IM1002-232422M
NOTES MACHINE LEARNING

Open Universiteit

Author
Jan Baljan
07-02-2024

This document contains (lecture) notes that were made during the course **Machine Learning** at the Open Universiteit in the Netherlands. The notes are based on the book, *Introduction to Machine Learning*, by Alpaydin [1] and several pre-prints/articles that will be referenced throughout the text. Furthermore, the course workbook [2] was used as a guide for the topics, and additional examples and explanations were added by me.

Notations

x	Scalar value
$\mathbf{x}$	Vector
$\mathbf{X}$	Matrix
X	Random variable
$E[X]$	Expected value of random variable X
$\mathcal{X}$	Input
$E(\boldsymbol{\theta} \mid \mathcal{X})$	Error of model parameterised by $\boldsymbol{\theta}$ given input $\mathcal{X}$

Contents

1 Introduction to Machine Learning

At the core of computer science lies the idea of an **algorithm**. An algorithm is a sequence of instructions, that when carried out, transforms some input into some output. They have become an indispensable part of our everyday life, both professionally and socially. Despite decades of research, for some tasks, we do not have an algorithm. Some of these tasks we as human beings can do, and do effortlessly, without being aware of knowing how we do them, i.e., recognise a person from a photograph, playing chess, driving a car, and holding conversations in a foreign language.

In **machine learning** (ML), the idea is to learn these types of tasks with an initial general *model* that contains many *parameters*, and after proper adjustments of the parameters, yields a mechanism that performs these tasks successfully. **Learning** then corresponds to adjusting the values of these parameters so that the model matches best with the data it sees during training. Based on this training data, the model becomes specialised to a particular task that underlies the data, which becomes the algorithm for the task. Machine learning can, therefore, be defined as *the field of study that gives computers the ability to learn without being explicitly programmed.*

As an example, consider a supermarket chain that is interested in understanding customer behaviour. The chain stores details of each transaction via its virtual store, consisting of customer ID, goods bought and their amount, and total money spent. This data is complex, as customer behaviour changes based on time and by geographic location. Nevertheless, the data is not completely *random*. When people buy beer, they often buy chips; and people buy ice cream in summer, but hot chocolate in winter. These patterns are present in the data. It may not be possible to identify the process completely, but we can construct a **good and useful approximation**. That approximation may not *explain* everything, but it may account for some parts of the data. Such patterns, detected through the data, may help us understand the process, or we can use those patterns to make *predictions*.

Applications of machine learning methods to large databases is called **data mining**. The analogy is that large volumes of earth and raw materials are extracted from mines, that after processing, result in a small amount of valuable materials. Similarly, in data mining, large volumes of data is processed to construct simple models with valuable use, e.g., models with high predictive accuracy. In addition, machine learning is also part of **artificial intelligence**. To be intelligent, a system that is in a changing environment should have the ability to learn.

Machine learning is programming computers to **optimise** a **performance criterion** using example data or past experience. Using a model with parameters and training data or past experience, then learning amounts to the execution of a computer program to optimise the parameters of the model. The model may be **predictive** to make predictions in the *future*, or **descriptive** to gain *knowledge* from data, or both.

1.1 Machine Learning Paradigms

Traditionally, there are three types of learning approaches for machine learning models:

- **Supervised learning** which involves learning to predict a value or a category on the basis of a set of *features*, e.g., learning to predict who is likely to develop heart problems on the basis of cholesterol levels and blood pressure.
- **Unsupervised learning** which involves finding patterns in data, e.g., finding similar computer users on the basis of mouse movements or keystrokes.
- **Reinforcement learning** which involves learning a process from feedback on the outcome, e.g., learning to play the game *GO*.

1.1.1 Association Rules

In the case of retail, e.g., a supermarket chain, one application of machine learning is **basket analysis**. In this analysis, we try to find associations between products bought by customers, i.e., if people who buy X typically also buy Y, and if there is a customer who buys X and does not buys Y, then they are a potential new Y customer. Therefore, we are interested in learning a conditional probability of the form $P(Y \mid X)$, where Y is the product and X is the condition, which may be the set of products the customer has already purchased. For example $P(\mathit{chips} \mid \mathit{beer}) = 0.7$ implies the following **association rule**:

70 percent of customers who buy beer also buy chips.

Customer attributes, such as age, gender, etc., could also be included in the condition as the random variable D. Thus, the conditional probability may become $P(Y \mid X, D)$.

1.1.2 Classification

In **credit scoring**, a bank calculates the financial risk for providing loans given the amount of credit and the information of a customer. The customer information includes income, savings, profession, age, and financial history. The aim is then to infer a general rule about the association between customer's attributes and risk. This is an example of a **classification problem**, where there are two classes, i.e., low-risk and high-risk customers. After training with past data, a classification rule may be of the form

$$\text{IF income} > \theta_1 \text{ AND savings} > \theta_2 \text{ THEN high-risk ELSE low-risk}$$

where θ_1 and θ_2 are trained to be suitable values. This is an example of a **discriminant**, i.e., a function that separates the examples of different classes. Consider figure 1.1 that denotes the discriminant as a line that separates both classes. This plot is a function of θ_1 and θ_2.

If the future behaves similar to the past, then we can use the above rule to make correct **predictions** for novel instances. Given new instances with a certain income and savings, we can easily decide whether it is low-risk or high-risk. In terms of probability, we can compute the distribution $P(Y \mid X)$ where Y is the risk assessment $0/1$ (low-risk/high-risk) and X is the set of customer information. Then, $P(X = 1 \mid X = \mathcal{X}) = 0.8$ for a given customer

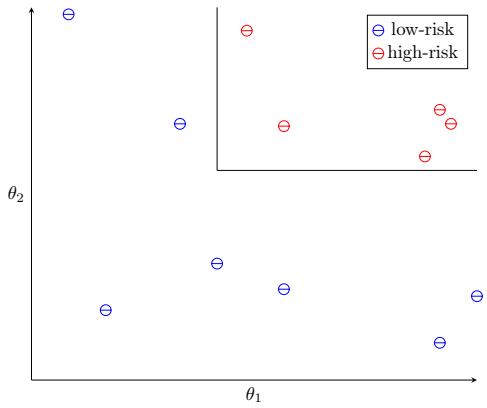


Figure 1.1: Example classification of low-risk and high-risk customers eligible for bank loans.

with information $\mathcal{X}$ denotes an 80 percent probability of the customer being high-risk, or equivalently a 20 percent probability of being low-risk.

Learning a rule from data allows for **knowledge extraction**. The rule is a simple model that explains the data, and looking at this model we have an explanation about the process underlying the data. Once the discriminant separating low-risk and high-risk customers is learnt, we have the knowledge of the properties of low-risk customers. We can use this information to target low-risk customers more efficiently, e.g., through advertisements. Learning also results in **compression**, in that the rule learnt by data is simpler than the data itself in the available time and space dimension.

1.1.3 Regression

If the task was to predict the price of a used car with attributes brand name, year, engine capacity, mileage, and other information, then the machine learning task is a **regression problem**. These are problems where the output is a number (continuous variable), such as the price of a used car.

Let X denote the car attributes and let Y denote the price of the car. Using past transactions, we can collect a training dataset and let the machine learning program fit a function to learn Y as a function of X

$$y = \mathbf{w}x + \mathbf{w}_0$$

where $\mathbf{w}$ and $\mathbf{w}_0$ are suitable learnt values. Both regression and classification are *supervised learning problems* where there is an input $\mathcal{X}$ and an output $\mathcal{Y}$, and the task is to learn the mapping from input to output.

In machine learning, we assume a model defined up to some parameters

$$\mathcal{Y} = g(\mathcal{X} \mid \theta)$$

where $g(\cdot)$ is the model and θ are its parameters. This function is the regression function, or in classification, it is the discriminant function that separates instances of different classes. The machine learning program optimises the parameters θ such that the approximation error is minimised.

1.1.4 Unsupervised Learning

In supervised learning, the aim is to learn a mapping from the input to the output whose correct values are provided by a *supervisor*. In **unsupervised learning**, there is no such supervisor and we have only input data. The aim is therefore to find regularities in the input. There is often structure in the input space such that certain patterns occur more often than others. In statistics, this is called **density estimation**.

One method for density estimation is **clustering**, where the aim is to find clusters or groupings of input. In the case of a company with a dataset of past customers, the company may want to see the distribution of the profile of its customers, so that it is clear what kind of customer frequently occurs. Clustering allocates similar customers to the same group. This is called *customer segmentation*. Other uses of clustering include *document clustering*, where documents are subdivided into groups of documents with similar traits.

1.1.5 Reinforcement Learning

In some applications, the output of the system is a sequence of *actions*. In such cases, a policy is more important compared to a single action. A **policy** is the sequence of correct actions to reach the goal. The machine learning program should be able to assess the goodness of policies and learn from past *good* action sequences to be able to generate a policy. Such learning methods are called **reinforcement learning** algorithms.

For example, in playing a game, a single move by itself is not important, but the sequence of right moves is important. A move is good if it is part of a good game playing policy.

1.2 Related Topics for Machine Learning

Machine learning is not an independent or stand-alone field of science. It directly relates to domains such as mathematics, security, or hardware engineering.

- **High-Performance Computing**: The research area of efficient distribution of storage and processing over many computers is becoming increasingly important in ML. For example, training in parallel using subsets of the data, and subsequently, merging the results, or distributing the inference engine in parallel over many processors.

- **Data Privacy and Security**: The data used in ML is subject to questions related to where the data is coming from. Data has now become a valuable commodity because its analysis can lead to valuable results, and as such, care must be paid to its collection, use, and storage. *Privacy-preserving algorithms* is a recent line of research in ML that allows training while at the same time preserving the anonymity of individual instances.

- **Model Interpretability and Trust**: Problems of model sensitivity and model randomness prevent us from being able to interpret and trust the learning algorithms fully. The former problem is reflected by some trained models that lead to big changes in the output due to small changes in the input, i.e., *adversarial examples*. The latter problem is reflected by the fact that training and test sets are random samples, and additionally, the learning algorithm itself may have some randomness to it, which

results in different predictive models based on the same test data. Making a model understandable is the main aim of *explainable artificial intelligence* (XAI).

- **Data Science**: ML and related data analysis methods are often coined with the new field of data science. It is composed of machine learning as its theoretical component, high-performance computing as its engineering component, and a third component about social aspects such as data privacy and security, as well as the ethical and legal implications of making decisions based on data.

1.3 Interpretability of ML models

As stated by Doshi-Velez and Kim [3], a popular fallback in cases where it is not possible to enumerate all unit tests or all confounds of a system is the criterion of **interpretability**. If the system can **explain** its reasoning, we then can verify whether that reasoning is sound with respect to auxiliary criteria.

1.3.1 What is interpretability?

Interpret means *to explain or to present in understandable terms.* In the context of ML systems, **interpretability** can be defined as the ability to explain or to present in understandable terms to a human. There exist many auxiliary criteria that one may wish to optimise. Notions of **fairness** and **unbiasedness** imply that protected groups (explicit or implicit) are not somehow discriminated against. **Privacy** relates to the method to protect sensitive information in the data. Properties such as **reliability** and **robustness** measure the ability of algorithms to reach certain levels of performance in the face of parameters or input variation. **Causality** implies that the predicted change in the output due to perturbations will also occur in the real system. **Usable** methods provide information that assist users to accomplish a task, while **trusted** systems have the confidence of human users. However, merely using explanations is not enough, as explanations may highlight **incompleteness**.

Not all ML systems require interpretability, such as advertisement servers, postal code sorting, air craft collision avoidance systems, because these compute their output *without* human intervention. Explanation is not necessary either because:

1. there are no significant consequences for unacceptable results; or
2. the problem is sufficiently well-studied and validated in real applications that we trust the system's decision, even if the system is not perfect.

The need for interpretability stems from an **incompleteness** in the problem formalisation, which creates a *fundamental barrier* to optimisation and evaluation. There is, however, a distinction between incompleteness and uncertainty. *Uncertainty* is reflected in cases where unknowns result in quantified variance, e.g., trying to learn from small datasets or with limited sensors, whereas *incompleteness* produces some kind of unquantified bias, e.g., the effect of domain knowledge during model selection. In the presence of incompleteness, explanations are a way to ensure that effects of gaps in problem formalisation are visible to us.

1.3.2 Taxonomy of Interpretability Evaluations

Evaluations should match the claimed contribution and should demonstrate *generalisability* via a carefully chosen variety of *synthetic* and *standard* benchmarks. Evaluation approaches include application-grounded, human-grounded, and functionally-grounded evaluations.

- **Application-grounded Evaluation** (real humans, real tasks): These involve conducting human experiments within a real application to show that it works in the intended application context. Expensive, due to the need for domain experts.
- **Human-grounded Metrics** (real humans, simplified tasks): These involve conducting simpler human-subject experiments that maintain the essence of the target application. This is appealing when experiments with the target community is challenging. Less expensive, because the human-subject pool is larger.
- **Functionally-grounded Evaluation** (no humans, proxy tasks): Instead of conducting human experiments, some formal definition of interpretability is used as a proxy for explanation quality. Compared to other two approaches, this is appealing as it requires minimal cost and time both to perform and to get ethical approval. The challenge is to determine what proxies to use, e.g., decision trees as they are generally interpretable.

1.3.3 Latent Dimension of Interpretability

Data-driven approaches to interpretability can use the ideas of matrix factorisation to decompose the evaluation in terms of simpler evaluations. However, to construct such a matrix, its latent factors are required. Disparate-seeming applications may share common factors. Below is a non-exhaustive list of explanations that have commonalities between their tasks:

- **Global vs Local**: Global interpretability implies knowing what patterns are present in general, while local interpretability implies knowing the reasons for a specific decision. The former may be important during scientific understanding or bias detection, whereas the latter is used to justify specific decisions.
- **Area, Severity of Incompleteness**: The types of explanations needed may vary depending on whether the source of concern is due to incompletely specified inputs, constraints, domains, internal model structure, costs, or even in the need to understand the training algorithm.
- **Time Constraints**: A decision that needs to be made at the bedside or during the operation of a plant must be understood quickly, while in scientific or anti-discrimination applications, the end-user may be willing to spend hours trying to fully understand an explanation.
- **Nature of User Expertise**: The nature of the user's own expertise will also influence what level of sophistication they expect in their explanations. For example, domain experts may expect or prefer a somewhat larger and sophisticated model, which confirms facts that they know, over a smaller, more opaque one.

Some important factors to the quality of explanations are the following, where *cognitive chunk* refers to the basic units of an explanation.

- **Form of cognitive chunks**: What are the basic units of the explanation? Are they raw features? etc.
- **Number of cognitive chunks**: How many cognitive chunks does the explanation contain?
- **Level of compositionality**: How are the cognitive chunks organised (rules, hierarchies, other abstractions)?
- **Monotonicity and other interactions between cognitive chunks**: Does it matter that the cognitive chunks are combined in linear or nonlinear ways?
- **Uncertainty and stochasticity**: How well do people understand uncertainty measures?

2 Data Science Methodology

CRISP-DM is a well known data science methodology that describes the life cycle of a data science project[1]. It has six phases:

1. **Business understanding**: this phase focuses on understanding the objectives and requirements of the project. Activities consist of (1) determining business objectives, (2) assess current situation, (3) determine data mining goals, (4) produce project plan.

2. **Data understanding**: this phase puts the focus on identifying, collecting, and analysing the data sets that can help to accomplish the project goals. The tasks within this phase consist of (1) initial data collection, (2) describing the data, (3) exploring the data, (4) verifying data quality.

3. **Data preparation**: rule of thumb, 80% of the project is related to data preparation. In this phase, the data sets are prepared for modelling. It has five tasks: (1) select data based on inclusion/exclusion criteria, (2) clean data by correcting, imputing, or removing erroneous values, (3) construct data by deriving new attributes, (4) integrate data by combining data from multiple sources, (5) format data.

4. **Modelling**: build and assess various models based on several different modelling techniques. This phase consists of the tasks: (1) select modelling techniques, (2) generate test design (training/test/validation sets), (3) build model, (4) assess model by interpreting the model results with domain knowledge.

5. **Evaluation**: whereas task 4 of modelling (assess model) focuses on the technical model assessment, the evaluation phase looks more broadly at which model best meets the *business requirements*. The tasks are as follows: (1) evaluate results, (2) review process by asking whether anything was overlooked, and summarise findings for further correction, (3) determine next steps (proceed to deployment, iterate further, or initiate new project).

6. **Deployment**: a model is not particularly useful unless the customer can access its results. The final phase has four tasks: (1) plan deployment, (2) plan monitoring and maintenance, (3) produce final report, (4) review project (project retrospective).

CRISP-DM does not outline what to do **after** the project, i.e., during *operations*. If the model is going to production, then maintenance is required, i.e., constant monitoring and occasional model tuning.

2.1 Data Preparation and Analysis

Once it is clear what problem has to be solved, the process of *data collection*, *processing*, and *analysis* starts. A first step to data analysis is to explore the data set at hand. For one-dimensional data, the computation of a few summary statistics provides an overview about the data, e.g., size of dataset, minimum, maximum, mean, standard deviation, etc. However, exclusively computing summary statistics can be misleading, as the shape of the

[1]CRISP-DM: `https://www.datascience-pm.com/crisp-dm-2/`

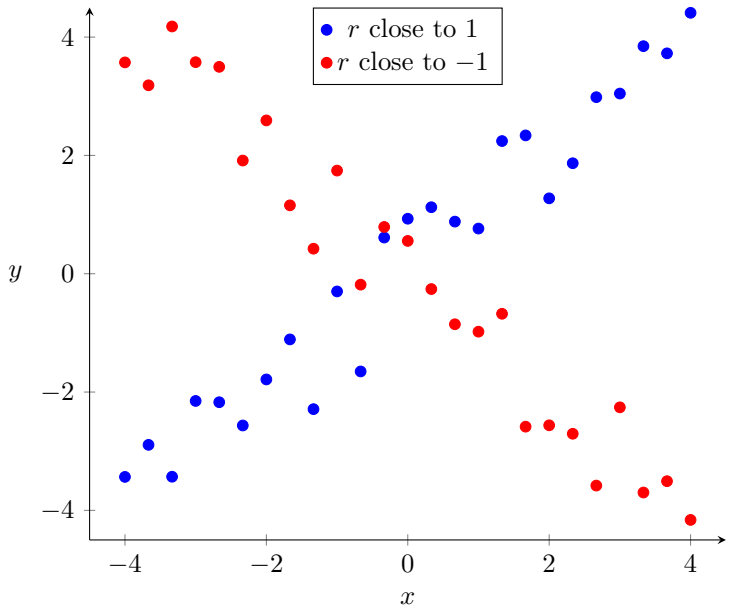


Figure 2.1: Scatter plot of two normally distributed variables Y_1 and Y_2, in which the blue marked data points Y_1 have a strong positive correlation with X, and the red marked data points Y_2 have a strong negative correlation with X.

data in a plot can be drastically different, while maintaining the same summary statistics[2]. Therefore, a good next step is to visualise the data in a *histogram* or *scatter plot*, where the data is grouped into distinctive buckets which indicate the count of data points that fall in each bucket.

For two-dimensional data, individually understanding the variables via a histogram is useful to determine whether they are normally distributed. In addition, it is also useful to understand the strength of the relationship between variables via Pearson's correlation coefficient. The correlation coefficient is defined by r or $\text{Corr}(\mathbf{x}, \mathbf{y})$ as:

$$r = \text{Corr}(\mathbf{x}, \mathbf{y}) = \frac{\sum(x_i - \overline{x})(y_i - \overline{y})}{\sqrt{\sum(x_i - \overline{x})^2 \sum(y_i - \overline{y})^2}}$$

The value r lies in the range $[-1, 1]$. If r is positive and approaching 1, then there is a *strong positive correlation*. If r is negative and approaching -1, then there is a *strong negative correlation*. If r is close to 0, then the correlation is considered *weak*. For instance, consider the scatter plot in figure 2.1 that illustrates a strong positive (blue) and a strong negative (red) correlation.

If the data consists of many variables, i.e., dimensions, we can create a **correlation matrix R** that denotes how all dimensions relate to one another. For an entry in the correlation matrix, row i is the ith dimension and column j is the jth dimension of the data.

[2]Anscombe's quartet: `https://matplotlib.org/stable/gallery/specialty_plots/anscombe.html`

2.1.1 Example: Correlation matrix

As an example, consider a dataset with 3 features, let $\mathbf{x}$ denote the random variable with discrete values in the range [1..7], let $\mathbf{y}_{exp} = e^{\mathbf{x}}$, and let $\mathbf{y}_{square} = \mathbf{x}^2$. The correlation matrix is a 3×3 matrix because we have 3 variables, with ones on the diagonal (correlation between a variable $\mathbf{v}$ and $\mathbf{v}$ is always 1). The order of the columns is $\mathbf{x}$, $\mathbf{y}_{exp}$, $\mathbf{y}_{square}$.

$$\mathbf{R} = \begin{bmatrix} 1 & 0.801 & 0.977 \\ 0.801 & 1 & 0.904 \\ 0.977 & 0.904 & 1 \end{bmatrix}$$

From this matrix, we can read that the correlation between $\mathbf{x}$ and $\mathbf{y}_{square}$ is 0.977 and the correlation between $\mathbf{x}$ and $\mathbf{y}_{exp}$ is 0.801. In other words, for the (small) range of values in $\mathbf{x}$, the squared dimension $\mathbf{y}_{square}$ has a stronger correlation than the exponentiation dimension $\mathbf{y}_{exp}$. Between the $\mathbf{y}$ dimensions, the correlation is 0.904, probably because for small values of $x \in \mathbf{x}$, there is a strong linear relationship between $\mathbf{y}_{exp}$ and $\mathbf{y}_{square}$. However, for large values, both functions will deviate too much, thus reducing the correlation. This conclusion also reinforces the importance of a good and representative (sample) size for the data.

Often, if a strong positive or negative correlation is observed between variables, it provides reasons to believe that there is a dependence upon the variables. Therefore, variables could be reduced based on the dependence assessment. In this case, $\mathbf{x}$ squared is dependent on $\mathbf{x}$ and has a high positive correlation, which means that this data does not provide much additional information.

2.1.2 Rescaling

After exploring the data, the data usually needs to be manipulated so that it is in a usable format. One of the steps in this process is the step of *rescaling* the data. For instance, consider the small dataset of height and weight in table 1.

Table 1: Small dataset example of height (in inches and centimetres) and weight (in pounds).

Person	**Height (inches)**	**Height (centimetres)**	**Weight (pounds)**
A	63	160	150
B	67	170.2	160
C	70	177.8	171

The task is to analyse the data by identifying *clusters* of body sizes, i.e., height with respect to weight. Clusters represent points that are near each other, which means we need a distance metric. Using the Euclidean distance, defined for two data points as

$$\mathcal{D}(x_1, y_1, x_2, y_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

we can compute the distances between the 3 data points. However, doing so results in differences between using *heights in inches* and *heights in centimetres*. Specifically, B's nearest neighbour is A if we measure height in inches, whereas B's nearest neighbour is C if we measure in centimetres instead.

This is a problem if changing *units* also changes the result. Therefore, when dimensions are not compatible due to varying units, we can **rescale** the data, so that each dimension has mean 0 and standard deviation 1. This is also known as **standardisation** (or z-normalisation). The formula is the following:

$$\mathbf{x}_s^t = \frac{\mathbf{x}^t - \mu}{\sigma}$$

Applying this operation to the features in the dataset prior to computing the distance measures results in consistent measures, as all features have the same **measuring units**.

2.2 Dimensionality Reduction

During data preparation, several decision are made about removing missing values, replacing these values, and what to do with outliers. The next step is concerned with determining which variables to include in the subsequent analysis. When the dataset contains many variables, it may be useful to apply automatic methods to **condense** the information in these variables. The process of converting a large set of variables into a smaller set of variables is known as **dimensionality reduction**. Methods such as **Linear Discriminant Analysis** (LDA) and **Principal Component Analysis** (PCA) can be used.

Ideally, the process of *feature selection* or *feature extraction* should not form a separate process, because the classifier or regressor should be able to use whichever features it finds necessary, while discarding the irrelevant features. However, there are several reasons for reducing dimensionality as a separate preprocessing step:

- The complexity of most learning algorithms depends on the input dimensionality (columns), d, as well as the size of the dataset (rows), N. Decreasing d also decreases the complexity of the *inference algorithm* during testing.
- Unnecessary input features are excluded, which saves the cost of extraction.
- Simpler models are more robust on *small datasets*, and thus, have less variance.
- Data can be better explained with fewer features by understanding the process that underlies the data, which allows for *knowledge extraction*. The fewer features can then be interpreted as *hidden* or *latent* factors.
- Reduced dimensionality without loss of information may allow for analysis via visualisations, e.g., to understand structure and detect outliers.

There are two methods for reducing dimensionality. (1) In **feature selection**, we are interested in finding k of the d dimensions that give us the most information, and we discard the other $(d-k)$ dimensions. (2) In **feature extraction**, we are interested in finding a new set k dimensions that are combinations of the original d dimensions. Depending on whether these methods use the output information, they can be supervised or unsupervised.

2.2.1 Subset Selection

In subset selection, we are interested in finding the best **subset** of features that contain the least number of dimensions, while still contributing to the accuracy. Therefore, unimportant

dimensions are discarded. Using a suitable error function (or accuracy function), this can be used in both classification and regression.

The powerset of any feature set with dimensions d is 2^d, i.e., there are 2^d possible subsets to select. Testing every subset is costly (unless d is small). Therefore, **heuristics** are used to get a reasonable, but not optimal, solution in reasonable polynomial time.

Forward Selection In forward selection, we start with no variables in the feature set, and add them one by one, at each step adding the one that decreases the error the most, until any further addition does not decrease the error (or decreases it only slightly). Checking the error should be done on a validation set distinct from the training set, because the goal is to maintain generalisation accuracy, while dropping a subset of features. More features generally result in a lower training error, but not necessarily a lower validation error.

In *sequential forward selection*, we start with an empty feature set, $F = \varnothing$. At each step, for all possible $\mathbf{x}_i \in D$, where D is the set of all features, we train a model on the training set and compute $E(F \cup \mathbf{x}_i)$ on the validation set. Then, we choose the feature $\mathbf{x}_j$ that causes the least error by:

$$j = \arg\min_i E(F \cup \mathbf{x}_i)$$

Then, we add $\mathbf{x}_j$ to F if $E(F \cup \mathbf{x}_j) < E(F)$. In other words, we only add $\mathbf{x}_j$ to F if the error *with* $\mathbf{x}_j$ is smaller than the error *without* $\mathbf{x}_j$. We stop if adding any feature does not decrease E any further. This algorithm is also known as the **wrapper** approach, because the process of *feature selection* "wraps" around the learning algorithm, i.e., the learner is a subroutine of the feature selector.

The process of testing features one by one may be costly, because to decrease the dimensions from d to k, we need to train and test the system $d + (d-1) + (d-2) + \cdots + (d-k)$ times, which has a time complexity of $\mathcal{O}(d^2)$. This is a local greedy search procedure and thus not guaranteed to find the optimal subset, i.e., the minimal subset causing the minimal error. For instance, $\mathbf{x}_i$ and $\mathbf{x}_j$ by themselves might not be good candidates to be included in F, but they might decrease the error if taken together in the feature set. However, because the algorithm is greedy, and adds features one at the time, it may not be able to detect this.

Adding multiple features at once, enabling backtracking, and checking which previously added features can be removed after a current addition are all valid extensions of the algorithm. However, they increase the search space or add more steps to the procedure, and thus complexity increases.

Backward Selection In backward selection, we start with every variable in the feature set, and remove them one by one, at each step removing the one that decreases the error the most (or increases it only slightly), until any further removal increases the error significantly.

Similar to the sequential forward selection, in *sequential backward selection*, we start with F containing all features, except now we remove one feature from F at a time. Specifically, the feature that causes the least error by:

$$j = \arg\min_i E(F - \mathbf{x}_i)$$

Then, we add $\mathbf{x}_j$ to F if $E(F - \mathbf{x}_j) < E(F)$. We stop if removing a feature does not decrease the error any further. All variants for forward selection search are applicable for backward selection search. The complexity is also similar, except that training a system with many features is more costly, because we have to consider most of the features in the first iterations. Therefore, forward selection may be preferable if we expect there to be many useless features.

Subset selection is *supervised*, because outputs are used by the regressor or classifier to calculate the error. Any regressor or classifier can be used. In an application like face recognition, feature selection is not a good method, because individual pixels by themselves do not carry much discriminative information, whereas groups of pixels forming patterns do carry information about face identity.

2.2.2 Principal Component Analysis

Principal Component Analysis (PCA) is an unsupervised method, as it does not use the output information. Rather, the criterion to be maximised is the *variance*. PCA is a projection method that finds a mapping from the inputs in the original d-dimensional space to a new ($k < d$)-dimensional space, with minimum loss of information. By the *first principal component*, we mean a *column* vector $\mathbf{w}_1$, such that after projecting the sample data matrix $\mathbf{X}$ onto $\mathbf{w}_1$, the result is most spread out, that is, the differences between the sample data points become most apparent. Let $\mathbf{x}^t$ denote the tth row vector of the $N \times d$ matrix $\mathbf{X}$. For consistency, we assume that every vector is in column form, so $\mathbf{x}^t$ has shape $d \times 1$.

$$\mathbf{w}_1 = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_d \end{bmatrix} \qquad \mathbf{X} = \begin{bmatrix} — & \mathbf{x}^1 & — \\ — & \mathbf{x}^2 & — \\ & \cdots & \\ — & \mathbf{x}^N & — \end{bmatrix}$$

Intuitively, let $\mathbf{X}$ represent a group of people positioned on a Cartesian grid, and let $\mathbf{w}_1$ represent the vector on to which the camera man will project his camera. The camera man wants to find a vector $\mathbf{w}_1$ such that, after projecting the group's shadow on that vector, every person is visible on $\mathbf{w}_1$ and as spread out as possible. The vector $\mathbf{w}_1$ may represent the picture (because images are also projections of objects onto a plane). Therefore, the camera man needs to find $\mathbf{w}_1$ that maximises the variance between every person. Figure 2.2 illustrates this idea, where the red line can be seen as the picture ($\mathbf{w}_1$) onto which the group's position is projected.

To allow for unique solutions and to make the direction the most important factor, we require $||\mathbf{w}_1|| = 1$, i.e., $\mathbf{w}_1$ is a unit vector, which is equivalent to the condition that $\mathbf{w}_1^T\mathbf{w}_1 = 1$. The projection of $\mathbf{X}$ onto $\mathbf{w}_1$ is $z_1^t = \mathbf{w}_1^T\mathbf{x}^t, \forall t$, where we seek to find $\mathbf{w}_1$ such that the variance $\text{Var}(\mathbf{z}_1)$ is maximised with respect to the constraint that $\mathbf{w}_1^T\mathbf{w}_1 = 1$ is satisfied. Note that we can also project the entire training set onto $\mathbf{w}_1$ via

$$\mathbf{z}_1 = \mathbf{X}\mathbf{w}_1$$

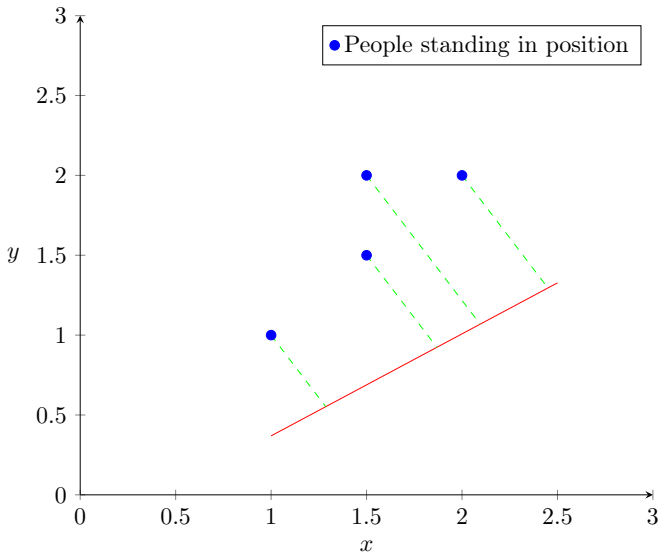


Figure 2.2: Intuitive explanation of PCA. Blue dots are people standing in space, and a camera man wants to take a picture such that everyone is as visible as possible, i.e., spread out as possible. The red line illustrates an example picture where all blue dots are projected onto this line such that the variance of the dots on the line is maximised.

Definition 2.1. Useful variance identities

The standard variance formula for a random variable X is:

$$\mathrm{Var}(X) = \frac{\sum_{i=1}^{N}(x_i - \overline{x})^2}{N-1}$$

Alternatively, for any random variable X, the variance of X is the **expected value** of the **squared difference** between X and its expected value:

$$\mathrm{Var}(X) = E[(X - E[X])^2] = E[X^2] - (E[X])^2$$

Since variance measures the spread of one variable along one dimension, it cannot measure the joint variability of two variables along two dimensions. For this, the **covariance** is defined:

$$\mathrm{Cov}(X,Y) = \frac{\sum_{i=1}^{N}(x_i - \overline{x})(y_i - \overline{y})}{N-1}$$

If the *greater* (resp. *lesser*) values of X correspond with the *greater* (resp. *lesser*) values of Y, then the covariance is **positive**. Conversely, if the *greater* (resp. *lesser*) values of X correspond with the *lesser* (resp. *greater*) values of Y, then the covariance is **negative**. Therefore, the covariance provides information about the strength of the linear relationship between X and Y, and can thus be used as an alternative way to formulate *Pearson's correlation coefficient*:

$$\mathrm{Corr}(X,Y) = \frac{\mathrm{Cov}(X,Y)}{\sigma_X \sigma_Y}$$

Starting with the derivation of the first principal component, let $\mathbf{\Sigma} = \text{Cov}(\mathbf{X})$ denote the covariance matrix of $\mathbf{X}$. Furthermore, we abuse notation here to denote $\mathbf{x}$ as being one or more column vectors of size $d \times N$, i.e., $\mathbf{X}^T$. It can then be shown via the properties of the expected value[3] that

$$\begin{aligned}
\text{Var}(z_1) = \text{Var}(\mathbf{w}_1^T\mathbf{x}) &= E[(\mathbf{w}_1^T\mathbf{x} - E[\mathbf{w}_1^T\mathbf{x}])^2] && (2.1)\\
&= E[(\mathbf{w}_1^T\mathbf{x} - \mathbf{w}_1^T E[\mathbf{x}])^2] && (2.2)\\
&= E[(\mathbf{w}_1^T\mathbf{x} - \mathbf{w}_1^T\boldsymbol{\mu})^2] && (2.3)\\
&= E[(\mathbf{w}_1^T\mathbf{x} - \mathbf{w}_1^T\boldsymbol{\mu})(\mathbf{w}_1^T\mathbf{x} - \mathbf{w}_1^T\boldsymbol{\mu})] && (2.4)\\
&= E[\mathbf{w}_1^T(\mathbf{x} - \boldsymbol{\mu})\mathbf{w}_1^T(\mathbf{x} - \boldsymbol{\mu})] && (2.5)\\
&= E[\mathbf{w}_1^T(\mathbf{x} - \boldsymbol{\mu})(\mathbf{x} - \boldsymbol{\mu})^T\mathbf{w}_1] && (2.6)\\
&= \mathbf{w}_1^T E[(\mathbf{x} - \boldsymbol{\mu})(\mathbf{x} - \boldsymbol{\mu})^T]\mathbf{w}_1 = \mathbf{w}_1^T\mathbf{\Sigma}\mathbf{w}_1 && (2.7)
\end{aligned}$$

Some notes on the above derivation: in 2.1, the alternative variance identity is used to work with expected values (and utilise its properties). In 2.2, the scaling property $E[AX] = AE[X]$ is used. In 2.6, we know that $\mathbf{X}^T$ has dimensions $d \times N$ and $\mathbf{w}_1$ is a projection vector, which means it has to have dimensions $d \times 1$. Thus, the property $\mathbf{v}^T\mathbf{B} = (\mathbf{B}^T\mathbf{v})^T$ holds. This works out, because multiplying $\mathbf{X}$ to the left of this yields a matrix with compatible dimensions. And in 2.7, again using the scaling property of the expected value, the $\mathbf{w}_1$ vectors can be factorised out of $E[\cdot]$. Moreover, in 2.7, it can be shown[4] that the expected value of the **outer product** of $\mathbf{X}^T - E[\mathbf{X}^T]$ with itself is the same as the covariance matrix of $\mathbf{X}^T$, i.e., $\text{Cov}(\mathbf{X}^T) = \mathbf{\Sigma}$. Recall that the inner product (dot product) is A^TA, whereas the outer product is AA^T. Thus, the derivation leads to the following.

$$\text{Var}(z_1) = \mathbf{w}_1^T\mathbf{\Sigma}\mathbf{w}_1$$

To maximise this expression with respect to the unit vector constraint $\mathbf{w}_1^T\mathbf{w}_1 = 1$, a Lagrange multiplier α can be introduced, and the optimisation problem becomes:

$$\max_{\mathbf{w}_1} \mathbf{w}_1^T\mathbf{\Sigma}\mathbf{w}_1 - \alpha(\mathbf{w}_1^T\mathbf{w}_1 - 1)$$

To solve this problem, we take the derivative with respect to $\mathbf{w}_1$ and set the result equal to 0. Using the following vector matrix derivative rules [4]

$$\begin{aligned}
\frac{\partial}{\partial\mathbf{x}}\mathbf{x}^T\mathbf{B}\mathbf{x} &\implies 2\mathbf{B}\mathbf{x} && (2.1)\\
\frac{\partial}{\partial\mathbf{x}}\mathbf{x}^T\mathbf{x} &\implies 2\mathbf{x} && (2.2)\\
\frac{\partial}{\partial\mathbf{x}}\mathbf{x}^T\mathbf{b} &\implies \mathbf{b} && (2.3)
\end{aligned}$$

[3] Properties of expected value: `https://www.statlect.com/fundamentals-of-probability/expected-value-properties`

[4] Covariance and matrix properties: `https://www.randomservices.org/random/expect/Matrices.html`

the derivative of the Lagrange optimisation problem becomes:

$$\frac{\partial}{\partial \mathbf{w}_1} \mathbf{w}_1^T \mathbf{\Sigma} \mathbf{w}_1 - \alpha \mathbf{w}_1^T \mathbf{w}_1 - \alpha \implies 2\mathbf{\Sigma}\mathbf{w}_1 - 2\alpha \mathbf{w}_1$$

And setting it to 0 to find the extrema (maximum in this case) yields:

$$\begin{aligned} 2\mathbf{\Sigma}\mathbf{w}_1 - 2\alpha \mathbf{w}_1 &= 0 \\ 2\mathbf{\Sigma}\mathbf{w}_1 &= 2\alpha \mathbf{w}_1 \\ \mathbf{\Sigma}\mathbf{w}_1 &= \alpha \mathbf{w}_1 \end{aligned}$$

Interestingly, the final expression $\mathbf{\Sigma}\mathbf{w}_1 = \alpha \mathbf{w}_1$ holds if $\mathbf{w}_1$ is the eigenvector of $\mathbf{\Sigma}$, and α is the corresponding eigenvalue. Pre-multiplying $\mathbf{w}_1^T$ in the equation yields $\mathbf{w}_1^T \mathbf{\Sigma} \mathbf{w}_1 = \alpha \mathbf{w}_1^T \mathbf{w}_1$, where it holds that this reduces to α because $\mathbf{w}_1^T \mathbf{w}_1 = 1$, i.e., we get the expression $\mathbf{w}_1^T \mathbf{\Sigma} \mathbf{w}_1 = \alpha$. Therefore, to maximise the variance, i.e., $\mathbf{w}_1^T \mathbf{\Sigma} \mathbf{w}_1$, we choose the eigenvector with the **largest** eigenvalue α.

The principal component is then the *eigenvector* of the covariance matrix $\text{Cov}(\mathbf{X}^T)$ corresponding to the largest *eigenvalue*, $\lambda_1 = \alpha$.

The second principal component, $\mathbf{w}_2$, should also maximise variance, be of unit length, and additionally, be *orthogonal* to $\mathbf{w}_1$. This latter requirement about orthogonality ensures that after the projection $z_2 = \mathbf{w}_2^T \mathbf{x}$, it is **uncorrelated** with z_1. In other words, z_1 and z_2 are perpendicular. This is an additional constraint formulated as $\mathbf{w}_2^T \mathbf{w}_1 = 0$, because this is equivalent to the *vector dot product*, and orthogonality is realised when the dot product is 0. The optimisation problem now becomes:

$$\max_{\mathbf{w}_2} \mathbf{w}_2^T \mathbf{\Sigma} \mathbf{w}_2 - \alpha(\mathbf{w}_2^T \mathbf{w}_2 - 1) - \beta(\mathbf{w}_2^T \mathbf{w}_1 - 0)$$

Using the vector matrix derivative rules and setting the result equal to zero, the derivative with respect to $\mathbf{w}_2$ becomes:

$$2\mathbf{\Sigma}\mathbf{w}_2 - 2\alpha \mathbf{w}_2 - \beta \mathbf{w}_1 = 0$$

This results in $\mathbf{\Sigma}\mathbf{w}_2 = \alpha \mathbf{w}_2$, which implies that $\mathbf{w}_2$ should be the eigenvector of $\mathbf{\Sigma}$ with the *second largest eigenvalue*, $\lambda_2 = \alpha$. Similarly, this holds for the other dimensions as well.

To conclude the PCA procedure, note that the k eigenvectors with nonzero eigenvalues are the dimensions of the new reduced space. The first eigenvector, $\mathbf{w}_1$, also denoted as the first principal component, explains the largest proportion of the variance; the second principal component, $\mathbf{w}_2$, explains the second largest proportion; and so on. Thus, PCA can be defined by first centering the data matrix $\mathbf{X}$ at the origin to have mean 0, and then projecting it onto $\mathbf{W}$

$$\mathbf{z} = \mathbf{W}^T(\mathbf{x} - \mathbf{m})$$

where the k columns of $\mathbf{W}$ are the k leading *eigenvectors* of the covariance matrix of $\mathbf{X}^T$.

$$\mathbf{W} = \begin{bmatrix} | & | & & | \\ \mathbf{w}_1 & \mathbf{w}_2 & \dots & \mathbf{w}_k \\ | & | & & | \end{bmatrix}$$

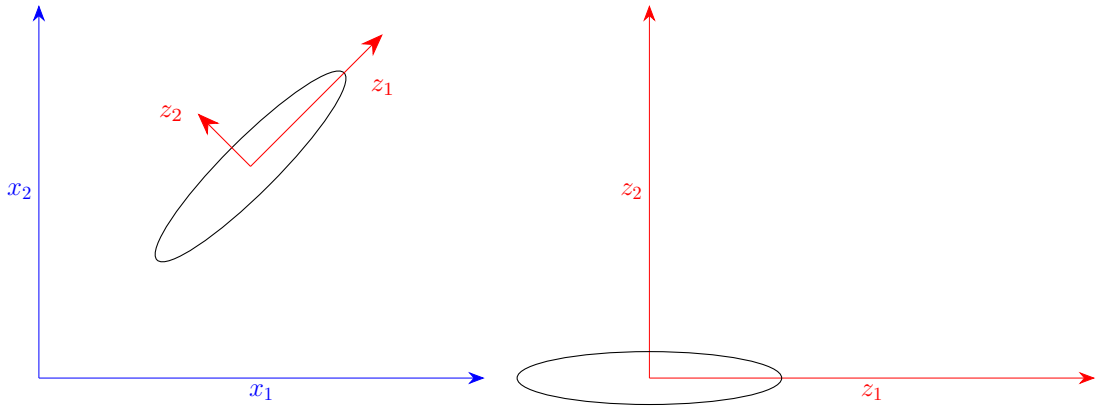


Figure 2.3: Visualisation of the PCA procedure. z_1 and z_2 represent the new axes after centering the data and projection.

In practice, we can also project the entire data matrix $\mathbf{X}$ in one go with the expression:

$$\mathbf{Z} = (\mathbf{X} - \mathbf{m})\mathbf{W}$$

where $\mathbf{m}$ is the $d \times 1$ sample mean vector that we first broadcast to match the $N \times d$ shape of $\mathbf{X}$.

On another note, even if all eigenvalues are greater than 0, some eigenvalues may contribute little to the variance, and can therefore, be discarded. We take into account only the leading k principal components that *cumulatively* explain more than, say 90% of the variance. When the eigenvalues λ_i are sorted in descending order, the *proportion of variance* explained by the k principal components is

$$\text{Proportion of Variance} = \frac{\lambda_1 + \lambda_2 + \cdots + \lambda_k}{\lambda_1 + \lambda_2 + \cdots + \lambda_k + \cdots + \lambda_d}$$

In general, if the dimensions of the original data matrix are *highly correlated*, then there will be a small number of eigenvectors with large eigenvalues, which results in k being much smaller than d. In other words, large dimensionality reduction will be attained. In contrast, if the dimensions of the data matrix are not correlated, then k will be as large as d, and PCA will not be of any benefit for dimensionality reduction. Figure 2.3 illustrates the procedure.

If the variances of the dimensions $\mathbf{x}_i$ of $\mathbf{X}, i = 1, \ldots, d$ vary considerably, they affect the direction of the principal components more than the correlations. For example, due to different units of measurements in the variables of the dataset. Therefore, before applying PCA, a common procedure is to preprocess the data so that each dimension has mean 0 and unit variance (z-normalisation). Alternatively, instead of the eigenvectors of the covariance matrix $\mathbf{\Sigma}$, one could also use the eigenvectors of the correlation matrix $\mathbf{R}$.

Given a d-dimensional data matrix indexed by its instances (records) with t and projected onto k-dimensions via $z^t = \mathbf{W}^T(\mathbf{x}^t - \mathbf{m})$, we can also **backproject** this specific instance to the original d-dimensional space via

$$\hat{\mathbf{x}}^t = \mathbf{W}z^t + \mathbf{m}$$

$\hat{\mathbf{x}}^t$ is the **reconstruction** of $\mathbf{x}^t$ from its representation in the z-space.

Definition 2.2. Reconstruction Error
The reconstruction error is the distance between the instance $\mathbf{x}^t$ and the reconstruction $\hat{\mathbf{x}}^t$ from the lower-dimensional space:

$$\sum_t ||\mathbf{x}^t - \hat{\mathbf{x}}^t||^2$$

If during dimensionality reduction, we discard some eigenvectors with nonzero eigenvalues, then there is *reconstruction error*, and its magnitude will depend on the proportion of the discarded eigenvalues.

2.2.3 Feature Embedding

Recall that the dot product between two vectors measures the degree of similarity. If the dot product is zero, then the vectors are orthogonal (or perpendicular). If the dot product is not zero, it tells us how much one vector is in the same direction as another vector. For two vectors $\mathbf{v}$ and $\mathbf{w}$ the dot product is defined as $\mathbf{v} \cdot \mathbf{w}$. In matrix algebra, it can be defined by $\mathbf{v}^T\mathbf{w}$. Using this idea, the covariance matrix of an $N \times d$ data matrix $\mathbf{X}$ is the $d \times d$ matrix equal to

$$\text{Cov}(\mathbf{X}) = \frac{\mathbf{X}^T\mathbf{X}}{N-1}$$

if $\mathbf{X}$ is centered to have zero mean. Principal component analysis uses the eigenvectors of $\mathbf{X}^T\mathbf{X}$, because these are proportional to the eigenvectors of $\mathbf{\Sigma}$, i.e., $\mathbf{\Sigma} \propto \mathbf{X}^T\mathbf{X}$.

A useful identity of $\mathbf{X}^T\mathbf{X}$ is defined by its **spectral decomposition**

$$\mathbf{X}^T\mathbf{X} = \mathbf{W}\mathbf{D}\mathbf{W}^T$$

where $\mathbf{W}$ is $d \times d$ and contains the eigenvectors of $\mathbf{X}^T\mathbf{X}$ and $\mathbf{D}$ is a $d \times d$ diagonal matrix with the corresponding eigenvalues. Furthermore, we assume that the eigenvectors are sorted according to their eigenvalues such that the first column of $\mathbf{W}$ is the eigenvector with the largest eigenvalue in $\mathbf{D}_{ii}$. Figure 2.4 illustrates the spectral decomposition of $\mathbf{X}^T\mathbf{X}$.

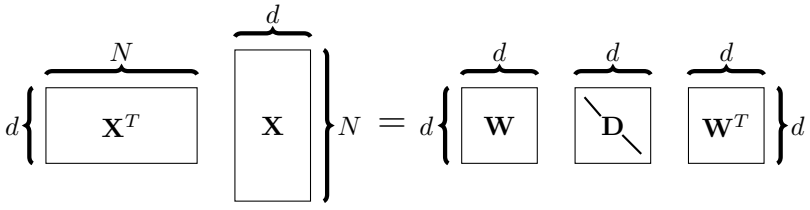


Figure 2.4: Visualisation of the spectral decomposition of $\mathbf{X}^T\mathbf{X}$

Alternatively, it can be derived from $\mathbf{X}^T\mathbf{X} = \mathbf{W}\mathbf{D}\mathbf{W}^T$ that the spectral decomposition of $\mathbf{X}\mathbf{X}^T$ is

$$\mathbf{X}\mathbf{X}^T = \mathbf{V}\mathbf{E}\mathbf{V}^T$$

where $\mathbf{V}$ is the $N \times N$ matrix containing the eigenvectors of $\mathbf{X}\mathbf{X}^T$ in its columns, and

$\mathbf{E}$ is the $N \times N$ diagonal matrix with the corresponding eigenvalues. The N-dimensional eigenvectors of $\mathbf{X}\mathbf{X}^T$ are the coordinates in the new space. Deriving these new coordinates is called **feature embedding**. See figure 2.5 for a similar visual decomposition of the dimensions.

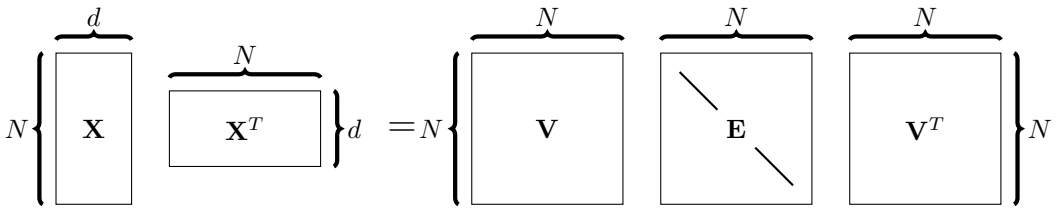


Figure 2.5: Visualisation of the spectral decomposition of $\mathbf{X}\mathbf{X}^T$

The element (i, j) of $\mathbf{X}\mathbf{X}^T$ is equal to the dot product of the instances (records) i and j, that is, $(\mathbf{x}^i)^T(\mathbf{x}^j)$, where $i, j = 1, \ldots, N$. Since the dot product (among other interpretations) measures similarity, we can consider $\mathbf{X}\mathbf{X}^T$ as an $N \times N$ matrix of **pairwise similarities**. The idea of **feature embedding** is then to place the instances in a k-dimensional space such that pairwise similarities in the new space respect the original pairwise similarities.

- Use **PCA** when $d < N$, i.e., use the identity of $\mathbf{X}^T\mathbf{X}$.
- Use **Feature Embedding** when $d > N$, i.e., use the identity of $\mathbf{X}\mathbf{X}^T$.

The caveat is that PCA learns *projection vectors* by which new data can be mapped to the projected (lower) dimensional space. This is not possible in feature embedding, because there are no projection vectors. Rather, the coordinates are derived directly from the data. When new data arrives, it should be added to $\mathbf{X}$ and the calculation should be done again.

For example, consider a dataset of 40 face images (four images per person), each image is $256 \times 256 = 65536$-dimensional. Thus the data matrix has the shape 40×65536. Feature embedding would allow us to reduce the dimensionality to a 40×40 matrix of pairwise similarities of face images.

2.2.4 Linear Discriminant Analysis

Linear discriminant analysis is a *supervised method* for dimensionality reduction, specifically, for classification problems. In the binary classification case, there are two classes, denoted by two samples C_1 and C_2, where we want to find the direction, as given by $\mathbf{w}$, such that when the data is projected onto $\mathbf{w}$, the examples from the two classes are as well separated as possible. For PCA, we saw that

$$z = \mathbf{w}^T\mathbf{x}$$

is the projection of the vector $\mathbf{x}$ onto the vector $\mathbf{w}$, and thus is a dimensionality reduction from d to 1.

Let $\mathbf{m}_1 \in \mathbb{R}^d$ be the mean of the samples from class C_1, and let $m_1 \in \mathbb{R}$ be the mean of the sample after projection from d dimensions to 1 dimension. Likewise for $\mathbf{m}_2$ and m_2 with respect to class C_2. The sample is $\mathcal{X} = \{\mathbf{x}^t, r^t\}$, $t = 1, \ldots, N$ such that $r^t = 1$ if $\mathbf{x}^t \in C_1$ and $r^t = 0$ if $\mathbf{x}^t \in C_2$.

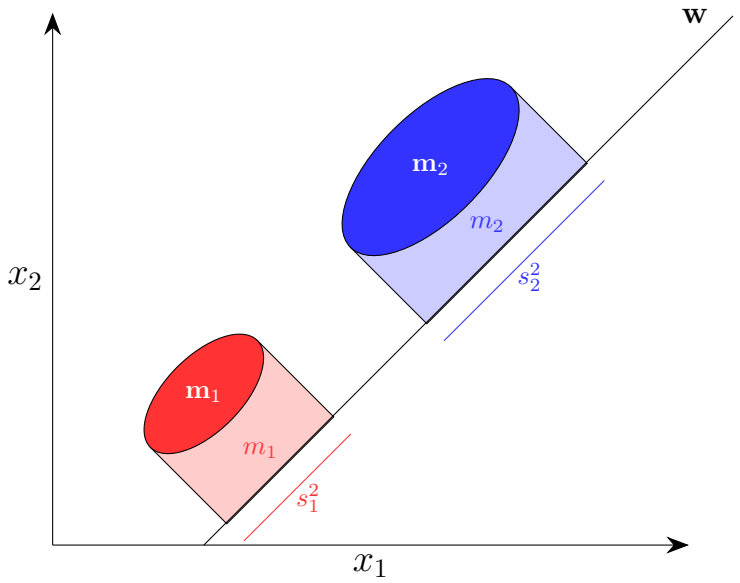


Figure 2.6: Visualisation of LDA, where there are two features x_1, x_2 on the axes, and two classes (blue/red). Each class is projected onto the single dimension such that the distance between $\mathbf{m}_1$ and $\mathbf{m}_2$ is maximised and the total scatter $s_1^2 + s_2^2$ is minimised.

The mean after projection can be found via the following identities. Here, it is essential to see that $\mathbf{w}^T\mathbf{m}_n, n = 1, 2$ is an easy way to find m_n, i.e., projecting the vector of feature means is the same as projecting the data and then taking the mean.

$$m_1 = \frac{\sum_t \mathbf{w}^T\mathbf{x}^t r^t}{\sum_t r^t} = \mathbf{w}^T\mathbf{m}_1$$
$$m_2 = \frac{\sum_t \mathbf{w}^T\mathbf{x}^t (1 - r^t)}{\sum_t (1 - r^t)} = \mathbf{w}^T\mathbf{m}_2$$

The amount of **scatter** (variability) of samples from C_1 and C_2 after projection is:

$$s_1^2 = \sum_t (\mathbf{w}^T\mathbf{x}^t - m_1)^2 r^t$$
$$s_2^2 = \sum_t (\mathbf{w}^T\mathbf{x}^t - m_2)^2 (1 - r^t)$$

The idea is now that after projection, for the two classes to be well separated, we want the means to be as far as possible and the examples of classes to be scattered in as small region as possible. In other words, we want to find the projection vector $\mathbf{w}$ such that the variability **between the classes** is *maximised* and the variability **within the classes** is *minimised*.

Therefore, $|m_1 - m_2|$ needs to be large and $s_1^2 + s_2^2$ needs to be small. Figure 2.6 illustrates this idea. **Fisher's linear discriminant** is $\mathbf{w}$ that maximises

$$J(\mathbf{w}) = \frac{(m_1 - m_2)^2}{s_1^2 + s_2^2}$$

The numerator can be rewritten to get a matrix product in the following way:

$$\begin{aligned}(m_1 - m_2)^2 &= (\mathbf{w}^T\mathbf{m}_1 - \mathbf{w}^T\mathbf{m}_2)^2 \\ &= (\mathbf{w}^T\mathbf{m}_1 - \mathbf{w}^T\mathbf{m}_2)(\mathbf{w}^T\mathbf{m}_1 - \mathbf{w}^T\mathbf{m}_2) \\ &= \mathbf{w}^T(\mathbf{m}_1 - \mathbf{m}_2)\mathbf{w}^T(\mathbf{m}_1 - \mathbf{m}_2) \\ &= \mathbf{w}^T(\mathbf{m}_1 - \mathbf{m}_2)(\mathbf{m}_1 - \mathbf{m}_2)^T\mathbf{w} \\ &= \mathbf{w}^T\mathbf{S}_B\mathbf{w}\end{aligned}$$

where $\mathbf{S}_B = (\mathbf{m}_1 - \mathbf{m}_2)(\mathbf{m}_1 - \mathbf{m}_2)^T$ is the **between-class scatter matrix**. The denominator is the sum of the scatter of examples from a class around the mean after projection. It can be rewritten as:

$$\begin{aligned}s_1^2 &= \sum_t (\mathbf{w}^T\mathbf{x}^t - m_1)^2 r^t \\ &= \sum_t (\mathbf{w}^T\mathbf{x}^t - m_1)(\mathbf{w}^T\mathbf{x}^t - m_1) r^t \\ &= \sum_t (\mathbf{w}^T\mathbf{x}^t - \mathbf{w}^T\mathbf{m}_1)(\mathbf{w}^T\mathbf{x}^t - \mathbf{w}^T\mathbf{m}_1) r^t \\ &= \sum_t \mathbf{w}^T(\mathbf{x}^t - \mathbf{m}_1)\mathbf{w}^T(\mathbf{x}^t - \mathbf{m}_1) r^t \\ &= \sum_t \mathbf{w}^T(\mathbf{x}^t - \mathbf{m}_1)(\mathbf{x}^t - \mathbf{m}_1)^T\mathbf{w} r^t \\ &= \mathbf{w}^T \sum_t (r^t(\mathbf{x}^t - \mathbf{m}_1)(\mathbf{x}^t - \mathbf{m}_1)^T)\mathbf{w} \\ &= \mathbf{w}^T\mathbf{S}_1\mathbf{w}\end{aligned}$$

where

$$\mathbf{S}_1 = \sum_t r^t(\mathbf{x}^t - \mathbf{m}_1)(\mathbf{x}^t - \mathbf{m}_1)^T$$

is the **within-class scatter matrix** for C_1. This scatter matrix can be used to estimate the covariance matrix via

$$\mathbf{\Sigma}_1 = \frac{\mathbf{S}_1}{\sum_t r^t}.$$

Similarly, $s_2^2 = \mathbf{w}^T\mathbf{S}_2\mathbf{w}$, where $\mathbf{S}_2 = \sum_t (1 - r^t)(\mathbf{x}^t - \mathbf{m}_2)(\mathbf{x}^t - \mathbf{m}_2)^T$. Using both aforementioned derivations, we get the following simplification:

$$s_1^2 + s_2^2 = \mathbf{w}^T\mathbf{S}_W\mathbf{w}$$

where $\mathbf{S}_W = \mathbf{S}_1 + \mathbf{S}_2$ is the total within-class scatter matrix. Using these simplifications, the maximisation statement $J(\mathbf{w})$ can be formulated as:

$$J(\mathbf{w}) = \frac{\mathbf{w}^T\mathbf{S}_B\mathbf{w}}{\mathbf{w}^T\mathbf{S}_W\mathbf{w}}$$

Taking the derivative of this expression, setting it equal to 0, and solving for $\mathbf{w}$, we find

$$\mathbf{w} = c\, \mathbf{S}_W^{-1}(\mathbf{m}_1 - \mathbf{m}_2)$$

where c is some constant. Fortunately, we are only interested in the direction of the projection, and not the magnitude. Therefore, we can set c equal to 1, and find $\mathbf{w}$.

Although Fisher's discriminant analysis works best on normally distributed classes, it can also be used when the classes are not normally distributed. Furthermore, after projecting from d to 1 dimension, any classification method can be used. The projection vector (direction) of LDA is superior to that of PCA in terms of ease of discrimination. Furthermore, we see that $\mathbf{S}_W$ should he invertible. If this is not the case, then first use PCA to get rid of the singularity and then apply LDA to its result.

2.2.5 Example: Linear Discriminant Analysis

Consider the following data frame in table 2, where the $\mathbf{X}$ columns are features and the $\mathbf{y}$ column denotes the binary (0/1) labels for classes (C_1/C_2) respectively.

Table 2: Dataset of features $\mathbf{x}_1$, $\mathbf{x}_2$, and binary labels y.

	$\mathbf{X}$		
$\mathbf{t}$	$\mathbf{x}_1$	$\mathbf{x}_2$	$\mathbf{y}$
1	1	1	0
2	2	3	1
3	3	4	1
4	2	1	0

We start with indexing $\mathbf{X}$ by t such that each entry is a pair $\langle (x_1, x_2), y \rangle$. To compute $\mathbf{w}$, we first compute $\mathbf{m}_1$ and $\mathbf{m}_2$, where $\mathbf{m}_1$ is the mean of class C_1 (label 0) and $\mathbf{m}_2$ is the mean of class C_2 (label 1):

$$\mathbf{m}_1 = (1.5, 1)^T$$
$$\mathbf{m}_2 = (2.5, 3.5)^T$$

Then, we compute $\mathbf{S}_1$ and $\mathbf{S}_2$:

$$\begin{aligned}
\mathbf{S}_1 &= 1 \times (\mathbf{x}^{t1} - \mathbf{m}_1)(\mathbf{x}^{t1} - \mathbf{m}_1)^T + 1 \times (\mathbf{x}^{t4} - \mathbf{m}_1)(\mathbf{x}^{t4} - \mathbf{m}_1)^T \\
&= 1 \times \left(\begin{bmatrix} 1 \\ 1 \end{bmatrix} - \begin{bmatrix} 1.5 \\ 1 \end{bmatrix} \right) \left(\begin{bmatrix} 1 \\ 1 \end{bmatrix} - \begin{bmatrix} 1.5 \\ 1 \end{bmatrix} \right)^T + 1 \times \left(\begin{bmatrix} 2 \\ 1 \end{bmatrix} - \begin{bmatrix} 1.5 \\ 1 \end{bmatrix} \right) \left(\begin{bmatrix} 2 \\ 1 \end{bmatrix} - \begin{bmatrix} 1.5 \\ 1 \end{bmatrix} \right)^T \\
&= \begin{bmatrix} -0.5 \\ 0 \end{bmatrix} \begin{bmatrix} -0.5 & 0 \end{bmatrix} + \begin{bmatrix} 0.5 \\ 0 \end{bmatrix} \begin{bmatrix} 0.5 & 0 \end{bmatrix} = \begin{bmatrix} 0.5 & 0 \\ 0 & 0 \end{bmatrix} \\
\mathbf{S}_2 &= 1 \times (\mathbf{x}^{t2} - \mathbf{m}_2)(\mathbf{x}^{t2} - \mathbf{m}_2)^T + 1 \times (\mathbf{x}^{t3} - \mathbf{m}_2)(\mathbf{x}^{t3} - \mathbf{m}_2)^T \\
&= 1 \times \left(\begin{bmatrix} 2 \\ 3 \end{bmatrix} - \begin{bmatrix} 2.5 \\ 3.5 \end{bmatrix} \right) \left(\begin{bmatrix} 2 \\ 3 \end{bmatrix} - \begin{bmatrix} 2.5 \\ 3.5 \end{bmatrix} \right)^T + 1 \times \left(\begin{bmatrix} 3 \\ 4 \end{bmatrix} - \begin{bmatrix} 2.5 \\ 3.5 \end{bmatrix} \right) \left(\begin{bmatrix} 3 \\ 4 \end{bmatrix} - \begin{bmatrix} 2.5 \\ 3.5 \end{bmatrix} \right)^T \\
&= \begin{bmatrix} -0.5 \\ -0.5 \end{bmatrix} \begin{bmatrix} -0.5 & -0.5 \end{bmatrix} + \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} \begin{bmatrix} 0.5 & 0.5 \end{bmatrix} = \begin{bmatrix} 0.5 & 0.5 \\ 0.5 & 0.5 \end{bmatrix}
\end{aligned}$$

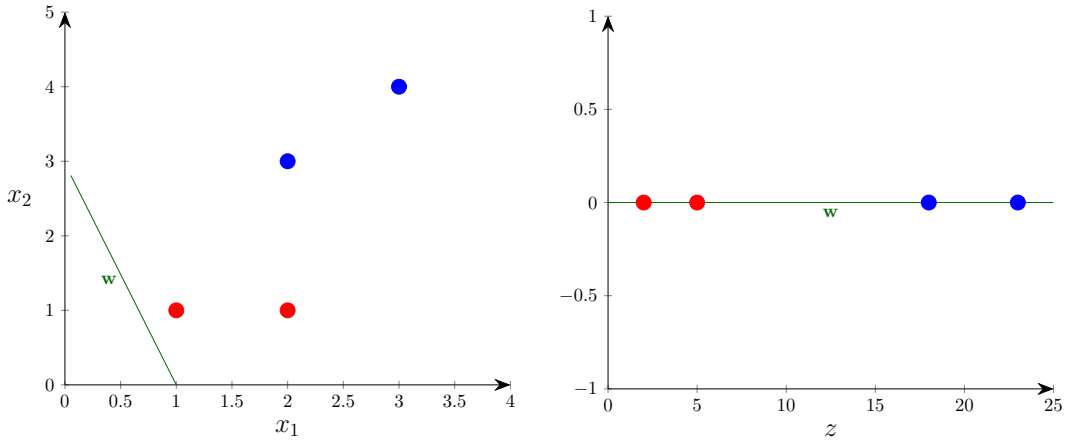


Figure 2.7: Applied LDA on scatter plot of data. Left plot shows the projection line, right plot shows the result after projection.

Combining all ingredients, we get the following calculation for $\mathbf{w}$:

$$\mathbf{w} = \left(\begin{bmatrix} 0.5 & 0 \\ 0 & 0 \end{bmatrix} + \begin{bmatrix} 0.5 & 0.5 \\ 0.5 & 0.5 \end{bmatrix} \right)^{-1} \left(\begin{bmatrix} 1.5 \\ 1 \end{bmatrix} - \begin{bmatrix} 2.5 \\ 3.5 \end{bmatrix} \right) = \begin{bmatrix} -3 \\ 8 \end{bmatrix}$$

We can now project the data onto $\mathbf{w}$ to reduce dimensionality from 2 to 1 such that the difference between class means is as large as possible, and the total variance (scatter) of samples is as small as possible.

$$z^t = \mathbf{w}^T \mathbf{x}^t, \forall t$$

$$\mathbf{z} = \mathbf{X}\mathbf{w} = \begin{bmatrix} 1 & 1 \\ 2 & 3 \\ 3 & 4 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} -3 \\ 8 \end{bmatrix} = \begin{bmatrix} 5 \\ 18 \\ 23 \\ 2 \end{bmatrix}$$

Figure 2.7 illustrates this procedure geometrically via the green line that denotes the projection vector $\mathbf{w}$.

2.3 Overfitting and Underfitting

Learning can be seen as the search problem within a particular **hypothesis class** $\mathcal{H}$ where we want to find an hypothesis $h \in \mathcal{H}$ that is consistent with the training data. The set of assumptions we make so that learning is possible is referred to as the **inductive bias** of the learning algorithm. One way to introduce inductive bias is when we assume a hypothesis class $\mathcal{H}$.

When the data by itself is not sufficient to find a **unique solution** to a learning problem, the problem is classified as an **ill-posed problem**. In classification and regression, as we see more training examples, we know more about the underlying function, and we reject more hypotheses that are inconsistent from the hypothesis class. The problem is ill-posed if, after learning the training examples, there are still many consistent hypotheses that can be considered as valid. This is the reason why we introduce an inductive bias, so that the

additional assumptions can help to find unique solutions.

The question is now how to choose the right bias. This is called **model selection**, which means that we choose between possible hypothesis classes $\mathcal{H}$. The aim of machine learning is rarely to learn a function that replicates the training data. Rather, the prediction of new unseen examples is more useful. This is called **generalisation**.

For the right amount of generalisation, we should match the *complexity* of the hypothesis class $\mathcal{H}$ with the *complexity* of the function underlying the data. If $\mathcal{H}$ is less complex than the function, we have **underfitting**. For example, if $\mathcal{H}$ represents the class of linear regression models, and we try to fit a model of this class to data sampled from a third-order polynomial, then the linear regression model will not be able to fit the function, i.e., training error will be high. As we increase the complexity of the hypothesis class, the model will fit the data better, thus decreasing the training error. However, if we use $\mathcal{H}$ that is too complex, then the data is not enough to constrain the learning potential of the hypothesis $h \in \mathcal{H}$. For instance, a sixth-order polynomial hypothesis will also learn the noise in noisy data sampled from a third-order polynomial. This is called **overfitting**.

- **Overfitting**: model is too specific, implies **high variance**, i.e., high sensitivity to fluctuations from the training set.
- **Underfitting**: model is too general, implies **high bias**, i.e., too many erroneous inductive bias assumptions.

A zeroth-order polynomial makes too many assumptions (high bias), but gives pretty similar results for different training sets (low variance). On the other hand, a ninth-order polynomial fits the dataset perfectly. It has very low bias, but very high variance, because any two training sets will likely result in very different models.

Definition 2.3. Triple trade-off
In all learning algorithms that are trained from example data, there is a trade-off between three factors:

- The complexity of the hypothesis we fit to the data, i.e., the capacity of $\mathcal{H}$.
- The amount of training data.
- The generalisation error on new examples.

If the training data increases, then the generalisation error decreases. If the complexity of $\mathcal{H}$ increases, then the generalisation error decreases first, and then starts increasing (overfitting).

We can measure the ability to generalise by dividing the data set into a **training set**, used for fitting a hypothesis, and a **validation set**, used to test the generalisation ability. In other words, given a set of possible hypothesis classes $\mathcal{H}_i$, for each we fit the best model $h_i \in \mathcal{H}$ on the training set. Then, assuming large enough training and validation sets, the hypothesis that is the most accurate on the validation set is the best one, i.e., the one that has the best *inductive bias*. This is called **cross-validation**.

To report on the expected error of the best found model, we should not use the validation error. By using the validation set during training, it has effectively become part of the

training set. Therefore, a third set is needed, a **test set**, which contains examples not in the training nor validation sets.

A last important consideration is to also not use the same split between training/validation sets. Note that the training data used is a random sample. This means that if data is collected multiple times, we will end up with slightly different datasets, the fitted h will be slightly different with a different validation error. Thus, in choosing between $\mathcal{H}_i$ and $\mathcal{H}_j$, we will use them both multiple times on a number of training and validation sets and check if the difference between average errors of h_i and h_j is larger than the average difference between multiple h_i.

3 Supervised Learning

3.1 Learning a Class from Examples

For binary classification tasks in ML, we use **positive examples** and **negative examples**. For example, we want to learn the class C that indicates whether a car is a family car or not. We have a set of examples of cars and ask people to classify (label) the cars with 1 if it is a family car (positive examples) and 0 if it is not a family car (negative examples). Class learning is then the task of finding a description that is shared by *all* positive examples and *none* of the negative examples. Given such a description, we can make a prediction on a car that was never seen before.

For instance, suppose the factors that best describe the model are *price* and *engine volume*. That is, other features of cars are assumed irrelevant. Then, the dataset may consist of features $\mathbf{x}_1$ (price) and $\mathbf{x}_2$ (engine volume), and these vectors make up the features of the classification task, i.e., $\mathbf{X} = (\mathbf{x}_1, \mathbf{x}_2)^T$. One specific example of features is denoted by the vector $\mathbf{x}$ indexed by t. The labels are defined by:

$$r = \begin{cases} 1 & \text{if } \mathbf{x} \text{ is a positive example} \\ 0 & \text{if } \mathbf{x} \text{ is a negative example} \end{cases}$$

Each car is represented by an ordered pair $(\mathbf{x}, r)$ and the training set contains N examples such that $\mathcal{X} = \{\mathbf{x}^t, r^t\}_{t=1}^N$ where t indexes different examples in the set.

Suppose our *inductive bias* assumes the use of a *range* to conclude whether a car belongs to the class or not. That is, we select the hypothesis class $\mathcal{H}$ of *rectangular classifiers* such that

$$(p_1 \leq \text{price} \leq p_2) \text{ AND } (e_1 \leq \text{engine power} \leq e_2)$$

for suitable values (parameters) of p_1, p_2, e_1, e_2. Thus we want to find a hypothesis $h \in \mathcal{H}$ where

$$h(\mathbf{x}) = \begin{cases} 1 & \text{if } h \text{ classifies } \mathbf{x} \text{ as a positive example} \\ 0 & \text{if } h \text{ classifies } \mathbf{x} \text{ as a negative example} \end{cases}$$

We do not know the real classification function $C(\mathbf{x})$, so we cannot evaluate how well $h(\mathbf{x})$ matches $C(\mathbf{x})$. We do have a training set $\mathcal{X}$, which is a small subset of all possible examples. The **empirical error** is the proportion of training instances where predictions of h do not match the required values given by $\mathcal{X}$.

$$E(h \mid \mathcal{X}) = \sum_{t=1}^{N} 1(h(\mathbf{x}^t) \neq r^t)$$

where $1(a \neq b)$ is 1 if $a \neq b$ and 0 if $a = b$. The problem of generalisation can appear because for *real values* of the parameters of h, that is, the quadruple $(p_1^h, p_2^h, e_1^h, e_2^h) \in \mathbb{R}$, there are infinitely many possible rectangles for which the error is 0. However, some will correctly classify future predictions, while others will not.

- One possibility is to find the **most specific hypothesis**, S, such that it is the *tightest* rectangle that includes all positive examples and none of the negative examples. This gives a single hypothesis $h = S$. The area of the actual class C may be larger, but is

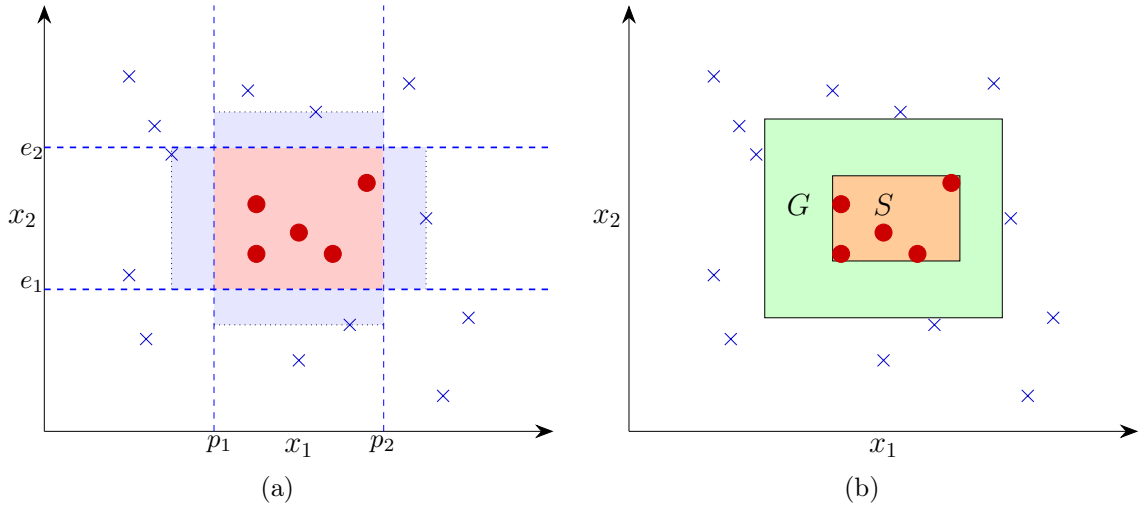


Figure 3.1: (a) an Example of a hypothesis $h \in \mathcal{H}$ where $\mathcal{H}$ is the set of rectangular classification boundaries. The blue shaded areas denote the generalisation bounds of the possible hypotheses. (b) the most specific hypothesis S and the most general hypothesis G.

never smaller than the area of S.

- The **most general hypothesis**, G, is the largest rectangle possible to draw around all the positive examples and none of the negative examples.

Any $h \in \mathcal{H}$ that is between S and G is a valid hypothesis with zero error, and is consistent with the training set. Such h makes up the **version space**. That is, the set of all hypotheses $h \in \mathcal{H}$ that correctly classify the training examples in $\mathbf{X}$. Given another sample of data, i.e., a new training set, then S, G, the version space, the parameters, and thus h, can be different.

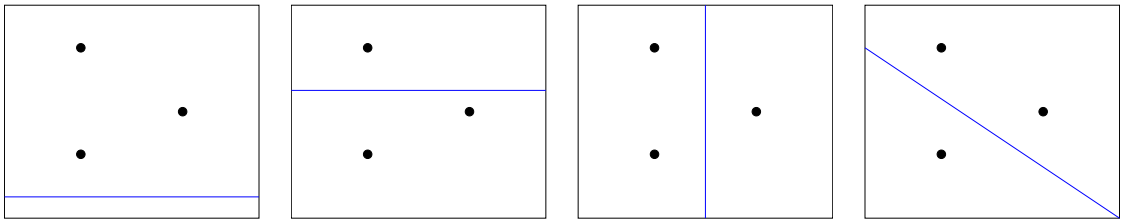
Figure 3.1a shows an example hypothesis, where the bounds of the G-set are shaded in blue. Every member of the S-set is consistent with all instances, and there are no hypotheses that are **more specific**. Similarly, every member of the G-set is consistent with all instances, and there are no consistent hypotheses that are **more general**.

It is intuitive to choose h halfway between S and G, so that the **margin** increases, that is, the distance between the boundary and the examples closest to it. Furthermore, notice in figure 3.1b that if examples fall between the boundaries of S and G, then there is no evidence in the training data to conclude a proper classification. This is a case of *doubt*, that is, it is uncertain whether the example belongs to the class C or not.

For this example, we can assume that $C \in \mathcal{H}$, i.e., there is some $h \in \mathcal{H}$, such that the error $E(h \mid \mathcal{X}) = 0$. More often than not, we cannot fully learn C because there is no $h \in \mathcal{H}$ for which the error is zero. Therefore, $\mathcal{H}$ needs to be flexible enough or have enough "capacity" to learn C.

3.2 Vapnik-Chervonenkis Dimension

To measure the complexity of a hypothesis class $\mathcal{H}$, we can use the *Vapnik-Chervonenkis (VC) dimension*. The VC dimension measures complexity by the number of distinct instances from $\mathbf{X}$ that can be completely discriminated by some $h \in \mathcal{H}$.



(a) (b) (c) (d)

Figure 3.2: Eight possible dichotomies (2^3) to separate three points in the XY-plane using hypothesis from the class of linear decision surfaces. This class successfully shatters three instances.

Using the definition from Tom Mitchell [5], we can define the notion of $\mathcal{H}$ **shattering** N data points. Consider $\mathbf{X}$ to have N data points. There are 2^N different possible subsets of $\mathbf{X}$ that partition the positive and negative examples. This partition is also referred to as a *dichotomy*. In fact, each $h \in \mathcal{H}$ imposes some dichotomy (partition) on the N points. That is, h partitions $\mathbf{X}$ into two subsets $\{\mathbf{x}^t \in \mathbf{X} \,|\, h(\mathbf{x}^t) = 1\}$ and $\{\mathbf{x}^t \in \mathbf{X} \,|\, h(\mathbf{x}^t) = 0\}$. The idea is that since there are 2^N possible dichotomies, $\mathcal{H}$ may not be able to represent some of these. Therefore, we say that $\mathcal{H}$ shatters $\mathbf{X}$ if every dichotomy (partition) of $\mathbf{X}$ can be represented by some hypothesis from the class $\mathcal{H}$. Consider figure 3.2, where the hypothesis class of linear decision surfaces, i.e., linear lines, is able to shatter three instances in the XY-plane.

Definition 3.1. Shattering hypothesis class
A set of instances $\mathbf{X}$ is **shattered** by hypothesis class $\mathcal{H}$ if and only if, for every dichotomy of $\mathbf{X}$ there exists some hypothesis $h \in \mathcal{H}$ that is consistent with this dichotomy.

If $\mathcal{H}$ cannot shatter a set of instances, then it must be possible to define a dichotomy for these instances. If a dichotomy is definable and cannot be shattered by $\mathcal{H}$, then it means that the instances cannot be represented by the hypothesis class $\mathcal{H}$. Therefore, the ability of $\mathcal{H}$ to shatter a set of instances is thus a measure of its **capacity** to represent target concepts defined over these instances.

The ability to shatter a set of instances is closely related to the *inductive bias* of a hypothesis class. Theoretically, the **unbiased hypothesis class** would be capable of representing every possible concept (dichotomy) definable over some instance space $\boldsymbol{\mathcal{X}}$. For example, the instance space of real valued XY-instances or the instance space of 100-dimensional instances, or the instance space of people described by their (age, height) attributes are all subsets of the instance space $\boldsymbol{\mathcal{X}}$. In other words, the unbiased hypothesis class shatters the instance space $\boldsymbol{\mathcal{X}}$. Of course, most (if not all) $\mathcal{H}$ will probably not be able to shatter $\boldsymbol{\mathcal{X}}$, but it may be able to shatter some subset of $\boldsymbol{\mathcal{X}}$. Therefore, it is reasonable to say that the larger the subset of $\boldsymbol{\mathcal{X}}$ that can be shattered by $\mathcal{H}$, the more **expressive** $\mathcal{H}$ is. The VC dimension of $\mathcal{H}$ is precisely this measure.

Definition 3.2. Vapnik-Chervonenkis Dimension
The $VC(\mathcal{H})$ of a hypothesis class $\mathcal{H}$ defined over an instance space $\boldsymbol{\mathcal{X}}$ is the size of the largest finite subset of that can be shattered by $\mathcal{H}$. If arbitrarily large subsets of $\boldsymbol{\mathcal{X}}$ can be shattered by $\mathcal{H}$, then $VC(\mathcal{H}) = \infty$.

The rules to determine the VC dimension are as follows:

- The examples can be arranged in any way possible.
- During the procedure of shattering, the same arrangement of examples must holds.
- If every subset of N examples can be shattered, then the lower bound $VC(\mathcal{H}) \geq N$ holds.
- If some subset of $N+1$ examples *cannot* be shattered, then the upper bound $VC(\mathcal{H}) < N + 1$ holds.
- Given $VC(\mathcal{H}) \geq N$ and $VC(\mathcal{H}) < N + 1$, then $VC(\mathcal{H}) = N$.

For instance, the hypothesis class of rectangular classifiers can shatter 4 data points in the XY-plane. Intuitively, being able to learn 4 data points without any error does not sound useful. However, in a smoothly changing world, instances close by will most of the times have the same labeling. The hypothesis class of linear decision surfaces in the same XY-instance space only has a VC dimension of 3. Thus rectangular classifiers are more expressive than linear classifiers. For example, the continuous XOR problem can be solved with a rectangular classifier, but not with a linear classifier. Generalising the 2-dimensional instance space of linear decision surfaces to the k-dimensional instance space, it is stated that VC of k-dimensional linear decision surfaces is $k + 1$.

3.3 Probably Approximately Correct Learning

The *Probably Approximately Correct* (PAC) Learning model enables us to reason about how many instances (N) are required and how much computation is required for a learning problem. First, the setting is introduced, then PAC learnability is introduced using [5].

Again consider $\boldsymbol{\mathcal{X}}$ to be the set of all possible instances over which *target functions* may be defined, e.g., set of cars, each described by (*price*, *engine volume*). Let $\boldsymbol{\mathcal{C}}$ refer to some set of target concepts that our learner might be instructed to learn. Each target concept $C \in \boldsymbol{\mathcal{C}}$ corresponds to some subset of $\mathbf{X} \subseteq \boldsymbol{\mathcal{X}}$, or equivalently to some boolean-valued function $C : \mathcal{X} \mapsto \{0, 1\}$. For example, the target concept $C \in \boldsymbol{\mathcal{C}}$ might be "cars that are family cars". If $\mathbf{x}$ is a positive example of C, then $C(\mathbf{x}) = 1$, and otherwise $C(\mathbf{x}) = 0$. Furthermore, we assume that instances $\mathbf{X}$ are generated from $\boldsymbol{\mathcal{X}}$, i.e., $\mathbf{X} \subseteq \boldsymbol{\mathcal{X}}$ according to some *stationary* distribution $\mathcal{D}$, and that the training set is presented as:

$$\mathcal{X} = \{(\mathbf{x}^t, C(\mathbf{x}^t) \,|\, \mathbf{x}^t \in \mathbf{X})\}_{t=1}^{N}$$

The **learner** L considers some subset $H \subseteq \mathcal{H}$ of possible hypotheses when attempting to learn the target concept. After observing the training data $\mathcal{X}$, the learner L must output some hypothesis $h \in H$, which is its estimate of C. In reality, L evaluates on a new set of data (test set) sampled from $\boldsymbol{\mathcal{X}}$ according to the same distribution $\mathcal{D}$.

We are interested in finding how close the learner's output hypothesis h approximates the actual target concept C. Therefore, the **true error** of h with respect to C and $\mathcal{D}$ will evaluate this approximation.

Definition 3.3. True error
The true error of hypothesis h with respect to target concept C and distribution $\mathcal{D}$ is the *probability* that h will *misclassify* an instance drawn at random according to $\mathcal{D}$.

$$error_{\mathcal{D}}(h) = P(C(\mathbf{x}) \neq h(\mathbf{x}) \,|\, \mathbf{x} \in \mathcal{D})$$

Note that the learner L does not know about the true error of h with respect to C. L can only observe the performance of h over *training examples*, and chooses its output hypothesis on this basis. Unfortunately, getting a true error $error_{\mathcal{D}}(h) = 0$ is not possible for two reasons.

1. Unless we provide a training set that contains every possible instance from the instance space $\boldsymbol{\mathcal{X}}$ (which is an unrealistic assumption), there may be multiple hypotheses consistent with the training examples, and the learner cannot be certain to pick the one corresponding to the target concept C.

2. Given the training samples are drawn randomly, there will always be some non-zero probability that the examples encountered by the learner will be misleading, i.e., probability of a continuous variable within an infinitesimal range is never zero.

Therefore, the learner should output an error that is bounded by some constant, ϵ, that can be made arbitrarily small. Furthermore, the learner is **not** required to *succeed* on every sequence of randomly drawn training examples. Rather, the probability of failure to classify should be bounded by some constant, δ, that can be made arbitrarily small. Thus, the learner L is required to **probably** learn a hypothesis that is **approximately correct**.

For some class $\boldsymbol{\mathcal{C}}$ of possible target concepts and a learner L using hypothesis space $H \subseteq \mathcal{H}$, we say that $\boldsymbol{\mathcal{C}}$ is PAC-learnable by L using H if, for all target $C \in \boldsymbol{\mathcal{C}}$, L will with probability $(1-\delta)$ output hypothesis $h \in H$ with $error_{\mathcal{D}}(h) \leq \epsilon$, after observing a reasonable amount of training examples and performing a reasonable amount of computation.

Definition 3.4. PAC-Learnability
Consider a concept class $\boldsymbol{\mathcal{C}}$ defined over a set of instances $\boldsymbol{\mathcal{X}}$ of length N and a learner L using hypothesis space $H \subseteq \mathcal{H}$. $\boldsymbol{\mathcal{C}}$ is **PAC-learnable** by L using H if for all $C \in \boldsymbol{\mathcal{C}}$, distributions $\mathcal{D}$ over $\boldsymbol{\mathcal{X}}$, ϵ such that $0 < \epsilon < 0.5$, and δ such that $0 < \delta < 0.5$, learner L will with probability at least $(1 - \delta)$ output a hypothesis $h \in H$ such that $error_{\mathcal{D}}(h) \leq \epsilon$, in time that is polynomial in $1/\epsilon$, $1/\delta$, N, and size(C).

Therefore, L must with arbitrarily high probability $(1 - \delta)$ output a hypothesis having arbitrarily low error (ϵ).

3.3.1 Example: Sample size of Rectangular classifier

As an example, consider the car classification problem where the inductive bias is captured by the hypothesis class of rectangular classifiers. Using the tightest rectangle S as our hypothesis, we want to find how many training examples N are needed to find a hypothesis with probability $(1 - \delta)$ that outputs an error of at most ϵ.

Because we consider $h = S$, and S is the tightest bound, the error region between C and h is approximately (for simplicity) the sum of the four strips between S and G (see figure 3.1b). The probability of a positive example from the test set falling in one of the regions of the four strips (and causing an error) should be at most ϵ. Note that we assume S, and therefore, consider every positive example that falls outside of S as a misclassification. To allow for an upper bound of ϵ, for any of these strips, the upper bound must be $\epsilon/4$, thus the error is at most $4(\epsilon/4) = \epsilon$. Overlaps of strips are counted twice, thus making this a simplified approximation of the upper bound error.

1. The probability that a randomly drawn example misses one of the strips is then $1-\epsilon/4$.
2. The probability that all N independent draws miss one of the strips is then $(1-\epsilon/4)^N$.
3. The probability that all N independent draws miss any of the four strips is then $4(1 - \epsilon/4)^N$.

We would like $4(1 - \epsilon/4)^N$ to be at most δ. Via $4(1 - \epsilon/4)^N \leq \delta$, we derive the following:

$$N \geq (4/\epsilon) \ln 4/\delta$$

which gives us the size of the training data so that with a confidence probability *at least* $(1 - \delta)$, a given example will be misclassified with probability *at most* ϵ. For example, let $\delta = 0.001$ and $\epsilon = 0.001$, then we need at least $N \approx 33176$ examples in the training set, so that with a probability of at least 0.999, a model h will be produced by the learner that has an error smaller or equal to 0.001.

3.4 Noisy data

Noise is any *unwanted anomaly* in the data, and due to noise, the class may be more difficult to learn, making an error of zero infeasible with simple hypothesis classes. There are several interpretations of noise:

- There may be imprecisions during the recording and collection of the input attributes, which may shift the data points in the input space.
- There may be errors during the labeling of the data points, which may lead to positive instances being labeled as negative instances, and vice versa. This is sometimes called *teacher noise*.
- There may be additional variables, not taken into account, that affect the label of an instance. These may be hidden/latent, as they might not be observable. The effect of such attributes is modelled as a random component and included as noise.

Usually, one can overcome noise by using a hypothesis class of higher complexity that is capable of learning more patterns via its parameters. However, there are considerations for keeping the model *simple* and accepting the noise as a random component by which the error is not likely to ever be zero.

1. For a simple model, it is easy to check whether an example is positive or negative.
2. For a simple model, it is easy to train due to its fewer parameters, compared to a complex model, e.g., a rectangular shape has four parameters, whereas an arbitrary shape may have many parameters.
3. A simple model has *less variance* between samples of training data, whereas a complex model is very sensitive to fluctuations between samples of training data.
4. A model that is too simple has *more bias*, and thus, assumes more, is more rigid, and may fail if the underlying class is more complex.
5. A simple model is easy to explain, which makes extraction of information from the training data possible.
6. Due to noise, the simple model will be less affected by single instances containing noise, and is therefore a better discriminant than a complex model. However, it may also make more mistakes on the training examples.
7. Simpler models generalise better than complex models. This is known as **Occam's razor**, that states that simpler explanations are more plausible and any unnecessary complexity should be *shaved off*.

3.5 Learning Multiple Classes

In the family car classification problem, we had two classes, i.e., yes/no or 1/0. In the general case, we have K classes denoted as $C_i, i = 1, \ldots, K$, and an input example belongs to one and exactly one of these classes. The training set is now of the form

$$\mathcal{X} = \{\mathbf{x}^t, \mathbf{r}^t\}_{t=1}^N$$

where $\mathbf{r}^t$ is a vector of size K, and $\mathbf{r}$ is a matrix with dimensions $K \times N$, and

$$r_i^t = \begin{cases} 1 & \text{if } \mathbf{x}^t \in C_i \\ 0 & \text{if } \mathbf{x}^t \in C_j, j \neq i \end{cases}$$

Figure 3.3 illustrates multi-label classification with the hypothesis class of rectangles. The idea is now to learn the boundaries separating the instances of one class from the instances of all other classes. Therefore, we view a K-class classification problem as K two-class problems. The training examples belonging to C_i are the positive examples of hypothesis h_i and the examples of all other classes are the negative examples of h_i. We need to learn K hypotheses for multi-label classification

$$h_i(\mathbf{x}^t) = \begin{cases} 1 & \text{if } \mathbf{x}^t \in C_i \\ 0 & \text{if } \mathbf{x}^t \in C_j, j \neq i \end{cases}$$

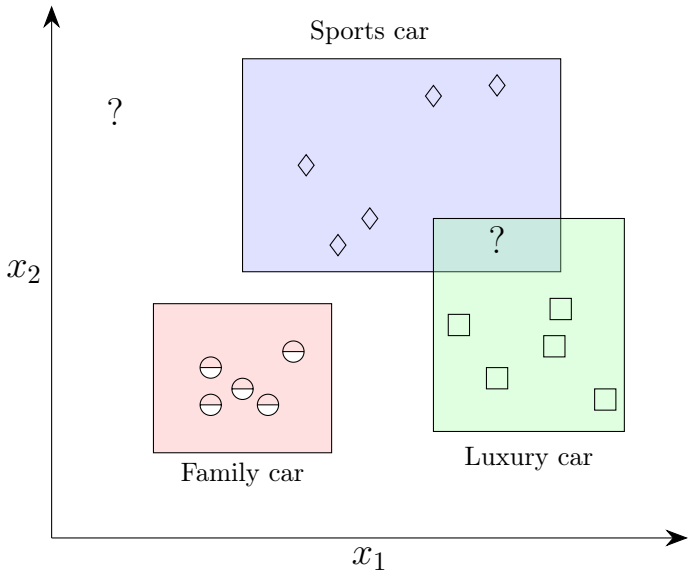


Figure 3.3: Three classes induced, each one covering the instances of one class and leaving outside the instances of the other classes. "?" are rejected regions, where no, or more than one class, is chosen.

The empirical error is the sum over the predictions for all classes over all instances.

$$E(\{h_i\}_{i=1}^K \mid \mathcal{X}) = \sum_{t=1}^{N} \sum_{i=1}^{K} 1(h_i(\mathbf{x}^t) \neq r_i^t)$$

Ideally, out of all h_i only one $h_i(\mathbf{x}^t) = 1$ if $\mathbf{x}^t$ belongs to C_i. When there are no, or two or more, h_i classifying as 1, then we cannot determine a class. This is the *doubt* of the classifier and it rejects such cases.

In the above, a **One-vs-Rest** approach is formulated, which means that for each class C_i, we train a binary classifier that accepts the particular instance as a member of its class and rejects everything else. In figure 3.3, the first model h_1 accepts family cars and rejects everything else. Similarly, the second model h_2 accepts luxury cars and rejects everything else. And so on.

In contrast, another strategy is the **One-vs-One** approach, where for every pair of classes, a binary classification model is trained. The formula for the amount of binary models is then $(K(K-1))/2$.

3.6 Regression

When the output is a numeric value, we would like to learn a **numeric function**. The function is not known, but we have a training set of examples drawn from the *unknown function*.

$$\mathcal{X} = \{\mathbf{x}^t, r^t\}_{t=1}^N$$

where $r^t \in \mathbb{R}$. If there is no noise assumed, then the task is **interpolation**. That is, we want to find a function $f(\mathbf{x})$ that passes through these points such that

$$r^t = f(\mathbf{x}^t)$$

In **polynomial interpolation**, given N data points, we want to find the polynomial of degree $(N-1)$ to predict the output of any $\mathbf{x}$. If $\mathbf{x}$ is outside of the range of the $\mathbf{x}^t$ records in the training set, then this is called **extrapolation**.

In **regression**, however, there is also noise added to the output of the unknown function

$$r^t = f(\mathbf{x}^t) + \epsilon$$

where $f(\mathbf{x}) \in \mathbb{R}$ is the unknown function and ϵ is the random noise. The reason for ϵ is that there are often extra unobserved hidden or latent variables that affect the variables of the training set, i.e., exogenous variables, for which the true unknown function might be

$$r^t = f^*(\mathbf{x}^t, \mathbf{z}^t)$$

where $\mathbf{z}^t$ denotes the **hidden variables**. We approximate this function with our model $g(\mathbf{x})$. The empirical error on the training set $\mathcal{X}$ is

$$E(g \,|\, \mathcal{X}) = \frac{1}{N} \sum_{t=1}^{N} (r^t - g(\mathbf{x}^t))^2$$

We want to find $g(\cdot)$ that minimises the empirical error. First, we assume the hypothesis class for $g(\cdot)$ of linear functions with a (relatively) small amount of parameters.

$$g(\mathbf{x}) = w_0 + w_1 x_1 + \cdots + w_d x_d = w_0 + \sum_{j=1}^{d} w_j x_j = w_0 + \mathbf{w}^T \mathbf{x}$$

For simplicity, we can consider the linear function of one variable x to estimate some quantity $g(x)$.

$$g(x) = w_1 x + w_0$$

where w_1 and w_0 are the parameters to learn from data, referred to as the coefficient and intercept, respectively. These parameters should minimise the empirical error, i.e.

$$E(w_1, w_0 \,|\, \mathcal{X}) = \frac{1}{N} \sum_{t=1}^{N} (r^t - (w_1 x + w_0))^2$$

To compute the minimum point, we can take the partial derivative of E with respect to w_1 and w_0, then set them to 0, and solve for the two parameters.

$$\frac{\partial E}{\partial w_0} = \frac{1}{N} \sum_{t=1}^{N} (r^t - (w_1 x^t + w_0))^2 \implies \frac{1}{N} \sum_{t=1}^{N} (r^t - w_1 x^t - w_0)^2 \implies \frac{1}{N} \sum_{t=1}^{N} -2(r^t - w_1 x^t - w_0)$$

$$\frac{\partial E}{\partial w_1} = \frac{1}{N} \sum_{t=1}^{N} (r^t - (w_1 x^t + w_0))^2 \implies \frac{1}{N} \sum_{t=1}^{N} (r^t - w_1 x^t - w_0)^2 \implies \frac{1}{N} \sum_{t=1}^{N} -2x^t(r^t - w_1 x^t - w_0)$$

Note that the identity $\sum_i^N x_i = N\overline{x}$, which is directly derived from the formula for the mean, is very useful in the following derivations. Starting with $\partial E/\partial w_0$, we get:

$$\begin{aligned}
\frac{1}{N}\sum_{t=1}^{N} -2(r^t - w_1 x^t - w_0) &= 0 \\
\sum_{t=1}^{N} r^t - w_1 x^t - w_0 &= 0 \\
\sum_{t=1}^{N}(r^t) - w_1 \sum_{t=1}^{N}(x^t) - w_0 &= 0 \\
N \cdot \overline{r} - w_1 \cdot N \cdot \overline{x} - w_0 &= 0 \\
-w_0 &= -\overline{r} + w_1\overline{x} \\
w_0 &= \overline{r} - w_1\overline{x}
\end{aligned}$$

And for $\partial E/\partial w_1$, we get:

$$\begin{aligned}
\frac{1}{N}\sum_{t=1}^{N} -2x^t(r^t - w_1 x^t - w_0) &= 0 \\
\sum_{t=1}^{N} -2x^t r^t + 2w_1(x^t)^2 + 2w_0 x^t &= 0 \\
\sum_{t=1}^{N} x^t r^t - w_1(x^t)^2 - w_0 x^t &= 0 \\
\sum_{t=1}^{N} x^t r^t - w_1(x^t)^2 - (\overline{r} - w_1\overline{x})x^t &= 0 \\
\sum_{t=1}^{N}(x^t r^t) - w_1 \sum_{t=1}^{N}(x^t)^2 - \overline{r}\sum_{t=1}^{N}(x^t) + w_1\overline{x}\sum_{t=1}^{N}(x^t) &= 0 \\
\sum_{t=1}^{N}(x^t r^t) - w_1 \sum_{t=1}^{N}(x^t)^2 - \overline{r} \cdot N \cdot \overline{x} + w_1 \cdot N \cdot \overline{x}^2 &= 0 \\
-w_1 \sum_{t=1}^{N}(x^t)^2 + w_1 \cdot N \cdot \overline{x}^2 &= -\sum_{t=1}^{N}(x^t r^t) + \overline{r} \cdot N \cdot \overline{x} \\
w_1(-\sum_{t=1}^{N}(x^t)^2 + N \cdot \overline{x}^2) &= -\sum_{t=1}^{N}(x^t r^t) + \overline{r} \cdot N \cdot \overline{x} \\
w_1 &= \frac{-\sum_{t=1}^{N}(x^t r^t) + \overline{r} \cdot N \cdot \overline{x}}{-\sum_{t=1}^{N}(x^t)^2 + N \cdot \overline{x}^2} \\
w_1 &= \frac{\sum_{t=1}^{N}(x^t r^t) - N\overline{r}\overline{x}}{\sum_{t=1}^{N}(x^t)^2 - N\overline{x}^2}
\end{aligned}$$

Now we can for any training set compute the parameters (weights) of the linear model. If the model incurs an error that is too large, then a higher order function may yield better

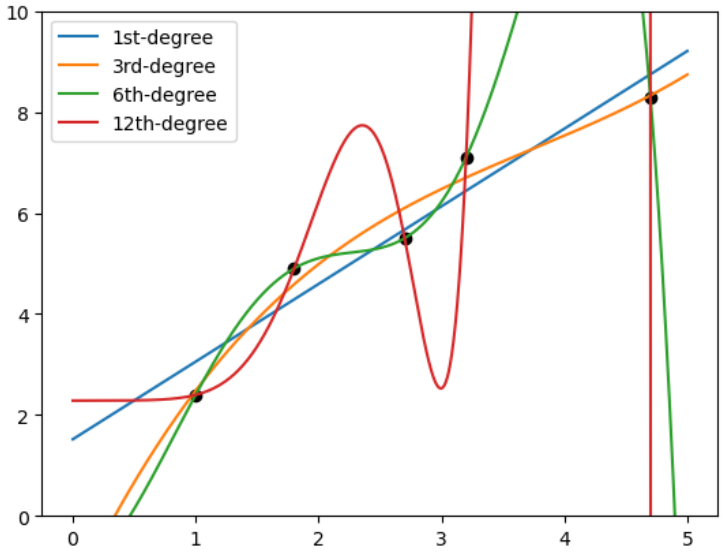


Figure 3.4: Polynomial regression of degrees one, three, six, and twelve. Notice how the higher polynomials overfit more, due to the underlying data being pretty simple.

results. For example, the quadratic

$$g(\mathbf{x}) = w_2 x_2^2 + w_1 x_1 + w_0$$

where we can similarly find an analytic solution for the parameters. A higher-order polynomial decreases the error on the training data, but it will capture individual examples more closely, instead of capturing a general trend. See figure 3.4 for an illustration of some orders of polynomials trying to capture the data.

3.7 Bias/Variance Dilemma

Given a training set $\mathcal{X} = \{\mathbf{x}^t, r^t\}$, drawn from some unknown joint probability distribution $P(\mathbf{x}, r)$, we can construct our estimate for the regression function $g(\cdot)$. The *expected squared error*, i.e., **Mean Squared Error**, can be written using the expected value $E[\cdot]$ as

$$E\left[(r - g(\mathbf{x}))^2 \,|\, \mathbf{x}\right] = \underbrace{E\left[(r - E\left[r \,|\, \mathbf{x}\right]) \,|\, \mathbf{x}\right]}_{\textbf{noise}} + \underbrace{(E\left[r \,|\, \mathbf{x}\right] - g(\mathbf{x}))^2}_{\textbf{squared error}}$$

Here, the first term on the right of the equals sign is the variance of the output r given the input $\mathbf{x}$. In other words, it is the variance of the added noise, and as such, does not depend on $g(\cdot)$ or $\mathcal{X}$, and therefore, is the part that can never be removed by any estimator $g(\cdot)$.

The second term on the right of the equals sign quantifies how much $g(\mathbf{x})$ deviates from the regression function, $E\left[r \,|\, \mathbf{x}\right]$. Another notation is often used as well:

$$g(\mathbf{x}) = \hat{y}$$
$$E\left[r \,|\, \mathbf{x}\right] = f(\mathbf{x}) = y$$

where $\hat{y}$ is the predicted value and y is the true value. The squared error depends on the

type of estimator $g(\cdot)$ and the training set, i.e., $g(\mathbf{x})$ may be a good fit for one training set but a bad fit for another. Therefore, to quantify how well the estimator $g(\cdot)$ performs, we average over many possible datasets. The average value, which is the average over samples $\mathcal{X}$, all of size N and drawn from the same distribution, is:

$$\underbrace{E_{\mathcal{X}}\left[\left(E\left[r \mid \mathbf{x}\right]-g(\mathbf{x})\right)^2 \mid \mathbf{x}\right]}_{\textbf{average expected error}} = \underbrace{\left(E\left[r \mid \mathbf{x}\right]-E_{\mathcal{X}}\left[g(\mathbf{x})\right]\right)^2}_{\textbf{bias}} + \underbrace{E_{\mathcal{X}}\left[\left(g(\mathbf{x})-E_{\mathcal{X}}\left[g(\mathbf{x})\right]\right)^2\right]}_{\textbf{variance}}$$

Bias measures how much the estimator is wrong, i.e., how many erroneous assumptions it makes, whereas **variance** measures how much the estimator $g(\mathbf{x})$ fluctuates around the expected value, $E\left[g(\mathbf{x})\right]$, for different samples. *Both should be as small as possible.*

We can compute bias and variance, given some known function $f(\cdot)$ with added noise, from which we sample M datasets, i.e., $\mathcal{X}_i = \{\mathbf{x}_i^t, r_i^t\}, i = 1, \ldots, M$. In real settings, this is not possible because we do not know $f(\cdot)$.

Starting with $E_{\mathcal{X}}\left[g(\mathbf{x})\right]$, it can be estimated by the average of each $g_i(\mathbf{x})$:

$$\overline{g}(\mathbf{x}) = \frac{1}{M}\sum_{i=1}^{M} g_i(\mathbf{x})$$

Using this identity, and the identity $E\left[r \mid \mathbf{x}\right] = f(\mathbf{x})$, we can formulate the bias and variance:

$$\text{bias}^2(g) = \frac{1}{N}\sum_t \left[\overline{g}(\mathbf{x}^t) - f(\mathbf{x}^t)\right]^2$$

$$\text{variance}(g) = \frac{1}{NM}\sum_t \sum_i \left[g_i(\mathbf{x}^t) - \overline{g}(\mathbf{x}^t)\right]^2$$

For an estimator that is a constant fit, e.g., $g_i(\mathbf{x}) = 2$, there is no variance, because the data is not used and every $g_i(\mathbf{x})$ outputs the same. However, the bias is very high, because the model makes many erroneous assumptions about the underlying data.

Another case is if we take the average of the target values r^t instead of the constant 2. The bias decreases, because the average constant is a better estimate than simply some constant. However, this increases the variance, because different samples $\mathcal{X}_i$ may have different average values.

In polynomial regression, small changes in the dataset cause a greater change in the fitted polynomials, and therefore, variance increases. However, a complex model will on average allow for a better fit of the data, thus bias decreases.

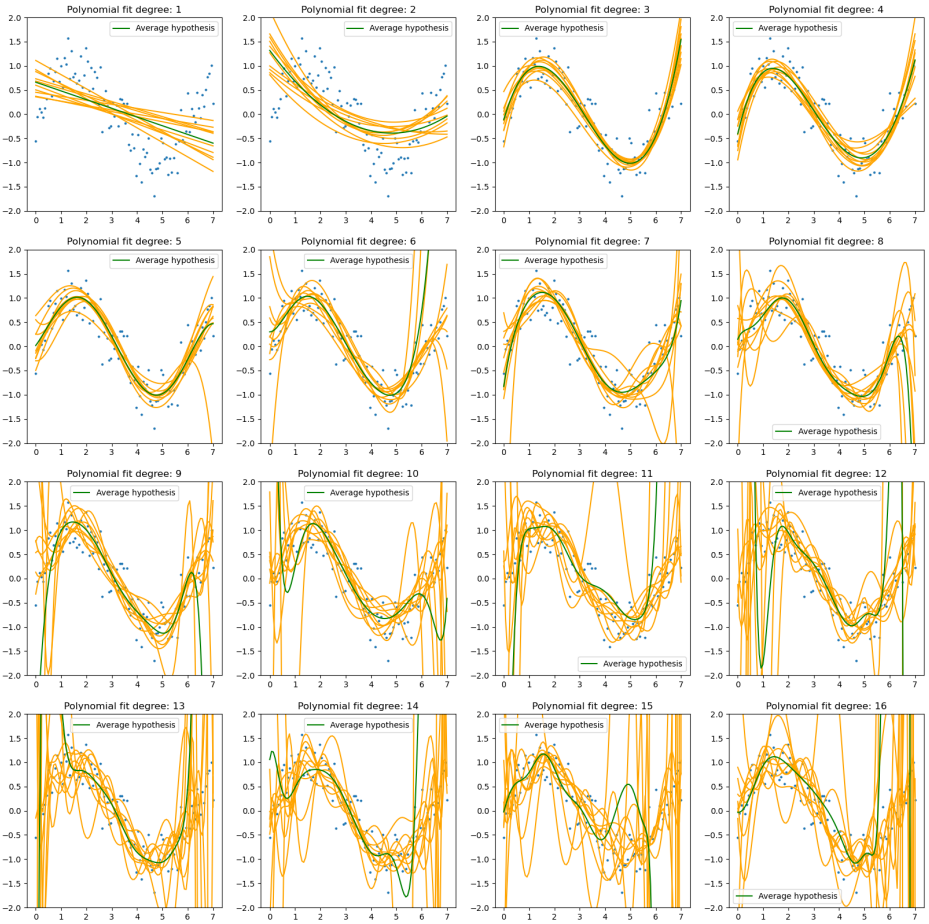


Figure 3.5: Polynomial regression of degrees 1-16, where the average hypothesis (green) of 10 models is derived from $n = 20$ out of $N = 100$.

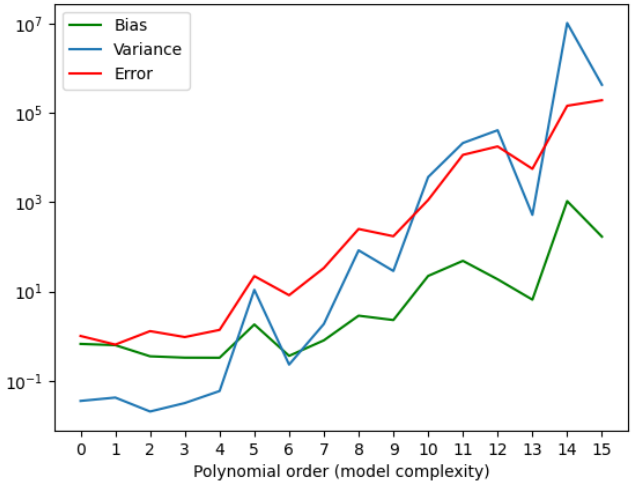


Figure 3.6: Plot of bias, variance, and error for each degree polynomial.

Figure 3.5 illustrates this idea for polynomials from degree 1 to 16, where the increase of degree leads to overfitting. In figure 3.6, the bias, variance, and mean squared error is plotted for these polynomials. In general, for an estimator $g(\cdot)$, an underlying function $f(\cdot)$, and a measurement of the hypothesis complexity $\mathcal{HC}(\cdot)$ (e.g., the VC-dimension):

- If $\mathcal{HC}(g) > \mathcal{HC}(f)$, then **overfitting**
- If $\mathcal{HC}(g) < \mathcal{HC}(f)$, then **underfitting**
- If $\mathcal{HC}(g) \approx \mathcal{HC}(f)$, then **bias/variance balance**

3.8 Dimensions of supervised ML algorithms

The aim is to build a good and useful approximation of r^t using the model $g(\mathbf{x}^t \mid \theta)$. In doing so, there are three decisions to make:

1. **Model**: A model $g(\cdot)$ defines a hypothesis class $\mathcal{H}$, i.e., an inductive bias, and a particular value of θ, i.e., the parameters, which instantiates one hypothesis $h \in \mathcal{H}$.

2. **Loss function**: The loss function $L(\cdot)$ computes the difference between the desired output, r^t, and the approximation of it, $g(\mathbf{x}^t \mid \theta)$, given the current values of the parameters θ. The *approximation error*, or loss, is the sum of losses over the individual instances
$$E(\theta \mid \mathcal{X}) = \sum_t L(r^t, g(\mathbf{x}^t \mid \theta))$$

3. **Optimisation procedure**: to find θ^* that minimises the total error:
$$\theta^* = \arg\min_\theta E(\theta \mid \mathcal{X})$$
where arg min returns the argument that minimises. In polynomial regression, the optimum could be solved analytically, but this is not always the case.

4 Multivariate Methods

Several measurements or events can be formulated as an observation vector. The sample of data is then a **data matrix**:

$$\mathbf{X} = \begin{bmatrix} X_1^1 & X_2^1 & \dots & X_d^1 \\ X_1^2 & X_2^2 & \dots & X_d^2 \\ \vdots & \vdots & \ddots & \vdots \\ X_1^N & X_2^N & \dots & X_d^N \end{bmatrix}$$

where the d columns correspond to d variables (inputs). The N rows correspond to **independent** and **identically distributed** (i.i.d) *observations*, *examples*, or *instances* on N individuals or events.

Typically, these variables are *correlated*. If they are not, then there is no need for **multivariate analysis**. The aim may be to summarise this large body of data by means of relatively few parameters (**simplification**). Or, we may be interested in generating hypotheses about the data (**exploratory**). Another use case is predicting the value of one variable from the values of other variables. If the predicted variable is discrete, this is *multivariate classification*, and if it is numeric, this is *multivariate regression*.

4.1 Parameter estimation

The **mean vector** $\boldsymbol{\mu}$ is defined such that each of its elements corresponds to the mean of one column (variable) of $\mathbf{X}$:

$$E[\mathbf{X}] = \boldsymbol{\mu} = [\mu_1, \dots, \mu_d]$$

The variance for a specific variable X_i is denoted as σ_i^2, and the covariance of two variables X_i and X_j is defined as:

$$\sigma_{ij} = \text{Cov}(X_i, X_j) = E[(X_i - \mu_i)(X_j - \mu_j)] = E[X_i X_j] - \mu_i \mu_j$$

with $\sigma_{ij} = \sigma_{ji}$, and when $i = j$, then $\sigma_{ij} = \sigma_i^2$. A data matrix with d variables has d variances, and due to symmetry of the covariance matrix, has $d(d+1)/2$ covariances. The $d \times d$ covariance matrix $\boldsymbol{\Sigma}$ whose (i,j)th elements correspond to σ_{ij} is:

$$\boldsymbol{\Sigma} = \begin{bmatrix} \sigma_2^1 & \sigma_{12} & \dots & \sigma_{1d} \\ \sigma_{21} & \sigma_2^2 & \dots & \sigma_{2d} \\ \vdots & \vdots & \ddots & \vdots \\ \sigma_{d1} & \sigma_{d2} & \dots & \sigma_d^2 \end{bmatrix}$$

In vector-matrix notation the covariance matrix is defined as:

$$\boldsymbol{\Sigma} = \text{Cov}(\mathbf{X}) = E\left[(\mathbf{X} - \boldsymbol{\mu})(\mathbf{X} - \boldsymbol{\mu})^T\right] = E\left[\mathbf{X}\mathbf{X}^T\right] - \boldsymbol{\mu}\boldsymbol{\mu}^T$$

If two variables have a linear relationship, then the strength of the relationship will dictate how positive or negative the covariance is. Normalising the covariance between -1 and 1

results in the correlation:

$$\text{Corr}(X_i, X_j) = \rho_{ij} = \frac{\sigma_{ij}}{\sigma_i \sigma_j}$$

Maximum likelihood estimator Given a multivariate sample, the maximum likelihood estimator for the mean $\boldsymbol{\mu}$ is the *sample mean* $\mathbf{m}$. Its ith dimension is the average of the ith column of $\mathbf{X}$:

$$\mathbf{m} = \frac{\sum_{t=1}^{N} \mathbf{x}^t}{N}, \quad \text{with} \quad m_i = \frac{\sum_{t=1}^{N} x_i^t}{N}, i = 1, \ldots, d$$

The estimator of $\boldsymbol{\Sigma}$ is $\mathbf{S}$, the *sample covariance matrix*, with entries:

$$s_i^2 = \frac{\sum_{t=1}^{N} (x_i^t - m_i)^2}{N}$$
$$s_{ij} = \frac{\sum_{t=1}^{N} (x_i^t - m_i)(x_j^t - m_j)}{N}$$

Note that these are biased estimates, because we divide by N and not $N-1$. However, if in an application the estimates vary significantly depending on N or $N-1$, there is serious trouble anyway. The **sample correlation coefficients** are:

$$r_{ij} = \frac{s_{ij}}{s_i s_j}$$

and the sample correlation matrix $\mathbf{R}$ contains r_{ij}.

4.2 Multivariate Normal Distribution

In the multivariate case, where $\mathbf{x}$ is a d-dimensional vector of variables, and they are normally distributed, we have the *Gaussian* or *Normal* distribution

$$P(\mathbf{x}) = \frac{1}{(2\pi)^{d/2} |\boldsymbol{\Sigma}|^{1/2}} \exp\left[-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu})\right]$$

and we write $\mathbf{x} \sim \mathcal{N}_d(\boldsymbol{\mu}, \boldsymbol{\Sigma})$, where $\boldsymbol{\mu}$ is the mean vector and $\boldsymbol{\Sigma}$ is the covariance matrix. To break down a part of the normal distribution, recall that for the univariate case, we have that

$$\frac{(X-\mu)^2}{\sigma^2} = \frac{(X-\mu)(X-\mu)}{\sigma^2} = (X-\mu)(\sigma^2)^{-1}(X-\mu)$$

is the squared distance between X and the mean μ in the units of the standard deviation, where we divide by the variance to normalise. This form is similar to the multivariate case, where the **Mahalanobis distance** is used:

$$(\mathbf{x} - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu}) = c^2$$

where c^2 is the d-dimensional hyperellipsoid centered at $\boldsymbol{\mu}$, and its shape and orientation are determined by $\boldsymbol{\Sigma}$. The distance is characterised by comparing the variance between two variables. If one variable has a larger variance than another, it receives less weight in the Mahalanobis distance. This happens because of the use of the inverse of $\boldsymbol{\Sigma}$ that inverts the relationship. Similarly, two highly correlated variables do not contribute as much to the assigned weight as two less correlated variables. This has the effect of standardising all

variables to unit variance and eliminating correlations.

A special *naive* case is where the components of the variables $\mathbf{X}$ are independent and $\text{Cov}(X_i, X_j) = 0$, for $i \neq j$, and $\text{Var}(X_i) = \sigma_i^2, \forall i$. Then, the covariance matrix is diagonal and the joint density is simply the product of the individual univariate densities:

$$P(\mathbf{x}) = \prod_{i=1}^{d} P(X_i) = \frac{1}{(2\pi)^{d/2} \prod_{i=1}^{d} \sigma_i} \exp\left[-\frac{1}{2}\sum_{i=1}^{d}(\frac{X_i - \mu_i}{\sigma_i})^2\right]$$

4.2.1 Example: Bivariate Normal Distribution

In the **bivariate** case where $d = 2$, we can visualise the distribution. See Figues 4.1 and 4.2. When the two variables X_1 and X_2 are independent, then these are simply two Gaussian that are parallel to each other on the XY-axes (upper left plots). The density becomes an ellipse if the variances of σ_1^2 and σ_2^2 are different (upper middle and right plots). The density rotates depending on the sign of the covariances (lower middle and right plots). This is due to the linear relationship that the covariance represents.

The mean vector is then $\boldsymbol{\mu} = \begin{bmatrix}\mu_1 & \mu_2\end{bmatrix}^T$, and the covariance matrix is

$$\boldsymbol{\Sigma} = \begin{bmatrix}\sigma_1^2 & \rho\sigma_1\sigma_2 \\ \rho\sigma_1\sigma_2 & \sigma_2^2\end{bmatrix}$$

where ρ is the correlation coefficient. The *joint bivariate density* can be expressed in the form

$$P(X_1, X_2) = \frac{1}{2\pi\sigma_1\sigma_2\sqrt{1-\rho^2}} \exp\left[-\frac{1}{2(1-\rho^2)}(z_1^2 - 2\rho z_1 z_2 + z_2^2)\right]$$

where $z_i = (X_i - \mu_i)/\sigma_i$ for $i = 1, 2$. The z_i terms are standardised variables, which is also referred to as **z-normalisation**.

Because $z_1^2 - 2\rho z_1 z_2 + z_2^2 = c^2$ is a constant, if $|\rho| \leq 1$, this becomes the equation for an ellipse, where the value of ρ determines the slope of the major axis of the ellipse. In the lower middle and right plots of figure 4.1, we see how the slope is negative when $\rho < 0$ and positive when $\rho > 0$. Recall that ρ is the correlation between the two variables. If $\rho = 0$, then there is no linear relationship between X_1 and X_2, and therefore, we get a distribution like the upper left plot. If $\rho = 1$, then one variable tells us everything about the other, and thus, the bivariate distribution reduces to a univariate distribution. This indicates that we can effectively reduce dimensionality.

And finally, the lower left plot shows the case where the covariance values are small, i.e., the determinant of the covariance matrix $|\boldsymbol{\Sigma}|$ is small. In this case, samples are closer to the mean vector $\boldsymbol{\mu}$, just like in the univariate case where a small value of σ^2 indicates that samples are close to μ. Furthermore, a small $|\boldsymbol{\Sigma}|$ may also indicate that there is high correlation between variables.

$\boldsymbol{\Sigma}$ is a *symmetric positive definite* matrix. This is the multivariate equivalent of saying that $\text{Var}(X) > 0$. If this is not the case, then $\boldsymbol{\Sigma}$ is considered singular and its determinant is 0. This is either due to *linear dependence* between the dimensions or because one of the dimensions has variance 0. In such a case, dimensionality should be reduced (via PCA for example) to get a positive definite matrix.

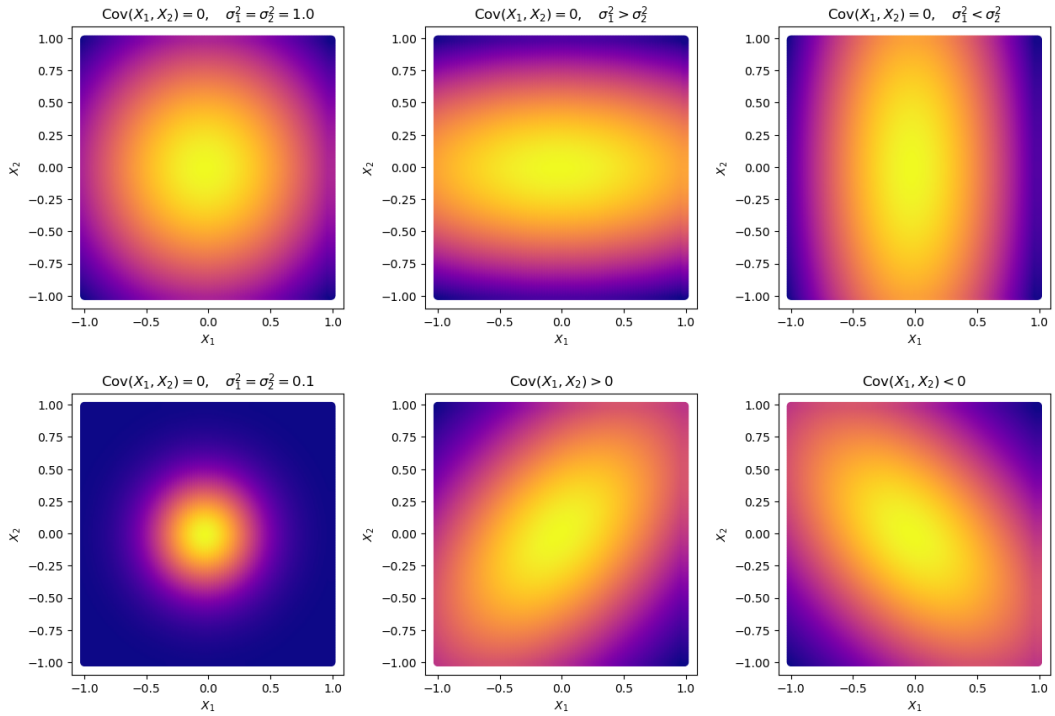


Figure 4.1: Bivariate normals as a heatmap in 2D with varying covariance matrices.

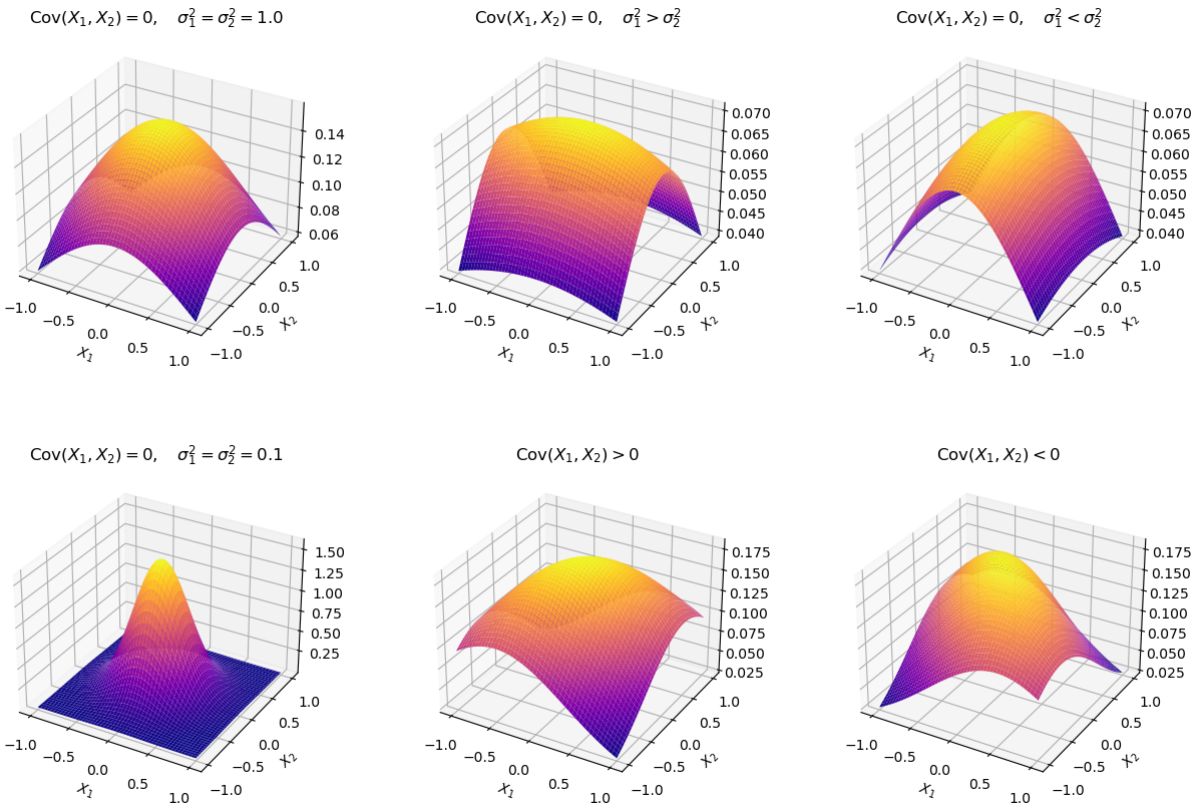


Figure 4.2: Bivariate normals as a surface in 3D with varying covariance matrices.

4.2.2 Projection of normals

To project a d-dimensional Gaussian on to the vector $\mathbf{w}$, we assume $\mathbf{X} \sim \mathcal{N}_d(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ and $\mathbf{w} \in \mathbb{R}^d$. Then,

$$\begin{aligned}\mathbf{w}^T\mathbf{X} &= \begin{bmatrix} w_1 & w_2 & \dots & w_d \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \\ \vdots \\ X_d \end{bmatrix} \\ &= w_1X_1 + w_2X_2 + \cdots + w_dX_d \sim \mathcal{N}_d(\mathbf{w}^T\boldsymbol{\mu}, \mathbf{w}^T\boldsymbol{\Sigma}\mathbf{w})\end{aligned}$$

In the general case, if $\mathbf{W}$ is a $d \times k$ matrix with rank $k < d$, then the k-dimensional projection $\mathbf{W}^T\mathbf{X}$ is k-variate normal:

$$\mathbf{W}^T\mathbf{X} \sim \mathcal{N}_d(\mathbf{W}^T\boldsymbol{\mu}, \mathbf{W}^T\boldsymbol{\Sigma}\mathbf{W})$$

That is, if we project a d-dimensional normal distribution to a space that is k-dimensional, then it projects to a k-variate normal. This is analogous to taking a slice out of the upper left plot of figure 4.2 by which we effectively project the bivariate normal to a univariate normal.

Note: the rank of a matrix $\mathbf{A}$ is the maximum number of *linearly independent* columns of $\mathbf{A}$.

4.3 Multivariate Classification

For classification, we want to know the probability of class C_i given the samples $\mathbf{x}$, where $\mathbf{x} \in \mathbb{R}^d$ is a vector of variables. Formulated as a conditional probability, this is $P(C_i \mid \mathbf{x})$. Using Bayes' rule, this can be rewritten as

$$P(C_i \mid \mathbf{x}) = \frac{P(\mathbf{x} \mid C_i)P(C_i)}{P(\mathbf{x})}$$

If we now assume that $P(\mathbf{x} \mid C_i)$ is taken from a normal density, $\mathcal{N}_d(\boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i)$, we have

$$P(\mathbf{x} \mid C_i) = \frac{1}{(2\pi)^{d/2}|\boldsymbol{\Sigma}_i|^{1/2}} \exp\left[-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu}_i)^T\boldsymbol{\Sigma}_i^{-1}(\mathbf{x} - \boldsymbol{\mu}_i)\right]$$

The main reason why this assumption is reasonable is its analytical simplicity. Furthermore, the normal density is a model for many naturally occurring phenomena, because the examples of most classes can be seen as mildly changed versions of a single example *prototype*, namely $\boldsymbol{\mu}_i$, and the covariance matrix, $\boldsymbol{\Sigma}_i$, denotes the amount of *noise* in each variable and the *correlations* between these variables.

Although the multivariate normal is a robust and useful approximation, its main requirement is that the sample of a class should form a single group. If there are multiple groups, one should use a **mixture model**.

Derivation of Discriminant function It is often useful to work with the log likelihood instead of the regular likelihood estimation, because probabilities may become too small

for computers to deal with (due to limited floating point precision). Using the following logarithm rules, we derive the discriminant function $g_i(\mathbf{x})$:

- $\log(ab) = \log(a) + \log(b)$
- $\log\left(\frac{a}{b}\right) = \log(a) - \log(b)$
- $\log(a^b) = b\log(a)$

Thus, starting with

$$P(C_i \mid \mathbf{x}) = \frac{P(\mathbf{x} \mid C_i)p(C_i)}{P(\mathbf{x})} \tag{4.1}$$

$$\log(P(C_i \mid \mathbf{x})) = \log\left(\frac{P(\mathbf{x} \mid C_i)p(C_i)}{P(\mathbf{x})}\right) \tag{4.2}$$

$$\log(P(C_i \mid \mathbf{x})) = \log(P(\mathbf{x} \mid C_i)p(C_i)) - \log(P(\mathbf{x})) \tag{4.3}$$

$$\log(P(C_i \mid \mathbf{x})) = \log(P(\mathbf{x} \mid C_i)) + \log(p(C_i)) - \log(P(\mathbf{x})) \tag{4.4}$$

Since $\log(P(\mathbf{x}))$ is a normalisation constant and the same for all classes, we can ignore it. This gives the discriminant function:

$$g_i(\mathbf{x}) = \log(P(C_i \mid \mathbf{x})) = \underbrace{\log(P(\mathbf{x} \mid C_i))}_{\textbf{likelihood}} + \underbrace{\log(p(C_i))}_{\textbf{prior}}$$

Using this form as the discriminant for classification, we can find the most likely class by taking the class C_i associated with the maximum value of the discriminant $g_i(\mathbf{x})$. In other words,

$$\arg\max_i g_i(\mathbf{x})$$

Assuming $P(\mathbf{x} \mid C_i) \sim \mathcal{N}_d(\boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i)$, we can derive the following useful form for $\log(P(\mathbf{x} \mid C_i))$. Note that $\log(P(C_i))$ is already in the right form.

$$\log(P(\mathbf{x} \mid C_i)) = \log\left(\frac{1}{(2\pi)^{d/2}|\boldsymbol{\Sigma}_i|^{1/2}} \exp\left[-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu}_i)^T \boldsymbol{\Sigma}_i^{-1}(\mathbf{x} - \boldsymbol{\mu}_i)\right]\right) \tag{4.1}$$

$$\log(P(\mathbf{x} \mid C_i)) = \log\left(\frac{1}{(2\pi)^{d/2}|\boldsymbol{\Sigma}_i|^{1/2}}\right) + \log\left(\exp\left[-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu}_i)^T \boldsymbol{\Sigma}_i^{-1}(\mathbf{x} - \boldsymbol{\mu}_i)\right]\right) \tag{4.2}$$

$$\log(P(\mathbf{x} \mid C_i)) = \log(1) - \log\left((2\pi)^{d/2}|\boldsymbol{\Sigma}_i|^{1/2}\right) + \left(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu}_i)^T \boldsymbol{\Sigma}_i^{-1}(\mathbf{x} - \boldsymbol{\mu}_i)\right) \tag{4.3}$$

$$\log(P(\mathbf{x} \mid C_i)) = -\log\left((2\pi)^{d/2}\right) - \log\left(|\boldsymbol{\Sigma}_i|^{1/2}\right) + \left(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu}_i)^T \boldsymbol{\Sigma}_i^{-1}(\mathbf{x} - \boldsymbol{\mu}_i)\right) \tag{4.4}$$

$$\log(P(\mathbf{x} \mid C_i)) = -\frac{d}{2}\log(2\pi) - \frac{1}{2}\log(|\boldsymbol{\Sigma}_i|) - \frac{1}{2}(\mathbf{x} - \boldsymbol{\mu}_i)^T \boldsymbol{\Sigma}_i^{-1}(\mathbf{x} - \boldsymbol{\mu}_i) \tag{4.5}$$

With this, the derivation is complete and the discriminant $g_i(\mathbf{x})$, for each class C_i, becomes:

$$g_i(\mathbf{x}) = -\frac{d}{2}\log(2\pi) - \frac{1}{2}\log(|\boldsymbol{\Sigma}_i|) - \frac{1}{2}(\mathbf{x} - \boldsymbol{\mu}_i)^T \boldsymbol{\Sigma}_i^{-1}(\mathbf{x} - \boldsymbol{\mu}_i) + \log(P(C_i))$$

4.3.1 Maximum Likelihood Estimates for each class

Given a training sample with $K \geq 2$ classes and $\mathcal{X} = \{\mathbf{x}^t, \mathbf{r}^t\}$, where $r_i^t = 1$ if $\mathbf{x}^t \in C_i$, and 0 otherwise, then, separately for each class, the estimates for the *means* and *covariances* are

found using maximum likelihood:

$$\begin{aligned}\hat{P}(C_i) &= \frac{\sum_t r_i^t}{N} \\ \mathbf{m}_i &= \frac{\sum_t r_i^t \mathbf{x}^t}{\sum_t r_i^t} \\ \mathbf{S}_i &= \frac{\sum_t r_i^t (\mathbf{x}^t - \mathbf{m}_i)(\mathbf{x}^t - \mathbf{m}_i)^T}{\sum_t r_i^t}\end{aligned}$$

These are then used in the discriminant function to get estimates for the discriminants. If we ignore the first constant term, plug these in, and expand, we have

$$g_i(\mathbf{x}) = -\frac{1}{2}\log(|\mathbf{S}_i|) - \frac{1}{2}(\mathbf{x}^T\mathbf{S}_i^{-1}\mathbf{x} - 2\mathbf{x}^T\mathbf{S}_i^{-1}\mathbf{m}_i + \mathbf{m}_i^T\mathbf{S}_i^{-1}\mathbf{m}_i) + \log(\hat{P}(C_i))$$

which defines a **quadratic discriminant**, as it can be written in the usual matrix-vector quadratic form:

$$g_i(\mathbf{x}) = \underbrace{\mathbf{x}^T\mathbf{W}_i\mathbf{x}}_{x^2 \text{ term}} + \underbrace{\mathbf{w}_i^T\mathbf{x}}_{x \text{ term}} + \underbrace{w_{i0}}_{\text{const}}$$

where

$$\begin{aligned}\mathbf{W}_i &= -\frac{1}{2}\mathbf{S}_i^{-1} \\ \mathbf{w}_i &= \mathbf{S}_i^{-1}\mathbf{m}_i \\ w_{i0} &= -\frac{1}{2}\mathbf{m}_i^T\mathbf{S}_i^{-1}\mathbf{m}_i - \frac{1}{2}\log(|\mathbf{S}_i|) + \log(\hat{P}(C_i))\end{aligned}$$

The number of parameters to be estimated are $K \cdot d$ for the means and $K \cdot d(d+1)/2$ for the covariance matrices. For the estimates to be reliable on small samples, one may want to decrease dimensionality, via subset selection or PCA.

4.3.2 Shared Covariance matrix Estimates

If we make a simplification by assuming that all variables share a common covariance matrix, the number of parameters decreases. We *pool* the covariance matrices by weighing them against the estimate for the prior $\hat{P}(C_i)$, and element-wise summing the matrices.

$$\mathbf{S} = \sum_i \hat{P}(C_i)\mathbf{S}_i$$

In this case, the first term of $g_i(\mathbf{x})$, which normalised based on each $\mathbf{S}_i$ in the case of different covariance matrices, reduces, and we are left with

$$\begin{aligned}g_i(\mathbf{x}) &= -\frac{1}{2}(\mathbf{x} - \mathbf{m}_i)^T\mathbf{S}^{-1}(\mathbf{x} - \mathbf{m}_i) + \log(\hat{P}(C_i)) \\ &= -\frac{1}{2}(\mathbf{x}^T\mathbf{S}_i^{-1}\mathbf{x} - 2\mathbf{x}^T\mathbf{S}_i^{-1}\mathbf{m}_i + \mathbf{m}_i^T\mathbf{S}_i^{-1}\mathbf{m}_i) + \log(\hat{P}(C_i))\end{aligned}$$

The number of parameters is now $K \cdot d$ for the means and only $d(d+1)/2$ for the shared covariance matrix.

If, in addition to a shared covariance matrix, the priors are equal, then the optimal decision rule is to assign the input data to the class whose mean has the smallest *Mahalanobis*

distance to the input data. In contrast, unequal priors leads to the decision boundary being shifted towards the less likely class. Geometrically, shifting the decision boundary towards the less likely class means that the more likely class ranges over more space, and thus is sensitive to the mass of the prior.

The quadratic term from the derivation, $\mathbf{x}^T\mathbf{S}^{-1}\mathbf{x}$, cancels because it is common in all discriminants, and the decision boundaries are linear, leading to a **linear discriminant**.

$$g_i(\mathbf{x}) = \mathbf{w}_i^T\mathbf{x} + w_{i0}$$

where

$$\begin{aligned}\mathbf{w}_i &= \mathbf{S}^{-1}\mathbf{m}_i \\ w_{i0} &= -\frac{1}{2}\mathbf{m}_i^T\mathbf{S}^{-1}\mathbf{m}_i + \log(\hat{P}(C_i))\end{aligned}$$

4.3.3 Naive Bayes' Classifier

A further simplification is possible if we assume that all off-diagonals of the covariance matrix, $\sigma_{ij}, i \neq j$, are 0. In other words, all variables are independent. This is the **Naive Bayes' Classifier** where $P(X_j \mid C_i), \forall i \forall j$ are univariate Gaussian. $\mathbf{S}$ with entries s_j and its inverse $1/s_j$ are diagonal, which simplifies the discriminant function:

$$g_i(\mathbf{x}) = -\frac{1}{2}\sum_{j=1}^{d}\left(\frac{X_j - m_{ij}}{s_j}\right)^2 + \log(\hat{P}(C_i))$$

Note that the term $(X_j - m_{ij})/s_j$ is the *z-normalisation* and measures the distance between X_j and m_{ij} in terms of standard deviation units. The number of parameters in this case is $K \cdot d$ for the means and d for the variances. The complexity of $\mathbf{S}$ is reduced from $\mathcal{O}(d^2)$ to $\mathcal{O}(d)$.

4.3.4 Nearest Mean Classifier

Simplifying even further, if we assume that all variances are also equal, then the Mahalanobis distance reduces to the *Euclidean distance*. This means that $|\mathbf{S}| = s^2 d$ and $\mathbf{S}^{-1} = (1/s^2)\mathbf{I}$. The number of parameters in this case is $K \cdot d$ for the means and 1 for s^2.

$$\begin{aligned}g_i(\mathbf{x}) &= -\frac{||\mathbf{x} - \mathbf{m}_i||^2}{2s^2} + \log(\hat{P}(C_i)) \\ &= -\frac{1}{2s^2}\sum_{j=1}^{d}(X_j^t - m_{ij})^2 + \log(\hat{P}(C_i))\end{aligned}$$

If the priors are equal, then the $\log(\hat{P}(C_i))$ term is for all classes the same, and thus can be removed. Furthermore, $1/2s^2$ is a constant, and can also be removed. What is left is then $g_i(\mathbf{x}) = -||\mathbf{x} - \mathbf{m}_i||^2$. This is named the **nearest mean classifier**, because it assigns the input to the class of the nearest mean. If we think of each mean as the *ideal* or *prototype* for the corresponding class, then this is a template matching procedure.

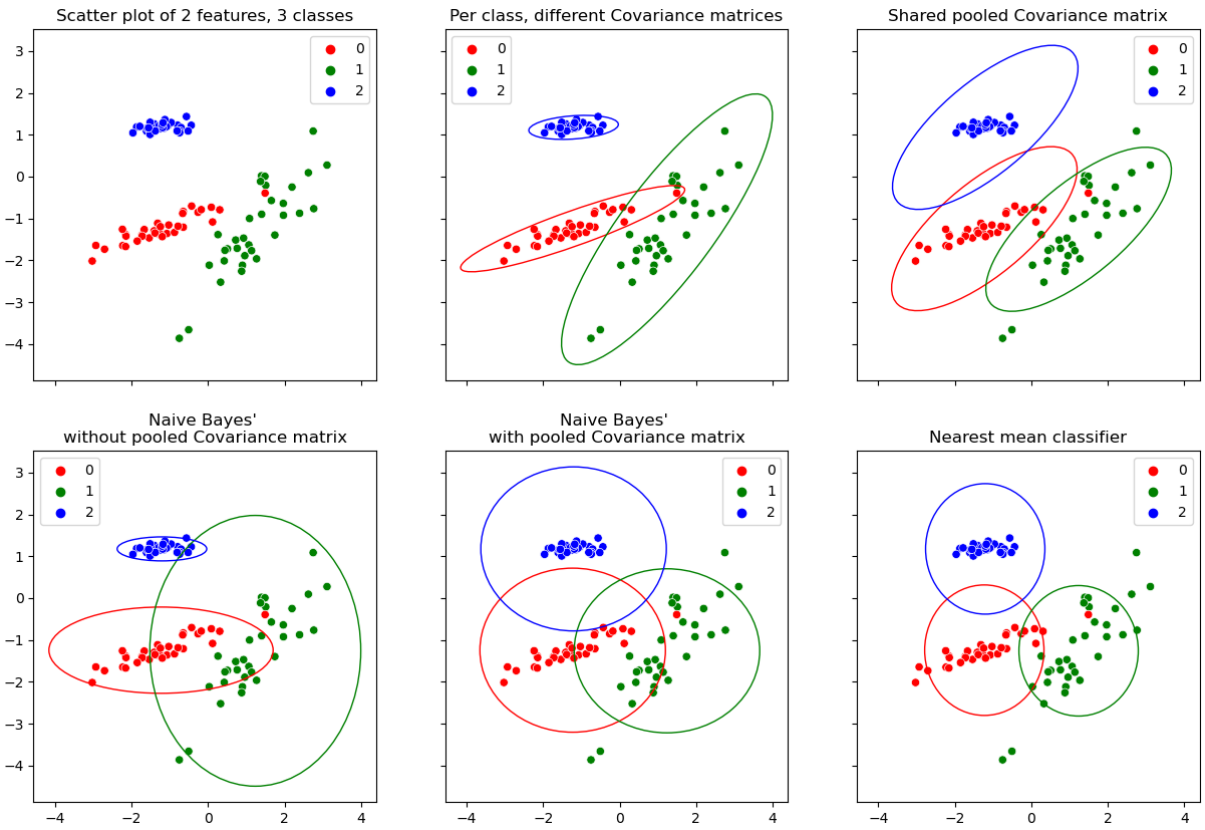


Figure 4.3: Summary of multivariate classification techniques. The ellipses are drawn with a scale of three times the standard deviation. The models are ordered from most parameters to least parameters.

4.3.5 Distance as discriminant

An alternative to the task of classification via a discriminant function, is the task of finding the best distance function. Instead of learning a discriminant function $g_i(\mathbf{x})$, this approach to classification is concerned with learning the best suitable distance function $\mathcal{D}(\mathbf{x}_1, \mathbf{x}_2)$. That is, given any $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3$, where $\mathbf{x}_1$ and $\mathbf{x}_2$ belong to the same class, i.e., $\mathbf{x}_1, \mathbf{x}_2 \in C_1$, and $\mathbf{x}_1$ and $\mathbf{x}_3$ belong to different classes, i.e., $\mathbf{x}_3 \in C_2$, we would like to have

$$\mathcal{D}(\mathbf{x}_1, \mathbf{x}_2) < \mathcal{D}(\mathbf{x}_1, \mathbf{x}_3)$$

4.4 Tuning Complexity

Figure 4.3 shows a summary of the multivariate classification methods. From the top, left to right, to the bottom, left to right, the classification methods increase in the amount of assumptions, and decrease in the amount of parameters. This is another example of the bias/variance trade-off. When we make simplifying assumptions about the covariance matrices and decrease the amount of parameters to be estimated, we create the risk of introducing bias. On the other hand, if no such assumptions are made and the matrices are arbitrary, the quadratic discriminant may have large variance on small datasets.

When we have a small dataset, with potentially different covariance matrices, it may be better to assume a shared covariance matrix. A single covariance matrix has fewer parameters and it can be estimated with data from *all classes*, instead of each specific class.

In **regularised discriminant analysis** (RDA), the quadratic classifier, linear classifier, and the nearest mean classifier are combined by using additional hyperparameters. *Regularisation* corresponds to approaches where one starts with high variance and constrains towards lower variance, at the risk of increasing bias. For *parametric classification*, the covariance matrices can be written as a weighted average of the three special cases:

$$\mathbf{S}_i' = \alpha\sigma^2\mathbf{I} + \beta\mathbf{S} + (1-\alpha-\beta)\mathbf{S_i}$$

Where $\mathbf{S}_i$ is the estimate for the covariance of C_i, and $\mathbf{S}$ is the shared pooled estimate for the covariance matrix. For the parameters α and β, we get the following results:

- when $\alpha = \beta = 0$, this leads to $\mathbf{S}_i' = \mathbf{S}_i$, which is the *quadratic classifier*.
- when $\alpha = 0$ and $\beta = 1$, the covariance matrices are shared, $\mathbf{S}_i' = \mathbf{S}$, and we get a *linear classifier*.
- when $\alpha = 1$ and $\beta = 0$, the covariance matrices are diagonal with σ^2 on the diagonals, i.e., $\mathbf{S}_i' = \sigma^2\mathbf{I}$, and we get the *nearest mean classifier*.

In between these extremes for α and β, we get a variety of classifiers, where cross-validation can help weed out the bad hypotheses.

When the dataset is small, another approach to regularisation is the use of Bayesian approach by defining priors over $\boldsymbol{\mu}_i$ and $\mathbf{S}_i$, or using cross-validation by choosing the best of the four multivariate classification methods.

4.5 Discrete Features

A discrete feature is one that takes on one of n different values. For example, color $\in \{\text{red}, \text{blue}, \text{green}\}$. Let us say that X_j are binary (Bernoulli) variables in the form of

$$p_{ij} = P(X_j = 1 \,|\, C_i)$$

meaning that for each C_i, this is the conditional probability of variable X_j being true. If we assume that all X_j are independent binary variables, then we have

$$P(\mathbf{x} \,|\, C_i) = \prod_{j=1}^{d} p_{ij}^{X_j}(1-p_{ij})^{(1-X_j)}$$

This uses the *Binomial distribution* denoted by a set of independent Bernoulli trials. This is also another example of the naive Bayes' classifier where $P(X_j \,|\, C_i)$ are Bernoulli. The discriminant function is

$$\begin{aligned} g_i(\mathbf{x}) &= \log P(\mathbf{x} \,|\, C_i) + \log(P(C_i)) \\ &= \sum_j [X_j \log p_{ij} + (1-X_j)\log 1 - p_{ij}] + \log(P(C_i)) \end{aligned}$$

This is linear. The estimator for p_{ij} is

$$\hat{p}_{ij} = \frac{\sum_t X_j^t r_i^t}{\sum_t r_i^t}$$

Example applications are *document categorisation* and *spam filtering*. In the **bag of words** representation, we choose *a priori* d words that we believe give information about the underlying class. Each text is then a d-dimensional binary vector where $X_j = 1$ if the word associated with j occurs in the text, and 0 otherwise.

For example, if $d = 3$ and we have the binary variables X_{cat}, X_{tree}, X_{mouse}, we can order them in a vector

$$\mathbf{x} = \begin{bmatrix} X_{\text{cat}} \\ X_{\text{tree}} \\ X_{\text{mouse}} \end{bmatrix}$$

Now, given the input text $\mathcal{X}$ = "The cat is in the tree", we can create a feature embedding of $X_j = 1$ if $j \in \mathcal{X}$. Therefore,

$$\mathbf{x} = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}$$

Note that since we assume the ordering of words is independent from other words, this representation loses all ordering information in the input text (we create an arbitrary ordering), hence the name **bag of words**.

After training, $\hat{p}_{ij}$ estimates the probability that word j occurs in a text that is of class i. In the general case, instead of binary features, let us say we have the multinomial X_j chosen from the set $\{v_1, v_2, \ldots, v_k\}$. To work with this type of data, we need to convert it to new $0/1$ dummy variables.

$$z_{jk}^t = \begin{cases} 1 & \text{if } X_j^t = v_k \\ 0 & \text{otherwise} \end{cases}$$

This way the variable $X_{\text{color}} \in \{\text{red}, \text{blue}, \text{green}\}$ is converted to three separate dummy variables $z_{\text{color.red}}$, $z_{\text{color.blue}}$, $z_{\text{color.green}}$.

Let p_{ijk} denote the probability that X_j takes on the value v_k and belongs to class C_i.

$$p_{ijk} = P(z_{jk} = 1 \,|\, C_i) = P(X_j = v_k \,|\, C_i)$$

Assuming independence between the features, we have

$$P(\mathbf{x} \,|\, C_i) = \prod_{j=1}^{d} \prod_{k=1}^{n_j} p_{ijk}^{z_{jk}}$$

The discriminant function is then

$$g_i(\mathbf{x}) = \sum_j \sum_k z_{jk} \log p_{ijk} + \log P(C_i)$$

The maximum likelihood estimator for p_{ijk} is

$$\hat{p}_{ijk} = \frac{\sum_t z_{jk}^t r_i^t}{\sum_t r_i^t}$$

4.6 Multivariate Regression

Previously, we considered regression with a single variable to find the estimator $g(\mathbf{x}) = w_0 + \mathbf{w}^T\mathbf{x}$. In the general case, we are working with multiple variables, $X_1, X_2, \ldots, X_d$, and noise. This is often called **multiple regression**. The multivariate linear model is

$$\begin{aligned} r^t &= g(\mathbf{x}^t \,|\, w_0, w_1, \ldots, w_d) + \epsilon \\ &= w_0 + w_1 x_1^t + w_2 x_2^t + \cdots + w_d x_d^t + \epsilon \end{aligned}$$

We assume ϵ to be normal with mean 0 and constant variance. Because the linear model is a linear combination (plus noise), we can formulate this problem via matrix-vector notation.

$$\mathbf{X} = \begin{bmatrix} 1 & x_1^1 & x_2^1 & \ldots & x_d^1 \\ 1 & x_1^2 & x_2^2 & \ldots & x_d^2 \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_1^N & x_2^N & \ldots & x_d^N \end{bmatrix}, \quad \mathbf{w} = \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_d \end{bmatrix}, \quad \mathbf{r} = \begin{bmatrix} r_0 \\ r_1 \\ \vdots \\ r_N \end{bmatrix}$$

Note that the first column of $\mathbf{X}$ has all 1's to account for the constant term, w_0, in the linear model. The equation now becomes:

$$\mathbf{X}\mathbf{w} = \mathbf{r}$$

The error, as before, is the *sum of squared differences*, and to *maximise* the likelihood is the same as *minimising* the sum of squared differences.

$$E(w_0, w_1, \ldots, w_d \,|\, \mathcal{X}) = \frac{1}{N}\sum_{t=1}^{N}(r^t - w_0 - w_1 x_1^t - w_2 x_2^t \cdots - w_d x_d^t)^2$$

Formulated with matrix-vector notation, we have

$$E(\mathbf{w} \,|\, \mathcal{X}) = ||\mathbf{r} - \mathbf{X}\mathbf{w}||^2 = (\mathbf{r} - \mathbf{X}\mathbf{w})^T(\mathbf{r} - \mathbf{X}\mathbf{w})$$

To solve for $\mathbf{w}$, we can take the partial derivative of the error with respect to each of the parameters, $w_j, j = 0, \ldots, d$, which results in a set of **normal equations**. Using matrix calculus, and the following calculus identities, the solution of $\mathbf{w}$ can easily be found.

1. Derivative of **linear function**:

$$\frac{\partial}{\partial \mathbf{x}}\mathbf{a}\cdot\mathbf{x} = \frac{\partial}{\partial \mathbf{x}}\mathbf{a}^T\mathbf{x} = \frac{\partial}{\partial \mathbf{x}}\mathbf{a}\mathbf{x}^T = \mathbf{a}$$

   analogous to $\frac{d}{dx}ax = a$.

2. Derivative of **quadratic function** if $\mathbf{A}$ is symmetric:

$$\frac{\partial}{\partial \mathbf{x}}\mathbf{x}^T\mathbf{A}\mathbf{x} = 2\mathbf{A}\mathbf{x}$$

analogous to $\frac{d}{dx}ax^2 = 2ax$. If $\mathbf{A}$ is not symmetric, then the quadratic derivative becomes:

$$\frac{\partial}{\partial \mathbf{x}}\mathbf{x}^T\mathbf{A}\mathbf{x} = (\mathbf{A} + \mathbf{A}^T)\mathbf{x}$$

Before using these identities, we can simplify and expand the error:

$$\begin{aligned}
\frac{\partial}{\partial \mathbf{w}}E(\mathbf{w} \mid \mathcal{X}) &= \frac{\partial}{\partial \mathbf{w}}(\mathbf{r} - \mathbf{X}\mathbf{w})^T(\mathbf{r} - \mathbf{X}\mathbf{w}) && (4.1)\\
&= \frac{\partial}{\partial \mathbf{w}}(\mathbf{r}^T - (\mathbf{X}\mathbf{w})^T)(\mathbf{r} - \mathbf{X}\mathbf{w}) && (4.2)\\
&= \frac{\partial}{\partial \mathbf{w}}(\mathbf{r}^T - \mathbf{w}^T\mathbf{X}^T)(\mathbf{r} - \mathbf{X}\mathbf{w}) && (4.3)\\
&= \frac{\partial}{\partial \mathbf{w}}(\mathbf{r}^T - \mathbf{w}^T\mathbf{X}^T)\mathbf{r} - (\mathbf{r}^T - \mathbf{w}^T\mathbf{X}^T)\mathbf{X}\mathbf{w} && (4.4)\\
&= \frac{\partial}{\partial \mathbf{w}}\mathbf{r}^T\mathbf{r} - \mathbf{w}^T\mathbf{X}^T\mathbf{r} - \mathbf{r}^T\mathbf{X}\mathbf{w} + \mathbf{w}^T\mathbf{X}^T\mathbf{X}\mathbf{w} && (4.5)\\
&= \frac{\partial}{\partial \mathbf{w}}\mathbf{r}^T\mathbf{r} - \mathbf{w}^T\mathbf{X}^T\mathbf{r} - (\mathbf{w}^T\mathbf{X}^T\mathbf{r})^T + \mathbf{w}^T\mathbf{X}^T\mathbf{X}\mathbf{w} && (4.6)\\
&= \frac{\partial}{\partial \mathbf{w}}\mathbf{r}^T\mathbf{r} - 2\mathbf{w}^T\mathbf{X}^T\mathbf{r} + \mathbf{w}^T\mathbf{X}^T\mathbf{X}\mathbf{w} && (4.7)
\end{aligned}$$

Note that in step 4.5, $\mathbf{r}^T\mathbf{X}\mathbf{w} = \mathbf{r}^T(\mathbf{X}\mathbf{w}) = ((\mathbf{X}\mathbf{w})^T\mathbf{r})^T = (\mathbf{w}^T\mathbf{X}^T\mathbf{r})^T$. And in step 4.6, note that $(\mathbf{w}^T\mathbf{X}^T\mathbf{r})^T = \mathbf{w}^T\mathbf{X}^T\mathbf{r}$, because it is a scalar[5].

Now we take the derivative of the form in 4.7, set it to 0, and solve for $\mathbf{w}$.

$$\begin{aligned}
\frac{\partial}{\partial \mathbf{w}}\left(\mathbf{r}^T\mathbf{r} - 2\mathbf{w}^T\mathbf{X}^T\mathbf{r} + \mathbf{w}^T\mathbf{X}^T\mathbf{X}\mathbf{w}\right) \implies -2\mathbf{X}^T\mathbf{r} + 2\mathbf{X}^T\mathbf{X}\mathbf{w} &= 0\\
2\mathbf{X}^T\mathbf{X}\mathbf{w} &= 2\mathbf{X}^T\mathbf{r}\\
\mathbf{X}^T\mathbf{X}\mathbf{w} &= \mathbf{X}^T\mathbf{r}\\
\mathbf{w} &= (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{r}
\end{aligned}$$

For *multivariate polynomial regression*, the method is the same if we define the variables as $X_1 = X_2$, $X_2 = X_2^2$, etc. In other words, exponentiation of the polynomial powers. Other arbitrary non-linear functions are also possible as *basis functions*. For example, $X_2 = sin(X_2)$ or $X_3 = \exp X_3^2\pi$.

The advantage of linear models is that after the regression function is computed, we can inspect the weight vector $\mathbf{w}$, to extract knowledge. The signs of w_j tell us whether X_j has a positive or negative effect on the output. The absolute values of each w_j tells us how important the feature is, by which we can rank them in terms of importance, and remove the features whose w_j is close to 0.

[5] https://math.stackexchange.com/questions/369694/matrix-calculus-in-least-square-method

5 Linear Discrimination

Recall that in classification, we define a set of *discriminant functions* $g_j(\mathbf{x}), j = 1, \ldots, K$, and then we

$$\text{choose } C_i \text{ if } g_i(\mathbf{x}) = \max_{j=i}^{K} g_j(\mathbf{x})$$

In other words, we choose $C_i = \arg\max_i g_i(\mathbf{x})$. We first estimated the prior probability, $\hat{P}(C_i)$, and the class likelihoods, $\hat{P}(\mathbf{x} \mid C_i)$, and then used Bayes' rule to calculate the posterior densities. the discriminant function was the *log likelihood* of the posterior $\hat{P}(C_i \mid \mathbf{x})$. This is called **likelihood-based classification**. This section discusses the alternative **discriminant-based classification**, where we assume a model directly from the discriminant, rather than from Gaussians or correlations.

Thus, the discriminant model is defined as

$$g_i(\mathbf{x} \mid \mathbf{\Phi}_i)$$

where it is explicitly parameterised with the set of parameters $\mathbf{\Phi}_i$, as opposed to likelihood-based methods that have implicit parameters that define the densities, e.g., mean vector and covariance matrix. The difference between likelihood and discriminant-based classification is characterised as follows:

- **Inductive bias**: Instead of assuming the form of the class densities (e.g., Gaussian), we assume the form of the boundaries separating the classes.
- **Learning approach**: Instead of searching for the parameters that maximise sample likelihoods separately for each class, we learn the model parameters $\mathbf{\Phi}$ by optimising for the quality of the separation, i.e., estimating the *boundaries* between classes.

5.1 Generalising the Linear Model

When the linear model $g_i(\mathbf{x} \mid \mathbf{w}_i, w_{i0}) = \mathbf{w}_i^T \mathbf{x} + w_{i0}$ is not flexible enough, we can increase complexity by using the **quadratic discriminant function**.

$$g_i(\mathbf{x} \mid \mathbf{W}_i, \mathbf{w}_i, w_{i0}) = \mathbf{x}^T \mathbf{W}_i \mathbf{x} + \mathbf{w}_i \mathbf{x} + w_{i0}$$

This has $\mathcal{O}(d^2)$. Equivalently, we can preprocess the input by adding **higher-order terms** (product terms). For example, given two input values x_1, x_2, we can define new variables.

$$\mathbf{z} = \begin{bmatrix} x_1 \\ x_2 \\ x_1^2 \\ x_2^2 \\ x_1 x_2 \end{bmatrix}$$

These denote the same quadratic terms used before. This linear function defined in the 5-dimensional $\mathbf{z}$ space corresponds to a non-linear function in the 2-dimensional $\mathbf{x}$ space.

Thus, for K input values in $\mathbf{x}$, we write the discriminant as

$$g_i(\mathbf{x}) = \sum_{j=1}^{K} w_{ij}\phi_j(\mathbf{x}) + w_{i0}$$

where $\phi_j(\mathbf{x})$ are the **basis functions** or **potential functions**. Other examples of basis functions are $\sin(x_1), \exp(x_2)$, etc.

5.2 Geometry of the Linear Discriminant

5.2.1 Two Classes

For the case of two classes, it is sufficient to have one discriminant function. The reason for this is that for two discriminants $g_1(\mathbf{x})$ and $g_2(\mathbf{x})$, their difference tells us how to discriminate between the two classes. Therefore, we can reduce them via the following:

$$\begin{aligned} g(\mathbf{x}) &= g_1(\mathbf{x}) - g_2(\mathbf{x}) \\ &= (\mathbf{w}_1^T\mathbf{x} + w_{10}) - (\mathbf{w}_2^T\mathbf{x} + w_{20}) \\ &= \mathbf{w}_1^T\mathbf{x} + w_{10} - \mathbf{w}_2^T\mathbf{x} - w_{20} \\ &= (\mathbf{w}_1^T - \mathbf{w}_2^T)\mathbf{x} + (w_{10} - w_{20}) \\ &= \mathbf{w}^T\mathbf{x} + w_0 \end{aligned}$$

where we

$$\text{choose} \begin{cases} C_1 & \text{if } g(\mathbf{x}) > 0 \\ C_2 & \text{otherwise} \end{cases}$$

This defines a hyperplane where $\mathbf{w}$ is the **weight vector** and w_0 is the **threshold**, because the decision rule is now: choose C_1 if $\mathbf{w}^T\mathbf{x} > -w_0$, and choose C_2 otherwise.

The hyperplane divides the input space into two half-spaces, also denoted as decision regions. Any $\mathbf{x} \in C_1$ implies $\mathbf{x} \in \mathcal{R}_1$ is *positive*, and any $\mathbf{x} \in C_2$ implies $\mathbf{x} \in \mathcal{R}_2$ is *negative*.

For two points $\mathbf{x}_1$ and $\mathbf{x}_2$, both on the decision boundary, that is, $g(\mathbf{x}_1) = g(\mathbf{x}_2) = 0$, then

$$\begin{aligned} \mathbf{w}^T\mathbf{x}_1 + w_0 &= \mathbf{w}^T\mathbf{x}_2 + w_0 \\ \mathbf{w}^T(\mathbf{x}_1 - \mathbf{x}_2) &= 0 \end{aligned}$$

The interpretation of this is the following. Although $\mathbf{x}_1$ and $\mathbf{x}_2$ make the discriminant output 0, they do not lie on the decision boundary, i.e., they do not point in the same direction as the hyperplane. Rather, they are vectors that have their tip on the hyperplane. Therefore, if we subtract one from the other, we get a vector that points in the same direction as the hyperplane.

Thus, we say that $\mathbf{w}$ is normal (orthogonal) to any vector lying on the hyperplane. In other words, $\mathbf{w}$ is orthogonal to all vectors on the hyperplane, where these vectors are constructed by taking the difference between two points $\mathbf{x}_1$ and $\mathbf{x}_2$ that have their tip on the hyperplane.

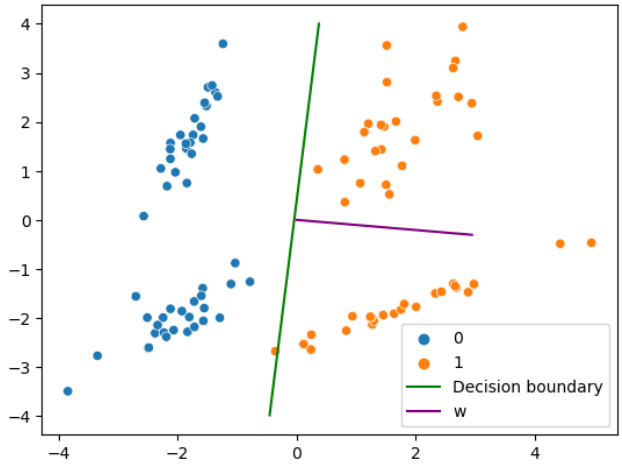


Figure 5.1: Partition of the input space in decision regions.

For more geometric intuition, we can rewrite the vector $\mathbf{x}$

$$\mathbf{x} = \mathbf{x}_p + r\frac{\mathbf{w}}{||\mathbf{w}||}$$

where $\mathbf{x}_p$ is the projection of $\mathbf{x}$ onto the hyperplane, and r is the distance from $\mathbf{x}$ to the hyperplane, which may be negative if $\mathbf{x}$ is on the negative side and positive if $\mathbf{x}$ is on the positive side. The formula basically says that to obtain $\mathbf{x}$, we follow the projected vector $\mathbf{x}_p$, then add to it r times the normalised (orthogonal) vector $\mathbf{w}/||\mathbf{w}||$.

Note that since $\mathbf{x}_p$ lies on the hyperplane, we know that $g(\mathbf{x}_p) = 0$. The distance from $\mathbf{x}$ to the hyperplane is then:

$$r = \frac{g(\mathbf{x})}{||\mathbf{w}||}$$

and the distance from the origin to the hyperplane is then:

$$r_0 = \frac{w_0}{||\mathbf{w}||}$$

Therefore, w_0 determines the *location* of the hyperplane with respect to the origin, and $\mathbf{w}$ determines its *orientation*.

5.2.2 Example: Binary Linear Discriminant

Consider a sample $\mathcal{X} = \{\mathbf{x}^t, r^t\}_{t=1}^{100}$ with $\mathbf{x}^t = \begin{bmatrix} x_1 & x_2 \end{bmatrix}$ and $r^t \in \{0, 1\}$. Using linear discriminants, we find the 2-dimensional weight vector $\mathbf{w}$. See figure 5.1 for the sample scatter plot, the decision boundary, and the normal $\mathbf{w}$. Calculating the accuracy can be done via:

$$acc = \frac{1}{N}\sum_{t=1}^{N} 1(1(\mathbf{w}^T\mathbf{x} + w_0 > 0) = r^t)$$

where $1(a = b)$ results in 1 if $a = b$, and 0 otherwise. The accuracy with respect to the sample is 0.99 due to one misclassification.

5.2.3 Multiple Classes

When there are $K > 2$ classes, there are K discriminant functions. We assume that the parameters are learned, and obtain

$$g_i(\mathbf{x} \mid \mathbf{w}_i, w_{i0}) = \begin{cases} > 0 & \text{if } \mathbf{x} \in C_i \\ \leq 0 & \text{otherwise} \end{cases}$$

This is again the **One-vs-Rest** approach to classification. Using such discriminant functions, we assume that all classes are **linearly separable**, that is, for each class C_i, there exists a hyperplane H_i, such that all $\mathbf{x} \in C_i$ lie on its *positive* side, and all $\mathbf{x} \in C_j, j \neq i$, lie on its *negative* side.

Ideally, there should be a single $g_i(\mathbf{x})$ greater than 0 and all others less than or equal to zero. This is not always possible because boundaries/regions may overlap or it could be that every $g_i(\mathbf{x}) \leq 0$. We could *reject* these cases, but usually we assign $\mathbf{x}$ to the class with the highest discriminant:

$$\text{Choose } C_i \text{ if } g_i(\mathbf{x}) = \max_{j=1}^{K} g_j(\mathbf{x})$$

5.3 Pairwise Separation

If the classes are not linearly separable, we could divide it into a set of linear problems. This is the **One-vs-One** approach to classification, where a linear discriminant is used for each pair of distinct classes.

$$g_{ij}(\mathbf{x} \mid \mathbf{w}_{ij}, w_{ij0}) = \mathbf{w}_{ij}^T \mathbf{x} + w_{ij0}$$

The parameters $\mathbf{w}_{ij}, i \neq j$ yield the following piece-wise function:

$$g_{ij}(\mathbf{x}) = \begin{cases} > 0 & \text{if } \mathbf{x} \in C_i \\ \leq 0 & \text{if } \mathbf{x} \in C_j \\ \text{irrelevant} & \text{otherwise} \end{cases}$$

If we are training the model $g_{ij}(\mathbf{x})$ and we have a training input $\mathbf{x}^t \in C_k$, where $k \neq i$, and $k \neq j$, then $\mathbf{x}^t$ is not used during training. In other words, to produce $g_{ij}(\mathbf{x})$, we first filter the dataset, and only train on the sub-sample corresponding to class C_i and C_j.

$$\begin{aligned} &\text{Choose } C_i \text{ if } \forall j \neq i, g_{ij}(\mathbf{x}) > 0 \\ &\qquad \implies (g_{i1}(\mathbf{x}) > 0) \wedge (g_{i2}(\mathbf{x}) > 0) \wedge \cdots \wedge (g_{i(K-1)}(\mathbf{x}) > 0) \end{aligned}$$

In many cases, this is too strict for class C_i. We can relax the conjunction by using a summation and choosing the maximum.

$$g_i(\mathbf{x}) = \sum_{j \neq i} g_{ij}(\mathbf{x})$$

5.4 Parametric Distribution

In the previous section, we saw that if the class densities $P(\mathbf{x} \mid C_i)$ are Gaussian and share a common *pooled* covariance matrix, then the discriminant function is linear.

$$g_i(\mathbf{x}) = \mathbf{w}_i^T \mathbf{x} + w_{i0}$$

where the parameters $\mathbf{w}_i$ and w_{i0} can be derived analytically by using the Gaussian formula. Now imagine there are two classes, where we define $y = P(C_1 \mid \mathbf{x})$ and $1 - y = P(C_2 \mid \mathbf{x})$. In classification, we choose C_1 if $y > 0.5$, and C_2 otherwise. This is equivalent to:

$$\begin{aligned} y &> 0.5 \\ \frac{y}{1-y} &> \frac{0.5}{1-0.5} = 1 \\ \log \frac{y}{1-y} &> \log 1 = 0 \end{aligned}$$

This last equivalence $\log\left(y/(1-y)\right)$ is known as the **logit transformation** or the *log odds* of y. The logit is *linear* in the case of two Gaussian (normal) classes if they share a common covariance matrix.

$$\begin{aligned} \text{logit}(P(C_1 \mid \mathbf{x})) &= \log \frac{P(C_1 \mid \mathbf{x})}{1 - P(C_1 \mid \mathbf{x})} = \log \frac{P(C_1 \mid \mathbf{x})}{P(C_2 \mid \mathbf{x})} \\ &= \log \frac{P(\mathbf{x} \mid C_1) P(C_1)}{P(\mathbf{x} \mid C_2) P(C_2)} \\ &= \log \frac{P(\mathbf{x} \mid C_1)}{P(\mathbf{x} \mid C_2)} + \log \frac{P(C_1)}{P(C_2)} \\ &= \log \frac{\frac{1}{(2\pi)^{d/2}|\mathbf{\Sigma}|^{1/2}} \exp\left[-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu}_1)^T \mathbf{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu}_1)\right]}{\frac{1}{(2\pi)^{d/2}|\mathbf{\Sigma}|^{1/2}} \exp\left[-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu}_2)^T \mathbf{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu}_2)\right]} + \log \frac{P(C_1)}{P(C_2)} \\ &= \log \frac{\exp\left[-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu}_1)^T \mathbf{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu}_1)\right]}{\exp\left[-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu}_2)^T \mathbf{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu}_2)\right]} + \log \frac{P(C_1)}{P(C_2)} \\ &= -\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu}_1)^T \mathbf{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu}_1) + \frac{1}{2}(\mathbf{x}-\boldsymbol{\mu}_2)^T \mathbf{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu}_2) + \log \frac{P(C_1)}{P(C_2)} \\ &= \mathbf{w}^T \mathbf{x} + w_0 \end{aligned}$$

where

$$\begin{aligned} \mathbf{w} &= \mathbf{\Sigma}^{-1}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2) \\ w_0 &= -\frac{1}{2}(\boldsymbol{\mu}_1 + \boldsymbol{\mu}_2)^T \mathbf{\Sigma}^{-1}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2) + \log \frac{P(C_1)}{P(C_2)} \end{aligned}$$

The inverse of *logit* is the **logistic function** or the **sigmoid function**. See figure 5.2.

$$P(C_1 \mid \mathbf{x}) = \text{sigmoid}(\mathbf{w}^T \mathbf{x} + w_0) = \frac{1}{1 + \exp\left[-(\mathbf{w}^T \mathbf{x} + w_0)\right]}$$

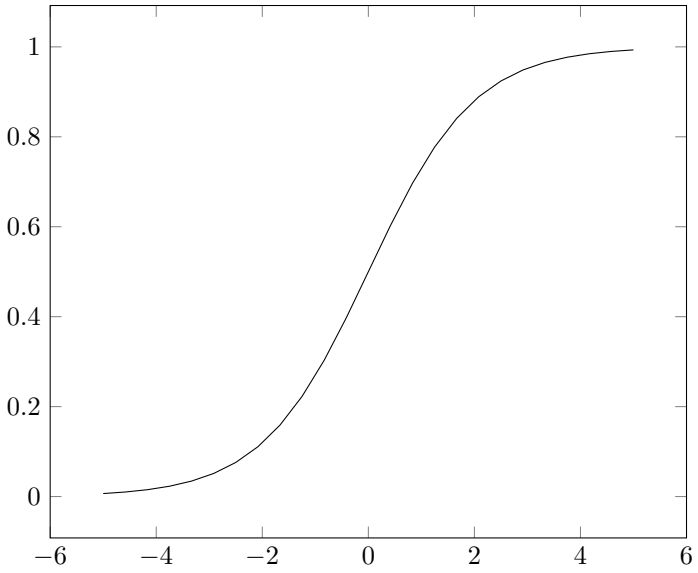


Figure 5.2: Sigmoid or logistic function

We can derive the inverse of *logit* by setting $y = \text{logit}(x)$ and solving for x, as follows:

$$\begin{aligned}
y &= \log \frac{x}{1-x} \\
\exp y &= \frac{x}{1-x} \\
1 + \exp y &= \frac{1-x}{1-x} + \frac{x}{1-x} \\
1 + \exp y &= \frac{1}{1-x} \\
(1 + \exp y)(1-x) &= 1 \\
1 - x &= \frac{1}{1+\exp y} \\
x &= 1 - \frac{1}{1+\exp y} \\
x &= \frac{1+\exp y}{1+\exp y} - \frac{1}{1+\exp y} \\
x &= \frac{\exp y}{1+\exp y} \\
x &= \frac{1}{\frac{1}{\exp y} + \frac{\exp y}{\exp y}} \\
x &= \frac{1}{1+\exp[-y]}
\end{aligned}$$

We can utilise the logit function and the sigmoid function as follows. During training, we estimate $\mathbf{m}_1$, $\mathbf{m}_2$, and $\mathbf{S}$, and use these to derive $\mathbf{w}$ and w_0. Then, during testing, we can use the sigmoid function to get posterior probabilities, instead of raw discriminant values:

$$\text{Choose } C_1 \text{ if } \text{sigmoid}(\mathbf{w}^T\mathbf{x} + w_0) > 0.5$$

5.5 Gradient Descent

In likelihood-based classification, we estimate the parameters via maximum likelihood. In discriminant-based classification, the parameters are those of the discriminant, i.e., $\mathbf{w}$, w_0, etc. These are optimised to minimise the classification error on the training set. When $\mathbf{w}$ denotes the set of parameters and $E(\mathbf{w} \mid \mathcal{X})$ is the error with parameters $\mathbf{w}$ on the given training set $\mathcal{X}$, the task is to find

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} E(\mathbf{w} \mid \mathcal{X})$$

Often, there is no analytic solution, which requires us to use *iterative optimisation methods*. The most commonly employed method is **gradient descent**. Given that $E(\mathbf{w})$ is a differentiable function of a vector of parameter variables, we compute the *gradient vector*, which is composed of the partial derivatives

$$\nabla_{\mathbf{w}} E = \left[\frac{\partial E}{\partial w_1} \quad \frac{\partial E}{\partial w_2} \quad \cdots \quad \frac{\partial E}{\partial w_d} \right]^T$$

The gradient descent procedure to minimise the error $E(\mathbf{w})$ starts with random $\mathbf{w}$, and at each step, updates $\mathbf{w}$ in the *opposite direction* of the gradient:

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}, \quad \forall i$$
$$w_i = w_i + \Delta w_i$$

where η is the **stepsize** or **learning rate**, and determines how much to move in the direction opposite to the gradient. In contrast, **gradient ascent** is the procedure to maximise a function and goes in the direction of the gradient. When the derivative is 0, we know that we hit a local extrema, i.e., either a minimum or a maximum. This is the point where the procedure terminates. The use of a good value for the learning rate η is critical, because a small value leads to slow *convergence* to a local minimum, whereas a large value leads to oscillations or even *divergence*.

5.6 Logistic Discrimination

5.6.1 Two Classes

In **logistic discrimination** (or *logistic regression*), we do not model the conditional densities for each class, $P(\mathbf{x} \mid C_i)$, but instead, their *ratios*. We assume that for two classes, their log likelihood ratio (logit) is linear:

$$\text{logit}(P(\mathbf{x} \mid C_1)) = \log \frac{P(\mathbf{x} \mid C_1)}{1 - P(\mathbf{x} \mid C_1)} = \log \frac{P(\mathbf{x} \mid C_1)}{P(\mathbf{x} \mid C_2)} = \mathbf{w}^T \mathbf{x} + w_0^o$$

This only holds when the conditional densities for the classes are Gaussian. Fortunately, logistic discrimination has a much wider scope of applicability, i.e., $\mathbf{x}$ may be composed of discrete attributes or may be a mixture of continuous and discrete attributes. We already

saw that using Bayes' rule, we get:

$$\text{logit}(P(C_1 \mid \mathbf{x})) = \log \frac{P(C_1 \mid \mathbf{x})}{1 - P(C_1 \mid \mathbf{x})} = \mathbf{w}^T \mathbf{x} + w_0$$

where

$$w_0 = w_0^o + \log \frac{P(C_1)}{P(C_2)}$$

defines both a constant parameter w_0^o as well as the log ratio of the class priors.

By rearranging the terms, we get the *sigmoid* function again (see section 5.4 for this derivation):

$$y = \hat{P}(C_1 \mid \mathbf{x}) = \frac{1}{1 + \exp[-(\mathbf{w}^T \mathbf{x} + w_0)]}$$

Recall that sigmoid allows for the transformation of a discriminant value into the range of probabilities. $\hat{P}(C_1 \mid \mathbf{x})$ will be our estimator for $P(C_1 \mid \mathbf{x})$.

Parameter Learning We are going to learn $\mathbf{w}$ and w_0. Given a sample of two classes, $\mathcal{X} = \{\mathbf{x}^t, r^t\}$, where $r^t = 1$ if $\mathbf{x} \in C_1$ and $r^t = 0$ if $\mathbf{x} \in C_2$. We make the assumption that, given a feature vector $\mathbf{x}^t$, the value of $r^t \in \{0, 1\}$ is a Bernoulli variable, with probability $y = P(C_1 \mid \mathbf{x}^t)$. Thus, we can write:

$$r^t \mid \mathbf{x}^t \sim \text{Bernoulli}(y^t)$$

where we see the difference between likelihood-based methods that model $P(\mathbf{x} \mid C_1)$, whereas in discriminant-based methods, we model the value $r^t \mid \mathbf{x}^t$ directly. Using the binomial distribution, the sample likelihood, $l(\cdot)$, is defined as

$$l(\mathbf{w}, w_0 \mid \mathcal{X}) = \prod_t (y^t)^{(r^t)} (1 - y^t)^{(1 - r^t)}$$

The goal is to maximise the likelihood $l(\cdot)$, which we can do by minimising the error function. For the above likelihood, the error is defined as $E = -\log l$, which is the **cross-entropy** or the *negative log likelihood*:

$$\begin{aligned}
E(\mathbf{w}, w_0 \mid \mathcal{X}) &= -\log \left(l(\mathbf{w}, w_0 \mid \mathcal{X}) \right) \\
&= -\log \left(\prod_t (y^t)^{(r^t)} (1 - y^t)^{(1 - r^t)} \right) \\
&= -\sum_t \log \left((y^t)^{(r^t)} (1 - y^t)^{(1 - r^t)} \right) \\
&= -\sum_t \log \left((y^t)^{(r^t)} \right) + \log \left((1 - y^t)^{(1 - r^t)} \right) \\
&= -\sum_t r^t \log \left(y^t \right) + (1 - r^t) \log \left(1 - y^t \right)
\end{aligned}$$

Because sigmoid is a non-linear function, we cannot solve analytically, and need to use *gradient descent* to minimise the cross-entropy. Fortunately, the derivative of the sigmoid

function is trivial. If $y = \text{sigmoid}(x)$, then

$$\frac{\partial y}{\partial x} = \frac{\partial \text{sigmoid}(x)}{\partial x} = \frac{\partial}{\partial x}\frac{1}{1+\exp[-x]} = y(1-y)$$

If we decompose the error functions in pieces, we can define:

$$\begin{aligned} g^t &= \mathbf{w}^T\mathbf{x}^t + w_0 \\ y^t &= \text{sigmoid}(g^t) \\ E &= -\sum_t (r^t)\log\left(y^t\right) + (1-r^t)\log\left(1-y^t\right) \end{aligned}$$

To derive the update equations of gradient descent for the parameters $\mathbf{w}$ and w_0, the chain tells us that

$$\frac{\partial E}{\partial w_j} = \frac{\partial E}{\partial y^t}\frac{\partial y^t}{\partial g^t}\frac{\partial g^t}{\partial w_j}, \qquad \text{and} \qquad \frac{\partial E}{\partial w_0} = \frac{\partial E}{\partial y^t}\frac{\partial y^t}{\partial g^t}\frac{\partial g^t}{\partial w_0}$$

In other words, if we derive the derivatives of the three chain rule components, and multiply them together, we get the derivative with respect to parameter w_j. Recall that the derivative of $\log x$ is $1/x$, where we use the natural logarithm.

$$\begin{aligned} \frac{\partial E}{\partial y^t} &= -\sum_t \left(r^t\frac{1}{y^t} + (1-r^t)\frac{1}{1-y^t}\cdot -1\right) = -\sum_t \left(\frac{r^t}{y^t} - \frac{1-r^t}{1-y^t}\right) \\ \frac{\partial y^t}{\partial g^t} &= y^t(1-y^t) \\ \frac{\partial g^t}{\partial w_j} &= \frac{\partial}{\partial w_j}\mathbf{w}^T\mathbf{x}^t + w_0 = \frac{\partial}{\partial w_j}w_0 + \sum_j^d w_j x_j^t = x_j^t \\ \frac{\partial g^t}{\partial w_0} &= \frac{\partial}{\partial w_j}\mathbf{w}^T\mathbf{x}^t + w_0 = 1 \end{aligned}$$

Combined together, we get the update equations:

$$\begin{aligned} \Delta w_j &= -\eta\frac{\partial E}{\partial w_j} = \eta\sum_t \left(\frac{r^t}{y^t} - \frac{1-r^t}{1-y^t}\right) y^t(1-y^t)x_j^t \\ &= \eta\sum_t \left(r^t(1-y^t)x_j^t - (1-r^t)y^t x_j^t\right) \\ &= \eta\sum_t \left(r^t - r^t y^t - y^t + y^t r^t\right) x_j^t \\ &= \eta\sum_t \left(r^t - y^t\right) x_j^t, \qquad j = 1,\ldots,d \\ \Delta w_0 &= -\eta\frac{\partial E}{\partial w_0} = \eta\sum_t \left(r^t - y^t\right) \end{aligned}$$

Because we take a weighted sum of our inputs, it is a good idea to apply z-normalisation, so that all inputs are centered, have mean zero, and unit variance.

It is also best to initialise w_j with random values that are close to zero, for example, drawn from a uniform distribution in the range $[-0.01, 0.01]$. The reason is if the weighted sum is large in magnitude, it may saturate the sigmoid. That is, in figure 5.2, notice that if the weights are close to 0, the derivative at this region is always non-zero, and thus an

update can be made. But if the weights are very large (either positive or negative), then the sigmoid will shift them to the far most right or the far most left, depending on the sign. This results in a derivative that is almost 0, and therefore, the weights might not get updated.

We continue training until the number of misclassifications does not decrease, which will be 0 if the classes are linearly separable. Stopping early is also possible and a form of regularisation.

5.6.2 Multiple Classes

To generalise to $K > 2$ classes, we take a reference class, C_K, and assume, just like before, that

$$\log \frac{P(\mathbf{x} \mid C_i)}{P(\mathbf{x} \mid C_K)} = \mathbf{w}_i^T \mathbf{x} + w_{i0}$$

Here, using the logit formula the term $P(\mathbf{x} \mid C_K)$ is actually $1 - P(\mathbf{x} \mid C_i)$. Now, we can establish the class densities via:

$$\frac{P(C_i \mid \mathbf{x})}{P(C_K \mid \mathbf{x})} = \exp[\mathbf{w}_i^T \mathbf{x} + w_{i0}]$$

where $w_{i0}^o = w_{i0} + \log P(C_i)/P(C_K)$. The ratio of the sum of K classes, C_i, where $i \neq K$, and C_K is the same as the ratio of the the sum of $K - 1$ classes and C_K:

$$\sum_{i=1}^{K-1} \frac{P(C_i \mid \mathbf{x})}{P(C_K \mid \mathbf{x})} = \sum_{i=1}^{K-1} \exp[\mathbf{w}_i^T \mathbf{x} + w_{i0}] = \frac{1 - P(C_K \mid \mathbf{x})}{P(C_K \mid \mathbf{x})}$$

The reason why the last identity holds is because summing over all conditional class probabilities $P(C_i \mid \mathbf{x})$ must result to 1.

$$\sum_{i=1}^{K} P(C_i \mid \mathbf{x}) = 1 \tag{5.1}$$

$$P(C_K \mid \mathbf{x}) + \sum_{i=1}^{K-1} P(C_i \mid \mathbf{x}) = 1 \tag{5.2}$$

$$\sum_{i=1}^{K-1} P(C_i \mid \mathbf{x}) = 1 - P(C_K \mid \mathbf{x}) \tag{5.3}$$

With this, we can derive the following useful identity:

$$\begin{aligned}
\frac{1 - P(C_K \mid \mathbf{x})}{P(C_K \mid \mathbf{x})} &= \sum_{i=1}^{K-1} \exp[\mathbf{w}_i^T \mathbf{x} + w_{i0}] \\
\frac{1}{P(C_K \mid \mathbf{x})} - \frac{P(C_K \mid \mathbf{x})}{P(C_K \mid \mathbf{x})} &= \sum_{i=1}^{K-1} \exp[\mathbf{w}_i^T \mathbf{x} + w_{i0}] \\
\frac{1}{P(C_K \mid \mathbf{x})} &= 1 + \sum_{i=1}^{K-1} \exp[\mathbf{w}_i^T \mathbf{x} + w_{i0}] \\
P(C_K \mid \mathbf{x}) &= \frac{1}{1 + \sum_{i=1}^{K-1} \exp[\mathbf{w}_i^T \mathbf{x} + w_{i0}]}
\end{aligned}$$

With this, we can transform our original assumption into:

$$\frac{P(C_i \mid \mathbf{x})}{P(C_K \mid \mathbf{x})} = \exp[\mathbf{w}_i^T \mathbf{x} + w_{i0}]$$

$$\frac{P(C_i \mid \mathbf{x})}{\frac{1}{1 + \sum_{j=1}^{K-1} \exp[\mathbf{w}_j^T \mathbf{x} + w_{j0}]}} = \exp[\mathbf{w}_i^T \mathbf{x} + w_{i0}]$$

$$P(C_i \mid \mathbf{x})(1 + \sum_{j=1}^{K-1} \exp[\mathbf{w}_j^T \mathbf{x} + w_{j0}]) = \exp[\mathbf{w}_i^T \mathbf{x} + w_{i0}]$$

$$P(C_i \mid \mathbf{x}) = \frac{\exp[\mathbf{w}_i^T \mathbf{x} + w_{i0}]}{1 + \sum_{j=1}^{K-1} \exp[\mathbf{w}_j^T \mathbf{x} + w_{j0}]}, \qquad i = 1, \ldots, K-1$$

To treat all classes uniformly, we can write:

$$Y_i = \hat{P}(C_i \mid \mathbf{x}) = \frac{\exp[\mathbf{w}_i^T \mathbf{x} + w_{i0}]}{\sum_{j=1}^{K} \exp[\mathbf{w}_j^T \mathbf{x} + w_{j0}]}$$

which is called the **softmax function**. The reason why we can rewrite this is because this is the generalised case. Recall that for the case of $K = 2$, we had the logistic function (sigmoid), and given two exponents z_1 and z_2 that can be substituted for the discriminant output of two classes, we simply fixed one variable, say $z_2 = 0$, so that $e^0 = 1$ and the other variable is able to vary, $z_1 = x$. This is the reason why sigmoid can be written as $\exp[z_1]/(\sum_{i=1}^{2} \exp[z_i]) = \exp[x]/(1 + \exp[x])$ and why $P(C_2 \mid \mathbf{x}) = 1 - P(C_1 \mid \mathbf{x})$.

Softmax ensures that if the weighted sum for one class is sufficiently larger than for the others (after exponentiation and normalisation), its corresponding y_i value will be close to 1. At the same time, the other values will be close to 0. Therefore, softmax is like taking the maximum, except that is is *differentiable* (hence the name softmax). Softmax guarantees $\sum_i y_i = 1$.

Parameter Learning To learn the parameters, each sample point is a *multinomial trial* with one draw. That is, each trial leads to a 1 for exactly one class and 0 otherwise.

$$\mathbf{r}^t \mid \mathbf{x} \sim \text{Multinomial}_K(1, \mathbf{y}^t), \qquad \text{where } y_i^t = P(C_i \mid \mathbf{x}^\mathbf{t})$$

The sample likelihood is then

$$l(\{\mathbf{w}_i, w_{i0}\} \mid \mathcal{X}) = \prod_t \prod_i (y_i^t)^{r_i^t}$$

Again, to maximise the likelihood, we need to minimise the error, i.e., the negative log likelihood. The error becomes again cross-entropy.

$$\begin{aligned} E(\{\mathbf{w}_i, w_{i0}\} \mid \mathcal{X}) &= -\log \left(\prod_t \prod_i (y_i^t)^{r_i^t} \right) \\ &= -\sum_t \sum_i \log \left((y_i^t)^{r_i^t} \right) \end{aligned}$$

$$= -\sum_t \sum_i r_i^t \log\left(y_i^t\right)$$

To use gradient descent, we need to know how to differentiate $y_i = \exp[a_i]/\sum_j \exp[a_j]$. Its derivative is the following.

$$\frac{\partial y_i}{\partial a_j} = y_i(\delta_{ij} - y_j)$$

where δ_{ij} is the **Kronecker delta**, which is 1 if $i = j$ and 0 if $i \neq j$. Again, using gradient descent, we can formulate the components of the chain rule.

$$\frac{\partial E}{\partial \mathbf{w}_j} = \frac{\partial E}{\partial y_i^t}\frac{\partial y_i^t}{\partial g_j^t}\frac{\partial g_j^t}{\partial w_j}, \qquad \text{and} \qquad \frac{\partial E}{\partial w_0} = \frac{\partial E}{\partial y_i^t}\frac{\partial y_i^t}{\partial g_j^t}\frac{\partial g_j^t}{\partial w_0}$$

Thus, the update equations are the following:

$$\Delta \mathbf{w}_j = \eta \sum_t \sum_i \left(\frac{r_i^t}{y_i^t}\right) y_i^t(\delta_{ij} - y_j^t)\mathbf{x}^t \tag{5.1}$$

$$= \eta \sum_t \sum_i r_i^t(\delta_{ij} - y_j^t)\mathbf{x}^t \tag{5.2}$$

$$= \eta \sum_t \left(\sum_i r_i^t \delta_{ij} - y_j^t \sum_i r_i^t\right)\mathbf{x}^t \tag{5.3}$$

$$= \eta \sum_t (r_j^t - y_j^t \cdot 1)\mathbf{x}^t \tag{5.4}$$

$$= \eta \sum_t (r_j^t - y_j^t)\mathbf{x}^t \tag{5.5}$$

$$\Delta w_0 = \eta \sum_t (r_j^t - y_j^t) \tag{5.6}$$

Note that in 5.4, two simplifications are at play: $\sum_i r_i = 1$ and $\sum_i r_i \delta_{ij} = r_j$. The latter is trivial because the Kronecker delta allows us to reduce the entire sum, except for the case where $i = j$. Only then is the delta equal to 1, and we keep r_j.

The normalisation in softmax also affects all $\mathbf{w}_j$ and w_{j0}. Not only those $\mathbf{x}^t \in C_j$, but also $\mathbf{x}^t \in C_i, j \neq i$.

When the data is normally distributed, the logistic discriminant has a comparable error rate to the parametric, Gaussian-based linear discriminant. As long a classes are linearly separable, logistic regression can be used.

As mentioned earlier, any specified function parameterised by the basic variables can be included as $\mathbf{x}$-variates. For example, we can write the discriminant as a linear sum of nonlinear basis functions

$$\log \frac{P(\mathbf{x} \mid C_i)}{P(\mathbf{x} \mid C_K)} = \mathbf{w}_i^T \mathbf{\Phi}(\mathbf{x}) + w_{i0}$$

where $\mathbf{\Phi}(\cdot)$ are the basis functions (potentials), which can be viewed as transformed variables. In neural network terminology, this is a **multilayer perceptron**.

5.7 Learning to Rank

Ranking is different from classification and regression, as we are tasked to put two or more instances in the correct order. For example, $\mathbf{x}^u$ and $\mathbf{x}^v$ represent two movies. If the user has

enjoyed u more than v, then we label their relationship as $r^u \prec r^v$. Instead of a discriminant or regression function, we learn a **score function** $g(\mathbf{x} \mid \theta)$. The idea is to give a higher score to $\mathbf{u}$, so that $g(\mathbf{x}^u \mid \theta) > g(\mathbf{x}^v \mid \theta)$. After training the model, we can make recommendations to the user.

$$\text{Choose } u \text{ if } g(\mathbf{x}^u \mid \theta) = \max_t g(\mathbf{x}^t \mid \theta)$$

or sometimes, we can recommend a list of the highest K.

Rankers are more natural for humans to use. If movie enjoyment is classified as "enjoyed"/"not enjoyed", it is a classification. If movie enjoyment is rated on a scale of 1 to 10, it is a regression problem. In ranking, neither is done, and instead, users are simply asked for two movies which movie they ranked higher.

For all (u, v) pairs where $r^u \prec r^v$ is defined, the error is given by $g(\mathbf{x}^v \mid \theta) > g(\mathbf{x}^u \mid \theta)$. That is, we expect u to be ranked higher than v, but instead the model $g(\cdot)$ outputs a higher score for v. In general, we do not have a full ordering over all N^2 pairs, but instead, we have a *partial order*. The sum of differences is again the error.

$$E(\mathbf{w} \mid \{r^u, r^v\}) = \sum_{r^u \prec r^v} \max[0, g(\mathbf{x}^v \mid \theta) - g(\mathbf{x}^u \mid \theta)]$$

where the max denotes taking the maximum value between 0 and the difference of scores.

Using a linear model, we have

$$g(\mathbf{x} \mid \mathbf{w}) = \mathbf{w}^T \mathbf{x}$$

Note that w_0 is not used, because it just adds the same value for all inputs and does not affect the ranking. The error then becomes

$$E(\mathbf{w} \mid \{r^u, r^v\}) = \sum_{r^u \prec r^v} \max[0, \mathbf{w}^T(\mathbf{x}^v - \mathbf{x}^u)]$$

Using gradient descent, we make small steps in the opposite gradient with respect to $\mathbf{w}$. For each $r^u \prec r^v$ where $g(\mathbf{x}^v \mid \theta) > g(\mathbf{x}^u \mid \theta)$, we have the update equation:

$$\Delta w_j = -\eta \frac{\partial E}{\partial w_j} - -\eta(\mathbf{x}_j^v - \mathbf{x}_j^u), \qquad j = 1, \ldots, d$$

With this, $\mathbf{w}$ is chosen such that after projecting the instances onto $\mathbf{w}$, the correct orderings are obtained.

6 Nonparametric methods

In parametric methods, we assume a model that is valid for the whole input space. The advantage is that it reduces the problem of estimating a probability density function, discriminant, or regression function, to estimating the values of a small number of parameters. These parameters then describe the data. The disadvantage is that we cannot always make such assumptions.

In **nonparametric estimation**, all we assume is that *similar inputs* have *similar outputs*. This is reasonable, because phenomena in the world are often smooth (e.g., Gaussian) and its functions often change slowly. Similar instances mean similar things. Thus, using a suitable *distance measure*, we find similar instances, and interpolate from them to find the right output. Nonparametric methods are also called **instance-based** or **memory-based** learning algorithms, because they store training instances in a lookup table and (often in a lazy manner) interpolate from them.

6.1 Nonparametric Density Estimation

Given a univariate sample $\mathcal{X} = \{x^t\}_{t=1}^N$, we can define the nonparametric estimator for the *cumulative distribution function*, $F(x)$, which is the *proportion* of sample points that are less than or equal to x:

$$\hat{F}(x) = \frac{\#\{x^t \leq x\}}{N}$$

where $\#\{x^t \leq x\}$ denotes the number of training instances x^t that are less than or equal to the input x. If we take the derivative of the cumulative distribution function, we get the nonparametric estimate of the *probability density function*.

$$\hat{P}(x) = \frac{1}{h}\left[\frac{\#\{x^t \leq x+h\} - \#\{x^t \leq x\}}{N}\right]$$

Here, h is the length of the interval, where we assume that if instances x^t fall within this interval, they are considered "close enough".

6.1.1 Histogram Estimator

The oldest and most popular method is the histogram, where the input space is divided into equal-sized intervals named *bins*. Given a point of origin x_o and a bin width h, the bins are the intervals $[x_o + mh, x_o + (m+1)h]$ for positive and negative integers m. The density estimate is then given as

$$\hat{P}(x) = \frac{\#\{x^t \text{ in the same bin as } x\}}{Nh}$$

The origin moves the histogram, whereas the bin width affects the estimates. Small bins result in a spiky estimate, and large bins result in a smooth estimate. The advantage of the histogram is that once the bin estimates have been computed and stored, we do not need the training set anymore.

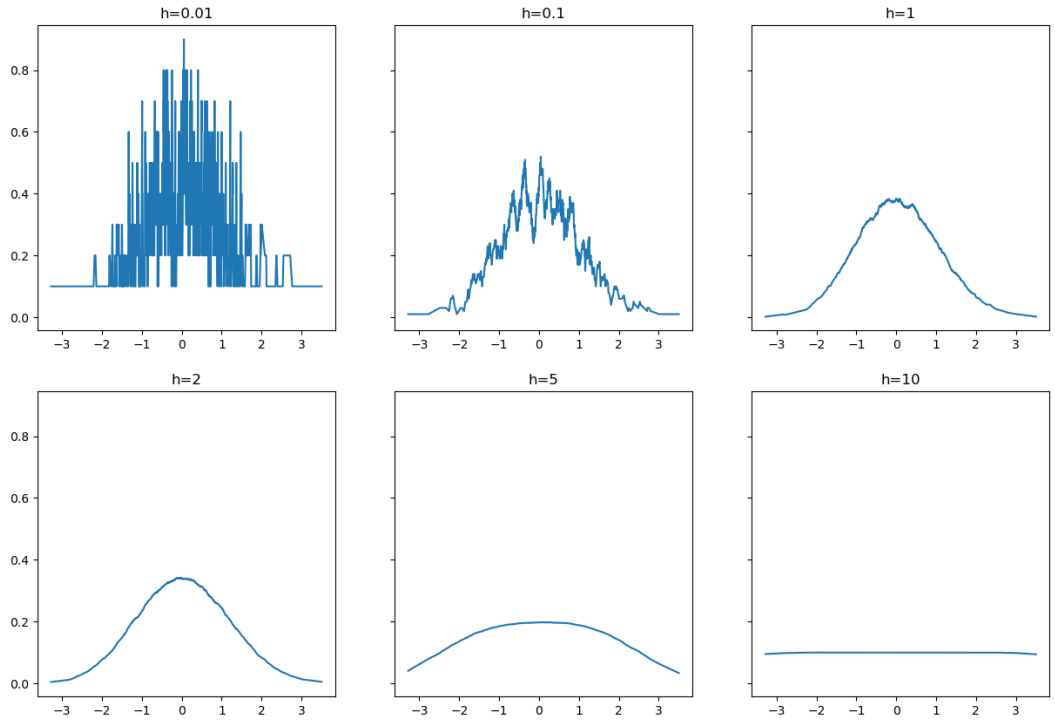


Figure 6.1: Sample of 1000 drawn from univariate Gaussian, $X \sim \mathcal{N}(0, 1)$, and estimated using the naive histogram estimator.

The **naive estimator** frees us from setting an origin. It is defined as

$$\hat{P}(x) = \frac{\#\{x - h/2 < x^t \leq x + h/2\}}{Nh}$$

and is equal to the histogram estimate where x is always at the center of a bin of size h. Figure 6.1 shows the naive estimator with different values for h. The estimator can alternatively be written as

$$\hat{P}(x) = \frac{1}{Nh} \sum_{t=1}^{N} w\left(\frac{x - x^t}{h}\right)$$

with the *weight function* defined as

$$w(u) \begin{cases} 1 & \text{if } |u| < 1/2 \\ 0 & \text{otherwise} \end{cases}$$

In this formulation, each x^t has a symmetric region of size h around it, and contributes 1 if x falls within this region. The nonparametric estimate is just the sum of these *influences* of x^t whose regions include x. Note that since this region is hard (0 or 1), the estimate is discrete and not continuous, i.e., it jumpt at $x^t \pm h/2$.

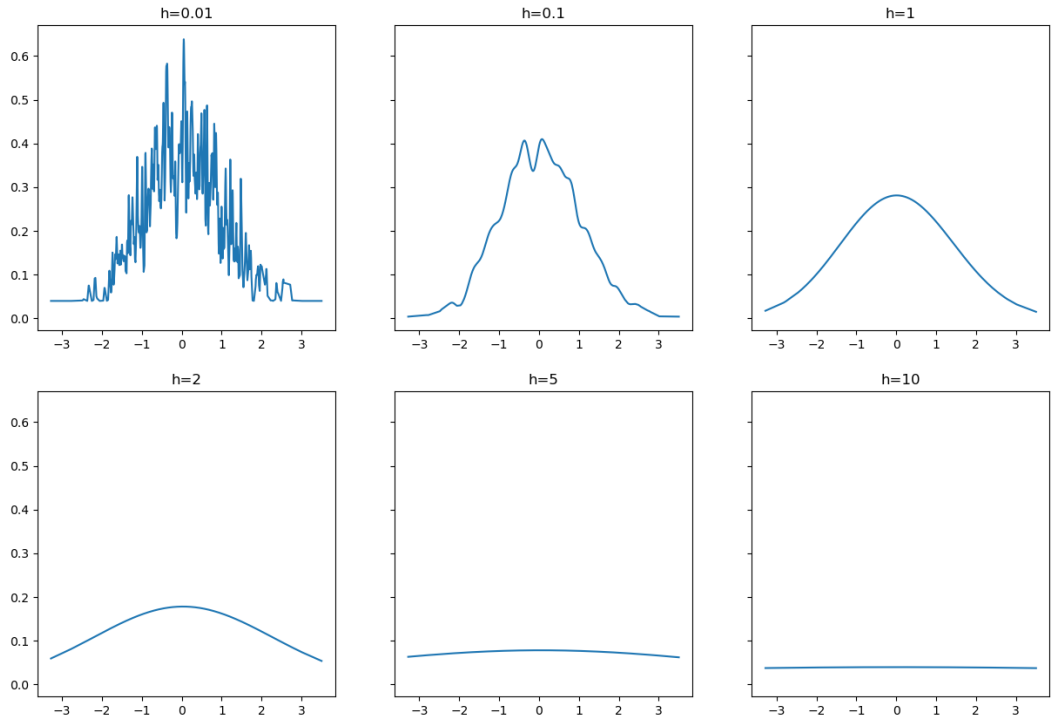


Figure 6.2: Sample of 1000 drawn from univariate Gaussian, $X \sim \mathcal{N}(0, 1)$, and estimated using the kernel estimator.

6.1.2 Kernel Estimator

To get a *smooth estimate*, we use a smooth weight function called a **kernel function**. The most popular is the Gaussian kernel

$$K(u) = \frac{1}{\sqrt{2\pi}} \exp\left[-\frac{u^2}{2}\right]$$

The kernel estimator, also called the **Parzen windows**, is then

$$\hat{P}(x) = \frac{1}{Nh} \sum_{t=1}^{N} K\left(\frac{x - x^t}{h}\right)$$

The kernel function $K(\cdot)$ determines the shape of the influences and the window width determines the width. To simplify $K(\cdot)$, we can return 0 if $|x - x^t| > 3h$. Other kernels also exist, as long as $K(u)$ is maximum for $u = 0$ and decreasing symmetrically as $|u|$ increases.

When h is small, each training instance has a large effect in a small region and no effect on distant points. In other words, a small neighbourhood of training instances will produce an effect. When h is large, there is more overlap between the *kernels* (bumps) and we get a smoother estimate. This is shown in figure 6.2. If $K(\cdot)$ is non-negative everywhere and its integral evaluates to 1, i.e., it conforms to the rules of being a density function, then the estimator $\hat{P}(\cdot)$ will also be a density function.

6.1.3 *K*-Nearest Neighbour Estimator

The nearest neighbour class of estimators adapt the amount of smoothing to the *local density* of the data. The degree of smoothing is now controlled by k, which is the number of neighbours smaller than the sample size N taken into account. We can start by defining a distance function between points a and b, such as the absolute value of the difference, $|a-b|$. Then, for each x, we define

$$d_1(x) \leq d_2(x) \leq \dots d_N(x)$$

to be the distances from x to the points in the sample. This ordering is done in ascending order. For example, $d_1(x)$ is the distance from x to the nearest point x^t, where $x \neq x^t$. Similarly, $d_2(x)$ is the distance to the *second* nearest point, and so on. Therefore, $d_1(x) = \min_t |x - x^t|$, and if we set i to be the index of the nearest neighbour, $i = \arg\min_i |x - x^i|$, then we can define $d_2(x) = \min_{j \neq i} |x - x^j|$, and so forth.

Using broadcast semantics this is a trivial procedure. Let $\mathbf{x}$ be a univariate vector of size $d \times 1$, then $\mathbf{x} - \mathbf{x}^T$ is not normally defines, because there is a mismatch between dimensions. However, if we allow broadcasting, we can cast both vectors to be $d \times d$.

$$abs\left(\begin{bmatrix} — & \mathbf{x} & — \end{bmatrix} - \begin{bmatrix} | \\ \mathbf{x} \\ | \end{bmatrix}\right) \overset{\text{Broadcast}}{\Longrightarrow} abs\left(\begin{bmatrix} — & \mathbf{x} & — \\ — & \mathbf{x} & — \\ & \dots & \\ — & \mathbf{x} & — \end{bmatrix} - \begin{bmatrix} | & | & & | \\ \mathbf{x} & \mathbf{x} & \dots & \mathbf{x} \\ | & | & & | \end{bmatrix}\right)$$

$$= abs\left(\begin{bmatrix} x_1 & x_1 & \dots & x_1 \\ x_2 & x_2 & \dots & x_2 \\ \vdots & \vdots & & \vdots \\ x_d & x_d & \dots & x_d \end{bmatrix} - \begin{bmatrix} x_1 & x_2 & \dots & x_d \\ x_1 & x_2 & \dots & x_d \\ \vdots & \vdots & & \vdots \\ x_1 & x_2 & \dots & x_d \end{bmatrix}\right)$$

$$= abs\left(\begin{bmatrix} 0 & x_1 - x_2 & \dots & x_1 - x_d \\ x_2 - x_1 & 0 & \dots & x_2 - x_d \\ \vdots & \vdots & & \vdots \\ x_d - x_1 & x_d - x_2 & \dots & 0 \end{bmatrix}\right) = \begin{bmatrix} 0 & |x_1 - x_2| & \dots & |x_1 - x_d| \\ |x_2 - x_1| & 0 & \dots & |x_2 - x_d| \\ \vdots & \vdots & & \vdots \\ |x_d - x_1| & |x_d - x_2| & \dots & 0 \end{bmatrix}$$

This computation results in a symmetric matrix with zero diagonal, because $x - x = 0$ and with the distances for each x_i in the columns. If we sort each column in ascending order, we can easily index into its values to obtain specific $d_k(x)$.

The k-nearest neighbour (k-nn) density estimate is

$$\hat{P}(x) = \frac{k}{2Nd_k(x)}$$

This is just like the naive histogram estimator where $h = 2d_k(x)$. The difference is then in the numerator. In the naive estimator case, we count how many instances fall within the range $x - h/2 < x^t \leq x + h/2$, thus making this count a variable amount for each x. Instead,

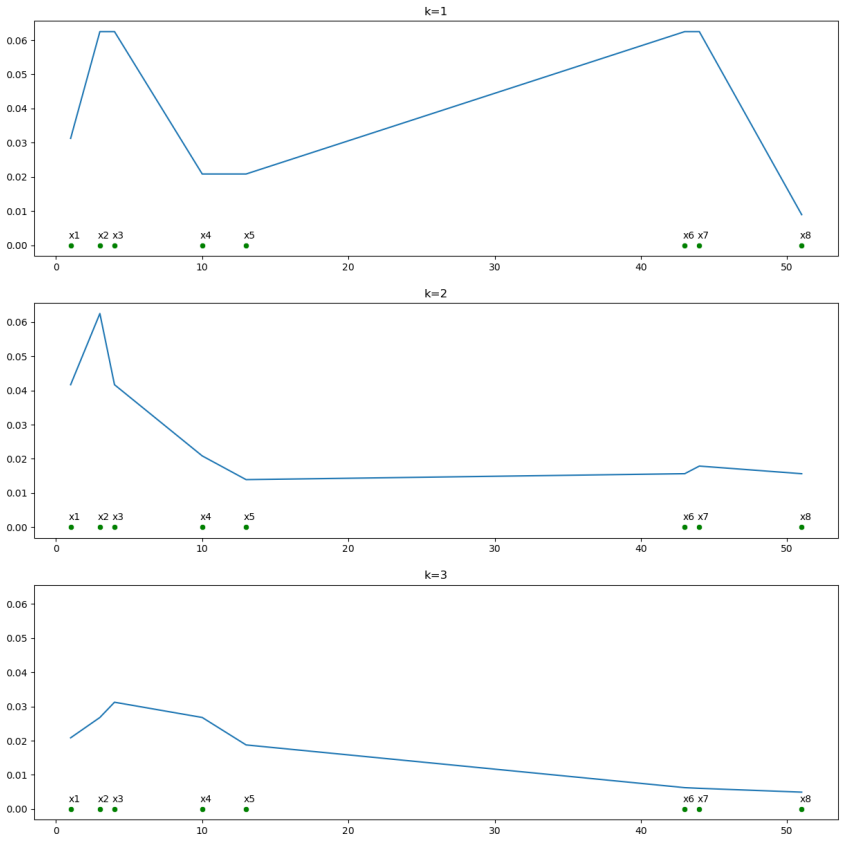


Figure 6.3: k-nearest neighbours estimate for various k values.

in k-nn, we denote the amount of observations that fall within a bin as k, and compute the bin size, i.e., we fix k. Therefore, $h = 2d_k(x)$ is now variable to change, whereas it was fixed in the naive case, and the count in the numerator is now fixed, whereas it was variable in the naive case. If the density is high, the bins have a small width, and if the density is low, the bins have a larger width.

The k-nn estimator is not continuous, since its derivative has discontinuities. It is also not a probability density function because it integrates to ∞ and not 1. Therefore, to get a smooth estimate, we can use a kernel function whose effect decreases with increasing distance.

$$\hat{P}(x) = \frac{1}{2Nd_k(x)} \sum_{t=1}^{N} K\left(\frac{x - x^t}{d_k(x)}\right)$$

Consider figure 6.3, where we see that when $k = 1$, the high densities are at the points where two neighbours are very close (x_2, x_3 and x_6, x_7). When k increases, larger neighbourhoods are given more of the density mass, which results in less spikes, as can be seen when $k = 3$. In general, we see from this simple example that k-nn allows us to estimate the mass of the densities purely by looking at the local neighbourhood of points.

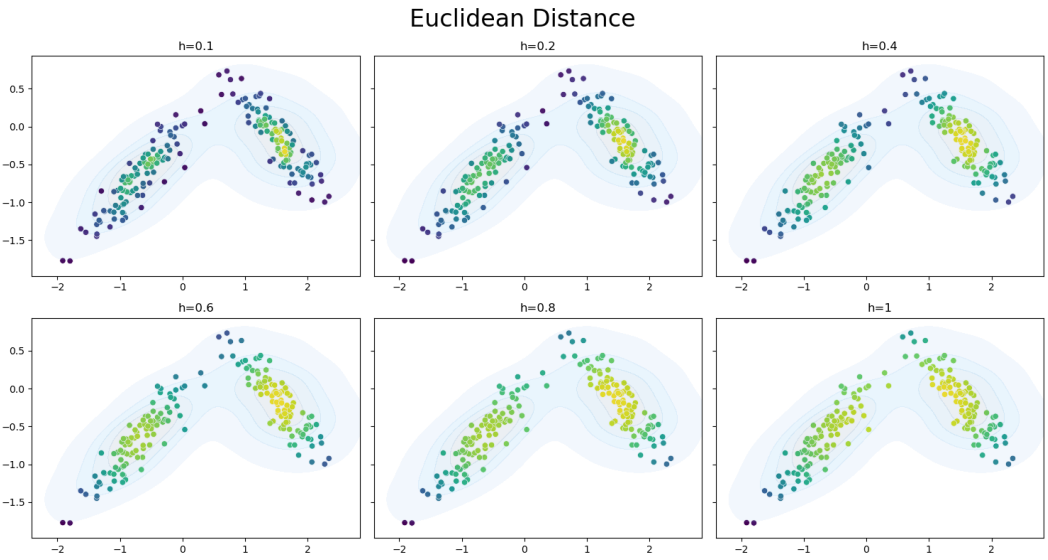


Figure 6.4: Bivariate samples with kernel density estimates using Euclidean distance.

6.2 Generalisation to Multivariate Data

Given a sample of d-dimensional observations $\mathcal{X} = \{\mathbf{x}^t\}_{t=1}^N$, the multivariate kernel density estimator is

$$\hat{P}(\mathbf{x}) = \frac{1}{Nh^d} \sum_{t=1}^{N} K\left(\frac{\mathbf{x} - \mathbf{x}^t}{h}\right)$$

with the requirement that the *kernel function* must integrate to 1, i.e.,

$$\int_{\mathbb{R}^d} K(\mathbf{x}) d\mathbf{x} = 1$$

A natural candidate for this is the multivariate Gaussian kernel:

$$K(\mathbf{u}) = \left(\frac{1}{\sqrt{2\pi}}\right)^d \exp\left[-\frac{||\mathbf{u}||^2}{2}\right]$$

Figure 6.4 shows the kernel density estimation on a bivariate sample for various values of h. Higher values of h imply that a larger region will be regarded as "neighbouring".

Due to the **curse of dimensionality**, one should be careful when applying nonparametric methods to high-dimensional spaces. For example, using a histogram with 10 bins, and a univariate $\mathbf{x}$, there will be 10 bins. For bivariate $\mathbf{X}$, there are two dimensions, thus $10^2 = 100$ bins. For d-variate $\mathbf{X}$, there are d dimensions, and therefore, 10^d bins. The number of bins grow exponentially with the number of dimensions d. In high dimensional spaces with 10^d bins, most of the bins will be empty (unless we have lots of data), and their estimate will be zero. The concept of *closeness* between data points also becomes difficult to elucidate, and we should be careful in choosing h.

The Euclidean norm, $||\mathbf{u}||^2$, used in the kernel, implies that the kernel is scaled equally on all dimensions. Therefore, it is good to first normalise the data to have the same variance before applying the norm. However, the correlations between the dimensions are not taken

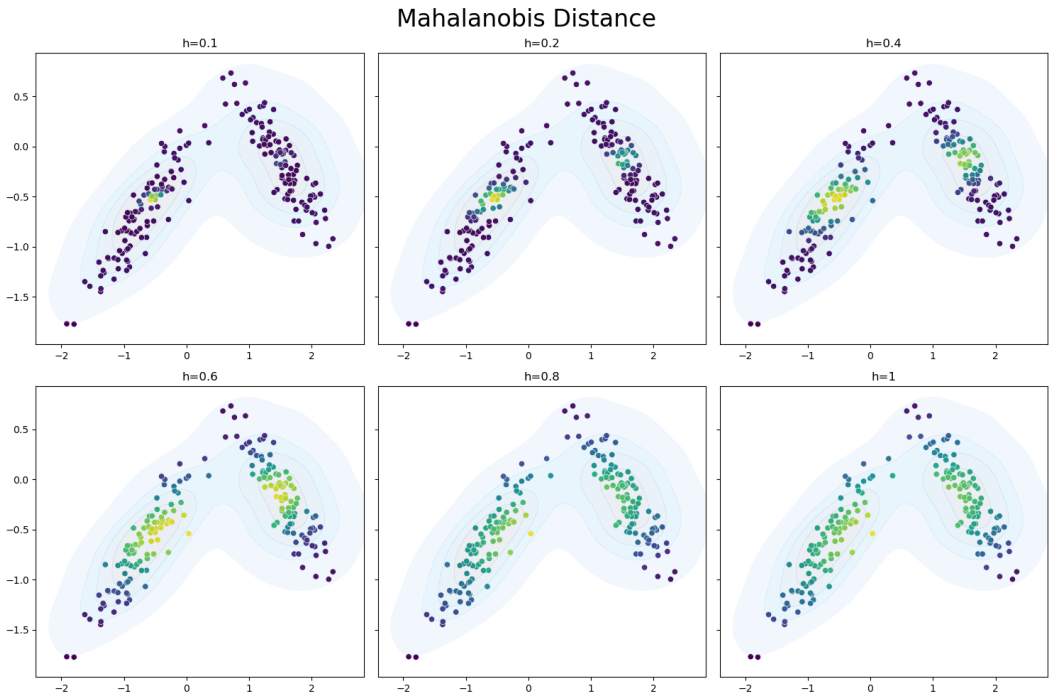


Figure 6.5: Bivariate samples with kernel density estimates using Mahalanobis distance.

into account, and better results are achieved when the kernel has the same form as the underlying distribution.

$$K(\mathbf{u}) = \frac{1}{(2\pi)^{d/2}|\mathbf{S}|^{1/2}} \exp\left[-\frac{1}{2}\mathbf{u}^T\mathbf{S}^{-1}\mathbf{u}\right]$$

where $\mathbf{S}$ is the sample covariance matrix. With this, the Mahalanobis distance is used instead of the Euclidean distance. Figure 6.5 shows the same bivariate sample as in figure 6.4 but now with the Mahalanobis distance.

6.3 Nonparametric Classification

The nonparametric approach to classification entails the estimate of the class-conditional densities, that is, $P(\mathbf{x} \mid C_i)$. The kernel estimator is given as

$$\hat{P}(\mathbf{x} \mid C_i) = \frac{1}{N_i h^d} \sum_{t=1}^{N} K\left(\frac{\mathbf{x} - \mathbf{x}^t}{h}\right) r_i^t$$

where r_i^t is 1 if $\mathbf{x}^t \in C_i$, and 0 otherwise. N_i is the number of labeled instances belonging to C_i. In other words, $N_i = \sum_t r_i^t$. For the **maximum likelihood estimator**, the prior is $\hat{P}(C_i) = N_i/N$, i.e., the fraction of the class i. Then, the discriminant can be written as:

$$\begin{aligned} g_i(\mathbf{x}) &= \hat{P}(C_i \mid \mathbf{x}) = \hat{P}(\mathbf{x} \mid C_i)\hat{P}(C_i) \\ &= \frac{N_i}{N} \cdot \frac{1}{N_i h^d} \sum_{t=1}^{N} K\left(\frac{\mathbf{x} - \mathbf{x}^t}{h}\right) r_i^t \end{aligned}$$

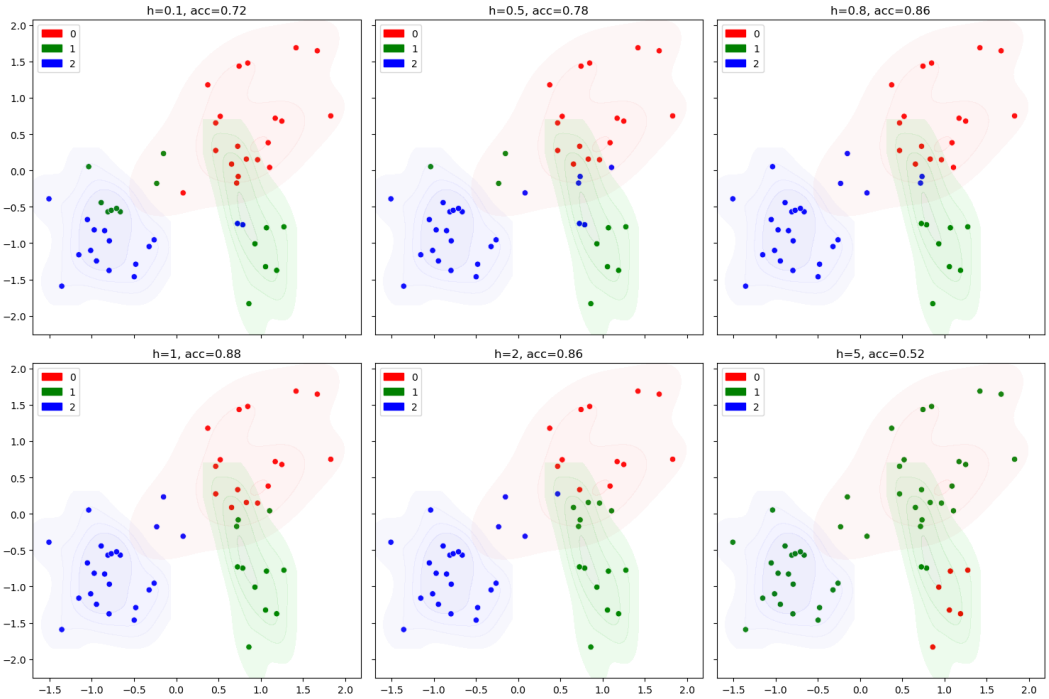


Figure 6.6: Kernel density estimator for classification of bivariate dataset.

$$= \frac{1}{Nh^d} \sum_{t=1}^{N} K\left(\frac{\mathbf{x} - \mathbf{x}^t}{h}\right) r_i^t$$

We then choose for some $\mathbf{x}$ the class for which $g_i(\mathbf{x})$ is the maximum. Each training instance *votes* for its class and has no effect on other classes. The kernel function $K(\cdot)$ provides the weight for the vote. Closer instances typically get more weight.

Consider figure 6.6 where there are 3 classes in the bivariate dataset $\mathcal{X}$. A split is made between train set and test set, and the kernel density estimator is used to classify the instances. For each class, the discriminant $g_i(\mathbf{x})$ is computed by taking the instances labeled i from the train set, appending them with the entire test set, and then estimating the density of the test set specifically. This yields a set of three discriminant outputs where the maximum of each test instance is taken, and its corresponding label is returned. In the figure, the predicted labels are plotted. This analysis is done for various values of h.

For the k-nn estimator, we have

$$\hat{P}(\mathbf{x} \,|\, C_i) = \frac{k_i}{N_i V^k(\mathbf{x})}$$

Here, k_i is the number of neighbours out of the first k nearest neighbours that belong to class C_i and $V^k(\mathbf{x})$ is the *volume* of the d-dimensional sphere centered at $\mathbf{x}$. To compute the class-conditioned densities, this can be simplified and written as:

$$\hat{P}(C_i \,|\, \mathbf{x}) = \frac{\hat{P}(\mathbf{x} \,|\, C_i)\hat{P}(C_i)}{\hat{P}(\mathbf{x})} = \frac{k_i}{k}$$

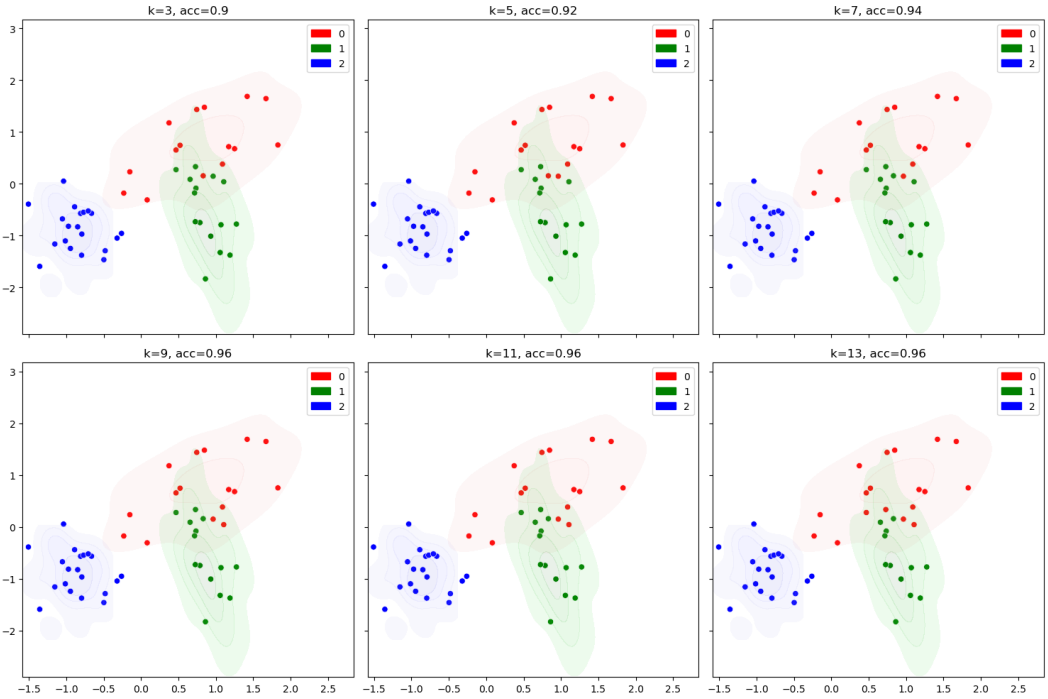


Figure 6.7: k-nn classifier of bivariate dataset.

The k-nn classifier assigns the input $\mathbf{x}$ to the class C_i that has the most examples among the k neighbours. All neighbours have equal weight behind their vote, i.e., their presence as a neighbour, and the class having the maximum number of voters among the first k neighbours is chosen. k is generally taken to be an **odd number** to minimise ties between distances.

6.3.1 Example: *k*-nn classifier

Consider the same sample as in the example of figure 6.6 with 50 test instances, 150 train instances, and three classes (0/1/2). The goal is to classify the 50 test instances with as much accuracy as possible using the k-nn classifier.

The first step is to compute the *absolute differences* between test instances and train instances. This results in a distance matrix $\mathbf{D}$ of size 150×50, where the column vectors $\mathbf{d}_i$ are the distances of each test instance $\mathbf{x}_j$ with respect to every train instance $\mathbf{x}^t \in \mathbf{X}$. Then, we sort the distance matrix $\mathbf{D}$ by taking into account which label corresponds to the distances in $\mathbf{D}$. The easiest way to implement this is to broadcast the vector of labels $\mathbf{y}$ of size 150×1 into the same size as $\mathbf{D}$, i.e., 150×50. In this case, this results in a new matrix where the vector $\mathbf{y}$ is copied 50 along its second axis (columns).

$$\mathbf{Y} = \begin{bmatrix} | \\ \mathbf{y} \\ | \end{bmatrix} \overset{\text{Broadcast}}{\Longrightarrow} \overbrace{\begin{bmatrix} | & | & \cdots & | \\ \mathbf{y} & \mathbf{y} & \cdots & \mathbf{y} \\ | & | & \cdots & | \end{bmatrix}}^{50}$$

Instead of sorting $\mathbf{D}$, we can use its sorting indices to sort $\mathbf{Y}$. Then, we take the first k rows of $\mathbf{Y}$, which are the k nearest neighbours for each test instance. Let us denote this subset of rows as $\mathbf{Y}_k$. For each test instance $\mathbf{x}_j$, we can count the values in the corresponding $\mathbf{y}_j \in \mathbf{Y_k}$ column, and choose class C_i for which the count is the maximum.

Figure 6.7 illustrates the results, where several values for k are used. Note that the performance of k-nn is better than the performance of the kernel density estimator.

6.4 Condensed Nearest Neighbour

The time and space complexity of nonparametric methods are proportional to the size of the training set. **Condensing methods** have been proposed to decrease the number of stored instances without degrading performance. The idea is to select the smallest subset $\mathcal{Z}$ of $\mathcal{X}$ such that when $\mathcal{Z}$ is used instead of $\mathcal{X}$, the error does not increase.

The **condensed nearest neighbour** uses a 1-nn nonparametric estimator for classification. 1-nn approximates the discriminant in a piecewise linear manner, and only keeps the instances that define the discriminant, i.e., the boundary regions between classes are kept.

Instances that fall inside a class region can be reduced, because *their* nearest neighbour is also from the same class. Therefore, not keeping these instances will not increase the error (on the train set). The subset where these within-class instances are ignored is called a *consistent subset*, and we want to find the minimal consistent subset.

To find $\mathcal{Z}$, a greedy algorithm can be used. Starting with $\mathcal{Z} = \varnothing$, we pass over all train instances in $\mathcal{X}$ one by one in random order, and check whether they can be correctly classified by the 1-nn using the instances stored in $\mathcal{Z}$. If an instance is misclassified, it is added to $\mathcal{Z}$. If an instance is correctly classified, then $\mathcal{Z}$ is unchanged. We pass over the train set multiple times until no further instances are added. This is a local search algorithm, which means that the order in which training instances are seen may result in different subsets being produced.

6.5 Distance-Based Classification

The k-nearest neighbour classifier assigns instances to the class that is represented the most among its neighbours. The idea is that the more similar the instances, the more likely it is that they belong to the same class.

Most classification algorithms can be reformulated as a distance-based classifier. For the parametric approach with Gaussian classes, the nearest mean classifier classifies C_i if

$$\mathcal{D}(\mathbf{x}, \mathbf{m}_i) = \min_{j=1}^{K} \mathcal{D}(\mathbf{x}, \mathbf{m}_j)$$

If the dimensions are assumed independent and are all in the same scale, the distance measure is the euclidean

$$\mathcal{D}(\mathbf{x}, \mathbf{m}_i) = ||\mathbf{x} - \mathbf{m}_i||$$

In all other cases, it is the **Mahalanobis distance**:

$$\mathcal{D}(\mathbf{x}, \mathbf{m}_i) = (\mathbf{x} - \mathbf{m}_i)^T \mathbf{S}_i^{-1} (\mathbf{x} - \mathbf{m}_i)$$

where $\mathbf{S}_i$ is the covariance matrix of C_i.

In the *nonparametric case*, the distance metric can be more flexible. For each small region (neighbourhood) in the input space, we may have a different distance metric. In other words, we define **locally adaptive distance functions**.

The idea is then to perform **distance learning** by parameterising $\mathcal{D}(\mathbf{x}, \mathbf{x}^t \mid \theta)$. We learn θ from a labeled sample in a supervised manner, and then use k-nn. It is common to use the Mahalanobis distance, as it is the multi-dimensional generalisation of measuring how many standard deviations $\mathbf{x}^t$ is away from the target mean $\mathbf{m}_i$. In other words, it gives the distance between an instance and a distribution:

$$\mathcal{D}(\mathbf{x}, \mathbf{x}^t \mid \mathbf{M}) = (\mathbf{x} - \mathbf{x}^t)^T \mathbf{M} (\mathbf{x} - \mathbf{x}^t)$$

where the parameter is the positive definite matrix $\mathbf{M}$. An example is the **large margin nearest neighbour** algorithm, where $\mathbf{M}$ is estimated so that for all instances in the train set, the distance to a neighbour with the same label is always *less* than the distance to a neighbour with a different label.

Furthermore, when working with high-dimensional input and to avoid overfitting, one approach is to add sparsity constraints on $\mathbf{M}$. Another approach is to use a *low rank approximation* where we factor $\mathbf{M}$ in $\mathbf{M} = \mathbf{L}^T \mathbf{L}$. Here, $\mathbf{L}$ is $k \times d$ with $k < d$. This gives:

$$\begin{aligned}
\mathcal{D}(\mathbf{x}, \mathbf{x}^t \mid \mathbf{M}) &= (\mathbf{x} - \mathbf{x}^t)^T \mathbf{M} (\mathbf{x} - \mathbf{x}^t) = (\mathbf{x} - \mathbf{x}^t)^T \mathbf{L}^T \mathbf{L} (\mathbf{x} - \mathbf{x}^t) \\
&= ((\mathbf{x} - \mathbf{x}^t)^T \mathbf{L}^T)(\mathbf{L}(\mathbf{x} - \mathbf{x}^t)) \\
&= (\mathbf{L}(\mathbf{x} - \mathbf{x}^t))^T (\mathbf{L}(\mathbf{x} - \mathbf{x}^t)) \\
&= (\mathbf{L}\mathbf{x} - \mathbf{L}\mathbf{x}^t)^T (\mathbf{L}\mathbf{x} - \mathbf{L}\mathbf{x}^t) \\
&= (\mathbf{z} - \mathbf{z}^t)^T (\mathbf{z} - \mathbf{z}^t) \\
&= ||\mathbf{z} - \mathbf{z}^t||^2
\end{aligned}$$

where $\mathbf{z} = \mathbf{L}\mathbf{x}$ is the k-dimensional projection of $\mathbf{x}$. From the last derivation, we see that the Mahalanobis distance in the original d-dimensional input space corresponds to the squared Euclidean distance in the new k-dimensional space. Therefore, the *ideal* distance measure is defined as the Euclidean distance in a new space whose fewest dimensions are extracted from the original inputs in the "best" way possible.

In the case of discrete data, the **Hamming distance** can be used that counts the number of non-matching attributes:

$$HD(\mathbf{x}, \mathbf{x}^t) = \sum_{j=1}^{d} 1(x_j \neq x_j^t)$$

7 Decision Trees

A **decision tree** is a hierarchical model for supervised learning, where local regions are identified in a sequence of recursive splits. A decision tree is composed of internal *decision nodes* and *terminal nodes*. In each decision node m, a test function $f_m(\mathbf{x})$ is implemented that outputs a discrete value that corresponds to the possible out-going branches of m. Given an input, at each node, the test function is applied and one of the branches is taken depending on the outcome. This starts at the root and is repeated until a *leaf node* is hit, at which point the value in the leaf is the output of the decision tree.

Decision trees are nonparametric as we do not assume any parametric form for the class densities and the tree structure is not fixed a priori. Each $f_m(\mathbf{x})$ is a discriminant in d-dimensional space that divides the input space in smaller regions that are further subdivided as a path is taken from the root to a leaf. Therefore, we decompose a *complex function* into a series of simple decisions.

The hierarchical placement of decisions allows for fast localisation of some region in the input. For binary decisions, in the best case, each decision eliminates half of the possibilities in the space. Furthermore, decision trees are interpretable, as the tree can be converted to a set of *IF-THEN* rules.

7.1 Univariate Trees

In *univariate trees*, the test of each decision node only uses one of the input dimensions. If the input dimension, X_j, is discrete with n possible values, the test checks the value of X_j against all n values, which results in n-splits from the decision node. For numeric X_j, the values should first be ordered, and then the output must be discretised. For example:

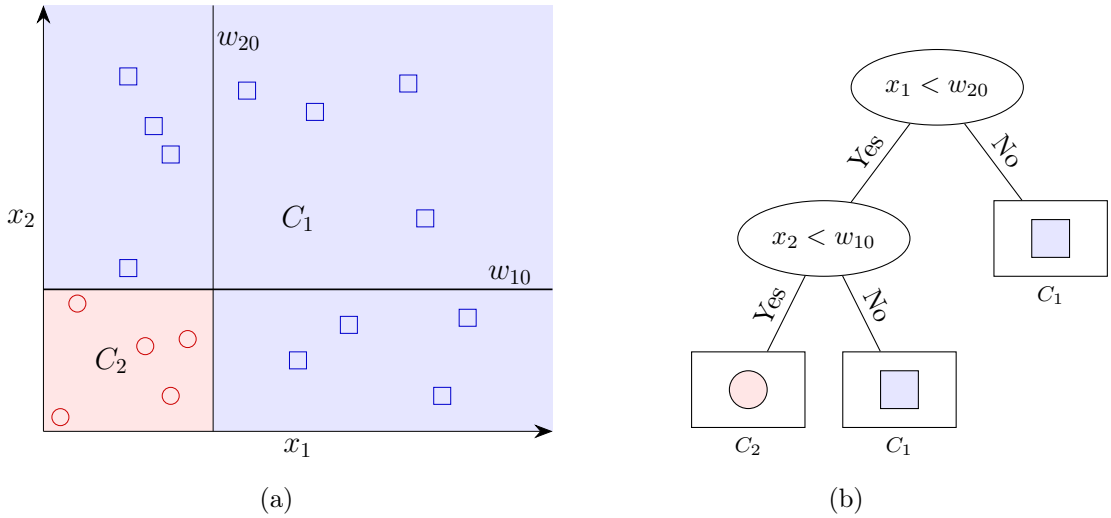
$$\begin{aligned} &X_j \text{ is } \textbf{discrete} && f_{mi}(\mathbf{x}) : X_j = c_i, && \forall c_i \in C_{X_j} \\ &X_j \text{ is } \textbf{numeric} && f_m(\mathbf{x}) : X_j > w_{m0} \end{aligned}$$

where C_{X_j} is the set of all unique values that X_j can take on. Both test functions output a discrete boolean value. w_{m0} is then a suitably chosen threshold value that divides the input space into two:

$$\begin{aligned} L_m &= \{\mathbf{x} \,|\, x_j > w_{m0}\} \\ R_m &= \{\mathbf{x} \,|\, x_j \leq w_{m0}\} \end{aligned}$$

This is a **binary split**, which divides the space in two after each decision node, by which the leaf nodes represent hyper-rectangles in the input space. See figure 7.1 for an example tree and corresponding hyper-rectangular decision boundaries.

The idea is to induce a tree given the training instances that is the smallest among all possible zero-error trees. The tree size is measured by the number of nodes in the tree and the complexity of the decision nodes. Due to tree search being NP-complete, we use a local search procedures based on heuristics that yield *reasonable trees* in *reasonable time*.



(a) (b)

Figure 7.1: (a) Decision regions denoted by the discriminants w_{10} and w_{20}. (b) Corresponding decision tree.

7.1.1 Classification Trees

For classification problems, the *goodness of split* is quantified by the **impurity measure**. A split is considered pure if after the split, for all branches, all instances that have been allocated to that branch belong to the same class. For some decision node m, let us say that N_m is the number of training instances that reached m. Naturally, there are N instances that reached the root node. From the N_m instances, we can count the number of instances that belong to class C_i, and these are denoted by N_m^i. As such, $\sum_i N_m^i = N_m$ must hold. Given the input data $\mathbf{X}$ and the decision node m, we can compute an estimate for the probability of class C_i as follows:

$$\hat{P}(C_i \,|\, \mathbf{X}, m) = p_m^i = \frac{N_m^i}{N_m}$$

Node m is considered **pure** if p_m^i for all i are either 0 (when $N_m^i = 0$) or 1 (when $N_m^i = N_m$). Once a pure node is induces, we stop splitting and add a leaf node labeled with the class for which $p_m^i = 1$ (or the posterior probabilities of all classes).

To measure impurity, we can use the entropy measure, which is plotted in figure 7.2, and is defined as follows:

$$\mathcal{I}_m = -\sum_{i=1}^{K} p_m^i \log_2 p_m^i$$

where $0 \log_2 0 = 0$. **Information entropy** specifies the minimum number of bits needed to encode the class code of an instance. It is derived from the analogy of communication, where the task for the receiver of some information channel is to identify what data was generated by the source through signals. The logarithm base two is used to denote the output in units of bits. If there is uncertainty, i.e., some data was lost during communication, then the entropy will be higher. Analogously, we must then send b bits back to the source to denote that there was loss of information, which is precisely the interpretation of the entropy metric.

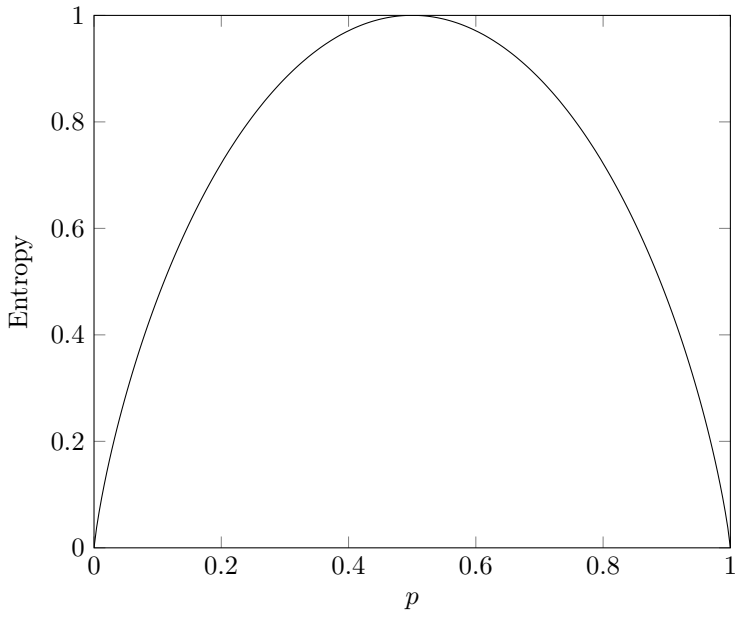


Figure 7.2: Entropy function for a two-class problem

In binary classification, If $p^1 = 1$ and $p^2 = 0$, all instances are of class C_1 and none are of class C_2, and we do not need to send anything back to the source, i.e., lossless information. Therefore, entropy is 0. If, however, $p^1 = p^2 = 0.5$, we need to send a bit to signal equal uncertainty in the two classes, and the entropy is thus 1. When $K > 2$, the same holds and the largest entropy is now $\log_2 K$ when $p^i = 1/K$.

In general, for a two-class problem, $\phi(p, 1-p)$ is a non-negative function measuring impurity of a split if it satisfies the following properties:

- $\phi(1/2, 1/2) \geq \phi(p, 1-p)$, for any $p \in [0, 1]$.
- $\phi(0, 1) = \phi(1, 0) = 0$.
- $\phi(p, 1-p)$ is increasing in p on the interval $[0, 0.5]$ and decreasing in p on the interval $[0.5, 1]$.

For example:

1. **Entropy**: $\phi(p, 1-p) = -p \log_2 p - (1-p) \log_2 (1-p)$
2. **Gini index**: $\phi(p, 1-p) = 2p(1-p)$
3. **Misclassification error**: $\phi(p, 1-p) = 1 - \max(p, 1-p)$

These can be generalised to $K > 2$ classes. There is not a significant difference between these three metrics.

If node m is not pure, then the instances should be split to decrease impurity. We look for the split that minimises impurity *after* the split, because we want to generate the smallest tree. If the subsets after the split are closer to pure, then fewer splits will be needed afterwards. We have no guarantee of finding the smallest decision tree, because this is *locally optimal*.

Let us say that at node m, there are N_m instances, and N_{mj} of those take branch j. For discrete attributes with n values, there are n possible outcomes, i.e., n possible branches. For numeric attributes there are two outcomes ($n = 2$). Either way, counting all the subsets that took a particular branch gives the total subset that arrived at node m, i.e.,

$$\sum_{j=1}^{n} N_{mj} = N_m$$

Intuitively, if N_{mj} took branch j at node m, then we say that N_{mj}^i is the number of instances of N_{mj} that belong to class C_i. Therefore,

$$\sum_{i=1}^{K} N_{mj}^i = N_{mj}$$

Similarly, summing over the class specific counts of each branch results in the total count of the class that arrived at node m, i.e.,

$$\sum_{j=1}^{n} N_{mj}^i = N_m^i$$

Then, given that at node m, the test function returns outcome j, the estimate for the probability of class C_i is:

$$\hat{P}(C_i \,|\, \mathbf{X}, m, j) = p_{mj}^i = \frac{N_{mj}^i}{N_{mj}}$$

And the total impurity after the test function is given as:

$$\mathcal{I}'_m = -\sum_{j=1}^{n} \frac{N_{mj}}{N_m} \sum_{i=1}^{K} p_{mj}^i \log_2 p_{mj}^i$$

To compute this, we need to use the test function that is parameterised by w_{m0}. There are $N_m - 1$ possible values to assign w_{m0}. It is enough to test at halfway between these points. The best split will always be between adjacent points that belong to different classes. In other words, the discriminant should *discriminate* between distinct classes.

Therefore, for all features, and for all splits, we compute impurity and choose the one feature and split that has the minimum entropy. This continues recursively for all branches that are not pure until all are pure (or close to pure). Another way to think about this is that we choose the split that causes the largest decrease in impurity. This is the difference between the impurity of data reaching node m, and the total entropy of reaching its branches after the split.

$$\text{split}_{mi} = \arg\max(\mathcal{I}_m - \mathcal{I}'_m)$$

Overfitting can take place when the data is noisy. Growing a tree until it is pure may lead to variance between training samples. Therefore, we require that tree construction ends when entropy $\mathcal{I}$ is under a certain threshold, i.e., $\mathcal{I} < \theta_I$. Here, θ_I is the complexity parameter, just like h or k in nonparametric estimation. When θ is small, variance is high, and when θ is large, variance is lower which results in smaller trees. Evidently, a small tree has more bias.

7.1.2 Regression Trees

A regression tree is constructed in almost the same way as a classification tree. The exception is that the impurity measure differs. For some node m, we say that $\mathcal{X}_m \subseteq \mathcal{X}$ is the subset reaching m. In other words, for all $\mathbf{x} \in \mathcal{X}_m$, all conditions on the decision nodes from the root to m have been satisfied. We define

$$b_m(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{x} \in \mathcal{X}_m, \text{ i.e., } \mathbf{x} \text{ reaches } m \\ 0 & \text{otherwise} \end{cases}$$

For regression, we can measure the *goodness of split* by the mean square error from the estimated value. Let g_m be the estimated value in node m.

$$E_m = \frac{1}{N_m} \sum_t (r^t - g_m)^2 b_m(\mathbf{x}^t)$$

where $N_m = |\mathcal{X}_m| = \sum_t b_m(\mathbf{x}^t)$.

To estimate the value at node m, i.e., g_m, we use the mean (or median if too much noise) of the required outputs of instances reaching the node.

$$g_m = \frac{\sum_t b_m(\mathbf{x}^t) r^t}{\sum_t b_m(\mathbf{x}^t)}$$

If at a node, the error is acceptable, that is $E_m < \theta_r$, then a leaf node is created with the value of g_m stored in it. If the error is not acceptable, then data reaching node m is split further such that the sum of the errors in the branches is minimum.

Similar to classification trees, we define $\mathcal{X}_{mj}$ as the subset of $\mathcal{X}_m$ that takes branch j, i.e., $\bigcup_{j=1}^n \mathcal{X}_{mj} = \mathcal{X}_m$. We define

$$b_{mj}(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{x} \in \mathcal{X}_{mj}, \text{ i.e., } \mathbf{x} \text{ reaches } m \text{ and takes branch } j \\ 0 & \text{otherwise} \end{cases}$$

Then, g_{mj} is the estimated value in branch j of node m:

$$g_{mj} = \frac{\sum_t b_{mj}(\mathbf{x}^t) r^t}{\sum_t b_{mj}(\mathbf{x}^t)}$$

And the error *after* the split becomes:

$$E'_m = \frac{1}{N_m} \sum_j \sum_t (r^t - g_{mj})^2 b_{mj}(\mathbf{x}^t)$$

Thus, we are looking for a split where the drop in error is the maximum, i.e.,

$$\text{split}_{mi} = \arg\max(E_m - E'_m)$$

Besides the mean squared error, there is also the **worst possible error**, defined as

$$E_m = \max_j \max_t |r^t - g_{mj}| b_{mj}(\mathbf{x}^t)$$

This error function guarantees that the error for any instance is never larger than a given threshold.

Another option is to expand on the definition of g_m. Instead of taking an average of the instances reaching m, which implements a **constant fit** for each leaf node, we can also do a **linear regression fit** over the instances reaching the node:

$$g_m(\mathbf{x}) = w_m^T \mathbf{x} + w_{m0}$$

With this, the estimate at a leaf node depends on $\mathbf{x}$, which results in smaller trees, but at the expense of extra computation at a leaf node.

7.2 Pruning

A node is not split further if the number of training instances that reaches the node is smaller than a certain percentage of the training set, e.g., 5 percent. For example, for some node m, if $N_m < 0.05N$, then we stop splitting regardless of error or impurity. The idea is that any decision that is based on too few instances introduces variance, and therefore, generalisation error. Stopping tree construction early on before it is a full tree is called **prepruning** the tree.

In contrast, **postpruning** often works better than prepruning. In postpruning, we acknowledge that growing a tree is a *greedy* operation, where we generate a decision node, and never backtrack to try out alternative decision nodes. Postpruning is then the procedure to prune unnecessary sub-trees after construction. We grow the tree to full with full purity and zero error, and then find sub-trees that cause overfitting and prune them. From the initial training set, we set aside a **pruning set** (which is distinct from the validation set). For each sub-tree, we replace it with a leaf node by using the training instances covered by the sub-tree. If the leaf node does not perform worse than the sub-tree on the pruning set, we prune the sub-tree and keep the leaf node. In other words, the additional complexity of the sub-tree is not justified.

7.3 Rule Extraction from Trees

A decision tree does its own feature extraction. In univariate trees, only necessary features are used. Some features may not appear in the tree. The features closer to the root are said to be more important globally. Another advantage is **interpretability**, i.e., the decision nodes carry conditions that are simple to understand. Each path from the root to a leaf corresponds to a conjunction of tests (propositions), which can be written as *IF-THEN* rules. The collection of such rules is a **rule base**.

For example, consider figure 7.3, in which a regression tree is seen that uses features x_1, x_2, and x_4, but not x_3. Based on the hierarchy of the tree, we see that x_1 (Age) is the most important feature. The decision tree can also be written down as the following rule base.

$$\text{R1: IF } (\texttt{age} > 38.5) \text{ AND } (\texttt{years\_in\_job} > 2.5) \text{ THEN } y = 0.8$$

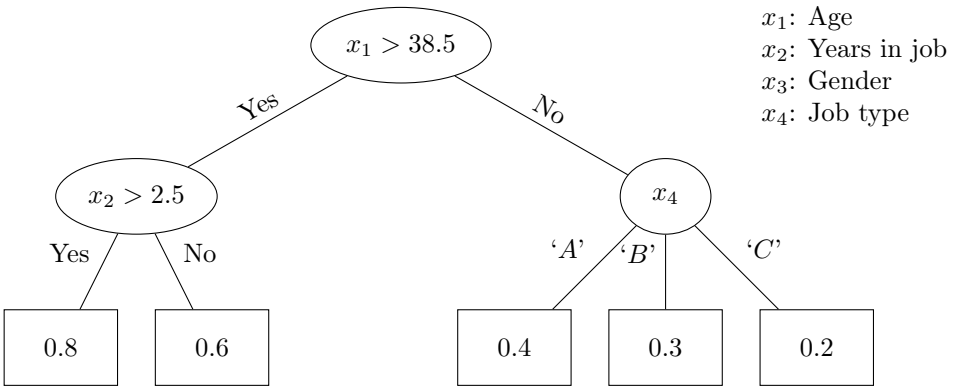


Figure 7.3: Example of regression tree, where each path from the root to a leaf can be written as a set of conditions in the form of a conjunctive rule.

R2: IF (`age` $>$ 38.5) AND (`years_in_job` $\leq$ 2.5) THEN $y = 0.6$

R3: IF (`age` $\leq$ 38.5) AND (`job_type = 'A'`) THEN $y = 0.4$

R4: IF (`age` $\leq$ 38.5) AND (`job_type = 'B'`) THEN $y = 0.3$

R5: IF (`age` $\leq$ 38.5) AND (`job_type = 'C'`) THEN $y = 0.2$

Rule bases allow for **knowledge extraction**. Experts can verify the model learned from data. **Rule support** refers to the percentage of training data covered by that rule.

For classification trees, multiple paths (and thus rules) may have a leaf labeled with the same class. In such cases, the conjunctive rules can be expressed as disjunctive rules. The region of the class in the input space then corresponds to a union of multiple patches.

Rules can also be *pruned* for simplification. Pruning sub-trees corresponds to pruning the individual terms from a set of rules at the same time. However, in pruning rules, we simplify by removing terms from one rule without touching the other rules. For example, in R3, if we see that all whose job_type='A' have their outcomes close to $y = 0.4$, regardless of x_1 (Age), then R3 can be pruned as:

R3: IF (`job_type = 'A'`) THEN $y = 0.4$

After pruning, it may not be possible to restore the rule in its original form.

7.4 Learning Rules from Data

Besides training a decision tree and then converting it to a set of rules, we can also learn rules directly. **Rule induction** works similar to tree induction except that rule induction does a depth-first search and generates one path (rule) at the time, whereas tree induction does a breadth-first search and generates all paths simultaneously.

A rule is said to **cover** an example if the example satisfies all conditions of the rule. Once a rule is *grown* and *pruned*, it is added to the rule base, and all training examples covered by the rule are removed from the training set. This continues until enough rules are added. This procedure is called **sequential covering**. There is an outer loop that adds one rule at a time, and within it, an inner loop that adds one condition at the time. Both

these loops are greedy, and thus cannot guarantee optimal rules.

7.5 Multivariate Trees

Recall that only one input dimension is used at each split of a univariate decision tree. In contrast, in **multivariate trees**, at each decision node, all input dimensions can be used, which increases generality. When all inputs are numeric, a binary linear multivariate node is defined as

$$f_m(\mathbf{x}) : \mathbf{w}_m^T \mathbf{x} + w_{m0} > 0$$

For the weighted sum of this multivariate linear node to take effect, the discrete features should be represented as $0/1$ numeric dummy variables. The above equation defines a hyperplane, and cuts the space in two with arbitrary orientation. the leaf nodes then represent polehedra, since they are the result of arbitrarily many cuts to the original input space. The univariate node is a special case of the multivariate node, where every w_{mj} is 0, except for one. Therefore, it is also a linear discriminant, but one that is orthogonal to the axis x_j.

Going from univariate to multivariate, the decision node becomes more flexible, because it takes the input into account for its test function. For more flexibility, we can add non-linearity with, for example, a quadratic

$$f_m(\mathbf{x}) : \mathbf{x}^T \mathbf{W}_m \mathbf{x} + \mathbf{w}_m^T \mathbf{x} + w_{m0} > 0$$

Other possibilities for non-linear nodes are multi-layer perceptrons or sphere nodes. For learning multivariate decision trees, there are several possibilities:

- CART algorithm, which fine-tunes the weights w_{mj} one by one to decrease impurity.
- Assume that all classes are Gaussian with a common covariance matrix, which implies that linear discriminants are used to separate each class from another.
- Training linear discriminants to minimise classification error.
- Heuristic to group $K > 2$ classes into two super groups, and then binary multivariate trees can be learned. Either by using two-means clustering to group the data into two, and then apply LDA to find the discriminant.

8 Engineering Machine Learning Experiments

This section is concerned with two questions:

1. How can we assess the expected error of a learning algorithm on a problem?
2. Given two learning algorithms, how can we say that one has less error than the other, for a given application?

As discussed before, we need a validation set together with a training set and a test set. However, calculating the error over a validation set *once* may not be enough, because:

- Training and validation sets may be small and may contain *exceptional* instances, such as noise or outliers, which may mislead us.
- The learning method may depend on other random factors affecting generalisation, e.g., initial weights of multi-layer perceptron.

Therefore, we need several runs to average over such sources of randomness. If we train and validate only once, we cannot test for the effect of such factors. The exception is if the learning method is so costly that it can be trained and validated only once. The idea is to generate multiple learners from the same learning algorithm, and test them on multiple validation sets. This results in multiple **validation errors**, i.e., a distribution of validation errors, which we base our evaluation on. For example, we can use this distribution to assess the **expected error** or compare it to the distribution of another learning algorithm.

The following considerations are important:

- Whatever conclusion we draw from analysis is always conditioned on the dataset we are given. **No Free Lunch theorem**, which states that there is no such thing as the *best* learning algorithm.
- The division of the dataset into train/validation is only for testing purposes. Once the tests are complete, and we have a decision on the final learner, we retrain it with *all* labeled data.
- During testing of the model, the validation set effectively becomes part of the training set. Hence the need for the unused and unseen *test set*.
- We compare learning algorithms by their error rates, but there are other metrics: risks, time/space complexity, interpretability, or programmability.

When we train a learner on a training set and test its accuracy on some validation set, and try to draw conclusions, we are actually doing **experimentation**. Statistics defines a methodology to design experiments correctly and analyse the collected data in a manner to be able to extract significant conclusions.

8.1 Guidelines for Machine Learning Experiments

8.1.1 Factors, Response, and Strategy of Experimentation

We do experiments to get information about the process under scrutiny. The process is in this case a learner that is trained on some dataset, and generates an output for a given

input. An **experiment** is a test or a series of tests where we play with the **factors** that affect the output. We, then, observe the changes in the **response** to be able to extract information. For example, identifying (un)important features or optimise accuracy given a feature set.

The aim is to conduct ML experiments, and analyse the results, so that the effect of *chance* can be eliminated and we obtain conclusions that are **statistically significant**. There are two types of factors that a learner depends on to produce an output:

- **Controllable factors**: the learning algorithm, the hyper-parameters, the dataset, and the input representation.
- **Uncontrollable factors**: adds undesired variability, e.g., noise, the particular training subset, randomness in the optimisation process.

The output is used to generate the **response variable**. For example, average classification error on the test set, using some loss function, or other measures, such as precision and recall.

The idea is to find the best configuration of these factors, to generate the best response, or in general, determine the effect of these factors on the response variable. For example, using PCA to reduce dimensions to d, and then using k-nn implies that d and k are the factors.

- **Best-guess strategy**: we start at some setting for the factors d and k that we believe is a good configuration, and test the response there. Then, we tune the factors one at the time and try again, until a response state is reached that we consider good.
- **One factor at a time**: all factors are set to a baseline and we try different levels for one factor while keeping all other factors fixed. The disadvantage is that we assume that there is no *interaction* between the factors.
- **Factorial design**: often called **grid search**, where factors are varied together, i.e., exhaustive search. With F factors at L levels each, this takes $\mathcal{O}(L^F)$ time.

8.1.2 Experimentation Guidelines

The following guidelines are described:

1. **Aim of the study**: the problem needs to be states clearly, in which the objectives are defined. For example,
   - assessing the expected error on a learning algorithm and checking whether the error is lower than some acceptance level;
   - given two or more learning algorithms and one or many datasets, determining which learning algorithm has the least generalisation error on which dataset;
2. **Selection of response variable**: decide what to use as the *quality measure*. Most commonly, the error of misclassification or mean squared error is used.
3. **Choice of factors and levels**: the levels of a factor should be carefully chosen so that good configurations are not missed and unnecessary experimentation is avoided.

4. **Choice of experimental design**: always better to do a factorial design unless we are certain that factors do not interact. Number of resamples is also important so that distribution comparisons are feasible.

5. **Performing the experiment**: before running large factorial experiments, run a few trials to check that all is expected. Save intermediate results, use code from reliable sources, be unbiased, and provide good documentation.

6. **Statistical analysis of the data**: The conclusions from data analysis should not be subjective or due to chance. Use hypothesis testing, and check whether the sample supports the hypothesis.

7. **Conclusions and recommendations**: start with pilot studies to investigate, then invest more resources to continue iterative experiments. Statistical testing never entails that a hypothesis is correct or false, but how much the sample seems to agree with the hypothesis.

8.2 Randomisation and Pairing

8.2.1 Randomisation, Replication, and Blocking

Three basic principles of experimental design:

- **Randomisation**: the order in which the runs are carried out should be randomly determined so that the results are independent.

- **Replication**: for the same configuration of controllable factors, the experiment should be run a number of times to average over the effect of uncontrollable factors, i.e., resampling/cross-validation. this results in an estimate of experimental error.

- **Blocking**: reduce or eliminate variability due to *nuisance factors* that influence the response, but in which we are not interested. For example, comparing learning algorithms should be done on the same resampled subsets, otherwise the differences in accuracy will not only depend on the learning algorithm but also on the different subsets.

8.2.2 Interval Estimation

To conduct hypothesis testing, we need to define **interval estimation**. A *point estimator*, such as the maximum likelihood estimator, specifies a value for a parameter θ. In interval estimation, we specify an interval in which θ lies with a certain degree of confidence. To obtain this interval estimator, we use the probability distribution of the *point estimator*.

As an example, consider the estimation of the mean μ of a normal density from a sample $\mathcal{X} = \{x^t\}_{t=1}^N$. The point estimator of μ is the sample average $m = \sum_t x^t/N$. Because m is the sum of normally distributed random values, it is itself also from a normal distribution, i.e, $m \sim \mathcal{N}(\mu, \sigma^2/N)$.

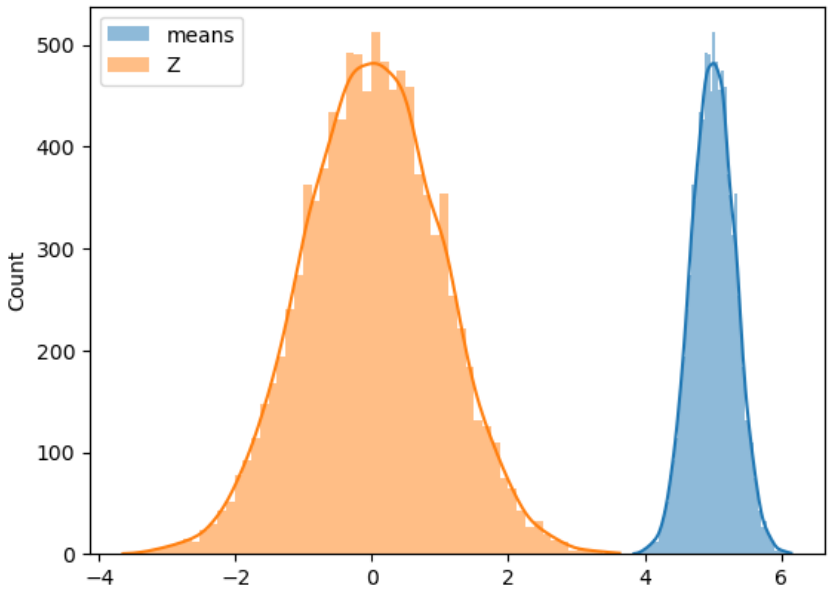


Figure 8.1: Distribution of means $mu \sim \mathcal{N}(\mu, \sigma^2/N)$ in blue and distribution of z-normalised means $(m-\mu)/(\sigma/\sqrt{N}) \sim \mathcal{Z}$ in orange, where $\mu = 5$ and $\sigma = 1$ with 10000 resamples of 10 observations from the normal distribution.

Derivation of Standard Error The variance of the distribution of m is also called the *standard error*, σ^2/N, and is derived as follows.

$$T = \sum_t x^t$$

where T is the total sum of the samples. Then, to compute the variance of T, we can compute the sum of the variances of the sample, which is equal to the sample size times the population variance.

$$\text{Var}(T) = \sum_t \text{Var}(x^t) = N\sigma^2$$

It is then trivial to see that the sample mean is $m = T/N$, and the variance of m is then:

$$\text{Var}(m) = \text{Var}\left(\frac{T}{N}\right) = \frac{1}{N^2}\text{Var}(T) = \frac{1}{N^2}N\sigma^2 = \frac{\sigma^2}{N}$$

Two-sided confidence interval We can define the *unit normal distribution* (or the *z-normalisation*) as

$$\frac{(m-\mu)}{\sigma/\sqrt{N}} \sim \mathcal{Z}$$

See figure 8.1 for a comparison between the distribution of means and the distribution of z-normalised means.

We know that 95% of $\mathcal{Z}$ lies in the interval $(-1.96, 1.96)$, that is, $P(-1.96 < \mathcal{Z} < 1.96) = 0.95$. We can rewrite this as

$$P\left(-1.96 < \mathcal{Z} < 1.96\right) = 0.95$$

$$
\begin{aligned}
&P\left(-1.96 < \frac{(m-\mu)}{\sigma/\sqrt{N}} < 1.96\right) = 0.95 \\
&P\left(-1.96 < \sqrt{N}\frac{(m-\mu)}{\sigma} < 1.96\right) = 0.95 \\
&P\left(-1.96\frac{1}{\sqrt{N}} < \frac{(m-\mu)}{\sigma} < \frac{1}{\sqrt{N}}1.96\right) = 0.95 \\
&P\left(-1.96\frac{\sigma}{\sqrt{N}} < (m-\mu) < 1.96\frac{\sigma}{\sqrt{N}}\right) = 0.95 \\
&P\left(-m - 1.96\frac{\sigma}{\sqrt{N}} < -\mu < -m + 1.96\frac{\sigma}{\sqrt{N}}\right) = 0.95 \\
&P\left(m + 1.96\frac{\sigma}{\sqrt{N}} > \mu > m - 1.96\frac{\sigma}{\sqrt{N}}\right) = 0.95 \\
&P\left(m - 1.96\frac{\sigma}{\sqrt{N}} < \mu < m + 1.96\frac{\sigma}{\sqrt{N}}\right) = 0.95
\end{aligned}
$$

In other words, with 95% confidence, the population mean μ will lie within $1.96\sigma/\sqrt{N}$ units of the sample average m. This is a two-sided confidence interval. If we want more confidence, the interval will get larger. The interval gets smaller, as N increases.

To generalise this for any required confidence, let us denote z_α such that

$$P(\mathcal{Z} > z_\alpha) = \alpha, \qquad 0 < \alpha < 1$$

Because the $\mathcal{Z}$ distribution is symmetric around the mean 0 (see figure 8.1), this means that values at $-z_{\alpha/2}$ are equivalent to values at $z_{1-\alpha/2}$. These two values denote the symmetric tails of the normal distribution, where

$$P(X < -z_{\alpha/2}) = P(X > z_{\alpha/2}) = \alpha/2$$

α denotes the symmetric probability measured from both tails of the distribution. Therefore, $1-\alpha$ is opposite area under the distribution, also known as the level of confidence:

$$P(-z_{\alpha/2} < \mathcal{Z} < z_{\alpha/2}) = 1 - \alpha$$

Substituting for $\mathcal{Z}$, we have the two-sided interval estimation for the true population mean μ.

$$P\left(m - z_{\alpha/2}\frac{\sigma}{\sqrt{N}} < \mu < m + z_{\alpha/2}\frac{\sigma}{\sqrt{N}}\right) = 1 - \alpha$$

One-sided confidence interval Similarly, we know that $P(\mathcal{Z} < 1.64) = 0.95$, which captures the area up until the last 5% of the distribution. Therefore, we have

$$
\begin{aligned}
&P\left(\sqrt{N}\frac{(m-\mu)}{\sigma} < 1.64\right) = 0.95 \\
&P\left(m - 1.64\frac{\sigma}{\sqrt{N}} < \mu\right) = 0.95
\end{aligned}
$$

Thus, the interval $(m - 1.64\sigma/\sqrt{N}, \infty)$ is a 95% *one-sided upper confidence interval* for μ, which defines a lower bound. In other words, the probability that m is smaller than μ with

a difference of $1.64\sigma/\sqrt{N}$ is 0.95.

t-distribution So far, we used σ and thus assumed that the population variance is known. If it is not known, one can use the sample variance instead of σ^2:

$$S^2 = \frac{\sum_t x^t - m}{N-1}$$

We then know that $\sqrt{N}(m-\mu)/S$ is t-distributed with $N-1$ degrees of freedom:

$$\frac{\sqrt{N}(m-\mu)}{S} \sim t_{N-1}$$

Instead of the values from the unit normal $\mathcal{Z}$, we can use the values from the t-distribution for any $\alpha \in (0, 0.5)$. This gives the probability:

$$P\left(m - t_{\alpha/2,N-1}\frac{S}{\sqrt{N}} < \mu < m + t_{\alpha/2,N-1}\frac{S}{\sqrt{N}}\right) = 1-\alpha$$

The t-distribution has larger spread (longer tails) than the unit normal distribution, and generally the interval given by the t-distribution is larger. This is expected because additional uncertainty exists due to the *unknown variance*.

8.3 Statistical Testing

8.3.1 Hypothesis Testing

In hypothesis testing, if the random sample is **consistent** with the hypothesis under consideration, we "fail to reject" the hypothesis, and otherwise, we say that it is "rejected". We do **not** say that the hypothesis is *true* or *false*, but rather that the sample data appears to be consistent with a certain degree of confidence with the hypothesis, or not.

The approach is as follows: we define a *statistic* that obeys a certain distribution **if** the hypothesis is correct. If we then see that the statistic calculated from the sample has very low probability of being drawn from this hypothesised distribution, we reject the hypothesis. And otherwise, we fail to reject it.

Two-sided test Given a sample from a normal distribution with unknown mean μ and known variance σ^2, and we want to test a specific hypothesis about μ, e.g., whether it is equal to some constant μ_0. This hypothesis is then the **null hypothesis** H_0, which is tested against the alternative hypothesis:

$$H_0 : \mu = \mu_0, \qquad H_1 : \mu \neq \mu_0$$

The point estimate of μ is the sample mean m, and we reject H_0 if m is too far from μ_0. Because m represents μ only up to a certain degree, we need an interval of confidence with a **level of significance** α, from which we reject H_0 if μ_0 lies outside the $100(\alpha-1)$ percent confidence interval.

$$\frac{\sqrt{N}(m-\mu_0)}{\sigma} \in (-z_{\alpha/2}, z_{\alpha/2})$$

We reject the null hypothesis H_0 if the Z statistic (above expression) falls outside the confidence interval, on either side. Therefore, this is a **two-sided test**.

If we reject the null hypothesis when it is actually correct, this is a **type-I error**. The significance level α defines how much type I error we can tolerate. A **type II error** is if we fail to reject the null hypothesis when it should actually be rejected (true mean μ not equal to μ_0). The probability of making a type II error is denoted by β and is given by:

$$\beta = P\left(-z_{\alpha/2} < \frac{m - \mu_0}{\sigma/\sqrt{N}} < z_{\alpha/2}\right) = P\left(-z_{\alpha/2} < Z < z_{\alpha/2}\right)$$

The power of the test is equal to $1 - \beta$, and denotes the probability of correctly rejecting the null hypothesis. See table 3.

Table 3: Type I error, Type II error, and power of a test.

	Fail to reject	**Reject**
H_0 is **true**	Correct	Type I error
H_0 is **false**	Type II error	Correct (*power*)

One-sided test A one-sided test has the form:

$$H_0 : \mu \leq \mu_0, \qquad H_1 : \mu > \mu_0$$

The one-sided test with level of significance α defines the $100(1 - \alpha)$ percent confidence interval bounded on one side, in which m should lie, for the hypothesis **not** to be rejected. In other words, we **fail to reject** if

$$\frac{\sqrt{N}(m - \mu_0)}{\sigma} \in (-\infty, z_\alpha)$$

and reject outside of this interval.

If the variance is also unknown, we use the sample variance, and use the t-distribution to account for the additional uncertainty. For example

$$\frac{\sqrt{N}(m - \mu_0)}{S} \in (-t_{\alpha/2,N-1}, t_{\alpha/2,N-1})$$

which is known as the **two-sided *t* test**. Likewise for the one-sided t test.

8.3.2 Assessing a Classification Algorithm's Performance

Binomial Test Given a single training set $\mathcal{T}$ and a single validation set $\mathcal{V}$, we train our classifier on $\mathcal{T}$ and test it on $\mathcal{V}$. The probability that the classifier makes a misclassification is p, which is unknown, and it is what we want to estimate or test a hypothesis about. Let x^t denote the correctness of the classifier on some decision, which is a $0/1$ Bernoulli random variable (1 if misclassification and 0 if correct). Then, the **binomial random variable** X is the total of errors:

$$X = \sum_t x^t$$

We test whether the error probability p is less than or equal to some value p_0:

$$H_0 : p \leq p_0, \qquad H_1 : p > p_0$$

Given that the probability of error is p, we can use the binomial distribution to compute what the probability is of committing j errors out of N:

$$P(X = j) = \binom{N}{j} p^j (1-p)^{N-j}$$

If we then assume H_0 to be true, that is, we assume the true error probability p to be less than or equal to p_0, then we can use p_0 in the above formula instead to compute $P(X = e)$ for some value e, i.e., the probability of seeing e errors. However, we want to compute the probability of seeing e errors or more. Therefore, we sum over all probabilities greater than or equal to $X = e$, which results in $P(X \geq e)$.

$$P(X \geq e) = \sum_{x=e}^{N} \binom{N}{x} p_0^x (1-p_0)^{N-x}$$

Thus, we reject $H_0 : p \leq p_0$ if $P(X \geq e) < \alpha$, where α is the significance level, usually set to 0.05.

Approximate Normal Test If p is the probability of error (misclassification), our point estimate for p is $\hat{p} = X/N$. It is reasonable to reject the null hypothesis if $\hat{p}$ is much larger than p_0.

$$H_0 : p \leq p_0, \qquad H_1 : p > p_0$$

Because X is the sum of independent random variables from the same distribution, the central limit theorem states that for large N, the quantity X/N is approximately normal with mean p_0 and variance $p_0(1-p_0)/N$. The Z statistic is then

$$\frac{m - \mu_0}{\sqrt{\sigma^2/N}} = \frac{X/N - p_0}{\sqrt{p_0(1-p_0)/N}} \simeq \mathcal{Z}$$

where $\simeq$ denotes **approximately distributed**. The *approximately normal test* states that we fail to reject H_0 for $X = e$ if

$$Z \in (-\infty, z_\alpha)$$

and we reject if $Z > z_\alpha$. This only works when N is large enough and p is not close to 0 or 1. The rule is that $Np \geq 5$ and $N(1-p) \geq 5$.

t Test If we run the same algorithm K times on K training/validation set pairs, we get K errors, $p_i, i = 1, \ldots, K$:

$$p_i = \frac{\sum_{t=1}^{N} x_i^t}{N}$$

the average over these error percentages and the variance are then

$$m = \frac{\sum_{i=1}^{K} p_i}{K}, \qquad S^2 = \frac{\sum_{i=1}^{K} (p_i - m)^2}{K-1}$$

The t-statistic is then calculated using the usual formula

$$t = \frac{\sqrt{K}(m - p_0)}{S} \sim t_{K-1}$$

With this statistic, we can reject $H_0 : p \leq p_0$ at significance α if the t statistic is greater than $t_{\alpha,K-1}$, i.e., fail to reject if $t \leq t_{\alpha,K-1}$, and reject if $t > t_{\alpha,K-1}$.

Typically, K is 10 or 30, and the $t_{0.05,9} = 1.83$, and $t_{0.05,29} = 1.70$.

8.3.3 Comparing Two Classification Algorithms

Given two learning algorithms, we want to test whether they construct classifiers that have the same expected error rate.

McNemar's Test For both trained algorithms, we count the frequencies of (mis)classification in a frequency table like the following:

e_{00}: misclassified by both	e_{01}: misclassified by 1 but not 2
e_{10}: misclassified by 2 but not 1	e_{11}: correctly classified by both

Under the null hypothesis that the classification algorithms have the same error rate, i.e., $H_0 : p_1 = p_2$, we expect that the off-diagonals are the same, i.e., $e_{01} = e_{10}$, and that these are equal to $(e_{01} - e_{10})/2$. Using the chi-square statistic with one degree of freedom, we have

$$\frac{(|e_{01} - e_{10}| - 1)^2}{e_{01} + e_{10}} \sim \mathcal{X}_1^2$$

The **McNemar test** rejects the null hypothesis $H_0 : p_1 = p_2$ at significance level α if this value is greater than $\mathcal{X}^2_{\alpha,1}$. For $\alpha = 0.05$, $\mathcal{X}^2_{0.05,1} = 3.84$.

K-Fold Cross-Validated *t* Test We use K-fold cross-validation to get K training/-validation set pairs. The two classification algorithms are trained on the training sets $\mathcal{T}_i, i = 1, \ldots, K$, and tested on the validation sets $\mathcal{V}_i$. The error rates are then recorded for each fold i as p_i^1 and p_i^2.

If the null hypothesis is that the two classifiers have the same error rate, then we expect them to have the same mean. In other words, the difference of their means is 0, i.e., $p_i = p_i^1 - p_i^2$. This is a **paired test**, because for each fold i, both algorithms see the same training and validation sets. We get a distribution of p_i containing K points. Given that p_i^1 and p_i^2 are *approximately normal*, their difference p_i is also normal. The null hypothesis is then:

$$H_0 : \mu = 0, \qquad H_1 : \mu \neq 0$$

We define the average and variance over folds as

$$m = \frac{\sum_{i=1}^K p_i}{K}, \qquad S^2 = \frac{\sum_{i=1}^K (p_i - m)^2}{K - 1}$$

Under the null hypothesis that $\mu = 0$, we have a statistic that is t-distributed with $K - 1$

degrees of freedom:

$$\frac{\sqrt{K}(m-0)}{S} = \frac{\sqrt{K}\cdot m}{S} \sim t_{K-1}$$

Therefore, the *K-fold cv paired t test* rejects the null hypothesis $H_0 : \mu = 0$ at significance level α if the t-statistic is outside the interval $(-t_{\alpha/2,K-1}, t_{\alpha/2,K-1})$, where $t_{0.025,9} = 2.26$ and $t_{0.025,29} = 2.05$.

8.3.4 Comparing Multiple Algorithms: Analysis of Variance

Given multiple algorithms L, that we want to compare in terms of expected error rates, we train them on K training sets, and induce K classifiers belonging to L groups of algorithms, and we test them on K validation sets. This gives L groups of K error values. This is an experiment with a single factor with L levels, i.e., the learning algorithms, and there are K replications for each level.

In **analysis of variance** (ANOVA), we consider L independent samples, each of size K, composed of normal random variables of unknown mean μ_j and unknown common variance σ^2.

$$X_{ij} \sim \mathcal{N}(\mu_j, \sigma^2), \quad j = 1, \ldots, L, \quad i = 1, \ldots, K$$

We want to test the hypothesis H_0 that all means are equal:

$$H_0 : \mu_1 = \mu_2 = \cdots = \mu_L, \qquad H_1 : \mu_r \neq \mu_s, \text{ for at least one pair } (r, s)$$

In concrete terms, the comparison of (unknown) error rates of multiple classification algorithms fits this scheme. We have L classification algorithms and for each one, we have K error rates from validation folds. X_{ij} is the number of errors made by classification algorithm j on fold i. Each X_{ij} is binomial and approximately normal.

The idea of ANOVA is to derive two estimators for the *common* variance σ^2. One estimator is designed such that it is true only when H_0 is true, and the second is always a valid estimator, regardless of whether H_0 is true or not. ANOVA rejects the null hypothesis H_0 that the L samples are drawn from the same population if the two estimators differ significantly.

First estimator assuming H_0 is true The first estimator of σ^2 is valid only if $\mu_j = \mu, j = 1, \ldots, L$, i.e., all means are equal. If the binomial random variable indicating the error count is drawn from the normal distribution $X_{ij} \sim \mathcal{N}(\mu, \sigma^2)$, then the group average

$$m_j = \sum_{i=1}^{K} \frac{X_{ij}}{K}$$

is also normal with mean μ and variance σ^2/K. Therefore, if we assume H_0 to be true, then the sample means $m_j, j = 1, \ldots, L$ must be L instances drawn from $\mathcal{N}(\mu, \sigma^2/K)$. Their mean and variance are then:

$$m = \frac{\sum_{j=1}^{L} m_j}{L}, \qquad S^2 = \frac{\sum_{j=1}^{L}(m_j - m)^2}{L-1}$$

The sample variance only accounts for the means of L learning algorithms, but not the K folds. Therefore, to obtain the estimate $\hat{\sigma}_b^2$, we do $K \cdot S^2$, namely

$$\hat{\sigma}_b^2 = K \frac{\sum_{j=1}^{L}(m_j - m)^2}{L-1}$$

This is also the between-group variance estimate. The between-group sum of squares is

$$SS_b = K \sum_{j=1}^{L}(m_j - m)^2$$

note that the K here denotes that we need to sum over all i folds. The K could also be replaced by a summation, which is more in line with the within-group sum of squares discussed next.

Second estimator regardless of truth of H_0 The second estimator of σ^2 is the average of group variances, S_j^2, defined as

$$S_j^2 = \frac{\sum_{i=1}^{K}(X_{ij} - m_j)^2}{K-1}$$

And their average is then the within-group variance estimate:

$$\hat{\sigma}_w^2 = \sum_{j=1}^{L} \frac{S_j^2}{L} = \sum_j \sum_i \frac{(X_{ij} - m_j)^2}{L(K-1)}$$

The within-group sum of squares is then also defined as

$$\sum_j \sum_i (X_{ij} - m_j)^2$$

Comparing the two variances for equality When H_0 is true, we have $SS_b/\sigma^2 \sim \mathcal{X}^2_{L-1}$, and regardless of the truth of H_0, we have $SS_w/\sigma^2 \sim \mathcal{X}^2_{L(K-1)}$. In other words, the sum of squares for both the between-group and within-group is chi-square distributed with $L-1$ and $L(K-1)$ degrees of freedom, respectively.

To compare the variance estimates, we need to compute their ratio and check whether it is close to 1. The ratio of two independent chi-square random variables divided by their respective degrees of freedom is a F-distributed random variable. Therefore, if H_0 is true,

$$F_0 = \frac{\left(\frac{SS_b/\sigma^2}{L-1}\right)}{\left(\frac{SS_w/\sigma^2}{L(K-1)}\right)} = \frac{\left(\frac{SS_b}{L-1}\right)}{\left(\frac{SS_w}{L(K-1)}\right)} = \frac{\hat{\sigma}_b^2}{\hat{\sigma}_w^2} \sim F_{L-1,L(K-1)}$$

For any given significance level, the hypothesis H_0 is rejected if $F_0 > F_{\alpha,L-1,L(K-1)}$.

If H_0 is not true, then the variance of each m_j around m will be larger than expected, and $\hat{\sigma}_b^2$ will overestimate σ^2, which will make the ratio greater than 1. For $\alpha = 0.05, L = 5$, and $K = 10$, we have $F_{0.05,4,45} = 2.6$.

The name **analysis of variance** is derived from a partitioning of the total variability

in the data into its components:

$$SS_T = \sum_j \sum_i (X_{ij} - m)^2$$

If we divide SS_T by its degree of freedom, i.e., $K \cdot L - 1$, we get the sample variance of X_{ij}. The total variability is also equal to $SS_T = SS_b + SS_w$.

If the hypothesis is rejected, we know that there is *some* difference between the L groups, but we do not know where. We do **post hoc testing** to test subsets of groups against each other, for example, pairs. **Fisher's least squares difference** test compares groups in a pairwise manner. For each group, we have $m_i \sim \mathcal{N}(\mu_i, \sigma_w^2 = MS_w/K)$, and $m_i - m_j \sim \mathcal{N}(\mu_i - \mu_j, 2\sigma_w^2)$. Then, under the null hypothesis that $H_0 : \mu_i = \mu_j$, we have the t-statistic:

$$t = \frac{m_i - m_j}{\sqrt{2}\sigma_w} \sim t_{L(K-1)}$$

We reject H_0 if $|t| > t_{\alpha/2, L(K-1)}$. Similarly, one-sided tests can be defined.

When doing multiple tests to draw *one* conclusion, also called **multiple comparisons**, we need to keep in mind that if T hypotheses are tested, each at significance level α, then the probability that at least one hypothesis is incorrectly rejected (type I error) is at most $T\alpha$. For example, if we test 6 hypotheses against significance level $\alpha = 0.05$, then the probability that at least one of these is incorrectly rejected is $6 \times 0.05 = 0.3$. Therefore, to ensure correct confidence intervals, we set α to be α/T. This is the **Bonferroni correction**.

If ANOVA rejects and none of the pairwise post hoc tests find a significant difference, then the conclusion should be that there is a difference, but that we need more data to pinpoint exactly where this difference lies.

8.4 Measuring Classifier Performance

In binary classification, there are 4 possible cases, as shown in table 4. There are two types of errors. (1) **false negatives** occur when we predict negative, but this is false, and was actually a positive. (2) **false positives** occur when we predict positive, but this is false, and was actually a negative.

Table 4: Confusion matrix for two classes.

	Predicted class		
True class	**Positive**	**Negative**	*Total*
Positive	tp: true positive	fn: false negative	p
Negative	fp: false positive	tn: true negative	n
Total	p'	n'	N

More performance measures are listed in table 5. The two types of errors may have different meanings in certain applications. For example, consider a test to determine if a person is pregnant.

Table 5: Performance measures for two classes.

Name	**Formula**
error	$(fp + fn)/N$
accuracy	$(tp + tn)/N = 1 - \text{error}$
tp-rate	tp/p
fp-rate	fp/n
precision	tp/p'
recall	$tp/p = \text{tp-rate}$
sensitivity	$tp/p = \text{tp-rate}$
specificity	$tn/n = 1 - \text{fp-rate}$

- If the test is positive, but the person is a man, then this is a false positive, also known as a *false alarm*.
- If the test is negative, but the person is actually pregnant, then this is a false negative.

It is clear that we would prefer less false negatives than false positives, as incorrectly classifying a women to **not** be pregnant makes the test very unreliable. Rather, if the test produces more false positives, but less false negatives, the consequences are manageable.

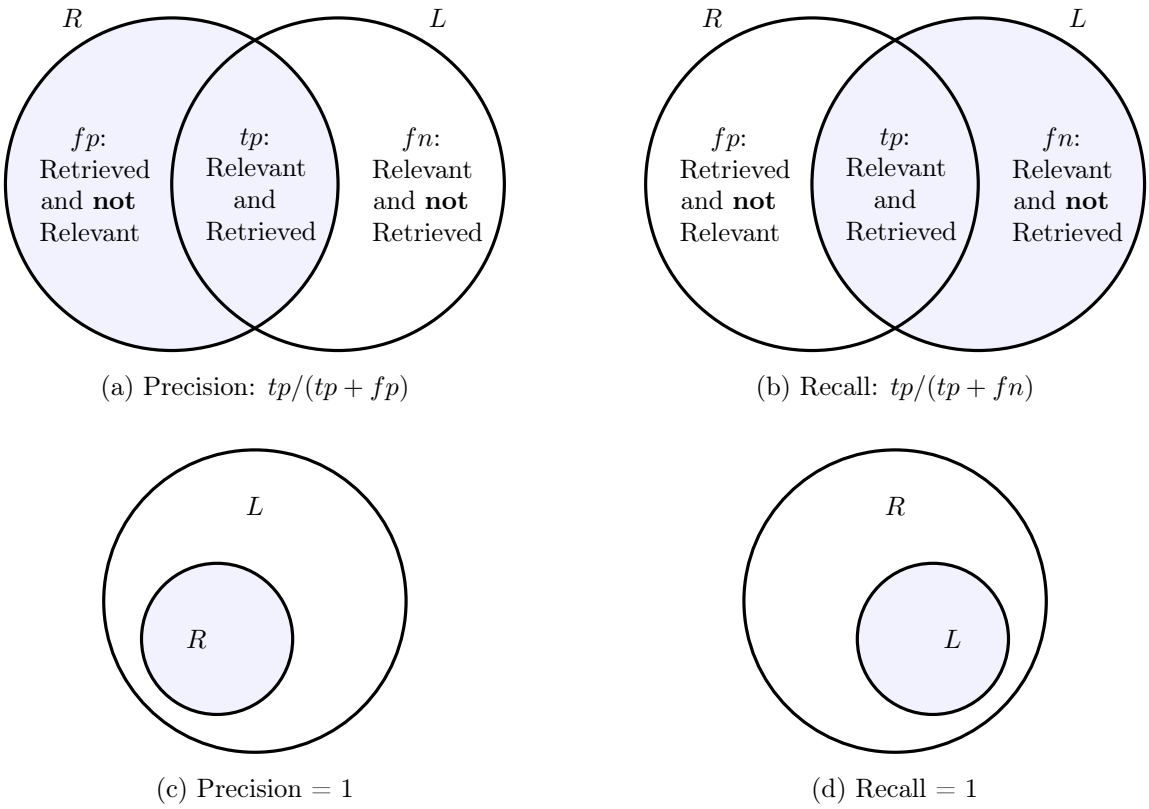
If a system returns $\hat{P}(C_1 \mid \mathbf{x})$ for the probability of a positive class, then $\hat{P}(C_2 \mid \mathbf{x}) = 1 - \hat{P}(C_1 \mid \mathbf{x})$ is the probability for a negative class. Normally, we choose positive if $\hat{P}(C_1 \mid \mathbf{x}) > \theta$ for $\theta = 0.5$, but this threshold can be arbitrary. If θ is close to 1, we do not choose the positive class often, i.e., no false positives but also few true positives. As θ decreases, there are more false positives.

The **Receiver Operating Curve (ROC)** is a plot of the fp-rate against the tp-rate for various values of θ. Given two classifiers, we say one is better if its ROC curve is above the other's ROC curve. We can reduce the curve to a single number by calculating the **Area Under the Curve (AUC)**. AUC values closer to 1 are ideal.

Information retrieval Consider a database of records, where we can make queries by using keywords, and the system returns a number of records classified by the query. There are *relevant records* for the query, and the system might retrieve some of them (true positives), but there are also records that the system classifies as irrelevant that are actually relevant to the query (false negatives). It may also retrieve records that are classified as relevant, but are not relevant (false positives). Figure 8.2 shows Venn diagrams of the relevant and retrieved records.

- **Precision**: number of retrieved and relevant records (true positives) divided by all retrieved records (true positives and false positives), i.e., $tp/(tp + fp)$.
- **Recall**: number of retrieved and relevant records (true positives) divided by all relevant records (true positives and false negatives), i.e., $tp/(tp + fn)$.

As shown in figure 8.2c and d, if precision is 1, all retrieved records may be relevant but there may still be records that are relevant but were not retrieved. Likewise, if recall is 1,



(a) Precision: $tp/(tp + fp)$

(b) Recall: $tp/(tp + fn)$

(c) Precision = 1

(d) Recall = 1

Figure 8.2: (a) and (b) Venn diagram definition of precision and recall. (c) Precision is 1 if all retrieved records are also relevant. (d) Recall is 1 if all relevant records are also retrieved.

all relevant records may be retrieved, but there may also be irrelevant records that were retrieved.

From another perspective, there is **sensitivity**, which is the same as tp-rate and recall. **Specificity** says how well we detect the negatives. In other words, how well do we detect the number of true negatives divided by the total number of negatives. This is 1 minus the false alarm rate, i.e., 1 − fp-rate.

8.5 Cross-validation and resampling

For large datasets $\mathcal{X}$, we can randomly divide it into K parts, and then randomly divide each part into two and use one half for training and the other half for validation. K is typically 10 or 30. However, $\mathcal{X}$ is often not that large. Therefore, the idea of cross-validation is to repeatedly split the same dataset differently. This makes the error percentages dependent because the different sets share data with each other.

8.5.1 *K*-Fold Cross Validation

The dataset $\mathcal{X}$ is divided randomly into K equal sized parts, $\mathcal{X}_i, i = 1, \ldots, K$. Each pair of training/validation set is generated by using one of the K as the validation set and combining the remaining $K - 1$ parts to form the training set. Doing this K times, we get

K pairs.

$$\begin{aligned}
\mathcal{V}_1 &= \mathcal{X}_1 & \mathcal{T}_1 &= \mathcal{X}_2 \cup \mathcal{X}_3 \cup \cdots \cup \mathcal{X}_K \\
\mathcal{V}_2 &= \mathcal{X}_2 & \mathcal{T}_2 &= \mathcal{X}_1 \cup \mathcal{X}_3 \cup \cdots \cup \mathcal{X}_K \\
&\vdots & & \\
\mathcal{V}_K &= \mathcal{X}_K & \mathcal{T}_K &= \mathcal{X}_1 \cup \mathcal{X}_2 \cup \cdots \cup \mathcal{X}_{K-1}
\end{aligned}$$

Some consequences of K-Fold are the following:

- Validation sets are small to keep training sets large, and training sets overlap (any two training sets share $K-2$ parts).
- As K increases, the percentage of training instances increases, and we get more robust estimators, but the validation sets become smaller.
- As N increases, K can be smaller. If N is small, K should be large to allow large enough training sets.
- **Leave-one-out**: Given N instances, only one instance is left out as the validation set and $N-1$ instances are used for training. We get N pairs by leaving a different instance out at each iteration.

8.5.2 5 × 2 Cross-Validation

We divide $\mathcal{X}$ randomly into two parts $\mathcal{V}_1 = \mathcal{X}_1^{(1)}$ and $\mathcal{T}_1 = \mathcal{X}_1^{(2)}$, which is our first pair. We then swap the two parts and get $\mathcal{V}_1 = \mathcal{X}_1^{(2)}$ and $\mathcal{T}_1 = \mathcal{X}_1^{(1)}$. This is the first fold, where $\mathcal{X}_i^{(j)}$ denotes half j of fold i. For the next folds, we shuffle $\mathcal{X}$ and repeat this process. From each fold, we get 2 pairs, and doing this 5 times, we get 10 training and validation sets.

Doing this for more than 5 folds may result in too much overlap because validation errors become too dependent, i.e., validation errors do not add new information.

8.5.3 Bootstrapping

An alternative to cross-validation is bootstrapping that generates new samples by drawing instances from the original sample **with replacement**. Bootstrap samples may overlap more than cross-validation samples and are therefore more dependent. This is nonetheless considered the best way to do resampling for very small datasets.

1. In the bootstrap, we sample N instances from a dataset of size N with replacement.
2. Original dataset or the instances not selected by the bootstrap are used as the validation set.
3. Probability that we pick an instance is $1/N$.
4. Probability that we do not pick an instance is $1-1/N$.
5. Probability that we do not pick an instance after N draws is $(1-1/N)^N \approx e^{-1} = 0.368$.

This means that the training set contains approximately 63.2 percent of the instances, which means that the system will not be trained on 36.8 percent of the data, and the error estimate will be pessimistic. The solution is **replication**: repeat the process of bootstrapping many times and look at the average behaviour.

8.6 Comparison over Multiple Datasets

If we are interested in comparing two or more algorithms on several datasets, then depending on how well the chosen inductive bias matches the problem, the algorithms will behave differently on different datasets. The resulting error values cannot be said to be normally distributed around some mean accuracy. Therefore, the **parametric tests** based on binomials being approximately normal are no longer applicable, and we need to resort to nonparametric tests.

Parametric tests are robust, especially if the sample is large, whereas nonparametric tests are distribution free, but are less efficient. Nonparametric tests assume no knowledge about the distribution of the population, but rather that the values can be compared or ordered.

When algorithm A and B are trained on a number of different datasets, the average of their errors on these datasets is not a meaningful value. Rather, on any given dataset, if A is more accurate than B, we count in favour of A, and otherwise B. Therefore, we count the number of times A is more accurate than B, and check whether this could have been by chance if they indeed were equally accurate.

8.6.1 Comparing Two Algorithms

In comparing two algorithms, let us say we train and validate them on $i = 1, \ldots, N$ different datasets in a paired manner, that is, all conditions are the same except for the learning algorithms. The results are two vectors indexed by i as e_i^1 and e_i^2, and these are averages of K values in K-fold cross-validation.

The idea behind the **sign test** is that if both algorithms have equal error, then on each dataset, there should be 0.5 probability that the first has less error than the second. We *expect* the first to win on $N/2$ datasets.

$$X_i = \begin{cases} 1 & \text{if } e_i^1 < e_i^2 \\ 0 & \text{otherwise} \end{cases}, \quad \text{and } X = \sum_{i=1}^{N} X_i$$

Because we expect both algorithms to win $N/2$ times, we can formulate the hypothesis:

$$H_0 : \mu_1 \geq \mu_2, \qquad H_1 : \mu_1 < \mu_2$$

If the null hypothesis is correct, then X is binomial in N trials with probability of either algorithm winning $p = 0.5$. Given that the first algorithm wins on $X = e$ datasets, the probability that we have e *or less* wins when $p = 0.5$ (H_0 assumption) is:

$$P(X \leq e) = \sum_{x=0}^{e} \binom{N}{x} 0.5^x (1 - 0.5)^{N-x}$$

and we reject the null hypothesis if this probability is too small, i.e., less than α. If there are t ties, then we add $t/2$ to e.

Alternatively, in testing the hypothesis:

$$H_0 : \mu_1 \leq \mu_2, \qquad H_1 : \mu_1 > \mu_2$$

we reject if $P(X \geq e) < \alpha$.

And for the two sided test

$$H_0 : \mu_1 = \mu_2, \qquad H_1 : \mu_1 \neq \mu_2$$

we reject the null hypothesis if e is too small or too large.

- If $e < N/2$, we **reject** if $2 \times P(X \leq e) < \alpha$.
- If $e > N/2$, we **reject** if $2 \times P(X \geq e) < \alpha$.

In other words, we find the corresponding tail, and multiply it by 2 because it is a two-tailed test.

8.6.2 Comparing Multiple Algorithms

The **Kruskal-Wallis test** is the nonparametric version of ANOVA and is a multiple sample generalisation of a rank test. Given $M = L \times N$ observations consisting of error rates of L algorithms on N datasets. We define $X_{ij}, i = 1, \ldots, L, j = 1, \ldots, N$ as the error sets, and rank them from smallest to largest and assign them ranks R_{ij}, between 1 and M. In case of ties, we take averages. If the null hypothesis

$$H_0 : \mu_1 = \mu_2 = \cdots = \mu_L$$

is assumed to be true, then the average of ranks of algorithm i should be approximately halfway between 1 and M, i.e., $(M + 1)/2$.

We denote the sample average rank of algorithm i by $\overline{R}_i$ and we reject the hypothesis if the average ranks seem to differ from halfway. The test statistic is

$$H = \frac{12}{(M+1)L} \sum_{i=1}^{L} \left(\overline{R}_i - \frac{M+1}{2} \right)$$

is approximately chi-square distributed with $L-1$ degrees of freedom and we reject the null hypothesis if the statistic exceeds $\mathcal{X}_{\alpha,L-1}$.

8.7 Multivariate tests

The tests discussed this far are univariate, that is, they use a single performance measure, e.g., error, AUC, etc. A bivariate test would for example consist of [precision, recall] or [tp-rate, fp-rate]. Since most measures are calculated in terms of the entries in the confusion matrix, we can define a 4-variate test on [tp, fp, tn, fn].

8.7.1 Comparing Two Algorithms

Assuming that $\mathbf{x}_{ij}$ are p-variate normal distributions, we have $i = 1, \ldots, K$ folds and the comparison of two algorithms, $j = 1, 2$. If we use the 4-variate distribution of confusion matrix entries, we get for each of the two algorithms, K vectors of size 4.

We want to test whether the two populations (algorithms) have the same mean vector in the p-dimensional space.

$$H_0 : \boldsymbol{\mu}_1 = \boldsymbol{\mu}_2, \qquad H_1 : \boldsymbol{\mu}_1 \neq \boldsymbol{\mu}_2$$

In the case of paired testing, we calculate the paired differences, $\mathbf{d}_i = \mathbf{x}_{i1} - \mathbf{x}_{i2}$, and test whether the mean of these i differences is zero:

$$H_0 : \boldsymbol{\mu}_d = \mathbf{0}, \qquad H_1 : \boldsymbol{\mu}_d \neq \mathbf{0}$$

To test for this, we calculate the sample average and covariance matrix:

$$\mathbf{m} = \sum_{i=1}^{K} \frac{\mathbf{d}_i}{K}, \qquad \mathbf{S} = \frac{1}{K-1} \sum_{i=1}^{K} (\mathbf{d}_i - \mathbf{m})(\mathbf{d}_i - \mathbf{m})^T$$

Under the null hypothesis, the **Hotelling's multivariate test** statistic

$$T'^2 = K \mathbf{m}^T \mathbf{S}^{-1} \mathbf{m}$$

is Hotelling's T^2 distributed with p and $K - 1$ degrees of freedom. We reject the null hypothesis if $T'^2 > T^2_{\alpha,p,K-1}$.

When $p = 1$, this multivariate test reduces to the paired t-test. The t-statistic in the univariate case, $\sqrt{K}m/S$ measures the normalised distance to 0 in one dimension, whereas here $K\mathbf{m}^T\mathbf{S}^{-1}\mathbf{m}$ measures the squared Mahalanobis distance to $\mathbf{0}$ in p dimensions.In both cases, we reject if the distance to $\mathbf{0}$ is so large that it can only occur at most with probability α.

If the multivariate test rejects the null hypothesis, we can do p separate post hoc univariate tests to check which one of the variates caused the rejection. If the multivariate test on [fp, fn] rejects the null hypothesis, we can check whether the difference is due to a significant difference in false positives, false negatives, or both.

The advantage of the multivariate test is that it accounts for combinations of differences that may result in a significant difference, whereas none of the univariate differences may yield a significant difference. The linear combination of variates that causes the maximum difference is then

$$\mathbf{w} = \mathbf{S}^{-1}\mathbf{m}$$

The vector $\mathbf{w}$ gives the effect of the different univariate dimensions. If $p = 4$ with the entries of the confusion matrix, then $\mathbf{w}$ is a new performance measure from the original four entries. This is the same as Fisher's LDA direction, as it is the direction that maximises the separation of two groups of data.

8.7.2 Comparing Multiple Algorithms

The multivariate version of ANOVA, namely MANOVA, is defined by the hypothesis:

$$H_0 : \boldsymbol{\mu}_1 = \boldsymbol{\mu}_2 = \cdots = \boldsymbol{\mu}_L, \qquad H_1 : \boldsymbol{\mu}_r \neq \boldsymbol{\mu}_s, \text{for at least one pair } r, s$$

The multivariate ANOVA (MANOVA) calculates the two matrices of between- and within-scatter:

$$\mathbf{H} = K \sum_{j=1}^{L} (\mathbf{m}_j - \mathbf{m})(\mathbf{m}_j - \mathbf{m})^T, \qquad \text{between-scatter}$$

$$\mathbf{E} = \sum_{j=1}^{L} \sum_{i=1}^{K} (\mathbf{x}_{ij} - \mathbf{m}_j)(\mathbf{x}_{ij} - \mathbf{m}_j)^T, \qquad \text{within-scatter}$$

Then, the test statistic is defined as

$$\Lambda' = \frac{|\mathbf{E}|}{|\mathbf{E} + \mathbf{H}|}$$

is **Wilks'** Λ distributed with $p, L(K-1), L-1$ degrees of freedom. We then reject the null hypothesis if $\Lambda' > \Lambda_{\alpha,p,L(K-1),L-1}$.

9 Kernel Methods

9.1 Optimal Separating Hyperplane

Given a classification setup, we start with two classes and use labels $-1/+1$ for the classes. The sample is $\mathcal{X} = \{\mathbf{x}^t, r^t\}$, where $r^t = +1$ if $\mathbf{x}^t \in C_1$ and $r^t = -1$ if $\mathbf{x}^t \in C_2$. We would like to find $\mathbf{w}$ and w_0 such that

$$\begin{aligned} \mathbf{w}^T\mathbf{x}^t + w_0 &\geq +1 \quad \text{for} \quad r^t = +1 \\ \mathbf{w}^T\mathbf{x}^t + w_0 &\leq -1 \quad \text{for} \quad r^t = -1 \end{aligned}$$

We can combine both inequalities by noticing that multiplying $r^t \in \{-1, +1\}$ with the linear discriminant will always make it positive. Thus, this can be rewritten as

$$r^t(\mathbf{w}^T\mathbf{x}^t + w_0) \geq +1$$

The instances should be separated by a hyperplane, where the positive instances fall on the positive side of the hyperplane and the negative instances fall on the negative side of the hyperplane. The distance between the instances closest to the hyperplane (on either side) is called the **margin**, which we want to maximise for better *generality*. It is better to fit a model halfway between the most specific hypothesis S and the most general hypothesis G, so that noise will not drastically affect classification results.

Similarly, now that we are using the hypothesis class of lines, the **optimal separating hyperplane** is the one that maximises the margin.

The distance from an instance $\mathbf{x}^t$ to the discriminant (line) is the absolute value of the projection of $\mathbf{x}^t$ onto the discriminant, divided by the norm (or square root of dot product):

$$\frac{|\mathbf{w}^T\mathbf{x}^t + w_0|}{||\mathbf{w}||}$$

When the labels exclusively take values $r^t \in \{-1, +1\}$, this can be rewritten as

$$\frac{r^t(\mathbf{w}^T\mathbf{x}^t + w_0)}{||\mathbf{w}||}$$

The idea is now that we would like the distance to the projection line (discriminant) to be *at least* some value ρ:

$$\frac{r^t(\mathbf{w}^T\mathbf{x}^t + w_0)}{||\mathbf{w}||} \geq \rho, \quad \forall t$$

This is also shown in figure 9.1, where we want to maximise the distance between every point and the discriminant line with respect to some ρ. However, by scaling $\mathbf{w}$, there are an infinite amount of solutions to maximise ρ. Therefore, we fix $\rho||\mathbf{w}|| = 1$, from which we can derive the total distance between the positive side margin and the negative side margin. In other words, the total distance between the two parallel lines (on both sides of the discriminant line).

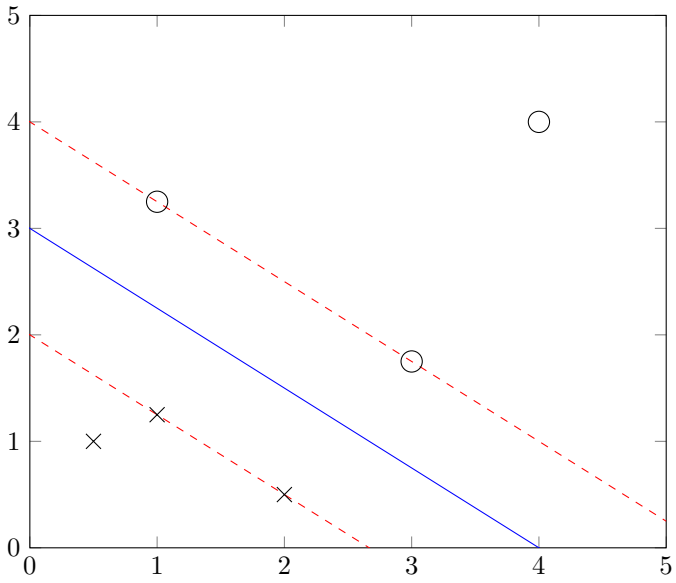


Figure 9.1: Decision boundary with dashed margins on both sides.

Substituting $\rho||\mathbf{w}|| = 1 \implies \rho = 1/||\mathbf{w}||$ in the distance equation, we get:

$$\begin{aligned}
\frac{r^t(\mathbf{w}^T\mathbf{x}^t + w_0)}{||\mathbf{w}||} &\geq \rho \\
\frac{r^t(\mathbf{w}^T\mathbf{x}^t + w_0)}{||\mathbf{w}||} &\geq \frac{1}{||\mathbf{w}||} \\
r^t(\mathbf{w}^T\mathbf{x}^t + w_0) &\geq 1 \\
r^t(\mathbf{w}^T\mathbf{x}^t + w_0) - 1 &\geq 0
\end{aligned}$$

If $\mathbf{x}^t$ is on the margin parallel to the decision boundary, then this inequality becomes the equality $r^t(\mathbf{w}^T\mathbf{x}^t + w_0) - 1 = 0$. To find the distance between the margins on both sides, consider a positive example $\mathbf{x}_+$, and a negative example $\mathbf{x}_-$, that lie exactly on the positive margin and negative margin. Using this knowledge, the following two identities can be derived:

$$\begin{aligned}
\mathbf{w}^T\mathbf{x}^t &= 1 - w_0, && \text{for } \mathbf{x}_+ \\
\mathbf{w}^T\mathbf{x}^t &= -1 - w_0, && \text{for } \mathbf{x}_-
\end{aligned}$$

The difference $(\mathbf{x}_+ - \mathbf{x}_-)$ gives the vector from $\mathbf{x}_-$ to $\mathbf{x}_+$. If we take the dot product of this difference with the normalised vector $\mathbf{w}/||\mathbf{w}||$, we get the distance between the two hyperplanes (margin lines parallel to decision boundary):

$$\begin{aligned}
\text{width} &= (\mathbf{x}_+ - \mathbf{x}_-) \cdot \frac{\mathbf{w}}{||\mathbf{w}||} \\
&= \frac{(\mathbf{x}_+ - \mathbf{x}_-)\mathbf{w}}{||\mathbf{w}||} \\
&= \frac{\mathbf{x}_+ \cdot \mathbf{w} - \mathbf{x}_- \cdot \mathbf{w}}{||\mathbf{w}||}
\end{aligned}$$

$$= \frac{1 - w_0 - (-1 - w_0)}{||\mathbf{w}||}$$
$$= \frac{2}{||\mathbf{w}||}$$

Thus, the expression that we need to maximise is $2/||\mathbf{w}||$, which is equivalent to the task:

$$\min \frac{1}{2}||\mathbf{w}||^2, \quad \text{subject to constraints} \qquad r^t(\mathbf{w}^T\mathbf{x}^t + w_0) - 1 \geq 0, \forall t$$

In finding the optimal hyperplane, we can convert the optimisation problem to a form whose complexity depends on N, the number of training instances, and not on d, the input dimensionality. We first write the above constrained optimisation problem as an *unconstrained optimisation problem* using Lagrange multipliers α^t for each of the N constraints:

$$\begin{aligned} L_p &= \frac{1}{2}||\mathbf{w}||^2 - \sum_{t=1}^{N} \alpha^t \left[r^t(\mathbf{w}^T\mathbf{x}^t + w_0) - 1\right] \\ &= \frac{1}{2}||\mathbf{w}||^2 - \sum_{t=1}^{N} \alpha^t r^t(\mathbf{w}^T\mathbf{x}^t + w_0) - \alpha^t \\ &= \frac{1}{2}||\mathbf{w}||^2 - \left(\sum_{t=1}^{N} \alpha^t r^t(\mathbf{w}^T\mathbf{x}^t + w_0) - \sum_{t=1}^{N} \alpha^t\right) \\ &= \frac{1}{2}||\mathbf{w}||^2 - \sum_{t=1}^{N} \alpha^t r^t(\mathbf{w}^T\mathbf{x}^t + w_0) + \sum_{t=1}^{N} \alpha^t \end{aligned}$$

The optimisation function L_p is denoted as the *primal*, where we want to minimise L_p with respect to $\mathbf{w}, w_0$ and maximise with respect to $\alpha^t \geq 0$.

We can reformulate this as the *dual*, where we maximise L_p with respect to α^t, subject to the constraints that the gradient of L_p with respect to $\mathbf{w}$ and w_0 are 0, and also that $\alpha^t \geq 0$. In other words,

$$\min_{\mathbf{w}, w_0} \max_{\alpha^t \geq 0} L_p = \max_{\alpha^t \geq 0} \min_{\mathbf{w}, w_0} L_d$$

Taking the partial derivative of L_p with respect to $\mathbf{w}$ and w_0, and setting it equal to 0, we get:

$$\frac{\partial L_p}{\partial \mathbf{w}} = 0 \implies \mathbf{w} - \sum_{t=1}^{N} \alpha^t r^t \mathbf{x}^t = 0 \implies \mathbf{w} = \sum_{t=1}^{N} \alpha^t r^t \mathbf{x}^t$$
$$\frac{\partial L_p}{\partial w_0} = 0 \implies \sum_{t=1}^{N} \alpha^t r^t = 0$$

Plugging these into the primal, we get the dual:

$$\begin{aligned} L_d &= \frac{1}{2}(\mathbf{w}^T\mathbf{w}) - \sum_{t=1}^{N} \alpha^t r^t(\mathbf{w}^T\mathbf{x}^t + w_0) + \sum_{t=1}^{N} \alpha^t \\ &= \frac{1}{2}(\mathbf{w}^T\mathbf{w}) - \left(\sum_{t=1}^{N} \alpha^t r^t \mathbf{w}^T\mathbf{x}^t + \alpha^t r^t w_0\right) + \sum_{t=1}^{N} \alpha^t \end{aligned}$$

$$
\begin{aligned}
&= \frac{1}{2}(\mathbf{w}^T\mathbf{w}) - \mathbf{w}^T \sum_{t=1}^{N} \alpha^t r^t \mathbf{x}^t - w_0 \sum_{t=1}^{N} \alpha^t r^t + \sum_{t=1}^{N} \alpha^t \\
&= \frac{1}{2}(\mathbf{w}^T\mathbf{w}) - \mathbf{w}^T\mathbf{w} - w_0 \times 0 + \sum_{t=1}^{N} \alpha^t \\
&= \mathbf{w}^T\mathbf{w}\left(\frac{1}{2} - 1\right) + \sum_{t=1}^{N} \alpha^t \\
&= -\frac{1}{2}\mathbf{w}^T\mathbf{w} + \sum_{t=1}^{N} \alpha^t \\
&= -\frac{1}{2}\sum_{t=1}^{N} \alpha^t r^t \left(\mathbf{x}^t\right)^T \sum_{s=1}^{N} \alpha^s r^s \mathbf{x}^s + \sum_{t=1}^{N} \alpha^t \\
&= -\frac{1}{2}\sum_{t=1}^{N}\sum_{s=1}^{N} \alpha^t \alpha^s r^t r^s \left(\mathbf{x}^t\right)^T \mathbf{x}^s + \sum_{t=1}^{N} \alpha^t \\
&= -\frac{1}{2}\mathbf{\Lambda}^T\mathbf{D}\mathbf{\Lambda} + \mathbf{\Lambda}^T\mathbf{1}
\end{aligned}
$$

Where $\mathbf{\Lambda} = \left[\alpha^1, \alpha^2, \ldots, \alpha^N\right]^T$ and $\mathbf{1}$ is the unit vectors with all 1's, and $\mathbf{D}$ is the symmetric $N \times N$ matrix with elements $\mathbf{D}_{ij} = r^i r^j \left(\mathbf{x}^i\right)^T \mathbf{x}^j$.

The above *dual expression* allows us to simply maximise with respect to α^t only, subject to the constraints

$$\sum_t \alpha^t r^t = 0 \text{ due to the constraint of } \partial L_p/\partial w_0 \text{ and } \alpha^t \geq 0, \forall t$$

Once we solve for α^t, we see that even though there are N values of α, most vanish with $\alpha^t = 0$, and only a small number have $\alpha^t > 0$. We say that the set of $\mathbf{x}^t$ whose corresponding $\alpha^t > 0$ are the **support vectors**. Furthermore, because the partial derivative of $\partial L_p/\partial \mathbf{w}$ set at 0 evaluates to the expression

$$\mathbf{w} = \sum_{t=1}^{N} \alpha^t r^t \mathbf{x}^t$$

the positive α^t values remain, which means that $\mathbf{w}$ is actually the weighted sum of the support vectors multiplied by the α^t and r^t values. The support vectors $\mathbf{x}^t$ also satisfy $r^t(\mathbf{w}^T\mathbf{x}^t + w_0) = 1$ and lie on the margin. Using this fact, we can calculate w_0 from any *support vector* by taking $\mathbf{w}^T\mathbf{x}^t + w_0 = r^t$ and solving for w_0:

$$w_0 = r^t - \mathbf{w}^T\mathbf{x}^t$$

For numerical stability, it is advised that w_0 is taken as the average over all such calculations.

The resulting discriminant $g(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + w_0$ is called the **support vector machine** (SVM). For $\alpha^t = 0$, the corresponding $\mathbf{x}^t$ lie too far away from the discriminant and have no effect on the hyperplane. Instances that are not support vectors carry no information. This idea is similar to the condensed nearest neighbours algorithm. During testing, we calculate $g(\mathbf{x})$ and choose C_1 if $g(\mathbf{x}) > 0$ and C_2 otherwise.

9.2 The Non-separable Case: Soft Margin Hyperplane

If the two classes are not linearly separable, such that there is no hyperplane to separate them, we look for the hyperplane that results in the least error. We define **slack variables** $\xi^t \geq 0$, which store the deviation from the margin. There are two types or deviation. (1) an instance may lie on the wrong side of the hyperplane, entailing a misclassification, or, (2) an instance may be on the right side but it lies within the space between the discriminant and the margin, i.e., not sufficiently far away from the hyperplane. Therefore, using the slack variables, we can define the constraints:

$$r^t(\mathbf{w}^T\mathbf{x}^t + w_0) \geq 1 - \xi^t$$

- If $\xi^t = 0$, there is no problem with $\mathbf{x}^t$.
- If $0 < \xi^t < 1$, then $\mathbf{x}^t$ is correctly classified but within the margin space.
- If $\xi^t \geq 1$, then $\mathbf{x}^t$ is misclassified.

The number of misclassifications is therefore $\#\{\xi^t \geq 1\}$, and the number of non-separable instances is $\#\{\xi^t \geq 0\}$, which is the total of the misclassified instances and the instances that are not sufficiently far away from the margin. This total is summed to form the **soft error** as defined by

$$\sum_t \xi^t$$

and we add this error term as a **penalty term** with penalty factor C to the primal optimisation problem L_p:

$$L_p = \frac{1}{2}||\mathbf{w}||^2 + C\sum_t \xi^t$$

The penalty factor C allows for regularisation and trades off complexity. For better generalisation, note that the soft error not only penalises misclassifications but also instances that fall within the margin space.

the new Lagrangian expression now becomes:

$$L_p = \frac{1}{2}||\mathbf{w}||^2 + C\sum_t \xi^t - \sum_{t=1}^{N} \alpha^t \left[r^t(\mathbf{w}^T\mathbf{x}^t + w_0) - 1 + \xi^t\right] - \sum_t \mu^t\xi^t$$

where μ^t are the new Lagrange parameters to guarantee that ξ^t are positive. Taking the derivative with respect to the parameters $\mathbf{w}, w_0, \xi^t$, and setting them to 0, we get:

$$\begin{aligned}
\frac{\partial L_p}{\partial \mathbf{w}} = 0 &\implies \mathbf{w} = \sum_t \alpha^t r^t \mathbf{x}^t \\
\frac{\partial L_p}{\partial w_0} = 0 &\implies \sum_t \alpha^t r^t = 0 \\
\frac{\partial L_p}{\partial \xi^t} = 0 &\implies C - \alpha^t - \mu^t = 0
\end{aligned}$$

Since we introduced μ^t for the purpose of keeping ξ^t positive, it is constrained by $\mu^t \geq 0$, which means that the last derivative implies that $0 \leq \alpha^t \leq C$. Again plugging these into

the primal, we get the dual optimisation problem:

$$
\begin{aligned}
L_d &= \frac{1}{2}||\mathbf{w}||^2 + C\sum_{t=1}^{N}\xi^t - \sum_{t=1}^{N}\alpha^t r^t(\mathbf{w}^T\mathbf{x}^t + w_0) - \alpha^t + \alpha^t\xi^t - \sum_{t=1}^{N}\mu^t\xi^t \\
&= \frac{1}{2}||\mathbf{w}||^2 + C\sum_{t=1}^{N}\xi^t - \left(\sum_{t=1}^{N}\alpha^t r^t\mathbf{w}^T\mathbf{x}^t + \alpha^t r^t w_0 - \alpha^t + \alpha^t\xi^t\right) - \sum_{t=1}^{N}\mu^t\xi^t \\
&= \frac{1}{2}||\mathbf{w}||^2 + C\sum_{t=1}^{N}\xi^t - \left(\mathbf{w}^T\sum_{t=1}^{N}\alpha^t r^t\mathbf{x}^t + w_0\sum_{t=1}^{N}\alpha^t r^t - \sum_{t=1}^{N}\alpha^t + \alpha^t\xi^t\right) - \sum_{t=1}^{N}\mu^t\xi^t \\
&= \frac{1}{2}||\mathbf{w}||^2 + C\sum_{t=1}^{N}\xi^t - \left(\mathbf{w}^T\mathbf{w} + w_0 \times 0 - \sum_{t=1}^{N}\alpha^t + \alpha^t\xi^t\right) - \sum_{t=1}^{N}\mu^t\xi^t \\
&= \frac{1}{2}||\mathbf{w}||^2 + C\sum_{t=1}^{N}\xi^t - \mathbf{w}^T\mathbf{w} + \sum_{t=1}^{N}\alpha^t + \alpha^t\xi^t - \sum_{t=1}^{N}\mu^t\xi^t \\
&= -\frac{1}{2}||\mathbf{w}||^2 + \sum_{t=1}^{N}C\xi^t + \alpha^t + \alpha^t\xi^t - \mu^t\xi^t \\
&= -\frac{1}{2}||\mathbf{w}||^2 + \sum_{t=1}^{N}\xi^t(C + \alpha^t - \mu^t) + \alpha^t \\
&= -\frac{1}{2}||\mathbf{w}||^2 + \sum_{t=1}^{N}\xi^t(0) + \alpha^t \\
&= -\frac{1}{2}||\mathbf{w}||^2 + \sum_{t=1}^{N}\alpha^t
\end{aligned}
$$

This last form is again equivalent to the separable case:

$$
L_d = -\frac{1}{2}\sum_{t=1}^{N}\sum_{s=1}^{N}\alpha^t\alpha^s r^t r^s \left(\mathbf{x}^t\right)^T\mathbf{x}^s + \sum_{t=1}^{N}\alpha^t
$$

subject to the constraints:

$$
\sum_{t}^{N}\alpha^t r^t = 0 \quad \text{and} \quad 0 \le \alpha^t \le C, \forall t
$$

The same ideas hold for the non-separable case. $\alpha = 0$ if $\mathbf{x}^t$ is on the correct side of the boundary with sufficient margin. For the support vectors, $\alpha^t < C$, and these are used to calculate $\mathbf{w}$. The instances not sufficiently far away from the boundary have $\alpha^t = C$. We can calculate w_0 by using the support vectors, because they have $\xi^t = 0$.

We can say that the number of support vectors is an upper-bound estimate for the expected number of errors. The expected test error rate is then

$$
E_N[P(error)] \le \frac{E_N[\#\text{ of support vectors}]}{N}
$$

where $E_N[\cdot]$ denotes the expectation over training sets of size N.

We define error if an instance is on the wrong side or if the margin is less than 1. This

is called the **hinge loss**, defined as

$$L_{\text{hinge}}(y^t, r^t) = \begin{cases} 0 & \text{if } y^t r^t \geq 1 \\ 1 - y^t r^t & \text{otherwise} \end{cases}$$

The penalty term C is the regularisation parameter fine-tuned via cross-validation. It defines the trade-off between margin maximisation and error minimisation. If C is too large, we have a high penalty for non-separable instances, and we may store many support vectors and therefore, overfit. If C is too small, we may find too simple solutions that underfit.

9.3 Kernel Trick

Recall that if the problem is nonlinear, instead of trying to fit a nonlinear model, we can map the problem to a *new space* by doing a nonlinear transformation using suitably chosen **basis functions**, and then use a linear model in this new space. The linear model in the new space corresponds to the nonlinear model in the original space.

Given a suitable set of basis functions, $\boldsymbol{\phi}$, we map the original d-dimensional space into the new k-dimensional space

$$\mathbf{z} = \boldsymbol{\phi}(\mathbf{x}) \quad \text{where} \quad z_j = \phi_j(\mathbf{x}), j = 1, \ldots, k$$

$$\mathbf{z} = \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_k \end{bmatrix} = \begin{bmatrix} \phi_1(\mathbf{x}) \\ \phi_2(\mathbf{x}) \\ \vdots \\ \phi_k(\mathbf{x}) \end{bmatrix}$$

In other words, $\boldsymbol{\phi} : \mathbb{R}^d \mapsto \mathbb{R}^k$, where k is generally much larger than d and may even be larger than N. The discriminant is then

$$g(\mathbf{z}) = \mathbf{w}^T \mathbf{z}$$

$$g(\mathbf{x}) = \mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}) = \sum_{j=1}^{k} w_j \phi_j(\mathbf{x})$$

Note that there is no separate w_0. The possible explosion of dimensionality from the new space is overcome because the complexity of the dual form depends on N, whereas the primal form would indeed depend on k. Using the basis functions, the same steps can be applied as before and the dual is defined as

$$L_d = -\frac{1}{2} \sum_{t=1}^{N} \sum_{s=1}^{N} \alpha^t \alpha^s r^t r^s \boldsymbol{\phi}\left(\mathbf{x}^t\right)^T \boldsymbol{\phi}(\mathbf{x}^s) + \sum_{t=1}^{N} \alpha^t$$

subject to the constraints:

$$\sum_{t}^{N} \alpha^t r^t = 0 \quad \text{and} \quad 0 \leq \alpha^t \leq C, \forall t$$

The idea in **kernel machines** is to replace the inner product of basis functions, $\boldsymbol{\phi}\left(\mathbf{x}^t\right)^T \boldsymbol{\phi}(\mathbf{x}^s)$, by a **kernel function** $K(\mathbf{x}^t, \mathbf{x}^s)$, between instances in the original input space. The *trick* is then to directly apply the kernel function in the original space, instead of mapping the instances first to the new $\mathbf{z}$-space and doing a dot product there.

$$L_d = -\frac{1}{2}\sum_{t=1}^{N}\sum_{s=1}^{N}\alpha^t\alpha^s r^t r^s K\left(\mathbf{x}^t, \mathbf{x}^s\right) + \sum_{t=1}^{N}\alpha^t$$

The discriminant then becomes

$$\begin{aligned} g(\mathbf{x}) &= \mathbf{w}^T\boldsymbol{\phi}(\mathbf{x}) \\ &= \sum_t \alpha^t r^t \boldsymbol{\phi}(\mathbf{x}^t)^T\boldsymbol{\phi}(\mathbf{x}) \\ &= \sum_t \alpha^t r^t K(\mathbf{x}^t, \mathbf{x}) \\ &= (\boldsymbol{\Lambda}\odot\mathbf{r})^T K(\mathbf{X}, \mathbf{x}) \end{aligned}$$

This last equivalent form uses matrix-vector notation where $\boldsymbol{\Lambda} = \left[\alpha^1, \alpha^2, \ldots, \alpha^N\right]^T$ and $\mathbf{r} = \left[r^1, r^2, \ldots, r^N\right]^T$, which are first multiplied element-wise (taking the Hamadard product $\odot$), and then its transpose is matrix-multiplied to the left of the kernel function $K(\mathbf{X}, \mathbf{x})$.

The matrix of kernel values, $\mathbf{K}$, where $\mathbf{K}_{t,s} = K(\mathbf{x}^t, \mathbf{x}^s)$, is called the **Gram matrix**, which should be *symmetric* and *positive semi-definite*.

9.4 Vectorial Kernels

Some of the popular kernel functions are hereafter enumerated:

Polynomials of degree q

$$K(\mathbf{x}^t, \mathbf{x}) = (\mathbf{x}^T\mathbf{x}^t + c)^q$$

where q is selected by the user and denotes the degree of the polynomial and c is a regularisation term that trades-off influences from higher order versus lower-order terms in the polynomial.

For example, when $d = 2$, $q = 2$, and $c = 1$, we have

$$\begin{aligned} K(\mathbf{x}, \mathbf{y}) &= (\mathbf{x}^T\mathbf{y} + 1)^2 \\ &= \left(\begin{bmatrix} x_1 & x_2 \end{bmatrix}^T \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} + 1\right)^2 \\ &= (x_1y_1 + x_2y_2 + 1)^2 \\ &= 1 + 2x_1y_1 + 2x_2y_2 + 2x_1x_2y_1y_2 + x_1^2y_1^2 + x_2^2y_2^2 \end{aligned}$$

The reason why the kernel trick is so useful is now evident, as the above corresponds to the

inner product of the basis functions:

$$\boldsymbol{\phi}(\mathbf{x}) = \begin{bmatrix} \phi_1(\mathbf{x}) \\ \phi_2(\mathbf{x}) \\ \phi_3(\mathbf{x}) \\ \phi_4(\mathbf{x}) \\ \phi_5(\mathbf{x}) \\ \phi_6(\mathbf{x}) \end{bmatrix} = \begin{bmatrix} 1 \\ \sqrt{2}x_1 \\ \sqrt{2}x_2 \\ \sqrt{2}x_1x_2 \\ x_1^2 \\ x_2^2 \end{bmatrix}$$

We can therefore avoid the overhead of computing the basis functions by directly computing the kernel function. When $d = 1$, we have the **linear kernel**, and when $c = 0$, the kernel is called *homogeneous*.

Radial-basis functions

$$K(\mathbf{x}^t, \mathbf{x}) = \exp\left[-\frac{||\mathbf{x}^t - \mathbf{x}||^2}{2s^2}\right]$$

defines a spherical kernel just like the kernel in *Parzen windows*, where $\mathbf{x}^t$ is the center and s, supplied by the user, defines the radius.

To generalise from the Euclidean distance, we can define the Mahalanobis kernel:

$$K(\mathbf{x}^t, \mathbf{x}) = \exp\left[-\frac{1}{2}(\mathbf{x}^t - \mathbf{x})^T\mathbf{S}^{-1}(\mathbf{x}^t - \mathbf{x})\right]$$

where $\mathbf{S}$ is the covariance matrix. Alternatively, in the most general case

$$K(\mathbf{x}^t, \mathbf{x}) = \exp\left[-\frac{\mathcal{D}(\mathbf{x}^t, \mathbf{x})}{2s^2}\right]$$

for some distance function $\mathcal{D}(\mathbf{x}^t, \mathbf{x})$.

Sigmoidal functions

$$K(\mathbf{x}^t, \mathbf{x}) = \tanh(\gamma\mathbf{x}^T\mathbf{x}^t + c)$$

where $\tanh(\cdot)$ has the same shape as sigmoid, except that it ranges between -1 and $+1$. The kernel parameters are γ and c, which allow for scaling and regularisation, respectively.

10 Neural Networks

Artificial neural networks are simplified models of the brain that use **neurons** as their processing units and **synapses** as the connections between neurons. The brain can be seen as an *information processing system*, from which we attempted to model it artificially. Analogously, we started modelling flying machines after birds, and then discovered aerodynamics, which brought us to the modern way of flying and made the use of feathers irrelevant. The same might hold true for the current ideas of neural networks, where we started replicating the brain with simple mathematical models, which might become irrelevant in favour of the true understanding of intelligence.

10.1 The Perceptron

The **perceptron** is the basic processing element. Its inputs may come from the environment or may be outputs of other perceptrons. Each input to the perceptron, $x_j \in \mathbb{R}, j = 1, \ldots, d$ has a *connection weight* or *synaptic weight*, $w_j \in \mathbb{R}$. In the simplest case, the output y is a weighted sum of the inputs

$$y = \sum_{j=1}^{d} w_j x_j + w_0$$

Here, w_0 is the intercept value that makes the model more general because it acts as an offset to the linear transformation. It is generally modelled as the weight coming from an extra **bias unit**, x_0, which is always $+1$. We can therefore write the output as

$$y = \mathbf{w}^T \mathbf{x}$$

where $\mathbf{w} = [w_0, w_1, \ldots, w_d]^T$ and $\mathbf{x} = [1, x_1, \ldots, x_d]^T$ are **augmented vectors** to also include the bias weight and input.

When $d = 1$, the perceptron implements a linear fit, $y = wx + w_0$. When $d > 1$, the perceptron implements a **hyperplane**, which can be used in a multivariate linear fit. The hyperplane divides the input space into two: the half-space where it is positive and the half-space where it is negative. The perceptron can be used to implement a linear discriminant function that separates two classes by checking the sign of the output. We can define $s(\cdot)$ as the *hard* threshold function

$$s(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

and then choose C_1 if $s(\mathbf{w}^T \mathbf{x}) > 0$, and C_2 otherwise.

Instead of a hard threshold, if we need the posterior probability, then we can use the sigmoid function as the output.

$$o = \mathbf{w}^T \mathbf{x}$$
$$y = \text{sigmoid}(o) = \frac{1}{1 + \exp\left[-\mathbf{w}^T \mathbf{x}\right]}$$

Figure 10.1a shows the computation as a network of neurons.

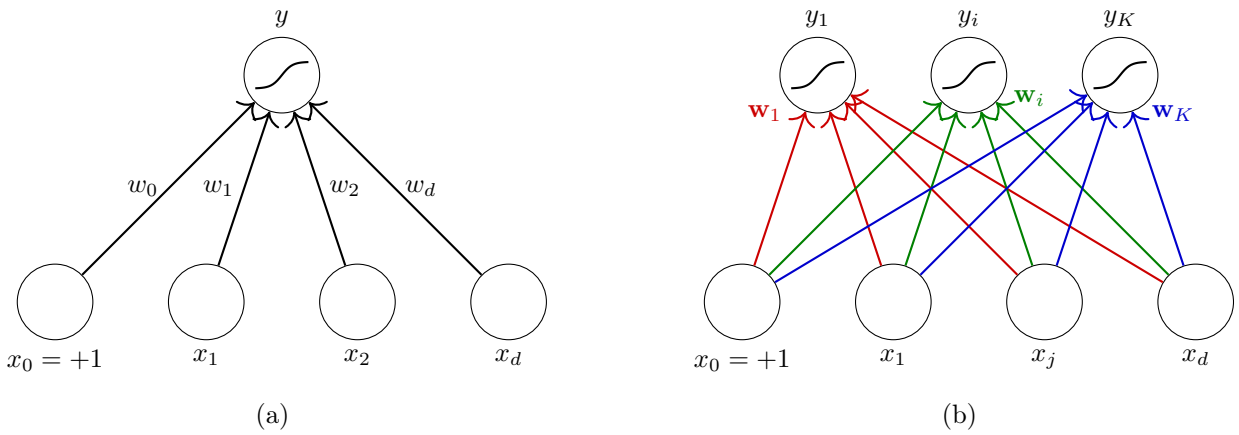


Figure 10.1: (a) simple perceptron with sigmoid activation function. (b) K parallel perceptrons with weight vector for each perceptron and sigmoid outputs for each y_i.

When there are $K > 2$ outputs, there are K perceptrons, each of which has a weight vector $\mathbf{w}_i$ associated with it. See figure 10.1b.

$$y_i = \mathbf{w}_i^T \mathbf{x}$$
$$\mathbf{y} = \mathbf{W}\mathbf{x}$$

where $\mathbf{W}$ is the $K \times (d+1)$ weight matrix in which the rows are $\mathbf{w}_i^T, i = 1, \ldots, K$. During classification, we choose C_i if $y_i = \max_k y_k$. In other words, the class corresponding to the maximum value of the output vector $\mathbf{y}$.

Again, if we need posterior probabilities instead of *hard* decisions (by using the max function), we can apply the **softmax function**. For this, we need the values of all outputs before applying softmax, which results in a two-stage process. First the weighted sums are calculated, and then, the softmax values.

$$\mathbf{o} = \mathbf{W}\mathbf{x}$$
$$y_i = \frac{\exp o_i}{\sum_j \exp o_j}$$
$$\mathbf{y} = \text{softmax}(\mathbf{o}) = \left[\frac{\exp o_1}{\sum_j \exp o_j} \quad \frac{\exp o_2}{\sum_j \exp o_j} \quad \cdots \quad \frac{\exp o_K}{\sum_j \exp o_j}\right]^T$$

10.2 Training the Perceptron

The weight values can be calculated **offline**, and plugged into the perceptron model to produce output values. However, in training neural networks, we often are not given the whole sample, but instead instances one by one. We need to use **online learning** in this case to update the weights of the network after *seeing* each instance. This approach is important because:

1. it saves us the cost of storing the training sample in external memory and storing the intermediate results during optimisation (unlike SVM's which can be costly with large samples).

2. the problem may be changing in time, which means that the sample distribution is not *fixed*, and thus, a training sample cannot be chosen a priori.

3. due to physical changes in the system, e.g., sensors may degrade over time.

In **online learning**, we do not write the error function over the whole sample but on individual instances. Starting with random initial weights, at each iteration, we adjust the parameters a little bit to minimise the error. The idea is to not forget what is already learned by the network and during optimisation. Gradient descent can be used if the error function is differentiable.

10.2.1 Example: Online Update for Regression

In the case of regression, the error on the single instance pair with index t, $(\mathbf{x}^t, r^t)$, is

$$E^t(\mathbf{w} \mid \mathbf{x}^t, r^t) = (r^t - y^t)^2 = (r^t - \mathbf{w}^T\mathbf{x}^t)^2$$

and for $j = 0, \ldots, d$, we take the derivative of the error with respect to w_j, which results in the online update

$$\Delta w_j^t = 2\eta(r^t - y^t)x_j^t \implies \Delta\mathbf{w}^t = 2\eta(r^t - y^t)\mathbf{x}^t \tag{10.1}$$

where η is the learning rate, which is gradually decreased in time for convergence. Often, the error is also multiplied by a scalar $1/2$ constant so that the 2 in the update equation vanishes. We can think of the $1/2$ constant as being part of the learning rate η. When we pick instances one by one in a random order, compute the update, and apply it, this is known as **stochastic gradient descent**. It is a good idea to normalise the inputs so that they are centered around 0 with the same scale, i.e., mean 0 and variance 1.

10.2.2 Example: Online Update for Classification

Using logistic discrimination, we can update after seeing each pattern, instead of summing them and doing the update after a complete pass over the training set. In the binary case, the output is $y^t = \text{sigmoid}(\mathbf{w}^T\mathbf{x}^t)$ and the cross-entropy error is

$$E^t(\mathbf{w} \mid \mathbf{x}^t, r^t) = -r^t \log y^t - (1 - r^t)\log(1 - y^t)$$

using gradient descent, we get the following online update rule for $j = 0, \ldots, d$

$$\Delta w_j^t = \eta(r^t - y^t)x_j^t \implies \Delta\mathbf{w}^t = \eta(r^t - y^t)\mathbf{x}^t \tag{10.2}$$

When there are $K > 2$ classes, we have $\mathbf{w}_i$ for each class, and we can use the softmax function from which the outputs are

$$y_i^t = \frac{\exp \mathbf{w}_i^T\mathbf{x}^t}{\sum_k \exp \mathbf{w}_k^T\mathbf{x}^t}$$

and the cross-entropy error becomes

$$E^t(\{\mathbf{w}_i\}_i \,|\, \mathbf{x}^t, \mathbf{r}^t) = -\sum_i r_i^t \log y_i^t$$

where $\{\mathbf{w}_i\}_i$ is the ith subset containing the ith row of $\mathbf{W}$. Using gradient descent, we get the following online update rule, for $i = 1, \ldots, K, j = 0, \ldots, d$:

$$\Delta w_{ij}^t = \eta(r_i^t - y_i^t)x_{ij}^t \implies \Delta \mathbf{w}_i^t = \eta(r_i^t - y_i^t)\mathbf{x}^t \tag{10.3}$$

10.2.3 Learning rule

The equations 10.1, 10.2, and 10.3 all have the same form, except for the regression update equation, but this is just a scalar that we can ignore. The general form is

$$\text{Update} = \text{LearningFactor} \cdot (\text{ActualOutput} - \text{PredictedOutput}) \cdot \text{Input}$$
$$\Delta \mathbf{w} = \eta \cdot (r - y) \cdot \mathbf{x}$$

- If the predicted output is equal to the actual output, $y = r$, no update is performed.
- The magnitude of the update *increases* as the difference between predicted and actual *increases*.
- If $y < r$, then the difference is positive and the update is positive if $\mathbf{x}$ is positive, and negative if $\mathbf{x}$ is negative. This increases the predicted output, which decreases the difference.
- If $y > r$, then the difference is negative and the update is negative if $\mathbf{x}$ is positive, and positive if $\mathbf{x}$ is negative. This decreases the predicted output, which decreases the difference.

Furthermore, the magnitude of the update depends on the input, $\mathbf{x}$. If the input is close to 0, its effect on the predicted output is small and therefore its weight is updated by a small amount. The greater the input, the greater the update of the weight. The magnitude also depends on η. If η is too large, then updates depend too much on recently seen instances, i.e., the system has short term memory. If η is too small, many updates are needed.

10.3 Multilayer Perceptron

A perceptron that has a single layer of weights, such as the ones in figure 10.1, can only approximate linear functions. In **feedforward networks** with an intermediate or **hidden layer** between the input and the output, such limitations do not apply, and we can approximate nonlinear functions. These models are called **multilayer perceptrons** (MLP).

The input vector $\mathbf{x}$ is fed to the input layer (including the bias), and the "activation" function propagates in the *forward direction* which results in the computation of the hidden units z_h. Each hidden unit is a perceptron consisting of a weighted sum fed to the nonlinear sigmoid activation function.

$$z_h = \text{sigmoid}(\mathbf{w}_h^T \mathbf{x}) = \frac{1}{1 + \exp\left[-\left(\sum_{j=1}^d w_{hj} x_j + w_{h0}\right)\right]}, h = 1, \ldots, H$$

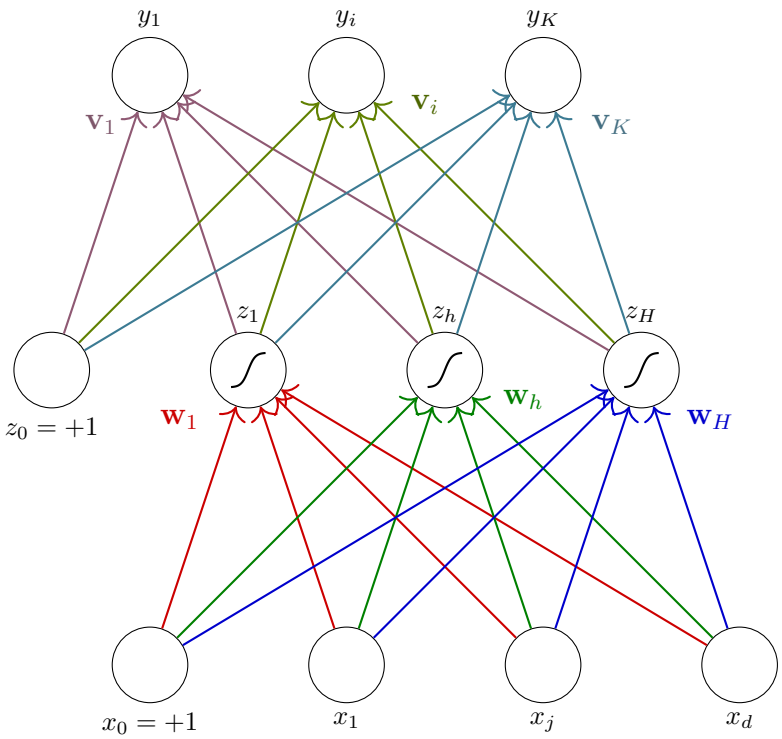


Figure 10.2: Structure of multilayer perceptron, where $x_j, j = 0, \ldots, d$ are the inputs, and $z_h, h = 1, \ldots, H$ are the hidden units. H is the dimensionality of the hidden space. x_0 and z_0 are the bias terms for each layer. $y_i, i = 1, \ldots, K$ are the output units.

Then, we use the output of the first layer, z_h, to compute the output of the second layer, y_i, as follows

$$y_i = \mathbf{v}_i^T \mathbf{z} = \sum_{h=1}^{H} v_{ih} z_h + v_{i0}$$

The indexing of the weights v_{ih} denote the connection from z_h to y_i. Likewise for the first layer, the weights w_{hj} denote the connection from x_j to z_h. Figure 10.2 illustrates this two-stage forward process.

The reason why the output is left without nonlinear transformation is because depending on the problem, we might apply a different transformation. In the case of regression, there is no non-linearity in the output. In a two-class classification task, we apply a sigmoid to the output layer, and when $K > 2$, we apply a softmax to the output layer.

Note that if the output of the hidden units, z_h, were linear, then the hidden layer becomes irrelevant, because a linear combination of linear combinations is *still* a linear combination. Sigmoid is the *continuous differentiable* version of *thresholding*.

10.4 MLP as a Universal Approximator

Any arbitrary function with continuous inputs and outputs can be approximated with an MLP. The MLP with two hidden layers is said to be a **universal approximator**. For every input case (or region), that region of the input can be delimited by hyperplanes on all using the hidden units in the first hidden layer. A hidden unit in the *second layer* then

combines them to bound the region. This gives a **piece-wise constant approximation** of the function. The accuracy may be increased by increasing the number of hidden units. Nevertheless, an MLP with one hidden layer and arbitrary amount of hidden units can learn any nonlinear function of the input.

10.5 Backpropagation Algorithm

Training an MLP is analogous to training a perceptron, except that there may be multiple layers, and thus, we need to propagate the update backwards through the network. Given a two-layer MLP, following from figure 10.2, the forward pass is

$$z_h = \text{sigmoid}(\mathbf{w}_h^T \mathbf{x}), h = 1, \ldots, H$$
$$y_i = \mathbf{v}_i^T \mathbf{z}, i = 1, \ldots, K$$

Using the chain rule and a defined error function E, the partial derivatives with respect to the parameters of the network are:

$$\frac{\partial E}{\partial w_{hj}} = \frac{\partial E}{\partial y_i} \frac{\partial y_i}{\partial z_h} \frac{\partial z_h}{\partial w_{hj}} \qquad \text{and} \qquad \frac{\partial E}{\partial v_{ij}} = \frac{\partial E}{\partial y_i} \frac{\partial y_i}{\partial v_{ij}}$$

In the case of nonlinear regression with a single output, we have

$$y^t = \sum_{h=1}^{H} v_h z_h^t + v_0$$

The error function over the whole sample is the mean squared error:

$$E(\mathbf{W}, \mathbf{V} \mid \mathcal{X}) = \frac{1}{2} \sum_t (r^t - y^t)^2$$

where we can divide by 2 instead of N because the learning rate η will account for the factor. This simplifies to the least-squares update rule to update the weights of the second layer:

$$\Delta v_h = \eta \sum_t (r^t - y^t) z_h^t$$
$$\Delta \mathbf{v} = \eta \sum_t (r^t - y^t) \mathbf{z}^t$$

For the weights in the first layer, the forward flow depends on several sub-expressions, i.e.,

$$a_h = \mathbf{w}_h^T \mathbf{x}, h = 1, \ldots, H$$
$$z_h = \text{sigmoid}(a_h), h = 1, \ldots, H$$
$$y_i = \mathbf{v}_i^T \mathbf{z}$$

Therefore, we cannot use the least-squares rule directly and need to use the chain rule.

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

$$
\begin{aligned}
&= -\eta \sum_t \frac{\partial E^t}{\partial y^t} \frac{\partial y^t}{\partial z_h^t} \frac{\partial z_h^t}{\partial w_{hj}} \\
&= -\eta \sum_t \underbrace{-(r^t - y^t)}_{\partial E^t/\partial y^t} \underbrace{v_h}_{\partial y^t/\partial z_h^t} \underbrace{z_h^t(1 - z_h^t)x_j^t}_{\partial z_h^t/\partial w_{hj}} \\
&= \eta \sum_t (r^t - y^t) v_h z_h^t (1 - z_h^t) x_j^t
\end{aligned}
$$

This last update can also be written as

$$\Delta \mathbf{W} = \eta((\mathbf{r} - \mathbf{y})^T \mathbf{V} \odot \mathbf{Z}(1 - \mathbf{Z}))\mathbf{X}$$

with sample size n, where

- $\mathbf{X}$ is the $n \times d$ matrix containing the batch sample, $\mathbf{x}^t$, as row vectors. This is equivalent to the transpose of the instances $\mathbf{x} \in \mathcal{X}$, because $\mathcal{X}$ contains column vectors.
- $\mathbf{Z}$ is the $H \times n$ matrix containing $\mathbf{z}^t$ as column vectors.
- $\mathbf{V}^T$ is the $H \times 1$ matrix (i.e., vector in this case because output dimension is scalar y) containing the transposed weights of the second layer.
- $\mathbf{r}$ and $\mathbf{y}$ are $n \times 1$ vectors containing the ground truth and the prediction, respectively. Then, $(\mathbf{r} - \mathbf{y})^T$ is a $1 \times n$ vector.
- $(\mathbf{r} - \mathbf{y})^T \mathbf{V}$ broadcasts both vectors to have matching dimensions which results in a $H \times n$ matrix.
- $\odot$ is the Hamadard element-wise multiplication, which can be applied because left and right have matching $H \times n$ dimensions. Finally, matrix multiplying $\mathbf{X}$ to the right of this yields the resulting $H \times d$ matrix.

A complete pass over the training sample is called an **epoch**. Based on the size of the sample n, there are different approaches to updating the weights.

- **Online learning** (*stochastic gradient descent*) if $n = 1$, i.e., randomly choose instance and update based on that instance.
- **Batch learning** (*gradient descent*) if $n = N$, i.e., take the whole training set and update based on the average of the individual gradients.
- **Minibatch learning** if $n = m$ where m is a small batch, for example 32, from which we do a single update.

Online learning had the consequence that the weights may be affected by outlier instances, whereas in batch learning, we take the average over individual gradients to update the weights, but this can be slow on large datasets. Minibatch is therefore a nice in-between.

In the case of regression with multiple outputs, several regression problems are learned at the same time. Therefore, we have

$$y_i^t = \sum_{h=1}^{H} v_{ih} z_h^t + v_{i0}$$

and the error becomes

$$E(\mathbf{W}, \mathbf{V} \mid \mathcal{X}) = \frac{1}{2} \sum_t \sum_i (r_i^t - y_i^t)^2$$

The batch update rules are then

$$\Delta v_{ih} = \eta \sum_t (r_i^t - y_i^t) z_h^t$$

$$\Delta w_{hj} = \eta \sum_t \left[\sum_i (r_i^t - y_i^t) v_{ih} \right] z_h^t (1 - z_h^t) x_j^t$$

Here, $\sum_i (r_i^t - y_i^t) v_{ih}$ can be seen as the accumulated backpropagated error of hidden unit h from all output units.

Note that in this case, all output units $y_i, i = 1, \ldots, K$ share the same hidden units and thus use the same hidden representation $z_h, h = 1, \ldots, H$. Therefore, we assume that the output units are related to the same prediction problems. Alternatively, we can train separate MLPs for the separate regression problems, each with its own separate hidden units.

The update equations for regression and classification are identical. However, this does not mean that the values of the weights will be identical. The loss functions are also different, one being the mean squared loss, while the other is the cross-entropy loss.

10.6 Overtraining

An MLP with d inputs, H hidden units, and K outputs has $H(d + 1)$ weights in the first layer and $K(H+1)$ weights in the second layer. Both space and time complexity of the MLP are $\mathcal{O}(H \cdot (K + d))$. Let e denote the number of training epochs, then the time complexity becomes $\mathcal{O}(e \cdot H \cdot (K + d))$.

The values for d and K are predetermined, depending on the dataset and problem at hand. The value for H is a hyperparameter. An overcomplex model will *memorise* the noise in the data, which increases the variance between models and decreases the bias that the solution will entail. A large H will deteriorate the generalisation accuracy.

Similarly, as training continues for more epochs, the training error decreases, but the error on the validation set increases. Initially, all the weights are close to 0, and as training continues, some of the weights start to be utilised and really become parameters to the system. But as training continues for too long, every weight moves away from zero and effectively becomes a parameter. Therefore, learning should be *stopped early* to alleviate this problem of overtraining. The optimal point to stop and the optimal value for H are determined via cross-validation.

11 Clustering

We saw that in parametric approaches, instances from a class form a single group in the d-dimensional space, where the center and the shape of this group is given by the *mean* and *covariance*, respectively. In many applications, however, the sample is not one group. There may be several groups.

For example, the number seven (7) is written differently between the European writing 7̶ (that has a horizontal bar distinguishing itself from the number 1) and the American writing 7. If the sample contains examples from both continents, the class for the number seven should be represented as the disjunction of the two groups. If each group can be represented by a Gaussian, the class is a *mixture* of two Gaussians. We call this approach **semiparametric density estimation**.

11.1 Mixture Densities

The **mixture density** is written as

$$P(\mathbf{x}) = \sum_{i=1}^{k} P(\mathbf{x} \mid G_i) P(G_i)$$

where G_i are the mixture components. These are also called *groups* or *clusters*. $P(\mathbf{x} \mid G_i)$ are the **component densities** and denote the probability of $\mathbf{x}$ given that it belongs to cluster G_i, and $P(G_i)$ are the **mixture proportions**. The number of components (clusters), k, is a hyperparameter specified beforehand.

Learning then correspond to estimating the component densities, $P(\mathbf{x} \mid G_i)$, and proportions, $P(G_i)$. When we assume that the component densities obey a parametric model, we only need to estimate their parameters. If the component densities are multivariate Gaussian, we have $P(\mathbf{x} \mid G_i) \sim \mathcal{N}(\boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i)$, and the set of parameters for the mixture becomes

$$\boldsymbol{\Phi} = \{P(G_i), \boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i\}_{i=1}^{k}$$

which are the parameters that should be estimated from the iid sample $\mathcal{X} = \{\mathbf{x}^t\}_{t=1}^{N}$.

Parametric classification is a simplified form of a mixture model where the groups (clusters), G_i, correspond to the classes, C_i, and therefore, component densities, $P(\mathbf{x} \mid G_i)$, correspond to class densities, $P(\mathbf{x} \mid C_i)$, and mixture proportions, $P(G_i)$, correspond to class priors, $P(C_i)$:

$$P(\mathbf{x}) = \sum_{i=1}^{K} P(\mathbf{x} \mid C_i) P(C_i)$$

In this *supervised learning case*, we know how many groups there are, i.e., number of classes, and learning the parameters is trivial because we are given the labels, $\mathbf{r}^t$, where $r_i^t = 1$ if $\mathbf{x}^t \in C_i$, and 0 otherwise. Using maximum likelihood, the parameters are calculated as follows:

$$\hat{P}(C_i) = \frac{\sum_{t=1}^{N} r_i^t}{N}$$

$$\mathbf{m}_i = \frac{\sum_{t=1}^{N} r_i^t \mathbf{x}^t}{\sum_{t=1}^{N} r_i^t}$$

$$\mathbf{S}_i = \frac{\sum_{t=1}^{N} r_i^t (\mathbf{x}^t - \mathbf{m}_i)(\mathbf{x}^t - \mathbf{m}_i)^T}{\sum_{t=1}^{N} r_i^t}$$

In this section, instead of being given the labels, we consider the **unsupervised case**, where we are given the sample $\mathcal{X} = \{\mathbf{x}^t\}_{t=1}^N$. Therefore, we estimate the labels and then the parameters given the labels.

1. Estimate the labels, r_i^t, i.e., the component that a given instance belongs to.
2. Estimate the parameters of the components, given the set of instances that belong to them.

11.2 k-Means Clustering

Given a sample $\mathcal{X} = \{\mathbf{x}^t\}_{t=1}^N$, we initialise k reference vectors, $\mathbf{m}_j, j = 1, \ldots, k$, which correspond to the means of k clusters. Therefore, $\mathbf{M}$ should be a $d \times k$ dimensional matrix.

Assuming we have computed the vectors $\mathbf{m}_j$, we can assign labels to a data point $\mathbf{x}^t$ by looking at the closest mean cluster $\mathbf{m}_j$ using the Euclidean distance:

$$||\mathbf{x}^t - \mathbf{m}_i|| = \min_j ||\mathbf{x}^t - \mathbf{m}_j||$$

Then, we can estimate the labels $\mathbf{b}^t$ using this distance with

$$b_i^t = \begin{cases} 1 & \text{if } ||\mathbf{x}^t - \mathbf{m}_i|| = \min_j ||\mathbf{x}^t - \mathbf{m}_j|| \\ 0 & \text{otherwise} \end{cases}$$

To compute $\mathbf{m}_i$, notice that when we represent an instance $\mathbf{x}^t$ by $\mathbf{m}_i$, there is an error that is proportional to the distance $||\mathbf{x}^t - \mathbf{m}_i||$. This error should be minimal by having these distances as small as possible. The **total reconstruction error** is defined as

$$E(\mathbf{M} \,|\, \mathcal{X}) = \sum_t \sum_i b_i^t ||\mathbf{x}^t - \mathbf{m}_i||^2$$

We need to use an iterative optimisation procedure to solve this problem, as it cannot be solved analytically. We start with initialising $\mathbf{m}_i$ randomly. Then, at each iteration, we first compute the estimates for b_i^t for all $\mathbf{x}^t$. If b_i^t is 1, we say that $\mathbf{x}^t$ belongs to the cluster $\mathbf{m}_i$. Having these label estimates, we minimise the reconstruction error by taking the derivative with respect to $\mathbf{m}_i$, and setting it equal to zero.

$$\frac{\partial}{\partial \mathbf{m}_i} \sum_t \sum_i b_i^t ||\mathbf{x}^t - \mathbf{m}_i||^2 \tag{11.1}$$

$$\sum_t \frac{\partial}{\partial \mathbf{m}_i} \sum_i b_i^t ||\mathbf{x}^t - \mathbf{m}_i||^2 \tag{11.2}$$

$$\sum_t \frac{\partial}{\partial \mathbf{m}_i} b_i^t ||\mathbf{x}^t - \mathbf{m}_i||^2 \tag{11.3}$$

$$\sum_t \frac{\partial}{\partial \mathbf{m}_i} b_i^t (\mathbf{x}^t - \mathbf{m}_i)^T (\mathbf{x}^t - \mathbf{m}_i) \tag{11.4}$$

$$\sum_t b_i^t \cdot 2 \cdot (\mathbf{x}^t - \mathbf{m}_i) \cdot -1 \tag{11.5}$$

Note that in equation 11.3 the summation vanishes because the derivative of every term becomes 0, except for the ith term. Also, in equation 11.5, recall that the derivative of $\mathbf{v}^T\mathbf{v}$ with respect to $\mathbf{v}$ is simply $2\mathbf{v}$, and with the chain rule, we get an additional -1 factor due to $-\mathbf{m}_i$.

Then, we set the derivative to zero and solve for $\mathbf{m}_i$:

$$\begin{aligned}
-2\sum_t b_i^t(\mathbf{x}^t - \mathbf{m}_i) &= 0 \\
\sum_t b_i^t \mathbf{x}^t - b_i^t \mathbf{m}_i &= 0 \\
\sum_t b_i^t \mathbf{x}^t - \mathbf{m}_i \sum_t b_i^t &= 0 \\
-\mathbf{m}_i \sum_t b_i^t &= -\sum_t b_i^t \mathbf{x}^t \\
\mathbf{m}_i &= \frac{\sum_t b_i^t \mathbf{x}^t}{\sum_t b_i^t}
\end{aligned}$$

In other words, the *reference vector* $\mathbf{m}_i$ is set to the mean of all instances that it represents. Note that this is the same as the formula to compute the mean in the supervised case, except that estimated labels b_i^t are used instead of ground truth labels r_i^t. This is iterative because once the new $\mathbf{m}_i$ are calculated, the b_i^t values change and need to be recalculated, which in turn affects $\mathbf{m}_i$. We repeat until convergence of $\mathbf{m}_i$.

The final $\mathbf{m}_i$ highly depends on the initial $\mathbf{m}_i$. There are various methods for initialisation:

- Randomly selected k instances as the initial $\mathbf{m}_i$.
- The sample mean plus small random vectors can be added as the initial $\mathbf{m}_i$.
- Compute the principal component, divide its range into k equal intervals, partition the data into k groups, and take the means of these k groups as the initial centers $\mathbf{m}_i$.
- **k-means++**: choose clusters based on probability distribution of closest already chosen clusters.

We can measure how good the clusters are when they would be used for classification with the **purity metric**. Purity is an external evaluation criterion of cluster quality[6]:

$$\text{purity} = \frac{1}{N} \sum_{i=1}^{k} \max_j |G_i \cap C_j|$$

[6]See `https://nlp.stanford.edu/IR-book/html/htmledition/evaluation-of-clustering-1.html`

In other words, we count the amount of correct instances in cluster i and class j and take the maximum of these with respect to j, then, we sum over all of these j-counts. This corresponds to taking the maximum of the rows in the confusion matrix, and then summing these maxima, where rows represent clusters G_i and columns represent C_j. This sum is divided by the sample size to obtain the purity measure.

11.3 Gaussian Mixture Models and the EM Algorithm

In k-means, the idea of clustering is to find mean vectors $\mathbf{m}_i$ that minimise the total reconstruction error. The probabilistic approach is to assign soft values to variables (instead of hard 0/1 values), and as such, we can apply the **Expectation-Maximisation** (EM) algorithm. In the context of clustering, this corresponds to a *Gaussian Mixture Model* (GMM), where we are looking for the component density parameters that maximise the likelihood of the sample. The likelihood of the sample $\mathcal{X} = \{\mathbf{x}^t\}_{t=1}^N$ is

$$P(\mathbf{x} \mid \mathbf{\Phi}) = \sum_{i=1}^{k} P(\mathbf{x} \mid G_i) P(G_i)$$

And the **log likelihood**, denoted as $\mathcal{L}(\mathbf{\Phi} \mid \mathcal{X})$, is:

$$\begin{aligned}
\mathcal{L}(\mathbf{\Phi} \mid \mathcal{X}) &= \log \prod_t P(\mathbf{x}^t \mid \mathbf{\Phi}) \\
&= \sum_t \log P(\mathbf{x}^t \mid \mathbf{\Phi}) \\
&= \sum_t \log \sum_{i=1}^{k} P(\mathbf{x} \mid G_i) P(G_i)
\end{aligned}$$

where $\mathbf{\Phi}$ includes the priors $P(G_i)$ and the parameters of the component densities $P(\mathbf{x} \mid G_i)$, e.g., mean $\mathbf{m}$ and covariance matrix $\mathbf{S}$ if we assume Gaussian component densities.

Solving analytically is not possible and we use iterative approaches, i.e., expectation-maximisation. EM is used in *maximum likelihood estimation* (MLE) when there are two sets of random variables; one set X is observed, and another set Z is *hidden*. The goal of the algorithm is to find the parameter vector $\mathbf{\Phi}$ that maximises the likelihood $\mathcal{L}(\mathbf{\Phi} \mid \mathcal{X})$ of the observed values of X.

In cases where this is not possible, we associate the hidden variables Z and express the underlying model using both X and Z. In other words, we maximise the joint distribution of X and Z, which is referred to as maximising the **complete likelihood** $\mathcal{L}_c(\mathbf{\Phi} \mid \mathcal{X}, \mathcal{Z})$.

Since Z is not observed, we instead work with the expectation of Z given $\mathcal{X}$ and the current parameter vector $\mathbf{\Phi}^l$, where l indexes the iterations. The **expectation** is denoted by $\mathcal{Q}$ and corresponds to the E-step of the algorithm. Then, in the **maximisation** M-step, we look for the new parameters values, $\mathbf{\Phi}^{l+1}$, that maximises the expectation.

$$\begin{aligned}
\text{E-step} \quad &: \quad \mathcal{Q}(\mathbf{\Phi} \mid \mathbf{\Phi}^l) = E\left[\mathcal{L}_c(\mathbf{\Phi} \mid \mathcal{X}, \mathcal{Z}) \mid \mathcal{X}, \mathbf{\Phi}^l\right] \\
\text{M-step} \quad &: \quad \mathbf{\Phi}^{l+1} = \arg\max_{\mathbf{\Phi}} \mathcal{Q}(\mathbf{\Phi} \mid \mathbf{\Phi}^l)
\end{aligned}$$

To derive a **lower bound** for the expectation, we can use Jensen's inequality, which states

that:

1. for any **convex** function $g(x)$, we have $E[g(x)] \geq g(E[x])$; and
2. for any **concave** function $g(x)$, we have $E[g(x)] \leq g(E[x])$

The likelihood function is not easy to maximise analytically, mainly due to the $\log(\sum_i)$ part. Assuming that the log in the likelihood refers to the natural logarithm, we know that it is concave on its entire domain. Therefore, we can add an additional (redundant) factor, $P(G_i \mid \mathbf{x}^t)/P(G_i \mid \mathbf{x}^t)$, so that it conforms to Jensen's inequality and then derive the lower bound. In other words,

$$\mathcal{L}(\mathbf{\Phi} \mid \mathcal{X}) = \sum_t \log \sum_{i=1}^k P(\mathbf{x} \mid G_i) P(G_i) \underbrace{\frac{P(G_i \mid \mathbf{x}^t)}{P(G_i \mid \mathbf{x}^t)}}_{=1}$$

Recall that the expectation $E[\cdot]$ is really just a weighted average where the weights are probabilities, i.e., $E[x] = \sum_i P(x_i)x_i$. This is the reason why the added factor is relevant, as it acts as the probability weight for the expectation. Now, let

$$\begin{aligned} g(x) &= \log x \\ u &= \frac{P(\mathbf{x}^t \mid G_i)P(G_i)}{P(G_i \mid \mathbf{x}^t)} \\ E[u] &= \sum_{i=1}^k P(G_i \mid \mathbf{x}^t) u \end{aligned}$$

And applying Jensen's inequality, we obtain the lower bound for the expectation:

$$\sum_t \underbrace{\log \sum_{i=1}^k P(G_i \mid \mathbf{x}^t) \frac{P(\mathbf{x} \mid G_i)P(G_i)}{P(G_i \mid \mathbf{x}^t)}}_{g(E[x])} \geq \sum_t \underbrace{\sum_{i=1}^k P(G_i \mid \mathbf{x}^t) \log \frac{P(\mathbf{x}^t \mid G_i)P(G_i)}{P(G_i \mid \mathbf{x}^t)}}_{E[g(x)]}$$

This lower bound is much easier to optimise. A further simplification is the removal of the term in the denominator, $P(G_i \mid \mathbf{x}^t)$, as it will not play a part in the M-step. Thus, the expectation becomes:

$$\begin{aligned} \mathcal{Q}(\mathbf{\Phi} \mid \mathbf{\Phi}^l) &= \sum_t \sum_{i=1}^k P(G_i \mid \mathbf{x}^t) \log P(\mathbf{x}^t \mid G_i) P(G_i) \\ &= \sum_t \sum_{i=1}^k P(G_i \mid \mathbf{x}^t) \left[\log P(\mathbf{x}^t \mid G_i) + \log P(G_i)\right] \\ &= \sum_t \sum_{i=1}^k h_i^t \left[\log P_i(\mathbf{x}^t \mid \mathbf{\Phi}) + \log \pi_i\right] \end{aligned}$$

where

$$h_i^t = P(G_i \mid \mathbf{x}^t) = \underbrace{\frac{P(\mathbf{x}^t \mid G_i)P(G_i)}{\sum_j P(\mathbf{x}^t \mid G_j)P(G_j)}}_{\text{Bayes' rule}}, \qquad P_i(\mathbf{x}^t \mid \mathbf{\Phi}) = P(\mathbf{x}^t \mid G_i), \qquad \pi_i = P(G_i)$$

Thus, in the E-step, we calculate h_i^t given the sample $\mathcal{X}$ and the current parameters, $\mathbf{\Phi}^l, \pi^l$, and then maximise the expectation, $\mathcal{Q}$, with respect to the parameters, to find $\mathbf{\Phi}^{l+1}, \pi^{l+1}$. For each iteration, the lower bound can be solved analytically and guarantees that the likelihood increases monotonically. However, EM is not guaranteed to find the global optimum.

Starting with π_i, we take the derivative of the expectation with respect to π_i, set it equal to zero and solve for π_i. Since we know that the sum over the priors must equal to 1, i.e., $\sum_i \pi_i = 1$, we introduce a Langrange multiplier λ to constrain it:

$$\begin{aligned}
\frac{\partial \mathcal{Q}(\mathbf{\Phi} \mid \mathbf{\Phi}^l)}{\partial \pi_i} &= \frac{\partial}{\partial \pi_i} \sum_t \sum_{i=1}^k h_i^t \left[\log P_i(\mathbf{x}^t \mid \mathbf{\Phi}) + \log \pi_i\right] - \lambda \left(\sum_i \pi_i - 1\right) \\
&= \sum_t \frac{\partial}{\partial \pi_i} \left(h_i^t \log P_i(\mathbf{x}^t \mid \mathbf{\Phi}) + h_i^t \log \pi_i\right) - \frac{\partial}{\partial \pi_i} \lambda \left(\sum_i \pi_i - 1\right) \\
&= \sum_t h_i^t \frac{1}{\pi_i} - \lambda
\end{aligned}$$

Setting this equal to zero, we get $\pi_i = \sum_t h_i^t / \lambda, \forall i = 1, \ldots, k$. To find the value of λ, we sum over all i, and solve for λ:

$$\begin{aligned}
\sum_i \pi_i &= \frac{\sum_i \sum_t h_i^t}{\lambda} \\
1 &= \frac{N}{\lambda} \\
\lambda &= N
\end{aligned}$$

Informally, N data points are each softly assigned to k components where the sum over k components is 1. Therefore, we get:

$$\pi_i^{l+1} = \frac{\sum_t h_i^t}{N}$$

which is analogous to the calculations of the priors in the supervised case.

For the parameters $\mathbf{\Phi}$, the derivative of the expectation with respect to $\mathbf{\Phi}$ is unconstrained. If we assume $\hat{P}_i(\mathbf{x}^t \mid \mathbf{\Phi}) \sim \mathcal{N}(\mathbf{m}_i, \mathbf{S}_i)$, then we can replace $P_i(\mathbf{x}^t \mid \mathbf{\Phi})$ with the Gaussian equation and take the derivative with respect to $\mathbf{m}_i$ and $\mathbf{S}_i$. This gives the M-step:

$$\begin{aligned}
\frac{\partial \mathcal{Q}(\mathbf{\Phi} \mid \mathbf{\Phi}^l)}{\partial \mathbf{m}_i} &= \frac{\partial}{\partial \mathbf{m}_i} \sum_t \sum_{i=1}^k h_i^t \left[\log P_i(\mathbf{x}^t \mid \mathbf{\Phi}) + \log \pi_i\right] \\
&= \sum_t \frac{\partial}{\partial \mathbf{m}_i} h_i^t \log \hat{P}_i(\mathbf{x}^t \mid \mathbf{\Phi}) \\
&= \sum_t \frac{\partial}{\partial \mathbf{m}_i} h_i^t \log \left(\frac{1}{(2\pi)^{d/2} |\mathbf{S}_i|^{1/2}} \exp\left[-\frac{1}{2}(\mathbf{x}^t - \mathbf{m}_i)^T \mathbf{S}_i^{-1} (\mathbf{x}^t - \mathbf{m}_i)\right]\right) \\
&= \sum_t \frac{\partial}{\partial \mathbf{m}_i} h_i^t \left(\log \left(\frac{1}{(2\pi)^{d/2} |\mathbf{S}_i|^{1/2}}\right) - \frac{1}{2}(\mathbf{x}^t - \mathbf{m}_i)^T \mathbf{S}_i^{-1} (\mathbf{x}^t - \mathbf{m}_i)\right) \\
&= \sum_t \frac{\partial}{\partial \mathbf{m}_i} h_i^t \left(\log(1) - \frac{d}{2}\log(2\pi) + \frac{1}{2}\log|\mathbf{S}_i| - \frac{1}{2}(\mathbf{x}^t - \mathbf{m}_i)^T \mathbf{S}_i^{-1} (\mathbf{x}^t - \mathbf{m}_i)\right)
\end{aligned}$$

$$
\begin{aligned}
&= \sum_t \frac{\partial}{\partial \mathbf{m}_i} h_i^t \left(-\frac{d}{2}\log(2\pi) + \frac{1}{2}\log|\mathbf{S}_i| - \frac{1}{2}(\mathbf{x}^t - \mathbf{m}_i)^T \mathbf{S}_i^{-1}(\mathbf{x}^t - \mathbf{m}_i) \right) \\
&= \sum_t h_i^t \left(-\frac{1}{2} \cdot -2\mathbf{S}_i^{-1}(\mathbf{x}^t - \mathbf{m}_i) \right) \\
&= \sum_t h_i^t \mathbf{S}_i^{-1}(\mathbf{x}^t - \mathbf{m}_i)
\end{aligned}
$$

Notice that in the derivation, we used the rule $\partial(\mathbf{x}-\mathbf{s})^T\mathbf{W}(\mathbf{x}-\mathbf{s})/\partial\mathbf{x} = -2\mathbf{W}(\mathbf{x}-\mathbf{s})$, which is only possible if the $\mathbf{W}$ matrix is symmetric. Setting this equal to zero and solving for $\mathbf{m}_i$, we get:

$$\sum_t h_i^t \mathbf{S}_i^{-1}(\mathbf{x}^t - \mathbf{m}_i) = 0 \tag{11.1}$$

$$\sum_t h_i^t \mathbf{x}^t - \sum_t h_i^t \mathbf{m}_i = 0 \tag{11.2}$$

$$\sum_t h_i^t \mathbf{m}_i = \sum_t h_i^t \mathbf{x}^t \tag{11.3}$$

$$\mathbf{m}_i^{l+1} = \frac{\sum_t h_i^t \mathbf{x}^t}{\sum_t h_i^t} \tag{11.4}$$

which is analogous to the mean estimate in the supervised case. Notice in step 11.2 that $\mathbf{S}_i^{-1}$ is implicitly factored out and multiplied by $\mathbf{S}_i$ to simplify the expression. For the covariance matrix, we get the same steps, i.e., take derivative of with respect to $\mathbf{S}_i$, set it equal to zero, and solve for $\mathbf{S}_i$.

$$\mathbf{S}_i^{l+1} = \frac{\sum_t h_i^t (\mathbf{x}^t - \mathbf{m}_i^{l+1})(\mathbf{x}^t - \mathbf{m}_i^{l+1})^T}{\sum_t h_i^t}$$

And finally, the expectations h_i^t become:

$$
\begin{aligned}
h_i^t &= \frac{\pi_i \hat{P}_i(\mathbf{x}^t \mid \mathbf{\Phi})}{\sum_j \pi_j \hat{P}_j(\mathbf{x}^t \mid \mathbf{\Phi})} \\
&= \frac{\pi_i |\mathbf{S}_i^{-1/2}| \exp\left[-(1/2)(\mathbf{x}^t - \mathbf{m}_i)^T \mathbf{S}_i^{-1}(\mathbf{x}^t - \mathbf{m}_i)\right]}{\sum_j \pi_j |\mathbf{S}_j^{-1/2}| \exp\left[-(1/2)(\mathbf{x}^t - \mathbf{m}_j)^T \mathbf{S}_j^{-1}(\mathbf{x}^t - \mathbf{m}_j)\right]}
\end{aligned}
$$

The estimated *soft* labels h_i^t replace the actual unknown *hard* labels r_i^t. We can initialise EM by k-means, and run EM instead after a few iterations. Just like parametric classification, we can make simplifying assumptions, i.e., assuming a shared covariance matrix or independent variables via a diagonal covariance matrix. The latter reduces to the case of k-means clustering, which implies that k-means is a special case of EM where input variables are assumed independent with equal and shared variance, equal priors, and *hard* labels. In other words, k-means creates circles in the input space, whereas EM creates ellipses of arbitrary shapes, orientations, and coverage proportions.

11.4 Mixtures of Latent Variable Models

Given a sample $\mathcal{X} = \{\mathbf{x}^t\}_{t=1}^N$ with d-dimensional vectors $\mathbf{x}^t$, if d is large while N is small, one risks overfitting. To overcome this problem, we can reduce the amount of parameters. Assuming a shared covariance matrix may not be a good idea if the clusters have really

different shapes. Assuming diagonal matrices removes all correlations, which makes it worse.

The alternative is to do dimensionality reduction in the clusters, which decreases the number of parameters while capturing the correlations. **Factor analysis** in the clusters corresponds to looking for *latent* or *hidden variables* or *factors* that **generate** the data in the clusters.

$$P(\mathbf{x}^t \mid G_i) \sim \mathcal{N}(\mathbf{m}_i, \mathbf{V}_i\mathbf{V}_i^T + \mathbf{\Psi}_i)$$

where $\mathbf{V}_i$ are the **factor loadings** and $\mathbf{\Psi}_i$ are the specific variances of cluster G_i. Mixture models can be extended to find **mixtures of factor analyzers**. The above formulation for the component densities can be used instead of the regular parameters of the Gaussian in the E-step, and in the M-step, the parameters can be maximised, i.e., $\mathbf{V}_i$ and $\mathbf{\Psi}_i$ instead of $\mathbf{S}_i$.

Ghahramani and Hinton [6] provide the derivation for mixtures of factor analyzers. In regular maximum likelihood factor analysis, d-dimensional $\mathbf{x}$ is modelled using k-dimensional factors $\mathbf{z}$, where $k < d$. The generative model is given by:

$$\mathbf{x} = \mathbf{V}\mathbf{z} + \mathcal{N}(0, \mathbf{\Psi})$$

The properties that we need for mixtures of factor analyzers are the expectation of $\mathbf{z}$ and $\mathbf{z}\mathbf{z}^T$ given $\mathbf{x}$:

$$\begin{aligned} E[\mathbf{z} \mid \mathbf{x}] &= \boldsymbol{\beta}\mathbf{x} \\ E[\mathbf{z}\mathbf{z}^T \mid \mathbf{x}] &= \boldsymbol{I} - \boldsymbol{\beta}\mathbf{V} + \boldsymbol{\beta}\mathbf{x}\mathbf{x}^T\boldsymbol{\beta}^T \end{aligned}$$

where $\boldsymbol{\beta} = \mathbf{V}^T(\mathbf{\Psi} + \mathbf{V}\mathbf{V}^T)^{-1}$. To translate this to the case of mixtures, in the E-step, we compute h_i^t in the same way as before. In the M-step, we update the parameters as follows

$$\begin{aligned} \begin{bmatrix} \mathbf{V}_i^{l+1} & \mathbf{m}_i \end{bmatrix} = \tilde{\mathbf{V}}_i^{l+1} &= \left(\sum_t h_i^t \mathbf{x}^t E[\tilde{\mathbf{z}} \mid \mathbf{x}^t, G_i]^T \right) \left(\sum_s h_i^s E[\tilde{\mathbf{z}}\tilde{\mathbf{z}}^T \mid \mathbf{x}^s, G_i] \right)^{-1} \\ \mathbf{\Psi}^{l+1} &= \frac{1}{N} diag \left\{ \sum_t \sum_i h_i^t (\mathbf{x}^t - \tilde{\mathbf{V}}_i^{l+1} E[\tilde{\mathbf{z}} \mid \mathbf{x}^t, G_i])(\mathbf{x}^t)^T \right\} \\ \pi_i^{l+1} &= \frac{\sum_t h_i^t}{N} \end{aligned}$$

where, for mathematical convenience, we introduce augmented vector/matrices

$$E[\tilde{\mathbf{z}} \mid \mathbf{x}^t, G_i] = \begin{bmatrix} E[\mathbf{z} \mid \mathbf{x}^t, G_i] \\ 1 \end{bmatrix}$$

and

$$E[\tilde{\mathbf{z}}\tilde{\mathbf{z}}^T \mid \mathbf{x}^s, G_i] = \begin{bmatrix} E[\mathbf{z}\mathbf{z}^T \mid \mathbf{x}^s, G_i] & E[\mathbf{z} \mid \mathbf{x}^s, G_i] \\ E[\mathbf{z} \mid \mathbf{x}^s, G_i]^T & 1 \end{bmatrix}$$

11.5 Spectral Clustering

Instead of clustering in the original space, we can also first map the data to a new space with reduced dimensionality such that similarities between data points are made more apparent. Then, we cluster in this new space, because similar points are expected to be placed nearby. From the work of Ng et al. [7], the algorithm for **spectral clustering** is trivial:

1. Form the affinity matrix $\mathbf{B} \in \mathbb{R}^{N \times N}$ defined by $\mathbf{B}_{ij} = \exp(-||\mathbf{x}^i - \mathbf{x}^j||^2/2\sigma^2)$ if $i \neq j$, and $\mathbf{B}_{ii} = 0$. This is the same as the Mahalanobis kernel except that we zero out the diagonal.

2. Define $\mathbf{D}$ to be a diagonal matrix whose (i,i)-element is the sum of the ith row of $\mathbf{B}$. Then construct the normalised symmetric matrix $\mathbf{L} = \mathbf{D}^{-1/2}\mathbf{B}\mathbf{D}^{-1/2}$.

3. Choose hyperparameter k as the reduced dimensionality of the *hidden space*, and find the k largest eigenvectors of $\mathbf{L}$ with respect to the k largest eigenvalues. Form the matrix $\mathbf{\Lambda} = \begin{bmatrix} \boldsymbol{\lambda}_1 & \boldsymbol{\lambda}_2 & \dots & \boldsymbol{\lambda}_k \end{bmatrix}$, which has size $N \times k$, by stacking the eigenvectors in the columns.

4. Re-normalise each row of $\mathbf{\Lambda}$ to have unit length (or sum to 1).

5. Finally, assign the original point $\mathbf{x}^i$ to cluster j if and only if row i of the matrix $\mathbf{\Lambda}$ was assigned to cluster j.

The value of σ is a hyperparameter of the model that can be optimised via cross-validation.

11.6 Hierarchical Clustering

Instead of a probabilistic approach, in hierarchical clustering, we aim to only use similarities between instances, without any other assumptions on the data, e.g., normally distributed data. We need a distance metric between instances, such as the Euclidean distance, or more generally, the **Minkowski distance**:

$$\mathcal{D}_m(\mathbf{x}^r, \mathbf{x}^s) = \left[\sum_{j=1}^{d} |x_j^r - x_j^s|^p \right]^{1/p}$$

This is the Euclidean distance when $p = 2$ and the Manhatten distance (city-block) when $p = 1$. An **agglomerative clustering** algorithm starts with N groups, each initially containing one training instance. Then, similar groups are merged to form larger groups, until there is a single group. In contrast, a **divisive clustering** algorithm starts with 1 group, and dividing it into smaller groups, until there are N groups.

At each iteration of an agglomerative algorithm, we choose the two closest groups to merge:

- In **single-link clustering**, the distance between clusters G_i and G_j is defined as the smallest distance between all possible pairs of elements of the two clusters:

$$\mathcal{D}(G_i, G_j) = \min_{\mathbf{x}^r \in G_i, \mathbf{x}^s \in G_j} \mathcal{D}(\mathbf{x}^r, \mathbf{x}^s)$$

- In **complete-link clustering**, the distance between clusters G_i and G_j is defined as the largest distance between all possible pairs of elements of the two clusters:

$$\mathcal{D}(G_i, G_j) = \max_{\mathbf{x}^r \in G_i, \mathbf{x}^s \in G_j} \mathcal{D}(\mathbf{x}^r, \mathbf{x}^s)$$

Besides these measures, there is the *average-link* method that uses the average of distances between all pairs of a cluster, and the *centroid* distance that measures the distance between the centroids (means) of the two clusters.

Since the time and space complexity of hierarchical clustering are $\mathcal{O}(n^3)$ and $\Omega(n^2)$, respectively, the method is too slow for even medium sized datasets. The alternative is then to use k-means or mixture models. Nonetheless, We can get an idea of the clusters by sampling small portions and using hierarchical clustering.

12 Ensemble Learning

In ensemble learning, we combine multiple base learners to improve accuracy. With cheaper computation and memory, such approaches have become popular. Two questions are relevant for this discussion:

- How do we generate base-learners that complement each other?
- How do we combine the outputs of base-learners for maximum accuracy?

12.1 Generating Diverse Learners

The aim is to find a set of **diverse learners** who differ in their decisions, but also have good accuracy. This way the different learners **complement** each other. Therefore, we need to maximise individual learner accuracies and also maximise the diversity between learners. There are different ways of achieving this:

- **Different Algorithms**: Using different algorithms to train the base learner, we enable the learners to make different assumptions about the data that lead to different classifiers. Deciding on a single algorithm has the consequence that we are putting emphasis on that single method and ignoring other methods.
- **Different Hyperparameters**: Using the same algorithm but with different hyperparameters, we can average over the hyperparameter factors and reduce variance, and therefore, reduce error.
- **Different Input Representations**: Using different representations of the same input as the input for the learners, we enable learners to make different characteristics explicit, which allows for better identification. Naively, in a multi-modal setting, we could concatenate all types of input (e.g., image, text) into one vector and use that for a single model. However, large input dimensionality make systems more complex and require larger samples for the estimators to be accurate. Instead, we treat each input representation as its own dataset, train separate models on each, and combine the models for better accuracy. Even if there is a single input representation, we can apply the **random subspace method** by choosing random subsets of the features, and training learners on these feature subsets, which has the effect that the learners will look at the same problem from different perspectives, and therefore, will be more robust.
- **Different Training Sets**: Alternatively, we can train different base-learners by different subsets of the training set. In **bagging**, we randomly draw training sets from the given sample. The learners can also be trained sequentially, so that training instances on which the base-learners are not accurate, are given more emphasis in the forthcoming base-learners. Examples include **boosting** and **cascading**, which actively try to generate *complementary* learners, instead of leaving this to chance. Another approach is to partition the training sample based on locality of the input space so that each base-learner is trained on instances in a certain local part of the input space. This is called **mixture of experts**.

- **Diversity vs. Accuracy**: We want multiple base-learners to be *reasonably* accurate and they are not required to be optimised separately. Base-learners are not chosen for their accuracy, but for their simplicity. Base-learners need to be diverse, i.e., accurate on different instances and specialising in subdomains of the problem. We care about the final combined accuracy.

In summary, we are trying to increase the chance that very different base-learners will make different types of errors.

12.2 Model Combination Schemes

There are different ways the base-learners are combined to generate the final output:

- **Multi-expert combination** methods have parallel working base-learners. These can be divided in two approaches:
  - In the **global approach**, also called *learner fusion*, given an input, all base-learners generate an output, and all these outputs are used, e.g., *voting* and *stacking*.
  - In the **local approach**, also called *learner selection*, there is a *gating model* that looks at the input and chooses one (or a few) base-learners that get the responsibility to generate the output, e.g., mixture of experts where the input space is divided to solve sub-problems.
- **Multi-stage combination** methods use a serial (sequential) approach where the next base-learner is trained (or tested) on only the instances where the previous base-learners are not accurate enough.

Given that we have L base-learners, we denote by $d_j(\mathbf{x}), j = 1, \ldots, L$ the prediction of base-learner $\mathcal{M}_j$ given arbitrary input data $\mathbf{x}$. In the case of different input representations, each $\mathcal{M}_j$ uses a different input representation $\mathbf{x}_j$. The final prediction is calculated from the predictions of the base-learners:

$$y = f(d_1, d_2, \ldots, d_L \,|\, \mathbf{\Phi})$$

where $f(\cdot)$ is the **combining function** and $\mathbf{\Phi}$ is the set of parameters. In the case of K outputs, we get for each learner $\mathcal{M}_j$, K outputs, $d_{ji}(\mathbf{x}), i = 1, \ldots, K, j = 1, \ldots, L$. Combining them we generate K values $y_i, i = 1, \ldots, K$. In classification, we could then choose the class with the maximum y_i value:

$$\text{Choose } C_i \text{ if } y_i = \max_{k=1}^{K} y_k$$

The simplest way to combine classifiers is by **voting**, which corresponds to taking a linear combination of the learners.

$$y_i = \sum_{j=1}^{L} w_j d_{ji} \qquad \text{where } w_j \geq 0, \sum_{j=1}^{L} w_j = 1$$

Table 6: Classification combination rules.

Rule	**Fusion function $f(\cdot)$**	**Description**
Sum	$y_i = \frac{1}{L}\sum_{j=1}^{L} d_{ji}$	Intuitive average over base-learners.
Weighted sum	$y_i = \sum_{j=1}^{L} w_j d_{ji}, w_j \geq 0, \sum_{j=1}^{L} w_j = 1$	Weighted sum where w_j represent prior probabilities over the base-learners.
Median	$y_i = \text{median}_j(d_{ji})$	More robust to outliers.
Minimum	$y_i = \min_j(d_{ji})$	Pessimistic.
Maximum	$y_i = \max_j(d_{ji})$	Optimistic.
Product	$y_i = \prod_j(d_{ji})$	Each learner has veto powers, regardless of the others, i.e., if one learner has output 0, the overall output goes to 0.

This is also known as **ensembles** and **linear opinion pools**. Table 6 gives an overview of common combination rules. Note that after the combination rule is applied, y_i may not sum to 1.

Voting schemes can also be seen as approximations under a Bayesian formulation, called *Bayesian model combination*. In classification, we have $w_j = P(\mathcal{M}_j)$ and $d_j = P(C_i \mid \mathbf{x}, \mathcal{M}_j)$, then the voting equation becomes:

$$P(C_i \mid \mathbf{x}) = \sum_{\mathcal{M}_j \in \boldsymbol{\mathcal{M}}} P(C_i \mid \mathbf{x}, \mathcal{M}_j) P(\mathcal{M}_j)$$

Simple voting uses a uniform prior. We can also compute the priors using the accuracy of the model on a separate validation set.

12.3 Bagging

In **bagging**, we use a voting method whereby base-learners are made different by training them over slightly different training sets. Using the bootstrap method, we generate L slightly different samples from a given sample $\mathcal{X}$ of size N. We draw N instances randomly from $\mathcal{X}$ **with replacement**. This leads to the possibility that some samples are drawn more than once and that certain instances are not drawn at all. The base-learners d_j are trained with these L samples $\mathcal{X}_j, j = 1, \ldots, L$.

A learning algorithm is an **unstable algorithm** if small changes in the training set

causes a large difference in the generated learner. In other words, the learning algorithm has high variance. **Bagging**, short for *bootstrap aggregating*, uses bootstrap to generate L training sets, trains L base-learners using an unstable learning procedure, and then during testing, takes an average. In the case of regression, using the median instead of the average is often more robust.

Averaging reduces variance. Therefore, it is required to use unstable learning algorithms in bagging, so that each base-learner has low bias but high variance, and thus results in a *diverse* base-learner. After averaging, the bias will remain the same, but the variance will decrease, which is a method to overcome the bias-variance trade-off. Examples of unstable models are decision trees and multi-layer perceptrons. If the original training set is large, we may want to generate smaller sets of size $N' < N$ from them using bootstrap, since otherwise, $\mathcal{X}_j$ will be too similar, and d_j will be highly correlated.

12.4 Boosting

In bagging, it is clear that generating complementary base-learners is left to chance and to the choice of unstable learning method. In **boosting**, we actively try to generate complementary base-learners by training the *next learner* on the mistakes of the previous learners. The original boosting algorithm combines three *weak learners* to generate a *strong learner*. A **weak learner** has error probability less than 0.5, which makes it at least better than random guesses in a two-class problem. A **strong learner** has arbitrarily small error probability.

12.4.1 Original Boosting algorithm

Given a large training set, we randomly divide it into three. We use $\mathcal{X}_1$ to train d_1. Then, we feed $\mathcal{X}_2$ into d_1, and predict the outputs. All instances misclassified by d_1 on $\mathcal{X}_2$ are taken as the training set for d_2. After training d_2 on the misclassified subset of $\mathcal{X}_2$, we take $\mathcal{X}_3$, and feed it to both d_1 and d_2. Then, the instances on which d_1 and d_2 disagree form the training set for d_3.

During testing, given an instance, we give it to d_1 and d_2, and if they agree, then that is the response output. Otherwise, we feed the disagreed upon instance to d_3, and its output is taken as the response.

It was shown that such boosting systems reduces error, and the error can be reduced by such systems recursively. That is, instead of 3 base-learners, we use L learners in some sequential configuration.

Although it is quite successful, the original boosting method requires a very large training sample. The second and third classifiers are only trained on subsets of the original training set on which the first misclassified.

12.4.2 AdaBoost

There is also a variant, AdaBoost, short for *adaptive boosting*, that uses the same training set over and over, and therefore, does not need to be large. The requirement, however, is that the classifiers should be simple so that they do not overfit. AdaBoost can combine an arbitrary amount of base-learners.

In the original algorithm, the idea is to modify the probabilities of drawing the instances as a function of the error. Let p_j^t denote the probability that the instance pair $(\mathbf{x}^t, r^t)$ is drawn to train the jth base-learner. Initially, we set all $p_1^t = 1/N$. Then we add new base-learners as follows, starting from $j = 1$. Let ϵ_j denote the error rate of d_j. AdaBoost requires that learners are weak, i.e., $\epsilon_j < 0.5, \forall j$. If this does not hold, then we stop adding new base-learners. This error rate refers to the error on the resampled dataset used at step j. We define

$$\beta_j = \frac{\epsilon_j}{1 - \epsilon_j} < 1$$

and we update p_{j+1}^t as follows:

$$p_{j+1}^t = \begin{cases} \beta_j p_j^t & \text{if } d_j \text{ correctly classifies } \mathbf{x}^t \\ p_j^t & \text{otherwise} \end{cases}$$

Because p_{j+1}^t represent probabilities, we normalise by dividing by the sum $\sum_t p_{j+1}^t$. This has the effect that the probability of a correctly classified instance decreases, while the probability of a misclassified instances increases. Then a new sample of the same size is drawn from the original training set according to the modified probabilities, p_{j+1}^t, *with replacement*, and this is used to train d_{j+1}.

The effect of d_{j+1} is that it will focus on instances misclassified by d_j. That is why AdaBoost requires base-learners that are simple and not accurate, otherwise the next training sample will only contain outliers and noise repeated many times.

Once training is done, AdaBoost uses a weighted sum voting method, where, given an instance, all d_j decide with weights proportional to the base-learner's accuracy values:

$$w_j = \log(1/\beta_j)$$

The success of AdaBoost is due to its property of increasing the **margin**. That is, if the margin increases, the training instances are better separated and error is less likely. AdaBoost is therefore similar to *support vector machines*.

12.5 Stacked Generalisation

Stacked generalisation is a technique that extends voting because the way the outputs of the base-learners is combined is not required to be linear, but rather is learned through a combiner system $f(\cdot \mid \mathbf{\Phi})$, which is another learner, whose parameters $\mathbf{\Phi}$ are also trained.

$$y = f(d_1, d_2, \ldots, d_L \mid \mathbf{\Phi})$$

The combiner learns what the correct output is when the base-learners give a certain output combination. However, we cannot train the combiner function on the training data because the base-learners may be memorising the training set. The combiner should instead learn how the base-learners make errors. **Stacking** is a means of estimating and correcting for the biases of the base-learners.

The combiner is not required to be linear, as with the simple voting combination rule. Rather, it can be any non-linear function, e.g., a MLP with $\mathbf{\Phi}$ as the connection weights.

The outputs of the base-learners d_j define a new L-dimensional space in which the output discriminant/regression function is learned by the combiner function.

In stacked generalisation, we want the base-learners to be as different as possible so that they complement each other. It is therefore best to use different learning algorithms.

13 Reinforcement Learning

In the case of learning to play a game, a supervised approach is not usable, because:

1. It is very costly to have a teacher that will take the learner through many games, and indicate which move is the best in every position, e.g., chess.
2. There is no such thing as the best move, that is, the goodness of a move depends on the moves that follow, i.e., a sequence of moves is good if afterwards, we win the game.

The learner is an **agent** placed in an **environment**. At any time, the environment is in a certain **state**, one from a set of possible states. The *decision maker* has a set of possible **actions**, and once an action is chosen and taken by the decision maker, the state changes. We get feedback in the form of **rewards** for certain actions. The learning agent learns, based on the reward function, the best sequence of actions to solve a problem, where "best" refers to the *maximum cumulative reward*. This is the setting for **reinforcement learning**. This approach to learning differs in that the learner learns by a **critic**, as opposed to a *teacher* in supervised learning. A critic does not tell the learner what to do, but only tells the learner how well it has been doing in the past.

13.1 Elements of Reinforcement Learning

The agent interacts with the environment by using its **sensors** to decide on its *current state*, and takes an *action* that modifies its state. When the agent takes an action, the environment provides a **reward**.

- Time steps are discrete, $t = 0, 1, 2, \ldots,$
- $s_t \in S$ is the state of the agent at time step t, where S is the set of all possible states.
- $a_t \in \mathcal{A}(s_t)$ denotes the action that the agent takes at time step t, where $\mathcal{A}(s_t)$ is the set of possible actions in state s_t.
- An agent in state s_t that takes the action a_t causes the clock to tick, and the reward $r_{t+1} \in \mathbb{R}$ is received, which has the consequence that the agent moves to the next state, s_{t+1}.

We can model the problem as a **Markov decision process** (MDP).

- The reward is sampled from $P(r_{t+1} \mid s_t, a_t)$ and the next state is sampled from $P(s_{t+1} \mid s_t, a_t)$. In this Markov system, the state and reward in the next time step **only** depend on the current state and action.
- The sequence of actions from the start to the terminal state is an **episode**, or **trial**.
- The **policy**, π, defines the agent's behaviour and is a mapping from the states in the environment to the actions, $\pi : S \mapsto \mathcal{A}$. The policy defines the action to be taken in any state $s_t : a_t = \pi(s_t)$.
- The **value** of a policy π, denoted by $V^\pi(s_t)$, is the expected cumulative reward that will be received while the agent follows the policy, starting from its current state s_t.

In the **finite-horizon** or *episodic* model, the agent tries to maximise the expected reward for the next T time steps:

$$V^{\pi}(s_t) = E[r_{t+1} + r_{t+2} + \cdots + r_{t+T}] = E\left[\sum_{i=1}^{T} r_{t+i}\right]$$

Certain tasks have no fixed end point, that is, there is no prior fixed limit to the episode. In the **infinite-horizon** model, there is no sequence limit, but we deal with this by introducing a **discount**, γ:

$$V^{\pi}(s_t) = E[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots] = E\left[\sum_{i=1}^{\infty} \gamma^{i-1} r_{t+i}\right]$$

where $0 \leq \gamma < 1$ is the *discount rate* so that the return is finite. If $\gamma = 0$, only the immediate reward counts. As γ approaches 1, rewards further in the future count more, and we say that the agent becomes more *farsighted*. We prefer rewards sooner rather than later, and thus set $\gamma < 1$, because it is not certain how long we will survive (e.g., in the game).

For each policy π, there is an associated *value* $V^{\pi}(s_t)$, and we want to find the optimal policy π^* such that

$$V^*(s_t) = \max_{\pi} V^{\pi}(s_t), \forall s_t$$

In other words, for all possible states, s_t, we maximise the value over all possible policies π.

In some applications, instead of working with the values of states $V(s_t)$, we prefer to work with the values of state-action pairs, $Q(s_t, a_t)$.

- $V(s_t)$ denotes how good it is for the agent to be in state s_t.
- $Q(s_t, a_t)$ denotes how good it is to perform action a_t when in state s_t.

We can then define $Q^*(s_t, a_t)$ as the value, that is, the expected cumulative reward, of action a_t taken in state s_t and then obeying the optimal policy afterwards. The value of a state is equal to the value of the best possible action. This gives:

$$\begin{aligned}
V^*(s_t) &= \max_{a_t} Q^*(s_t, a_t) \\
&= \max_{a_t} E\left[\sum_{i=1}^{\infty} \gamma^{i-1} r_{t+i}\right] \\
&= \max_{a_t} E\left[\gamma^{1-1} r_{t+1} + \sum_{i=2}^{\infty} \gamma^{i-1} r_{t+i}\right] \\
&= \max_{a_t} E\left[r_{t+1} + (\gamma^{2-1} r_{t+2} + \gamma^{3-1} r_{t+3} + \ldots)\right] \\
&= \max_{a_t} E\left[r_{t+1} + \gamma(\gamma^{1-1} r_{t+2} + \gamma^{2-1} r_{t+3} + \ldots)\right] \\
&= \max_{a_t} E\left[r_{t+1} + \gamma \sum_{i=1}^{\infty} \gamma^{i-1} r_{t+i}\right] \\
&= \max_{a_t} E\left[r_{t+1} + \gamma V^*(s_{t+1})\right] \\
&= \max_{a_t} \left(E[r_{t+1}] + \gamma E[V^*(s_{t+1})]\right)
\end{aligned}$$

$$V^*(s_t) = \max_{a_t} \left(E[r_{t+1}] + \gamma \sum_{s_{t+1}} P(s_{t+1} \mid s_t, a_t) V^*(s_{t+1}) \right)$$

Note that the expected value of the optimal value of a policy in state s_t, $E[V^*(s_{t+1})]$, is the optimal value in state s_t weighted against the probabilities of moving to the next state s_{t+1}, and continuing from there using the optimal policy. We sum over all possible next states, and discount it (γ) because it is one step later. Adding the immediate expected reward, $E[r_{t+1}]$, to the expected cumulative reward, $E[V^*(s_{t+1})]$, we get the total expected cumulative reward for action a_t. We then choose the best of possible actions. Similarly, we can also write this in terms of the optimal value of state-action pairs, $Q^*(s_t, a_t)$, by replacing $V^*(s_t)$:

$$Q^*(s_t, a_t) = E[r_{t+1}] + \gamma \sum_{s_{t+1}} P(s_{t+1} \mid s_t, a_t) \max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1})$$

Once we have values for $Q^*(s_t, a_t)$, we can define the policy π as taking the action a_t^*, which has the highest value among all $Q^*(s_t, a_t)$:

$$\pi^*(s_t): \text{ Choose } a_t^* \text{ where } Q^*(s_t, a_t^*) = \max_{a_t} Q^*(s_t, a_t)$$

In other words, if we have the values for $Q^*(s_t, a_t)$, we can use a greedy search at each local step to get the optimal sequence of steps that maximises the *cumulative reward*.

13.2 Model-Based Learning

In the model-based learning setting, we completely know the environment model parameters, $P(r_{t+1} \mid s_t, a_t)$ and $P(s_{t+1} \mid s_t, a_t)$. In other words, we know the distribution from which the environment will reward us and what the next state is, given we are in the current state and take a certain action. We do not need any exploration, since we already know the environment parameters, and can directly solve for the optimal value function and policy using *dynamic programming*.

- In **value iteration**, we use an iterative procedure to find the optimal value function. This has been shown to converge to $V^*(s_t)$.
- In **policy iteration**, we store and update the policy rather than doing this indirectly over the values. The idea is to start with a policy $\pi(s)$ and improve it repeatedly until there is no change.

13.3 Temporal Difference learning

Given that we know the environment parameters, i.e., the reward and next state probability distributions, we can solve for the optimal policy using dynamic programming. However, these methods are costly, and we seldom have such perfect knowledge about the environment. It is more interesting and realistic to use reinforcement learning in cases where we do not have the environment parameters, i.e., the model.

This requires exploration of the environment to query the model. We do not assume full knowledge of the environment, but we do require that the environment is stationary. When we explore and get to see the value of the next state and reward, we use this information

to update the value of the current state. These algorithms are called **temporal difference** algorithms because we look at the difference between our current estimate of the value of a state (or state-action pair) and the discounted value of the next state and the reward received.

13.3.1 Exploration strategies

To explore the environment, one possibility is to use ϵ-*greedy* search where:

- with probability ϵ, we choose one action uniformly randomly among all possible actions, which corresponds to **exploring**.
- with probability $1 - \epsilon$, we choose the best action, which corresponds to **exploiting**.

Once we have explored enough, we start exploiting, and thus, we start with a high ϵ value, and decrease it gradually.

13.3.2 Nondeterministic Rewards and Actions

If the rewards and the result of actions are not deterministic, then we use a probability distribution for the reward $P(r_{t+1} \mid s_t, a_t)$ from which rewards are sampled, and we use a probability distribution for the next state $P(s_{t+1} \mid s_t, a_t)$. These help us model the uncertainty in the system that may be due to forces we cannot control in the environment, e.g., the opponent in chess.

In such a case, we have:

$$Q(s_t, a_t) = E[r_{t+1}] + \gamma \sum_{s_{t+1}} P(s_{t+1} \mid s_t, a_t) \max_{a_{t+1}} Q(s_{t+1}, a_{t+1})$$

Due to the nondeterministic nature of the environment, we cannot do a direct assignment. For the same state and action, we may receive different rewards or move to different states. What we can do is keep a **running average**. This is known as the ***Q*-learning algorithm**:

$$\hat{Q}(s_t, a_t) \leftarrow \hat{Q}(s_t, a_t) + \eta(r_{t+1} + \gamma \max_{a_{t+1}} \hat{Q}(s_{t+1}, a_{t+1}) - Q(s_t, a_t))$$

where the hat denotes that this is an estimate. We can think of $r_{t+1} + \gamma \max_{a_{t+1}} \hat{Q}(s_{t+1}, a_{t+1})$ as a sample of instances for each (s_t, a_t) pair and we would like $\hat{Q}(s_t, a_t)$ to converge to its mean. As usual, η, is gradually decreased for convergence, and it has been shown that this algorithm converges to the optimal Q^* values.

We can also think of the Q-learning update equation as reducing the difference between the current Q value and the backed-up estimate, from one time step later. Furthermore, this is an **off-policy** method as the value for the best next action is used *without* using the policy π. In an **on-policy** method, the policy is used to determine the next action. The on-policy version of Q-learning is the **Sarsa** algorithm, where the update rule is defined as:

$$\hat{Q}(s_t, a_t) \leftarrow \hat{Q}(s_t, a_t) + \eta(r_{t+1} + \gamma \hat{Q}(s_{t+1}, a'_{t+1}) - Q(s_t, a_t))$$

where a'_{t+1} is chosen based on the policy $\pi(s_{t+1})$ derived from Q.

13.3.3 Eligibility Traces

An **eligibility trace** is a record of the occurrence of past visits that enable us to implement *temporal credit assignment*. This means that we can also update the values of previously occurred visits as well.

To store the eligibility trace, we require an additional memory variable associated with each state-action pair, $e(s, a)$, initialised to 0. When the state-action pair is visited, that is, when we take action a in state s, its eligibility is set to 1. The eligibility of all other state-action pairs are multiplied by $\gamma\lambda$, where $0 \leq \lambda \leq 1$ is the trace decay parameter.

$$e_t(s, a) = \begin{cases} 1 & \text{if } s = s_t \text{ and } a = a_t \\ \gamma\lambda e_{t-1}(s, a) & \text{otherwise} \end{cases}$$

If a state-action pair has never been visited, its eligibility remains 0. If it has been visited, as time passes and other state-actions are visited, its eligibility decays depending on the value of γ and λ. In the *Sarsa* algorithm, the temporal error at time t is

$$\delta_t = r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)$$

If we introduce eligibility traces, this becomes the *Sarsa*(λ) update, and all state-action pairs are updated using:

$$Q(s, a) \leftarrow Q(s, a) + \eta\delta_t e_t(s, a), \forall s, a$$

This updates all eligible state-action pairs, where the update depends on how far they have occurred in the past. The value for λ defines the **temporal credit**. If $\lambda = 0$, only a one-step update is done. This is the case for Q-learning and *Sarsa* prior to introducing eligibility traces, which we denote as $Q(0)$ and *Sarsa*(0). As λ gets closer to 1, more of the previous steps are considered. When $\lambda = 1$, all previous steps are updated and the credit given to them falls only by the γ factor per step.

13.4 Generalisation

Until now, we assumed that the $Q(s, a)$ or $V(s)$ values are stored in a lookup table, thus making the algorithms **tabular algorithms**. There are several issues with this approach:

1. When the number of states and the number of actions is large, the size of the table may become quite large.
2. States and actions may be continuous rather than discrete, e.g., turning the steering wheel by a certain angle. Discretised states/values may cause error.
3. When the search space is large, too many episodes may be needed to fill in all the entries of the table with acceptable accuracy.

Instead of storing the Q values as they are, we can consider this a **regression problem**. In this supervised learning problem, we define the regressor $Q(s, a \,|\, \theta)$, where we take s and a as inputs, parameterised by θ, and try to predict the corresponding Q values. This can be a neural network with s and a as inputs and θ as the connection weights.

If we know that similar (s, a) pairs have similar Q values, we can generalise from past cases and come up with good $Q(s, a)$ values, even if that state-action pair has never been seen before. As for the training set, we define:

- Input (s_t, a_t) pairs.
- Predicted output: $Q(s_t, a_t)$.
- Ground truth: $r_{t+1} + \gamma Q(s_{t+1}, a_{t+1})$.
- Squared difference error: $E^t(\theta) = [r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]^2$

If we are using gradient descent on a neural network, the weights are updated as:

$$\Delta\theta = \eta[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1} \,|\, \theta) - Q(s_t, a_t \,|\, \theta)] \frac{\partial}{\partial\theta} Q(s_t, a_t \,|\, \theta)$$

for any θ in the network using backpropagation, which has the same form as the standard delta update rule.

In theory, any regression method can be used to train the Q function, but the task has the following requirements:

- Allow for generalisations, that is, we really need to guarantee that similar states and similar actions have similar Q values. This also requires that the representation of the inputs (s, a) is good, so that similarities are more apparent.
- Using local learners, that is, reinforcement learning updates provide instances one by one and not as the whole training set. The learning algorithm should therefore be able to do individual updates to learn a new instance without forgetting what has already been learned. In local learners, only a local part of the learner is updated without possibly corrupting the information in another part.

In the case of a multilayer perceptron, online updates as the episode continues is not a good idea as these training examples are highly correlated. We instead need to store them in memory and sample randomly from there. This is called **experience replay**, and is one of the tricks used to train deep networks.

References

[1] E. Alpaydin, *Introduction to Machine Learning*, 4th ed., ser. Adaptive Computation and Machine Learning. Cambridge, MA: MIT Press, 2014.

[2] C. Maathuis, G. M. de Buy Wenniger, S. Bromuri, F. Hermens, and L. Curier, *Machine Learning*, 2nd ed. Open Universiteit, 2023.

[3] F. Doshi-Velez and B. Kim, "Towards a rigorous science of interpretable machine learning," 2017.

[4] K. B. Petersen and M. S. Pedersen, "The matrix cookbook," nov 2012, version 20121115. [Online]. Available: http://www2.compute.dtu.dk/pubdb/pubs/3274-full.html

[5] T. Mitchell, *Machine Learning*. McGraw-Hill Education, 1997.

[6] Z. Ghahramani and G. E. Hinton, "The em algorithm for mixtures of factor analyzers," 1996. [Online]. Available: https://api.semanticscholar.org/CorpusID:18270595

[7] A. Ng, M. Jordan, and Y. Weiss, "On spectral clustering: Analysis and an algorithm," in *Advances in Neural Information Processing Systems*, T. Dietterich, S. Becker, and Z. Ghahramani, Eds., vol. 14. MIT Press, 2001. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2001/file/801272ee79cfde7fa5960571fee36b9b-Paper.pdf