whether the watermelon is ripe:

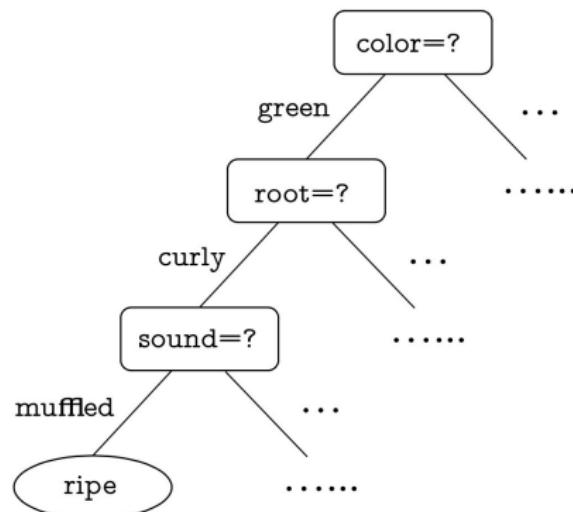


Figure 3.1: Decision tree of the watermelon problem.

- Produces a tree that can generalize to predict unseen samples.

Example: Algorithm

Input: Training set $D = \{(x_1, y_1), (x_2, y_2), \dots, (x_m, y_m)\}$;
Feature set $A = \{a_1, a_2, \dots, a_d\}$.

Process: Function TreeGenerate(D, A)

```
1: Generate node  $i$ ;  
2: if All samples in  $D$  belong to the same class  $C$  then  
3:   Mark node  $i$  as a class  $C$  leaf node; return  
4: end if  
5: if  $A = \emptyset$  OR all samples in  $D$  take the same value on  $A$  then  
6:   Mark node  $i$  as a leaf node, and its class label is the majority class  
   in  $D$ ; return  
7: end if  
8: Select the optimal splitting feature  $a_*$  from  $A$ ;  
9: for each value  $a_*^v$  in  $a_*$  do  
10:   Generate a branch for node  $i$ ; Let  $D_v$  be the subset of samples  
    taking value  $a_*^v$  on  $a_*$ ;  
11:   if  $D_v$  is empty then  
12:     Mark this child node as a leaf node, and label it with the major-  
      ity class in  $D$ ; return  
13:   else  
14:     Use TreeGenerate( $D_v, A \setminus \{a_*\}$ ) as the child node.  
15:   end if  
16: end for
```

Output: A decision tree with root node i .

Recursive return, case (1).

Recursive return, case (2).

We will discuss the optimal split
selection in the next section.

Recursive return, case (3).

Exclude a_* from A .

Figure 3.2: Decision tree pseudocode.

- As shown in this algorithm, the tree is generated **recursively**.

3.2 Split Selection

- As the splitting process proceeds, we wish more samples within each node to belong to a single class, that is, increasing the **purity** of each node.
- One of the most commonly used measures for purity is **information entropy**, or simply **entropy**.
- Let p_k denotes the proportion of the k th class in the current data set D , where $k = 1, 2, \dots, |\mathcal{Y}|$. Then, the entropy is defined as:

$$\text{Ent}(D) = - \sum_{k=1}^{|\mathcal{Y}|} p_k \log_2 p_k.$$

- The lower $\text{Ent}(D)$, the higher the purity of D .

Example: Watermelon Data Set

Table 3.1: Watermelon data set 2.0

ID	color	root	sound	texture	umbilicus	surface	ripe
1	green	curly	muffled	clear	hollow	hard	true
2	dark	curly	dull	clear	hollow	hard	true
3	dark	curly	muffled	clear	hollow	hard	true
4	green	curly	dull	clear	hollow	hard	true
5	light	curly	muffled	clear	hollow	hard	true
6	green	slightly curly	muffled	clear	slightly hollow	soft	true
7	dark	slightly curly	muffled	slightly blurry	slightly hollow	soft	true
8	dark	slightly curly	muffled	clear	slightly hollow	hard	true
9	dark	slightly curly	dull	slightly blurry	slightly hollow	hard	false
10	green	straight	crisp	clear	flat	soft	false
11	light	straight	crisp	blurry	flat	hard	false
12	light	curly	muffled	blurry	flat	soft	false
13	green	slightly curly	muffled	slightly blurry	hollow	hard	false
14	light	slightly curly	dull	slightly blurry	hollow	hard	false
15	dark	slightly curly	muffled	clear	slightly hollow	soft	false
16	light	curly	muffled	blurry	flat	hard	false
17	green	curly	dull	slightly blurry	slightly hollow	hard	false

- This example is analyzed in more detail in tutorial 3, P-Exercise 3.1.

Information Gain (1/2)

- The discrete feature a has V possible values $\{a^1, a^2, \dots, a^V\}$. Splitting D by a produces V **child nodes**, where the v th child node D^v includes all samples in D taking value a^v for feature a .
- Since there are different numbers of samples in the child nodes, a **weight** $\frac{|D^v|}{|D|}$ is assigned to reflect the importance of each node.
- The **information gain** of splitting data set D with feature a is calculated as:

$$\text{Gain}(D, a) = \text{Ent}(D) - \sum_{v=1}^V \frac{|D^v|}{|D|} \text{Ent}(D^v).$$

- The Iterative Dichotomiser ID3 algorithm takes the information gain as guideline for selecting the split features.

Information Gain (2/2)

- The higher the information gain, the more purity improvement we can expect by splitting D with feature a .
- Repeat calculating the information gain for all features and every node, to obtain the final decision tree.
- The information gain criterion is **biased** towards features with more possible values (example: the feature ID in table 3.1 is the most informative feature on the training data, but a poor feature for generalization on the test data).

Gain Ratio

- To reduce this bias, the **gain ratio** is used to select features instead of employing information gain.
- The **gain ratio** of feature a is defined as:

$$\text{Gain ratio}(D, a) = \frac{\text{Gain}(D, a)}{IV(a)}.$$

- The **intrinsic value** $IV(a)$ is large when feature a has many possible values:

$$IV(a) = - \sum_{v=1}^V \frac{|D^v|}{|D|} \log_2 \frac{|D^v|}{|D|}.$$

- The **gain ratio** is **biased** towards features with fewer possible values.
 - The information gain criterion is used by the C4.5 algorithm.
- This is analyzed in more detail in tutorial 3, P-Exercise 3.2.

Gini Index (1/2)

- As alternative to information gain and gain ratios, the Gini index is employed by the Classification and Regression Tree (**CART**) algorithm, to select the splitting feature.
- The Gini value of data set D is defined as:

$$\begin{aligned}\text{Gini}(D) &= \sum_{k=1}^{|\mathcal{Y}|} \sum_{k' \neq k} p_k p_{k'} \\ &= 1 - \sum_{k=1}^{|\mathcal{Y}|} p_k^2.\end{aligned}$$

- Gini represents the likelihood of two samples randomly selected from data set D belonging to different classes.
- The lower the $\text{Gini}(D)$, the higher the purity of D .

Gini Index (2/2)

- The Gini index of feature a is defined as:

$$\text{Gini index}(D, a) = \sum_{v=1}^V \frac{|D^v|}{|D|} \text{Gini}(D^v).$$

- Select the feature with the lowest Gini index as splitting feature.
- This is analyzed in more detail in tutorial 3, P-Exercise 3.4.

3.3 Pruning

- Pruning is the primary strategy of decision tree learning algorithms to deal with overfitting.
 - The general pruning strategies include:
 - **Pre-pruning** evaluates the improvement of generalization ability of each split and cancels a split if the improvement is small, that is, the node is marked as a leaf node.
 - **Post-pruning** re-examines the non-leaf nodes of a fully grown decision tree, and a node is replaced with a leaf node if the replacement leads to improved generalization ability.
 - On the one hand the pruning process may reduce some branches of the decision tree, which reduces the risk of overfitting. Whereas on the other hand this may introduce the risk of underfitting.
 - Test if the generalization ability has been improved.
- This is analyzed in more detail in tutorial 3, P-Exercise 3.4.

3.4 Continuous and Missing Values

- In practice, we often deal with continuous values, such as temperature.
- In addition, it often happens that individual values of the data set are missing. Taking medical diagnosis data as an example, feature values could be unavailable due to privacy concerns.
- For these two cases, we need additional tools to deal with them.

Handling Continuous Values (1/3)

Table 3.2: Watermelon data set 3.0

ID	color	root	sound	texture	umbilicus	surface	density	sugar	ripe
1	green	curly	muffled	clear	hollow	hard	0.697	0.460	true
2	dark	curly	dull	clear	hollow	hard	0.774	0.376	true
3	dark	curly	muffled	clear	hollow	hard	0.634	0.264	true
4	green	curly	dull	clear	hollow	hard	0.608	0.318	true
5	light	curly	muffled	clear	hollow	hard	0.556	0.215	true
6	green	slightly curly	muffled	clear	slightly hollow	soft	0.403	0.237	true
7	dark	slightly curly	muffled	slightly blurry	slightly hollow	soft	0.481	0.149	true
8	dark	slightly curly	muffled	clear	slightly hollow	hard	0.437	0.211	true
9	dark	slightly curly	dull	slightly blurry	slightly hollow	hard	0.666	0.091	false
10	green	straight	crisp	clear	flat	soft	0.243	0.267	false
11	light	straight	crisp	blurry	flat	hard	0.245	0.057	false
12	light	curly	muffled	blurry	flat	soft	0.343	0.099	false
13	green	slightly curly	muffled	slightly blurry	hollow	hard	0.639	0.161	false
14	light	slightly curly	dull	slightly blurry	hollow	hard	0.657	0.198	false
15	dark	slightly curly	muffled	clear	slightly hollow	soft	0.360	0.370	false
16	light	curly	muffled	blurry	flat	hard	0.593	0.042	false
17	green	curly	dull	slightly blurry	slightly hollow	hard	0.719	0.103	false

- This table includes two features with continuous values.

Handling Continuous Values (2/3)

- We cannot directly split nodes with continuous features since their values are infinite.
- A discretization strategy is the **bi-partition** method.
 - Given a data set D and a continuous feature a , suppose n values of a are observed in D , and we sort these values in ascending order, denotes by $\{a^1, a^2, \dots, a^n\}$.
 - With a split point t , D is partitioned into the subsets D_t^- and D_t^+ , where D_t^- includes the samples with the value of a not greater than t , and D_t^+ includes the samples with the value of a greater than t .
 - Set of candidate split points:

$$T_a = \left\{ \frac{a^i + a^{i+1}}{2} \mid 1 \leq i \leq n - 1 \right\}.$$

Handling Continuous Values (3/3)

- $\text{Gain}(D, a, t)$ is the **information gain** of bi-partitioning D by t , and the split point with the largest $\text{Gain}(D, a, t)$ is selected.

$$\begin{aligned}\text{Gain}(D, a) &= \max_{t \in T_a} \text{Gain}(D, a, t) \\ &= \max_{t \in T_a} \text{Ent}(D) - \sum_{\lambda \in \{-, +\}} \frac{|D_t^\lambda|}{|D|} \text{Ent}(D_t^\lambda).\end{aligned}$$

- Unlike discrete features, a continuous feature can be used as a splitting feature more than once in a decision sequence.

Handling Missing Values (1/2)

- In practice, data is often incomplete, that is, some feature values are missing.
- Learning from incomplete samples raises two problems:
Q.1: How to choose the splitting features when there are missing values ?
Q.2: How to split a sample with the splitting feature value missing ?
- Given a training set D and a feature a , let \tilde{D} be the subset of samples in D that has values of a .

Handling Missing Values (2/2)

Table 3.3: Watermelon data set 2.0 α

ID	color	root	sound	texture	umbilicus	surface	ripe
1	-	curly	muffled	clear	hollow	hard	true
2	dark	curly	dull	clear	hollow	-	true
3	dark	curly	-	clear	hollow	hard	true
4	green	curly	dull	clear	hollow	hard	true
5	-	curly	muffled	clear	hollow	hard	true
6	green	slightly curly	muffled	clear	-	soft	true
7	dark	slightly curly	muffled	slightly blurry	slightly hollow	soft	true
8	dark	slightly curly	muffled	-	slightly hollow	hard	true
9	dark	-	dull	slightly blurry	slightly hollow	hard	false
10	green	straight	crisp	-	flat	soft	false
11	light	straight	crisp	blurry	flat	-	false
12	light	curly	-	blurry	flat	soft	false
13	-	slightly curly	muffled	slightly blurry	hollow	hard	false
14	light	slightly curly	dull	slightly blurry	hollow	hard	false
15	dark	slightly curly	muffled	clear	-	soft	false
16	light	curly	muffled	blurry	flat	hard	false
17	green	-	dull	slightly blurry	slightly hollow	hard	false

Dealing with the First Problem (1/3)

Q.1: How to choose the splitting features when there are missing values ?

- We can simply use \tilde{D} to evaluate a .
- Let $\{a^1, a^2, \dots, a^V\}$ denotes the V possible values of a , \tilde{D}^v denotes the subset of samples in \tilde{D} taking the value a^v , and \tilde{D}_k denotes the subset of samples in \tilde{D} belonging to the k th class, where $k = 1, 2, \dots, |\mathcal{Y}|$.
- Then, we have $\tilde{D} = \bigcup_{k=1}^{|\mathcal{Y}|} \tilde{D}_k$ and $\tilde{D} = \bigcup_{v=1}^V \tilde{D}^v$.

Dealing with the First Problem (2/3)

- We assign a **weight** w_x to each sample x , and define:

$$\rho = \frac{\sum_{x \in \tilde{D}} w_x}{\sum_{x \in D} w_x},$$
$$\tilde{p}_k = \frac{\sum_{x \in \tilde{D}_k} w_x}{\sum_{x \in \tilde{D}} w_x} \quad (1 \leq k \leq |\mathcal{Y}|),$$
$$\tilde{r}_v = \frac{\sum_{x \in \tilde{D}^v} w_x}{\sum_{x \in \tilde{D}} w_x} \quad (1 \leq v \leq V).$$

- Variable ρ represents the proportion of samples without missing values.
- Variable \tilde{p}_k represents the proportion of the k th class in all samples without missing values.
- Variable \tilde{r}_v represents the proportion of samples taking feature value a^v in all samples without missing values.
- Then, we have $\sum_{k=1}^{|\mathcal{Y}|} \tilde{p}_k = 1$ and $\sum_{v=1}^V \tilde{r}_v = 1$.

Dealing with the First Problem (3/3)

- We extend the **information gain** to:

$$\begin{aligned}\text{Gain}(D, a) &= \rho \times \text{Gain}(\tilde{D}, a) \\ &= \rho \times \left(\text{Ent}(\tilde{D}) - \sum_{v=1}^V \tilde{r}_v \text{Ent}(\tilde{D}^v) \right),\end{aligned}$$

where,

$$\text{Ent}(\tilde{D}) = - \sum_{k=1}^{|\mathcal{Y}|} \tilde{p}_k \log_2 \tilde{p}_k.$$

- The extended information gain (or its extended alternatives: gain ratio and Gini index) can then be used to determine the optimal split feature, as shown in section 3.2.

Dealing with the Second Problem (1/2)

Q.2: How to split a sample with the splitting feature value missing ?

- When the value of *a* is known, we place the sample x into the corresponding child node without changing its weight w_x .
- When the value of *a* is unknown, we place the sample x into all child nodes, and set its weight in the child node of value a^v to \tilde{r}_{vx} .
- In other words, we place the same sample into different child nodes with different probabilities.
- The splitting process proceeds **recursively**.

Dealing with the Second Problem (2/2)

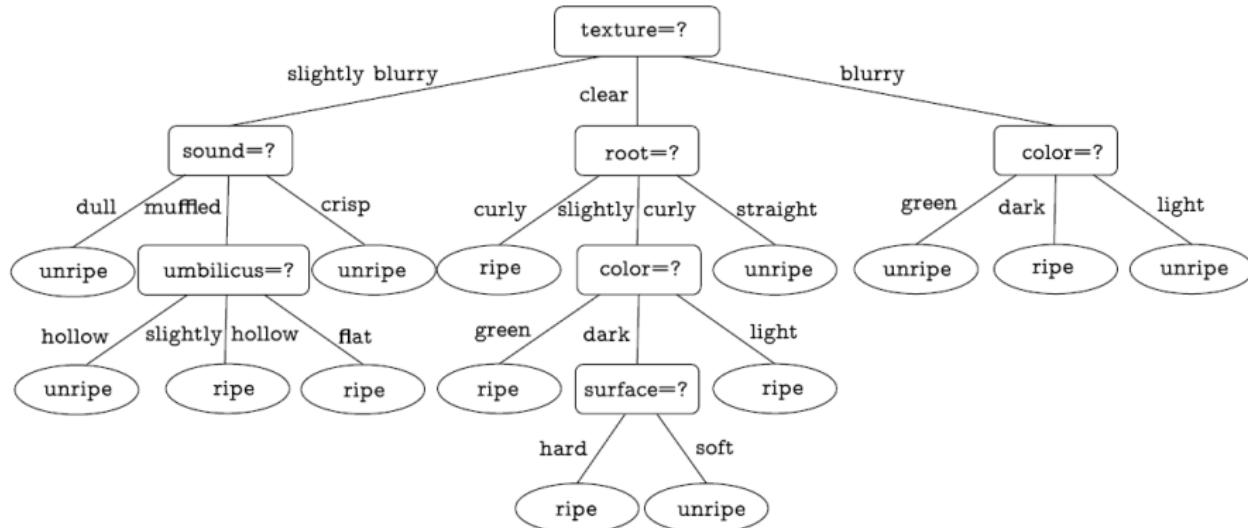


Figure 3.3: The information gain-based decision tree generated from table 3.3.

3.5 Multivariate Decision Trees (1/4)

- If we regard each feature as a coordinate axis in the coordinate space, a sample with d features corresponds to a point in the d -dimensional space.
- Classifying samples is then about finding the **decision boundaries** in this space to separate the samples of different classes.
- The decision boundaries have a distinct characteristic: axis-parallel, that is, the boundaries are multiple segments parallel to the axes.

3.5 Multivariate Decision Trees (2/4)

ID	density	sugar	ripe
1	0.697	0.460	true
2	0.774	0.376	true
3	0.634	0.264	true
4	0.608	0.318	true
5	0.556	0.215	true
6	0.403	0.237	true
7	0.481	0.149	true
8	0.437	0.211	true
9	0.666	0.091	false
10	0.243	0.267	false
11	0.245	0.057	false
12	0.343	0.099	false
13	0.639	0.161	false
14	0.657	0.198	false
15	0.360	0.370	false
16	0.593	0.042	false
17	0.719	0.103	false

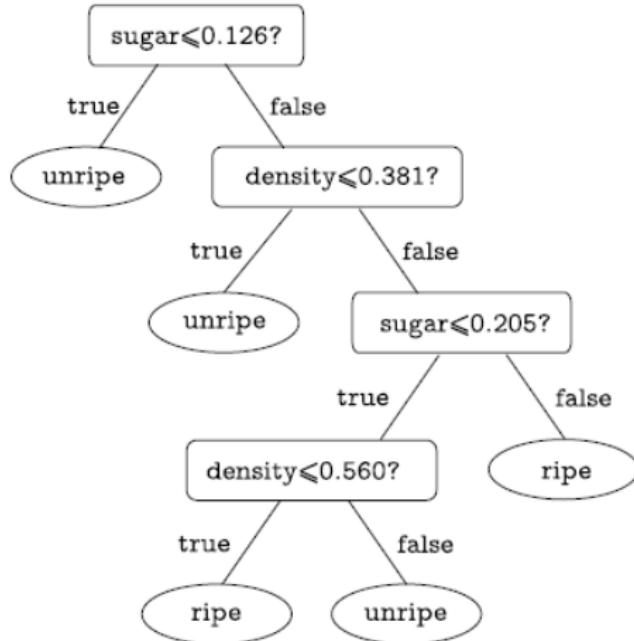


Figure 3.4: The decision tree generated from table on the left.

3.5 Multivariate Decision Trees (3/4)

- If we can make the decision boundaries **oblique**, then the decision tree model can be significantly simplified.

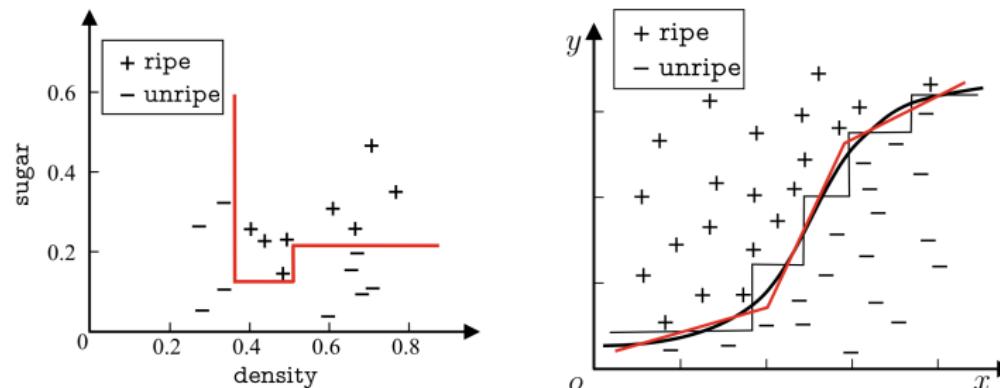


Figure 3.5: Parallel decision boundaries vs. oblique decision boundaries.

- **Multivariate decision tree** (also oblique decision tree) enables oblique partitions or even more complicated decision boundaries.

3.5 Multivariate Decision Trees (4/4)

- With oblique boundaries, each non-leaf node is no longer a test for a particular feature but a **linear combination** of features.
- Each non-leaf node is a linear classifier in the form of $\sum_{i=1}^d w_i a_i = t$, where w_i is the weight of feature a_i and w_i and t are learned from the data set and feature set of the node.
- Unlike the traditional univariate decision tree, the learning process of multivariate decision tree does not look for an optimal splitting feature but tries to establish a suitable linear classifier.

Summary

In this chapter we have learned

- how the basic process of decision trees works.
- what entropy, information gain, gain ratio and gini index is.
- which primary strategy decision trees use to deal with overfitting.
- how to handle continuous or missing values.
- the difference between univariate and multivariate decision trees.

Introduction to Machine Learning: Theory and Application

Chapter 4

Prof. Dr. Alexander Szimayer

Universität Hamburg

Summer Term 2024

4. Ensemble Learning

In this chapter we

- learn how to combine a set of individual learners to one ensemble learning method.
- discuss the two concepts boosting and bagging.
- study different combination strategies.

4.1 Individual and Ensemble

- **Ensemble learning**, also known as **multiple classifier system** and **committee-based learning**, trains and combines multiple learners to solve a learning problem.

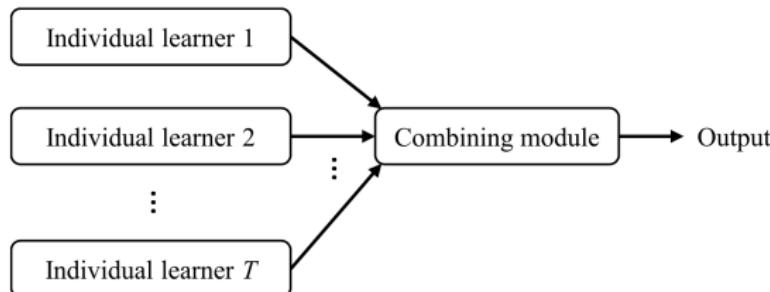


Figure 4.1: Workflow of ensemble learning.

- An ensemble is said to be **homogeneous** if all individual learners are of the same type, e.g. a decision tree ensemble contains only decision trees.

Homogeneous and Heterogeneous Ensembles

- Homogeneous ensembles: individual learners are all of the same type and are called **base learners**. The corresponding learning algorithms are called base learning algorithms.
- Heterogeneous ensembles: individual learners are not all of the same type and are called component learners or **individual learners**. There is no single base learner or base learning algorithm.
- By combining multiple learners, the generalization ability of an ensemble is often much stronger than that of an individual learner, and this is especially true for **weak learners**.
- Weak learner performance is relatively poor, usually its accuracy is just slightly above chance.

How can an ensemble produce better performance than the best individual learner?

- Taking binary classification as an example, where the ticks indicate the correct classifications, and the crosses indicate the incorrect classifications of each classifier h_i :

	Testing sample 1	Testing sample 2	Testing sample 3		Testing sample 1	Testing sample 2	Testing sample 3		Testing sample 1	Testing sample 2	Testing sample 3
h_1	✓	✓	✗	h_1	✓	✓	✗	h_1	✓	✗	✗
h_2	✗	✓	✓	h_2	✓	✓	✗	h_2	✗	✓	✗
h_3	✓	✗	✓	h_3	✓	✓	✗	h_3	✗	✗	✓
Ensemble	✓	✓	✓	Ensemble	✓	✓	✗	Ensemble	✗	✗	✗

(a) Ensemble helps.

(b) Ensemble doesn't help.

(c) Ensemble hurts.

Figure 4.2: Individual learners need to be accurate and diverse.

- Individual learners must be slightly better than chance and have **diversity** to result in a good ensemble learner.

Simple Analysis with Binary Classification (1/2)

- Now we do a simple analysis with binary classification:
 - $y \in \{-1, +1\}$ and ground-truth function is f ,
 - Error rate of each base learner is ϵ .
- For each base learner h_i , we have:

$$P(h_i(\mathbf{x}) \neq f(\mathbf{x})) = \epsilon,$$

where \mathbf{x} is a vector.

- Suppose ensemble learning combines the T base learners by voting, then the ensemble will make a correct classification if more than half of the base learners are correct (we assume T is odd for ease of discussion):

$$F(\mathbf{x}) = \text{sign} \left(\sum_{i=1}^T h_i(\mathbf{x}) \right).$$

Simple Analysis with Binary Classification (2/2)

- Assuming the error rates of base learners are independent, then, from **Hoeffding's inequality**, the error rate of the ensemble is:

$$\begin{aligned} P(F(\mathbf{x}) \neq f(\mathbf{x})) &= \sum_{k=0}^{\lfloor T/2 \rfloor} \binom{T}{k} (1-\epsilon)^k \epsilon^{T-k} \\ &\leq \exp\left(-\frac{1}{2}T(1-2\epsilon)^2\right), \end{aligned}$$

where it is assumed that the error rates are **independent** which is a critical assumption in practice.

- As the number of base learners T in the ensemble increases, the error rate decreases exponentially and eventually approaches zero.
- Accuracy** and **diversity** are two conflicting properties of individual learners. Generally, when the accuracy is already high, we usually need to sacrifice some accuracy if we wish to increase diversity.

Ensemble Learning Methods

- Current ensemble learning methods can be roughly grouped into two categories, depending on how the individual learners are generated.
- The first category, represented by **boosting**, creates individual learners with **strong correlations** and generates the learners sequentially.
- The second category, represented by **bagging and random forest**, creates individual learners independently and can parallelize the generation process.

4.2 Boosting

- Boosting is a family of algorithms that convert **weak learners** to **strong learners**.
- Boosting algorithms start with training a base learner and then adjust the distribution of the training samples according to the result of the base learner such that incorrectly classified samples will receive more attention by subsequent base learners.
- After training the first base learner, the second base learner is trained with the adjusted training samples, and the result is used to adjust the training sample distribution again.
- Such a process repeats until the number of base learners reaches a predefined value T , and finally, these base learners are weighted and combined.

AdaBoost Algorithm

Input: Training set $D = \{(x_1, y_1), (x_2, y_2), \dots, (x_m, y_m)\}$;

Base learner \mathcal{L} ;

Number of training rounds T .

Process:

1: $\mathcal{D}_1(x) = 1/m$;

2: **for** $t = 1, 2, \dots, T$ **do**

3: $h_t = \mathcal{L}(D, \mathcal{D}_t)$;

4: $\epsilon_t = P_{x \sim \mathcal{D}_t}(h_t(x) \neq f(x))$;

5: **if** $\epsilon_t > 0.5$ **then break**

6: $\alpha_t = \frac{1}{2} \ln \left(\frac{1-\epsilon_t}{\epsilon_t} \right)$;

7:
$$\begin{aligned} \mathcal{D}_{t+1}(x) &= \frac{\mathcal{D}_t(x)}{Z_t} \times \begin{cases} \exp(-\alpha_t), & \text{if } h_t(x) = f(x); \\ \exp(\alpha_t), & \text{if } h_t(x) \neq f(x); \end{cases} \\ &= \frac{\mathcal{D}_t(x) \exp(-\alpha_t f(x) h_t(x))}{Z_t}; \end{aligned}$$

8: **end for**

Output: $F(x) = \text{sign} \left(\sum_{t=1}^T \alpha_t h_t(x) \right)$.

Initialize the sample weight distribution.

Train classifier h_t using data set D that follows distribution \mathcal{D}_t .
Estimate the error of h_t .

Determine the weight of classifier h_t .

Update the sample distribution, where Z_t is the normalization factor that ensures \mathcal{D}_{t+1} is a valid distribution.

Figure 4.3: AdaBoost pseudocode.

Additive Model and Exponential Loss Function (1/3)

- The most well-known boosting algorithm is adaptive boosting also called **AdaBoost**.
- There are multiple ways to derive the AdaBoost algorithm, but one is using the linear combination of base learners:

$$H(\mathbf{x}) = \sum_{t=1}^T \alpha_t h_t(\mathbf{x}),$$

to minimize the **exponential loss function**:

$$\ell_{\text{exp}}(H|\mathcal{D}) = E_{x \sim \mathcal{D}} \left[e^{-f(x)H(x)} \right]. \quad (4.1)$$

where \mathcal{D} is the probability distribution of data D and α_t the respective weight of each classifier.

Additive Model and Exponential Loss Function (2/3)

- If $H(\mathbf{x})$ minimizes the exponential loss, then we consider the partial derivative of equation (4.1) with respect to $H(\mathbf{x})$,

$$\frac{\partial \ell_{\text{exp}}(H|\mathcal{D})}{\partial H(\mathbf{x})} = -e^{-H(\mathbf{x})}P(f(\mathbf{x}) = 1|\mathbf{x}) + e^{H(\mathbf{x})}P(f(\mathbf{x}) = -1|\mathbf{x}),$$

and setting it to zero gives:

$$H(\mathbf{x}) = \frac{1}{2} \ln \frac{P(f(\mathbf{x}) = 1|\mathbf{x})}{P(f(\mathbf{x}) = -1|\mathbf{x})},$$

where $P(\cdot|\mathbf{x})$ is the conditional probability mass function given \mathbf{x} .

Additive Model and Exponential Loss Function (3/3)

- Hence, we have,

$$\begin{aligned}\text{sign}(H(\mathbf{x})) &= \text{sign} \left(\frac{1}{2} \ln \frac{P(f(\mathbf{x}) = 1|\mathbf{x})}{P(f(\mathbf{x}) = -1|\mathbf{x})} \right) \\ &= \begin{cases} 1, & P(f(\mathbf{x}) = 1|\mathbf{x}) > P(f(\mathbf{x}) = -1|\mathbf{x}) \\ -1, & P(f(\mathbf{x}) = 1|\mathbf{x}) < P(f(\mathbf{x}) = -1|\mathbf{x}) \end{cases} \\ &= \arg \max_{y \in \{-1, 1\}} P(f(\mathbf{x}) = y|\mathbf{x}),\end{aligned}$$

which implies that $\text{sign}(H(\mathbf{x}))$ achieves the Bayes optimal error rate, which is loosely speaking the lowest possible classification error.

- In other words, the classification error rate is minimized when the exponential loss is minimized.
- Here we ignore the case of $P(f(\mathbf{x}) = 1|\mathbf{x}) = P(f(\mathbf{x}) = -1|\mathbf{x})$.

Minimization of the Exponential Loss Function (1/3)

- In the AdaBoost algorithm, the base learning algorithm generates the first base classifier h_1 from the original training data and then iteratively generates the subsequent base classifiers h_t and associated weights α_t .
- Once the classifier h_t is generated from the distribution \mathcal{D}_t , its weight α_t is estimated by letting $\alpha_t h_t$ minimize the exponential loss function,

$$\begin{aligned}\ell_{\text{exp}}(\alpha_t h_t | \mathcal{D}_t) &= E_{\mathbf{x} \sim \mathcal{D}_t} \left[e^{-f(\mathbf{x})\alpha_t h_t(\mathbf{x})} \right] \\ &= e^{-\alpha_t} (1 - \epsilon_t) + e^{\alpha_t} \epsilon_t,\end{aligned}$$

where $\epsilon_t = P_{\mathbf{x} \sim \mathcal{D}_t}(h_t(\mathbf{x}) \neq f(\mathbf{x}))$.

Minimization of the Exponential Loss Function (2/3)

- Setting the derivative of the exponential loss function,

$$\frac{\partial \ell_{\text{exp}}(\alpha_t h_t | \mathcal{D}_t)}{\partial \alpha_t} = -e^{-\alpha_t}(1 - \epsilon_t) + e^{\alpha_t}\epsilon_t,$$

to zero gives the optimal α_t as:

$$\alpha_t = \frac{1}{2} \ln \left(\frac{1 - \epsilon_t}{\epsilon_t} \right). \quad (4.2)$$

which is exactly the equation in line 6 of Figure 4.3 on slide 124.

Minimization of the Exponential Loss Function (3/3)

- The AdaBoost algorithm adjusts the sample distribution based on H_{t-1} such that the base learner h_t in the next round can correct some mistakes made by H_{t-1} .
- The minimization of the exponential loss function can be simplified and approximated to,

$$\begin{aligned}\ell_{\text{exp}}(H_{t-1} + h_t | \mathcal{D}) &= E_{x \sim \mathcal{D}} \left[e^{-f(\mathbf{x})H_{t-1}(\mathbf{x})} e^{-f(\mathbf{x})h_t(\mathbf{x})} \right] \\ &\simeq E_{x \sim \mathcal{D}} \left[e^{-f(\mathbf{x})H_{t-1}(\mathbf{x})} \left(1 - f(\mathbf{x})h_t(\mathbf{x}) + \frac{1}{2} \right) \right],\end{aligned}$$

since $f^2(\mathbf{x}) = h_t^2(\mathbf{x}) = 1$, the formula above can be approximated by Taylor expansion of order 2.

Ideal Classifier h_t (1/3)

- The ideal classifier is,

$$\begin{aligned} h_t(\mathbf{x}) &= \arg \min_h \ell_{\text{exp}}(H_{t-1} + h | \mathcal{D}) \\ &\simeq \arg \min_h E_{\mathbf{x} \sim \mathcal{D}} \left[e^{-f(\mathbf{x})H_{t-1}(\mathbf{x})} \left(1 - f(\mathbf{x})h(\mathbf{x}) + \frac{1}{2} \right) \right] \\ &= \arg \max_h E_{\mathbf{x} \sim \mathcal{D}} \left[e^{-f(\mathbf{x})H_{t-1}(\mathbf{x})} f(\mathbf{x})h(\mathbf{x}) \right] \\ &= \arg \max_h E_{\mathbf{x} \sim \mathcal{D}} \left[\frac{e^{-f(\mathbf{x})H_{t-1}(\mathbf{x})}}{E_{\mathbf{x} \sim \mathcal{D}} [e^{-f(\mathbf{x})H_{t-1}(\mathbf{x})}]} f(\mathbf{x})h(\mathbf{x}) \right], \end{aligned}$$

where $E_{\mathbf{x} \sim \mathcal{D}} \left[e^{-f(\mathbf{x})H_{t-1}(\mathbf{x})} \right]$ is a constant.

Ideal Classifier h_t (2/3)

- Let \mathcal{D}_t denote a distribution:

$$\mathcal{D}_t(\mathbf{x}) = \frac{\mathcal{D}(\mathbf{x})e^{-f(\mathbf{x})H_{t-1}(\mathbf{x})}}{E_{\mathbf{x} \sim \mathcal{D}} [e^{-f(\mathbf{x})H_{t-1}(\mathbf{x})}]}. \quad (4.3)$$

- According to the definition of mathematical expectation, the ideal classifier is equivalent to:

$$\begin{aligned} h(\mathbf{x}) &= \arg \max_h E_{\mathbf{x} \sim \mathcal{D}} \left[\frac{e^{-f(\mathbf{x})H_{t-1}(\mathbf{x})}}{E_{\mathbf{x} \sim \mathcal{D}} [e^{-f(\mathbf{x})H_{t-1}(\mathbf{x})}]} f(\mathbf{x}) h(\mathbf{x}) \right] \\ &= \arg \max_h E_{\mathbf{x} \sim \mathcal{D}_t} [f(\mathbf{x}) h(\mathbf{x})]. \end{aligned}$$

- Since $f(\mathbf{x}), h(\mathbf{x}) \in \{-1, +1\}$, we have,

$$f(\mathbf{x}) h(\mathbf{x}) = 1 - 2I(f(\mathbf{x}) \neq h(\mathbf{x})),$$

and the ideal classifier is:

$$h_t(\mathbf{x}) = \arg \min_h E_{\mathbf{x} \sim \mathcal{D}_t} [I(f(\mathbf{x}) \neq h(\mathbf{x}))]. \quad (4.4)$$

Ideal Classifier h_t (3/3)

- From equation (4.4), we see that the ideal classifier h_t minimizes the classification error under the distribution \mathcal{D}_t .
- Therefore, the weak classifier at round t is trained on the distribution \mathcal{D}_t , and its classification error should be less than 0.5 for \mathcal{D}_t .
- The AdaBoost algorithm can be derived by iteratively optimizing the exponential loss function based on an additive model as shown by equations (4.2) and (4.3).
- The updating rule for the data distribution D_t used in line 7 of the AdaBoost algorithm in Figure 4.3 can be derived accordingly.

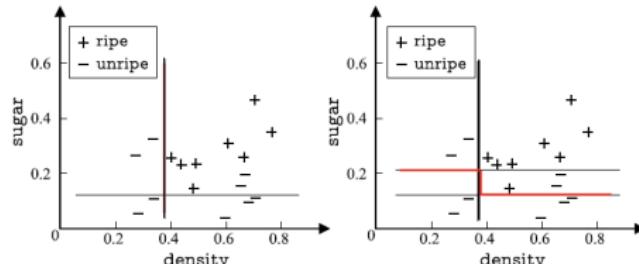
Boosting Algorithms (1/2)

- Boosting algorithms require the base learners to learn from specified sample distributions, and this is often accomplished by **re-weighting**.
- For base learning algorithms that do not accept weighted samples, **re-sampling** can be used. That is, in each round a new training set is sampled from the new sample distribution.
- In general, there is not much difference between re-weighting and re-sampling in terms of prediction performance.
- Note that in each round, there is a sanity check on whether the current base learner satisfies some basic requirements (see line 5 of the AdaBoost algorithm in Figure 4.3).

Boosting Algorithms (2/2)

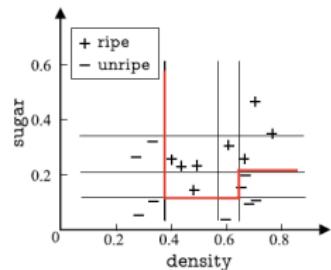
- If the requirements are not met, the current base learner is discarded and the learning process stops. In such cases, the number of rounds could still be far from the pre-specified limit T .
- However, if the re-sampling method is used, there is an option to “restart” in order to avoid early termination.
- Boosting mainly focuses on **reducing bias**, and this is why an ensemble of learners with weak generalization ability can be so powerful.

Decision Boundaries for AdaBoost



(a) 3 base learners.

(b) 5 base learners.



(c) 11 base learners.

Figure 4.4: AdaBoost on watermelon data set 3.0α for different depths of the decision tree, which is the base learner here.

4.3 Bagging and Random Forest

- We know that the generalization ability of an ensemble depends on the **independence** of base learners.
- One way of creating different base learners is to partition the original training set into several **non-overlapped subsets** and use each subset to train a base learner.
- However, if the subsets are totally different, then it implies that each subset contains only a small portion of the original training set, possibly leading to poor base learners.
- Since a good ensemble requires each base learner to be reasonably good, we often allow the subsets to **overlap** such that each of them contains sufficient samples.

4.3.1 Bagging

- Bagging is a representative method of **parallel ensemble learning** based on **bootstrap sampling**.
- Approximately 63.2% of the original samples will appear in the data set (c.f. bootstrapping in chapter 1)
- Basic workflow of Bagging:
 - Applying the bootstrapping process T times produces T data sets, and each contains m samples. Then, the base learners are trained on these data sets and combined.
 - When combining the predictions of base learners, Bagging adopts the simple **voting method** for classification tasks and the simple **averaging method** for regression tasks.
 - When multiple classes receive the same number of votes, we can choose one at random or further investigate the confidence of votes.

Bagging Algorithm

\mathcal{D}_{bs} is the distribution of a data set generated by bootstrap.

Input: Training set: $D = \{(x_1, y_1), (x_2, y_2), \dots, (x_m, y_m)\}$;
Base learning algorithm \mathcal{L} ;
Number of training rounds T .

Process:

- 1: **for** $t = 1, 2, \dots, T$ **do**
- 2: $h_t = \mathcal{L}(D, \mathcal{D}_{bs})$.
- 3: **end for**

Output: $H(x) = \arg \max_{y \in \mathcal{Y}} \sum_{t=1}^T \mathbb{I}(h_t(x) = y)$.

Figure 4.5: Bagging pseudocode.

Complexity of Bagging

- Suppose that the computational complexity of a base learner is $O(m)$, then the **complexity of bagging** is roughly $T(O(m) + O(s))$, where $O(s)$ is the complexity of voting or averaging.
- Since the complexity $O(s)$ is low and T is a constant that is often not too large, Bagging has the same order of complexity as the base learner, that is, bagging is an efficient ensemble learning algorithm.

Advantages of Bagging

- ① Unlike the standard AdaBoost, which only applies to binary classification, Bagging can be applied to **multiclass classification** and **regression** without modification.
- ② Since each base learner only uses roughly 63.2% of the original training samples for training, the remaining 36.8% samples can be used as a validation set to get an **out-of-bag estimate** of the generalization ability.
- ③ Bagging helps to **reduce the variance**, and this is particularly useful for unpruned decision trees or neural networks that are unstable to data manipulation.

Out-of-Bag Estimate

- To get the out-of-bag estimate, we need to track the training samples used by each base learner.
- Let D_t denote the set of samples used by the learner h_t , and $H^{\text{oob}}(x)$ denote the **out-of-bag prediction** of the sample x , that is, considering only the predictions made by base learners that did not use the sample x for training.
- Then we have,

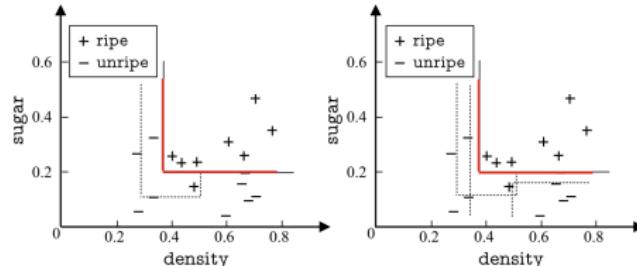
$$H^{\text{oob}}(x) = \arg \max_{y \in \mathcal{Y}} \sum_{t=1}^T I(h_t(x) = y) \cdot I(x \notin D_t),$$

and the out-of-bag estimate of the **generalization error** of bagging is,

$$\epsilon^{\text{oob}} = \frac{1}{|D|} \sum_{(x,y) \in D} I(H^{\text{oob}}(x) \neq y).$$

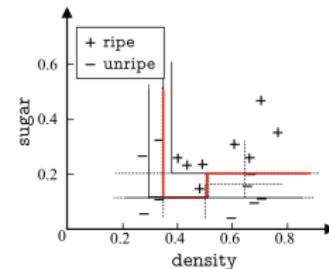
- Bagging helps to reduce the variance.

Decision Boundaries for Boosting



(a) 3 base learners.

(b) 5 base learners.



(c) 11 base learners.

Figure 4.6: Bagging on watermelon data set 3.0α for different depths of the decision tree, which is the base learner.

4.3.2 Random Forest

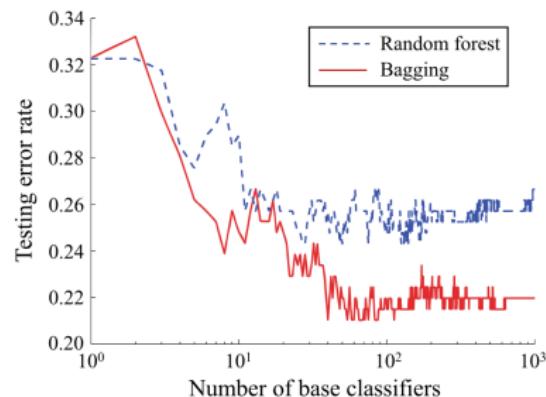
- Random Forest (RF) is an extension of bagging, where **randomized feature selection** is introduced on top of bagging.
- Traditional decision trees select an optimal split feature from the entire feature set d of each node, whereas RF selects from a subset of k features randomly generated from the feature set of the node.
- The parameter k controls the **randomness**, where the splitting is the same as in traditional decision trees if $k = d$, and a split feature is randomly selected if $k = 1$.
- Typically k is:

$$k = \log_2 d.$$

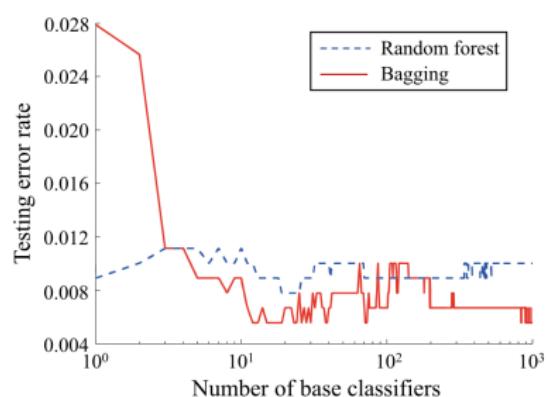
Properties of Random Forest

- ① With a small modification, RF introduces feature-based "diversity" to bagging by feature manipulation, while bagging considers sample-based "diversity" only.
- ② RF further enlarges the difference between base learners, which leads to ensembles with better generalization ability.
- ③ The convergence property of RF is comparable to that of Bagging.
- ④ RF often converges to a lower generalization error after adding more base learners to the ensemble.

Impact of the Ensemble Size on RF and Bagging



(a) The glass data set.



(b) The auto-mpg data set.

Figure 4.7: Impact of ensemble size on Rf and bagging on two different datasets.

- RF usually starts with a poor performance at the beginning, especially when there is only one base learner in the ensemble since the feature manipulation reduces the performance of each base learner.

4.4 Combination Strategies

- Combining individual learners is beneficial from three perspectives:
 - Statistical perspective:** combining multiple learners will reduce the risk of incorrectly choosing a single poor learner.
 - Computational perspective:** by repeating the learning process multiple times, the risk of being stuck in a terrible local optimum is reduced.
 - Representational perspective:** the hypothesis space extends by combining multiple learners, and hence it is more likely to find a better approximation to the ground-truth hypothesis.

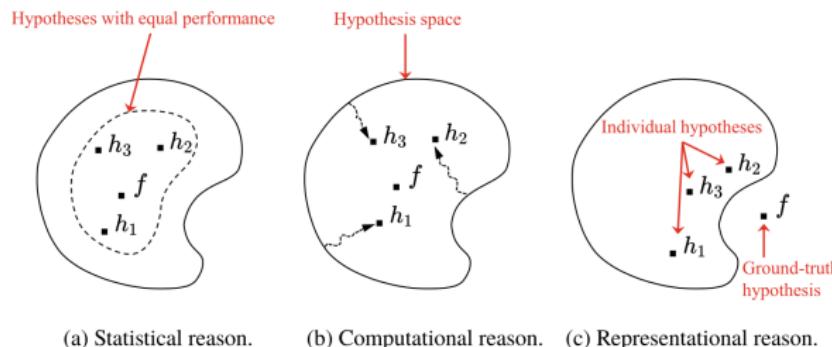


Figure 4.8: Benefits of combining learners.

Averaging (1/2)

- Suppose an ensemble contains T individual learners $\{h_1, h_2, \dots, h_T\}$, where $h_i(\mathbf{x})$ is the output of h_i on sample \mathbf{x} .
- Averaging is the most commonly used **combination strategy** for numerical output $h_i(\mathbf{x}) \in R$.
- Typical averaging methods include,
 - **Simple averaging:** $H(\mathbf{x}) = \frac{1}{T} \sum_{i=1}^T h_i(\mathbf{x})$,
 - **Weighted averaging:** $H(\mathbf{x}) = \sum_{i=1}^T w_i h_i(\mathbf{x})$,where w_i is the weight of individual learner h_i , and typically $w_i \geq 0$, $\sum_{i=1}^T w_i = 1$.
- Simple averaging is a special case of weighted averaging, $w_i = \frac{1}{T}$.

Averaging (2/2)

- Other combination methods than averaging can all be viewed as its special cases or variants.
- Different ensemble learning methods can be regarded as different ways of assigning the weights.
- Typically, weights are learned from training data. However, the learned weights are often unreliable due to data insufficiency or noise.
- This is particularly true for large ensembles with many base learners because trying to learn too many weights can easily lead to overfitting.
- The **weighted averaging method** is a better choice when individual learners have considerable different performance.
- The **simple averaging method** is preferred when individual learners share similar performance.

Voting

- For classification, a learner h_i predicts a class label from a set of N class labels $\{c_1, c_2, \dots, c_N\}$, and a common combination strategy is voting.
- Let the N -dimensional vector $(h_i^1(\mathbf{x}); h_i^2(\mathbf{x}); \dots; h_i^N(\mathbf{x}))$ denote the output on h_i on the sample \mathbf{x} , where $h_i^j(\mathbf{x})$ is the output of h_i on \mathbf{x} for the class label c_j .
- The following voting methods are defined:
 - Majority voting,
 - Plurality voting,
 - Weighted voting.

Majority Voting

- Output the class label that receives more than half of the votes or refuses to predict if none of the class labels receive more than half of the votes.

$$H(\mathbf{x}) = \begin{cases} c_j, & \text{if } \sum_{i=1}^T h_i^j(\mathbf{x}) > 0, 5 \sum_{k=1}^N \sum_{i=1}^T h_i^k(\mathbf{x}); \\ \text{reject}, & \text{otherwise.} \end{cases}$$

Plurality Voting

- Output the class label that receives the most votes and randomly select one in case of a tie.

$$H(\mathbf{x}) = c_{\arg \max_j \sum_{i=1}^T h_i^j(\mathbf{x})}.$$

Weighted Voting

- A plurality voting with weights assigned to learners, where w_i is the weight of h_i , and typically $w_i \geq 0$ and $\sum_{i=1}^T w_i = 1$ like the constraints on w_i in weighted averaging.

$$H(\mathbf{x}) = c_{\arg \max_j \sum_{i=1}^T w_i h_i^j(\mathbf{x})}.$$

Voting Methods (1/2)

- The standard majority voting offers a 'reject' option, which is an effective mechanism for tasks requiring reliability.
- If it is compulsory to make a prediction, then plurality voting can be used instead.
- For tasks that do not allow rejections, both the majority and the plurality voting methods are called majority voting, or just voting.
- Two common value types are:
 - Class label $h_i^j(x) \in \{0, 1\}$:
 - The output is 1 if h_i predicts the sample x as class c_j , and 0 otherwise.
 - The corresponding voting is known as **hard voting**.
 - Class probability $h_i^j(x) \in [0, 1]$:
 - An estimate of the posterior probability $P(c_j|x)$.
 - The corresponding voting is known as **soft voting**.

Voting Methods (2/2)

- For some learners, the class labels come with confidence values that can be converted into class probabilities.
- If the confidence values are not normalized, then **calibration techniques** need to be applied before using those as probabilities.
- Note that the class probabilities are not comparable if different types of base learners are used.
- In such cases, the class probabilities can be converted into class labels before voting, e.g., setting the largest $h_i^j(x)$ as 1 and the others as 0.

Combining by Learning

- Combining by learning is a combination strategy using a meta-learner to combine the individual learners.
- A representative of such methods is **stacking**, where the individual learners are called **first-level learners** and the learners performing the combination are called **second-level learners** or **meta-learners**.

Stacking Process

- Stacking starts by training the first-level learners using the original training set and then 'generating' a new data set to train the second-level learner.
- In the new data set, the outputs of the **first-level learners** are used as the input features, while the labels from the original training set remain unchanged.
- There would be a high risk of overfitting if the generated training set contains the exact training samples used by the first-level learners (e.g. use cross-validation to prevent).

Stacking Algorithm

Input: Training set $D = \{(x_1, y_1), (x_2, y_2), \dots, (x_m, y_m)\}$;
First-level learning algorithms $\mathcal{L}_1, \mathcal{L}_2, \dots, \mathcal{L}_T$;
Second-level learning algorithm \mathcal{L} .

Process:

```
1: for  $t = 1, 2, \dots, T$  do
2:    $h_t = \mathcal{L}_t(D)$ ;                                Generate first-level learner  $h_t$ 
3: end for                                         using first-level learning
4:  $D' = \emptyset$ ;                                 Generate second-level training
5: for  $i = 1, 2, \dots, m$  do                         set.
6:   for  $t = 1, 2, \dots, T$  do
7:      $z_{it} = h_t(x_i)$ ;                            Generate second-level learner  $h'$ 
8:   end for                                         using second-level learning
9:    $D' = D' \cup ((z_{i1}, z_{i2}, \dots, z_{iT}), y_i)$ ;    algorithm on  $D'$ .
10: end for
11:  $h' = \mathcal{L}(D')$ .
```

Output: $H(x) = h'(h_1(x), h_2(x), \dots, h_T(x))$.

Figure 4.9: Stacking pseudocode.

k-fold Cross-Validation as Example

- The original training set is partitioned into k roughly equal-sized partitions D_1, D_2, \dots, D_k .
- Denote D_j and $\overline{D}_j = D \setminus D_j$ as the testing set and the training set of the j th fold, respectively.
- For T first-level learning algorithms, the first-level learner $h_t^{(j)}$ is obtained by applying the t th learning algorithm on the subset \overline{D}_j .
- Let $z_{it} = h_t^{(j)}(\mathbf{x}_i)$, then, for each sample \mathbf{x}_i in D_j , the output from the first-level learner can be written as $\mathbf{z}_i = (z_{i1}; z_{i2}; \dots; z_{iT})$, which is used as the input for the second-level learner with the original label y_i .
- By finishing the entire cross-validation process on the T first-level learners, we obtain a data set $D' = \{(\mathbf{z}_i, y_i)\}_{i=1}^m$ for training the second-level learner.

Summary

In this chapter we have learned

- about the principals of ensemble learning.
- how boosting and especially the AdaBoost algorithm works.
- how to employ bagging and RF.
- combination methods like averaging and voting.

Introduction to Machine Learning: Theory and Application

Chapter 5

Alexander Szimayer

Universität Hamburg

Summer Term 2024

5. Support Vector Machine

In this chapter we

- classify a data set by finding separating **hyperplanes**.
- formulate the optimization problem to find the best separating hyperplanes, resulting in the **primal form of Support Vector Machines**.
- simplify this optimization problem by transforming it into a **dual problem**.
- discuss how to deal with samples that are not linearly separable using the **Kernel Trick** or a **Soft Margin**.

5.1 Margin and Support Vector

- Suppose a training set $D = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_m, y_m)\}$ is given. Where each sample $\mathbf{x}_i = (x_{i1}, x_{i2}, \dots, x_{id}) \in \mathcal{X}$ is a vector in the d -dimensional sample space and $y_i \in \{-1, +1\}$ is the associated label.
- The **basic idea** of classification is to use the training set D to find a **hyperplane** in the sample space that **separates** samples of different classes.
- The goal is to find the hyperplane with the **strongest generalization ability** and the **most robust classification results**.

Hyperplanes Example

- The separating hyperplane with the best tolerance to local data perturbation is the one right in the middle of two classes.
- Often samples in the test set are closer to the decision boundary due to the noises of the training set.

Therefore, many hyperplanes that perform well on the training set are likely to make mistakes.

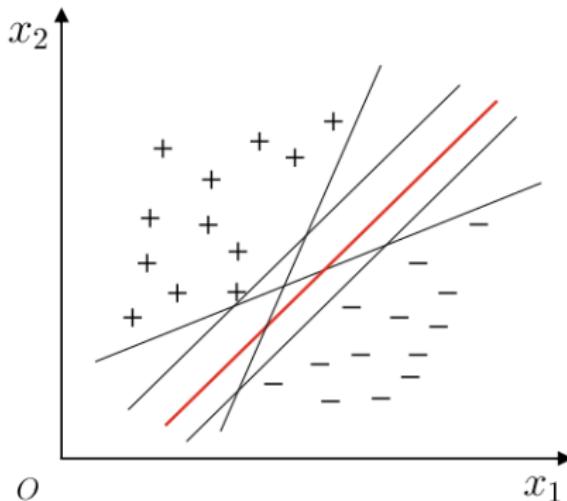


Figure 5.1: More than one hyperplanes can separate the training samples. The red line is the hyperplane which is right in the middle of the two classes.

Support Vectors and Margin (1/3)

- The following linear function describe a **separating hyperplane** in the sample space,

$$\mathbf{w}^\top \mathbf{x} + b = 0,$$

where $\mathbf{w} = (w_1; w_2; \dots; w_d)$ is the **normal vector** which controls the direction of the hyperplane, and b is the **bias** which controls the distance between the hyperplane and the origin.

- The normal vector \mathbf{w} and the bias b determine the separating hyperplane, denoted by (\mathbf{w}, b) .
- The distance r from any point \mathbf{x} in the sample space to the hyperplane (\mathbf{w}, b) can be written as:

$$r = \frac{|\mathbf{w}^\top \mathbf{x} + b|}{\|\mathbf{w}\|}.$$

Support Vectors and Margin (2/3)

- Suppose the hyperplane (\mathbf{w}, b) can correctly classify the training samples. That is for $(x_i, y_i) \in D$, $\mathbf{w}^\top \mathbf{x}_i + b > 0$ when $y_i = +1$, and $\mathbf{w}^\top \mathbf{x}_i + b < 0$ when $y_i = -1$. Let

$$\begin{cases} \mathbf{w}^\top \mathbf{x}_i + b \geq +1, & y_i = +1, \\ \mathbf{w}^\top \mathbf{x}_i + b \leq -1, & y_i = -1. \end{cases} \quad (5.1)$$

where the equality in (5.1) holds for the sample points closest to the hyperplane. These sample points are called **support vectors**.

- The total distance from two support vectors of different classes to the hyperplane is called **margin**, and can be computed by:

$$\gamma = \frac{2}{\|\mathbf{w}\|}.$$

Support Vectors and Margin (3/3)

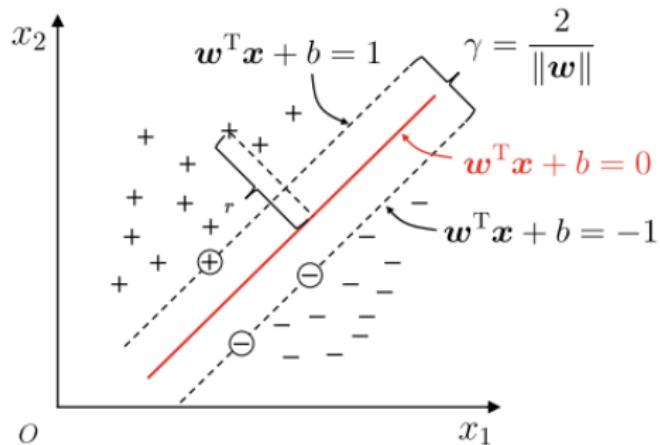


Figure 5.2: Support vectors and margin

Support Vectors Machine

- Finding the separating hyperplane with the **maximum margin** is equivalent to finding the parameters w and b that maximize γ subject to the constraints in equality (5.1), that is:

$$\max_{w,b} \frac{2}{\|w\|}, \quad (5.2)$$

$$s.t. \quad y_i(w^\top x_i + b) \geq 1, \quad i = 1, 2, \dots, m.$$

- The margin can be optimized by maximizing $\|w\|^{-1}$, which is equivalent to minimizing $\|w\|^2$. Therefore, we can rewrite (5.2) as:

$$\min_{w,b} \frac{1}{2} \|w\|^2, \quad (5.3)$$

$$s.t. \quad y_i(w^\top x_i + b) \geq 1, \quad i = 1, 2, \dots, m.$$

This is the **primal form of Support Vector Machine (SVM)**.

5.2 Dual Problem

- To obtain the **maximum margin separating hyperplane** model $f(x = \mathbf{w}^\top \mathbf{x} + b)$, where \mathbf{w} and b are the model parameters, we need to solve the primal form of SVM (5.3).
- There are several existing optimization packages to solve the convex quadratic programming problem (5.3).
- By applying the Lagrange multipliers to the problem, it will turn into a **dual problem**.

Lagrange Multiplier

- By introducing a Lagrange multiplier $\alpha_i \geq 0$ to each constraint in (5.3), we get the **Lagrange function**,

$$L(\mathbf{w}, b, \boldsymbol{\alpha}) = \frac{1}{2} \|\mathbf{w}\|^2 + \sum_{i=1}^m \alpha_i (1 - y_i (\mathbf{w}^\top \mathbf{x}_i + b)), \quad (5.4)$$

where $\boldsymbol{\alpha} = (\alpha_1; \alpha_2; \dots; \alpha_m)$.

- Setting the partial derivations of $L(\mathbf{w}, b, \boldsymbol{\alpha})$ with respect to \mathbf{w} and b to 0, gives:

$$\mathbf{w} = \sum_{i=1}^m \alpha_i y_i \mathbf{x}_i, \quad (5.5)$$

$$0 = \sum_{i=1}^m \alpha_i y_i. \quad (5.6)$$

The Dual Problem

- Substituting equation (5.5) into equation (5.4) eliminates \mathbf{w} from $L(\mathbf{w}, b, \alpha)$.
- With the constraint (5.6) the dual problem of (5.3) turns into:

$$\begin{aligned} & \max_{\alpha} \sum_{i=1}^m \alpha_i - \frac{1}{2} \sum_{i=1}^m \sum_{j=1}^m \alpha_i \alpha_j x_i y_i x_j^\top x_j \\ & \text{s.t. } \sum_{i=1}^m \alpha_i y_i = 0, \\ & \quad \alpha_i \geq 0, \quad i = 1, 2, \dots, m. \end{aligned} \tag{5.7}$$

- Solving this optimization problem, realizes α , and subsequently \mathbf{w} using (5.5) and b using the Karush-Kuhn-Tucker (KKT) conditions. This results in the desired model for the separating hyperplane:

$$\begin{aligned} f(\mathbf{x}) &= \mathbf{w}^\top \mathbf{x} + b \\ &= \sum_{i=1}^m \alpha_i y_i \mathbf{x}_i^\top \mathbf{x} + b. \end{aligned} \tag{5.8}$$

The Karush-Kuhn-Tucker Conditions

- The optimization problem with inequality constraints (5.3) must satisfy the KKT conditions:

$$\begin{cases} \alpha_i \geq 0; \\ y_i f(\mathbf{x}_i) - 1 \geq 0; \\ \alpha_i(y_i f(\mathbf{x}_i) - 1) = 0. \end{cases}$$

- Therefore, for any training sample (\mathbf{x}_i, y_i) holds either $\alpha_i = 0$ or $y_i f(\mathbf{x}_i) = 1$.
- **Case 1: ($\alpha_i = 0$)** The sample is not included in the summation in (5.8). Therefore, it has no impact on $f(\mathbf{x})$.
- **Case 2: ($\alpha_i > 0$)** This leads to $y_i f(\mathbf{x}_i) = 1$. The sample point is a support vector and lies on the maximum-margin hyperplanes.
- Therefore, an important property of SVMs is, that once training is completed, the final model only depends on the support vectors.

The Sequential Minimal Optimization (SMO)

- To solve the quadratic programming problem (5.7), quadratic programming algorithms can be used.
- However, the computational costs are often very high, since the complexity is seriously affected by the number of training samples.
- There are many efficient algorithms that exploit the the structure of the optimization problem, to overcome this limitation.
- One of them is the **Sequential Minimal Optimization**, where The basic idea is to iteratively find the local optimal solutions of α_i by fixing all the other parameters as constants.

5.3 Kernel Function

- So far, it was assumed that the training samples are linearly separable, which means, there are hyperplanes that can classify all training samples correctly.
- This assumption often does not hold in practice. One example is the exclusive disjunction problem shown in Figure 5.3.
- In such cases, it is possible to make the samples linearly separable by mapping the samples from the original feature space to a higher dimensional feature space.
- If the original feature space has a finite number of features, there must be a higher dimensional feature space in which the samples are linearly separable.

Example

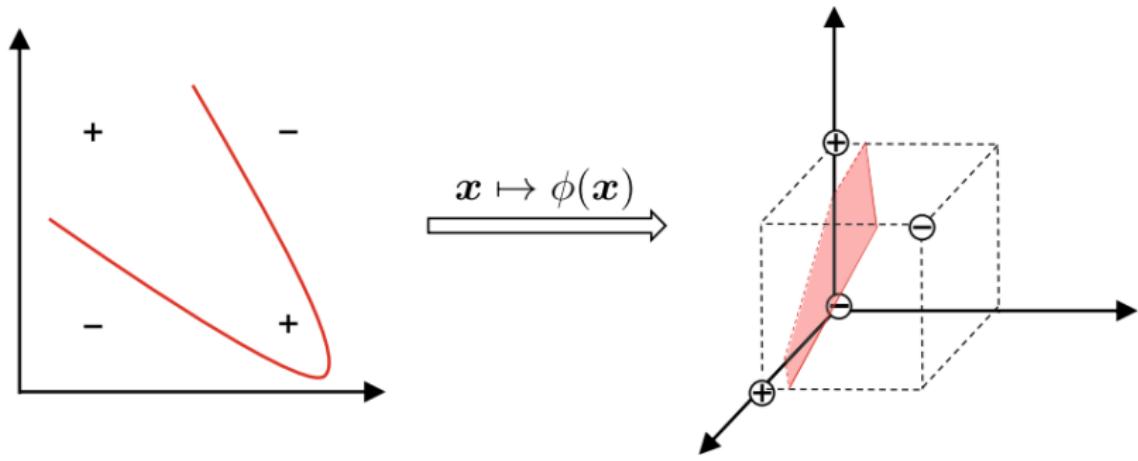


Figure 5.3: The XOR problem and non-linear mapping with the mapped feature vector $\phi(x)$. A qualified hyperplane can be found after mapping the 2-dimensional space to a 3-dimensional space.

Model

- Let $\phi(\mathbf{x})$ denote the mapped feature vector of \mathbf{x} . Then, the **separating hyperplane model** in the feature space is:

$$f(\mathbf{x}) = \mathbf{w}^\top \phi(\mathbf{x}) + b,$$

where \mathbf{w} and b are the model parameters.

- Similar to the problem 5.3, this leads to:

$$\min_{\mathbf{w}, b} \frac{1}{2} \|\mathbf{w}\|^2$$

$$s.t. \quad y_i(\mathbf{w}^\top \phi(\mathbf{x}_i) + b) \geq 1, \quad i = 1, 2, \dots, m.$$

Its dual problem is:

$$\begin{aligned} & \max_{\alpha} \sum_{i=1}^m \alpha_i - \frac{1}{2} \sum_{i=1}^m \sum_{j=1}^m \alpha_i \alpha_j y_i y_j \phi(\mathbf{x}_i)^\top \phi(\mathbf{x}_j) \\ & s.t. \quad \sum_{i=1}^m \alpha_i y_i = 0, \quad \alpha_i \geq 0, \quad i = 1, 2, \dots, m. \end{aligned} \tag{5.9}$$

The Kernel Trick (1/2)

- Solving the problem (5.9) involves the calculation of $\phi(\mathbf{x}_i)^\top \phi(\mathbf{x}_j)$, which is the inner product of the mapped feature vectors of \mathbf{x}_i and \mathbf{x}_j .
- It is often difficult to compute $\phi(\mathbf{x}_i)^\top \phi(\mathbf{x}_j)$, because of the very high or even infinite dimensionality the mapped feature space can have.
- To avoid this difficulty, we suppose there exists a function following:

$$\kappa(\mathbf{x}_i, \mathbf{x}_j) = (\phi(\mathbf{x}_i), \phi(\mathbf{x}_j)) = \phi(\mathbf{x}_i)^\top \phi(\mathbf{x}_j). \quad (5.10)$$

Which means, that the inner product of \mathbf{x}_i and \mathbf{x}_j in the feature space can be calculated using the function $\kappa(\cdot, \cdot)$.

- Therefore, it is possible to rewrite (5.9) as:

$$\begin{aligned} & \max_{\alpha} \sum_{i=1}^m \alpha_i - \frac{1}{2} \sum_{i=1}^m \sum_{j=1}^m \alpha_i \alpha_j y_i y_j \kappa(\mathbf{x}_i, \mathbf{x}_j) \\ & s.t. \quad \sum_{i=1}^m \alpha_i y_i = 0, \quad \alpha_i \geq 0, \quad i = 1, 2, \dots, m. \end{aligned} \quad (5.11)$$

The Kernel Trick (2/2)

- Solving (5.11) gives the following equation,

$$\begin{aligned} f(\mathbf{x}) &= \mathbf{w}^\top \phi(\mathbf{x}) + b = \sum_{i=1}^m \alpha_i y_i \phi(\mathbf{x}_i)^\top \phi(\mathbf{x}) + b \\ &= \sum_{i=1}^m \alpha_i y_i \kappa(\mathbf{x}, \mathbf{x}_i) + b, \end{aligned} \tag{5.12}$$

where $\kappa(\cdot, \cdot)$ is the **kernel function**.

- Equation (5.12) shows that the optimal solution can be expanded by training samples with the kernel functions, which is known as the **support vector expansion**.
- The kernel function $\kappa(\cdot, \cdot)$ can be derived if the details of the mapping $\phi(\cdot)$ are known.
- However, $\phi(\cdot)$ is often unknown in practice.

The Kernel Function (1/2)

Theorem 5.1 - Kernel Function: Let \mathcal{X} denote the input space and $\kappa(\cdot, \cdot)$ denote a symmetric function defined in $\mathcal{X} \times \mathcal{X}$. Then, κ is a kernel function if and only if the kernel matrix \mathbf{K} is positive semidefinite for any data set $D = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_m\}$:

$$\mathbf{K} = \begin{bmatrix} \kappa(\mathbf{x}_1, \mathbf{x}_1) & \dots & \kappa(\mathbf{x}_1, \mathbf{x}_j) & \dots & \kappa(\mathbf{x}_1, \mathbf{x}_m) \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ \kappa(\mathbf{x}_i, \mathbf{x}_1) & \dots & \kappa(\mathbf{x}_i, \mathbf{x}_j) & \dots & \kappa(\mathbf{x}_i, \mathbf{x}_m) \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ \kappa(\mathbf{x}_m, \mathbf{x}_1) & \dots & \kappa(\mathbf{x}_m, \mathbf{x}_j) & \dots & \kappa(\mathbf{x}_m, \mathbf{x}_m) \end{bmatrix}.$$

The Kernel Function (2/2)

- Theorem 5.1 is helpful to find out what kind of functions are valid kernel functions.
- It states, that every symmetric function with a positive semidefinite kernel matrix is a valid kernel function, and that there is always a corresponding mapping ϕ for each matrix.
- This means, that every kernel function implicitly defines a feature space known as the **Reproducing Kernel Hilbert Space (RKHS)**.

Common Kernel Functions

Name	Expression	Parameters
Linear kernel	$\kappa(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i^\top \mathbf{x}_j$	
Polynomial kernel	$\kappa(\mathbf{x}_i, \mathbf{x}_j) = (\mathbf{x}_i^\top \mathbf{x}_j)^d$	$d \geq 1$ is the degree of the polynomial
Gaussian kernel	$\kappa(\mathbf{x}_i, \mathbf{x}_j) = \exp\left(-\frac{\ \mathbf{x}_i - \mathbf{x}_j\ ^2}{2\sigma^2}\right)$	$\sigma > 0$ is the width of the Gaussian kernel
Laplacian kernel	$\kappa(\mathbf{x}_i, \mathbf{x}_j) = \exp\left(-\frac{\ \mathbf{x}_i - \mathbf{x}_j\ }{\sigma}\right)$	$\sigma > 0$
Sigmoid kernel	$\kappa(\mathbf{x}_i, \mathbf{x}_j) = \tanh(\beta \mathbf{x}_i^\top \mathbf{x}_j + \theta)$	\tanh is the hyperbolic tangent function, $\beta > 0$, $\theta < 0$

Table 5.1: Some common kernel functions.

Remarks

- The quality of the feature space is essential to the performance of SVMs, since the samples should be linearly separable.
- However, the feature mapping as well as the kernel functions are ‘good’ isn’t known.
- Therefore, the ‘choice of kernel’ is the biggest uncertainty of SVMs.
- A poor kernel will map the samples to a poor feature space, resulting in poor performance.

5.4 Soft Margin

- So far, it was assumed that the samples are linearly separable in either the sample space or the feature space.
- However, often it is difficult to find an appropriate kernel function to make the training samples linearly separable in the feature space.
- Even if such an appropriate kernel function is known, it is hard to tell if it is a result of overfitting.
- To alleviate this problem one can allow the SVM to make mistakes on a few samples. This idea is implemented by the concept of **soft margin**, as shown in Figure 5.4.

Example For a Soft Margin

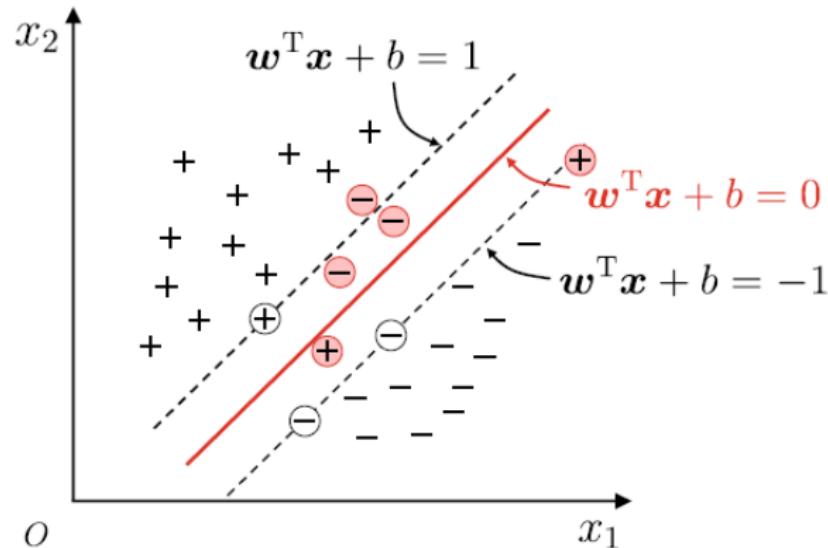


Figure 5.4: Soft margin. The samples in red violate the constraints

Definition

- The SVMs were introduced in the previous sections as subject to the constraints (5.1), that is, the **hard margins** requires all samples to be correctly classified.
- The **soft margin** allows a violation on the constraint:

$$y_i(\mathbf{w}^\top \mathbf{x}_i + b) \geq 1. \quad (5.13)$$

- Still, the number of samples violating the constraint should be minimized while maximizing the margin.

Soft Margin Optimization

- The optimization problem can be written as following,

$$\min_{w,b} \frac{1}{2} \|w\|^2 + C \sum_{i=1}^m \ell_{0/1}(y_i(w^\top x_i + b) - 1), \quad (5.14)$$

where $C > 0$ is a constant, and $\ell_{0/1}$ is the 0/1 loss function,

$$\ell_{0/1}(z) = \begin{cases} 1, & \text{if } z < 0; \\ 0, & \text{otherwise.} \end{cases} \quad (5.15)$$

- If C is **infinitely large**, (5.14) forces all samples to obey the constraint (5.13). Then (5.14) is equivalent to equation (5.3), which is the **hard margin**.
- If C takes a **finite value** some samples may violate the constraint (5.13).

Surrogate Loss Functions

- It is difficult to solve (5.14) directly, since $\ell_{0/1}$ has poor mathematical properties (non-convex and discontinuous).
- It is possible to replace $\ell_{0/1}$ with **surrogate loss functions**, which have nice mathematical properties and are upper bound of $\ell_{0/1}$.
- There are three commonly used surrogate functions:
 - ① **Hinge loss:** $\ell_{\text{hinge}}(z) = \max(0, 1 - z)$,
 - ② **Exponential loss:** $\ell_{\text{exp}}(z) = \exp(-z)$,
 - ③ **Logistic loss:** $\ell_{\text{log}}(z) = \log(1 + \exp(-z))$.

Surrogate Loss Functions

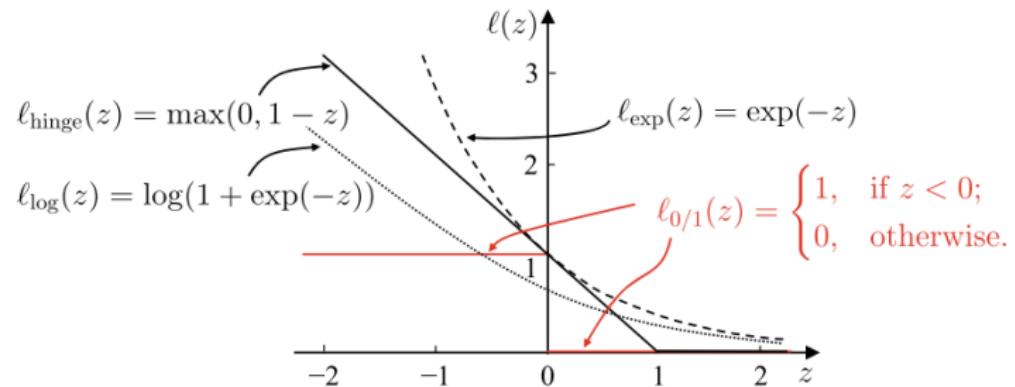


Figure 5.5: Three surrogate losses, namely, hinge loss, exponential loss, and logistic loss

The Commonly Used Soft Margin

- If **hinge loss** is used as the surrogate loss function, (5.14) becomes:

$$\min_{w,b} \frac{1}{2} \|w\|^2 + C \sum_{i=1}^m \max(0, 1 - y_i(w^\top x_i + b)).$$

- By introducing **slack variables** $\xi_i \geq 0$, the above can be rewritten as,

$$\begin{aligned} & \min_{w,b} \frac{1}{2} \|w\|^2 + C \sum_{i=1}^m \xi_i \\ & \text{s.t. } y_i(w^\top x_i + b) \geq 1 - \xi_i \quad \xi_i \geq 0, \quad i = 1, 2, \dots, m, \end{aligned} \quad (5.16)$$

which is the **commonly used Soft Margin Support Vector Machine**.

- Each sample in (5.16) has a corresponding slack variable which indicates the degree to which constraint (5.13) is violated.

Dual Problem for Soft Margin

- Again (5.16) is a quadratic programming problem. Hence, Lagrange multipliers are applied to obtain the dual problem of (5.16). After some calculation we obtain,

$$\max_{\alpha} \sum_{i=1}^m \alpha_i - \frac{1}{2} \sum_{i=1}^m \sum_{j=1}^m \alpha_i \alpha_j y_i y_j \mathbf{x}_i^\top \mathbf{x}_j \quad (5.17)$$

$$\text{s.t. } \sum_{i=1}^m \alpha_i y_i = 0, 0 \leq \alpha_i \leq C, \quad i = 1, 2, \dots, m,$$

where $C = \alpha_i + \mu_i$ for the Lagrange multipliers $\alpha_i \geq 0$ and $\mu_i \geq 0$.

- The only difference between the dual problem of soft margin (5.17) and of hard margin (5.7) is the constraint on dual variables:
 $0 \leq \alpha_i \leq C$ for soft margin, and $\alpha_i \geq 0$ for hard margin.
- Therefore, (5.17) can be solved the same way as in Section 5.2 and by introducing kernel function to obtain the support vector expansion as in (5.12).

KKT Conditions for Soft Margin SVM (1/2)

- The Karush-Kuhn-Tucker conditions for soft margin SVMs are the following:

$$\begin{cases} \alpha_i \geq 0, \quad \mu_i \geq 0, \\ y_i f(\mathbf{x}_i) - 1 + \xi_i \geq 0, \\ \alpha_i(y_i f(\mathbf{x}_i) - 1 + \xi_i) = 0, \\ \xi_i \geq 0, \quad \mu_i \xi_i = 0. \end{cases}$$

- Therefore, for any training sample (\mathbf{x}_i, y_i) holds either $\alpha_i = 0$ or $y_i f(\mathbf{x}_i) = 1 - \xi_i$.
- Case 1:** $\alpha_i = 0$. The sample has no impact on $f(\mathbf{x})$.
- Case 2:** $\alpha_i > 0$. Then $y_i f(\mathbf{x}_i) = 1 - \xi_i$ and this sample is a support vector.

KKT Conditions for Soft Margin SVM (2/2)

- If $\alpha_i < C$, then $\mu_i > 0$, and subsequently $\xi_i = 0$. This sample point then lies on the maximum-margin hyperplanes.
- If $\alpha_i = C$, then $\mu_i = 0$. This means the sample falls inside the margin if $\xi_i \leq 1$, and is incorrectly classified if $\xi_i > 1$.
- Therefore, the final model of soft margin SVM only depends on the support vectors, that is, the sparseness is preserved after using the hinge loss.

SVM vs. Logistic Regression

- Replacing the 0/1 loss function of (5.14) with the **logistic loss function** ℓ_{log} , gives a model that is almost identical to the logistic regression.
- The performance of logistic regression is similar to the SVM in many cases, since their optimization objectives are similar.
- The **main advantage of logistic regression** is that the output naturally carries **probability meanings**. Whereas the predictions of SVMs do not associate with probabilities.
- The logistic regression can be directly applied to multiclass classifications, while the SVM requires extensions.
- Since there is a **flat zero region** for hinge loss, the solution of SVMs is sparse. (See Figure 5.5.)
- The logistic loss function, which is a smooth and monotonically decreasing function, relies on more training samples and its prediction cost is higher, since it cannot derive the concept like support vectors.

5.5 Regularization

- Replacing the 0/1 loss function of (5.14) with other surrogate functions gives further learning models.
- These obtained models, which depend on the choice of the surrogate functions, have the following in common:
 - ① **The first term** in the objective function represents the '**margin**' size of the separating hyperplane.
 - ② **The other term** $\sum_{i=1}^m \ell(f(\mathbf{x}_i), y_i)$ represents the **error** on the training set.

Risk Minimization

- The loss can be rewritten in a more general form:

$$\min_f \Omega(f) + C \sum_{i=1}^m \ell(f(\mathbf{x}_i), y_i). \quad (5.18)$$

- The first term $\Omega(f)$ is known as the **structural risk** and represents some properties of the model f .
- The second term $\sum_{i=1}^m \ell(f(\mathbf{x}_i), y_i)$ is the **empirical risk**, which describes how well the model matches the training data.
- The constant C makes a **trade-off** between these two risks.

Regularization Problem

- From the perspective of minimizing the empirical risk, $\Omega(f)$ represents what kind of properties the model should have, which provides a way for incorporating domain knowledge and user's requirements.
- The structural risk is also helpful for reducing the hypothesis space, which reduces the overfitting risk of minimizing the training errors.
- With this, (5.18) is a **regularization problem**, where $\Omega(f)$ is the **regularization term**, and C is the **regularization constant**.

Remarks

- Regularization can be considered as a **penalty function method** which applies penalties to undesired outcomes such that the optimization is biased towards the desired outcome.
- From the Bayesian perspective, the regularization term can be seen as a **prior** to the model.
- A typical regularization term is the L_p norm, where the L_2 norm $\|\mathbf{w}\|_2$ is biased towards balanced weights \mathbf{w} .
The L_0 norm $\|\mathbf{w}\|_0$ and the L_1 norm $\|\mathbf{w}\|_1$ are biased towards making \mathbf{w} have sparse elements, that is, with only a few non-zero elements.

Summary

In this chapter we have learned

- how to classify a data set by finding separating **hyperplanes**.
- what support vectors and margins are and introduced the **primal form of Support Vector Machines**.
- how to transform the optimization problem into a **dual problem**.
- about the **Kernel Trick** and the **Soft Margin**.

Introduction to Machine Learning: Theory and Application

Chapter 6

Alexander Szimayer

Universität Hamburg

Summer Term 2024

6. Bayes Classifiers

In this chapter we

- discuss the concepts of **Bayesian Decision Theory**.
- introduce the theoretically optimal **Bayes Classifier**.
- define the **Naïve Bayes Classifier** that shortens the estimation process.
- learn about the **Semi-Naïve Bayes Classifier** which makes further assumptions to simplify the computation.

6.1 Bayesian Decision Theory

- Bayesian decision theory is a fundamental decision-making approach under the **probability framework**.
- In an **ideal situation** when all relevant probabilities were known, Bayesian decision theory makes **optimal classification decisions** based on the probabilities and costs of misclassifications.
- In the following, we demonstrate the basic idea of Bayesian decision theory with multiclass classification.

Bayesian Theory (1/3)

- Suppose, N distinct class labels, that is, $\mathcal{Y} = \{c_1, c_2, \dots, c_N\}$, where λ_{ij} denote the cost of misclassifying a sample of class c_j as class c_i .
- We can compute the **expected loss** of classifying a sample \mathbf{x} as class c_i , using the posterior probability $P(c_i|\mathbf{x})$, which is the **conditional risk** of the sample \mathbf{x} :

$$R(c_i|\mathbf{x}) = \sum_{j=1}^N \lambda_{ij} P(c_j|\mathbf{x}).$$

- Our objective is to find a decision rule $h : \mathcal{X} \mapsto \mathcal{Y}$ that minimizes the **overall risk**:

$$R(h) = \mathbb{E}_{\mathbf{x}}[R(h(\mathbf{x})|\mathbf{x})].$$

Bayesian Theory (2/3)

- We can minimize the overall risk $R(h)$ by minimizing the conditional risk $R(h(\mathbf{x})|\mathbf{x})$ of each sample \mathbf{x} .
- This leads to the **Bayes decision rule**: To minimize the overall risk, classify each sample as the class that minimizes the conditional risk $R(c|\mathbf{x})$,

$$h^*(\mathbf{x}) = \arg \min_{c \in \mathcal{Y}} R(c|\mathbf{x}),$$

where h^* is the **Bayes optimal classifier** and its overall risk $R(h^*)$ is the **Bayes risk**.

- The **best performance** that can be achieved by any classifier, is, $1 - R(h^*)$. This is the theoretically achievable upper bound of accuracy for any machine learning model.

Bayesian Theory (3/3)

- As a specific example consider the objective to **minimize the misclassification rate**. Then, we write the misclassification loss λ_{ij} as,

$$\lambda_{ij} = \begin{cases} 0, & \text{if } i = j; \\ 1, & \text{otherwise,} \end{cases}$$

while the conditional risk is,

$$R(c|\mathbf{x}) = 1 - P(c|\mathbf{x}).$$

- Then, the Bayes optimal classifier that **minimizes the misclassification rate** is:

$$h^*(\mathbf{x}) = \arg \max_{c \in \mathcal{Y}} P(c|\mathbf{x}). \quad (6.1)$$

which classifies each sample \mathbf{x} as the class that maximizes its posterior probability $P(c|\mathbf{x})$.

Posterior Probability

- Equation (6.1) shows that the Bayes decision rule relies on the **posterior probability** $P(c|x)$, which is often difficult to obtain.
- There are two strategies to accurately estimate the posterior probability from the finite training samples using machine learning:
 - ① **Discriminant models:** Predict c by estimating the posterior probability $P(c|x)$ directly,
 - ② **Generative models:** Estimate the joint probability $P(x, c)$ and then estimate the posterior probability $P(c|x)$.
- So far, we have only worked with discriminative models like decision trees and support vector machines.

Generative Models (1/2)

- We estimate the posterior probability by applying the Bayes' Theorem,

$$P(c|x) = \frac{P(c)P(x|c)}{P(x)}. \quad (6.2)$$

where $P(c)$ is the **prior** probability of c , $P(x|c)$ is the **class-conditional probability**, also known as the likelihood, of the sample x with respect to the class c , and $P(x)$ is the **evidence factor** for normalization.

- Given x , the evidence factor $P(x)$ is independent of the class, which transforms the estimation of $P(c|x)$ into estimating the prior $P(c)$ and the likelihood $P(x|c)$ from the training set D .

Generative Models (2/2)

- Given sufficient *i.i.d.* samples, the proportion of each class in the sample space $P(c)$, known as **prior probability**, can be estimated by the **frequency** of each class in the training set, by the law of large numbers.
- Computing the class-conditional probability $P(\mathbf{x}|c)$ is difficult since the joint probabilities of all features of \mathbf{x} are needed to compute $P(\mathbf{x}|c)$.
- It is infeasible to estimate $P(\mathbf{x}|c)$ directly by the frequencies in the training set, since ‘unobserved’ and ‘zero probability’ are generally different.
- The class-conditional probability can be estimated with the **Maximum Likelihood Estimation**.

6.2 Maximum Likelihood Estimation

- The class-conditional probability $P(\mathbf{x}|c)$ can be estimated by hypothesizing a fixed form of probability distribution and estimating the distribution parameters using the training set.
- $P(\mathbf{x}|c)$ denotes the class-conditional probability of class c and has a fixed form determined by a parameter vector θ_c . Therefore, we write $P(\mathbf{x}|c)$ as $P(\mathbf{x}|\theta_c)$.
- Our objective is to estimate θ_c from the training set D .

MLE (1/2)

- D_c denotes the set of class c samples in the training set D , where the samples are independent and identically distributed.
- The likelihood of D_c for a given parameter θ_c is:

$$P(D_c|\theta_c) = \prod_{x \in D_c} P(x|\theta_c). \quad (6.3)$$

- Using the MLE, we want to find a parameter $\hat{\theta}_c$ that **maximizes the likelihood** $P(D_c|\theta_c)$.
- The product of the sequence in (6.3) often leads to **underflow**. Therefore, we use the **log-likelihood** instead,

$$\begin{aligned} LL(\theta_c) &= \log P(D_c|\theta_c) \\ &= \sum_{x \in D_c} \log P(x|\theta_c), \end{aligned}$$

and the MLE of θ_c is,

$$\hat{\theta}_c = \arg \max_{\theta_c} LL(\theta_c).$$

MLE (2/2)

- Suppose the features are continuous and the probability density function follows the Gaussian distribution $p(\mathbf{x}|c) \sim \mathcal{N}(\mu_c, \sigma_c^2)$. Then the MLE of the parameters μ_c and σ_c^2 are,

$$\hat{\mu}_c = \frac{1}{|D_c|} \sum_{\mathbf{x} \in D_c} \mathbf{x},$$

$$\hat{\sigma}_c^2 = \frac{1}{|D_c|} \sum_{\mathbf{x} \in D_c} (\mathbf{x} - \hat{\mu}_c)(\mathbf{x} - \hat{\mu}_c)^T,$$

which shows, that the estimated mean of Gaussian distribution obtained by the MLE is the sample mean, and the estimated variance is the mean of $(\mathbf{x} - \hat{\mu}_c)(\mathbf{x} - \hat{\mu}_c)^T$.

- Conditional probabilities can be estimated similarly for discrete features.

Discussion

- This method simplifies the estimation of posterior probabilities.
- However, its **accuracy** heavily relies on whether the hypothetical probability distribution matches the unknown ground-truth data distribution.
- In practice, to guess a probability distribution can lead to **misleading results**, which means that often domain knowledge is needed to hypothesize a good approximation to the ground-truth data distribution.

6.3 Naïve Bayes Classifier

- Calculating the class-conditional probability $P(x|c)$ from the finite training sample is not easy since $P(x|c)$ is the joint probability on all attributes.
- Therefore, it is difficult to estimate the posterior probability $P(c|x)$ with the Bayes rule (6.2).
- The **Naïve Bayes classifier** avoids this problem by making the **attribute conditional independence assumption**:
Given any known class, assume all attributes are independent of each other.
- In other words, we assume each attribute influences the prediction result independently.

The Naïve Bayes Classifier

- Given the attribute conditional independence assumption, we can rewrite (6.2) as,

$$P(c|\mathbf{x}) = \frac{P(c)P(\mathbf{x}|c)}{P(\mathbf{x})} = \frac{P(c)}{P(\mathbf{x})} \prod_{i=1}^d P(x_i|c),$$

where d is the number of attributes and x_i is the value taken on the i th attribute of \mathbf{x} .

- Since $P(\mathbf{x})$ the same for all classes, we can rewrite the Bayes decision rule (6.1) as,

$$h_{nb}(\mathbf{x}) = \arg \max_{c \in \mathcal{Y}} P(c) \prod_{i=1}^d P(x_i|c), \quad (6.4)$$

which is the formulation of the **Naïve Bayes classifier**.

Training the Naïve Bayes Classifier (1/2)

- We train the Naïve Bayes classifier by first computing the **prior probability** $P(c)$ from the training set D , and then computing the **conditional probability** $P(x_i|c)$ for each attribute.
- Let D_c denote a subset of D containing all samples of class c . Given sufficient *i.i.d.* samples, we can estimate the **prior probability** by:

$$P(c) = \frac{|D_c|}{|D|}. \quad (6.5)$$

Training the Naïve Bayes Classifier (2/2)

- **Discrete attributes:** D_{c,x_i} denotes a subset of D_c containing all samples taking the value x_i on the i th attribute. Then we estimate the conditional probability $P(x_i|c)$ by:

$$P(x_i|c) = \frac{|D_{c,x_i}|}{|D_c|}. \quad (6.6)$$

- **Continuous attributes:** Suppose $p(x_i|c) \sim \mathcal{N}(\mu_{c,i}, \sigma_{c,i}^2)$, where $\mu_{c,i}$ and $\sigma_{c,i}^2$ are the mean and variance of the i th attribute of class c samples. Then, we have:

$$p(x_i|c) = \frac{1}{\sqrt{2\pi}\sigma_{c,i}} \exp\left(-\frac{(x_i - \mu_{c,i})^2}{2\sigma_{c,i}^2}\right). \quad (6.7)$$

Example (1/4)

- We want to train a Naïve Bayes classifier using the watermelon data set 3.0:

ID	color	root	sound	texture	umbilicus	surface	density	sugar	ripe
1	green	curly	muffled	clear	hollow	hard	0.697	0.460	true
2	dark	curly	dull	clear	hollow	hard	0.774	0.376	true
3	dark	curly	muffled	clear	hollow	hard	0.634	0.264	true
4	green	curly	dull	clear	hollow	hard	0.608	0.318	true
5	light	curly	muffled	clear	hollow	hard	0.556	0.215	true
6	green	slightly curly	muffled	clear	slightly hollow	soft	0.403	0.237	true
7	dark	slightly curly	muffled	slightly blurry	slightly hollow	soft	0.481	0.149	true
8	dark	slightly curly	muffled	clear	slightly hollow	hard	0.437	0.211	true
9	dark	slightly curly	dull	slightly blurry	slightly hollow	hard	0.666	0.091	false
10	green	straight	crisp	clear	flat	soft	0.243	0.267	false
11	light	straight	crisp	blurry	flat	hard	0.245	0.057	false
12	light	curly	muffled	blurry	flat	soft	0.343	0.099	false
13	green	slightly curly	muffled	slightly blurry	hollow	hard	0.639	0.161	false
14	light	slightly curly	dull	slightly blurry	hollow	hard	0.657	0.198	false
15	dark	slightly curly	muffled	clear	slightly hollow	soft	0.360	0.370	false
16	light	curly	muffled	blurry	flat	hard	0.593	0.042	false
17	green	curly	dull	slightly blurry	slightly hollow	hard	0.719	0.103	false

Table 6.1: The watermelon data set 3.0

Example (2/4)

- Therefore, we aim to classify the following watermelon T1:

ID	color	root	sound	texture	umbilicus	surface	density	sugar	ripe
T1	green	curly	muffled	clear	hollow	hard	0.697	0.460	?

- Step 1:** Estimate the prior probability $P(c)$:

$$P(\text{ripe} = \text{true}) = \frac{8}{17} \approx 0.471,$$

$$P(\text{ripe} = \text{false}) = \frac{9}{17} \approx 0.529.$$

Example (3/4)

- **Step 2:** Estimate the conditional probability of each feature $P(x_i|c)$:

$$P_{green|true} = P(\text{color} = \text{green} | \text{ripe} = \text{true}) = \frac{3}{8} = 0.375,$$

$$P_{green|false} = P(\text{color} = \text{green} | \text{ripe} = \text{false}) = \frac{3}{9} \approx 0.333,$$

$$P_{curly|true} = P(\text{root} = \text{curly} | \text{ripe} = \text{true}) = \frac{5}{8} = 0.625,$$

$$P_{curly|false} = P(\text{root} = \text{curly} | \text{ripe} = \text{false}) = \frac{3}{9} \approx 0.333,$$

...

$$\begin{aligned} P_{density:0.697|true} &= P(\text{density} = 0.697 | \text{ripe} = \text{true}) \\ &= \frac{1}{\sqrt{2\pi} \cdot 0.129} \exp\left(-\frac{(0.697 - 0.574)^2}{2 \cdot 0.129^2}\right) \approx 1.959, \end{aligned}$$

...

Example (4/4)

- **Step 3:** Classifying the ripeness of the watermelon in the testing sample:

$$\begin{aligned} P(\text{ripe} = \text{true}) &\times P_{\text{green}|\text{true}} \times P_{\text{curly}|\text{true}} \times P_{\text{muffled}|\text{true}} \\ &\times P_{\text{clear}|\text{true}} \times P_{\text{hollow}|\text{true}} \times P_{\text{hard}|\text{true}} \\ &\times P_{\text{density}:0.697|\text{true}} \times P_{\text{sugar}:0.460|\text{true}} \approx 0.052, \end{aligned}$$

$$\begin{aligned} P(\text{ripe} = \text{false}) &\times P_{\text{green}|\text{false}} \times P_{\text{curly}|\text{false}} \times P_{\text{muffled}|\text{false}} \\ &\times P_{\text{clear}|\text{false}} \times P_{\text{hollow}|\text{false}} \times P_{\text{hard}|\text{false}} \\ &\times P_{\text{density}:0.697|\text{false}} \times P_{\text{sugar}:0.460|\text{false}} \approx 6.80 \times 10^{-5}. \end{aligned}$$

Since $0.052 > 6.80 \times 10^{-5}$, the Naïve Bayes classifier classifies the testing sample T1 as ripe.

The Laplacian Correction (1/2)

- If a feature value has never appeared together with a particular class, it's difficult to use the probability (6.6) to predict the class with equation (6.4).
- Suppose, given a testing sample with $\text{sound} = \text{crisp}$, the Naïve Bayes classifier trained on the watermelon data set 3.0 will predict:

$$P_{\text{crisp}|\text{true}} = P(\text{sound} = \text{crisp} | \text{ripe} = \text{true}) = \frac{0}{8} = 0.$$

Therefore, the watermelon will always be classified as unripe even if the watermelon is obviously ripe, since the product of the sequence in (6.4) gives a probability of zero.

- Removing the information carried by other features can be avoided by **smoothing** the probability estimation. A common choice is **the Laplacian correction**.

The Laplacian Correction (2/2)

- Suppose, N denotes the number of **distinct classes** in the training set D , and N_i denotes the number of **distinct values** the i th feature can take. Then (6.5) and (6.6) can be corrected as:

$$\hat{P}(c) = \frac{|D_c| + 1}{|D| + N}, \quad (6.8)$$

$$\hat{P}(x_i|c) = \frac{|D_{c,x_i}| + 1}{|D_c| + N_i}. \quad (6.9)$$

- Using the **Laplacian correction**, the problem of zero probabilities caused by insufficient training samples can be avoided.
- As the size of the training set increases the prior introduced by the correction will become negligible.

Example

- Using the Laplacian correction in our example, the prior probabilities and the conditional probabilities can be estimated as:

$$\hat{P}(\text{ripe} = \text{true}) = \frac{8 + 1}{17 + 2} \approx 0.474,$$

$$\hat{P}(\text{ripe} = \text{false}) = \frac{9 + 1}{17 + 2} \approx 0.526,$$

$$\hat{P}_{\text{green}|\text{true}} = \hat{P}(\text{color} = \text{green}|\text{ripe} = \text{true}) = \frac{3 + 1}{8 + 3} \approx 0.364,$$

$$\hat{P}_{\text{green}|\text{false}} = \hat{P}(\text{color} = \text{green}|\text{ripe} = \text{false}) = \frac{3 + 1}{9 + 3} \approx 0.333,$$

...

- Also, the probability $P_{\text{crisp}|\text{true}}$, which was zero, is estimated as:

$$\hat{P}_{\text{crisp}|\text{true}} = \hat{P}(\text{crisp} = \text{true}|\text{ripe} = \text{true}) = \frac{0 + 1}{8 + 3} \approx 0.091.$$

Naïve Bayes Classifier in Practice

- There are different ways to use a Naïve Bayes Classifier in practice.
- Examples:
 - ① If the **speed of prediction** is important, a Naïve Bayes classifier can **pre-calculate** all relevant probabilities and **save** them. Afterwards, the prediction can be made by looking up the saved probability table.
 - ② If the training set changes frequently, we can choose the **lazy approach**. Where we estimate the probabilities once a prediction request is received, that is, no training before prediction.
 - ③ If we keep receiving new training samples, we can enable **incremented learning** by updating only the probabilities that are related to the new samples.

6.4 Semi-Naïve Bayes Classifier

- The conditional independence assumption we use for the Naïve Bayes classifier often does not hold in practice.
- To relax this assumption, **Semi-Naïve Bayes classifiers** are developed.
- The idea is to consider some strong dependencies among features without computing the complete joint probabilities.
- A general strategy is **One Dependent Estimator (ODE)**.

One Dependent Estimator (ODE)

- The ODE states that each feature can depend on at most one feature other than the class information, that is,

$$P(c|\mathbf{x}) \propto P(c) \prod_{i=1}^d P(x_i|c, pa_i), \quad (6.10)$$

where x_i depends on pa_i , and pa_i is called the **parent feature** of x_i .

- If the parent feature pa_i for each feature x_i is known, we can estimate $P(x_i|c, pa_i)$, similar as equation (6.9).
- Now the **key problem** becomes determining the parent feature. There are different approaches which lead to different one-dependent classifiers.

Feature Dependencies

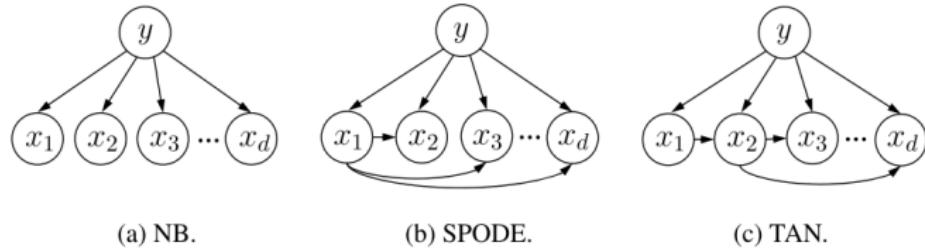


Figure 6.1: Feature dependencies of Naïve Bayes (NB) and Semi-Naïve Bayes classifiers, SPODE and TAN. In the following Super-Parent ODE (SPODE) and Tree Augmented Naïve Bayes (TAN) will be discussed in more detail.

Super-Parent ODE (SPODE)

- The **Super-Parent ODE** assumes that all features depend on just one feature called the **super parent**. (Figure 6.1b shows x_1 as a super-parent feature).
- The super parent feature can be selected by using **model selection methods** such as cross-validation.

Tree Augmented Naïve Bayes (TAN)

- The **Tree Augmented Naïve Bayes** is based on maximum weighted spanning trees and simplifies feature dependencies into a tree structure.
- This is done within four steps:
 - ① Compute the **conditional mutual** information for each pair of features:

$$I(x_i, x_j|y) = \sum_{x_i, x_j; c \in \mathcal{Y}} P(x_i, x_j|c) \log \frac{P(x_i, x_j|c)}{P(x_i|c)P(x_j|c)}.$$

- ② Construct a **complete undirected graph** in which the nodes are features. Set $I(x_i, x_j|y)$ as the weight of the edge between x_i and x_j .
 - ③ Construct a **maximum weighted spanning tree** and select a **root feature**. Set the direction of each edge outward from the root feature.
 - ④ Add a **class node** y and add direct edges from y to other feature nodes.
- Given the class information, $I(x_i, x_j, y)$ describes the correlation between x_i and x_j . Therefore, TAN keeps only the dependencies among highly correlated features.

Average One-Dependent Estimator (AODE) (1/2)

- AODE is a more **powerful** one-dependent classifier which takes advantage of ensemble learning.
- It uses each feature as a super-parent to build multiple SPODE models.
- The SPODE models are then integrated supported by sufficient training data, that is,

$$P(c|\mathbf{x}) \propto \sum_{\substack{i=1 \\ |D_{x_i}| \geq m'}} P(c, x_i) \prod_{j=1}^d P(x_j|c, x_i),$$

where D_{x_i} is the subset of samples taking the value x_i on the i th feature, and m' is a threshold constant.

Average One-Dependent Estimator (AODE) (2/2)

- We need to estimate $P(c, x_i)$ and $P(x_j|c, x_i)$.
- Similar to (6.9), we have,

$$\hat{P}(c, x_i) = \frac{|D_{c,x_i}| + 1}{|D| + N \times N_i},$$
$$\hat{P}(x_j|c, x_i) = \frac{|D_{c,x_i,x_j}| + 1}{|D_{c,x_i}| + N_j},$$

where N is the number of distinct classes in D , N_i is the number of distinct values the i th feature can take, D_{c,x_i} is the subset of class c samples taking the value x_i on the i th feature, and D_{c,x_i,x_j} is the subset of class c samples taking the value x_i on the i th feature while taking the value x_j on the j th feature.

AODE Example

- Looking at the watermelon data 3.0, we have:

$$\begin{aligned}\hat{P}_{\text{true}, \text{muffled}} &= \hat{P}(\text{ripe} = \text{true}, \text{sound} = \text{muffled}) \\ &= \frac{6 + 1}{17 + 3 \times 2} = 0.304,\end{aligned}$$

$$\begin{aligned}\hat{P}_{\text{hollow} | \text{true}, \text{muffled}} &= \hat{P}(\text{umbilicus} = \text{hollow} | \\ &\quad \text{ripe} = \text{true}, \text{sound} = \text{muffled}) \\ &= \frac{3 + 1}{6 + 3} = 0.444.\end{aligned}$$

- Hence, training process of AODE is about ‘counting’ the number of training samples satisfying the conditions, similar to Naïve Bayes classifiers.
- AODE does not require model selection, so it supports quick predictions with precalculations, lazy learning, and incremental learning.

Outlook

- We can see that relaxing the attribute conditional independency assumption to one-dependent assumption leads to better generalization ability.
- This leads to the question: Is it possible to make further improvement by considering higher order dependencies?
- Models that deal with this are **Bayesian Networks**, which utilize Directed Acyclic Graphs (DAG) to represent dependencies among features and build Conditional Probability Tables to describe the joint probabilities of features.
- It is worth mentioning that, improving the generalization performance requires abundant data.

Summary

In this chapter we have learned

- the concepts of **Bayesian Decision Theory**.
- why the **Bayes Classifier** is theoretically optimal.
- that the **Naïve Bayes Classifier** shortens the estimation process.
- how the **Semi-Naïve Bayes Classifier** makes further assumptions to simplify the computation.

Introduction to Machine Learning: Theory and Application

Chapter 7

Alexander Szimayer

Universität Hamburg

Summer Term 2024

7. Clustering

In this chapter we

- learn how to classify unlabeled data using **clustering algorithms**.
- define **performance measures** and **distance calculation** for those algorithms.
- introduce two **prototype-based clustering** algorithms which assume the clustering structure can be represented by a set of prototypes.
- discuss **density-based clustering** which determine the clustering structure by the density of sample distributions.

7.1 Clustering Problem

- **Unsupervised learning** uses unlabeled training samples to discover underlying properties and patterns.
- Among unsupervised learning techniques **Clustering** is the most researched and applied technique.
- The aim is to partition a data set into disjoint subsets, where each subset is called a **cluster**.
- Each cluster is potentially corresponding to a concept. In our watermelon example a cluster could be “light green watermelon” or “seedless watermelon”.
- ...

- ...
- But, the clustering algorithms are unaware of such concepts and are only responsible for creating the clusters. The concept carried by each cluster is **interpreted by the user**.
- Clustering can be used by itself to identify the inherent structure of data.
- Alternatively, it can lay the foundation for further data analysis and be used as a pre-processing technique for other learning tasks such as classification.

Definition

- Let $D = \{x_1, x_2, \dots, x_m\}$ be a data set with m unlabeled samples, where each sample $x_i = (x_{i1}; x_{i2}, \dots, x_{in})$ is an n -dimensional vector.
- A clustering algorithm partitions the data set D into k disjoint clusters C_l with $l = 1, 2, \dots, k$, where:

$$C_{l'} \bigcap_{l' \neq l} C_l = \emptyset,$$

$$D = \bigcup_{l=1}^k C_l.$$

- The **cluster label** of sample x_j is denoted as $\lambda_j \in \{1, 2, \dots, k\}$. The clustering results can then be represented as a cluster label vector $\lambda = \{\lambda_1; \lambda_2; \dots; \lambda_m\}$ with m elements.

7.2 Validity Indices

- Clustering results need to be evaluated via some **validity indices**.
- Embedding validity indices into the optimization objective generates clusters that are more aligned to the desired results.
- Clusters should be created such that they have a high **intra-cluster similarity** and a low **inter-cluster similarity**.
- There are two types of cluster validity indices:
 - ① The **external index** compares the clustering results against the reference model.
 - ② The **internal index** evaluates the clustering results without using any reference model.

External Indices (1/2)

- Given a data set $D = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_m\}$, suppose a cluster algorithm produces the clusters $\mathcal{C} = \{C_1, C_2, \dots, C_k\}$. The reference model gives the clusters $\mathcal{C}^* = \{C_1^*, C_2^*, \dots, C_s^*\}$, and λ and λ^* denote the clustering labels of \mathcal{C} and \mathcal{C}^* .
- For each pair of samples, we can define four terms:

$$a = |SS|, \quad SS = \{(\mathbf{x}_i, \mathbf{x}_j) | \lambda_i = \lambda_j, \lambda_i^* = \lambda_j^*, i < j\}, \quad (7.1)$$

$$b = |SD|, \quad SD = \{(\mathbf{x}_i, \mathbf{x}_j) | \lambda_i = \lambda_j, \lambda_i^* \neq \lambda_j^*, i < j\}, \quad (7.2)$$

$$c = |DS|, \quad DS = \{(\mathbf{x}_i, \mathbf{x}_j) | \lambda_i \neq \lambda_j, \lambda_i^* = \lambda_j^*, i < j\}, \quad (7.3)$$

$$d = |DD|, \quad DD = \{(\mathbf{x}_i, \mathbf{x}_j) | \lambda_i \neq \lambda_j, \lambda_i^* \neq \lambda_j^*, i < j\}. \quad (7.4)$$

- E.g. the set SS includes the sample pairs where both samples belong to the same cluster in \mathcal{C} and also belong in the same cluster in \mathcal{C}^* .
- Each sample pair $(\mathbf{x}_i, \mathbf{x}_j)$ with $i < j$ can only appear in one set.
Therefore, we have $a + b + c + d = m(m - 1)/2$.

External Indices (2/2)

- Commonly used external indices can be defined with equations (7.1)-(7.4):

- ① Jaccard Coefficient (JC):

$$JD = \frac{a}{a + b + c}.$$

- ② Fowlkes and Mallows Index (FMI):

$$FMI = \sqrt{\frac{a}{a + b} \times \frac{a}{a + c}}.$$

- ③ Rand Index (RI):

$$RI = \frac{2(a + d)}{m(m - 1)}.$$

- These external validity indices take values in the interval $[0, 1]$. The larger the index value, the better is the clustering quality.

Internal Indices (1/2)

- Internal validity indices evaluate the clustering quality without using a reference model. Given the generated clusters $\mathcal{C} = \{C_1, C_2, \dots, C_k\}$, four terms can be defined,

$$\text{avg}(C) = \frac{2}{|C|(|C| - 1)} \sum_{1 \leq i < j \leq |C|} \text{dist}(\mathbf{x}_i, \mathbf{x}_j), \quad (7.5)$$

$$\text{diam}(C) = \max_{1 \leq i < j \leq |C|} \text{dist}(\mathbf{x}_i, \mathbf{x}_j), \quad (7.6)$$

$$d_{\min}(C_i, C_j) = \min_{\mathbf{x}_i \in C_i, \mathbf{x}_j \in C_j} \text{dist}(\mathbf{x}_i, \mathbf{x}_j), \quad (7.7)$$

$$d_{cen}(C_i, C_j) = \text{dist}(\boldsymbol{\mu}_i, \boldsymbol{\mu}_j), \quad (7.8)$$

where $\boldsymbol{\mu} = \frac{1}{|C|} \sum_{l \leq i \leq |C|} \mathbf{x}_i$ denotes the centroid of cluster C and $\text{dist}(\cdot, \cdot)$ measures the distance between two samples.

Internal Indices (2/2)

- With equations (7.5)-(7.8) the commonly used internal validity indices can be defined:

- ① Davies-Bouldin Index (DBI):

$$DBI = \frac{1}{k} \sum_{i=1}^k \max_{j \neq i} \left(\frac{\text{avg}(C_i) + \text{avg}(C_j)}{d_{cen}(C_i, C_j)} \right).$$

- ② Dunn Index (DI):

$$DI = \min_{1 \leqslant i \leqslant k} \left\{ \min_{j \neq i} \left(\frac{d_{\min}(C_i, C_j)}{\max_{1 \leqslant l \leqslant k} \text{diam}(C_l)} \right) \right\}.$$

- A smaller value of DBI indicates better clustering quality, while a larger value of DI indicates better clustering quality.

7.3 Distance Calculation

- Given a function $dist(.,.)$, if it is a distance measure, then it must fulfill the following **distance measure axioms**:
 - 1 Non-negativity: $dist(x_i, x_j) \geq 0$,
 - 2 Identity of indiscernibles: $dist(x_i, x_j) = 0$, only if $x_i = x_j$,
 - 3 Symmetry: $dist(x_i, x_j) = dist(x_j, x_i)$,
 - 4 Subadditivity: $dist(x_i, x_j) \leq dist(x_i, x_k) + dist(x_k, x_j)$.

Distance Measures

- Given two samples $\mathbf{x}_i = (x_{i1}; x_{i2}; \dots; x_{in})$ and $\mathbf{x}_j = (x_{j1}; x_{j2}; \dots; x_{jn})$, we can compute the commonly used **Minkowski distance**:

$$dist_{mk}(\mathbf{x}_i, \mathbf{x}_j) = \left(\sum_{u=1}^n |x_{iu} - x_{ju}|^p \right)^{\frac{1}{p}}. \quad (7.9)$$

For $p \geq 1$, equation (7.9) satisfies the distance measure axioms.

- For $p = 2$, the Minkowski distance becomes the **Euclidean distance**:

$$dist_{ed}(\mathbf{x}_i, \mathbf{x}_j) = \|\mathbf{x}_i - \mathbf{x}_j\|_2 = \sqrt{\sum_{u=1}^n |x_{iu} - x_{ju}|^2}.$$

- For $p = 1$, the Minkowski distance becomes the **Manhattan distance**:

$$dist_{man}(\mathbf{x}_i, \mathbf{x}_j) = \|\mathbf{x}_i - \mathbf{x}_j\|_1 = \sum_{u=1}^n |x_{iu} - x_{ju}|.$$

Attribute Types

- Attributes are generally divided into two types:
 - ① **Continuous attributes**, which have infinite domains,
 - ② **Categorical attributes**, which have finite domains.
- For distance calculation, it is more important to consider whether the attributes include **ordinal information**:
 - ① Ordinal attributes: The distance of a categorical attribute with the domain $\{1, 2, 3\}$ can be calculated with attribute values, that is, '1' is closer to '2' than '3'.
 - ② Non-ordinal attributes: The distance of a discrete attribute with the domain $\{\text{aircraft}, \text{train}, \text{ship}\}$ cannot be directly calculated with the attribute values.
- Note that the Minkowski distance is only applicable to ordinal attributes.

The weighted Minkowski distance

- If different attributes have **different importance**, we can use a weighted distance.
- The weighted Minkowski distance is,

$$dist_{wmk}(x_i, x_j) = (w_1 \times |x_{i1} - x_{j1}|^p + \cdots + w_n \times |x_{in} - x_{jn}|^p)^{\frac{1}{p}},$$

where the weights $w_i \geq 0$ ($i = 1, 2, \dots, n$) represent the importance of attributes, and typically $\sum_{i=1}^n w_i = 1$.

Further Remarks

- Often, similarity measures are defined via some kinds of distances.
The larger the distance, the lower the similarity.
- However, the distances used in defining similarity measures are not required to satisfy all distance measure axioms, particularly the subadditivity (axiom 4).
- If subadditivity is not satisfied, the distances are called **non-metric distances**.
- In this chapter we defined the distance calculation methods in advance. In reality, it is sometimes necessary to determine the distance calculation method based on the data samples via distance metric learning.

7.4 Prototype Clustering

- In this section you will learn more about two well-known prototype-based clustering algorithms:
 - ① k-Means Clustering,
 - ② Learning Vector Quantization.
- Generally, prototype clustering is a family of clustering algorithms that assumes the clustering structure can be represented by a set of prototypes.
- These algorithms usually start with some initial prototypes, and then iteratively update and optimize the prototypes.

7.4.1 k-Means Clustering

- Given a data set $D = (x_1, x_2, \dots, x_m)$, the k-means algorithm **minimizes the squared error of clusters** $\mathcal{C} = \{C_1, C_2, \dots, C_k\}$,

$$E = \sum_{i=1}^k \sum_{x \in C_i} \|x - \mu_i\|_2^2, \quad (7.10)$$

where $\mu_i = \frac{1}{|C_i|} \sum_{x \in C_i} x$ is the mean vector of cluster C_i .

- Equation (7.10) describes the closeness between the mean vector of a cluster and the samples within that cluster. Hence, a small E indicates a high intra-cluster similarity.
- Minimizing equation (7.10) is not easy because it requires evaluations of all possible partitions of the data set D . Therefore, the k -means algorithm takes a **greedy strategy** and adopts an **iterative optimization method** to find an approximate solution of this equation.

k-Means Algorithm

Input: Data set $D = \{x_1, x_2, \dots, x_m\}$;

Number of clusters k .

Process:

```
1: Randomly select  $k$  samples as the initial mean vectors  
    $\{\mu_1, \mu_2, \dots, \mu_k\}$ ;  
2: repeat  
3:    $C_i = \emptyset (1 \leq i \leq k)$ ;  
4:   for  $j = 1, 2, \dots, m$  do  
5:     Compute the distance between sample  $x_j$  and each mean  
       vector  $\mu_i (1 \leq i \leq k)$ :  $d_{ji} = \|x_j - \mu_i\|_2$ ;  
6:     According to the nearest mean vector, decide the cluster label  
       of  $x_j$ :  $\lambda_j = \arg \min_{i \in \{1, 2, \dots, k\}} d_{ji}$ ;  
7:     Move  $x_j$  to the corresponding cluster:  $C_{\lambda_j} = C_{\lambda_j} \cup \{x_j\}$ ;  
8:   end for  
9:   for  $i = 1, 2, \dots, k$  do  
10:    Compute the updated mean vectors:  $\mu'_i = \frac{1}{|C_i|} \sum_{x \in C_i} x$ ;  
11:    if  $\mu'_i \neq \mu_i$  then  
12:      Update the current mean vector  $\mu_i$  to  $\mu'_i$ ;  
13:    else  
14:      Leave the current mean vector unchanged.  
15:    end if  
16:   end for  
17: until All mean vectors remain unchanged
```

Output: Clusters $\mathcal{C} = \{C_1, C_2, \dots, C_k\}$.

Figure 7.1: k-means pseudocode.

Example (1/4)

- Table 7.1 shows the watermelon data set 4.0, where x_i represent the sample with the ID i , and is a two-dimensional vector containing the attributes density and sugar.

ID	density	sugar	ID	density	sugar	ID	density	sugar
1	0.697	0.460	11	0.245	0.057	21	0.748	0.232
2	0.774	0.376	12	0.343	0.099	22	0.714	0.346
3	0.634	0.264	13	0.639	0.161	23	0.483	0.312
4	0.608	0.318	14	0.657	0.198	24	0.478	0.437
5	0.556	0.215	15	0.360	0.370	25	0.525	0.369
6	0.403	0.237	16	0.593	0.042	26	0.751	0.489
7	0.481	0.149	17	0.719	0.103	27	0.532	0.472
8	0.437	0.211	18	0.359	0.188	28	0.473	0.376
9	0.666	0.091	19	0.339	0.241	29	0.725	0.445
10	0.243	0.267	20	0.282	0.257	30	0.446	0.459

Table 7.1: The watermelon data set 4.0

Example (2/4)

- Suppose we set $k = 3$, then the algorithm randomly picks up three samples x_6 , x_{12} , and x_{24} as the initial mean vectors, that is:

$$\mu_1 = (0.403; 0.237), \quad \mu_2 = (0.343; 0.099), \quad \mu_3 = (0.478; 0.437).$$

- Then the distances for the sample $x_1 = (0.697; 0.460)$ to the three current mean vectors μ_1 , μ_2 , and μ_3 are 0.369, 0.506, and 0.220.
- If we compare the three distances, we can see that the distance to μ_3 is the shortest. Therefore, x_1 is assigned to cluster C_3 .

Example (3/4)

- After evaluating all samples in the data set, we can find the following cluster assignments:

$$C_1 = \{x_3, x_5, x_6, x_7, x_8, x_9, x_{10}, , x_{13}, x_{14}, x_{17}, x_{18}, x_{19}, x_{23}\},$$

$$C_2 = \{x_{11}, x_{12}, x_{16}\},$$

$$C_3 = \{x_1, x_2, x_4, x_{15}, x_{21}, x_{22}, x_{24}, x_{25}, x_{26}, x_{27}, x_{28}, x_{29}, x_{30}\}.$$

- From C_1 , C_2 , and C_3 , we can compute the new mean vectors:

$$\mu'_1 = (0.493; 0.207), \quad \mu'_2 = (0.394; 0.066), \quad \mu'_3 = (0.602, 0.396).$$

- This process repeats until convergence. For example, as illustrated in figure 7.2, the k-means algorithm finds the final cluster assignments when the 5th iteration produced the same clusters as the 4th iteration.

Example (4/4)

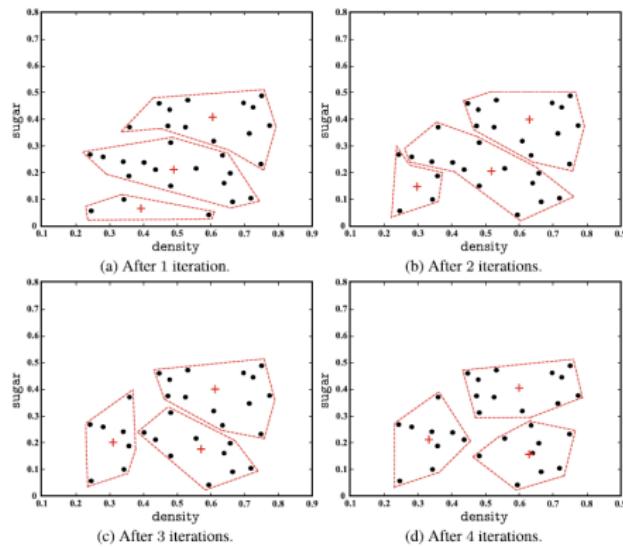


Figure 7.2: Results of the k-means algorithm on the watermelon data set 4.0 with $k = 3$. The samples and mean vectors are represented by ‘•’ and ‘+’, respectively. The red dashed lines are the boundaries of clusters

7.4.2 Learning Vector Quantization (LVQ)

- LVQ is similar to the k-means algorithm, with the difference, that it **requires labeled data samples**, that is, the clustering process is assisted by supervised information.
- Suppose a data set $D = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_m, y_m)\}$, where each sample \mathbf{x}_j is described by an n -dimensional feature vector $(x_{j1}; x_{j2}; \dots; x_{jn})$ and a class label $y_j \in \mathcal{Y}$.
- The objective of LVQ is to learn a set of n -dimensional prototype vectors $\{\mathbf{p}_1, \mathbf{p}_2, \dots, \mathbf{p}_q\}$, where each prototype vector represents one cluster with the class label $t_i \in \mathcal{Y}$.

Learning Vector Quantization Algorithm

Input: Training set $D = \{(x_1, y_1), (x_2, y_2), \dots, (x_m, y_m)\}$;

Number of prototype vectors q ;

Initial labels of prototype vectors $\{t_1, t_2, \dots, t_q\}$;

Learning rate η .

Process:

```

1: Initialize a set of prototype vectors  $\{p_1, p_2, \dots, p_q\}$ ;
2: repeat
3:   Randomly pickup a sample  $(x_j, y_j)$  from the data set  $D$ ;
4:   Compute the distance between  $x_j$  and  $p_i$  ( $1 \leq i \leq q$ ):  $d_{ji} = \|x_j - p_i\|_2$ ;
5:   Find the nearest prototype vector  $p_{i^*}$  for  $x_j$ , where  $i^* = \arg \min_{i \in \{1, 2, \dots, q\}} d_{ji}$ ;
6:   if  $y_j = t_{i^*}$  then
7:      $p' = p_{i^*} + \eta \cdot (x_j - p_{i^*})$ ;
8:   else
9:      $p' = p_{i^*} - \eta \cdot (x_j - p_{i^*})$ ;
10:  end if
11:  Update the prototype vector  $p_{i^*}$  to  $p'$ .
12: until The termination condition is met
Output: Prototype vectors  $\{p_1, p_2, \dots, p_q\}$ .
```

- Example for line 1:

Randomly select a sample with class label t_q as the prototype vector for the q th cluster.

- Example for line 12:

Maximum number of iterations is reached, or there is minor or even no update to the prototype vectors.

Figure 7.3: Learning Vector Quantization pseudocode.

Updating the Prototype Vectors

- Looking at the LVQ algorithm, the **core of LVQ** is updating the prototype vectors (line 6-10).
- If the class labels of a sample x_j and its nearest prototype vector p_{i^*} are the same, then p_{i^*} is moved toward x_j .
- In line 7 you can see, that the prototype vector is updated by,

$$\mathbf{p}' = \mathbf{p}_{i^*} + \eta \cdot (\mathbf{x}_j - \mathbf{p}_{i^*}),$$

and the distance between \mathbf{p}' and \mathbf{x}_j is,

$$\begin{aligned}\|\mathbf{p}' - \mathbf{x}_j\|_2 &= \|\mathbf{p}_{i^*} + \eta \cdot (\mathbf{x}_j - \mathbf{p}_{i^*}) - \mathbf{x}_j\|_2, \\ &= (1 - \eta) \cdot \|\mathbf{p}_{i^*} - \mathbf{x}_j\|_2.\end{aligned}$$

- If $\eta \in (0, 1)$, the prototype vector \mathbf{p}_{i^*} becomes closer to \mathbf{x}_j after updating \mathbf{p}' .
- If \mathbf{p}_{i^*} and \mathbf{x}_j have different class labels, the distance between the updated prototype vector and \mathbf{x}_j is increased to $(1 + \eta) \cdot \|\mathbf{p}_{i^*} - \mathbf{x}_j\|_2$. This is further away from \mathbf{x}_j .

The Voronoi Tessellation

- We can make clustering assignments for the sample space \mathcal{X} with a set of learned prototype vectors $\{\mathbf{p}_1, \mathbf{p}_2, \dots, \mathbf{p}_q\}$, which means, each sample \mathbf{x} is assigned to the cluster represented by the nearest prototype vector.
- We then get a **region** R_i , that is defined by a prototype vector \mathbf{p}_i . The distance from any sample in region R_i to \mathbf{p}_i is not larger than the distance to any other prototype vector $\mathbf{p}_{i'}(i' \neq i)$, that is:

$$R_i = \{\mathbf{x} \in \mathcal{X} \mid \|\mathbf{x} - \mathbf{p}_i\|_2 \leq \|\mathbf{x} - \mathbf{p}_{i'}\|_2, i' \neq i\}.$$

- The partition of sample space \mathcal{X} formed by $\{R_1, R_2, \dots, R_q\}$, is known as the **Voronoi tessellation**.

Learning Vector Quantization: Example (1/3)

- We again take the watermelon data set 4.0 from Table 7.1 as an example.
- Suppose, c_2 is the class label of the samples with ID 9-21, and c_1 is the class label for the rest of the samples.
- Set $q = 5$, so the goal is to find five prototype vectors p_1, p_2, p_3, p_4 , and p_5 and the corresponding class labels to c_1, c_2, c_2, c_1 , and c_1 . The pre-specified learning rate is $\eta = 0.1$.
- First, the algorithm is initializing the prototype vector of each cluster to a sample that has the same class label as the pre-defined class label of the cluster.

Learning Vector Quantization: Example (2/3)

- Let the selected samples for the five clusters be $x_5, x_{12}, x_{18}, x_{23}$, and x_{29} , and the randomly selected sample is x_1 .
- Then, in the first iteration the distances to the current prototype vectors p_1, p_2, p_3, p_4 , and p_5 are 0.283, 0.506, 0.434, 0.260 and 0.032.
- We can see that p_5 is the nearest prototype vector to x_1 , and they have the same class label c_1 . Therefore, LVQ will update p_5 to a new prototype vector:

$$\begin{aligned}p' &= p_5 + \eta \times (x_1 - x_5) \\&= (0.752; 0.445) + 0.1 \times ((0.697; 0.460) - (0.725; 0.445)) \\&= (0.722; 0.447).\end{aligned}$$

- Figure 7.4 shows the clustering results after different iterations.

Learning Vector Quantization: Example (3/3)

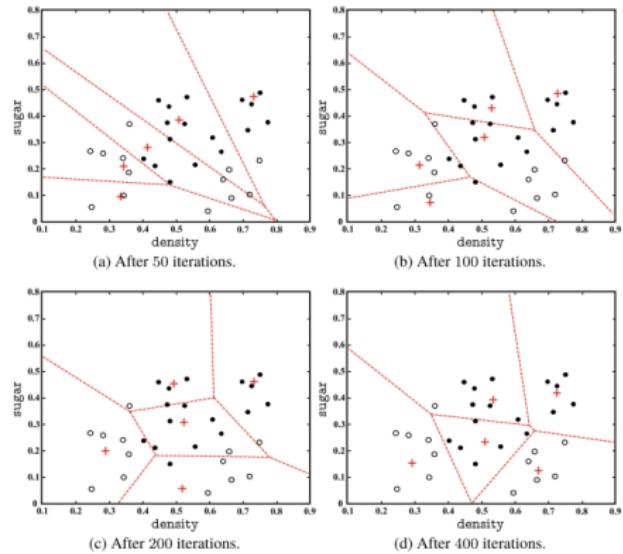


Figure 7.4: Results of the LVQ algorithm after different iterations on the watermelon data set 4.0 with $q = 5$. The symbols ‘●’, ‘○’, ‘+’ represent the class c_1 samples, the class c_2 samples, and the prototype vectors. The red dashed lines show the Voronoi tessellation

7.5 Density Clustering

- Clustering algorithms that assume the clustering structure can be determined by the densities of sample distributions, are called **density clustering** algorithms.
- They typically evaluate the connectivity between samples from the density perspective and expand the clusters by adding connectable samples.
- **DBSCAN** (Density-Based Spatial Clustering of Applications with Noise) is a density clustering algorithm. It characterizes the density of sample distribution by a pair of **neighborhood parameters** $(\epsilon, \text{MinPts})$.

DBSCAN (1/3)

- We need six concepts to understand how DBSCAN defines a cluster:

Given the data set $D = (\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_m)$,

- ① **ϵ -neighborhood:** The ϵ -neighborhood for $\mathbf{x}_j \in D$ includes all samples in D that have a distance to \mathbf{x}_j no larger than ϵ , that is,
 $N_\epsilon(\mathbf{x}_j) = \{\mathbf{x}_i \in D \mid \text{dist}(\mathbf{x}_i, \mathbf{x}_j) \leq \epsilon\}$.
- ② **Core object:** If the ϵ -neighborhood of \mathbf{x}_j includes at least MinPts samples, that is, $|N_\epsilon(\mathbf{x}_j)| \geq \text{MinPts}$, then \mathbf{x}_j is a core object.
- ③ **Directly density-reachable:** \mathbf{x}_j is directly density-reachable by \mathbf{x}_i , if \mathbf{x}_i is a core object and \mathbf{x}_j is in the ϵ -neighborhood of \mathbf{x}_i .
- ④ **Density-reachable:** \mathbf{x}_j is density-reachable by \mathbf{x}_i , if there exists a sequence of samples p_1, \dots, p_n , where $p_1 = \mathbf{x}_i, p_n = \mathbf{x}_j$, and p_{i+1} is directly density-reachable by p_i .
- ⑤ **Density-connected:** \mathbf{x}_i and \mathbf{x}_j are density-connected, if there exists \mathbf{x}_k such that both \mathbf{x}_i and \mathbf{x}_j are density-reachable by \mathbf{x}_k .
- ⑥ **Noisy sample:** \mathbf{x}_i is a noisy sample. If it is not reachable from any other \mathbf{x}_k .

DBSCAN (2/3)

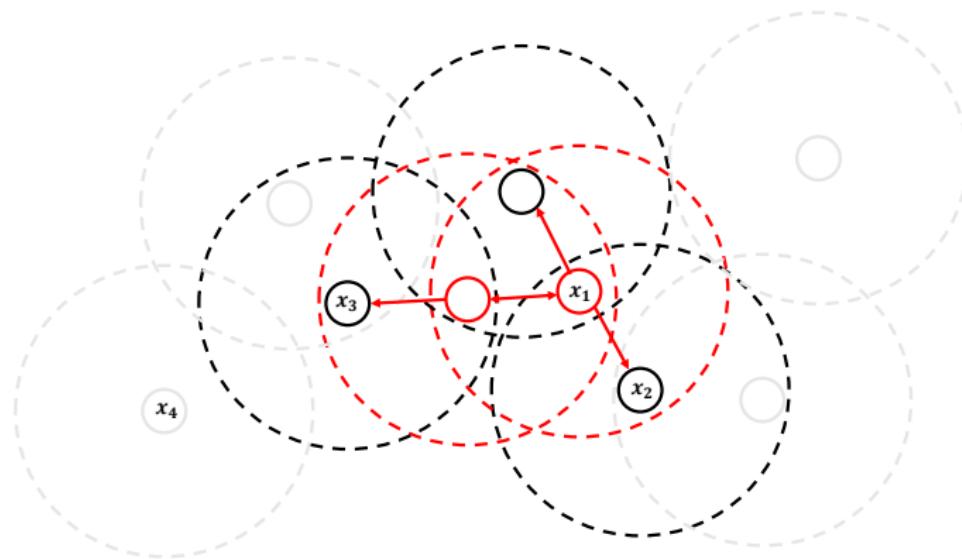


Figure 7.5: The basic concepts of DBSCAN ($\text{MinPts} = 3$): The dashed circles show the ϵ -neighborhood; x_1 is a core object; x_2 is directly density-reachable by x_1 ; x_3 is density-reachable by x_1 ; all red and black samples are density-connected and form the cluster (for example x_2 and x_3); x_4 is a noisy sample

DBSCAN (3/3)

- With the six concepts, the DBSCAN defines a cluster as the largest set of density-connected samples derived by density-reachable relationships.
- Formally: given the neighborhood parameters $(\epsilon, \text{MinPts})$, a cluster $C \subseteq D$ is a nonempty subset with the following properties:
 - Connectivity:** $x_i \in C, x_j \in C \Rightarrow x_i$ and x_j are density connected,
 - Maximality:** $x_i \in C, x_j$ is density-reachable by $x_i \Rightarrow x_j \in C$.
- Now we can identify the clusters using that, if x is a core object and $X = \{x' \in D | x'$ is density-reachable by $x\}$ denotes the set of samples density-reachable by x , then X is a cluster that satisfies the connectivity and the maximality.
- Therefore, the DBSCAN algorithm generates clusters by expanding from core objects, as illustrated in Algorithm 7.4.

DBSCAN Algorithm

Input: Data set $D = \{x_1, x_2, \dots, x_m\}$;
Neighborhood parameters $(\epsilon, MinPts)$.

Process:

```
1: Initialize the set of core objects:  $\Omega = \emptyset$ ;
2: for  $j = 1, 2, \dots, m$  do
3:   Determine the  $\epsilon$ -neighborhood  $N_\epsilon(x_j)$  of sample  $x_j$ ;
4:   if  $|N_\epsilon(x_j)| \geq MinPts$  then
5:     Add sample  $x_j$  to the set of core objects:  $\Omega = \Omega \cup \{x_j\}$ ;
6:   end if
7: end for
8: Initialize the number of clusters:  $k = 0$ ;
9: Initialize the set of unprocessed samples:  $\Gamma = D$ ;
10: while  $\Omega \neq \emptyset$  do
11:   Keep a copy of the current unprocessed data set:  $\Gamma_{old} = \Gamma$ ;
12:   Randomly select a core object  $o \in \Omega$ , and initialize the queue
     $Q = \{o\}$ ;
13:    $\Gamma = \Gamma \setminus \{o\}$ ;
14:   while  $Q \neq \emptyset$  do
15:     Dequeue the first sample  $q$  from  $Q$ ;
16:     if  $|N_\epsilon(q)| \geq MinPts$  then
17:       Letting  $\Delta = N_\epsilon(q) \cap \Gamma$ ;
18:       Enqueue the samples in  $\Delta$  into  $Q$ ;
19:        $\Gamma = \Gamma \setminus \Delta$ ;
20:     end if
21:   end while
22:    $k = k + 1$ , generate cluster  $C_k = \Gamma_{old} \setminus \Gamma$ ;
23:    $\Omega = \Omega \setminus C_k$ .
24: end while
Output: Clusters  $\mathcal{C} = \{C_1, C_2, \dots, C_k\}$ .
```

Figure 7.6: DBSCAN pseudocode.

DBSCAN: Example (1/3)

- We again take the watermelon data set 4.0 in Table 7.1 as an example.
- Given the neighborhood-parameters ($\epsilon = 0.11$, $MinPts = 5$), we have to find the ϵ -neighborhood for every sample to identify the set of core objects:

$$\Omega = \{x_3, x_5, x_6, x_8, x_9, x_{13}, x_{14}, x_{18}, x_{19}, x_{24}, x_{25}, x_{28}, x_{29}\}.$$

- We randomly select a core object from Ω as a seed, here x_8 .
- To include all density-reachable samples and expand from it to include all density-reachable samples. These samples will then form a cluster, here the first generated cluster is:

$$C_1 = \{x_6, x_7, x_8, x_{10}, x_{12}, x_{18}, x_{19}, x_{20}, x_{23}\}.$$

DBSCAN: Example (2/3)

- After the cluster is generated, the DBSCAN removes all core objects in C_1 from $\Omega = \Omega \setminus C_1 = \{x_3, x_5, x_9, x_{13}, x_{14}, x_{24}, x_{25}, x_{28}, x_{29}\}$.
- Afterwards, the next cluster is generated by randomly selecting another core object from the updates Ω as seed. This will be repeated until there no element in Ω left.
- Figure 7.6 shows the clusters generated in different rounds for our example are:

$$C_2 = \{x_3, x_4, x_5, x_9, x_{13}, x_{14}, x_{16}, x_{17}, x_{21}\},$$

$$C_3 = \{x_1, x_2, x_{22}, x_{26}, x_{29}\},$$

$$C_4 = \{x_{24}, x_{25}, x_{27}, x_{28}, x_{30}\}.$$

DBSCAN: Example (3/3)

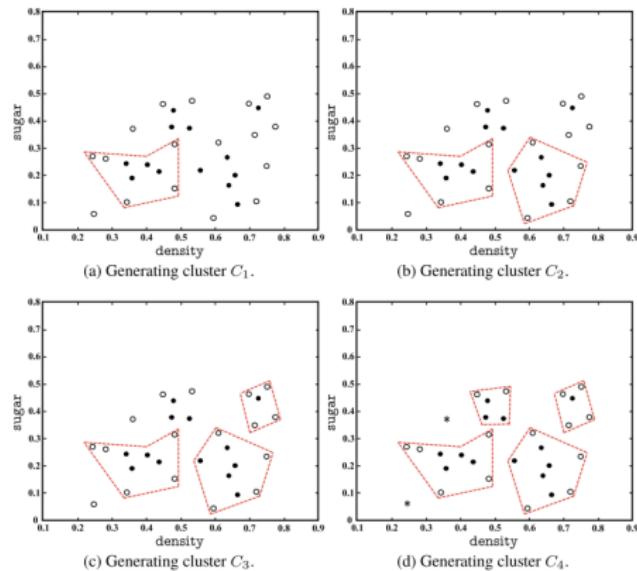


Figure 7.7: Results of the DBSCAN algorithm with $\epsilon = 0.11$ and MinPts = 5. The symbols ‘●’, ‘○’, ‘*’ represent the core objects, the non-core objects, and the noisy samples, respectively. The red dashed lines show the clusters

Summary

In this chapter we have learned

- how to classify unlabeled data using **clustering algorithms**.
- how to use **performance measures** and **distance calculation**.
- about two **prototype-based clustering** algorithms which assume the clustering structure can be represented by a set of prototypes.
- that **density-based clustering** determines the clustering structure by the density of sample distributions.

Introduction to Machine Learning: Theory and Application

Chapter 8

Prof. Dr. Alexander Szimayer

Universität Hamburg

Summer Term 2024

8. Neural Networks

In this chapter we

- define a simple neuron model.
- discuss a perceptron as well as a multi-layer network.
- work through the error backpropagation algorithm.
- discuss the global minimum and local minimum.

8.1 Neuron Model

- **Definition:**

'Artificial neural networks are massively parallel interconnected networks of simple (usually adaptive) elements and their hierarchical organizations which are intended to interact with the objects of the real world in the same way as biological nervous systems do.'

- In the context of machine learning, neural networks refer to 'neural networks learning', or in other words, the intersection of machine learning research and neural networks research.
- The basic element of neural networks is a **neuron** (also known as **unit**).

McCulloch-Pitts Model (M-P Neuron Model)

- In biological neural networks neurons are activated, when the electric potential exceeds a specific **threshold**.
- McCulloch and Pitts (1943) abstracted this process into a model.
- Each neuron receives input signals from ***n* neurons** via weighted connections.

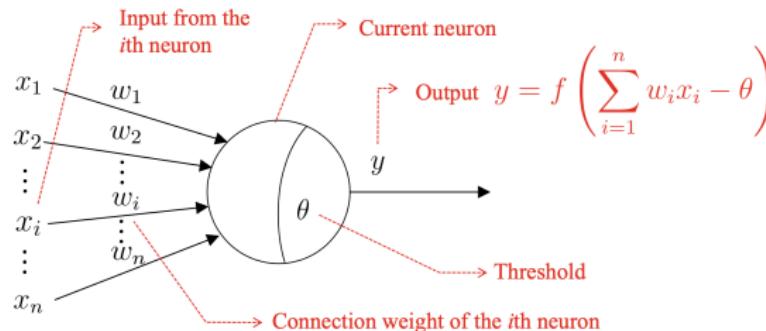


Figure 8.1: M-P neuron model.

- The weighted sum of the received signals is compared against the threshold (also known as **bias**).

Activation Function

- The output signal is produced by the **activation function** (also known as transfer function).
- The ideal activation function is the **step function**, which maps the input value to the output value.

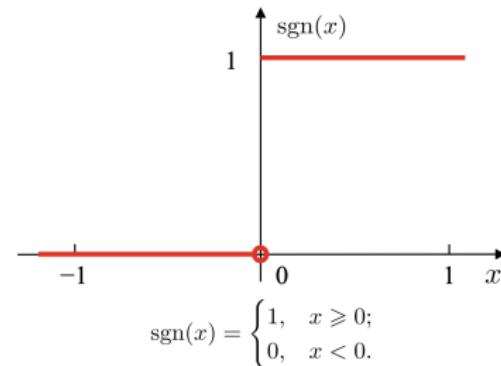


Figure 8.2: Step function.

- Since the step function has some undesired properties such as being **discontinuous** and **non-smooth**, the sigmoid function is used instead.

Sigmoid Function

- A typical sigmoid function squashes the input values from a large interval into the open unit interval $(0,1)$, and hence is also known as squashing function.

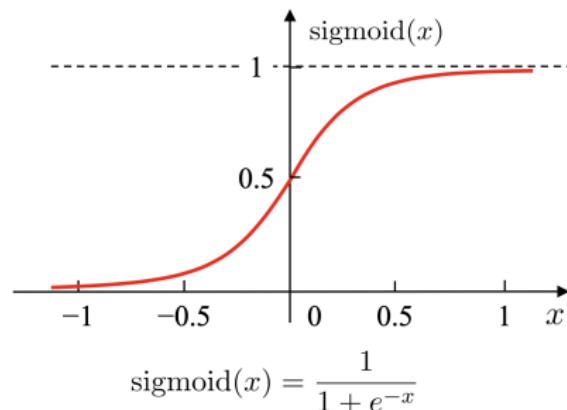


Figure 8.3: Sigmoid function.

Neural Network as Mathematical Model

- A neural network is derived by connecting the neurons into a layered structure.
- Regarding a neural network as a mathematical model with many parameters.
 - For example, for 10 pairwise linked neurons, there are 100 parameters, including 90 connection weights and 10 thresholds.
- The model consists of multiple functions, e.g., nesting
 $y_j = f(\sum_i w_i x_i - \theta_j)$ multiple times.

8.2 Perceptron and Multi-layer Network

- A **perceptron** is a binary classifier consisting of two layers of neurons.

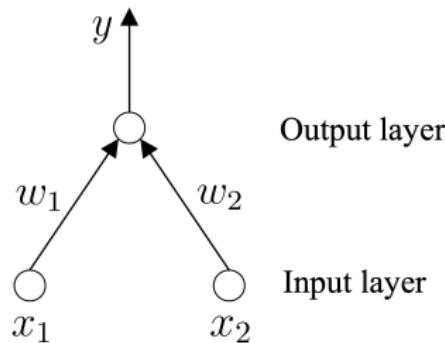


Figure 8.4: Perceptron with two input neurons.

- The input layer receives external signals and transmits them to the output layer, which is a M-P neuron (or **threshold logic unit**).

Logic Operations

- Suppose the function f in $y = f(\sum_i w_i x_i - \theta_j)$ is the step function.
- Perceptron can implement the following **logic operations**:
 - ① '**AND**' ($x_1 \wedge x_2$)
 - ② '**OR**' ($x_1 \vee x_2$)
 - ③ '**NOT**' ($\neg x_1$)
- The **weight** w_i and **threshold** θ can be learned from training data.
- Consider the threshold θ as a dummy node with the connection weight w_{n+1} and fixed input -1 , then the weight and threshold are unified as weight learning.

Learning of Perceptron (1/2)

- For training sample (x, y) , if the perceptron outputs \hat{y} , then the weight is updated by:

$$w_i \leftarrow w_i + \Delta w_i, \quad (8.1)$$

$$\Delta w_i = \eta(y - \hat{y})x_i, \quad (8.2)$$

where $\eta \in (0, 1)$ is known as the **learning rate**.

- Equation 8.1 shows the perceptron remains unchanged if it correctly predicts the sample.
- Otherwise, the weight is updated based on the degree of error.

Learning of Perceptron (2/2)

- The **learning ability** of perceptrons is rather weak since only the output layer has activation functions, that is, only one layer of functional neurons.
- ‘AND’, ‘OR’ and ‘NOT’ problems are all linearly separable.
- There must exist a linear hyperplane that can separate two classes if they are linearly separable.
- This means that the perceptron learning process is guaranteed to converge to an appropriate weight vector:

$$\mathbf{w} = (w_1; w_2; \dots; w_{n+1}).$$

- Otherwise, fluctuation will happen in the learning process, and no appropriate solution can be found since \mathbf{w} cannot be stabilized.

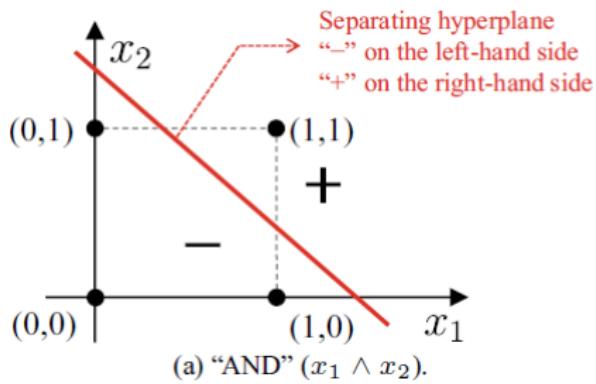
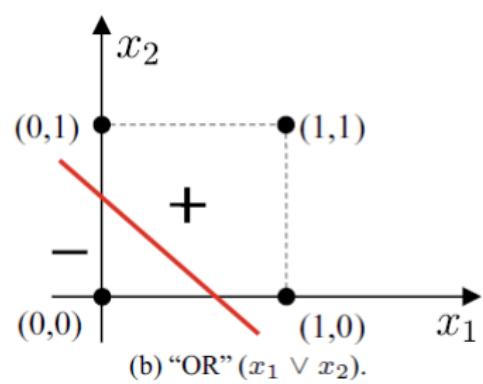
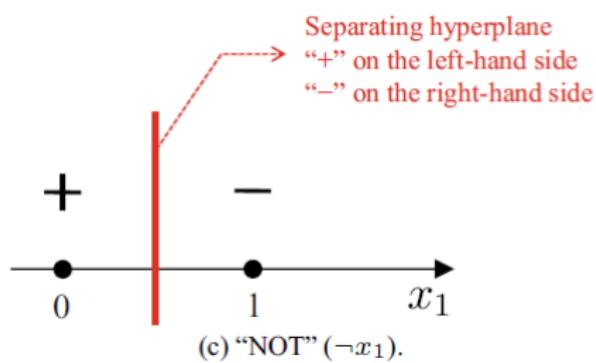
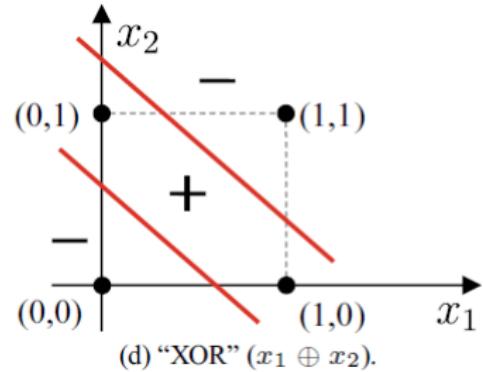
(a) "AND" ($x_1 \wedge x_2$).(b) "OR" ($x_1 \vee x_2$).(c) "NOT" ($\neg x_1$).(d) "XOR" ($x_1 \oplus x_2$).

Figure 8.5: "AND", "OR", and "NOT" are linearly separable problems. "XOR" is a nonlinearly separable problem.

Multi-Layer Feed-forward Neural Network (1/2)

- To solve nonlinear separable problems, multi-layer functional neurons can be used.
- The neuron layer between the input and output layer is known as the **hidden layer**, which has activation functions like the output layer does.

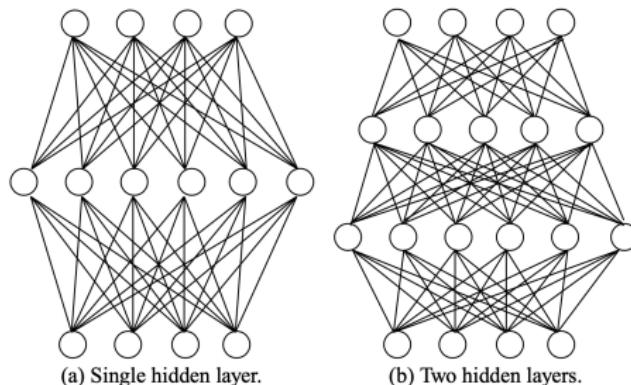


Figure 8.6: Two typical multi-layer neural network structures, in which the neurons in each layer are fully connected with the neurons in the next layer.

Multi-Layer Feed-Forward Neural Network (2/2)

- Neurons within the same layer or from non-adjacent layers are not connected.
- “**Feed-forward**” does not mean signals cannot be transmitted backward but refers to no recurrent or circular connections.
- The input layer receives external signals, the hidden and output layers process the signals, and the output layer outputs the processed signals.
- The “knowledge” learned by neural networks is in the **connection weights** and thresholds.

8.3 Error Backpropagation Algorithm

- The learning ability of multi-layer neural networks is much stronger than single-layer perceptrons.
- The **error backpropagation (BP) algorithm** is a representative and by far the most successful neural network learning algorithm, which trained most neural networks in real-world applications.
- The BP algorithm can train feed-forward but also **recurrent neural networks**.

BP Algorithm (1/3)

- Given a training set $D = \{(x_1, y_1), (x_2, y_2), \dots, (x_m, y_m)\}$, where $x_i \in R^d$, $y_i \in R^l$, that is, the input sample is described by d **attributes** and the output is a **l -dimensional** real-valued vector.

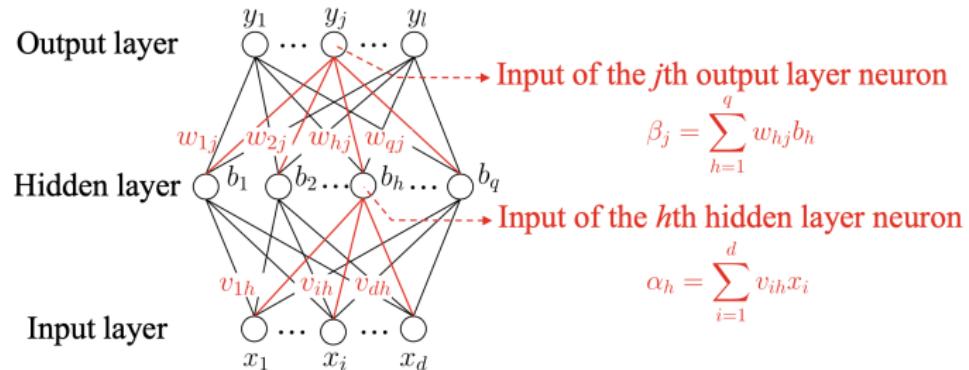


Figure 8.7: Multi-layer feed-forward neural network with d input neurons, l output neurons, and q hidden neurons.

BP Algorithm (2/3)

- Let θ_j denote the threshold of the j th neuron in the output layer, γ_h denote the threshold of the h th neuron in the hidden layer, v_{ih} denote the connection weight between the i th neuron of the input layer and the h th neuron of the hidden layer.
- Let w_{hj} denote the connection weight between the h th neuron of the hidden layer and the j th neuron of the output layer.
- The input received by the h th neuron in the hidden layer is given by:

$$\alpha_h = \sum_{i=1}^d v_{ih} x_i.$$

- The input received by the j th neuron in the output layer, where b_h is the output of the h neuron in the hidden layer is given by:

$$\beta_j = \sum_{h=1}^q w_{hj} b_h.$$

BP Algorithm (3/3)

- Suppose the neurons in both, the hidden layer and output layer, employ the sigmoid function.
- For training sample (x_k, y_k) , suppose the neural network outputs $\hat{y}_k = (\hat{y}_1^k, \hat{y}_2^k, \dots, \hat{y}_l^k)$, that is:

$$\hat{y}_l^k = f(\beta_j - \theta_j). \quad (8.3)$$

- The **MSE** of the neural network on sample (x_k, y_k) is:

$$E_k = \frac{1}{2} \sum_{j=1}^l (\hat{y}_j^k - y_j^k)^2. \quad (8.4)$$

BP Neural Network

- Multi-layer feed-forward neural network has:
 - in total $(d + l + 1)q + l$ parameters,
 - since there are $d \times q$ connection weights from input to hidden layer,
 - as well as $q \times l$ connections weights from hidden to output layer,
 - and q thresholds of hidden and l thresholds of output layer neurons.
- BP is an iterative learning algorithm, and each iteration employs the general form of perceptron learning rule to estimate and update parameter.
- The **update rule** of any parameter v is:

$$v \leftarrow v + \Delta v.$$

Gradient Descent Method (1/3)

- The BP algorithm employs the **gradient descent method** and tunes parameters toward the direction of negative gradient of objective.
- To demonstrate the derivation we use w_{hj} .
- For the error E_k and learning rate η , we have:

$$\Delta w_{hj} = -\eta \frac{\partial E_k}{\partial w_{hj}}.$$

- Note that w_{hj} first influences the input value β_j of the j th output layer neuron, then the output value \hat{y}_j^k , and finally the error E_k .

Gradient Descent Method (2/3)

- Hence, we have:

$$\frac{\partial E_k}{\partial w_{hj}} = \frac{\partial E_k}{\partial \hat{y}_j^k} \cdot \frac{\partial \hat{y}_j^k}{\partial \beta_j} \cdot \frac{\partial \beta_j}{\partial w_{hj}}.$$

- From the definition of β_j , we have:

$$\frac{\partial \beta_j}{\partial w_{hj}} = b_h.$$

- The sigmoid function has the following property:

$$f'(x) = f(x)(1 - f(x)).$$

Gradient Descent Method (3/3)

- Hence, from equations 8.3 and 8.4, we have:

$$\begin{aligned} g_j &= -\frac{\partial E_k}{\partial \hat{y}_j^k} \cdot \frac{\partial \hat{y}_j^k}{\partial \beta_j} \\ &= \hat{y}_j^k(1 - \hat{y}_j^k)(y_j^k - \hat{y}_j^k). \end{aligned} \tag{8.5}$$

- We have the update rule of w_{hj} as:

$$\Delta w_{hj} = \eta g_j b_h. \tag{8.6}$$

- Similarly, we can derive,

$$\Delta \theta_j = -\eta g_j, \tag{8.7}$$

$$\Delta v_{ih} = \eta e_h x_i, \tag{8.8}$$

$$\Delta \gamma_h = -\eta e_h, \tag{8.9}$$

with:

$$e_h = \frac{\partial E_k}{\partial b_h} \cdot \frac{\partial b_h}{\partial \alpha_h} = b_h(1 - b_h) \sum_{j=1}^l w_{hj} g_j. \tag{8.10}$$

Learning Rate η

- Learning rate $\eta \in (0, 1)$ controls the **step size** of the update in each round.
- An overly large learning rate may cause fluctuations, whereas a too small value leads to slow convergence.

Algorithm 1: Error Backpropagation

Input: Training set $D = \{(\mathbf{x}_k, \mathbf{y}_k)\}_{k=1}^m$; Learning rate η

Process:

1 Randomly initialize connection weights and thresholds from $(0, 1)$

repeat

2 **forall** $(x_k, (y_k)) \in D$ **do**

3 Compute the output \hat{y}_k for the current sample according to the
 current parameters and 8.3;

4 Compute the gradient term g_j of the hidden layer neurons
 according to 8.5;

5 Compute the gradient term e_h of the hidden layer neurons
 according to 8.10;

6 Update connection weights w_{hj} , v_{ih} and thresholds θ_j , γ_h
 according to 8.6 to 8.9.

7 **end**

8 **until** *The terminal condition is met*

Output: A feedforward neural network with determined connection
weights and thresholds.

Accumulated Error E

- The BP algorithm aims to minimize the accumulated error E on training set D :

$$E = \frac{1}{m} \sum_{k=1}^m E_k.$$

- The update rules in Algorithm 5.1 are derived from the error E_k of individual samples.
- If we use a similar method to derive the update rules for minimizing the accumulated error, then we have the **accumulated error backpropagation algorithm**.
- Both, standard and accumulated BP algorithm, are commonly used in practice.

Comparison of Standard and Accumulated BP Algorithm

- Standard BP algorithm:
 - The parameters are updated frequently since each update uses one sample, and hence the updates of different samples may “offset” each other.
 - As a result, the standard BP algorithm often needs more iterations to achieve the same minimum error.
- Accumulated BP algorithm:
 - The accumulated BP algorithm minimizes the accumulated error directly, and it tunes parameters less frequently since it tunes once after a full scan of the training set D .
 - When the training set D is large, the accumulated BP algorithm can become slow after the accumulated error decreases to a certain level. In contrast, the standard BP algorithm can achieve a reasonably good solution quicker.

Trial-By-Error

- A feed-forward neural network consisting of a single hidden layer with sufficient neurons that can approximate continuous functions of any complexity up to arbitrary accuracy.
- However, there is yet no principled method for setting the number of hidden layer neurons, and **trial-by-error** is usually used in practice.
- Along with the strong expressive power, BP neural networks suffer from **overfitting**, that is, the training error decreases while the testing error increases.
- There are **two general strategies** to alleviate the overfitting problem of BP neural networks:
 - Early Stopping,
 - Regularization.

First Strategy: Early Stopping

- Divide the data into training and validation set, where the training set is for calculating the gradient to update connection weights and thresholds, and the validation set is for estimating the error.
- Once the training error decreases while the validation error increases, the training process stops and returns the connection weights and thresholds corresponding to the minimum validation error.

Second Strategy: Regularization

- The main idea is to **add a regularization term** to the objective function, describing the complexity of neural network.
- Let E_k denote the error on the k th training sample and w_i denote the connection weights and thresholds, then the error objective function becomes:

$$E = \lambda \frac{1}{m} \sum_{k=1}^m E_k + (1 - \lambda) \sum_i w_i^2,$$

where $\lambda \in (0, 1)$ is a **trade-off** between the empirical error and the complexity of neural network.

- The value of λ is usually estimated by cross-validation.

8.4 Global Minimum and Local Minimum

- Since E represents the training error of the neural network, it is a function of connection weights w and thresholds θ .
- The training process of neural networks is a **parameter optimization process**, that is, searching for the set of parameters in parameter space that **minimizes E** .

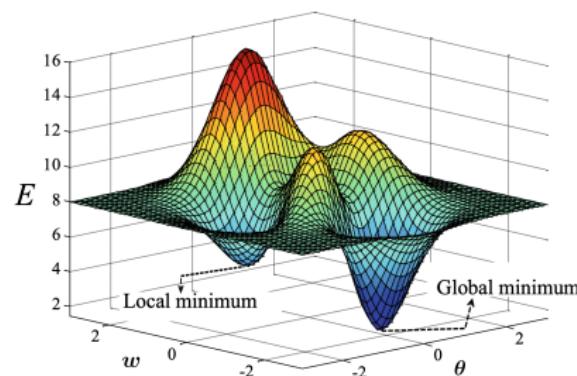


Figure 8.8: Two types of optimality: global and local minimum.

Formal Definition of Local and Global Minimum

- The weights and threshold combination $(\mathbf{w}^*; \theta^*)$ is a **local minimum** solution if there exists $\epsilon > 0$ such that:

$$E(\mathbf{w}; \theta) \geq E(\mathbf{w}^*; \theta^*), \forall (\mathbf{w}; \theta) \in \{(\mathbf{w}; \theta) | \|(\mathbf{w}; \theta) - (\mathbf{w}^*; \theta^*)\| \leq \epsilon\}.$$

- The weights and threshold combination $(\mathbf{w}^*; \theta^*)$ is the **global minimum** solution if,

$$E(\mathbf{w}; \theta) \geq E(\mathbf{w}^*; \theta^*),$$

holds for any $(\mathbf{w}; \theta)$ in the parameter space.

Gradient-based search Methods

- The most widely used parameter optimization methods are **gradient-based** search methods.
- These methods start from an initial solution and search for optimal parameters **iteratively**.
- In each round, the search direction at the current point is determined by the gradient of the error function.
- If there is more than one local minimum, the solution we found might not be the global minimum, and the parameter optimization is stuck at the local minimum, which is undesirable.

Strategies to ‘jump out’ from the Local Minimum

- ① Use **different sets of parameters** to initialize multiple neural networks and take the one with the smallest error.
- ② Use the **simulated annealing** technique, which accepts a worse solution at a certain **probability**. Decrease the probability of accepting suboptimal solutions as the search proceeds.
- ③ Use the **stochastic gradient descent method**, which introduces random factors to the gradient calculations rather than the exact calculations used in the standard gradient descent method.
- ④ The **genetic algorithm** is also used to train neural networks to better approximate the global minimum.

Summary

In this chapter we have learned

- how the McCulloch-Pitts model works.
- the difference between the activation and sigmoid function.
- about multi-layer feed-forward neural networks.
- how to apply the error backpropagation algorithm.
- the gradient descent method.
- two strategies how the BP algorithm deals with the overfitting problem.
- the difference between local and global minimum.