MATH30028: Statistical Machine Learning

J. Chakravarty, I. Gallagher, S. Power, L. Sharrock, N. Sonenberg[1]

January 2023

[1]Contributions to the notes have also been made by H. Alawiye, S. Tickle, and A. Wang

Contents

0.1 Prerequisites

Prerequisites: Lagrange multipliers. Basic optimization schemes (e.g. gradient descent). Norm notation (L1, L2, L infinity). Basic vector/matrix calculus. Rank of a matrix (column rank). Definiteness of matrices. Orthogonal matrices.

0.2 Resources

Hastie et al. [2008], Shah [2020].

0.3 Notation

- Set of reals is $\mathbb{R}$, expectation is $\mathbb{E}$.
- Matrix transpose is $\mathbf{X}^\top$. Identity matrix is $\mathbf{I}$ or $\mathbf{I}_N$ to specify $N \times N$.
- $[N]$ is the discrete set $\{1, \ldots, N\}$.
- Integrals are written as $\int f(x)\,\mathrm{d}x$ (using latex dif).
- For fixed data (i.e. a training set), the training pairs are $(x_1, y_1), \ldots, (x_N, y_N)$, where each $x_i = (x_{i1}, \ldots, x_{ip})^\top \in \mathbb{R}^p$ and $y_i \in \mathbb{R}$ for $i \in [N]$.
- The $N \times p$ design matrix is $\mathbf{X}$ and combined label vector $\mathbf{y} \in \mathbb{R}^N$:

$$\mathbf{X} = \begin{pmatrix} -- & x_1^\top & -- \\ -- & x_2^\top & -- \\ & \vdots & \\ -- & x_N^\top & -- \end{pmatrix}.$$

  An individual column of the design matrix $\mathbf{X}$ is $\mathbf{x}_j$, for $j \in [p]$.
- When referring to a generic random data point, write this as (X, Y), a pair of random variables taking values in $\mathbb{R}^p, \mathbb{R}$ respectively. In other words, the (x_i, y_i) are iid realisations of (X, Y). A generic value of X is written as x, so $\mathbb{E}[Y|X = x] = f(x)$.
- The regression function is f, so generally we have $Y = f(X) + \epsilon$.

Chapter 1

Regression

1.1 Linear Regression

1.1.1 Introduction

Suppose we have some data of the form $(x_1, y_1), \ldots, (x_N, y_N)$, where each $x_i = (x_{i1}, \ldots, x_{ip})^\top \in \mathbb{R}^p$, and each $y_i \in \mathbb{R}$, for $i = 1, \ldots, N$. The x_i are the *covariates*, also known as *predictors*, *regressors*, or *explanatory variables*, which we think of as the vectors of inputs. The y_i are the *response variables*, the real-valued outputs we would like to predict.

In general, we will view (x_i, y_i) as independent, identically distributed (i.i.d.) realisations of some random variables (X, Y), which take values in $\mathbb{R}^p \times \mathbb{R}$. The goal of *regression* is to find a function $f : \mathbb{R}^p \to \mathbb{R}$ to model the relationship

$$Y = f(X) + \varepsilon,$$

for some noise variable $\varepsilon = (\varepsilon_1, \ldots, \varepsilon_N)^\top$. The noise variable ε is independent of X, and assumed to satisfy $\mathbb{E}[\varepsilon] = 0$. We thus have $\mathbb{E}[Y|X = x] = f(x)$. In other words, the function f models the expectation of Y given the covariates X.

In general, the function f could be very complicated. We will begin, however, by recapping the simplest case, in which f is assumed to be a linear function. In particular, writing $X = (X_1, \ldots, X_p)$,

$$f(X) = \sum_{j=1}^{p} X_j \beta_j \tag{1.1}$$

where $\beta_1, \ldots, \beta_p \in \mathbb{R}$ are a set of real parameters. This means, in particular, that each realisation of the response y_i obeys a linear relationship with the covariates $x_{i1}, \ldots, x_{ip}$ of the form

$$y_i = x_i^\top \beta + \varepsilon_i = \sum_{j=1}^{p} x_{ij} \beta_j + \varepsilon_i,$$

More succinctly, in vector-matrix notation, we can rewrite this equation as

$$\mathbf{y} = \mathbf{X}\beta + \varepsilon,$$

where $\mathbf{y} = (y_1, \ldots, y_N)^\top \in \mathbb{R}^N$ is the vector of responses, $\beta = (\beta_1, \ldots, \beta_p)^\top \in \mathbb{R}^p$ is the vector of parameters, $\varepsilon = (\varepsilon_1, \ldots, \varepsilon_N)^\top \in \mathbb{R}^N$ is the vector of zero mean i.i.d. noise variables, and $\mathbf{X}$ is the $N \times p$ matrix of covariates

$$\mathbf{X} = \begin{pmatrix} x_1^\top \\ x_2^\top \\ \vdots \\ x_N^\top \end{pmatrix} = \begin{pmatrix} x_{11} & \ldots & x_{1p} \\ x_{21} & \ldots & x_{2p} \\ \vdots & \vdots & \vdots \\ x_{N1} & \ldots & x_{Np} \end{pmatrix},$$

whose i^{th} row $x_i^\top$ contains the i^{th} observations of all the covariates. This is often referred to as the *design matrix*.

We note that we have not explicitly included an intercept term in this model. This can be easily by remedied by including $X_0 := 1$ as an additional covariate, so that now

$$f(X) = \beta_0 + \sum_{i=1}^{p} X_j \beta_j. \tag{1.2}$$

Similar to before, in vector-matrix notation, our linear model can now be written as

$$\mathbf{y} = \mathbf{X}\beta + \varepsilon,$$

where $\mathbf{y} = (y_1, \ldots, y_N)^\top \in \mathbb{R}^N$ is the vector of responses, $\beta = (\beta_0, \beta_1, \ldots, \beta_p)^\top \in \mathbb{R}^{p+1}$ is the vector of parameters, now with β_0 denoting the intercept, $\varepsilon = (\varepsilon_1, \ldots, \varepsilon_N)^\top \in \mathbb{R}^N$ is the vector of zero mean i.i.d. noise variables, and $\mathbf{X}$ is the $N \times (p+1)$ matrix of covariates

$$\mathbf{X} = \begin{pmatrix} 1 & x_{11} & \ldots & x_{1p} \\ 1 & x_{21} & \ldots & x_{2p} \\ \vdots & \vdots & \vdots & \vdots \\ 1 & x_{N1} & \ldots & x_{Np} \end{pmatrix},$$

Technically speaking, we now have $\beta \in \mathbb{R}^{p+1}$ and $\mathbf{X} \in \mathbb{R}^{N \times (p+1)}$, rather than $\beta \in \mathbb{R}^p$ and $\mathbf{X} \in \mathbb{R}^{N \times p}$ as before. For convenience, in the subsequent sections, we will usually just use the notation $\beta \in \mathbb{R}^p$ and $\mathbf{X} \in \mathbb{R}^{N \times p}$. We do this with the understanding that, in most cases, we could substitute $\beta \in \mathbb{R}^{p+1}$ and $\mathbf{X} \in \mathbb{R}^{N \times (p+1)}$ if we wanted to make explicit the inclusion of an intercept.

1.1.2 Least Squares Estimation

Our statistical problem can be stated as follows. Given the training data $(x_1, y_1), \ldots, (x_N, y_N)$, we would like to infer the parameters β. We can estimate these parameters by minimising an appropriate *objective function* or *loss*

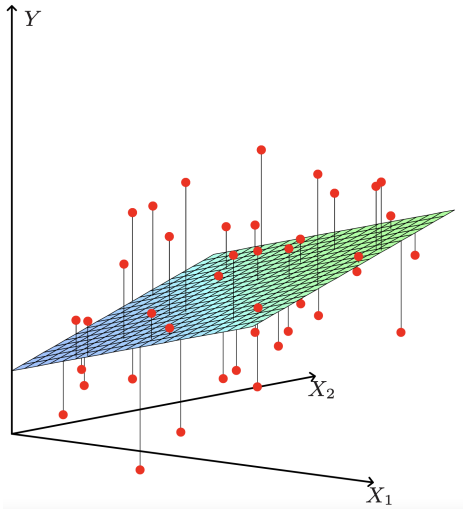


Figure 1.1: **Fitting the linear least squares model for $X = (X_1, X_2) \in \mathbb{R}^2$.** We look for the linear function of X (in this case, a plane) which minimises the sum of squared residuals from the observations Y (red). This figure is from Hastie et al. [2008].

function. A natural choice for the objective function is the *least squares* objective. In this case, we choose the parameters $\beta = (\beta_1, \ldots, \beta_p)^\top$ which minimise the residual sum of squares (RSS),

$$\begin{aligned} \mathrm{RSS}(\beta) &:= \sum_{i=1}^{N} \big(y_i - f(x_i)\big)^2 \\ &= \sum_{i=1}^{N} \Big(y_i - x_i^\top \beta\Big)^2 \\ &= \|\mathbf{y} - \mathbf{X}\beta\|_2^2. \end{aligned} \tag{1.3}$$

The geometry of least-squares fitting is shown in Figure 1.1. We assume for now that the matrix $\mathbf{X}$ has full column rank. In particular, this means that $p \leq N$.

Proposition 1. *Assume $\mathbf{X}$ has full column rank. The unique minimizer of the RSS objective is given by the ordinary least squares (OLS) estimator,*

$$\hat{\beta}^{\mathrm{OLS}} = \operatorname*{arg\,min}_{\beta \in \mathbb{R}^p} \mathrm{RSS}(\beta) = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}. \tag{1.4}$$

Proof. We begin by expanding the RSS in vector-matrix notation,

$$\begin{aligned} \mathrm{RSS}(\beta) &= (\mathbf{y} - \mathbf{X}\beta)^\top (\mathbf{y} - \mathbf{X}\beta) \\ &= \mathbf{y}^\top \mathbf{y} - (\mathbf{X}\beta)^\top \mathbf{y} - \mathbf{y}^\top (\mathbf{X}\beta) + (\mathbf{X}\beta)^\top (\mathbf{X}\beta) \\ &= \mathbf{y}^\top \mathbf{y} - 2\beta^\top \mathbf{X}^\top \mathbf{y} + \beta^\top \mathbf{X}^\top \mathbf{X}\beta. \end{aligned}$$

We now differentiate with respect to β, which yields[1]

$$\frac{\partial}{\partial \beta}\mathrm{RSS}(\beta) = -2\mathbf{X}^\top \mathbf{y} + 2\beta \mathbf{X}^\top \mathbf{X}.$$

Setting expression equal to zero, we have that

$$\frac{\partial}{\partial \beta}\mathrm{RSS}(\beta)\bigg|_{\beta=\hat{\beta}^{\mathrm{OLS}}} = -2\mathbf{X}^\top \mathbf{y} + 2\hat{\beta}^{\mathrm{OLS}} \mathbf{X}^\top \mathbf{X} = 0.$$

Rearranging, we obtain the so-called *normal equations*,

$$\hat{\beta}^{\mathrm{OLS}} \mathbf{X}^\top \mathbf{X} = \mathbf{X}^\top \mathbf{y}$$

Finally, using the assumption that $\mathbf{X}$ has full column rank, we can invert $\mathbf{X}^\top \mathbf{X}$ and thus obtain

$$\hat{\beta}^{\mathrm{OLS}} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}.$$

We can check that this is indeed a minimiser by differentiating a second time. This yields

$$\frac{\partial}{\partial \beta^2}\mathrm{RSS}(\beta) = 2\mathbf{X}^\top \mathbf{X}.$$

Under the assumption that $\mathbf{X}$ has full column rank, the matrix $\mathbf{X}^\top \mathbf{X}$ is positive definite, and thus we do indeed have a minimum. □

Based on this estimator, we can now compute the *fitted values* at the training inputs as

$$\hat{\mathbf{y}} = \mathbf{X}\hat{\beta}^{\mathrm{OLS}} = \mathbf{X}(\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}.$$

The matrix $\mathbf{H} = \mathbf{X}(\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top$ appearing in this equation is often referred to as the *hat matrix*, as it puts the hat on $\mathbf{y}$. This matrix is a projection matrix onto the space V spanned by the columns of X. We provide a graphical representation of this in Figure 1.2.

The residuals $\mathbf{r}$, defined as the difference between the observed and the fitted values, can also be obtained using the hat matrix, as

$$\mathbf{r} = \mathbf{y} - \hat{\mathbf{y}} = (\mathbf{I} - \mathbf{X}(\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top)\mathbf{y}.$$

Under the assumptions that $\mathbb{E}[\varepsilon_i] = 0$, and $\mathrm{Var}(\varepsilon_i) = \sigma^2$ for $i = 1, \ldots, N$, it is straightforward to compute the mean and variance of the OLS estimator. In particular, we have

$$\mathbb{E}[\hat{\beta}^{\mathrm{OLS}}] = \beta$$
$$\mathrm{Var}(\hat{\beta}^{\mathrm{OLS}}) = (\mathbf{X}^\top \mathbf{X})^{-1} \sigma^2.$$

[1] We use the several identities without proof. Let $\mathbf{A} \in \mathbb{R}^{p \times p}$ be a symmetric matrix, $\mathbf{b} \in \mathbb{R}^p$, and $\mathbf{c} \in \mathbb{R}^p$. Then we have

$$\frac{\partial}{\partial \mathbf{b}} \mathbf{b}^\top \mathbf{c} = \mathbf{c} \quad , \quad \frac{\partial}{\partial \mathbf{b}} \mathbf{b}^\top \mathbf{A} \mathbf{b} = 2\mathbf{b}\mathbf{A}$$

In particular, we will apply these identities with $\mathbf{A} = \mathbf{X}^\top \mathbf{X}$, $\mathbf{b} = \beta$, and $\mathbf{c} = \mathbf{X}^\top \mathbf{y}$.

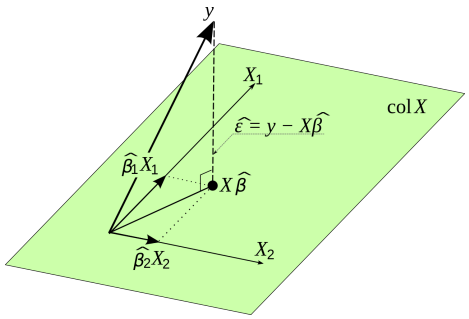


Figure 1.2: **The geometry of least squares regression.** The response vector $\mathbf{y}$ is projected orthogonally onto the plane spanned by the columns of the design matrix. The projection $\hat{\mathbf{y}}$ represents the vector of fitted values.

It follows, in particular, that $\hat{\beta}^{\text{OLS}}$ is an unbiased estimate of β. Typically, one estimates the variance σ^2 using the residuals:

$$\hat{\sigma}^2 = \frac{1}{N-p}\sum_{i=1}^{N} r_i^2.$$

Here, the factor $1/(N-p)$ rather than $1/N$ ensures that $\hat{\sigma}^2$ is also an unbiased estimator of σ^2. Under the additional assumption that that the errors are normally distributed, that is, $\varepsilon_i \sim N(0, \sigma^2)$, we then have

$$\hat{\beta}^{\text{OLS}} \sim N\left(\beta, (\mathbf{X}^\top \mathbf{X})^{-1}\sigma^2\right).$$

In this case, it also turns out that the OLS estimate $\hat{\beta}^{\text{OLS}}$ coincides with the maximum likelihood estimate $\hat{\beta}^{\text{MLE}}$, defined as the maximiser of the likelihood function

$$\begin{aligned}
\hat{\beta}^{\text{MLE}} &= \arg\max_{\beta} \mathcal{L}(\beta, \sigma^2) \\
&= \arg\max_{\beta} \prod_{i=1}^{N} \mathcal{N}(y_i | x_i, \beta, \sigma^2) \\
&= \arg\max_{\beta} \frac{1}{(2\pi\sigma^2)^{\frac{N}{2}}} \exp\left[-\frac{1}{2\sigma^2}\sum_{i=1}^{N}(y_i - x_i^\top \beta)^2\right].
\end{aligned}$$

By maximising the likelihood function with respect to σ^2, one can also show that

$$\hat{\sigma}^2_{\text{MLE}} = \frac{\text{RSS}(\hat{\beta}^{\text{OLS}})}{N}.$$

1.1.3 The Gauss-Markov Theorem

Even without the assumption of normality, it is possible to show that the OLS estimator $\hat{\beta}^{\text{OLS}}$ has the smallest variance of all linear unbiased estimates. This is the subject of the famous Gauss-Markov Theorem.

Theorem 2 (Gauss-Markov Theorem)**.** *Assume that the errors* $\varepsilon = (\varepsilon_1, \ldots, \varepsilon_N)$ *satisfy* $\mathbb{E}[\varepsilon_i] = 0$ *and* $\text{Var}(\varepsilon_i) = \sigma^2$ *for all* $i = 1, \ldots, N$*. Let* $\tilde{\beta}$ *be any unbiased linear estimator of* β*. Then the matrix*

$$\text{Var}(\tilde{\beta}) - \text{Var}(\hat{\beta}^{\text{OLS}})$$

is positive semi-definite. That is, $\hat{\beta}^{\text{OLS}}$ *is the best unbiased linear estimator (BLUE) of* β*.*

Proof. Since $\tilde{\beta}$ is a linear estimator, we can write $\tilde{\beta} = \mathbf{C}\mathbf{y}$, where $\mathbf{C}$ is a $p \times N$ matrix. Suppose we also define $\mathbf{D} = \mathbf{C} - (\mathbf{X}^\top \mathbf{X})^{-1}\mathbf{X}^\top$. We can then write

$$\begin{aligned}
\tilde{\beta} = \mathbf{C}\mathbf{y} &= (\mathbf{D} + (\mathbf{X}^\top \mathbf{X})^{-1}\mathbf{X}^\top)\mathbf{y} \\
&= \mathbf{D}\mathbf{y} + (\mathbf{X}^\top \mathbf{X})^{-1}\mathbf{X}^\top)\mathbf{y} \\
&= \mathbf{D}\mathbf{y} + \hat{\beta}^{\text{OLS}}.
\end{aligned}$$

Using this expression, we can write the expectation of $\tilde{\beta}$ as

$$\begin{aligned}
\mathbb{E}[\tilde{\beta}] &= \mathbb{E}[\mathbf{D}\mathbf{y} + \hat{\beta}^{\text{OLS}}] \\
&= \mathbb{E}[\mathbf{D}(\mathbf{X}\beta + \varepsilon)] + \mathbb{E}[\hat{\beta}^{\text{OLS}}] \\
&= \mathbf{D}\mathbb{E}[\mathbf{X}\beta] + \mathbf{D}\mathbb{E}[\varepsilon] + \mathbb{E}[\hat{\beta}^{\text{OLS}}] \\
&= \mathbf{D}\mathbf{X}\beta + \beta.
\end{aligned}$$

Since $\tilde{\beta}$ is unbiased, this immediately implies that $\mathbf{D}\mathbf{X} = 0$. Using this fact, we can now compute the variance of $\tilde{\beta}$ as

$$\begin{aligned}
\text{Var}(\tilde{\beta}) &= \text{Var}(\mathbf{D}\mathbf{y} + \hat{\beta}^{\text{OLS}}) \\
&= \text{Var}(\mathbf{D}(\mathbf{X}\beta + \varepsilon) + (\mathbf{X}^\top \mathbf{X})^{-1}\mathbf{X}^\top(\mathbf{X}\beta + \varepsilon)) \\
&= \text{Var}((\mathbf{D} + (\mathbf{X}^\top \mathbf{X})^{-1}\mathbf{X}^\top)\varepsilon) \\
&= (\mathbf{D} + (\mathbf{X}^\top \mathbf{X})^{-1}\mathbf{X}^\top)\text{Var}(\varepsilon)(\mathbf{D} + (\mathbf{X}^\top \mathbf{X})^{-1}\mathbf{X}^\top)^\top \\
&= \sigma^2(\mathbf{D}\mathbf{D}^\top + \mathbf{D}\mathbf{X}(\mathbf{X}^\top \mathbf{X})^{-1} + (\mathbf{X}^\top \mathbf{X})^{-1}(\mathbf{D}\mathbf{X})^\top + (\mathbf{X}^\top \mathbf{X})^{-1} \\
&= \sigma^2(\mathbf{D}\mathbf{D}^\top + (\mathbf{X}^\top \mathbf{X})^{-1}) \\
&= \sigma^2\mathbf{D}\mathbf{D}^\top + \text{Var}(\hat{\beta}^{\text{OLS}})
\end{aligned}$$

The conclusion now follows straightforwardly, using the fact that $\mathbf{D}\mathbf{D}^\top$ is positive semi-definite. □

In general, we can use the (matrix) *mean squared error* (MSE) as a criterion by which to assess a given estimator $\hat{\beta}$ of the parameter β. This is defined as[2]

$$\begin{aligned}\text{MSE}(\hat{\beta}) &= \mathbb{E}[(\hat{\beta}-\beta)(\hat{\beta}-\beta)^\top] \\ &= \text{Var}(\hat{\beta}) + \left(\mathbb{E}[\hat{\beta}]-\beta\right)\left(\mathbb{E}[\hat{\beta}]-\beta\right)^\top.\end{aligned}$$

In this expression, the first term is the variance, while the second term is the squared bias. From the Gauss-Markov Theorem, it is clear that $\hat{\beta}^{\text{OLS}}$ has the smallest MSE amongst all linear unbiased estimators of β. However, there may exist a biased estimator with a significantly smaller variance, and therefore a smaller overall MSE. This is sometimes referred to as the *bias-variance tradeoff*. We will discuss this issue in more detail in Chapter 5.

1.1.4 Linear Models in `R`

`R` makes it very straightforward to fit linear models via the `lm()` function. The basis syntax for this function is

```
model = lm(formula, data)
```

where `formula` is the formula for the linear model, in the form $\texttt{y} \sim \texttt{x}_1 + \texttt{x}_2 + \ldots$, and `data` is the name of the data frame that contains the data. The resulting linear model, in our case `model`, is an object of class `lm`.

This object contains many useful quantities, which we can extract using the syntax `model$quantity`. For example, `model$coefficients` will return the fitted model parameters, `model$fitted.values` will return the fitted values, and `model$residuals` will return the residuals.

We can also apply other built-in functions to this object. For example, `summary(model)` will return a summary of the model, including the parameter estimates, standard-errors, etc. Meanwhile, `predict(model,newdata)` will return the predicted value of the response, given a set of new values of the covariates `newdata`.

We will demonstrate some of this functionality using the Real Estate dataset, which can be found in the UCI machine learning repository. This dataset contains 414 observations, each containing 7 attributes relating to houses in New Taipei City, Taiwan. These attributes are:

- `date`. The transaction date.

[2] We note that the term *mean squared error* may also be used to denote the scalar valued quantity

$$\begin{aligned}\text{MSE}(\hat{\beta}) &= \mathbb{E}[(\hat{\beta}-\beta)^\top(\hat{\beta}-\beta)] \\ &= \text{Tr}\left[\text{Var}(\hat{\beta})\right] + \left(\mathbb{E}[\hat{\beta}]-\beta\right)^\top\left(\mathbb{E}[\hat{\beta}]-\beta\right).\end{aligned}$$

In this chapter, we will generally use the first of these definitions, unless explicitly stated otherwise.

- `age`. The age of the house.
- `distance`. The distance to the nearest metro station.
- `stores`. The number of convenience stores within walking distance.
- `latitude`. The latitude.
- `longitude`. The longitude.
- `price`. The house price per unit area.

We will treat the `price` as the response, and `date`, `age`, `distance`, `stores`, `latitude`, `longitude`, as the predictors.

```
# load data
data = read.csv('data/realestate.csv', row.names = 1)

# look at first few observations
head(data)

##       date  age   distance stores latitude longitude price
## 1 2012.917 32.0   84.87882     10 24.98298  121.5402  37.9
## 2 2012.917 19.5  306.59470      9 24.98034  121.5395  42.2
## 3 2013.583 13.3  561.98450      5 24.98746  121.5439  47.3
## 4 2013.500 13.3  561.98450      5 24.98746  121.5439  54.8
## 5 2012.833  5.0  390.56840      5 24.97937  121.5425  43.1
## 6 2012.667  7.1 2175.03000      3 24.96305  121.5125  32.1

# dimensions of the data
dim(data)

## [1] 414   7

# consider age, distance, price
newdata = data[, c(2,3,7)]

# look at pairwise plots (uncomment to view)
# pairs(newdata, upper.panel=NULL, main="Real Estate Data")

# fit linear model
model = lm(price ~ age + distance, data=newdata)

# print short model summary
print(model)

##
## Call:
## lm(formula = price ~ age + distance, data = newdata)
```

```
##
## Coefficients:
## (Intercept)          age     distance
##   49.885586    -0.231027    -0.007209

# look at all stored attributes
names(model)

##  [1] "coefficients"  "residuals"     "effects"       "rank"
##  [5] "fitted.values" "assign"        "qr"            "df.residual"
##  [9] "xlevels"       "call"          "terms"         "model"

# coefficients
beta = model$coefficients

# residuals
res = model$residuals

# fitted values
fitvals = model$fitted.values

# predict response at new data
xnew = data.frame(age=15.1,distance=517.0)
predict(model, newdata=xnew)

##        1
## 42.67023
```

1.2 Subset Selection

1.2.1 Motivation

In the previous section, we saw the OLS estimator $\hat{\beta}^{\text{OLS}}$ had the smallest variance among all unbiased estimators, under the assumption that $\mathbf{X}$ had full column rank (so we could invert $\mathbf{X}^\top\mathbf{X}$). In modern data scenarios, however, this assumption may not hold. For example, it does not hold when p, the number of variables, is large relative to the number of observations n.

In theory, in this scenario we could perfectly fit the data with the model. In practice, however, such a model would probably have very poor predictive capabilities. This is an example of *overfitting*. The fitted may also be difficult to interpret, since it contains a very large number of explanatory variables.

The key idea in this setting is to only include a subset of the most relevant covariates in our analysis, or to "shrink" the OLS estimator towards 0 in an appropriate sense. This will not only result in an estimator with smaller MSE than the OLS estimator, but it will allow us to enforce the notion of *sparsity*.

This is the idea that only a very small number $k \ll p$ of variables are actually relevant for predicting $\mathbf{y}$, and the remaining variables should be ignored (i.e., the corresponding coefficients should be set to zero).

The key question is then: how do we choose the small number k of relevant covariates from among the p possibilities? In this section, we will discuss a number of methods for variable subset selection in linear regression. We will then discuss shrinkage methods, as well as other dimension-reduction strategies. These methods all fall under the topic of *model selection*, which we will cover in greater generality in Chapter 5.

1.2.2 Best Subset Selection

Best subset selection does what it says on the tin. For any chosen $k \in \{0, 1, \ldots, p\}$, the best subset of size k is the subset of size k which returns the smallest RSS. This approach is summarised in Algorithm 1

Algorithm 1: Best Subset Selection

---

1. Let $\mathcal{M}_0$ denote the null model, which contains no predictors.
2. For $k = 1, 2, \ldots, p$:
   (a) Fit all $\binom{p}{k}$ models that contain exactly k predictors.
   (b) Choose the *best* model from among these $\binom{p}{k}$ models, and call it $\mathcal{M}_k$, where *best* is defined as having the smallest RSS.
3. Select a single best model from among $\mathcal{M}_0, \ldots, \mathcal{M}_p$ (see Section 1.2.4).

Interestingly, the best subset of size k need not include (any of) the variables in the best subset of size $k - 1$. In many cases, however, there is significant overlap between the best subsets of successive sizes. The best subsets of size $1, 2, \ldots,$ may even be *nested*. That is, for $k = 1, 2, \ldots,$ all of the covariates in the best subset of size k are also in the best subset of size $k + 1$.

In practice, an efficient algorithm known as the *leaps and bounds* procedure makes it possible to perform best subset regression for all $k \in \{1, \ldots, p\}$, for p as large as 30 or 40. For larger p, however, it is infeasible to search through all 2^p possible models, and we will need an alternative approach.

1.2.3 Forward and Backward Stepwise Selection

The following stepwise approaches provide a computationally feasible way of systematically searching for a model.

Forward stepwise selection. This *greedy algorithm* starts with the simplest possible model (intercept only), and iteratively adds to the model the predictor that most improves the model fit (e.g., most decreases the RSS). This procedure

is continued until we reach the model which includes all of the covariates, or else when some stopping criteria is satisfied (e.g., the model has the desired number of covariates). This algorithm is summarised in Algorithm 2.

Algorithm 2: Forward Stepwise Selection

---

1. Let $\mathcal{M}_0$ denote the null model, which contains no covariates.
2. For $k = 0, 1, \ldots, p-1$:
   (a) Consider all $p - k$ models that augment the covariates in $\mathcal{M}_k$ with one additional covariate.
   (b) Choose the *best* model among these $p - k$ models, and call it $\mathcal{M}_{k+1}$, where *best* is defined as having the smallest RSS.
3. Select a single best model from among $\mathcal{M}_0, \ldots, \mathcal{M}_p$ (see Section 1.2.4).

Backward stepwise selection. This algorithm starts with the model containing all p predictors, and iteratively removes from the model the covariate that has the least impact on the model fit (e.g., the covariate with the largest p-value). Once again, this procedure is continued until a certain stopping criteria is met (e.g., all of the p-values are below some threshold). This approach is summarised in Algorithm 3.

Algorithm 3: Backward Stepwise Selection

---

1. Let $\mathcal{M}_p$ denote the full model, which contains all p covariates.
2. For $k = p, p-1, \ldots, 1$:
   (a) Consider all k models that contain all but one of the covariates in $\mathcal{M}_k$, for a total of $k - 1$ covariates.
   (b) Choose the *best* model among these k models, and call it $\mathcal{M}_{k-1}$, where *best* is defined as having the smallest RSS, or by some alternative criteria.
3. Select a single best model from among $\mathcal{M}_0, \ldots, \mathcal{M}_p$ (see Section 1.2.4).

Forward and backward stepwise selection are not guaranteed to return the best model containing a particular subset of the p predictors. There are, however, several reasons to prefer these approaches over best subset selection. Firstly, even when p is very large, it is computationally feasible to compute the forward and backward stepwise sequences. Secondly, even if p is small enough to allow best subset selection, looking at all possible models may not be the

best thing to do, since it can lead to *overfitting*.

1.2.4 Choosing the Optimal Model

Best subset selection, forward selection, and backward selection all result in a set of models, each of which contains k of the p predictors. To apply these methods, we now need a way to determine which of these models - or, equivalently, which value of k - is optimal.

Our first thought might be to try to use the RSS to compare the models. However, this doesn't turn out to be very helpful. Indeed, adding variables to the model monotonically decreases the RSS, and so minimising the RSS among our candiate models would lead to us always choosing the model which includes all of the covariates. The problem is that a low RSS indicates a low *training error*, whereas we would like to choose a model with a low *test error*.

In order to obtain a parsimonious model with relatively few variables, which performs well on unseen data, there are a number of criteria that we can use. Roughly speaking, these criteria provide an indirect estimate of the test error, by making an adjustment to the training error which accounts for the bias due to overfitting. In particular, these criteria will all take the form

$$\text{goodness-of-fit} + \text{penalty on the number of variables}, \tag{1.5}$$

with common choices including the *Akaike information criterion* (AIC), the *Bayesian information criterion* (BIC), and Mallow's C_p. In the linear Gaussian case, these three criteria are given explicitly by

$$\text{AIC}(k) = N \log\left(\frac{\text{RSS}}{N}\right) + 2k \tag{1.6}$$

$$\text{BIC}(k) = N \log\left(\frac{\text{RSS}}{N}\right) + k \log(N) \tag{1.7}$$

$$C_p(k) = \text{RSS} + 2k\hat{\sigma}^2_{\text{full}} \tag{1.8}$$

where k denotes the number of variables included in the current model, and $\hat{\sigma}^2_{\text{full}}$ denotes an estimate of the residual variance based on the full model. Let's take a quick look at how Mallows' C_p arises. The following derivation is **non-examinable**.

Suppose, now, that in addition to the original *training data* $\mathcal{D}_N = \{(x_i, y_i\}_{i=1}^N$, we now have an additional set of *test data* $\mathcal{D}_N^* = \{(x_i, y_i^*)\}_{i=1}^N$. We note that the covariates are the same for the test data and the training data, but the responses in the test data are newly observed. We can then compute the expected

MSE for the training data and the test data as

$$\begin{aligned}
\mathbb{E}[||\mathbf{y} - \hat{\mathbf{y}}||^2] &= \mathbb{E}[||(\mathbf{I} - \mathbf{H})\mathbf{y}||^2] \\
&= \mathbb{E}[||(\mathbf{I} - \mathbf{H})(\mathbf{X}\beta + \varepsilon)||^2] \\
&= \mathbb{E}[||(\mathbf{I} - \mathbf{H})\varepsilon||^2] \\
&= \mathbb{E}[\mathrm{Tr}(\varepsilon^\top (\mathbf{I} - \mathbf{H})^\top (\mathbf{I} - \mathbf{H})\varepsilon)] \\
&= \mathbb{E}[\mathrm{Tr}((\mathbf{I} - \mathbf{H})^\top (\mathbf{I} - \mathbf{H})\varepsilon\varepsilon^\top)] \\
&= \mathrm{Tr}((\mathbf{I} - \mathbf{H})^\top (\mathbf{I} - \mathbf{H})\mathbb{E}[\varepsilon\varepsilon^\top]) = (n-k)\sigma^2
\end{aligned}$$

$$\begin{aligned}
\mathbb{E}[||\mathbf{y}^* - \hat{\mathbf{y}}||^2] &= \mathbb{E}[||(\mathbf{y}^* - \mathbf{X}\beta) + (\mathbf{X}\beta - \mathbf{X}\hat{\beta}^{\mathrm{OLS}})||^2] \\
&= \mathbb{E}[||\varepsilon^*||^2] + 2\mathbb{E}[\varepsilon^\top (\mathbf{X}\beta - \mathbf{X}\hat{\beta}^{\mathrm{OLS}})] + \mathbb{E}[||\mathbf{X}(\beta - \hat{\beta}^{\mathrm{OLS}})||^2] \\
&= n\sigma^2 + \mathbb{E}[\mathrm{Tr}((\hat{\beta}^{\mathrm{OLS}} - \beta)^\top \mathbf{X}^\top \mathbf{X}(\hat{\beta}^{\mathrm{OLS}} - \beta))] \\
&= n\sigma^2 + \mathrm{Tr}(\mathbf{X}^T\mathbf{X}\mathrm{Var}(\hat{\beta}^{\mathrm{OLS}})) \\
&= n\sigma^2 + \mathrm{Tr}(\mathbf{X}^T\mathbf{X}(\mathbf{X}^T\mathbf{X})^{-1})\sigma^2 \\
&= n\sigma^2 + \mathrm{Tr}(\mathbf{H})\sigma^2 = (n+k)\sigma^2.
\end{aligned}$$

We thus have, in particular, that

$$\mathbb{E}[||\mathbf{y}^* - \hat{\mathbf{y}}||^2] = \mathbb{E}[||\mathbf{y} - \hat{\mathbf{y}}||^2] + 2k\sigma^2. \tag{1.9}$$

This expression suggests that the training error plus $2k\hat{\sigma}^2$, where $\hat{\sigma}^2$ is an estimate of σ^2, will provide a reasonable approximation of the test error. We will return to some of these ideas in more detail in Chapter 5.

1.2.5 Subset Selection in R

There are several tools available in R for performing subset selection:

- The `regsubsets()` function from the `leaps` package allows us to perform best subset selection.
- The `step()` function (builtin) allows us to perform forward and backward stepwise selection.

We will demonstrate these tools using the `diabetes` dataset from the `lars` package. This dataset includes ten baseline variables (`age`, `sex`, `body mass index`, `blood pressure`, and six blood serum measurements) for each of 442 diabetes patients, as well as the outcome of interest, a quantitative measure of disease progression one year after baseline. Let's start by looking at best subset selection.

```
# load required packages
library(leaps)
```

```r
library(lars)

# load dataset
data(diabetes)
data = data.frame(cbind(diabetes$x, y = diabetes$y))
N = nrow(data)

# fit linear model (all covariates)
full_model = lm(y ~ ., data = data)
full_model_summary = summary(full_model)

# best subsets
best_subsets = regsubsets(x = as.matrix(data[, -11]),
    y = data[, 11], nvmax = 10)

# summary of results
best_subsets_summary = summary(best_subsets)
best_subsets_rss = best_subsets_summary$rss
best_subsets_summary
```

```
Subset selection object
10 Variables  (and intercept)
    Forced in Forced out
age     FALSE      FALSE
sex     FALSE      FALSE
bmi     FALSE      FALSE
map     FALSE      FALSE
tc      FALSE      FALSE
ldl     FALSE      FALSE
hdl     FALSE      FALSE
tch     FALSE      FALSE
ltg     FALSE      FALSE
glu     FALSE      FALSE
1 subsets of each size up to 10
Selection Algorithm: exhaustive
          age sex bmi map tc  ldl hdl tch ltg glu
1  ( 1 )  " " " " "*" " " " " " " " " " " "*" " "
2  ( 1 )  " " " " "*" " " " " " " " " " " "*" " "
3  ( 1 )  " " " " "*" "*" " " " " " " " " "*" " "
4  ( 1 )  " " " " "*" "*" "*" " " " " " " "*" " "
5  ( 1 )  " " "*" "*" "*" " " " " " " "*" " "*" " "
6  ( 1 )  " " "*" "*" "*" "*" "*" " " " " "*" " "
7  ( 1 )  " " "*" "*" "*" "*" "*" " " "*" "*" " "
8  ( 1 )  " " "*" "*" "*" "*" "*" " " "*" "*" "*"
9  ( 1 )  " " "*" "*" "*" "*" "*" "*" "*" "*" "*"
10  ( 1 ) "*" "*" "*" "*" "*" "*" "*" "*" "*" "*"
```

```r
# model size
modsize = apply(best_subsets_summary$which, 1, sum)

# compute AIC, BIC, Cp
AIC = N * log(best_subsets_rss/N) + 2 * modsize
BIC = N * log(best_subsets_rss/N) + modsize * log(N)
Cp = best_subsets_rss + 2 * modsize * full_model_summary$sigma^2

# best models
aic_best = best_subsets_summary$which[which(AIC ==
    min(AIC)), ]
bic_best = best_subsets_summary$which[which(BIC ==
    min(BIC)), ]
cp_best = best_subsets_summary$which[which(Cp == min(Cp)),
    ]

print(aic_best)
```

```
(Intercept)         age         sex         bmi         map
       TRUE       FALSE        TRUE        TRUE        TRUE
         tc         ldl         hdl         tch         ltg
       TRUE        TRUE       FALSE       FALSE        TRUE
        glu
      FALSE
```

```r
print(bic_best)
```

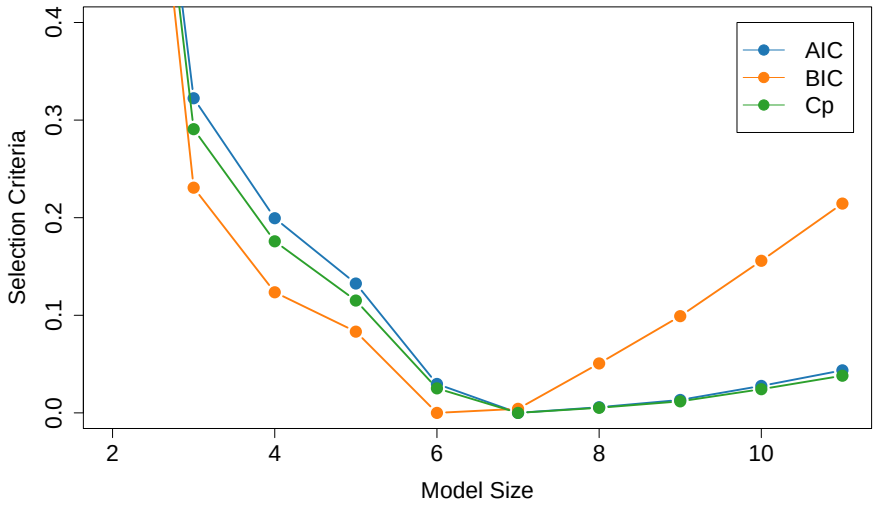
```
(Intercept)         age         sex         bmi         map
       TRUE       FALSE        TRUE        TRUE        TRUE
         tc         ldl         hdl         tch         ltg
      FALSE       FALSE        TRUE       FALSE        TRUE
        glu
      FALSE
```

```r
print(cp_best)
```

```
(Intercept)         age         sex         bmi         map
       TRUE       FALSE        TRUE        TRUE        TRUE
         tc         ldl         hdl         tch         ltg
       TRUE        TRUE       FALSE       FALSE        TRUE
        glu
      FALSE
```

```r
# rescale to [0,1] and plots results
rescale <- function(x) {
    return((x - min(x))/(max(x) - min(x)))
}
```

```
AIC = rescale(AIC)
BIC = rescale(BIC)
Cp = rescale(Cp)

par(mfrow = c(1, 1))
par(mar = c(5, 5, 2, 2))
plot(range(modsize), c(0, 0.4), type = "n", xlab = "Model Size",
    ylab = "Selection Criteria", cex.lab = 1.5, cex.axis = 1.5)
points(modsize, AIC, col = "#1f77b4", type = "b", pch = 19,
    cex = 1.5, lwd = 2)
points(modsize, BIC, col = "#ff7f0e", type = "b", pch = 19,
    cex = 1.5, lwd = 2)
points(modsize, Cp, col = "#2ca02c", type = "b", pch = 19,
    cex = 1.5, lwd = 2)
legend(x = 9.7, y = 0.4, legend = c("AIC", "BIC", "Cp"),
    col = c("#1f77b4", "#ff7f0e", "#2ca02c"), lty = rep(1,
        3), pch = 19, cex = 1.5)
```



We now take a look at forward and backward stepwise selection using the `steps()` function.

```
# fit null model
null_model = lm(y ~ 1, data = data)

# forward stepwise selection (using AIC)
step(null_model, direction = "forward", trace = 1,
    scope = list(upper = full_model, lower = ~1))

Start:  AIC=3841.99
```

```
y ~ 1

       Df Sum of Sq     RSS    AIC
+ bmi   1    901427 1719582 3657.7
+ ltg   1    839310 1781699 3673.4
+ map   1    510856 2110154 3748.2
+ tch   1    485646 2135363 3753.4
+ hdl   1    408507 2212502 3769.1
+ glu   1    383437 2237572 3774.1
+ tc    1    117824 2503186 3823.7
+ age   1     92527 2528482 3828.1
+ ldl   1     79403 2541607 3830.4
<none>              2621009 3842.0
+ sex   1      4860 2616149 3843.2

Step:  AIC=3657.7
y ~ bmi

       Df Sum of Sq     RSS    AIC
+ ltg   1    302888 1416694 3574.1
+ map   1    136477 1583105 3623.1
+ tch   1    111511 1608071 3630.1
+ hdl   1     97767 1621815 3633.8
+ glu   1     73738 1645844 3640.3
+ age   1     17087 1702495 3655.3
+ tc    1     12008 1707574 3656.6
<none>              1719582 3657.7
+ ldl   1      1228 1718354 3659.4
+ sex   1       197 1719385 3659.6

Step:  AIC=3574.06
y ~ bmi + ltg

       Df Sum of Sq     RSS    AIC
+ map   1     53986 1362708 3558.9
+ tc    1     27623 1389071 3567.4
+ hdl   1     26914 1389780 3567.6
+ ldl   1      9255 1407439 3573.2
+ sex   1      6881 1409813 3573.9
+ glu   1      6801 1409893 3573.9
<none>              1416694 3574.1
+ tch   1      2376 1414318 3575.3
+ age   1       176 1416518 3576.0

Step:  AIC=3558.88
```

```
y ~ bmi + ltg + map

       Df Sum of Sq     RSS    AIC
+ tc    1   31277.5 1331430 3550.6
+ hdl   1   29921.5 1332786 3551.1
+ sex   1   17532.7 1345175 3555.2
+ ldl   1   10809.9 1351898 3557.4
<none>              1362708 3558.9
+ tch   1    3218.7 1359489 3559.8
+ age   1    2106.6 1360601 3560.2
+ glu   1    1240.0 1361468 3560.5

Step:  AIC=3550.62
y ~ bmi + ltg + map + tc

       Df Sum of Sq     RSS    AIC
+ sex   1   20561.3 1310869 3545.7
+ ldl   1   18081.1 1313349 3546.6
+ tch   1   15188.2 1316242 3547.5
+ hdl   1   14360.6 1317070 3547.8
<none>              1331430 3550.6
+ glu   1    2898.7 1328532 3551.7
+ age   1     472.1 1330958 3552.5

Step:  AIC=3545.74
y ~ bmi + ltg + map + tc + sex

       Df Sum of Sq     RSS    AIC
+ ldl   1     39378 1271491 3534.3
+ tch   1     35592 1275277 3535.6
+ hdl   1     35002 1275867 3535.8
<none>              1310869 3545.7
+ glu   1      5287 1305581 3546.0
+ age   1        49 1310820 3547.7

Step:  AIC=3534.26
y ~ bmi + ltg + map + tc + sex + ldl

       Df Sum of Sq     RSS    AIC
<none>              1271491 3534.3
+ tch   1    3686.2 1267805 3535.0
+ glu   1    3532.4 1267959 3535.0
+ hdl   1     394.8 1271097 3536.1
+ age   1      10.9 1271480 3536.3
```

```
Call:
lm(formula = y ~ bmi + ltg + map + tc + sex + ldl, data = data)

Coefficients:
(Intercept)          bmi          ltg          map
      152.1        529.9        804.2        327.2
         tc          sex          ldl
     -757.9       -226.5        538.6
```

```r
# backward stepwise selection (using BIC, trace=0
# so no output)
step(full_model, direction = "backward", trace = 0,
    k = log(N))
```

```
Call:
lm(formula = y ~ sex + bmi + map + tc + ldl + ltg, data = data)

Coefficients:
(Intercept)          sex          bmi          map
      152.1       -226.5        529.9        327.2
         tc          ldl          ltg
     -757.9        538.6        804.2
```

1.3 Shrinkage Estimators

1.3.1 Motivation

By keeping a subset of the covariates and discarding the remainder, subset selection allows us to obtain a model that is interpretable. It may also have lower prediction error than the full model. On the other hand, since it is a discrete process (i.e., variables are either retained or removed), it often has high variance. Shrinkage methods are more continuous, and don't suffer as much from high variability.

1.3.2 Ridge regression

Our first shrinkage estimator is the ridge regression estimator $\hat{\beta}^{\mathrm{r}}_{\lambda}$, which minimizes the following quantity,

$$\hat{\beta}^{\mathrm{r}}_{\lambda} = \underset{\beta \in \mathbb{R}^p}{\arg\min} \left(\|\mathbf{y} - \mathbf{X}\beta\|_2^2 + \lambda \|\beta\|_2^2 \right), \tag{1.10}$$

where $\lambda \geq 0$ is a user specified *tuning parameter* which controls the degree of shrinkage: the larger the value of λ, the larger the amount of shrinkage. We

see that this is a *penalised* form of our RSS objective. In the case that $\lambda = 0$, the penalty term has no effect, and we recover the standard (unpenalised) RSS. Meanwhile, when $\lambda \to \infty$, all coefficients are shrunk towards 0.

The ridge estimator can be equivalently be found by solving a constrained optimization problem,

$$\begin{aligned} \hat{\beta}_t^{\mathrm{r}} = \arg\min_{\beta} \left(\|\mathbf{y} - \mathbf{X}\beta\|_2^2 \right), \\ \text{subject to } \|\beta\|_2^2 \leq t, \end{aligned} \tag{1.11}$$

where $t \geq 0$ is also a user-defined tuning parameter. This formulation makes explicit the size constraints on the parameters. There is a one-to-one correspondence between the parameters λ in (1.11) and the parameters t in (1.10). That is, for each value of λ in (1.10), there is a corresponding value of t in (1.11) such that $\hat{\beta}_t^{\mathrm{r}} = \hat{\beta}_\lambda^{\mathrm{r}}$, and vice-versa.

The ridge solutions are not invariant under scalings of the inputs, and so we usually standardise the inputs before solving (1.10). In particular, we center both the response and the regressors, setting $y_i \leftarrow y_i - \bar{y}$ and $x_{ij} \leftarrow x_{ij} - \bar{x}_j$, where $\bar{y} = N^{-1} \sum_{i=1}^N y_i$ and $\bar{x}_j = N^{-1} \sum_{i=1}^N x_{ij}$. This rescaling implies, in particular, that the intercept is identically zero, and thus we do not include a column of ones in our design matrix.

We can actually obtain the ridge regression estimator in closed form, as is shown in the following result.

Proposition 3. *The ridge regression estimator* (1.10) *is given by*

$$\hat{\beta}_\lambda^{\mathrm{r}} = (\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I}_p)^{-1} \mathbf{X}^\top \mathbf{y}.$$

Proof. The derivation is very similar to the derivation of the OLS estimator. In particular, we differentiate (1.10), and set the resulting expression equal to zero. We can then check the second derivatives to see that the solution is, indeed, a minimiser. □

It's worth making a couple of remarks about this estimator. The ridge regression solution is still linear in $\mathbf{y}$, but it is now biased (prove this!). Even if $\mathbf{X}$ is not full rank, $\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I}_p$ is still invertible for $\lambda > 0$. The solution adds a constant to the diagonal of $\mathbf{X}^\top \mathbf{X}$ before inversion. This is, in fact, the origin of the name 'ridge' regression.

The advantage of the ridge regression estimator over the OLS estimator is rooted in the bias-variance tradeoff. As the penalty parameter λ increases, the flexibility of the ridge regression fit decreases, leading to decreased variance but increased bias. At $\lambda = 0$, the ridge regression estimate $\hat{\beta}_\lambda^{\mathrm{r}}$ coincides with the OLS estimate $\hat{\beta}^{\mathrm{OLS}}$, which has zero bias but high variance. As λ increases, the shrinkage of $\hat{\beta}_\lambda^{\mathrm{r}}$ leads to a subtantial reduction in the variance relative to $\hat{\beta}^{\mathrm{OLS}}$, at the expense of a slight increase in the bias. As a result, it is possible to show that, for sufficiently small λ, the MSE of $\hat{\beta}_\lambda^{\mathrm{r}}$ is smaller than the MSE of $\hat{\beta}^{\mathrm{OLS}}$. This is the subject of the following theorem.

Theorem 4. *Assuming* $\mathbf{X}$ *has full column rank, then for* λ *sufficiently small, we have*

$$\mathbb{E}[(\hat{\beta}^{\mathrm{OLS}} - \beta)(\hat{\beta}^{\mathrm{OLS}} - \beta)^\top] - \mathbb{E}[(\hat{\beta}^{\mathrm{r}}_\lambda - \beta)(\hat{\beta}^{\mathrm{r}}_\lambda - \beta)^\top]$$

is positive definite.

Proof. Using the expression given in Proposition 3, we can compute the expectation of $\hat{\beta}^{\mathrm{r}}_\lambda$ as

$$\mathbb{E}[\hat{\beta}^{\mathrm{r}}_\lambda] = (\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^\top\mathbf{X}\beta.$$

We can also compute the variance as

$$\mathrm{Var}(\hat{\beta}^{\mathrm{r}}_\lambda) = \sigma^2(\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^\top\mathbf{X}(\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I})^{-1}.$$

It follows, after some simplification, that

$$\begin{aligned}&\mathbb{E}[(\hat{\beta}^{\mathrm{OLS}} - \beta)(\hat{\beta}^{\mathrm{OLS}} - \beta)^\top] - \mathbb{E}[(\hat{\beta}^{\mathrm{r}}_\lambda - \beta)(\hat{\beta}^{\mathrm{r}}_\lambda - \beta)^\top] \\ &= \lambda(\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I})^{-1}[\sigma^2\{2\mathbf{I} + \lambda(\mathbf{X}^\top\mathbf{X})^{-1}\} - \lambda\beta\beta^\top](\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I})^{-1}.\end{aligned}$$

Thus $\mathbb{E}[(\hat{\beta}^{\mathrm{OLS}} - \beta)(\hat{\beta}^{\mathrm{OLS}} - \beta)^\top] - \mathbb{E}[(\hat{\beta}^{\mathrm{r}}_\lambda - \beta)(\hat{\beta}^{\mathrm{r}}_\lambda - \beta)^\top]$ is positive definite for $\lambda > 0$ if and only if

$$\sigma^2\{2\mathbf{I} + \lambda(\mathbf{X}^\top\mathbf{X})^{-1}\} - \lambda\beta\beta^\top$$

is positive definite. This is true for all $0 < \lambda < 2\sigma^2/\|\beta\|_2^2$. □

The *singular value decomposition* (SVD) of the centered input matrix $\mathbf{X}$ can give us some additional insight into the nature of ridge regression. The SVD generalises the eigendecomposition of a square matrix, and can be applied to matrix of any dimensions. Given $\mathbf{X} \in \mathbb{R}^{N\times p}$, the SVD has the form

$$\mathbf{X} = \mathbf{U}\mathbf{D}\mathbf{V}^\top,$$

where $\mathbf{U} \in \mathbb{R}^{N\times p}$ and $\mathbf{V} \in \mathbb{R}^{p\times p}$ are orthogonal matrices, with the columns of $\mathbf{U}$ spanning the column space of $\mathbf{X}$, and the columns of $\mathbf{V}$ spanning the row space of $\mathbf{X}$. Meanwhile, $\mathbf{D} \in \mathbb{R}^{p\times p}$ is a diagonal matrix, with diagonal entries $D_{11} \geq D_{22} \geq \cdots \geq D_{pp} \geq 0$ called the singular values of $\mathbf{X}$. If one or more values $D_{jj} = 0$, then the matrix $\mathbf{X}$ is singular.

Using the singular value decomposition, after some algebra, we can write the least squares fitted values as

$$\begin{aligned}\mathbf{X}\hat{\beta}^{\mathrm{OLS}} &= \mathbf{X}(\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{y} \\ &= \mathbf{U}\mathbf{U}^\top\mathbf{y}.\end{aligned}$$

We note that $\mathbf{U}^\top\mathbf{y}$ are the coordinate of $\mathbf{y}$ with respect to the orthonormal basis $\mathbf{U}$. We can also write the the ridge solutions (i.e., the ridge fitted values) as

$$\begin{aligned}\mathbf{X}\hat{\beta}^{\mathrm{r}}_\lambda &= \mathbf{X}(\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^\top\mathbf{y} \\ &= \mathbf{U}\mathbf{D}(\mathbf{D}^2 + \lambda\mathbf{I})^{-1}\mathbf{D}\mathbf{U}^\top\mathbf{y} \\ &= \sum_{j=1}^{p} \mathbf{u}_j \frac{D_{jj}^2}{D_{jj}^2 + \lambda}\mathbf{u}_j^\top\mathbf{y},\end{aligned}$$

where $\mathbf{u}_j$ are the columns of $\mathbf{U}$. Thus, like linear regression, ridge regression computes the coordinate of $\mathbf{y}$ with respect to the orthonormal basis of $\mathbf{U}$. It then shrinks these coordinates by the factors $D_{jj}^2/(D_{jj}^2+\lambda) \leq 1$. This implies that a greater amount of shrinkage is applied to the coordinates of the basis vectors associated with smaller D_{jj}^2.

For a standard linear model with p covariates, we know that the *degrees-of-freedom* (DF) is simply equal to p, the number of free parameters. More formally, we have the definition[3]

$$\mathrm{DF} = \frac{1}{\sigma^2}\sum_{i=1}^{n}\mathrm{Cov}(y_i, \hat{y}_i) = \frac{1}{\sigma^2}\mathrm{Tr}[\mathrm{Cov}(\mathbf{y}, \hat{\mathbf{y}})],$$

which we can also use to calculate to compute the *effective degrees-of-freedom* for the ridge regression fit. In particular, we now have

$$\begin{aligned}\mathrm{DF}^{\mathrm{r}}(\lambda) &= \frac{1}{\sigma^2}\mathrm{Tr}[\mathrm{Cov}(\mathbf{y}, \mathbf{X}(\mathbf{X}^\top\mathbf{X}+\lambda\mathbf{I})^{-1}\mathbf{X}^T\mathbf{y})] \\ &= \mathrm{Tr}[\mathbf{X}(\mathbf{X}^\top\mathbf{X}+\lambda\mathbf{I})^{-1}\mathbf{X}^T] = \sum_{j=1}^{p}\frac{D_{jj}^2}{D_{jj}^2+\lambda}.\end{aligned}$$

We note that $\mathrm{DF}^{\mathrm{r}}(\lambda) \leq \mathrm{DF}^{\mathrm{OLS}} = p$, with equality if and only if $\lambda = 0$. That is, the effective degrees-of-freedom of the ridge regression fit is always smaller than the degrees-of-freedom of the standard linear regression fit, whenever the regularisation parameter λ isn't equal to zero.

1.3.3 The LASSO

The Least Absolute Shrinkage and Selection Operator (LASSO) is also a shrinkage estimator, which looks rather similar to ridge regression estimator. However, as we shall see below, it will turn out to have quite different and remarkable properties.

The Lasso estimator is defined as the solution of the following penalised least squares problem

$$\hat{\beta}_\lambda^{\mathrm{l}} = \underset{\beta\in\mathbb{R}^p}{\arg\min}\left\{\|\mathbf{y}-\mathbf{X}\beta\|_2^2 + \lambda\|\beta\|_1\right\}, \tag{1.12}$$

where $\lambda \geq 0$ is a tuning parameter, and $||\cdot||_1$ denotes the ℓ_1-norm, defined as

$$\|\beta\|_1 = \sum_{j=1}^{p}|\beta_j|.$$

[3]Using this definition, we can easily check that for the standard linear regression we have

$$\mathrm{DF}^{\mathrm{OLS}} = \frac{1}{\sigma^2}\mathrm{Tr}[\mathrm{Cov}(\mathbf{y}, \mathbf{X}(\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y})] = \mathrm{Tr}(\mathbf{X}(\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top) = \mathrm{Tr}(\mathbf{I}_p) = p.$$

It is worth noting the similarity with the definition of the ridge regression estimator: the previous ℓ_2 penalty $||\beta||_2$ has now been replaced by the ℓ_1 Lasso penalty $||\beta||_1$.

Similar to before, the Lasso estimator can equivalently be defined as the solution of a constrained optimization problem,

$$\begin{aligned} \hat{\beta}^{\mathrm{l}} = \arg\min_{\beta} \|\mathbf{y} - \mathbf{X}\beta\|_2^2, \\ \text{subject to } \|\beta\|_1 \leq t, \end{aligned} \tag{1.13}$$

for some parameter $t \geq 0$. Again there is a one-to-one correspondence between (1.12) and (1.13).

Unlike ridge regression, since we are now using the ℓ_1 norm, the Lasso solutions $\hat{\beta}^{\mathrm{l}}_\lambda$ are nonlinear in $\mathbf{y}$, and there is no longer a closed-form solution. However, efficient numerical algorithms exist for computing them.

The remarkable property of the Lasso is that, due to the nature of the constraint, it has the ability to set some parameters exactly equal to 0. This was not the case in ridge regression, where parameters were shrunk towards 0, but not set exactly equal to 0. Thus, the Lasso has the ability to do a form of continuous subset selection. As a result, models generated by the Lasso are generally much easier to interpret than those generated by ridge regression.

We provide a graphical representation of this in Figure 1.3. This figure depicts the Lasso (left) and ridge regression (right) estimates in the case that there are only two parameters. The residual sum of squares estimate has elliptical contours, centered at the OLS estimate. The constraint region for the Lasso is the diamond $|\beta_1| + |\beta_2| \leq t$, while the constraint region for ridge regression is the disk $\beta_1^2 + \beta_2^2 \leq t$. The Lasso estimate and the ridge regression estimate both correspond to the first point where the elliptical contours meet the constraint region. In the case of the Lasso, the constraint region has corners. Thus, if a solution occurs at a corner (e.g., the inner two diamonds in the figure), then one of the parameters will be set equal to zero.

1.3.4 Choosing the Tuning Parameter

Thus far, we haven't discussed how to determine the value of the tuning parameter λ. The most common way to choose λ is via *cross-validation*. Broadly speaking, the idea of this method is to choose the value of λ which minimises an estimate of the prediction error. We summarise one version of this method, known as *leave-one-out cross-validation*, in Algorithm 4. We will discuss this approach in greater detail in Chapter 5.

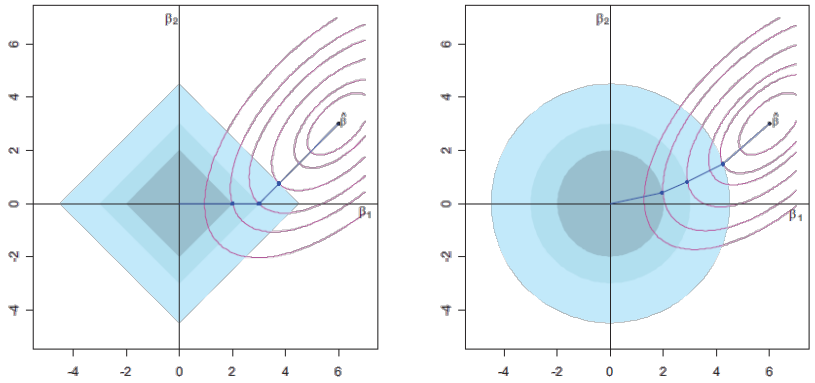


Figure 1.3: **Estimation using the Lasso (left) and ridge regression (right).** The plots show the contours of the error and the constraint functions. The solid blue areas represent the constraint regions $||\beta||_1 \leq t$ and $||\beta||_2^2 \leq t$, for different values of t. Meanwhile, the purple ellipses are the contours of the least squares function.

Algorithm 4: Leave-One-Out Cross-Validation

Inputs: grid of parameters $\lambda_1, \ldots, \lambda_M$, data $(x_1, y_1), \ldots, (x_N, y_N)$,
For: $j = 1, \ldots, M$,

- **For**: $i = 1, \ldots, N$,
  - Define the training and validation datasets

$$\mathcal{D}_{\text{train}} = \{(x_1, y_1), \ldots, (x_{i-1}, y_{i-1}), (x_{i+1}, y_{i+1}), \ldots, (x_N, y_N)\}$$
$$\mathcal{D}_{\text{val}} = \{(x_i, y_i)\}$$

  - Using $\mathcal{D}_{\text{train}}$, compute $\hat{\beta}^{\text{r}}_{-i,\lambda_j}$ or $\hat{\beta}^{\text{l}}_{-i,\lambda_j}$.
  - Using $\mathcal{D}_{\text{validation}}$, compute $\hat{y}_{i,\lambda_j} = x_i \hat{\beta}^{\text{r}}_{-i,\lambda_j}$ or $\hat{y}_{i,\lambda_j} = x_i \hat{\beta}^{\text{l}}_{-i,\lambda_j}$.
- Compute the MSE of the predictions,

$$\text{MSE}(\lambda_j) = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_{i,\lambda_j})^2$$

Output: λ^*, the penalty parameter with the lowest MSE,

$$\lambda^* = \underset{j \in \{1,\ldots,M\}}{\arg\min} \ \text{MSE}(\lambda_j).$$

1.3.5 Least Angle Regression

Least Angle Regression (LAR) is a relatively new method, and can be viewed as a 'democratic' version of forward stepwise regression. It is also intimately connected to the Lasso: in fact, it provides a very efficient algorithm for computing the entire Lasso path as the shrinkage parameter varies.

We previously discussed forward stepwise regression, which sequentially builds a model, adding one variable at a time. At each step, it identifies the best variable to include in the so-called *active set*, and then updates the least squares fit to include all of the active variables.

LAR uses a very similar strategy, but only includes 'as much' as a covariate as it 'deserves' to be there. In the first step, it identifies the variable which is most correlated with the response. Rather than fit this variable completely, LAR moves the coefficient of this variable continuously towards its least squares value. This causes its correlation with the residual to decrease in absolute value. Then, as soon as another variable 'catches up' in terms of correlation with the residual, the process is halted. The second variable then joins the active set, and their coefficient are moved together in a way which keeps their correlations tied and decreasing. This process is repeated until all of the variables are included in the model, and ends up at the full least-squares fit. This process is summarised in Algorithm 5.

Algorithm 5: Least Angle Regression.

---

1. Standardise covariates to have mean 0 and unit norm. Start with residual $\mathbf{r} = \mathbf{y} - \bar{\mathbf{y}}$, and set $\beta_1 = \cdots = \beta_p = 0$.

2. Find the covariate $\mathbf{x}_j$ most correlated with $\mathbf{r}$.

3. Move β_j from 0 towards its least squares coefficient $\langle \mathbf{x}_j, \mathbf{r} \rangle$, until some other $\mathbf{x}_k$ has as much correlation with the current residual as $\mathbf{x}_j$.

4. Move β_j and β_k in the direction defined by their joint least squares coefficients of the current residual on $(\mathbf{x}_j, \mathbf{x}_k)$, until some other competitor $\mathbf{x}_l$ has as much correlation with the current residual.

5. Continue in this fashion until all p covariates have been entered. The algorithm will terminate after $\min(N-1, p)$ steps, at which point we arrive at the full least-squares solution.

Suppose we write $\mathcal{A}_k \subset [p]$ for the active set of variables at the begin of the k^{th} step, and $\beta_{\mathcal{A}_k}$ for the coefficient vector of these variables at this step. There will be $k-1$ nonzero values and the one just entered will be zero. If the current

residual is $\mathbf{r}_k = \mathbf{y} - \mathbf{X}_{\mathcal{A}_k}\beta_{\mathcal{A}_k}$, where $\mathbf{X}_{\mathcal{A}_k}$ is the submatrix of $\mathbf{X}$ corresponding to the columns of $\mathbf{X}$ indexed by $\mathcal{A}_k$, then the direction for this step is

$$\delta_k = (\mathbf{X}_{\mathcal{A}_k}^\top \mathbf{X}_{\mathcal{A}_k})^{-1}\mathbf{X}_{\mathcal{A}_k}^\top \mathbf{r}_k,$$

and the coefficients evolves according to the update equation $\beta_{\mathcal{A}_k}(\alpha) = \beta_{\mathcal{A}_k} + \alpha \cdot \delta_k$. It is possible to show that the directions chosen in this way do indeed do what we claimed: they keep the correlations between the covariates tied and decreasing (show this!).

In fact, a simple modification of LAR will give an algorithm to compute the entire Lasso path, as a function of the tuning parameter λ.

Algorithm 6: LAR: Lasso modification

4a. If a nonzero coefficient hits zero, drop its variable from the active set of variables and recompute the current joint least squares direction.

1.3.6 Discussion

We have looked at several ways of restricting the linear regression model, which can handle the possibility of $p > n$: subset selection, ridge regression, and the Lasso. In the case that the design matrix $\mathbf{X}$ is orthonormal, so $\mathbf{X}^\top\mathbf{X} = \mathbf{I}_p$, these three procedures all have explicit solutions. In fact, each of these solutions correspond to simple transformations of the OLS estimate $\hat{\beta}^{\text{OLS}}$. This is shown in Table 1.1.

Ridge regression does a proportional shrinkage of the OLS coefficieints. The Lasso translate each coefficient by a constant factor λ, truncating at zero. This is referred to as 'soft thresholding'. Finally, best subset selection drops all variables with coefficients smaller than the k^{th} largest coefficient. This is a particular form of 'hard-thresholding'.

Estimator	Formula
Best subset (size M)	$\hat{\beta}_j^{\text{OLS}} \cdot I(\lvert\hat{\beta}_j^{\text{OLS}}\rvert \geq \lvert\hat{\beta}_{(M)}^{\text{OLS}}\rvert)$
Ridge	$\hat{\beta}_j^{\text{OLS}}/(1+\lambda)$
Lasso	$\text{sign}(\hat{\beta}_j^{\text{OLS}})(\lvert\hat{\beta}_j^{\text{OLS}}\rvert - \frac{\lambda}{2})_+$

Table 1.1: Estimators of β_j^* in the case of orthonormal design matrix $\mathbf{X}$. Recall $\text{sign}(x)$ sends a real number to its sign ± 1, and $(x)_+ = \max(x, 0)$ is the positive part.

1.3.7 Shrinkage Estimators in R

In R, we can fit ridge regression models and the LASSO using the function `glmnet()` from the package `glmnet`.

Let's take a look at an example, using the prostate cancer dataset `prostate` from the Prostate, Lung, Colorectal, and Ovariance Cancer Screening Trial, available via the `ElemStatLearn` package.

```
## Shrinkage Estimators ##

# load required packages
library(glmnet)
library(MASS)

# load data
data = read.csv('data/prostate.csv')
dim(data)

[1] 97  9

head(data)

      lcavol  lweight age      lbph svi       lcp gleason
1 -0.5798185 2.769459  50 -1.386294   0 -1.386294       6
2 -0.9942523 3.319626  58 -1.386294   0 -1.386294       6
3 -0.5108256 2.691243  74 -1.386294   0 -1.386294       7
4 -1.2039728 3.282789  58 -1.386294   0 -1.386294       6
5  0.7514161 3.432373  62 -1.386294   0 -1.386294       6
6 -1.0498221 3.228826  50 -1.386294   0 -1.386294       6
  pgg45       lpsa
1     0 -0.4307829
2     0 -0.1625189
3    20 -0.1625189
4     0 -0.1625189
5     0  0.3715636
6     0  0.7654678

# center and scale X
X = scale(data.matrix(data[, 1:8]), center = TRUE, scale = TRUE)

# center y
y = data[,9]
y = scale(y, center = TRUE, scale = FALSE)

# number of observations, covariates
N = nrow(X)
p = ncol(X)
```

```r
# range of lambda values from 10^10 to 10^-2
lambda_grid =  10^seq(1, -2, length = 100)

# ridge and lasso
ridge_fit = glmnet(X, y, lambda=lambda_grid,
                   standardize = TRUE, alpha = 0)
lasso_fit = glmnet(X, y, lambda=lambda_grid,
                   standardize = TRUE, alpha = 1)

betas_ridge = as.matrix(ridge_fit$beta)
betas_lasso = as.matrix(lasso_fit$beta)

# plot profile of coefficients
cols = c('#1f77b4', '#ff7f0e', '#2ca02c', '#d62728',
        '#9467bd', '#8c564b', '#e377c2', '#7f7f7f',
        '#bcbd22', '#17becf')
par(mfrow=c(1,2))
matplot(lambda_grid,t(coef(ridge_fit)[-1,]), log="x",
        type = "l", xlab = "Lambda", ylab = "Coefficients",
        lwd=2, lty=1, col=cols, main="Ridge",
        cex.lab=1.2, cex=1.2, cex.axis=1.2)
matplot(lambda_grid,t(coef(lasso_fit)[-1,]), log="x",
        type = "l", xlab = "Lambda", ylab = "Coefficients",
        lwd=2, lty=1, col=cols, main="Lasso",
        cex.lab=1.2, cex=1.2, cex.axis=1.2)

# use cross-validation to determine optimal lambda
ridge_fit_cv = cv.glmnet(X, y, standardize = TRUE, alpha=0)
lasso_fit_cv = cv.glmnet(X, y, standardize = TRUE, alpha=1)

# plot
plot(ridge_fit_cv, cex.lab=1.2)
plot(lasso_fit_cv, cex.lab=1.2)

# lambdas with smallest MSE
ridge_fit_cv$lambda.min
```

```
[1] 0.08434274
```

```r
lasso_fit_cv$lambda.min
```

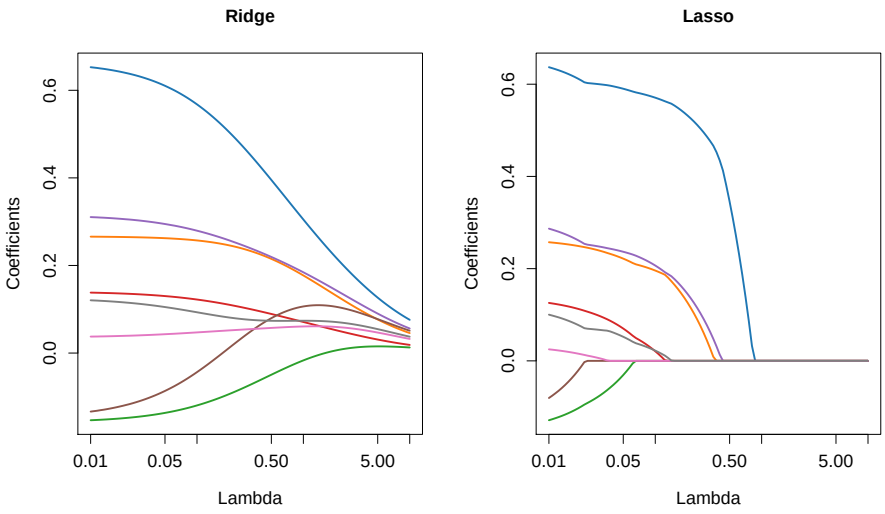
```
[1] 0.0269835
```

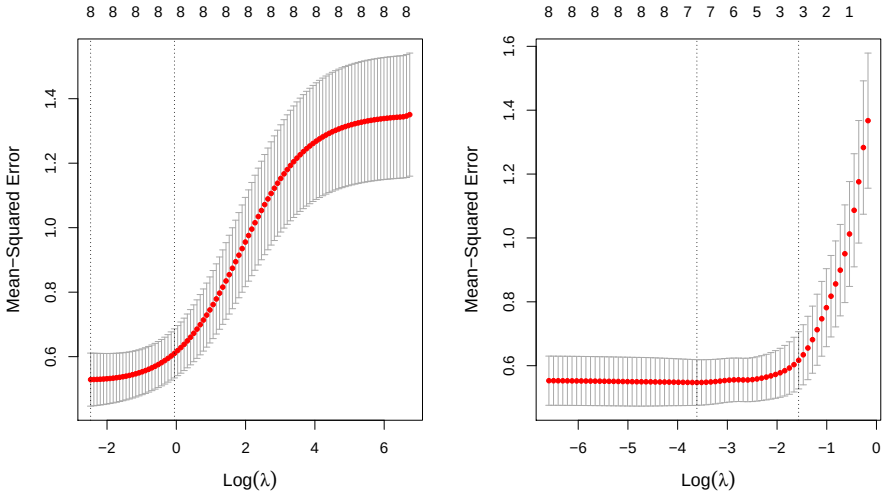
```r
# coefficients in final models
coef(ridge_fit_cv)
```

```
9 x 1 sparse Matrix of class "dgCMatrix"
                       s1
(Intercept)  3.580452e-17
lcavol       3.118804e-01
lweight      1.805362e-01
age         -1.902735e-02
lbph         7.241980e-02
svi          1.873329e-01
lcp          1.053539e-01
gleason      6.058316e-02
pgg45        7.376018e-02
```

```
coef(lasso_fit_cv)
```

```
9 x 1 sparse Matrix of class "dgCMatrix"
                      s1
(Intercept) 5.653623e-17
lcavol      5.286355e-01
lweight     1.201279e-01
age         .
lbph        .
svi         1.400748e-01
lcp         .
gleason     .
pgg45       .
```





1.4 Going Beyond Linearity

1.4.1 Introduction

Thus far in these notes, we have assumed that $f(x) = \mathbb{E}[Y|X = x]$, the true regression function, is a linear function in x. In practice, however, this is often unrealistic. Indeed, $f(x)$ will often be nonlinear and nonadditive in x. In this section, we will introduce several methods for going beyond linearity.

1.4.2 Basis Functions

The basic idea is to transform or augment the inputs X with additional variables which are transformations of X, and then use linear models on this new space of derived input features. Let $h_m(X) : \mathbb{R}^p \to \mathbb{R}$, denote a sequence of transformations, for $m = 1, \ldots, M$. We emphasise that these basis functions are fixed and known (i.e., we choose these ahead of time). Then, for $x \in \mathbb{R}^p$, instead of a linear model we will use

$$f(X) = \sum_{m=1}^{M} \beta_m h_m(X). \tag{1.14}$$

This represents a *linear basis expansion* in X. The nice thing about this approach is that, once we have chosen the functions h_m, the model is linear in these variables. We can thus all of our existing inference tools for linear models.

Some simple and widely used examples of the transformations h_m are as follows.

(i) $h_m(X) = X_m$, for $m = 1, \ldots, p$. This recovers the original linear model.

(ii) $h_m(X) = X_j^2$ or $X_j X_k$, for some $j, k \in 1, \ldots, p$. This allows us to model second-order effects.

(iii) $h_m(X) = \log(X_j), \sqrt{X_j}, \ldots$. This permits nonlinear transformations of single inputs. More generally, we could use non-linear transformations of multiple inputs (or all of the inputs).

(iv) $h_m(X) = 1(L_m \leq X_j < U_m)$, an indicator function for some interval $[L_m, U_m)$. This allows us to break up the range of x_j into nonoverlapping regions, and thus fit a model with piecewise constant contribution for X_j.

In some cases, the problem of interest will suggest a particular choice of basis functions h_m, e.g., power functions or logarithms. In most cases, however, we will use the basis expansion as a tool to achieve greater flexibility in our representation of $f(X)$. In the following sections, we consider two concrete examples: *piecewise polynomials* and *splines*.

1.4.3 Piecewise Polynomials and Regression Splines

We assume for now that the inputs are one-dimensional. A piecewise polynomial function $f(X)$ is obtained by dividing the domain of X into continuous intervals, and representing f by a separate polynomial on each interval.

The simplest case is when the polynomial on each region is degree 0, that is, a constant. This is sometimes referred to as *histogram regression*. For example, if we were to specify three basis functions, we would have

$$h_1(x) = I(x < \xi_1), \qquad h_2(x) = I(\xi_1 \leq x < \xi_2), \qquad h_3(x) = I(\xi_2 \leq x),$$

where $\xi_1 < \xi_2$ are the points of discontinuity, referred to as the *knots*. In this simple example, since the basis functions are positive over disjoint regions, the least squares estimate of the model $Y = \sum_{m=1}^{3} \beta_m h_m(X) + \varepsilon$ is given by $\hat{\beta}_m = \bar{Y}_m$, the mean of the response Y in the mth region. We plot an example of a piecewise constant fit in Figure 1.4 (left).

We can, of course, go beyond piecewise constant functions, and consider, say, piecewise linear functions. In the simple example considered above, where we specified two knots $\xi_1 < \xi_2$, this would mean using an additional three basis functions, namely $h_{m+3}(x) = h_m(x)x$ for $m = 1, 2, 3$. We plot an example of a piecewise linear fit in Figure 1.4 (right).

In general, it is typical to require that the fitted function f has some level of smoothness (e.g., continuity) at each knot. This is shown in Figure 1.5 (left), where we plot an example of a piecewise linear fit under with the constraint f is continuous at each of the knots. In general, these continuity constraints imply a set of linear constraints on the model parameters. For example, for the piecewise linear basis with two knots $\xi_1 < \xi_2$ considered above, the constraint that $f(\xi_1^-) = f(\xi_1^+)$ implies that $\beta_1 + \xi_1\beta_4 = \beta_2 + \xi_1\beta_5$, while the constraint that $f(\xi_2^-) = f(\xi_2^+)$ implies that $\beta_2 + \xi_2\beta_5 = \beta_3 + \xi_2\beta_6$. Thus, we now only have four free parameters, as opposed to six in the unconstrained case.

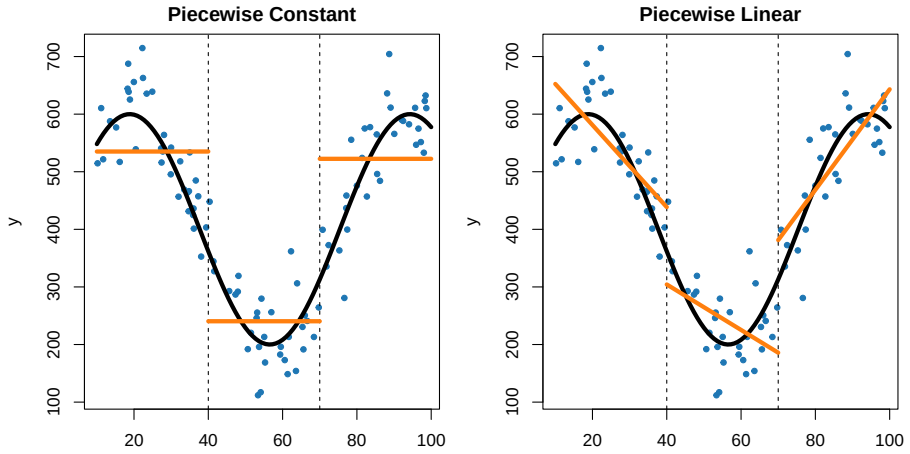


Figure 1.4: **Estimation using piecewise constant (left) and piecewise linear (right) basis functions.** The two panels show the piecewise constant (left) and piecewise linear (right) function fits to some artifical data. The artifical data (blue) were generated by adding Gaussian noise to the true curve (black).

In many cases, we may prefer smoother functions, which can be achieved by further increasing the order of the local polynomial (e.g., quadratic, cubic, etc.). In the case of a piecewise cubic function, which furthermore has continuous *first* and *second derivatives* at each knot, the function is known as a *cubic spline*. It is rare to go beyond cubic functions; it is said that cubic splines are the lowest-order spline for which the knot-discontinuity is not visible to the human eye.

There are many equivalent ways to represent cubic splines, using different choices of basis functions. The most direct way to represent a cubic spline is to start off with a basis for a cubic polynomial - namely, 1, x, x^2, and x^3 - and then include one *truncated power basis* function per knot. In terms of the knot ξ, a truncated power basis function is defined as

$$h(x,\xi) = (x-\xi)_+^3 = \begin{cases} (x-\xi)^3 & \text{if } x > \xi \\ 0 & \text{otherwise} \end{cases}$$

One can show that adding a basis function of this form to the model for a cubic polynomial will lead to a discontinuity only in the third derivative at ξ; the function will still be continuous, with continuous first and second derivative, at ξ. Returning to our example with two knots at $\xi_1 < \xi_2$, it follows from the discussion above that we have the basis

$$\begin{aligned} h_1(x) &= 1, \quad h_3(x) = x^2, \quad h_5(x) = (x-\xi_1)_+^3, \\ h_2(x) &= x, \quad h_4(x) = x^3, \quad h_6(x) = (x-\xi_2)_+^3. \end{aligned}$$

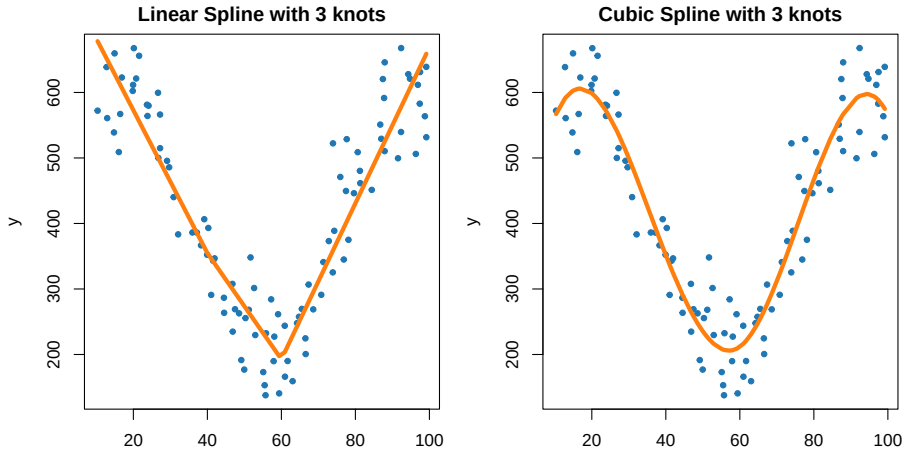


Figure 1.5: **Estimation using a linear spline (left) and a cubic spline (right).** The two panels show the linear spline (left) and cubic spline (right) fitted to the same artificial data used in Figure 1.4.

An example of this cubic spline is shown in Figure 1.5. More generally, in order to fit a cubic spline with k knots $\xi_1 < \xi_2 < \cdots < \xi_K$, we will have a basis given by $h_1(x) = 1$, $h_2(x) = x$, $h_3(x) = x^2$, $h_4(x) = x^3$, $h_{j+4}(x) = h(x, \xi_j)$, $j = 1 \ldots, k$. We are thus required to estimate a total of $m = k + 4$ regression coefficients. These fixed-knot splines are known as *regression splines*.

One issue with polynomial fits is that they can have high variance at the outer range of the predictors, and extrapolation can be very dangerous. These problems are exacerbated with splines. Indeed, the polynomials fit beyond the boundary of the knots behave even more erratically than the corresponding global polynomials in that region. One way to address this is to use a *natural cubic spline*, which includes the additional constraint that the function is *linear* beyond the boundary knots. This additional constraints means that natural splines generally produce more stable estimates at the boundaries.

In general, a natural cubic spline with k knots can be represented using k basis functions (as opposed to $k + 4$ basis functions for the standard cubic spline). In fact, starting from the truncated power series basis described above, it is possible to show that the basis functions for the natural cubic spline are given by

$$h_1(x) = 1, \qquad h_2(x) = x, \qquad h_{j+2}(x) = d_j(x) - d_{k-1}(x)$$

where, for $j = 1, \ldots, k - 2$,

$$d_k(x) = \frac{(x - \xi_j)_+^3 - (x - \xi_k)_+^3}{\xi_k - \xi_j}$$

It is straightforward to check that each of these basis functions has zero second

and third derivative for $x \geq \xi_k$.

In general, the order of the spline, the number of knots, and the locations of the knots, all need to be set by the user. One simple approach is to parameterise a family of splines by the desired number of basis functions (i.e., the degrees of freedom), and have the observations determine the positions of the knots (e.g., using percentiles). For example, specifying $m = 7$ basis functions implies that we will have $k = 3$ knots, since $m = k + 4$. We can then choose these knots to be at the 25^{th}, 50^{th}, and 75^{th} percentiles of the observed data. Another option is to use cross-validation.

1.4.4 Smoothing Splines

In the last section we discussed regression splines. These are created by specifying a set of knots, producing a sequence of basis functions, and then using least squares to estimate the spline coefficients. We now discuss a rather different approach which also produces a spline, but avoids the need select the knots altogether.

In fitting a smooth curve to a set of data, our general goal is to find some function, say $f(x)$, which fits the observed data well. That is, we want to find $f(x)$ such that $\text{RSS} = \sum_{i=1}^{n}(y_i - f(x_i))^2$ is small. There is, of course, a problem with this naive approach. In particular, if we don't include any constraints on $f(x)$, then we can always make the RSS zero by choosing $f(x)$ such that $y_i = f(x_i)$ at each x_i. That is, by choosing a function which *interpolates* the data. Instead, what we would like is a function that makes the RSS small, but that is also *smooth*.

The next question, then, is how we can ensure that f is smooth. There are a number of different ways to achieve this. One way is to consider the following optimisation problem: among all functions with two continuous derivatives, find the function f that minimises the penalised residual sum of squares

$$\text{PRSS}(f, \lambda) = \sum_{i-1}^{N}(y_i - f(x_i))^2 + \lambda \int f''(t)^2 \, \mathrm{d}t, \tag{1.15}$$

where $\lambda \geq 0$ is a fixed *smoothing parameter*. The first term measures closeness to the observed data, while the second term penalises twoo much curvature in the function. The parameter λ controls the trade-off between these two terms. There are two special cases of particular interest:

- If $\lambda = 0$, then f can be any function that exactly interpolates the data. This may be detrimental, since such an f is likely to *overfit* the data.
- If $\lambda = \infty$, then f coincides with the simple least squares fit, since the second derivative must be identically zero.

Between these two extremes, the minimiser of $\text{PRSS}(f, \lambda)$ will vary from very rough to vary smooth. The hope is that $\lambda \in (0, \infty)$ will index an interesting class of functions.

The optimisation problem above is defined on an infinite-dimensional space of functions: the so-called second order Sobolev space. Remarkably, however, it is possible to show that it has an explicit, finite-dimensional, unique minimiser. In fact, this function is a natural cubic spline with knots at the unique values of $x_1, \ldots, x_N$. This function is known as a *smoothing spline*. It may seem, at face value, that this function is still over-parameterised, since there are as many as N knots. It will turn out, however, that this is not the same natural cubic spline that one would obtain using the basis function approach described previously. Instead, the penalty term translates to a penalty on the spline coefficients, which are shrunk some of the way towards the linear fit, with the amount of shrinkage controlled by the parameter λ.

Let's provide a little more detail on this. Since the solution is a natural cubic spline, we can write it as

$$f(x) = \sum_{j=1}^{N} h_j(x)\beta_j,$$

where $\{h_j(x)\}_{j=1}^N$ are a set of basis functions for representing this family of natural cubic splines, and $\{\beta_j\}_{j=1}^N$ are a collection of constants. We can thus rewrite the objective (1.15), now in vector notation, as

$$\text{PRSS}(f, \lambda) = \|\mathbf{y} - \mathbf{N}\beta\|_2^2 + \lambda\beta^\top \mathbf{\Omega}_N \beta,$$

where $\mathbf{N}$ and $\mathbf{\Omega}_N$ are $N \times N$ matrices with entries $\mathbf{N}_{ij} = h_j(x_i)$, and $(\mathbf{\Omega}_N)_{jk} = \int h_j''(t)h_k''(t)\,\mathrm{d}t$, respectively. Using this expression, it is now possible to obtain the solution - try to prove this - as

$$\hat{\beta} = (\mathbf{N}^\top\mathbf{N} + \lambda\mathbf{\Omega}_N)^{-1}\mathbf{N}^\top\mathbf{y}.$$

This solution represents a generalized version of the ridge regression estimator. The fitted smoothing spline is then

$$\hat{f}(x) = \sum_{j=1}^{N} h_j(x)\hat{\beta}_j.$$

In fitting a smoothing spline, there is now no longer any need to select the number or the location of the knots: there will always be a knot at each training observation, $x_1, \ldots, x_N$. We are now faced with another problem, however: we need to choose a value of the smoothing parameter λ. It should come as no surprise that one possible solution to this problem is to use cross-validation. In particular, we can find the value of λ which minimises the cross-validation RSS (see Algorithm 4). Another option is to use a criteria such as the AIC or BIC.

We now discuss some other more intuitive approaches to pre-specifying the amount of smoothing. A smoothing spline with fixed λ is an example of a linear smoother, since the estimated parameters are a linear combination of the y_i.

Let's write $\hat{\mathbf{f}}$ for the vector of fitted values $\hat{f}(x_i)$ at the training predictors: $\hat{\mathbf{f}} = (\hat{f}(x_1), \ldots, \hat{f}(x_N))^\top$. We then have

$$\hat{\mathbf{f}} = \mathbf{N}(\mathbf{N}^\top\mathbf{N} + \lambda\boldsymbol{\Omega}_N)^{-1}\mathbf{N}^\top\mathbf{y} = \mathbf{S}_\lambda\mathbf{y}.$$

The fit is linear in $\mathbf{y}$, and the (finite) linear operator $\mathbf{S}_\lambda := \mathbf{N}(\mathbf{N}^\top\mathbf{N}+\lambda\boldsymbol{\Omega}_N)^{-1}\mathbf{N}^\top$ is referred to as the smoother matrix. One interesting consequence of this linearity is that obtaining the fitted values $\hat{\mathbf{f}}$ from the observations $\mathbf{y}$ does not actually depend on $\mathbf{y}$; $\mathbf{S}_\lambda$ only depends on the covariates x_i and the smoothing parameter λ.

Linear operators also appear in more traditional least squares. Indeed, suppose $\mathbf{N}_\xi$ is a $N \times M$ matrix of M cubic-spline basis functions evaluated at $(x_1, \ldots, x_N)$ with knot sequence ξ, with $M \ll N$. Then we can obtain the vector of fitted values as

$$\hat{\mathbf{f}} = \mathbf{N}_\xi(\mathbf{N}_\xi^\top\mathbf{N}_\xi)^{-1}\mathbf{N}_\xi^\top\mathbf{y} = \mathbf{H}_\xi\mathbf{y}.$$

In this case, the linear operator $\mathbf{H}_\xi := \mathbf{N}_\xi(\mathbf{N}_\xi^\top\mathbf{N}_\xi)^{-1}\mathbf{N}_\xi^\top$ is a projection operator, known as the hat matrix (see our earlier discussion).

There are some important similarities and differences to note between $\mathbf{H}_\xi$ and $\mathbf{S}_\lambda$:

- $\mathbf{H}_\xi$ and $\mathbf{S}_\lambda$ are both symmetric and positive semi-definite.
- $\mathbf{H}_\xi^2$ is idempotent, while $\mathbf{S}_\lambda^2 \preccurlyeq \mathbf{S}_\lambda$. This is a consequence of the shrinking nature of $\mathbf{S}_\lambda$ (see below).
- $\mathbf{H}_\xi$ has rank M, while $\mathbf{S}_\lambda$ has rank N.

In the former case, the dimension of the projection space is given by $M = \mathrm{Tr}(\mathbf{H}_\xi)$. This is also the number of basis functions, and thus the number of parameters involved in the fit. Analogously, we can define the effective degrees-of-freedom of a smoothing spline to be

$$\mathrm{df}(\lambda) = \mathrm{Tr}(\mathbf{S}_\lambda)$$

This definition provides a more intuitive way to parameterise the smoothing spline in a consistent fashion.

1.4.5 Splines and Smoothing Splines in `R`

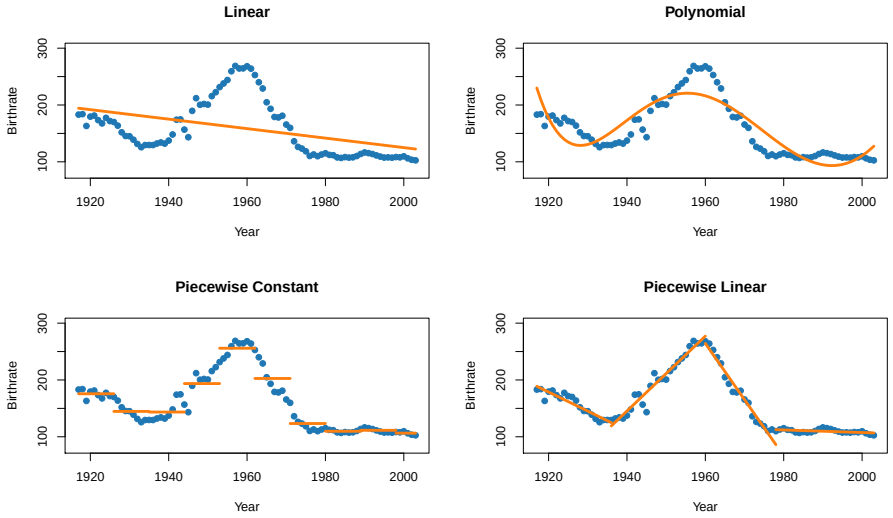
In `R`, we have various options when it comes to fitting (regression) splines and smoothing splines. In particular:

- We can fit splines and natural splines using the `bs()` and `ns()` from the package `splines`.
- We can fit smoothing splines using the `smooth.spline()` function in the `stats` package or the `ss` function in the `npreg` package.

We'll now take a look at how to do this practice, using the US birthrate dataset `birthrate`. This dataset includes the birthrate for all years between 1917 and 2003 (87 observations). We'll begin by fitting some models using some non-linear basis functions.

```
library(stats)
library(splines)

# load data
data = read.csv("data/birthrate.csv")
head(data)

  Year Birthrate
1 1917     183.1
2 1918     183.9
3 1919     163.1
4 1920     179.5
5 1921     181.4
6 1922     173.4

# fit a linear model
model1 = lm(Birthrate ~ Year, data = data)

# fit some non-linear models (3rd, 5th order
# polynomials)
model2_1 <- lm(Birthrate ~ poly(Year, 3), data = data)
model2_2 <- lm(Birthrate ~ poly(Year, 5), data = data)

# fit a piecewise constant model (10 intervals)
knots = 1917 + seq(9, 86, 9)
bounds = c(1917, knots, 2003)
basis = c((data$Year < knots[1]))

for (i in 1:(length(knots) - 1)) basis = cbind(basis,
    (data$Year >= knots[i]) * (data$Year < knots[i +
        1]))
basis = cbind(basis, data$Year >= knots[length(knots)])

model3 <- lm(data$Birthrate ~ . - 1, data = data.frame(basis))

# fit a piecewise linear model (4 intervals)
knots2 = c(1936, 1960, 1978)
bounds2 = c(1917, knots2, 2003)
basis2 = cbind((data$Year < knots2[1]), (data$Year >=
    knots2[1]) * (data$Year < knots2[2]), (data$Year >=
    knots2[2]) * (data$Year < knots2[3]), (data$Year >=
```

```
    knots2[3]), data$Year * (data$Year < knots2[1]),
    data$Year * (data$Year >= knots2[1]) * (data$Year <
        knots2[2]), data$Year * (data$Year >= knots2[2]) *
        (data$Year < knots2[3]), data$Year * (data$Year >=
        knots2[3]))
model4 <- lm(data$Birthrate ~ . - 1, data = data.frame(basis2))

# plots
par(mfrow = c(2, 2))

# linear model
plot(data, pch = 19, col = "#1f77b4", ylim = c(80,
    300), main = "Linear")
lines(data$Year, model1$fitted.values, col = "#ff7f0e",
    lwd = 3, main = "Linear")

# polynomial model
plot(data, pch = 19, col = "#1f77b4", ylim = c(80,
    300), main = "Polynomial")
lines(data$Year, model2_2$fitted.values, lty = 1, col = "#ff7f0e",
    lwd = 3)

# piecewise constant model
plot(data, pch = 19, col = "#1f77b4", ylim = c(80,
    300), main = "Piecewise Constant")
for (k in 1:10) points(c(bounds[k], bounds[k + 1]),
    rep(model3$coefficients[k], 2), type = "l", lty = 1,
    col = "#ff7f0e", lwd = 3)

# piecewise linear model
plot(data, pch = 19, col = "#1f77b4", ylim = c(80,
    300), main = "Piecewise Linear")
for (k in 1:4) points(c(bounds2[k], bounds2[k + 1]),
    model4$coefficients[k] + c(bounds2[k], bounds2[k +
        1]) * model4$coefficients[k + 4], type = "l",
    lty = 1, col = "#ff7f0e", lwd = 3)
```



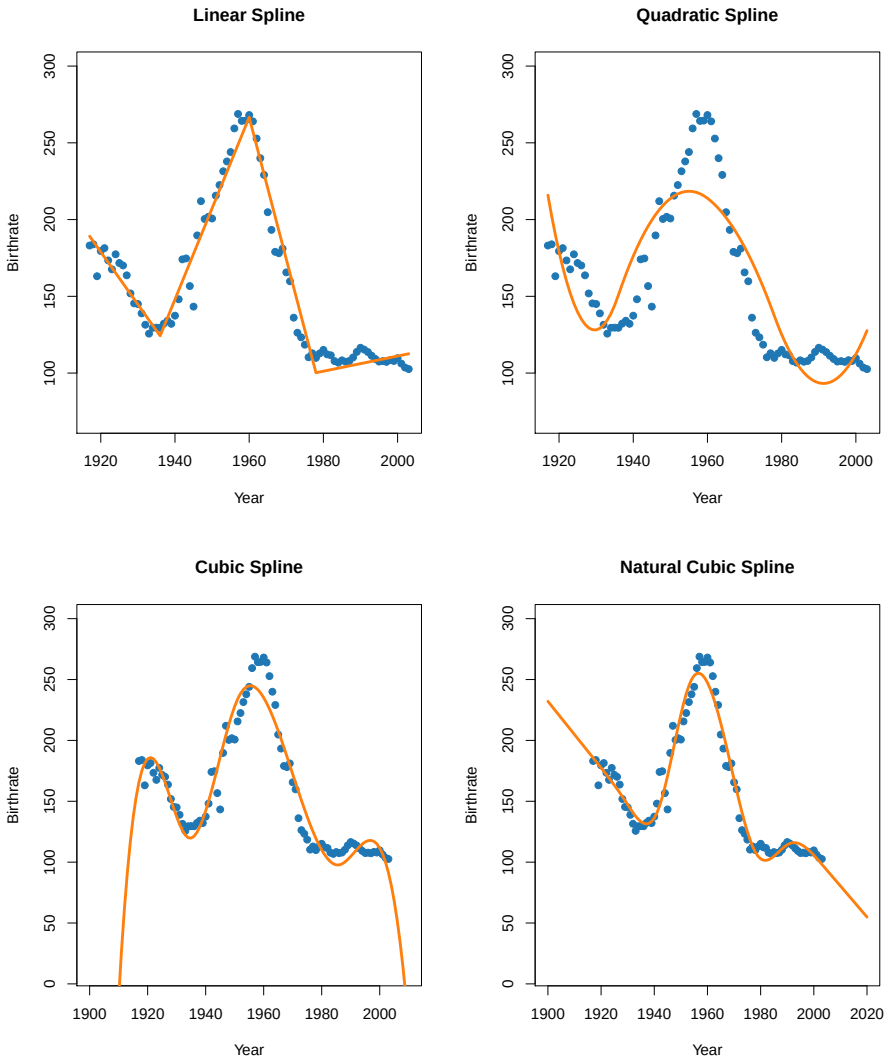
None of these models is particularly satisfactory. The linear and polynomial models don't seem to capture the true relationship between the covariate and the response. Meawnwhile, our piecewise constant and piecewise linear models are discontinuous at the knots. Let's try fitting some splines (linear, quadratic, cubic) to the data instead.

```
## Splines ##

# load packages
library(splines)

# setup knots
knots = c(1936, 1960, 1978)

# linear spline
linear_spline <- lm(Birthrate ~ bs(Year, degree = 1,
    knots = knots), data = data)

# quadratic spline
quad_spline <- lm(Birthrate ~ bs(Year, degree = 2,
    knots = knots), data = data)

# cubic and natural cubic splines we now
# specify degrees-of-freedom rather than
# the knots
cubic_spline <- lm(Birthrate ~ bs(Year, degree = 3,
    df = 7), data = data)
nat_cubic_spline <- lm(Birthrate ~ ns(Year,
```

```
    df = 7), data = data)

# plots
par(mfrow = c(1, 2))
plot(data, pch = 19, col = "#1f77b4", ylim = c(70,
    300))
lines(data$Year, linear_spline$fitted.values,
    lty = 1, col = "#ff7f0e", lwd = 3)
title("Linear Spline")


plot(data, pch = 19, col = "#1f77b4", ylim = c(70,
    300))
lines(data$Year, quad_spline$fitted.values,
    lty = 1, col = "#ff7f0e", lwd = 3)
title("Quadratic Spline")

par(mfrow = c(1, 2))
plot(data, pch = 19, col = "#1f77b4", xlim = c(1900,
    2010), ylim = c(10, 300))
lines(seq(1900, 2020), predict(cubic_spline,
    data.frame(Year = seq(1900, 2020))), lty = 1,
    col = "#ff7f0e", lwd = 3)
```

```
Warning in bs(Year, degree = 3L, knots = c(‘20%‘ = 1934.2, ‘40%‘ =
1951.4, :  some ’x’ values beyond boundary knots may cause ill-conditioned
bases
```

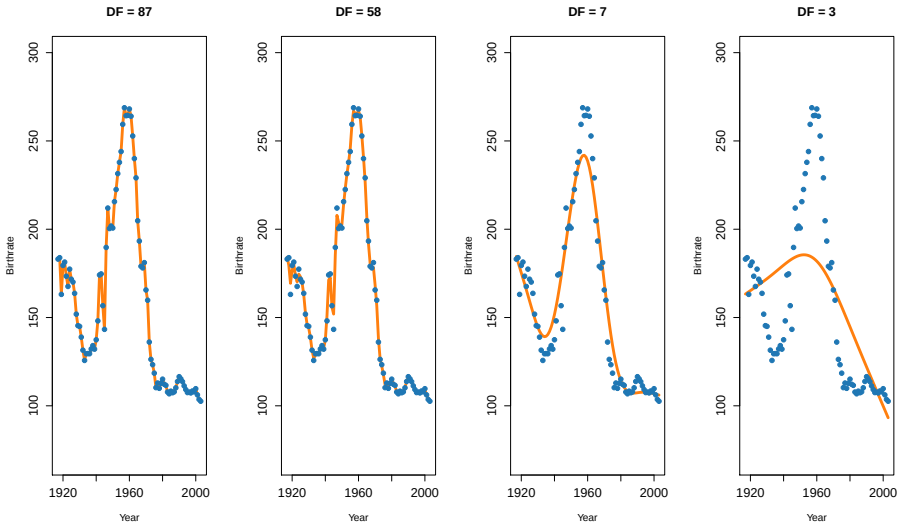
```
title("Cubic Spline")

plot(data, pch = 19, col = "#1f77b4", xlim = c(1900,
    2020), ylim = c(10, 300))
lines(seq(1900, 2020), predict(nat_cubic_spline,
    data.frame(Year = seq(1900, 2020))), lty = 1,
    col = "#ff7f0e", lwd = 3)
title("Natural Cubic Spline")
```



Observe, in particular, how the natural cubic spline stabilises the performance of the standard cubic spline at the boundaries. Finally, we'll fit some smoothing splines. Let's take a look at how the choice of the regularisation parameter λ effects the fit.

```
## load packages
library(stats)

## Smoothing Splines ##
par(mfrow = c(1, 4))
```

```r
# lambda ~ 0
smooth_spline1 <- smooth.spline(data$Year, data$Birthrate,
    all.knots = TRUE, lambda = 1e-14)
plot(smooth_spline1, ylim = c(70, 300), col = "#ff7f0e",
    lwd = 3, type = "l", main = paste("DF =",
        round(smooth_spline1$df)), cex.axis = 1.2,
    xlab = "Year", ylab = "Birthrate")
points(data, pch = 19, col = "#1f77b4", xlim = c(1900,
    2020), ylim = c(10, 300))

# GCV selection
smooth_spline2 <- smooth.spline(data$Year, data$Birthrate,
    all.knots = TRUE, cv = TRUE)
plot(smooth_spline2, ylim = c(70, 300), col = "#ff7f0e",
    lwd = 3, type = "l", main = paste("DF =",
        round(smooth_spline2$df)), cex.axis = 1.2,
    xlab = "Year", ylab = "Birthrate")
points(data, pch = 19, col = "#1f77b4", xlim = c(1900,
    2020), ylim = c(10, 300))

# lambda = 1e-3
smooth_spline3 <- smooth.spline(data$Year, data$Birthrate,
    all.knots = TRUE, lambda = 0.001)
plot(smooth_spline3, ylim = c(70, 300), col = "#ff7f0e",
    lwd = 3, type = "l", main = paste("DF =",
        round(smooth_spline3$df)), cex.axis = 1.2,
    xlab = "Year", ylab = "Birthrate")
points(data, pch = 19, col = "#1f77b4", xlim = c(1900,
    2020), ylim = c(10, 300))

# lambda = 1e-1
smooth_spline4 <- smooth.spline(data$Year, data$Birthrate,
    all.knots = TRUE, lambda = 0.1)
plot(smooth_spline4, ylim = c(70, 300), col = "#ff7f0e",
    lwd = 3, , type = "l", main = paste("DF =",
        round(smooth_spline4$df)), cex.axis = 1.2,
    xlab = "Year", ylab = "Birthrate")
points(data, pch = 19, col = "#1f77b4", xlim = c(1900,
    2020), ylim = c(10, 300))
```



1.4.6 Regularisation Methods and Reproducing Kernel Hilbert Spaces (Non-Examinable)

In this section, we cast smoothing splines into an even more general class of regularisation methods. In particular, we now consider a regularised optimisation problem of the form

$$\min_{f \in \mathcal{H}} \left[\sum_{i=1}^{N} L(y_i, f(x_i)) + \lambda J(f) \right] \tag{1.16}$$

where $L(y, f(x))$ is a loss function, $J(f)$ is a penalty functional, and $\mathcal{H}$ is a space of functions on which this functional is defined. We are particularly interested on an important subclass of problems of this form, in which the optimisation is defined over a space of functions known as a reproducing kernel Hilbert space (RKHS).

We now give a brief and somewhat simplified introduction to this space of functions. Let $\{x_i\}_{i=1}^{m} \in \mathbb{R}^p$. Let $k : \mathbb{R}^p \times \mathbb{R}^p \to \mathbb{R}$ be a continuous, positive-definite, symmetric function, known as the *kernel function*. We can then define the RKHS $\mathcal{H}_k$ associated with the kernel k as the space of functions

$$\mathcal{H}_k = \overline{\{f = \sum_{i=1}^{m} \alpha_i k(\cdot, x_i) | \alpha_i \in \mathbb{R}, m \in \mathbb{N}, x_i \in \mathbb{R}^p\}}$$

Given two functions $f = \sum_{i=1}^{m} \alpha_i k(\cdot, x_i)$ and $g = \sum_{j=1}^{p} \beta_j k(\cdot, x_j)$ in $\mathcal{H}_k$, we can define an inner product on this space by

$$\langle f, g \rangle_{\mathcal{H}_k} = \langle \sum_{i=1}^{m} \alpha_i k(\cdot, x_i), \sum_{j=1}^{m} \beta_j k(\cdot, x_j) \rangle_{\mathcal{H}_k} = \sum_{i=1}^{m} \sum_{j=1}^{p} \alpha_i \beta_j k(x_i, x_j).$$

Under some technical conditions, a result known as Mercer's Theorem states that the kernel function k can be expanded in terms of a set of orthonormal eigenfunctions as

$$k(x,y) = \sum_{i=1}^{\infty} \gamma_i \psi_i(x) \psi_i(y)$$

with $\gamma_i \geq 0$, and $\sum_{i=1}^{\infty} \gamma_i^2 < \infty$. Using these eigenfunctions, there is in fact another way to characterise the RKHS, given by

$$\mathcal{H}_k = \{f = \sum_{i=1}^{\infty} c_i \psi_i | c_i \in \mathbb{R}, \sum_{i=1}^{\infty} \frac{c_i^2}{\gamma_i} < \infty\}.$$

Given two functions $f = \sum_{i=1}^{\infty} c_i \psi_i$ and $g = \sum_{i=1}^{\infty} d_i \psi_i$ in $\mathcal{H}_k$, the inner product is now defined according to

$$\langle f, g \rangle_{\mathcal{H}_k} = \langle \sum_{i=1}^{\infty} c_i \psi_i(x), \sum_{i=1}^{\infty} d_i \psi_i(x) \rangle_{\mathcal{H}_k} = \sum_{i=1}^{\infty} \frac{c_i d_i}{\gamma_i}.$$

Another useful way to express the kernel function is in terms of the so-called *feature map* $x \mapsto \boldsymbol{\phi}(x)$, which takes points in the input space to points in the so-called *feature space*. This is defined as a possibly infinite-dimensional vector with elements given by

$$\boldsymbol{\phi}(x) = (\phi_1(x), \phi_2(x), \dots)^\top := (\sqrt{\gamma_1}\psi_1(x), \sqrt{\gamma_2}\psi_2(x), \dots)^\top$$

where $\{\psi_i\}_{i=1}^{\infty}$ and $\{\gamma_i\}_{i=1}^{\infty}$ are the eigenfunctions and eigenvalues of the kernel. From our previous expression for the kernel function, it is clear that we can now write the kernel as

$$k(x,y) = \boldsymbol{\phi}(x)^\top \boldsymbol{\phi}(y).$$

Thus, the kernel between two points corresponds to the inner product of these points, once they have been mapped to the feature space.

The defining feature of the RKHS is the so-called *reproducing property*, which states that evaluating the function $f \in \mathcal{H}_k$ at a point x_i is the same as calculating the inner product between f and $k_{x_i} = k(\cdot, x_i)$ in $\mathcal{H}_k$. In fact, this property can be used to define the RKHS. In particular, we have that

$$f(x_i) = \langle k(\cdot, x_i), f \rangle_{\mathcal{H}_k}$$

It is easily checked that this property holds, using either characterisation of $\mathcal{H}_k$ given above. By setting $f = k_{x_j} = k(\cdot, x_j)$, we obtain a particularly special case of this equation:

$$k(x_i, x_j) = \langle k(\cdot, x_i), k(\cdot, x_j) \rangle_{\mathcal{H}_k}.$$

Thus, in some sense, computing inner products is 'done by the kernel'. Let us now turn our attention back to the regularised optimisation problem in (1.16), with $\mathcal{H}$ given by the RKHS $\mathcal{H}_k$. Remarkably, in this case, for a suitable choice of the penalty functional J, it is possible to show that this problem admits an explicit, finite-dimensional solution. This is the subject of the following famous result, known as the *representer theorem*.

Theorem 5 (Representer Theorem)**.** *Consider the optimisation problem in* (1.16)*. Let* $\mathcal{J}(f) = \Omega(||f||^2_{\mathcal{H}_k})$*, for some strictly increasing function* $\Omega : [0, \infty] \to \mathbb{R}$*. Then any optimal solution must take the form*

$$\hat{f} = \sum_{j=1}^{N} \alpha_j k(\cdot, x_j).$$

Proof. We can write any $f \in \mathcal{H}_k$ in the form

$$f = \sum_{j=1}^{N} \alpha_j k(\cdot, x_j) + f_\perp$$

where $f_\perp = V^\perp$, with $V = \overline{\text{span}(k(\cdot, x_j), j = 1, \ldots, N\}}$. We will show that $f_\perp = 0$. By the reproducing property of the RKHS, we have that, for all $i = 1, \ldots, N$,

$$\begin{aligned} f(x_i) &= \langle k(\cdot, x_i), f \rangle_{\mathcal{H}_k} \\ &= \sum_{j=1}^{N} \alpha_j k(x_i, x_j) + \langle k(\cdot, x_i), f_\perp \rangle_{\mathcal{H}_k}. \end{aligned}$$

Since $k(\cdot, x_i) \in V$, and $f_\perp \in V^\perp$, their inner product $\langle k(\cdot, x_i), f_\perp \rangle_{\mathcal{H}_k}$ is zero. We thus have that

$$f(x_i) = \sum_{j=1}^{N} \alpha_j k(x_i, x_j).$$

Thus, for any observation x_i in the training data, evaluating $f(x_i)$ is identical when we use this finite representation. Thus, in particular, the value of $\sum_{i=1}^{N} L(y_i, f(x_i))$ in (1.16) is the same when we use the finite representation.

It remains to show that the finite representation minimises the penalty term $\mathcal{J}(f)$ in (1.16), among all possible functions $f \in \mathcal{H}_k$. Observe that

$$\begin{aligned} \Omega(||f||^2_{\mathcal{H}_k}) &= \Omega(||\sum_{i=1}^{N} \alpha_i k(\cdot, x_i) + f_\perp||^2_{\mathcal{H}_k}) \\ &= \Omega(||\sum_{i=1}^{N} \alpha_i k(\cdot, x_i)||^2_{\mathcal{H}_k} + ||f_\perp||^2_{\mathcal{H}_k}) \\ &\geq \Omega(||\sum_{i=1}^{N} \alpha_i k(\cdot, x_i)||^2_{\mathcal{H}_k}) \end{aligned}$$

where in the second line we have used Pythagoras, and in the final line we have used the monotonicity of Ω. Clearly, we have equality if and only if $||f_\perp||_{\mathcal{H}_k} = 0$ and thus $f_\perp = 0$. This completes the proof. □

Using this result, we now have the ability to fit a much more general class of *kernel ridge regression* models. In particular, suppose we now seek the solution of

$$\hat{f} = \arg\min_{f \in \mathcal{H}_k} \left[\sum_{i=1}^{N} (y_i - f(x_i))^2 + \lambda ||f||^2_{\mathcal{H}_k} \right], \tag{1.17}$$

where, as before, $\mathcal{H}_k$ denotes the RKHS with kernel k. From Theorem 5, we know that the solution of this minimisation problem must take the form

$$\hat{f}(x) = \sum_{j=1}^{N} \hat{\alpha}_j k(x, x_j).$$

Using this expression, it follows, after some algebra, that the optimisation problem in (1.17) can equivalently be formulated as

$$\hat{\boldsymbol{\alpha}} = \arg\min_{\boldsymbol{\alpha}} \left[||y - \mathbf{K}\boldsymbol{\alpha}||^2 + \lambda \boldsymbol{\alpha}^\top \mathbf{K} \boldsymbol{\alpha} \right]. \tag{1.18}$$

where $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_N)^\top$, and $\mathbf{K}$ is the $N \times N$ matrix with elements $[\mathbf{K}]_{ij} = k(x_i, x_j)$. This phenomenon, wherein the infinite-dimensional optimisation in (1.17) is reduced to the finite-dimensional optimisation in (1.18), is sometimes referred to as the *kernel property*.

Proposition 6. *The coefficients of the kernel ridge regression estimator in* (**??**) *are given by*

$$\hat{\boldsymbol{\alpha}} = (\mathbf{K} + \lambda \mathbf{I}_N)^{-1} \mathbf{y}. \tag{1.19}$$

Proof. Similar to the proof of the ridge regression estimator. Take derivatives of (1.18), set equal to zero, and check the second derivatives. □

We conclude this section by taking a look at a couple of examples of the kernel ridge regression estimator in practice.

- **Linear Ridge Regression**. We'll start by showing that, for a certain choice of kernel, we can recover the original (linear) ridge regression estimator discussed in Section 1.3.2. In particular, let us consider the kernel

  $$k(x_i, x_j) = x_i^\top x_j = \sum_{k=1}^{p} x_{ik} x_{jk},$$

  with $\mathbf{K} = \mathbf{X}\mathbf{X}^\top$. This is simply the kernel function associated with the linear feature map on $\mathbb{R}^p$, viz

  $$\boldsymbol{\phi}(x_i) = x_i = (x_{i1}, \ldots, x_{ip}).$$

  Using our expression for the kernel, we can now compute the kernel ridge regression coefficients as

  $$\hat{\boldsymbol{\alpha}} = (\mathbf{X}\mathbf{X}^\top + \lambda \mathbf{I}_N)^{-1} \mathbf{y},$$

with the fitted values given by

$$\hat{\mathbf{y}} = \mathbf{X}\mathbf{X}^\top(\mathbf{X}\mathbf{X}^\top + \lambda \mathbf{I}_N)^{-1}\mathbf{y} \tag{1.20}$$

Currently, this looks a little different to the previous formula we had for the fitted values. However, after some linear algebra, we can recover our previous formula

$$\hat{\mathbf{y}} = \mathbf{X}(\mathbf{X}^\top\mathbf{X} + \lambda \mathbf{I}_p)^{-1}\mathbf{X}^\top\mathbf{y}. \tag{1.21}$$

Observe that computing the fitted values according to (1.20) only requires us to compute $\mathbf{K} = \mathbf{X}\mathbf{X}^\top \in \mathbb{R}^{N\times N}$, while computing them according to (1.21) requires us to compute $\mathbf{X}^\top\mathbf{X} \in \mathbb{R}^{p\times p}$. In the regime $p \gg N$, the first of these is much cheaper than the second.

- **Polynomial Ridge Regression**. Let us now suppose, in the spirit of Section 1.4, that we wish to augment the original set of covariates in the linear ridge regression model with an additional set of nonlinear covariates which model *second order interactions*. That is, we now wish to model

$$f(x_i) = \sum_{m=1}^{1+p+p^2} \beta_m h_m(x_i)$$

where the basis functions $\boldsymbol{h}(x_i) = (h_1(x_i), \dots)^T$ are given by

$$\boldsymbol{h}(x_i) = (1, x_{i1}, \dots, x_{ip}, x_{i1}x_{i1}, \dots, x_{i1}x_{ip}, \dots, x_{ip}x_{i1}, \dots, x_{ip}x_{ip})^T.$$

We could, of course, perform standard ridge regression on this expanded set of covariates to obtain the coefficients $\hat{\beta}^{\mathrm{OLS}}$. If p is large, however, then this would be a very costly computation.

Instead, suppose we consider the kernel function defined according to

$$k(x_i, x_j) = (1 + x_i^\top x_j)^2 = (1 + \sum_{k=1}^{p} x_{ik}x_{jk})^2.$$

Expanding this equation, it is straightforward to show that this kernel can be written as $k(x_i) = \phi(x_i)^T\phi(x_j)$, where $\boldsymbol{\phi}$ is the feature map given by

$$\boldsymbol{\phi}(x_i) = (1, \sqrt{2}x_{i1}, \dots, \sqrt{2}x_{ip}, x_{i1}x_{i1}, \dots, x_{i1}x_{ip}, \dots, x_{ip}x_{i1}, \dots, x_{ip}x_{ip})^T.$$

Thus, if we define $\mathbf{K}_{ij} = (1 + x_i^\top x_j)^2$, for $i, j = 1, \dots, N$, and substitute this choice of $\mathbf{K}$ into our formula for the kernel ridge regression estimator (1.19), the resulting fitted values would be the same as if we had performed ridge regression on the augmented set of variables. Computing this new matrix $\mathbf{K}$ only requires a total of $O(N^2p)$ operations, as opposed to $O(N^2p^2 + n^3)$ if we had directly performed ridge regression.

1.5 Generalised Linear Models and Generalised Additive Models

1.5.1 Introduction

The linear models we have described so far are useful when the covariates and response take real values. However, in many practical applications our data may take discrete values, as is the case for classification problems, when the response takes one of a fixed number of labels (e.g. 'spam' versus 'not spam'), or when the response are counts (e.g., the number of infections in a given region).

1.5.2 Generalised Linear Models

In many situations, it is useful to retain a linear structure for the covariates, while permitting a more general form for the response. For example, we may wish to relax the following assumptions.

(i) The conditional distribution of the response Y, given the predictors $X_1, \ldots, X_p$, is normal.

(ii) The conditional mean of the response Y, given the predictors $X_1, \ldots, X_p$, is equal to a linear combination of the covariates.

In *generalised lienar models*, we relax both of these assumptions. In particular, the response is now allowed to originate from any distribution belonging to the *exponential family*. Meanwhile, the linear predictor and the mean response are related by a so-called *link function*. Let's provide a little more detail on the second of these points. Suppose we write $\mu(x) : \mathbb{R}^p \to \mathbb{R}$ for the *regression function* or conditional mean function, given by

$$\mu(x) := \mathbb{E}[Y|X = x].$$

Similar to before, we are interested in relating this conditional mean to a linear combination of the covariates. We will now assume that there exists a monotone and differentiable *link function* g, such that

$$g(\mu(x)) = x^\top \beta \iff \mu(x) = g^{-1}(x^\top \beta).$$

Some important examples of link functions are the following:

- For the standard linear model, we simply use the identity link function, $g(\mu) = \mu$. In this case, we recover the familiar expression

$$\mathbb{E}[Y|X = x] = x^\top \beta.$$

- For two-class classification, we are in a situation when the response $Y \in \{0, 1\}$. In a *logistic regression* model, we relate the mean of the binary

response $\mu(x) = \mathbb{P}(Y = 1|X = x)$ to a linear combination of the predictors using the *logit* link function:

$$g(\mu) = \text{logit}(\mu) := \log\left(\frac{\mu}{1-\mu}\right),$$

Thus, in particular, we have that

$$\log\left(\frac{\mu(x)}{1-\mu(x)}\right) = x^\top\beta \Leftrightarrow \mu(x) = \frac{\exp(x^\top\beta)}{1+\exp(x^\top\beta)}.$$

Another common choice which leads to *probit regression* is to use the link function $g(\mu) = \text{probit}(\mu) := \Phi^{-1}(\mu)$, where Φ is the standard Gaussian cumulative distribution function.

- For Poisson count data, it is standard to take the log link function, $g(\mu) = \log(\mu)$, so
$$\log(\mu(x)) = x^\top\beta \Leftrightarrow \mu(x) = \exp(x^\top\beta).$$

1.5.3 Generalised Additive Models

By relaxing the linear structure of the covariates, we obtain an even more general class of models known as *generalised additive models*, which take the form

$$g(\mu(x)) = \alpha + f_1(x_1) + f_2(x_2) + \cdots + f_p(x_p).$$

where the f_j's are a set of *unspecified* smooth functions, each belonging to some larger class of possible functions, and we have now explicitly included an intercept term. Previously, in Section 1.4.3, we pre-specified the form of the f_j's. We could thus directly fit the model using (penalised) least squares. Instead, we will now fit each function using an algorithm whose basic component is the so-called *scatterplot smoother*, an example of which is the cubic smoothing spline.

For simplicity, let us suppose that we use the identity link function $g(\mu) = \mu$. The additive model is then given by

$$Y = \alpha + \sum_{i=1}^{p} f_j(X_j) + \varepsilon,$$

where the error term ε is assumed to be mean zero. Suppose we are given some observations $(x_i, y_i)_{i=1}^N$. Then one way to fit the $f_j's$ and intercept α is using a *penalized residual sum of squares*, as we have seen before in (1.15):

$$\text{PRSS}(\alpha, f_1, \ldots, f_p) := \sum_{i=1}^{N}\left(y_i - \alpha - \sum_{j=1}^{p} f_j(x_{ij})\right)^2 + \sum_{j=1}^{p}\lambda_j \int f_j''(t_j)^2\,\mathrm{d}t_j, \tag{1.22}$$

for some choice of tuning parameters $\lambda_j \geq 0$. It can be shown that the minimizer of (1.22) is an *additive cubic spline model*. In particular, each f_j is a cubic spline in the component x_j, with knots at the unique values of the x_{ij}, $i \in [N]$.

Currently, however, without additional restrictions on the model, the intercept α is not identifiable. That is to say, if we added a constant shift to α and subtracted the same constant from the f_j, we would have the same loss. The typical convention is thus to enforce $\sum_{i=1}^{N} f_j(x_{ij}) = 0$, for each $j \in [p]$. In other words, the functions average zero over the data. In this case, the fitted intercept is simply the mean, $\hat{\alpha} = \sum y_i/N$.

In practice, we can fit the f_j's from the data using an iterative procedure known as *backfitting*. We begin by initialising $\hat{\alpha} = \sum y_i/N$, which subsequently never changes. We then apply a cubic smoothing spline $\mathcal{S}_j$ (see Section 1.4.4) to the targets $y_i - \hat{\alpha} - \sum_{k \neq j} \hat{f}_k(x_{ik})$, as a function of the x_{ij}, to obtain a new estimate $\hat{f}_j$. This is done for each predictor in turn, using the most up to date estimates of the other functions. This process is continued until the estimates stabilise. This procedure is described in Algorithm 7.

Algorithm 7: Backfitting algorithm for additive models

---

1. Initialize $\hat{\alpha} = \frac{1}{N} \sum_{i=1}^{N} y_i$, and set each $\hat{f}_j \equiv 0$, for each $j \in [p]$.
2. Cycle iteratively through $j = 1, 2, \ldots, p, 1, 2, \ldots, p, \ldots$, setting each time

$$\hat{f}_j \leftarrow \mathcal{S}_j \left[\left\{ y_i - \hat{\alpha} - \sum_{k \neq j} \hat{f}_k(x_{ik}) \right\}_{i=1}^{N} \right],$$

$$\hat{f}_j \leftarrow \hat{f}_j - \frac{1}{N} \sum_{i=1}^{N} \hat{f}_j(x_{ij}),$$

until the functions $\hat{f}_j$ change less than a prespecified threshold.

1.5.4 GLMs and GAMs in R

In R, we have several options for fitting GLMs and GAMs. These include the older `gam` package and the more recent `mgcv` package. We'll now take a look at how to do this in practice, fitting some GAMs to the `mcycle` dataset in the `MASS` package, and the `mpg` dataset in the `gamair` package.

```
## GLMs and GAMs ##

# load gam library
library("mgcv")
```

```r
# load data
data(mcycle, package = "MASS")
# head(mcycle)

# fit linear model
lm1 = lm(accel ~ times, data = mcycle)
summary(lm1)
```

```
Call:
lm(formula = accel ~ times, data = mcycle)

Residuals:
     Min       1Q   Median       3Q      Max
-104.114  -25.926    4.582   36.163   94.197

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  -53.008      8.713  -6.084  1.2e-08 ***
times          1.091      0.307   3.552 0.000532 ***
---
Signif. codes:
0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 46.33 on 131 degrees of freedom
Multiple R-squared:  0.08785,Adjusted R-squared:  0.08089
F-statistic: 12.62 on 1 and 131 DF,  p-value: 0.0005318
```

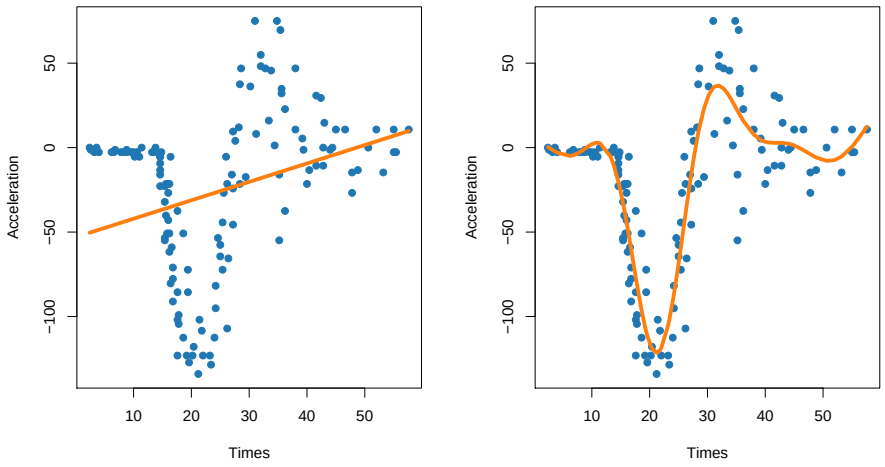
```r
# fit generalised additive model
gam1 = gam(accel ~ s(times), data = mcycle, method = "REML")
summary(gam1)
```

```
Family: gaussian
Link function: identity

Formula:
accel ~ s(times)

Parametric coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  -25.546      1.951  -13.09   <2e-16 ***
---
Signif. codes:
0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
Approximate significance of smooth terms:
           edf Ref.df    F p-value
s(times) 8.625  8.958 53.4  <2e-16 ***
---
Signif. codes:
0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

R-sq.(adj) =  0.783   Deviance explained = 79.7%
-REML = 616.14  Scale est. = 506.35    n = 133
```

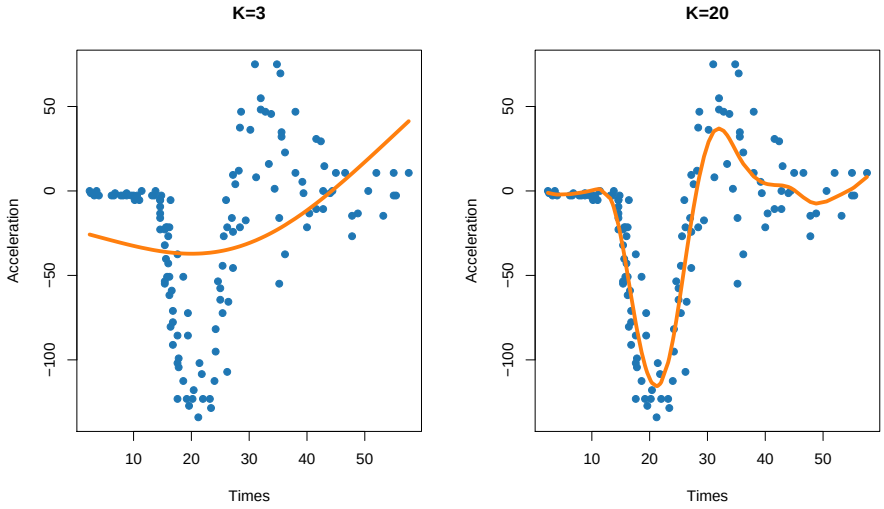
```
par(mfrow = c(1, 2))
plot(mcycle$times, mcycle$accel, pch = 19, col = "#1f77b4",
    xlab = "Times", ylab = "Acceleration")
lines(mcycle$times, lm1$fitted.values, lwd = 4, col = "#ff7f0e")

plot(mcycle$times, mcycle$accel, pch = 19, col = "#1f77b4",
    xlab = "Times", ylab = "Acceleration")
lines(mcycle$times, gam1$fitted.values, lwd = 4, col = "#ff7f0e")
coef(gam1)
```

```
(Intercept)  s(times).1  s(times).2  s(times).3  s(times).4
 -25.545865  -63.799013   41.202002 -109.827071  -23.305431
 s(times).5  s(times).6  s(times).7  s(times).8  s(times).9
  35.132376   90.759312   -8.721425 -106.596311   17.592814
```

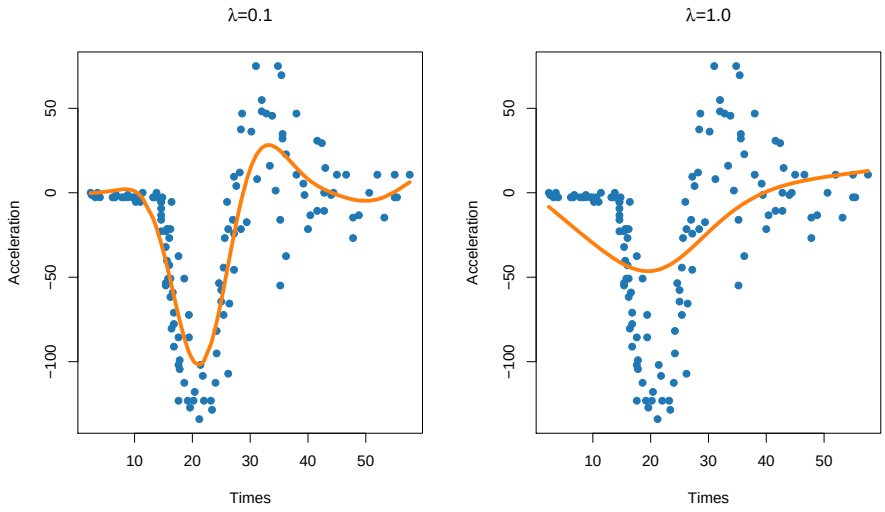



```
# vary the number of basis functions
par(mfrow = c(1, 2))
gam2 = gam(accel ~ s(times, k = 3), data = mcycle)
```

```r
gam3 = gam(accel ~ s(times, k = 20), data = mcycle)
plot(mcycle$times, mcycle$accel, pch = 19, col = "#1f77b4",
    xlab = "Times", ylab = "Acceleration", main = "K=3")
lines(mcycle$times, gam2$fitted.values, pch = 19, lwd = 4,
    col = "#ff7f0e")
plot(mcycle$times, mcycle$accel, pch = 19, col = "#1f77b4",
    xlab = "Times", ylab = "Acceleration", main = "K=20")
lines(mcycle$times, gam3$fitted.values, pch = 19, lwd = 4,
    col = "#ff7f0e")
```




```r
# vary the smoothing
par(mfrow = c(1, 2))
gam4 = gam(accel ~ s(times, sp = 0.01), data = mcycle)
gam5 = gam(accel ~ s(times, sp = 1), data = mcycle)
plot(mcycle$times, mcycle$accel, pch = 19,
    col = "#1f77b4", xlab = "Times", ylab = "Acceleration",
    main = expression(paste(lambda, "=0.1")))
lines(mcycle$times, gam4$fitted.values, pch = 19,
    lwd = 4, col = "#ff7f0e")
plot(mcycle$times, mcycle$accel, pch = 19,
    col = "#1f77b4", xlab = "Times", ylab = "Acceleration",
    main = expression(paste(lambda, "=1.0")))
lines(mcycle$times, gam5$fitted.values, pch = 19,
    lwd = 4, col = "#ff7f0e")
```



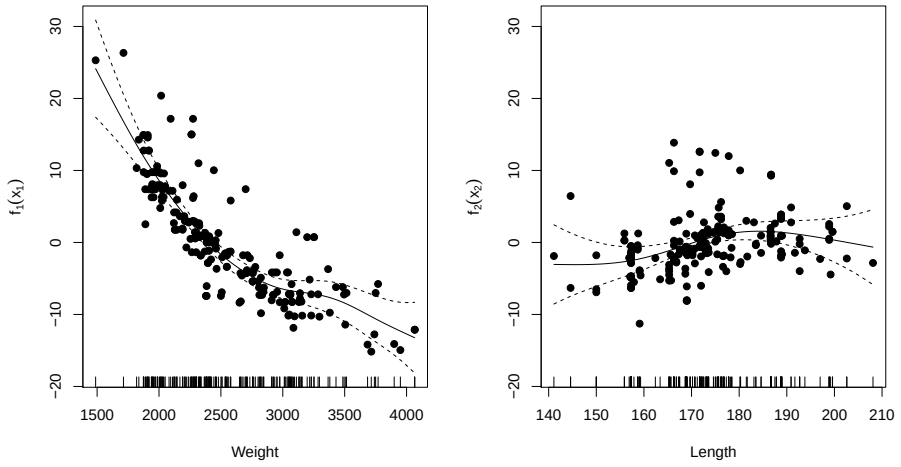
Thus far, the GAMs we've fitted at have only included one covariate. Including additional covariates, whether continuous or categorical, is very straightforward, as we now illustrate.

```
# load library library(gamair)

# load data data('mpg', package='gamair')
mpg = read.csv("data/mpg.csv", row.names = 1)
head(mpg)
```

```
  symbol loss        make fuel aspir doors       style
1      3   NA alfa-romero  gas   std   two convertible
2      3   NA alfa-romero  gas   std   two convertible
3      1   NA alfa-romero  gas   std   two   hatchback
4      2  164        audi  gas   std  four       sedan
5      2  164        audi  gas   std  four       sedan
6      2   NA        audi  gas   std   two       sedan
  drive eng.loc   wb length width height weight eng.type
1   rwd   front 88.6  168.8  64.1   48.8   2548     dohc
2   rwd   front 88.6  168.8  64.1   48.8   2548     dohc
3   rwd   front 94.5  171.2  65.5   52.4   2823     ohcv
4   fwd   front 99.8  176.6  66.2   54.3   2337      ohc
5   4wd   front 99.4  176.6  66.4   54.3   2824      ohc
6   fwd   front 99.8  177.3  66.3   53.1   2507      ohc
  cylinders eng.cc fuel.sys bore stroke comp.ratio  hp  rpm
1      four    130     mpfi 3.47   2.68        9.0 111 5000
2      four    130     mpfi 3.47   2.68        9.0 111 5000
3       six    152     mpfi 2.68   3.47        9.0 154 5000
4      four    109     mpfi 3.19   3.40       10.0 102 5500
```

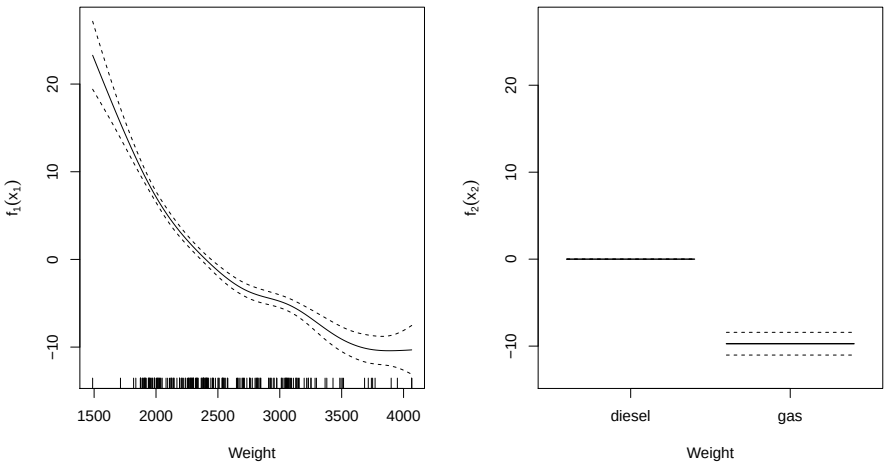
```
5      five    136     mpfi 3.19   3.40        8.0 115 5500
6      five    136     mpfi 3.19   3.40        8.5 110 5500
  city.mpg hw.mpg price
1       21     27 13495
2       21     27 16500
3       19     26 16500
4       24     30 13950
5       18     22 17450
6       19     25 15250
```

```
# fit GAM with two variables, two smoothers
gam1 <- gam(hw.mpg ~ s(weight) + s(length), data = mpg,
    method = "REML")

# plot the model
par(mfrow = c(1, 2))
plot(gam1, all.terms = TRUE, residuals = TRUE, pch = 19,
    select = 1, xlab = "Weight", ylab = expression(f[1](x[1])))
plot(gam1, all.terms = TRUE, residuals = TRUE, pch = 19,
    select = 2, xlab = "Length", ylab = expression(f[2](x[2])))
```

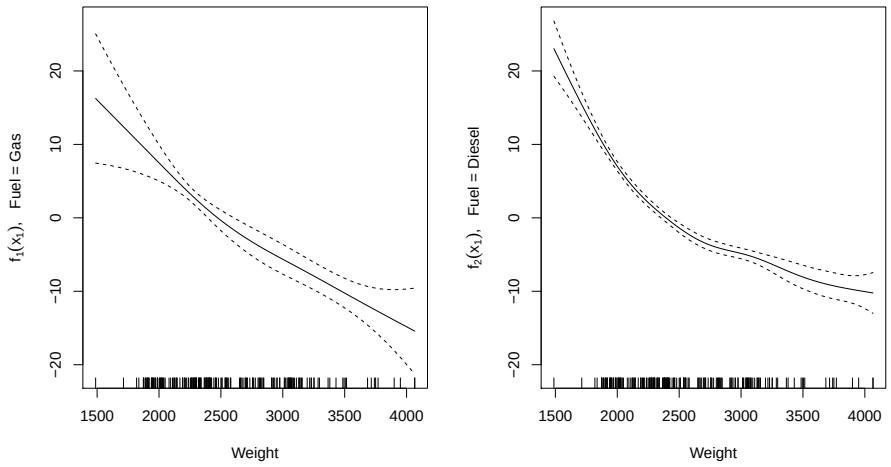



```
# fit GAM with two variables weight is
# continuous & non-linear, fuel is and
# categorical & linear
gam2 <- gam(city.mpg ~ s(weight) + fuel,
    data = mpg, method = "REML")
```

```r
# plot the model
par(mfrow = c(1, 2))
plot(gam2, all.terms = TRUE, pch = 19, select = 1,
    xlab = "Weight", ylab = expression(f[1](x[1])))
termplot(gam2, se = T, pch = 19, xlab = "Weight",
    ylab = expression(f[2](x[2])), col.se = "black",
    col.term = "black")
```




```r
# fit GAM which uses a different smoothing
# function according to the value of a
# categorical variable
gam3 <- gam(city.mpg ~ s(weight, by = fuel) + fuel,
    data = mpg, method = "REML")

# plot the model
par(mfrow = c(1, 2))
plot(gam3, all.terms = TRUE, residuals = TRUE, pch = 19,
    select = 1, xlab = "Weight", ylab = expression(paste(f[1](x[1]),
        ",   Fuel = Gas")))
plot(gam3, all.terms = TRUE, residuals = TRUE, pch = 19,
    select = 2, xlab = "Weight", ylab = expression(paste(f[2](x[1]),
        ",   Fuel = Diesel")))
```



1.6 Summary

1.6.1 Recap

In this section of the course on regression, we began with the familiar linear model, fit using ordinary least squares, and gradually departed from it. We first allowed for more covariates than we had data points, and controlled for this using explicit regularisation (Section 1.2 - Section 1.3). We then allowed for nonlinear transforms of the data (Section 1.4). Finally, we allowed for linear combinations of general nonlinear functions f_j (Section 1.5).

1.6.2 Teaser: Nonlinear Regression

Thus far, all of the models we have studied have ultimately retained the additive (linear) structure of the regression function. This allows for simpler algorithms and interpretable results, since the effect of the covariates is summed in some way. A natural question is whether there are plausible methods which entirely avoid this linearity entirely. This is indeed the case. We will cover such methods in detail later in the course (Section 5 and Section 6).

In Section 5, we will see that *regression trees* provide a thoroughly nonlinear way to classify data or to perform regression, by dividing up the space of covariates and making binary choices along the splits. In Section 6, we will discuss *deep learning*, which models the regression function f using a *neural network* $f(x;\theta)$. This is a highly nonlinear function which possesses many, many parameters $\theta \in \mathbb{R}^m$. Indeed, m could be thousands, millions, or more, and is typically much larger than the number of data points.

Chapter 2

Classification

2.1 Introduction

Where regression problems concern predicting quantitative responses from features, *classification* problems concern categorical responses. An example of classification is image recognition: given a training set of images (an array of pixel values) and labels of the objects within those images, can a statistical model correctly label new images. This is an important feature of autonomous vehicles where the classifier has to determine if an image contains another vehicle, a road sign or, most importantly, a pedestrian.

Figure 2.1: *Is this an image of a kettle or a toaster?* An example of machine learning with image recognition.

A simpler example of using classification for image recognition is handwritten digit recognition. Figure 2.2 shows some suitably standardised sample images of handwritten digits. The aim of the classifier is to correctly predict whether assign a digit label to a new image. Even humans cannot get this classification task 100% correct, and are outperformed by trained classifiers. For example, there are examples where the bottom loop of the digit `"5"` nearly joins up again to appear like the digit `"6"`, or a short horizontal top of the digit `"7"` making it appear like the digit `"1"`.

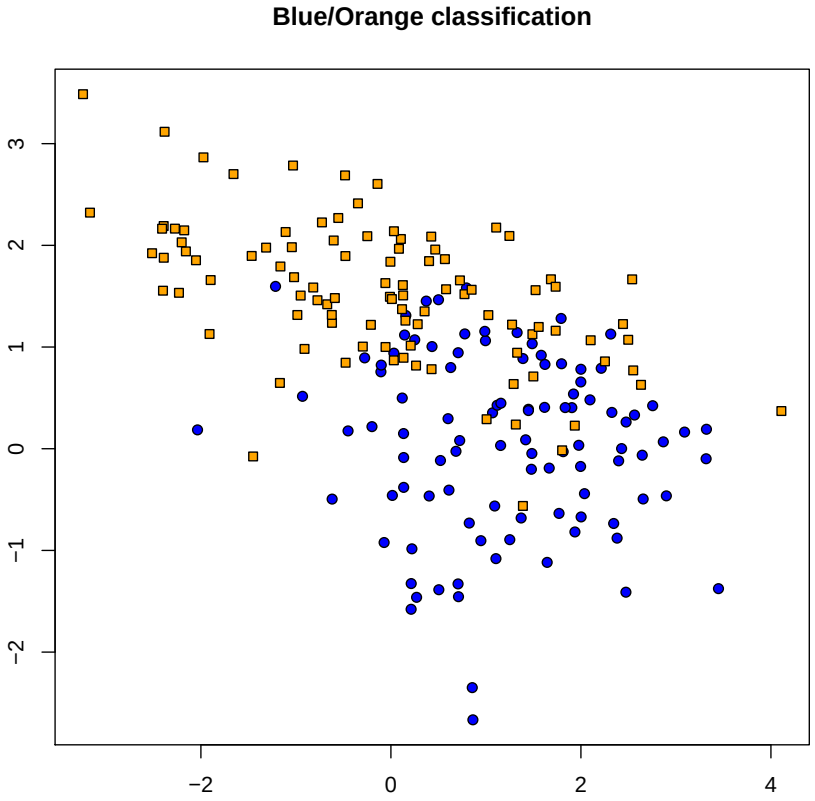
Figure 2.2: Sample digits from the MNIST test datset. Image created by Josef Steppan - Own work, CC BY-SA 4.0, `https://commons.wikimedia.org/w/index.php?curid=64810040`.

One special subclass of classification problems is the *binary classification* problem—one for which there are exactly two class labels. This might be something like a medical diagnosis problem: given some set of clinical measurements from medical tests, can a model correctly determine the presence or nonpresence of a disease or condition in each patient?

As before, we denote the input random variable by X, but we replace the continuous response variable Y with a categorical response variable G taking values in a discrete set $\mathcal{G}$. In the case of handwritten digit recognition, $\mathcal{G} = \{\texttt{"0"}, \texttt{"1"}, \ldots, \texttt{"9"}\}$, and for the medical diagnosis problem the two classes could be $\mathcal{G} = \{\text{deceased}, \text{healthy}\}$.

In this chapter, we will compare different classification algorithms using a synthetic data set consisting of 200 observations of two variables $\mathbf{X} \in \mathbb{R}^{200\times 2}$ representing two different classes $\mathcal{G} = \{0, 1\}$, represented by 100 blue and 100 orange observations. In this binary classification task, the goal is to predict the colour of new observation $x \in \mathbb{R}^2$. This will be displayed using a decision boundary. One side of the boundary will correspond to the set of observations $x \in \mathbb{R}^2$ that are predicted to be blue, the other side, observations that are predicted orange.

```
# Load data
load(file = 'Data/BlueOrange.Rdata')

# Plot data
plot(data$x1, data$x2, bg = c("blue", "orange")[data$y+1],
     pch = c(21, 22)[data$y+1], xlab = "", ylab = "",
     main = "Blue/Orange classification")
```



2.1.1 Linear least squares for classification

First, let us consider what happens if we use some of the machinery that we have already developed for regression to tackle our binary classification problem, namely the linear least squares model. Denote the two class labels by $\mathcal{G} = \{G_0, G_1\}$. To convert this classification problem into one of regression, we map the two class labels to $Y = 0$ and $Y = 1$ respectively, and treat the response as if it were real-valued, giving predictions $\hat{y} \leq 0.5$ the label G_0 and predictions $\hat{y} > 0.5$ the label G_1.

From the previous chapter, we know that the minimiser of the RSS objective (1.3) is given by the OLS estimator:

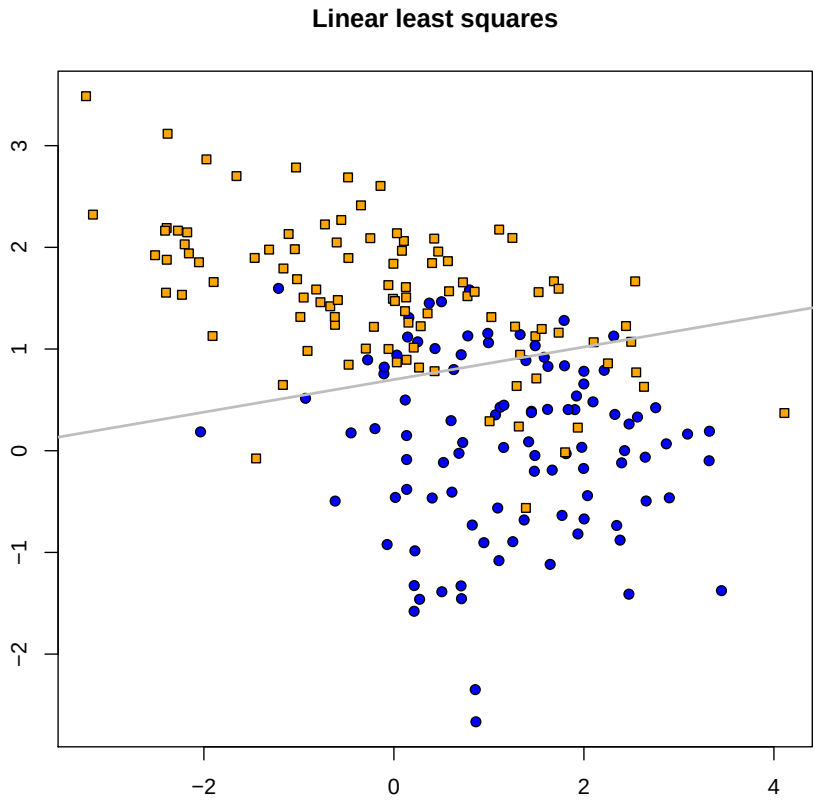
$$\hat{\beta}^{\text{OLS}} = \underset{\beta \in \mathbb{R}^p}{\arg\min}\, \text{RSS}(\beta) = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}. \tag{2.1}$$

Input points $x \in \mathbb{R}^p$ will be assigned a class label according to which side of the *decision boundary* $\{x \in \mathbb{R}^p : x^\top \hat{\beta} = 0.5\}$ they fall. In this particular case, the

decision boundary is a hyperplane, but in general it could be far less regular (it need neither be smooth nor connected, for instance). Points x_i for which the corresponding predicted class label $\hat{g}_i$ is incorrect are described as being *misclassified*.

This is not a standard approach and R does not have a specific function to treat binary classification as linear regression.

```r
# Fit linear regression model
lm.bo = lm(y ~ x1 + x2, data = data)

# Calculate coefficients of linear decision boundary
intercept = (0.5 - coef(lm.bo)[1]) / coef(lm.bo)[3]
gradient = - coef(lm.bo)[2] / coef(lm.bo)[3]

# Plot data and linear decision boundary
plot(data$x1, data$x2, bg = c("blue", "orange")[data$y+1],
     pch = c(21, 22)[data$y+1], xlab = "", ylab = "",
     main = "Linear least squares")
abline(intercept, gradient, col = "grey", lwd = 2)
```



This model suffers from two large problems. Firstly, it is extremely sensitive to outliers: the RSS objective gives undue weight to observations far from the decision boundary. Secondly, it generalises poorly to the multiple class case.

Although it will feature in a number of models that we consider in this course, the constraint of a linear classification boundary is somewhat severe, and there are certain kinds of data for which such models will perform very poorly. We now examine a very different classification model that features an extremely flexible decision boundary.

2.1.2 k-nearest neighbours

Perhaps the simplest approach to the classification problem is to predict a class label for a point x by finding the closest data point x^*,

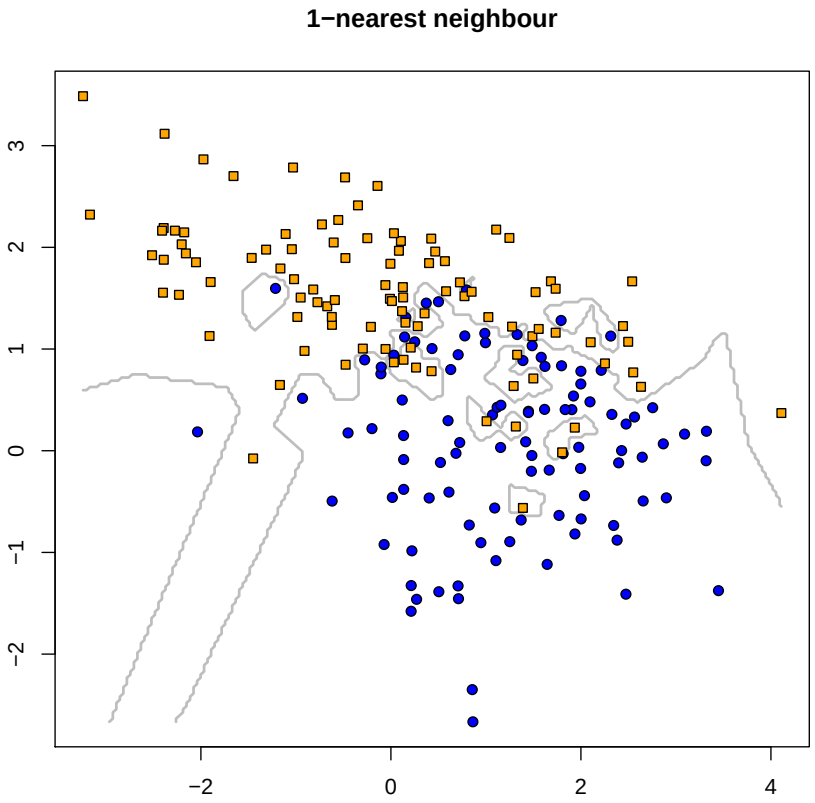
$$x^* = \underset{x_i \in \mathbf{X}}{\arg\min} ||x_i - x||. \tag{2.2}$$

Distance is usually evaluated using the Euclidean metric, but this can be changed to any other metric. For example, in some problem domains it is clear that one dimension of the input data should be highly weighted with respect to the others. The classification of x is given by the class label of the nearest point x^*, $\hat{g}(x) = g_i$. This describe the 1-nearest neighbour classifier.

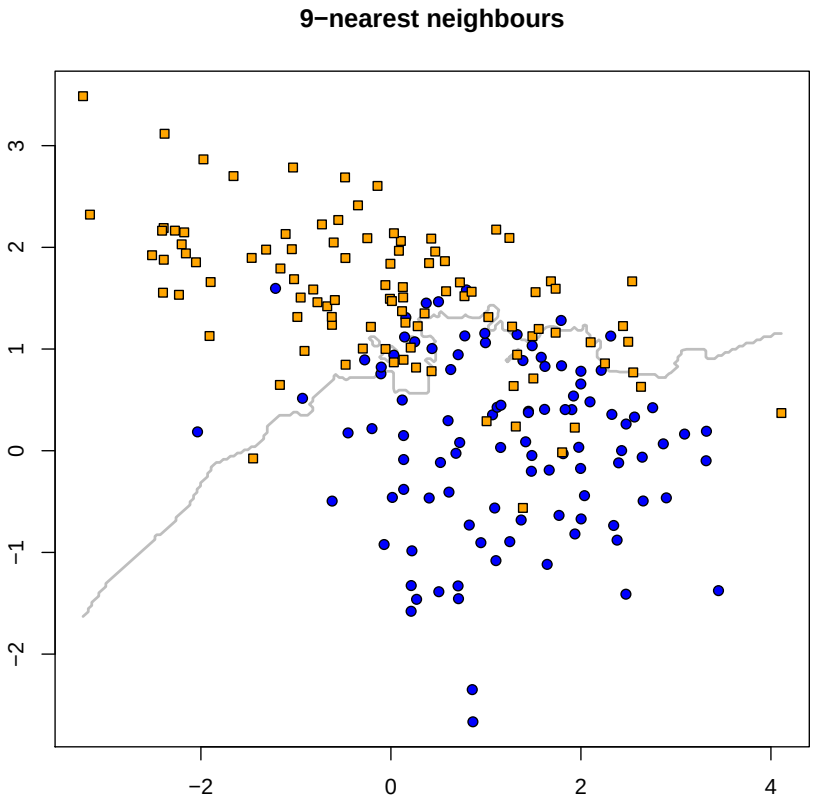
More generally, a classifier takes a majority vote of the class labels from the closest k data points to x. The resulting classifier is called the *k-nearest-neighbour* (KNN) method and is a typical example of a *nonparametric method*: one which does not consist of an explicitly parameterized model. In contrast to the linear least squares model encountered in the previous section, for which we restrict our predictor to a class of functions parameterized by β, in KNN the data itself is the model.

The choice of k has a huge effect on the decision boundaries found by the k-nearest neighbours algorithm. By choosing $k = 1$, this amounts to a highly irregular decision boundary, choosing larger k results in a smoother decision boundary.

```
# Load library for k-nearest neighbours algorithm
library(class)

# Grid of points to be predicted to find decision boundary
x1seq = seq(min(data$x1), max(data$x1), length = 200)
x2seq = seq(min(data$x2), max(data$x2), length = 200)
grid <- expand.grid(x1 = x1seq, x2 = x2seq)

# Plot 1-nearest neighbour decision boundary
classes.grid <- knn(cbind(data$x1, data$x2), grid, data$y,
                    k = 1, prob = TRUE)
prob.grid <- attr(classes.grid, "prob")
prob.grid <- ifelse(classes.grid == 0, prob.grid, 1 - prob.grid)

contour(x = x1seq, y = x2seq, z = matrix(prob.grid, nrow = 200),
        levels = 0.5, col = "grey", drawlabels = FALSE, lwd = 2,
        main = "1-nearest neighbour")
points(data$x1, data$x2, bg = c("blue", "orange")[data$y+1],
        pch = c(21, 22)[data$y+1], xlab = "", ylab = "")
```




```
# Plot 9-nearest neighbours decision boundary
classes.grid <- knn(cbind(data$x1, data$x2), grid, data$y,
                    k = 9, prob = TRUE)
prob.grid <- attr(classes.grid, "prob")
prob.grid <- ifelse(classes.grid == 0, prob.grid, 1 - prob.grid)

contour(x = x1seq, y = x2seq, z = matrix(prob.grid, nrow = 200),
        levels = 0.5, col = "grey", drawlabels = FALSE, lwd = 2,
        main = "9-nearest neighbours")
points(data$x1, data$x2, bg = c("blue", "orange")[data$y+1],
        pch = c(21, 22)[data$y+1], xlab = "", ylab = "")
```



It is difficult to determine an appropriate value of k from training data alone, as we can always reduce the training error (as measured by an objective such as RSS) to 0 by choosing $k = 1$: this amounts to drawing a highly irregular decision boundary that ensures that all of the training observations are correctly classified, but does not generalise well. Using larger k will produce a larger training error but will generalise better to new observations. In practice, an independent validation set is often used to tune this model hyperparameter.

For high-dimensional input random variables the curse of dimensionality may result in new observations being a large distance away from any of the training data and not representative of the new data point.

2.1.3 Loss functions for classification

Encoding class labels as real numbers and using a loss function such as residual sum of squares is somewhat unnatural approach to penalising errors for classification problems and does not extend naturally to problems that aren't binary. One alternative is the *zero-one loss* simply noting whether a prediction is right

or wrong,

$$L(G, \hat{G}(X)) = I(G \neq \hat{G}(X)), \tag{2.3}$$

where I is the indicator function.

In order to understand how a classifier is performing in more detail, we can construct the confusion matrix which counts how often an observation X with class label G is classified with label $\hat{G}(X)$. In the case where $\hat{G}(X) = G$ the classifier is correctly predicting the true label. When $\hat{G}(X) \neq G$ the classifier is making a misclassification which can inform us how the classifier is failing. For example, in the MNIST handwritten digits example, there may be many cases of the digit `"5"` being predicted as the digit `"6"`.

2.1.4 Approaches to classification

It is often beneficial to use a probabilistic framework to represent uncertainty about variables. Under this lens, the problem of *inference* is that of determining the joint probability distribution $\Pr(X, G)$ of the input and response variables. Once this has been determined, we can use it to make a *decision* on how to classify new points in a suitably optimal way. One approach is to find the class label G with the greatest conditional probability given the observation X,

$$\hat{G}(X) = \underset{G \in \mathcal{G}}{\arg\max} \Pr(G \mid X). \tag{2.4}$$

Alternatively, this can achieved by using the joint probability distribution $\Pr(X, G)$,

$$\begin{aligned} \underset{G \in \mathcal{G}}{\arg\max} \Pr(X, G) &= \underset{G \in \mathcal{G}}{\arg\max} \Pr(X) \Pr(G \mid X) \\ &= \underset{G \in \mathcal{G}}{\arg\max} \Pr(G \mid X) \\ &= \hat{G}(X). \end{aligned} \tag{2.5}$$

The joint probability distribution $\Pr(X, G)$ can be estimated by modelling both the conditional density $\Pr(X|G)$[1]and the prior probabilities $\Pr(G)$.

With this in mind, classification methods can broadly be divided into three groups depending on how much of this framework we wish to use:

1. Those that first model the joint distribution $\Pr(X, G)$ before using it to compute the posterior probabilities $\Pr(G \mid X)$ and make decisions. As it is also possible to marginalise the joint distribution and obtain the distribution of inputs $\Pr(X)$, these methods are called *generative models*—one can generate new input points by sampling from this distribution.

2. Those that directly model the posterior probabilities $\Pr(G \mid X)$ in order to make decisions. These models are known as *discriminative models*.

3. Those that directly model the decision boundaries between regions of $\mathbb{R}^p$ that correspond to different classes or otherwise directly predict the class labels without reference to probabilities.

The k-nearest neighbours algorithm is an example of class 3 as the training data defines the decision boundaries without reference to any probability model. In the next section, we will describe some discriminative models which aim to model $\Pr(G \mid X)$ directly.

2.2 Generative models

2.2.1 Linear discriminant analysis

Suppose that we know the within-class conditional density of x given that $G = k$, which we shall denote by $f_k(x)$. Then, Bayes theorem tells us that

$$\Pr(G = k \mid X = x) = \frac{f_k(x)\pi_k}{\sum_\ell f_\ell(x)\pi_\ell}, \tag{2.6}$$

where π_k is the prior probability of class k. By making different assumptions on the form of the densities f_k, we can derive a family of models with differing complexities and characteristics. One of the most simple is the *linear discriminant analysis* model (LDA), where we take each f_k to be Gaussian:

$$f_k(x) = \frac{1}{(2\pi)^{p/2}|\Sigma|^{1/2}} e^{-\frac{1}{2}(x-\mu_k)^\top \Sigma^{-1}(x-\mu_k)}, \tag{2.7}$$

with class mean μ_k and shared covariance Σ. Under this assumption, we can compute the log-odds for two classes k and ℓ:

$$\log \frac{\Pr(G = k \mid X = x)}{\Pr(G = \ell \mid X = x)} = \log \frac{\pi_k}{\pi_\ell} - \frac{1}{2}(\mu_k + \mu_\ell)^\top \Sigma^{-1}(\mu_k - \mu_\ell) + x^\top \Sigma^{-1}(\mu_k - \mu_\ell), \tag{2.8}$$

which is linear in x. Our aim is to find a discriminant function $\delta_k(x)$ so that the classifier predicts label $G = k$ if the associated discriminant function is larger than the other class labels, that is

$$\hat{G}(x) = \arg\max_{k \in \mathcal{G}} \delta_k(x). \tag{2.9}$$

This is equivalent to saying that a classifier choose label k if, for all $\ell \neq k$,

$$\Pr(G = k \mid X = x) > \Pr(G = \ell \mid X = x), \tag{2.10}$$

or, in terms of the log-odds,

$$\log \frac{\Pr(G = k \mid X = x)}{\Pr(G = \ell \mid X = x)} > 0. \tag{2.11}$$

This leads to the discriminant function

$$\delta_k(x) = x^\top \Sigma^{-1}\mu_k - \frac{1}{2}\mu_k^\top \Sigma^{-1}\mu_k + \log \pi_k, \tag{2.12}$$

[1]Note that the quantity $\Pr(X|G)$ is *not* in general a probability distribution over G.

which is a function of the observation x and the Gaussian distribution parameters μ_k and Σ. Thus, any boundary between regions of $\mathbb{R}^p$ that are assigned to a pair of classes k and ℓ is in fact linear.

In order to compute predictions, we need to estimate the π_k, μ_k and Σ. We can do this through maximum likelihood estimation. For convenience, we will restrict to the binary classification case for our exposition, but the same approach readily generalises to multiple classes.

Using the Gaussian probability density function given in Equation 2.7, ignoring the $(2\pi)^{p/2}$ constant, the log likelihood is given by

$$l(\pi, \mu_1, \mu_2, \Sigma) = \sum_{i=1}^{N} \Bigg[y_i \log \pi - \frac{1}{2} y_i (x_i - \mu_1)^\top \Sigma^{-1} (x_i - \mu_1) + (1 - y_i) \log(1 - \pi) \\ - \frac{1}{2}(1 - y_i)(x_i - \mu_2)^\top \Sigma^{-1} (x_i - \mu_2) \Bigg] - \frac{N}{2} \log |\Sigma| \tag{2.13}$$

where $\pi = \pi_1 = 1 - \pi_2$. Taking the derivative with respect to π gives

$$\frac{\partial}{\partial \pi} l(\pi, \mu_1, \mu_2, \Sigma) = \sum_{i=1}^{N} \left[\frac{y_i}{\pi} - \frac{1 - y_i}{1 - \pi} \right], \tag{2.14}$$

and setting it equal to 0 yields

$$\hat{\pi} = \frac{1}{N} \sum_{i=1}^{N} y_i = \frac{N_1}{N_1 + N_2}, \tag{2.15}$$

where N_k is the number of points in class k. Thus the maximum likelihood estimate for the prior probability of each class is simply the fraction of training points in that class.

Repeating the process for the class mean μ_1, we take the derivative with respect to μ_1

$$\frac{\partial}{\partial \mu_1} l(\pi, \mu_1, \mu_2, \Sigma) = \sum_{i=1}^{N} y_i \Sigma^{-1} (x_i - \mu_1), \tag{2.16}$$

and setting it equal to 0 yields

$$\hat{\mu}_1 = \frac{1}{N_1} \sum_{i=1}^{N} y_i x_i. \tag{2.17}$$

Similarly, for the class mean μ_2, gives

$$\hat{\mu}_2 = \frac{1}{N_2} \sum_{i=1}^{N} (1 - y_i) x_i. \tag{2.18}$$

These are the empirical means of the input points in each class. The result for the covariance matrix is a little more difficult to obtain, but can be computed

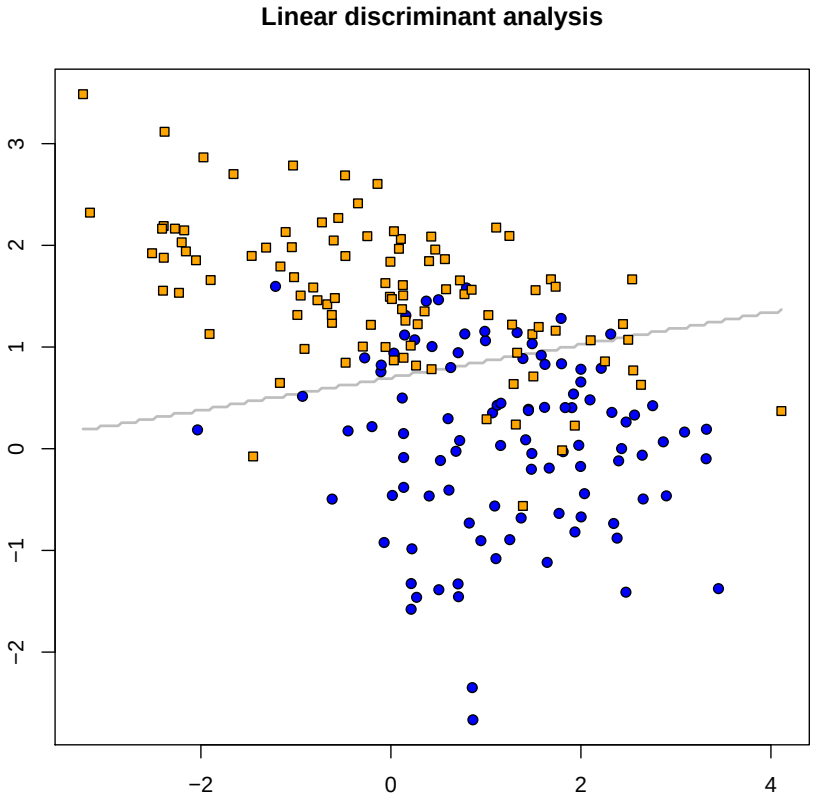
to be:

$$\hat{\Sigma} = \frac{1}{N}\sum_{i=1}^{N}\left[y_i(x_i - \hat{\mu}_1)(x_i - \hat{\mu}_1)^\top + (1 - y_i)(x_i - \hat{\mu}_2)(x_i - \hat{\mu}_2)^\top\right]. \quad (2.19)$$

This can be seen as a weighted sum of the covariance matrices associated with each of the two classes.

In the case of binary classification, the discriminant function leads to a single linear decision boundary, an example given below. For multiple classes, the decision boundaries will consist of piece-wise linear boundaries.

```r
# Load library for linear discriminant analysis algorithm
library(MASS)

# Learn LDA model
lda.bo = lda(y ~ x1 + x2, data = data)

# Grid of points to be predicted to find decision boundary
x1seq = seq(min(data$x1), max(data$x1), length = 200)
x2seq = seq(min(data$x2), max(data$x2), length = 200)
grid <- expand.grid(x1 = x1seq, x2 = x2seq)

# Plot LDA decision boundary
pred.grid <- as.numeric(predict(lda.bo, grid)$class)
contour(x = x1seq, y = x2seq, z = matrix(pred.grid, nrow=200),
        levels = 1.5, col = "grey", drawlabels = FALSE, lwd = 2,
        main="Linear discriminant analysis")
points(data$x1, data$x2, bg = c("blue", "orange")[data$y+1],
       pch = c(21, 22)[data$y+1], xlab = "", ylab = "")
```



2.2.2 Quadratic discriminant analysis

One way in which we can obtain a classification without linear decision boundaries is by dropping the assumption of a shared covariance matrix for the class density in LDA, giving each class its own covariance matrix Σ_k,

$$f_k(x) = \frac{1}{(2\pi)^{p/2}|\Sigma_k|^{1/2}} e^{-\frac{1}{2}(x-\mu_k)^\top \Sigma_k^{-1}(x-\mu_k)}. \tag{2.20}$$

Under this assumption, we can compute the log-odds for two classes k and ℓ:

$$\begin{aligned}\log \frac{\Pr(G=k|X=x)}{\Pr(G=\ell|X=x)} &= \log \pi_k - \frac{1}{2}\log|\Sigma_k| - \frac{1}{2}(x-\mu_k)^\top \Sigma_k^{-1}(x-\mu_k) \\ &\quad - \log \pi_\ell + \frac{1}{2}\log|\Sigma_\ell| + \frac{1}{2}(x-\mu_\ell)^\top \Sigma_k^{-1}(x-\mu_\ell), \end{aligned} \tag{2.21}$$

which is quadratic in x giving rise to the name *quadratic discriminant analysis* (QDA). The discriminant function then becomes

$$\delta_k(x) = -\frac{1}{2}\log|\Sigma_k| - \frac{1}{2}(x-\mu_k)^\top \Sigma_k^{-1}(x-\mu_k) + \log \pi_k, \tag{2.22}$$

a quadratic function of the observation x and the Gaussian distribution parameters μ_k and Σ.

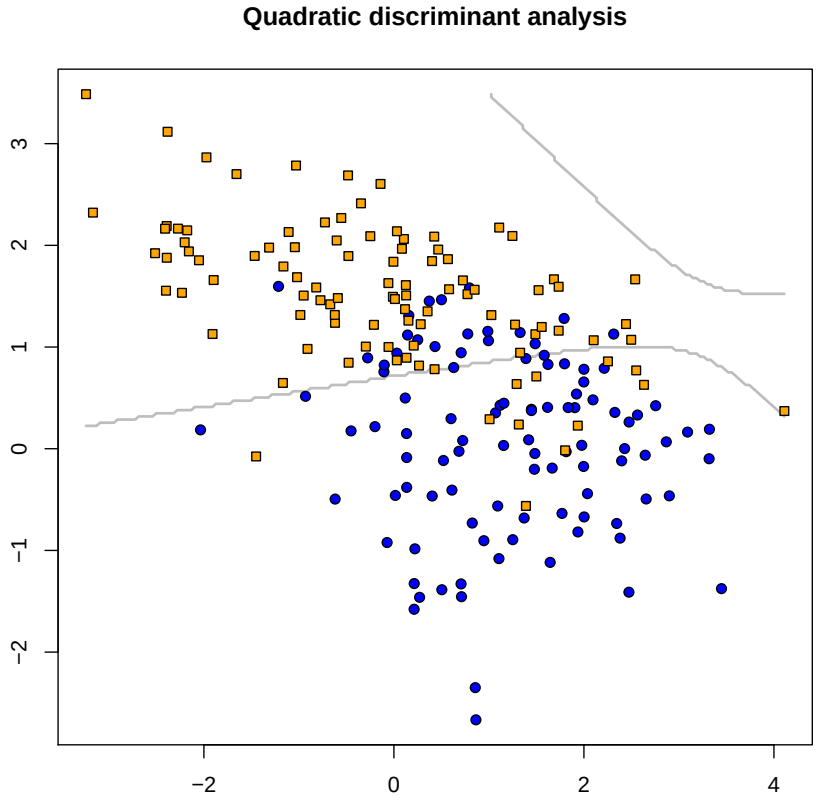
Again, in order to compute predictions, we need to estimate the π_k, μ_k and Σ_k using maximum likelihood estimation. The calculations are similar to these for linear discriminant analysis, for example, in the binary classification case the maximum likelihood estimates for π, μ_1 and μ_2 are the same as Equations 2.15, 2.17 and 2.18 respectively. The maximum likelihood estimates for Σ_1 and Σ_2 are the associated covariance matrices for the two classes:

$$\hat{\Sigma}_1 = \frac{1}{N_1}\sum_{i=1}^{N} y_i(x_i - \hat{\mu}_1)(x_i - \hat{\mu}_1)^\top, \tag{2.23}$$

$$\hat{\Sigma}_2 = \frac{1}{N_2}\sum_{i=1}^{N} (1-y_i)(x_i - \hat{\mu}_2)(x_i - \hat{\mu}_2)^\top. \tag{2.24}$$

In the case of binary classification, the discriminant function leads to a quadratic decision boundary. In the following example, we see that the points in the top-right of the plot are classified as blue points. This is due to the different shapes of the covariance matrices $\hat{\Sigma}_1$ and $\hat{\Sigma}_2$.

```
# Learn QDA model
qda.bo = qda(y ~ x1 + x2, data = data)

# Grid of points to be predicted to find decision boundary
x1seq = seq(min(data$x1), max(data$x1), length = 200)
x2seq = seq(min(data$x2), max(data$x2), length = 200)
grid <- expand.grid(x1 = x1seq, x2 = x2seq)

# Plot QDA decision boundary
pred.grid <- as.numeric(predict(qda.bo, grid)$class)
contour(x = x1seq, y = x2seq, z = matrix(pred.grid, nrow=200),
        levels = 1.5, col = "grey", drawlabels = FALSE, lwd = 2,
        main="Quadratic discriminant analysis")
points(data$x1, data$x2, bg = c("blue", "orange")[data$y+1],
       pch = c(21, 22)[data$y+1], xlab = "", ylab = "")
```



2.2.3 Regularised discriminant analysis

Regularised discriminant analysis aims to compromise between LDA, which fits a common covariance matrix across all classes, and QDA, which fits separate covariance matrices to each class. This is done using regularised covariance matrices of the form

$$\hat{\Sigma}_k(\alpha) = \alpha\hat{\Sigma}_k + (1-\alpha)\hat{\Sigma}, \tag{2.25}$$

where $\hat{\Sigma}$ is the maximum likelihood estimate of the common covariance matrix for LDA (Equation 2.19) and $\hat{\Sigma}_k$ is the maximum likelihood estimate of the separate covariance matrices for QDA (Equation 2.23 and 2.24). The choice of $\alpha \in [0,1]$ allows for a range of models with $\alpha = 0$ corresponding to LDA and $\alpha = 1$ corresponding to QDA. In practise the parameter α is chosen using a validation data set.

2.3 Logistic regression and extensions

2.3.1 Logistic regression

Another model that produces linear decision boundaries is *logistic regression*. This was introduced as a special kind of generalised linear model in Section 1.5. There it was considered for the binary classification problem. In this section, the approach will be extended to classification with $K = |\mathcal{G}|$ class labels.

Class label K is fixed and the log-odds for all other classes to class K is directed parameterised as being a linear function:

$$\log \frac{\Pr(G = k \mid X = x)}{\Pr(G = K \mid X = x)} = \beta_{k0} + \beta_k^\top x. \tag{2.26}$$

The conditional class probabilities sum to one as the data must be assigned to one of the possible classes. Using Equation 2.26 we can rewrite the expression using the posterior probability for class K,

$$\Pr(G = K \mid X = x) = \frac{1}{1 + \sum_{\ell=1}^{K-1} \exp(\beta_{\ell 0} + \beta_\ell^\top x)}. \tag{2.27}$$

This can be substituted back into Equation 2.26 to get the individual posterior probabilities for classes $k = 1, \ldots, K-1$,

$$\Pr(G = k \mid X = x) = \frac{\exp(\beta_{k0} + \beta_k^\top x)}{1 + \sum_{\ell=1}^{K-1} \exp(\beta_{\ell 0} + \beta_\ell^\top x)}. \tag{2.28}$$

We denote $\Pr(G = k \mid X = x) = p_k(x; \theta)$ where $\theta = \{\beta_{10}, \beta_1^\top, \ldots \beta_{(K-1)0}, \beta_{K-1}^\top\}$ is the parameter vector. The logistic regression model is fitted by maximising the log likelihood

$$l(\theta) = \sum_{i=1}^{N} \log p_{g_i}(x_i; \theta). \tag{2.29}$$

In the case of binary classification, the calculations simplify significantly to give some insight into how to solve this optimisation problem. Let $p_1(x; \theta) = p(x; \theta)$ and $p_2(x; \theta) = 1 - p(x; \theta)$ with parameter set $\theta = \{\beta_0, \beta\}$ where we have dropped the redundant k index from the β parameters. Define zero/one labels y_i such that $y_i = 0$ if $g_i = 1$ and $y_i = 1$ if $g_i = 2$. The log likelihood is given by

$$\begin{aligned} l(\theta) &= \sum_{i=1}^{N} \left[y_i \log p(x_i; \theta) + (1 - y_i) \log(1 - p(x_i; \theta)) \right] \\ &= \sum_{i=1}^{N} \left[y_i(\beta_0 + \beta^\top x_i) + \log(1 + \exp(\beta_0 + \beta^\top x_i)) \right]. \end{aligned} \tag{2.30}$$

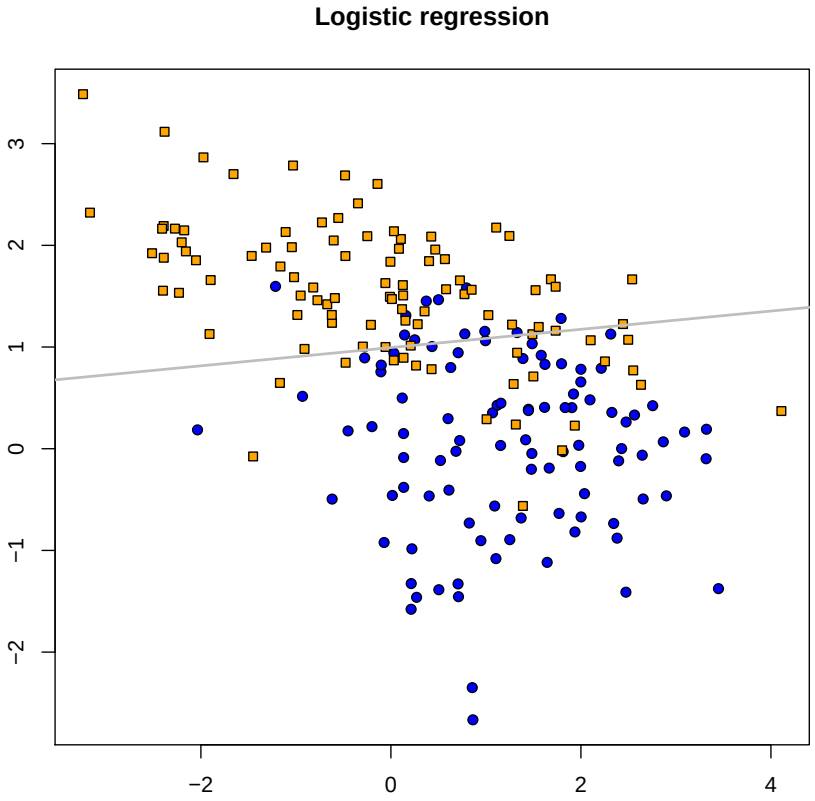
To maximise the log likelihood, we set the derivatives with respect to β_0 and β equal to zero,

$$\begin{aligned}\frac{\partial l(\theta)}{\partial \beta_0} &= \sum_{i=1}^{N}\left[y_i - \frac{\exp(\beta_0 + \beta^\top x_i)}{1+\exp(\beta_0+\beta^\top x_i)}\right] \\ &= \sum_{i=1}^{N}(y_i - p(x_i;\theta)) = 0 \qquad (2.31)\\ \frac{\partial l(\theta)}{\partial \beta} &= \sum_{i=1}^{N}\left[y_i x_i - \frac{\exp(\beta_0 + \beta^\top x_i)}{1+\exp(\beta_0+\beta^\top x_i)}x_i\right] \\ &= \sum_{i=1}^{N}(y_i - p(x_i;\theta))x_i = 0 \qquad (2.32)\end{aligned}$$

Note that Equation 2.31 states that the expected number in class 1, $\sum p(x_i;\theta)$, is equal to the observed number, $\sum y_i$.

These equations provide $p+1$ nonlinear conditions to solve, which can be solved using standard optimisation techniques like the Newton-Raphson algorithm. An initial parameter choice θ_0 is selected and updated using small steps until convergence is reached. When there are more than two classes, the optimisation task is more complex but can be solved using extension to techniques like Newton-Raphson.

```
# Fit logistic regression model
lr.bo <- glm(y ~ x1 + x2, family = binomial, data = data)

# Calculate coefficients of linear decision boundary
intercept = (0.5 - coef(lr.bo)[1]) / coef(lr.bo)[3]
gradient = - coef(lr.bo)[2] / coef(lr.bo)[3]

# Plot data and linear decision boundary
plot(data$x1, data$x2, bg = c("blue", "orange")[data$y+1],
     pch = c(21, 22)[data$y+1], xlab = "", ylab = "",
     main = "Logistic regression")
abline(intercept, gradient, col = "grey", lwd = 2)
```



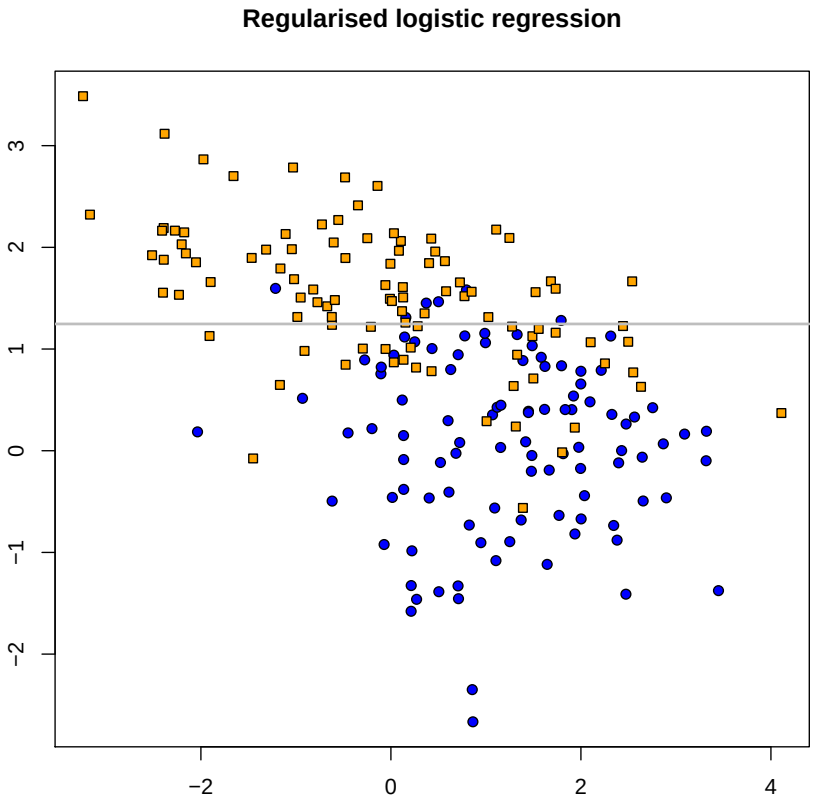
2.3.2 Regularised logistic regression

The ℓ_1 penalty using in the Lasso from Section 1.3.3 can be used to discourage the logistic regression model from too many variables. For binary classification using logistic regression, we aim to maximise a penalised version of the log likelihood given in Equation 2.30:

$$\max_{\beta_0,\beta} \left\{ \sum_{i=1}^{N} \left[y_i(\beta_0 + \beta^\top x_i) + \log(1 + \exp(\beta_0 + \beta^\top x_i)) \right] - \lambda \sum_{j=1}^{p} |\beta_j| \right\}. \quad (2.33)$$

As with the Lasso, using a large value of λ makes the optimisation want to set many of the model parameters $\beta_j = 0$ to avoid large penalties. With $\lambda = 0$, the optimisation reduces to standard logistic regression. In the following example, by choice $\lambda = 0.1$, the algorithm sets one $\beta_j = 0$ resulting in a horizontal line for the decision boundary.

```r
# Load library for regularised logistic regression
library(glmnet)

## Loading required package:  Matrix
## Loaded glmnet 4.1-2

# Fit regularised logistic regression model with lambda = 0.1
rlr.bo <- glmnet(cbind(data$x1, data$x2), data$y,
                 family = "binomial", alpha = 1, lambda = 0.1)

# Calculate coefficients of linear decision boundary
intercept = (0.5 - coef(rlr.bo)[1]) / coef(rlr.bo)[3]
gradient = - coef(rlr.bo)[2] / coef(rlr.bo)[3]

# Plot data and linear decision boundary
plot(data$x1, data$x2, bg = c("blue", "orange")[data$y+1],
     pch = c(21, 22)[data$y+1], xlab = "", ylab = "",
     main = "Regularised logistic regression")
abline(intercept, gradient, col = "grey", lwd = 2)
```



2.3.3 Nonparameter logistic regression

As remarked earlier, there are many situations where purely linear models have insufficient complexity to capture the structure of data. In the case of logistic regression, one way in which this linearity can be relaxed is through the use of a *generalised additive model* where the linear combination of input features $\beta^\top X$ is replaced by an additive combination of unspecified smooth functions f_i. For two-class logistic regression, the log odds are then given by

$$\log \frac{\Pr(G = 0 \mid X)}{\Pr(G = 1 \mid X)} = \alpha + \sum_{i=1}^{p} f_i(X). \tag{2.34}$$

Each of the functions f_i can then be fitted independently, for instance through the use of splines.

2.3.4 Comparison to linear discriminant analysis

The log-odds for linear discriminant analysis given by Equation 2.8 can be written in a similar way to the log-odds for logistic regression given by Equation 2.26,

$$\begin{aligned}\log \frac{\Pr(G=k \mid X=x)}{\Pr(G=K \mid X=x)} &= \log \frac{\pi_k}{\pi_K} - \frac{1}{2}(\mu_k + \mu_K)^\top \Sigma^{-1}(\mu_k - \mu_K) \\ &\quad + x^\top \Sigma^{-1}(\mu_k - \mu_K) \\ &= \alpha_{k0} + \alpha_k^\top x. \end{aligned} \tag{2.35}$$

While the two linear models have the same form, the way that the model parameters are estimated in different ways. In LDA the form of the linear coefficients of the log-odds is expressed in terms of the Gaussian parameters; whereas in logistic regression it is left totally unrestricted. To highlight the difference, consider the joint density of X and G using Bayes rule,

$$\Pr(X=x, G=k) = \Pr(X=x)\Pr(G=k \mid X=x). \tag{2.36}$$

Both algorithms have the same linear forms for $\Pr(G=k \mid X=x)$ but they differ in how they treat the marginal density $\Pr(X=x)$. For logistic regression, this density is ignored and all modelling is done directly on the conditional density function. For LDA, the density can be written as a mixture of Gaussian distributions,

$$\Pr(X=x) = \sum_{k=1}^{K} \pi_k \phi(x; \mu_k, \Sigma), \tag{2.37}$$

where ϕ is the Gaussian density function. Using this model, if the true f_k are from a Gaussian distribution, we are able to calculate the decision boundary more efficiently with fewer data points. Conversely, when the true f_k differ significantly from a Gaussian density, logistic regression is more general and will likely perform better.

2.4 Support vector machine

2.4.1 Linearly separable data

We have seen that both linear discriminant analysis and logistic regression in Sections 2.3.1 and 2.2.1 produce linear decision boundaries. In this section, we describe classifiers that directly try to place a hyperplane that best separates the data into the true classes.

Restricting attention to binary classification problems momentarily, we call a data set *linearly separable* if the two classes can be separated by a hyperplane. Points on one side of the hyperplane all belong to one class, points on the other side all belong to the other class.

In this setting it is convenient to map the two classes to labels $y_i \in \{-1, 1\}$. We define the linear decision boundary using parameters β_0, β,

$$\{x \in \mathbb{R}^p : \beta_0 + \beta^\top x = 0\}. \tag{2.38}$$

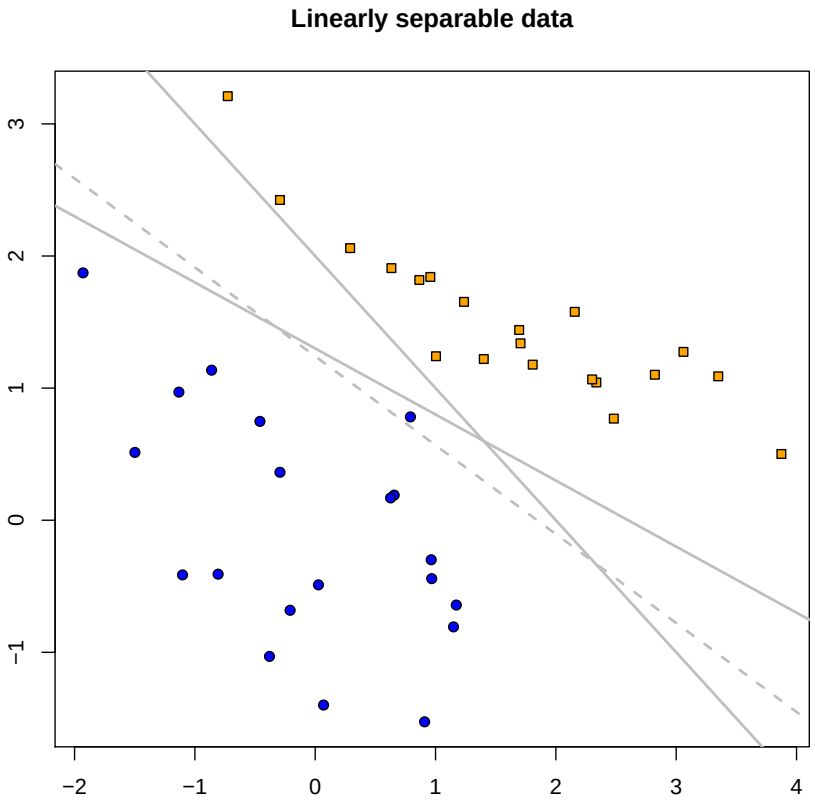
If a point x_i is correctly classified then the label y_i will have the same sign as the linear function evaluated at x_i,

$$y_i = \text{sgn}(\beta_0 + \beta^\top x_i). \tag{2.39}$$

Conversely, if a point x_i is misclassified the sign will not match the label $y_i \neq \text{sgn}(\beta_0 + \beta^\top x_i)$.

The following example shows 40 observations of two classes represented by 20 blue and 20 orange points that are linearly separable. There are infinitely many separating hyperplanes, the solid grey lines showing two such possible examples with very different gradients. Points either side of those lines are either all blue or all orange. The dashed grey line shows the linear decision boundary found by linear least squares regression for binary classification discussed in Section 2.1.1. This classifier does not separate the data into its two classes and one point is misclassified.

```
# Load data
load(file = 'Data/BlueOrangeLS.Rdata')

# Fit linear regression model
lm.LS = lm(y ~ x1 + x2, data = data_LS)

# Calculate coefficients of linear decision boundary
intercept = (0.5 - coef(lm.LS)[1]) / coef(lm.LS)[3]
gradient = - coef(lm.LS)[2] / coef(lm.LS)[3]

# Plot data, linear decision boundary and separating hyperplanes
plot(data_LS$x1, data_LS$x2, bg = c("blue","orange")[data_LS$y+1],
     pch = c(21, 22)[data_LS$y+1], xlab = "", ylab = "",
     main = "Linearly separable data")
abline(intercept, gradient, col = "grey", lwd = 2, lty = 2)
abline(1.3, -0.5, col = "grey", lwd = 2)
abline(2.0, -1.0, col = "grey", lwd = 2)
```



2.4.2 Perceptron

The previous example showed that linear least squares regression for binary classification may not separate linearly separable data, but this is potentially an issue for all the algorithms described so far. There also exists linearly separable data sets that do not get classified correctly when using LDA or logistic regression. In this section, we consider the *perceptron*—a method that explicitly constructs a linear decision boundary that separates the classes.

Given linearly separable data, the following algorithm iterates through each data points, checking if the current hyperplane correctly classifies the data. If not the hyperplane parameters β and β_0 are updated and the process is repeated until convergence.

Algorithm 8: Perceptron algorithm

Input: $\{x_i, y_i\}$ linearly separable
Initialize β, β_0;
$i \leftarrow 0$;
while *not converged* **do**
  $i \leftarrow (i \mod N) + 1$;
  $\hat{y}_i \leftarrow \text{sgn}(\beta_0 + \beta^\top x_i)$;
  if $\hat{y}_i \neq y_i$ **then**
    $\beta \leftarrow \beta + y_i x_i$;
    $\beta_0 \leftarrow \beta_0 + y_i$;
  else
    pass;
  end
end

The following theorem gives us a guarantee of a solution (the proof is beyond the scope of this course):

Theorem 7 (Perceptron Convergence Theorem)**.** *For a finite set of linearly separable data, the Perceptron algorithm will halt after a finite number of iterations (i.e. all data points will be correctly classified).*

Note that there is no assertion of uniqueness of the separating hyperplane, nor any bound on the number of iterations that may be required.

The Perceptron learning algorithm can be viewed as an example of *stochastic gradient descent*. The distance of a point x to the decision boundary is given by the magnitude of $\beta_0 + \beta^\top x$. We are interested in the total distance of misclassified points to the decision boundary,

$$D(\beta, \beta_0) = -\sum_{i \in \mathcal{M}} y_i(\beta_0 + \beta^\top x_i), \tag{2.40}$$

where $\mathcal{M} = \{i : \hat{y}_i \neq y_i\}$ is the set of points where the predicted class is not equal to the true class. Computing the gradient of this expression with respect to the parameters gives

$$\frac{\partial}{\partial \beta}\left(D(\beta, \beta_0)\right) = -\sum_{i \in \mathcal{M}} y_i x_i, \tag{2.41}$$

$$\frac{\partial}{\partial \beta_0}\left(D(\beta, \beta_0)\right) = -\sum_{i \in \mathcal{M}} y_i. \tag{2.42}$$

These gradient terms are what are used to update the parameters in the algorithm specified above.

Later, we will see two similar methods that are capable of classifying data that is not linearly separable. Firstly, multiple perceptrons can be composed in order to form the *multi-layer perceptron*, an example of some of the simplest *neural networks*. Secondly, the minimisation problem can be modified to allow some overlap but penalise the extent of it, yielding the *support vector classifier*.

2.4.3 Support vector classifier

In our discussion of the perceptron algorithm, we derived a procedure that was guaranteed to find a separating hyperplane for linearly separable data. We will now extend this procedure to find *the* hyperplane that separates the classes optimally in some sense. Instead of minimising the distance of misclassified points to the decision boundary, we can formulate a constrained optimization problem in which we maximise the *margin* between the two classes:

$$\max_{\beta_0,\beta,\|\beta\|=1} M \tag{2.43}$$

$$\text{subject to } y_i(\beta_0 + \beta^\top x_i) \geq M \quad i = 1, \ldots, N. \tag{2.44}$$

If the data is linearly separable, every data point is at least a distance M from the separating hyperplane. By making this distance as large as possible, the idea is to make the model as robust as possible by having the greatest margin of error.

We can rewriting the distance conditions using $\beta'_0 = \beta_0/M$ and $\beta' = \beta/M$

$$y_i(\beta'_0 + \beta'^\top x_i) \geq 1 \quad i = 1, \ldots, N, \tag{2.45}$$

when maximising M is now equivalent to minimising β'. Therefore we can rewrite the optimisation task while removing all mention all the margin:

$$\min_{\beta_0,\beta} \frac{1}{2}\|\beta\|^2 \tag{2.46}$$

$$\text{subject to } y_i(\beta_0 + \beta^\top x_i) \geq 1 \quad i = 1, \ldots, N, \tag{2.47}$$

where, by minimising $\frac{1}{2}\|\beta\|^2$ instead of $\|\beta\|$, it is easier to compute the derivatives. This is in the form of a general inequality-constrained optimization problem

$$\min_w f(w) \quad \text{subject to } c_j(w) \geq 0 \quad j \in \mathcal{I}. \tag{2.48}$$

Such problems can be solved by defining a *Lagrangian*

$$L_P(w, \lambda) = f(w) - \sum_{j \in \mathcal{I}} \lambda_j c_j(w), \tag{2.49}$$

and solving a set of equations known as the *Karush-Kuhn-Tucker (KKT) conditions*. Full technical details are of these are given in Appendix A. For Equation 2.47, the Lagrangian equation is given by

$$L_P(\beta_0, \beta, \alpha) = \frac{1}{2}\|\beta\|^2 - \sum_{i=1}^{N} \alpha_i \left[y_i(\beta_0 + \beta^\top x_i) - 1 \right]. \tag{2.50}$$

It is sometimes more computationally convenient to work with an alternative objective function called the *dual* to the *primal* Lagrangian given above:

$$L_D(\lambda) \coloneqq \inf_w L_P(w, \lambda) \tag{2.51}$$

We then maximise L_D subject to the KKT conditions. In the case of the support vector classifier with Lagrangian given in Equation 2.50, the dual objective function is given by

$$L_D(\alpha) = \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j x_i^\top x_j. \tag{2.52}$$

The optimal value $\hat{\beta}$ can be expressed in terms of the optimal values $\hat{\alpha}_i$, which can be determined by setting the partial derivative $\frac{\partial}{\partial \beta} L_P(\beta_0, \beta, \alpha)$ equal to zero.

$$\hat{\beta} = \sum_{i=1}^{N} \hat{\alpha}_i y_i x_i. \tag{2.53}$$

The support vector classifier uses the optimal values of $\hat{\alpha}_i$ and $\hat{\beta}$ to define a linear decision boundary and classifies new data x depending on which side of the hyperplane it lies:

$$\begin{aligned} \hat{G}(x) &= \operatorname{sgn}\left(\hat{\beta}_0 + \hat{\beta}^\top x\right) \\ &= \operatorname{sgn}\left(\hat{\beta}_0 + \sum_{i=1}^{N} \hat{\alpha}_i y_i x_i^\top x\right). \end{aligned} \tag{2.54}$$

2.4.4 Support vector classifier for non-separable data

In order to adapt it for non-separable data, we modify the hard margin constraint to read

$$y_i(\beta_0 + \beta^\top x_i) \geq M(1 - \xi_i), \tag{2.55}$$

where ξ_i is a *slack variable* subject to the constraints $\xi_i \geq 0$, $\sum_{i=1}^{N} \xi_i < K$ for some constant K. The essential idea is that points are now allowed to lie within the margin and even cross over to the wrong side of the decision boundary, but there is a global "budget" for these transgressions that cannot be exceeded. Using the same equivalence between maximization and minimization as above, the full problem becomes

$$\min_{\beta_0, \beta, \xi_i} \frac{1}{2} \|\beta\|^2 \tag{2.56}$$

$$\text{subject to } \begin{cases} y_i(\beta_0 + \beta^\top x_i) \geq 1 - \xi_i & i = 1, \ldots, N \\ \xi_i \geq 0,\ \sum_{i=1}^{N} \xi_i < K \end{cases}. \tag{2.57}$$

We will now detail how the support vector classifier can be fitted using techniques from the field of constrained optimization. First, we can rewrite the minimization problem above as

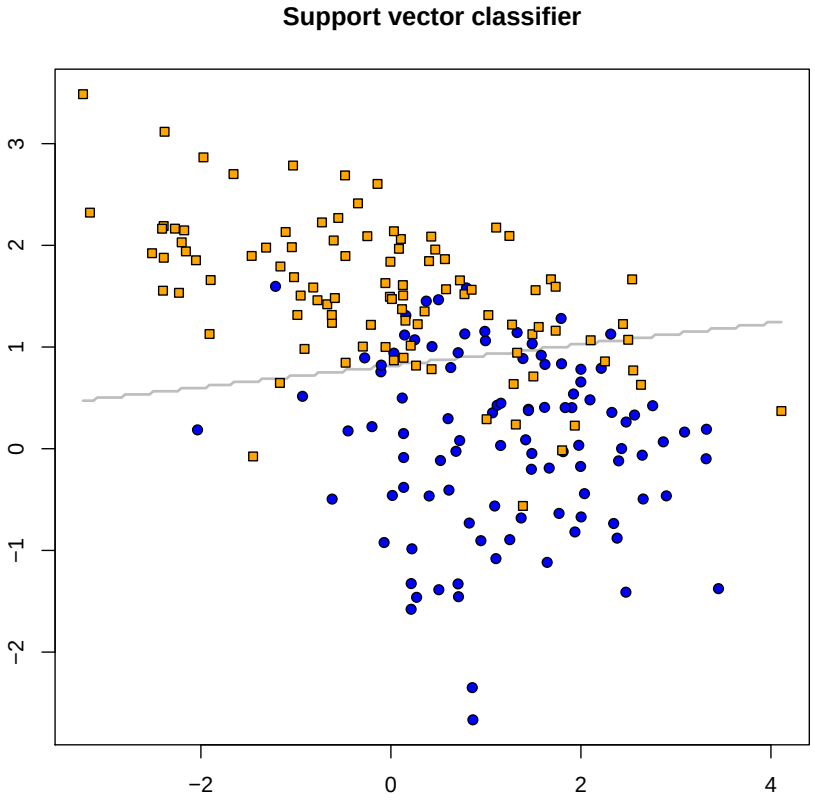
$$\min_{\beta_0, \beta, \xi_i} \frac{1}{2} \|\beta\|^2 + C \sum_{i=1}^{N} \xi_i \tag{2.58}$$

$$\text{subject to } \xi_i \geq 0,\ y_i(\beta_0 + \beta^\top x_i) \geq 1 - \xi_i \quad i = 1, \ldots, N. \tag{2.59}$$

Here, the cost parameter C imposes the constraint on the sum of the slack variables.

Efficient implementations of a variety of minimization algorithms can be found in software packages such as Python's `scipy.optimize` and R's `nloptr`. For practical purposes, these languages have build-in functions to train a support vector classifier.

```
# Load library for support vector machines
library(e1071)

# Fit support vector classifier
svc.bo = svm(as.factor(y) ~ x1 + x2, data = data,
             kernel = "linear", cost = 10, scale = FALSE)

# Grid of points to be predicted to find decision boundary
x1seq = seq(min(data$x1), max(data$x1), length = 200)
x2seq = seq(min(data$x2), max(data$x2), length = 200)
grid <- expand.grid(x1 = x1seq, x2 = x2seq)

# Plot SVC decision boundary
pred.grid <- as.numeric(predict(svc.bo, grid))
contour(x = x1seq, y = x2seq, z = matrix(pred.grid, nrow=200),
        levels = 1.5, col = "grey", drawlabels = FALSE, lwd = 2,
        main="Support vector classifier")
points(data$x1, data$x2, bg = c("blue", "orange")[data$y+1],
       pch = c(21, 22)[data$y+1], xlab = "", ylab = "")
```



2.4.5 Support vector machine

We can generalise the support vector classifier to be able to construct nonlinear decision boundaries by replacing the linear combination $\beta_0 + \beta^\top x$ with a nonlinear function $\beta_0 + \beta^\top h(x)$, where $h(x) = (h_1(x), \ldots, h_M(x))$ is a basis expansion in the manner described in Section 1.4.3. By finding a linear decision boundary for the points $h(x)$, this can be converted back into a non-linear decision boundaries for the points x.

Looking at the form of the dual objective function Equation 2.52, we see that it only depends on the values of x through the inner product $x^\top x$. The dual objective function for the problem with the transformed features $h(x)$ can similarly be written as

$$L_D(\alpha) = \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j \langle h(x_i), h(x_j) \rangle. \tag{2.60}$$

Likewise, from Equation (2.53) and 2.54 the decision function can also be expressed in terms of feature inner products:

$$\hat{G}(x) = \text{sgn}\left(\sum_{i=1}^{N} \hat{\alpha}_i y_i \langle h(x), h(x_i)\rangle + \hat{\beta}_0\right). \tag{2.61}$$

As such, instead of specifying $h(x)$, we can specify a *kernel function* K for which

$$K(x, x') = \langle h(x), h(x')\rangle. \tag{2.62}$$

Decision functions can be expressed as a linear combination of the kernel function evaluated at the training points as in (2.61). A few common examples of kernels are:

$$\begin{aligned} \text{Linear kernel: } & K(x, x') = x^\top x', \\ \text{Polynomial kernel: } & K(x, x') = (1 + x^\top x')^d, \\ \text{Gaussian kernel: } & K(x, x') = \exp\left(-\|x - x'\|^2/(2\sigma^2)\right). \end{aligned} \tag{2.63}$$

All of the kernels above produce positive semi-definite Gram matrices

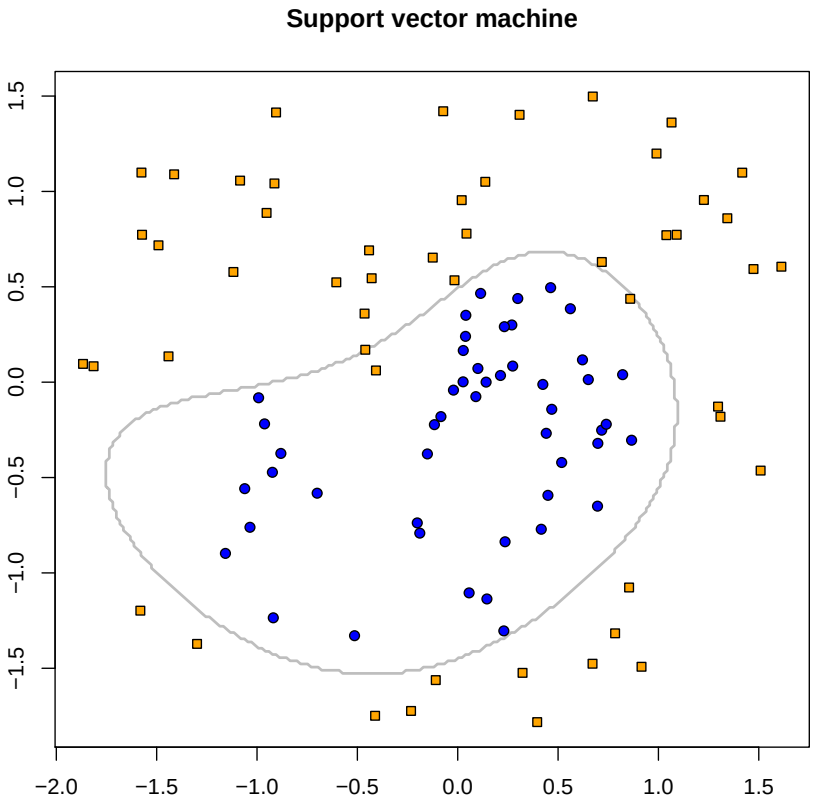
$$\mathbf{K} = \begin{pmatrix} k(x_1, x_1) & \cdots & k(x_1, x_N) \\ \vdots & \ddots & \vdots \\ k(x_N, x_1) & \cdots & k(x_N, x_N) \end{pmatrix}. \tag{2.64}$$

This condition is necessary for the application of the kernel trick (as it ensures that the kernel can be used as a valid inner product) and kernels satisfying it are called *Mercer kernels*. Kernels that can be expressed as a function of only the difference between the arguments (i.e. $k(x, x') = \tilde{k}(x - x')$) are known as *stationary kernels*.

In the following example, we fit a non-linear decision boundary to a binary classification problem using the Gaussian kernel, known as the `radial` kernel in this R package.

```
# Load library for support vector machines
library(e1071)

# Load data
load(file = "Data/BlueOrange2.Rdata")

# Fit support vector classifier
svm.bo = svm(as.factor(y) ~ x1 + x2, data = data2,
             kernel = "radial", cost = 10, scale = FALSE)

# Grid of points to be predicted to find decision boundary
x1seq = seq(min(data2$x1), max(data2$x1), length = 200)
```

```r
x2seq = seq(min(data2$x2), max(data2$x2), length = 200)
grid <- expand.grid(x1 = x1seq, x2 = x2seq)

# Plot SVM decision boundary
pred.grid <- as.numeric(predict(svm.bo, grid))
contour(x = x1seq, y = x2seq, z = matrix(pred.grid, nrow=200),
        levels = 1.5, col = "grey", drawlabels = FALSE, lwd = 2,
        main="Support vector machine")
points(data2$x1, data2$x2, bg = c("blue", "orange")[data2$y+1],
       pch = c(21, 22)[data2$y+1], xlab = "", ylab = "")
```



Some kernels (such as the Gaussian) do not correspond to any finite-dimensional set of feature vectors, the basis expansion $h(x) = (h_1(x), h_2(x), \ldots)$ is infinitely long. It is the relative ease and flexibility of specifying a single kernel function in comparison to an infinite-dimensional feature vector that makes kernel methods so appealing.

Given two Mercer kernels k_1 and k_2, one can use them to construct new

kernel functions in a variety of ways. For instance, each of the following are also Mercer kernels:

$$k(x, x') = ck_1(x, x'), \tag{2.65}$$
$$k(x, x') = k_1(x, x') + k_2(x, x'), \tag{2.66}$$
$$k(x, x') = k_1(x, x')k_2(x, x'), \tag{2.67}$$
$$k(x, x') = q\left(k_1(x, x')\right), \tag{2.68}$$
$$k(x, x') = \exp\left(k_1(x, x')\right), \tag{2.69}$$

where $c > 0$, and q is any polynomial with non-negative coefficients.

Note that kernel methods are not just restricted to support vector machines. Recall from Section 2.1.2 that the k-nearest neighbours algorithm finds the k nearest training points using Euclidean distance. Observe that the Euclidean distance between two points x and x' can be written as

$$\begin{aligned} \|x - x'\|_2^2 &= x^\top x - 2x^\top x' + x'^\top x' \\ &= \langle x, x\rangle_2 - 2\langle x, x'\rangle_2 + \langle x', x'\rangle_2 \end{aligned} \tag{2.70}$$

Thus, by replacing the Euclidean inner product $\langle \cdot, \cdot \rangle_2$ by a kernel function, we can neatly generalise the notion of distance that we use and apply the k-nearest neighbour classifier to any data consisting of structured objects for which we can define a kernel.

Chapter 3

Clustering

3.1 Lecture 1: Motivations

These lectures address the topic of clustering and mixture modelling.

Informally, clustering refers to the problem of partitioning a data set into disjoint subsets such that points within the same cluster look like one another, and points within different clusters look different to one another. A priori, this is not a statistical problem (aside from perhaps the use of the word 'data'); one could equally seek to apply clustering methods to deterministic problems. However, clustering methods are widely used in a statistical context, and so it is of interest to first understand their relevance.

3.1.1 'Data Lumping'

A simple example of when clustering might be relevant in a statistical analysis is when data has been collected from a number - say, K - of disjoint sources and then aggregated into a single composite dataset, without labels to indicate the source of each data point. This can happen for quite boring reasons, but it can and does happen, and thus poses a practical challenge. If one could successfully identify which data points originated from the same source as one another, then one can express the data set as a disjoint union of K simpler datasets, each of which has a more homogeneous structure, and is thus easier to work with.

Upon exhibiting this decomposition, it is then often the case that the statistical problem of interest will decouple into e.g. K independent parameter estimation tasks, wherein standard methods can be applied. In this respect, identifying the cluster structure of the data might be viewed as a pre-processing step which expresses the observed data in a more convenient form.

In a more goal-oriented fashion, even if one is primarily interested in prediction tasks (classification, regression, etc.), one observes that decoupling the problem into prediction on K more homogeneous sub-populations can also be expected to improve predictive performance. As such, even if modelling the data is not of direct interest, clustering can be a useful tool.

3.1.2 Modelling of Heterogeneous Data

Limitations of 'Standard' Parametric Families

One of the elementary tasks in statistical estimation is to fit a model to data. In a first course, one typically works with standard parametric families of densities (e.g. Gaussian, Poisson, Gamma, Binomial, etc.), which are convenient in various ways, and are often a good fit for data sources with simple structure. Naturally, these distributions are also quite limited in the phenomena which they can represent. To give a few concrete examples:

- Gaussian distributions are always unimodal, and are thus inappropriate for modelling data sources with a multi-modal structure.
- Poisson distributions have tails which decay super-exponentially fast. This can make them inappropriate for modelling counts of data which exhibit heavier tails.

Indeed, any parametric family with a fixed number of parameters is inherently limited in the type of data which it can model effectively. This is not in itself a bad thing; when these models are suitable, they are particularly good at identifying a tight fit to the data. Any methodology comes with limitations. With this in mind, it is sensible to consider how to model distributions which can provide us with some extra flexibility.

Mixture Modelling

One simple route to this is to model our data as coming from a combination of several simple sources. We might then posit that there are K 'simple' distributions $P_1, \ldots, P_K$ (each of which are of, say, parametric form, i.e. $P_k = P_{\phi_k}$ for some parameter $\phi_k \in \Phi$), such that each of our observed data points was drawn from one of these simple distributions. This is known as a *mixture model*, and can be viewed as a probabilistic interpretation of the intuition which underlies the idea of clustering.

Of course, mixture models themselves also have limitations in which distributions they can represent concisely. Nevertheless, they are a reasonable first step towards more flexible modeling of complex data sources.

3.1.3 Tension in the Mixture Model Problem

Suppose that with our data, we have been given labels which tell us which observations originate from the same component. In this setting, the parameter estimation task is relatively straightforward: for each component, collect the subset of observations which come from that component, and use them to estimate the parameters of that component, using e.g. the method of maximum likelihood:

$$\hat{\phi}_k := \arg\max_{\phi \in \Phi} \sum_{i \in C_k} \log p_\phi \left(x_i \right),$$

As such, one can say informally that given component labels, fitting a mixture model to data is simple.

Separately, suppose that we are told that our data was drawn from a mixture of the components distributions $P_1, \ldots, P_K$, for which we are told the parameters and relative weights π of the components. Given this information, it is again relatively straightforward to estimate which of the components correspond to each data point, by applying Bayes' rule:

$$\mathbf{P}\left(x \text{ comes from cluster } k\right) \propto \pi_k \cdot p_{\phi_k}\left(x\right).$$

One could also use this information to form the 'maximum a posteriori' (MAP) clustering of the data, essentially corresponding to our 'best guess' of the clustering under this information.

$$\hat{k}(x) := \arg\max_{k \in [K]} \pi_k \cdot p_{\phi_k}\left(x\right),$$

where $[K]$ denotes the integers from 1 up to K, i.e. $[K] = 1, 2, \ldots, K$; this notation will be used throughout the section.

That is, given the parameters and weights of a mixture model, clustering the data into its components is also simple.

The challenge of fitting mixture models in practice is that usually, *neither* of these conditions hold: we do not know which data points come from each component, and we do not know what each component looks like. We are thus trying to solve two simple problems simultaneously, which can be far from simple. Similar comments apply to various other formulations of the clustering problem.

3.1.4 Conclusion

It should be emphasised that both clustering and mixture modelling can be quite a challenging task in general, in the sense that obtaining rigorous statistical guarantees on the quality of the resulting model is difficult compared to e.g. parameter estimation in exponential families. Nevertheless, a number of algorithmic solutions have long been observed to work reliably in practice, and there is even a somewhat-rigorous mantra that "clustering works, except when it doesn't matter", i.e. if the data really does exhibit a clustered structure, then many algorithms will successfully recover this structure, and the failure modes of these algorithms correspond to "data which should not be clustered anyways".

The focus of these lectures will thus be to explain how some of these solutions are derived, describe their properties, and explain their implementation. Going forward, the term 'clustering' will be used to denote a partition of of a set into subsets $C_1, \cdots, C_K$ such that for each k, points within C_k are more similar to one another than they are to points in the other C_j. This will be made rigorous

in specific instances, but should be viewed mostly as reflecting how practitioners typically interpret the notion.

For simplicity, we rely on the following standing assumptions for the remainder of the chapter:

- Our observations $\{x_i\}_{i\in[N]}$ take values in $\mathbf{R}^p$.
- Our observations can be decomposed into K clusters, where K is known a priori.
- All clusters have roughly the same geometry, i.e. similar shapes and level of dispersion.

In practice, each of these may fail to hold. Some of these assumptions are relatively easy to adjust to. Others are more challenging to resolve. Hopefully, the specific examples presented in these lectures will shed light on which of these fall into each category.

3.2 Lecture 2: The K-Means Objective, and Lloyd's Algorithm

Recall that one of our first desiderata for clustering is that points in the same cluster should look similar to one another. One simple formalisation of this would be to say that if x and y lie in the same cluster, then it should hold that $\|x-y\|_2^2$ is small. This is the basis of many popular clustering algorithms, explicitly or otherwise.

Before getting into the main part of this section, let me describe some heuristic approaches to clustering which follow this intuition. These are 'algorithm-first' approaches, which are not necessarily developed with a specific objective function in mind. They may nevertheless be sensible.

1. One option: 'merge together' points in the dataset which look similar, until there are not many distinct points left in the data set. For example, one could find

$$(i_*, j_*) := \arg\min\left\{\|x_i - x_j\|_2^2 : i \neq j, x_i \neq x_j\right\},$$

compute $m = \frac{1}{2}\cdot\left(x_{i_*} + x_{j_*}\right)$, and replace each of (x_{i_*}, x_{j_*}) by m. This reduces the number of distinct points by at least one. We can then say that two points in the original data set are in the same cluster if this algorithm eventually 'merges' them into the same point. This is related to the notion of 'hierarchical clustering'.

2. A second option: clustering by 'transport', i.e. for each data point, find its nearest neighbour in the data set, and push it towards this point slightly:

$$\text{for } i \in [N], \quad j_*(i) := \arg\min\left\{\|x_i - x_j\|_2^2 : j \neq i\right\},$$
$$x_i \leftarrow x_i + \varepsilon\cdot(x_{j_*(i)} - x_i).$$

3. A third option: clustering by averaging, i.e. for each data point, identify its M nearest neighbours, and replace the data point by the average of these neighbours, i.e.

$$\begin{aligned} \text{for } i \in [N], \quad S(i) &:= \text{M\_Nearest\_ Neighbours}(x_i; X) \\ x_i &\leftarrow \text{mean}(S_i). \end{aligned}$$

This is related to the notion of 'mean-shift clustering'.

The appeal of these methods is that they are specified directly by their implementation, and so you can 'just do it'; most of the implementations are even relatively straightforward. The difficulty then comes when one wants to do some mathematics, or make some rigorous statements about the algorithms in question. We could want to ask:"what are these algorithms trying to achieve? what does the solution achieve? when is it 'safe' to declare that the algorithm has converged?" and similar, and these are not always so easy to answer, given only an algorithm.

By contrast, many of these questions have straightforward answers when we have an explicit objective function, and an algorithm which seeks to optimise it. We thus might consider an approach which provides such an objective function.

3.2.1 Developing the Objective

An elementary and popular approach to clustering is the so-called *K-Means* objective. This objective seeks to decompose a data set into a collection of disjoint subsets such that any two points in the same cluster lie close to the mean of points within that cluster.

That is, writing $C_k \subseteq [N]$ for the indices of the data points which lie in cluster k, the 'within-cluster-k-sum-of-squares' may be written as

$$\text{WCSS}\left(C_k\right) := \sum_{i \in C_k} \left\| x_i - m\left(C_k\right) \right\|_2^2,$$

where $m\left(C_k\right) := \frac{1}{|C_k|} \sum_{i \in C_k} x_i$ is the centroid of cluster k.

The full loss function can then be written as the sum of these terms over all clusters k, i.e.

$$\begin{aligned} \text{Loss}_1\left(\left\{C_k\right\}_{k \in [K]}\right) &:= \sum_{k \in [K]} \text{WCSS}\left(C_k\right) \\ &= \sum_{k \in [K]} \sum_{i \in C_k} \left\| x_i - m\left(C_k\right) \right\|_2^2. \end{aligned}$$

At present, this is a combinatorial problem, optimising over partitions C, and is not necessarily easy to solve directly.

Remark 1. *A 'principle' here which can be made fairly rigorous: let $f : \mathcal{X} \to \mathbf{R}$ be a 'nice' function which can be written as*

$$f(x) = \min\left\{ g(x, y) : y \in \mathcal{Y} \right\}$$

for some other nice function g. Then

$$\min\left\{f(x) : x \in \mathcal{X}\right\} = \min\left\{g(x,y) : (x,y)\mathcal{X} \times \mathcal{Y}\right\},$$

(i.e. the optimal values of these optimisation problems are equal) and the minimisers of these problems are in correspondence, i.e. if x_* minimises f, then there exists some y_* such that (x_*, y_*) minimises g, and vice versa.

Definition 3.2.1. *Write* $\text{Part}([N], K)$ *for the set of all partitions of* $[N] = \{1, 2, \cdots, N\}$ *into K disjoint subsets, some of which may be empty.*

To simplify our computational task, we note that WCSS is itself the solution to a continuous optimisation problem, namely

$$\text{WCSS}\left(C_k\right) = \min_{\mu_k \in \mathbf{R}^p} \sum_{i \in C_k} \|x_i - \mu_k\|_2^2,$$

with minimiser $\mu_k = m\left(C_k\right)$. As such, our earlier 'principle' suggest that one could equally aim to minimise the extended loss function

$$\text{Loss}_2\left(\left\{C_k\right\}_{k\in[K]}, \left\{\mu_k\right\}_{k\in[K]}\right) := \sum_{k\in[K]} \sum_{i \in C_k} \|x_i - \mu_k\|_2^2.$$

Definition 3.2.2. *For K a positive integer, define*

$$\text{OneHot}(K) = \{z \in \{0,1\}^K : z_k = 1 \text{ for exactly one value of } k \in [K]\}.$$

Lemma 8. *For positive integers (N, K), there is a bijection between* $\text{Part}([N], K)$ *and* $\text{OneHot}(K)^N$.

Proof. (Sketch) For one direction, let $z_{i,k} = 1$ if and only if i is in the kth cluster in the partition. For the other direction, let $C_k = \{i \in [N] : z_{i,k} = 1\}$. Verify that these two maps are mutual inverses. □

We will thus find that it is useful to encode the clustering C by introducing the 'label' variables $z_{i,k} = \mathbf{I}\left[i \in C_k\right]$. Abstracting away the clustering variable C, we can equivalently say that each vector $z_i = \left(z_{i,k}\right)_{k\in[K]}$ lies in the set $\text{OneHot}(K)^N$, leaving us to solve

$$\begin{aligned} \min \text{Loss}_3\left(\left\{z_i\right\}_{i\in[N]}, \left\{\mu_k\right\}_{k\in[K]}\right) &:= \sum_{k\in[K]} \sum_{i\in[N]} z_{i,k} \cdot \|x_i - \mu_k\|_2^2 \\ \text{such that} \quad \left\{z_i\right\}_{i\in[N]} &\subseteq \text{OneHot}(K), \\ \left\{\mu_k\right\}_{k\in[K]} &\subseteq \mathbf{R}^p. \end{aligned}$$

It bears mentioning that the K-Means problem is 'model-free', in the sense that the objective is not explicitly derived from some probabilistic model of the data. We have simply posited an (intuitively sensible) loss function, and hope that by minimising this function, we will learn something about the structure

of the data. This also suggests that this approach could remain appropriate for 'deterministic' clustering tasks, in which we are not prepared to model the data directly, but are still interested in understanding its structure.

Let us make some rigorous statements about these claimed equivalences.

Proposition 9. *The loss functions are related as follows:*

1. $\text{Loss}_1\left(\left\{C_k^1\right\}_{k\in[K]}\right) = \min_\mu \text{Loss}_2\left(\left\{C_k\right\}_{k\in[K]}, \left\{\mu_k\right\}_{k\in[K]}\right)$, *and the minimiser is attained at* $\mu_k = m(C_k)$.
2. *Under the bijection* $C_k = \{i \in [N] : z_{i,k} = 1\}$, $z_{i,k} = \mathbf{I}[i \in C_k]$, *it holds that* $\text{Loss}_2\left(\left\{C_k\right\}_{k\in[K]}, \left\{\mu_k\right\}_{k\in[K]}\right) = \text{Loss}_3\left(\left\{z_i\right\}_{i\in[N]}, \left\{\mu_k\right\}_{k\in[K]}\right)$.

In particular, the minimal values of all three loss functions are identical, and given a minimising input to any loss function, one can obtain a minimising input for both of the other loss functions.

Some other perspectives on the K-Means objective

One can also motivate the K-Means objective in a couple of other ways. For example, going back to our original motivation for clustering, one could consider evaluating the quality of a given cluster by computing the 'pairwise' within-cluster sum-of-squares, i.e.

$$\text{pWCSS}\left(C_k\right) := \sum_{i\in C_k}\sum_{j\in C_k}\left\|x_i - x_j\right\|_2^2,$$

It is a useful exercise to convince oneself that $\text{pWCSS}\left(C_k\right)$ and $\text{WCSS}\left(C_k\right)$ are the same, up to a multiplicative factor of $2\cdot|C_k|$. Thus, even if we originally sought to demand that the elements of a cluster be close to one another in a pairwise sense, it is equivalent to demand that they all be close to a 'cluster centroid', i.e. $m(C_K)$. Indeed, these conditions are somehow 'morally equivalent', by the triangle inequality: if all points are within a distance δ of m, then all points must be within a distance 2δ of each other.

With this in mind, we might re-frame our clustering goal as demanding that for each cluster C, there exists a point $m \in \mathbf{R}^p$, such that for all points $x \in C$, it holds that $x \approx m$.

Methods which seek to satisfy this demand are known variously as 'prototype methods', '(vector) quantisation methods', and also have an interpretation in terms of data set compression.

Compression Interpretation

The compression interpretation is as follows: if one wants to exactly communicate the data set X, one needs to write down N vectors, each of length P, giving a 'complexity' (broadly interpreted) of $\sim N \times P$. However, we could coarsely summarise our dataset by writing down the cluster centroids (i.e. K vectors

which are each of length P), and the number of data points belonging to each cluster component, which is a list of K integers. This would then have a total complexity of $\sim K \times (P+1)$.

Under this interpretation, we can quantify how much information is lost through this compression, i.e. the reconstruction error

$$\text{Reconstruction\_Error} := \sum_{i \in [N]} \|x_i - m_{k(i)}\|_2^2,$$

where $k(i)$ denotes the cluster component to which x_i belongs. This is another way of writing the K-Means objective.

Remark 2. *Note also that methods of this form are somehow an extreme form of dimension reduction: instead of projecting the data onto a lower-dimensional subspace, we are essentially projecting it onto a finite set (which, non-rigorously, is somehow a 0-dimensional subspace). This is to be contrasted with the upcoming section on dimension reduction methods.*

Invariance in the K-Means Objective

Under this heading, I will abbreviate $\text{Loss}_3\left(\{z_i\}_{i\in[N]}, \{\mu_k\}_{k\in[K]}\right)$ to $\text{Loss}(Z, \mu; X)$, where the X emphasises the dependence on the data set (which is otherwise not emphasised in these notes). Given a function $f : \mathbf{R}^p \to \mathbf{R}^p$, I will abuse notation to write $f(X)$ for $\{f(x_i) : i \in [N]\}$ and $f(\mu) = \{f(\mu_k) : k \in [K]\}$.

It can be useful to understand how the K-Means objective changes when we pre-process our data by transforming it in some way. Similarly, it is useful to understand which transformations do not affect the objective, i.e. to answer the question 'under which transformations is the K-Means problem invariant?'.

Example 1. *Let $c \in \mathbf{R}^p$ be a fixed vector, and consider the function $f(x) = x+c$. It then holds that*

$$\text{Loss}(Z, f(\mu); f(X)) = \text{Loss}(Z, \mu; X).$$

It follows that the K-means problems for the data sets X and $f(X)$ are essentially equivalent.

Example 2. *Let $\lambda \in (0, \infty)$ be a positive scalar, and consider the function $f(x) = \lambda \cdot x$. It then holds that*

$$\text{Loss}(Z, f(\mu); f(X)) = \lambda^2 \cdot \text{Loss}(Z, \mu; X).$$

It again follows that the K-means problems for the data sets X and $f(X)$ are essentially equivalent, though this time we pick up a scaling factor in the loss function.

Example 3. *Let $Q \in \mathbf{R}^{p \times p}$ be a fixed orthogonal matrix satisfying $Q^\top Q = I_p$. This ensures that the map $x \mapsto Qx$ preserves distance between points (we might call this a 'linear isometry'). Mappings of this form are typically multi-dimensional versions of rotations and reflections in a hyperplane.*

Consider now the function $f(x) = Qx$. It then holds that

$$\text{Loss}(Z, f(\mu); f(X)) = \text{Loss}(Z, \mu; X).$$

It follows once more that the K-means problems for the data sets X and $f(X)$ are essentially equivalent.

As might be expected, any composition of the above maps will also leave the problem invariant, e.g. if we had $f(x) = \lambda \cdot (Qx + c)$, then clustering $f(X)$ would also be equivalent to clustering X, in a suitable sense.

However, for general transformations of the data (even linear ones), things can change. An instructive exercise is the following: construct a data set X and a linear / nonlinear map f so that applying f to X changes the optimal clustering. In the linear case, I recommend focusing on $p = 2$ (so that you can visualise the solution), $K = 2$ (to keep things simple), and let f be a diagonal linear map, i.e. $(x, y) \mapsto (a \cdot x, b \cdot y)$ for some $a, b \in \mathbf{R}$.

The key observation is then that K-Means receives a metric space and data from that space; if the data is transformed in a way which neglects the metric structure of the space, then the objective will generally change in a meaningful way.

There is also a separate invariance which is not related to the *data*, but to how we *label* our clusters. Let $\text{Symm}([K])$ denote the set (group) of permutations of the set $[K]$, which we will interpret as 're-labelings' of our clusters. Define an action of $\text{Symm}([K])$ on cluster centroids by

$$(\sigma \cdot \mu)_k = \mu_{\sigma(k)}$$

and similarly, an action on cluster indicator labels by

$$(\sigma \cdot Z)_{i,k} = z_{i,\sigma(k)}.$$

One can then verify that

$$\text{Loss}((\sigma^{-1} \cdot Z), \sigma \cdot \mu; X) = \text{Loss}(Z, \mu; X),$$

i.e. if you re-label the clusters, and re-label the cluster indicators in the corresponding way, then you have a clustering of equal quality. This is known as the 'label-switching' invariance.

3.2.2 Optimisation of the K-Means Objective

Now, while the minimisation of Loss_3 still involves a combinatorial search problem over the variables $\{z_i\}_{i \in [N]}$, there is a key simplification which is enabled by this framing: if we hold $\{\mu_k\}_{k \in [K]}$ fixed, then one can solve

$$\begin{aligned} & \min \mathrm{Loss}_3 \left(\{z_i\}_{i\in[N]}, \{\mu_k\}_{k\in[K]} \right) \\ \text{such that} \quad & \{z_i\}_{i\in[N]} \subseteq \mathrm{OneHot}(K)^N, \end{aligned}$$

in *closed form*. In particular, writing

$$k_i = \arg\min_{k\in[K]} \|x_i - \mu_k\|_2^2,$$

(and breaking any ties arbitrarily), it holds that the minimising value of z_i satisfies

$$z_{i,k} = \begin{cases} 1 & k = k_i \\ 0 & \text{else.} \end{cases}$$

That is, holding the cluster centroids fixed, the optimal clustering is to assign each data point to cluster which contains the nearest centroid. It bears mentioning that the problem here is *separable*, i.e. when updating z_i, we can ignore what is happening for the other vectors z_j. (***)

Separately, we also know that if we hold the variables $\{z_i\}_{i\in[N]}$ fixed, then the problem

$$\begin{aligned} & \min \mathrm{Loss}_3 \left(\{z_i\}_{i\in[N]}, \{\mu_k\}_{k\in[K]} \right) \\ \text{such that} \quad & \{\mu_k\}_{k\in[K]} \subseteq \mathbf{R}^p \end{aligned}$$

is easily solved by setting

$$\mu_k = \begin{cases} \frac{\sum_{i\in[N]} z_{i,k}\cdot x_i}{\sum_{i\in[N]} z_{i,k}} & \sum_{i\in[N]} z_{i,k} \neq 0 \\ (\text{anything}) & \text{else.} \end{cases}$$

In words, the optimal collection of cluster centroids is obtained by computing the arithmetic mean of the data points in each cluster. Again, the problem is separable; in updating μ_k, we can forget about what is happening with the other cluster centroids μ_ℓ.

It is thus natural to propose an algorithm which aims to minimise Loss_3 by a block alternating minimisation scheme, in which one iteratively holds either the $\{z_i\}_{i\in[N]}$ or $\{\mu_k\}_{k\in[K]}$ block of variables fixed, and optimises over the other block. As applied to minimisation of the K-Means objective, this is known as *Lloyd's Algorithm*, though it is often informally (and erroneously) referred to simply as the K-Means Algorithm. Indeed, if you want somebody to know what you are talking about, you are probably slightly better off describing it as the K-Means algorithm.

Algorithm 9: Lloyd's Algorithm with data $\{x_i\}_{i=1}^N$ and K clusters.

1. Initialise $\mu_1, \ldots, \mu_K$ with distinct values.
2. Until convergence, do
   (a) For $i \in [N]$, set $K_i = \arg\min_{k \in [K]} \|x_i - \mu_k\|_2^2$, let k_i be the minimal element of K_i.
      - For $k \in [K]$, set $z_{i,k} = 1$ for $k = k_i$, and $z_{i,k} = 0$ otherwise.
   (b) For $k = 1, \ldots, K$, set $\mu_k = \frac{\sum_{i \in [N]} z_{i,k} \cdot x_i}{\sum_{i \in [N]} z_{i,k}}$ if $\sum_{i \in [N]} z_{i,k} > 0$, and set μ_k to its previous value otherwise.

An important aspect of Lloyd's algorithm is that it is guaranteed to converge to a fixed point in a finite number of steps. One can see this by revisiting the original combinatorial form of the clustering problem: there are only finitely many possible clusterings, and over the course of the algorithm, we always move towards better clusterings (in terms of objective value). If the algorithm did not converge, then we would have an infinite sequence of distinct, decreasing objective values. Since the set of clusterings is finite, this would lead to a contradiction.

A related remark is that there is an obvious termination criterion for Lloyd's algorithm: stop iterating once you obtain the same clustering for two successive iterations.

Some key advantages of the K-Means objective and its minimisation via Lloyd's Algorithm are the following:

- There is indeed a valid objective function which is being minimised. As such, if one really believes that making $\text{Loss}_1(C)$ small will provide a scientifically meaningful clustering of the data, then this approach is well-founded. This is to be contrasted with more heuristic approaches to clustering.
  - Note, however, that K-Means does not explicitly model the data distribution. It is a well-specified optimisation problem, but does not explicitly model the data as possessing a particular structure (though of course, its practical success will depend intimately on its structure).
- Lloyd's Algorithm leads to non-increasing values of the objective function, and so running the Algorithm for more iterations can never make things worse. Again, this is to be contrasted with more heuristic approaches.
- Moreover, Lloyd's Algorithm always terminates in finite time, returns a true local minimiser, and 'tells you when it is finished' with certainty. This is to be contrasted with continuous minimisation algorithms, which can appear to get stuck in 'flat' regions, and lead the user to spuriously declare that convergence has occurred.

- Each iteration of Lloyd's Algorithm has cost which is of polynomial size in each of (N, K, p) (actually, of order $N \times K \times p$). As such, the algorithm has a bearable computational cost, assuming that not too many iterations are required for convergence.

Some potential drawbacks and limitations include:

- The use of the squared distance as part of the K-Means objective suggests that the resulting clusterings may not be particularly robust to e.g. the presence of outliers.
- The initialisation of the initial cluster centroids can have substantial impact on the practical quality of the resulting clusterings. As such, initialisation should be treated carefully.
- There is no guarantee that any individual run of Lloyd's Algorithm will find the global minimiser of the loss function. Relatedly, there is no guarantee that any two runs of Lloyd's Algorithm from distinct initialisations will find the same local minimiser. As such, it is common practice to run multiple independent instances of Lloyd's Algorithm, to diagnose the robustness of its findings.
  - In practice, this usually amounts to running Lloyd's algorithm several times, and returning the best solution from those runs. However, one can imagine using these runs to estimate the variability of the algorithm's output, and using this estimate to decide how many runs are appropriate.

Some simple extensions of the K-Means objective which could be discussed: [Not examinable!]

- One can imagine replacing every instance of terms like $\|x - \mu\|_2^2$ in the objective with other functions of $\|x - \mu\|_2$, or even with functions of $\mathsf{d}(x, \mu)$ for some other metric d. Consider why such modifications might be of interest, and how Lloyd's algorithm might be modified to adjust for them (at perhaps moderately increased computational complexity).
  - A simple instance of this idea would be to apply a linear transformation to the data (so as to e.g. make the scale of each coordinate the same) and solve the K-Means problem for this data. This would equate to defining d to be a certain Mahalanobis distance.
- In much the same way that LASSO replaces the combinatorial problem of variable selection with a convex relaxation, the same can be done for K-Means. The so-called 'Sum-of-Norms' clustering objective with regularisation parameter $\lambda > 0$ is written as

$$\mathrm{Loss}_{\mathrm{SoN}}\left(\{\mu_i\}_{i\in[N]}\right) := \sum_{i\in[N]} \|x_i - \mu_i\|_2^2 + \lambda \cdot \sum_{i\in[N]} \sum_{j\in[N]} \left\|\mu_i - \mu_j\right\|_2 .$$

By analogy with the regularisation which is used in LASSO, consider why the optimal set of $\{\mu_i\}_{i\in[N]}$ might be expected to contain much fewer than N distinct elements, particularly for large values of λ (it might help to focus on the case $p = 1$, so that $\left\|\mu_i - \mu_j\right\|_2 = \left|\mu_i - \mu_j\right|$). Consider also why this objective function might be unfriendly or expensive to optimise, despite its convexity.

- Writing $Z = \left(z_{i,k}\right)_{i\in[N],k\in[K]} \in \{0,1\}^{N\times K}$, $\mu = \left[\mu_1 \mid \cdots \mid \mu_K\right]$ one sees that Loss_3 can be written in the form

$$\begin{aligned} &\min \left\|X - Z\mu^\top\right\|_{\text{F}}^2 \\ \text{such that } \; &Z \in \{0,1\}^{N\times K} \\ &\mu \in \mathbf{R}^{P\times K}, \end{aligned}$$

where $\|\cdot\|_{\text{F}}$ is the Frobenius norm for matrices, given by its element-wise ℓ^2 norm, i.e. $\|A\|_{\text{F}}^2 = \sum_{i,j}\left|A_{i,j}\right|^2$, with the additional constraint that the rows of Z each be one-hot. That is, the K-Means objective has the form of a low-rank matrix approximation problem, subject to constraints. One can imagine that this might provide useful insights concerning e.g. convex relaxations of the clustering problem. I mention briefly that many matrix problems with this character are known to be i) non-convex, but ii) still often solvable by simple iterative algorithms.

3.3 Lecture 3: The Gaussian Mixture Model, and the EM Algorithm

3.3.1 From Hard Clustering to Soft Clustering

The K-Means approach might be described as generating a 'hard' clustering, inx the sense that each point is assigned to exactly one cluster. Depending on the application, clusterings of this form may be associated with some level of overconfidence in the clustering, and thus lead to compromised estimation quality in downstream tasks. Equally, when considering data which is generated from a mixture of several simple distributions, it is not always the case that the individual component distributions are non-overlapping, and so individual data points *should* retain some uncertainty over their cluster component of origin, even in the presence of large amounts of data.

In principle, it could thus be useful to instead perform a 'soft' clustering, wherein we instead maintain a *probability distribution* over the cluster from which each data point came. An additional benefit of this is that the estimates of the cluster parameters may be better-calibrated, as the fitting procedure should now use information from all data points, rather than just the subset of points currently assigned to that cluster.

Accepting for the moment that soft clustering might be useful, we now need to design principled and practical algorithms which can allow us to deliver such an output. In what follows, given a base space $\mathcal{Y}$, we will write $\mathcal{P}(\mathcal{Y})$ for the set of probability distributions with base space $\mathcal{Y}$.

3.3.2 Some Generalities on Mixture Models

Definition 3.3.1. *For $K \in \mathbf{N}$, define the $(K-1)$-simplex as*

$$\Delta^{K-1} = \left\{ z \in [0,1]^K : \sum_{k \in [K]} z_k = 1 \right\}.$$

Given a parametric family of probability distributions

$$\mathcal{P} = \{P_\phi : \phi \in \Phi\},$$

define the class of mixture models over $\mathcal{P}$ with K components as

$$\mathrm{Mix}_K(\mathcal{P}) = \left\{ \sum_{k \in [K]} \pi_k P_{\phi_k} : \pi \in \Delta^{K-1}, \phi_1, \ldots, \phi_K \in \Phi \right\},$$

and the class of all finite mixture models over $\mathcal{P}$ as

$$\mathrm{Mix}(\mathcal{P}) = \bigcup_{K \geqslant 1} \mathrm{Mix}_K(\mathcal{P}). \tag{3.1}$$

In what follows, we will be interested in fitting models from $\mathrm{Mix}_K(\mathcal{P})$ to data, for specific choices of K and $\mathcal{P}$.

3.3.3 Gaussian Mixture Models

One common way to model data for which soft clustering is appropriate is through *mixture models*. In these notes, we focus initially on spherical Gaussian mixture models (GMM) with known variance, i.e. each component of the mixture has the form

$$P_k(x) = \mathcal{N}\left(x \mid \mu_k, \sigma^2 I_d\right),$$

where σ^2 is known and does not depend on k. We can refer to this family as $\mathcal{P} = \mathrm{SphrGauss}(p, \sigma^2)$. We will also allow for the setting of non-uniform component probabilities.

Writing $\theta = \left\{ \{\mu_k\}_{k \in [K]}, \{\pi_k\}_{k \in [K]} \right\} \in \Theta := (\mathbf{R}^p)^K \times \Delta^{K-1}$, we thus define the spherical Gaussian mixture model with K components and variance σ^2 in dimension p by

$$p_\theta(x) = \sum_{k \in [K]} \pi_k \cdot \mathcal{N}\left(x \mid \mu_k, \sigma^2 \cdot I_p\right).$$

One can imagine generating a sample from p_θ by drawing an index $z \sim \text{Categorical}\left(\{\pi_k\}_{k\in[K]}\right)$, and then drawing the sample $x \sim \mathcal{N}\left(x \mid \mu_z, \sigma^2 \cdot I_p\right)$. As such, it is a neat model for heterogeneous data, wherein individual data points can either be quite similar to one another (if they were generated with the same value of k), or quite different (if not). This meshes nicely with our abstract goals for clustering. In particular, if we can successfully fit a mixture model to our data, we can interpret each of the components of the mixture as genuine clusters.

Towards soft clustering, we will assume that our data was drawn iid from some spherical GMM with parameter θ, and attempt to estimate θ from the data. We can thus estimate the cluster centroids by the $\{\mu_k\}_{k\in[K]}$, and apply Bayes rule to estimate the probabilities of each data point lying in each cluster.

It bears mentioning that GMMs are a prototypical example of a *non-identifiable* model, in the sense that there exist parameter values $\theta_1 \neq \theta_2$ such that $p_{\theta_1} \equiv p_{\theta_2}$; this is easiest to see by considering permutations of the indices k. Nevertheless, for the purposes of clustering this will not necessarily be an issue: while the parameters θ are non-identifiable, the induced clusterings typically are. As such, we do not linger on this point further.

(A subtler point is that non-identifiability can also lead to slower rates of estimation for e.g. the centroids μ_k; we also do not dwell on this, as for mixtures with well-separated components, this is typically less of an issue. This whole comment is, of course, non-examinable)

3.3.4 Parameter Estimation in GMMs

When considering how to estimate the parameters θ of a statistical model, a natural first port of call is to consider maximum likelihood estimation, i.e. to solve

$$\max_{\theta\in\Theta} \sum_{i\in[N]} \log p_\theta\left(x_i\right).$$

For GMMs, this poses some difficulties. To begin with, the MLE problem is certainly not solvable in closed form. Moreover, the negative log-likelihood function which maps θ to $\sum_{i\in[N]} -\log p_\theta\left(x_i\right)$ is not convex in the parameters θ, and so the use of local iterative methods (e.g. gradient descent, Newton's method, etc.) for optimising this objective cannot be recommended without qualification (i.e. we cannot guarantee that it will converge well, to good solutions). We thus seek to use the structure of our statistical model to construct a new objective function with more favourable optimisation properties.

A first observation is that GMMs admit a so-called *latent variable representation*. That is, while $p_\theta\left(x\right)$ happens to admit an explicit form, there is an even more convenient representation of the GMM which involves unobserved or 'latent' variables.

Indeed, recall the algorithmic presentation of the GMM:

1. Draw an index $z \sim \text{Categorical}\left(\{\pi_k\}_{k\in[K]}\right)$, and
2. Draw a sample $x \sim \mathcal{N}\left(x \mid \mu_z, \sigma^2 \cdot I_p\right)$.

We can thus write down the joint distribution of (z, x) as

$$p_\theta(z, x) := \text{Cat}\left(z \mid \pi\right) \cdot \mathcal{N}\left(x \mid \mu_z, \sigma^2 \cdot I_p\right).$$

We then see that i) the law of z lies in an exponential family, and ii) for each z, the conditional law of x given z lies in an exponential family. Given the general simplicity of estimation in exponential families, this is a good sign that we have simplified things. This is related to the observation that if we had also observed the labels z, then our statistical problem would be simple.

This provides the marginal representation

$$p_\theta(x) = \sum_{z\in[K]} \text{Cat}\left(z \mid \pi\right) \cdot \mathcal{N}\left(x \mid \mu_z, \sigma^2 \cdot I_p\right),$$

or more suggestively,

$$p_\theta(x) = \int \text{Cat}\left(z \mid \pi\right) \cdot \mathcal{N}\left(x \mid \mu_z, \sigma^2 \cdot I_p\right).$$

The point here is not that we are computing an integral in the sense of e.g. Riemann integration, but that we are *marginalising out* our latent variable z. It will turn out that this formulation will generalise nicely to some other examples.

Given a latent variable representation of this form, we will try to recast the problem of Maximum Likelihood Estimation in a more favourable form.

Some Examples of Latent Variable Models

The usual simple examples of latent variable models (LVMs) tend to exhibit a sort of two-step structure, i.e. first, generate the unobserved variable z, and then second, use z to generate the observation x.

1. General finite mixtures (i.e. models in $\text{Mix}_K(\mathcal{P})$ for arbitrary $\mathcal{P}$ basically have this structure; z is the label of the component, and x is then drawn from the corresponding component.
2. One can also consider infinite mixtures with continuous labels, e.g. *scale mixture* models. For example, one can imagine generating draws from a Gaussian distribution with a random variance (this is a simple way of generating observations with heavier-than-Gaussian tails). This would correspond to drawing a random $z > 0$, and then sampling $x \sim \mathcal{N}\left(x \mid 0, z\right)$.
3. A more involved example is the *Hidden Markov Model*, in which z is given by some signal which is evolving in time, satisfying the Markov property (think of e.g. a random walk on a graph), and x is some indirect, noisy observation of the signal (e.g. look at the current state of the random walk,

and observe a randomly-chosen neighbouring state). These models have applications in weather prediction, language modelling, target tracking, and well beyond.

Latent variable models are a broadly popular way of injecting additional flexibility and robustness into a statistical model. By adding in extra variables which can explain the variability in observed data, one can handle strange observations and misspecified models more gracefully. Additionally, inferring the unobserved variables can be qualitatively useful for understanding the structure of one's data and model.

3.3.5 Maximum Likelihood Estimation in Latent Variable Models

In this section, we will speak about models more general than the GMM, though with frequent reference to the GMM, in order to keep things concrete. Throughout, we will work under a statistical model p_θ, for which we hope to compute the maximum likelihood estimator of θ under the data, i.e. to solve

$$\max_{\theta \in \Theta} \sum_{i \in [N]} \log p_\theta \left(x_i \right).$$

Suppose that for some space $\mathcal{Z}$, for each $\theta \in \Theta$, one can define a joint distribution on $\mathcal{X} \times \mathcal{Z}$ such that our statistical model p_θ admits the latent variable representation

$$p_\theta \left(x \right) = \int p_\theta \left(x, z \right) \, \mathrm{d}z$$

Remark 3. *Given two probability distributions p and q with the same support (more or less), a 'change of measure' refers to the procedure of expressing an expectation of one function with respect to p as an expectation of another function with respect to q, obtained in the following way:*

$$\begin{aligned}
\mathbf{E}_p[f(X)] &= \int p(x) \cdot f(x) \, \mathrm{d}x \\
&= \int p(x) \cdot \frac{q(x)}{q(x)} \cdot f(x) \, \mathrm{d}x \\
&= \int q(x) \cdot \frac{p(x)}{q(x)} \cdot f(x) \, \mathrm{d}x \\
&=: \int q(x) \cdot w(x) \cdot f(x) \, \mathrm{d}x \\
&= \mathbf{E}_q[w(X) \cdot f(X)],
\end{aligned}$$

where $w(x) = p(x)/q(x)$. Usually, this is interesting or relevant when p is in some way difficult to work with, and q is in some way simpler. Of course, there is no free lunch, since w will contain some complexity as well.

Writing $q(z)$ for some suitable probability distribution in z-space, perform a simple change of measure to write

$$\begin{aligned} p_\theta(x) &= \int p_\theta(x,z)\,\mathrm{d}z \\ &= \int q(z)\cdot\frac{p_\theta(x,z)}{q(z)}\,\mathrm{d}z, \end{aligned}$$

which can be understood as the expectation of the ratio $w(x,z,\theta) := \frac{p_\theta(x,z)}{q(z)}$ under $z \sim q$.

Remark 4. *Recall Jensen's inequality: for any random variable X and any concave function F, it holds that*

$$F\left(\mathbf{E}[X]\right) \geqslant \mathbf{E}\left[F(X)\right].$$

Moreover, equality holds when X is constant. ('Jensen's inequality' is also used to refer to the corresponding statement for convex F).

Proof. By concavity, for all x, y, it holds that $F(y) \leqslant F(x) + F'(x)(y-x)$. Take $x = \mathbf{E}[X], y = X$, and take expectations on both sides. □

Remark 5. *Recall that the Kullback-Leibler divergence (also 'relative entropy') between distributions p and q (on the same space) has the form*

$$\mathrm{KL}(p,q) = \int p(x)\cdot\log\left(\frac{p(x)}{q(x)}\right)\mathrm{d}x$$

and satisfies $\mathrm{KL}(p,q) \geqslant 0$, with equality holding when $p = q$.

Noting that $w \mapsto \log w$ is concave and applying Jensen's inequality, we may thus write that

$$\log p_\theta(x) \geqslant \int q(z)\cdot\log\left(\frac{p_\theta(x,z)}{q(z)}\right)\mathrm{d}z.$$

By inspection, one sees that this lower bound is tight precisely when $q(z) = p_\theta\left(z \mid x\right)$; indeed, one can write

$$\begin{aligned} \int q(z)\cdot\log\left(\frac{p_\theta(x,z)}{q(z)}\right)\mathrm{d}z &= \int q(z)\cdot\left\{\log\left(\frac{p_\theta\left(z\mid x\right)}{q(z)}\right)+\log p_\theta(x)\right\}\mathrm{d}z \\ &= -\int q(z)\cdot\log\left(\frac{q(z)}{p_\theta\left(z\mid x\right)}\right)\mathrm{d}z + \log p_\theta(x) \\ \rightsquigarrow \quad \log p_\theta(x) &= \int q(z)\cdot\log\left(\frac{p_\theta(x,z)}{q(z)}\right)\mathrm{d}z + \mathrm{KL}\left(q(z), p_\theta\left(z\mid x\right)\right), \end{aligned}$$

where the first equality follows from Bayes' rule and $\log(ab) = \log a + \log b$, the second inequality follows from $\log(1/r) = -\log r$, and the final line follows from rearrangement.

As such, for any (θ, x), one can write that

$$\log p_\theta(x) = \max_{q \in \mathcal{P}(\mathcal{Z})} \int q(z) \cdot \log\left(\frac{p_\theta(x,z)}{q(z)}\right) \mathrm{d}z,$$

where $\mathcal{P}(\mathcal{Z})$ denotes the set of probability distributions over the space $\mathcal{Z}$.

Now, recall the original MLE problem, namely, solving

$$\max_{\theta \in \Theta} \sum_{i \in [N]} \log p_\theta(x_i).$$

For each i, our earlier manipulations allow for us to write that

$$\log p_\theta(x_i) = \max_{q_i \in \mathcal{P}(\mathcal{Z})} \int q_i(z_i) \cdot \log\left(\frac{p_\theta(x_i,z_i)}{q_i(z_i)}\right) \mathrm{d}z_i.$$

As such, the MLE problem

$$\max_{\theta \in \Theta} \sum_{i \in [N]} \log p_\theta(x_i).$$

can be expanded to solving

$$\begin{aligned} &\max \mathcal{F}\left(\theta, \{q_i : i \in [N]\}\right) \\ \text{such that} \quad &\theta \in \Theta,\ \{q_i\}_{i \in [N]} \subseteq \mathcal{P}(\mathcal{Z}), \end{aligned}$$

where we define

$$\mathcal{F}\left(\theta, \{q_i : i \in [N]\}\right) := \sum_{i \in [N]} \int q_i(z_i) \cdot \log\left(\frac{p_\theta(x_i,z_i)}{q_i(z_i)}\right) \mathrm{d}z_i.$$

To formalise these manipulations, we could say the following:

Proposition 10. *Defining*

$$\begin{aligned} \text{MLE\_Objective}(\theta) &:= \sum_{i \in [N]} \log p_\theta(x_i) \\ \mathcal{F}\left(\theta, \{q_i\}_{i \in [N]}\right) &= \sum_{i \in [N]} \int q_i(z_i) \cdot \log\left(\frac{p_\theta(x_i,z_i)}{q_i(z_i)}\right) \mathrm{d}z_i, \end{aligned}$$

it holds that

$$\begin{aligned} &\text{MLE\_Objective}(\theta) = \max \mathcal{F}\left(\theta, \{q_i\}_{i \in [N]}\right) \\ &\textit{such that } \{q_i\}_{i \in [N]} \subseteq \mathcal{P}(\mathcal{Z}), \end{aligned}$$

and the maximum is achieved by taking $q_i(z_i) = p(z_i \mid x_i, \theta)$ for each $i \in [N]$. In particular, the maximal values of both objective functions are identical, and given a maximising input to either objective function, one can obtain a maximising input for the other objective function.

3.3.6 The Expectation-Maximisation (EM) Algorithm

As in the case of the K-Means objective, we have started with a simple objective in few variables, but which is difficult to optimise, and manipulated our problem into a form which contains even more variables. For this to be of value, we must demonstrate that this extended objective function admits some practical simplifications.

To this end, we make the following assumption:

1. For any $\{q_i\}_{i\in[N]} \subseteq \mathcal{P}(\mathcal{X})$, the function

$$\begin{aligned} Q\left(\cdot \mid \{q_i\}_{i\in[N]}\right) : \Theta &\to \mathbf{R} \\ &: \theta \mapsto \sum_{i\in[N]} \int q_i(z_i) \cdot \log p_\theta(x_i, z_i)\, \mathrm{d}z_i \end{aligned}$$

   may be maximised efficiently.

In practice, our q_i will usually take the form $p_{\tilde{\theta}}(z_i \mid x_i)$, but in the case of joint exponential families, this function will often be 'nice' under even greater generality.

While this appears to be a strong assumption a priori, it is commonly verified in practice. The most common scenario in which this holds is when $p_\theta(x, z)$ lies in a joint exponential family. We will see that for the GMM in particular, this assumption does indeed hold.

Under this assumption, one can endeavour to maximise $\mathcal{F}\left(\theta, \{q_i\}_{i\in[N]}\right)$ by a similar strategy to Lloyd's Algorithm; namely, applying a block alternating optimisation scheme. By our earlier discussion, we know that when θ is held fixed at $\tilde{\theta}$, it is optimal to set $q_i(z_i) = p_{\tilde{\theta}}\left(z_i \mid x_i\right)$. Holding q_i fixed at this value, $\mathcal{F}$ takes the form

$$\begin{aligned} \mathcal{F}\left(\theta; \tilde{\theta}\right) &:= \sum_{i\in[N]} \int p_{\tilde{\theta}}\left(z_i \mid x_i\right) \cdot \log\left(\frac{p_\theta(x_i, z_i)}{p_{\tilde{\theta}}\left(z_i \mid x_i\right)}\right) \mathrm{d}z_i \\ &= \sum_{i\in[N]} \int p_{\tilde{\theta}}\left(z_i \mid x_i\right) \cdot \log p_\theta(x_i, z_i)\, \mathrm{d}z_i + \tilde{\mathcal{F}}\left(\tilde{\theta}, \{x_i\}_{i\in[N]}\right), \end{aligned}$$

where $\tilde{\mathcal{F}}$ has no dependence on θ. The first term here is known as the 'expected log-likelihood' of θ, where the implied expectation is taken with respect to z.

It hence follows from our earlier assumption that maximising $\mathcal{F}$ with respect to θ with the $\{q_i\}_{i\in[N]}$ held fixed can be implemented efficiently.

This procedure of iteratively holding either the $\{q_i\}_{i\in[N]}$ or θ block of variables fixed, and optimising over the other block is known as the Expectation-Maximisation or *EM Algorithm*. The name is motivated by the fact that fixing the $\{q_i\}_{i\in[N]}$ allows for the terms in $\mathcal{F}$ to be expressed as integrals (expectations / expected log-likelihoods) which are then maximised as a function of θ. It bears mentioning that a key reason for the popularity of this algorithm is that for many problems of practical interest, the steps of the algorithm can often be implemented in closed form, without the need for any tuning parameters, step-sizes, etc.

Algorithm 10: EM Algorithm with data $\{x_i\}_{i=1}^N$, joint model $p_\theta(x, z)$

1. Initialise $q_1, \ldots, q_N$, $\hat{\theta}$.
2. Until convergence of $\hat{\theta}$, do
   (a) For $i = 1, \ldots, N$, set
   $$q_i(z_i) = p_{\hat{\theta}}\left(z_i \mid x_i\right). \tag{3.2}$$
   (b) Set
   $$\hat{\theta} = \arg\max_{\theta\in\Theta} \sum_{i\in[N]} \int q_i\left(z_i\right) \cdot \log p_\theta\left(x_i, z_i\right) \, \mathrm{d}z_i. \tag{3.3}$$

One question which might come to mind is the following: why do we not just jointly maximise over $\left(\theta, \{z_i\}_{i\in[N]}\right)$, i.e. 'joint maximum likelihood'? What would go wrong? Roughly speaking, one expects that this approach would underestimate our uncertainty about the latent values z_i, and thus 'overfit' to a point imputation of their values. 'True' maximum likelihood instead averages out over possible imputations of the latent data, and is able to extract more information as a result. This intuition can be formalised by studying the Fisher information matrix of the estimation problem.

3.3.7 Applying the EM Algorithm to Estimation of GMMs

In this Section, we work out the details of how the EM algorithm plays out in the context of Gaussian mixture models. Recall that our extended formulation of the problem takes the form

$$\begin{aligned} p_\theta\left(z, x\right) &:= \mathrm{Cat}\left(z \mid \pi\right) \cdot \mathcal{N}\left(x \mid \mu_z, \sigma^2 \cdot I_p\right) \\ &\propto \pi_z \cdot \mathcal{N}\left(x \mid \mu_z, \sigma^2 \cdot I_p\right). \end{aligned}$$

The conditional distribution can be computed as

$$p_\theta\left(z \mid x\right) = \frac{\pi_z \cdot \mathcal{N}\left(x \mid \mu_z, \sigma^2 \cdot I_p\right)}{\sum_{k\in[K]} \pi_k \cdot \mathcal{N}\left(x \mid \mu_k, \sigma^2 \cdot I_p\right)}.$$

The expected log-likelihood can be computed as

$$\begin{aligned}
\int p_{\tilde{\theta}}\left(z_i \mid x_i\right) \cdot \log p_\theta\left(x_i, z_i\right) \mathrm{d}z_i &= \sum_{k\in[K]} \tilde{w}_{i,k} \cdot \log p_\theta\left(x_i, z_i\right) \\
&= \sum_{k\in[K]} \tilde{w}_{i,k} \cdot \left(\log \pi_k - \frac{1}{2\sigma^2}\left\|x - \mu_k\right\|_2^2\right) + \text{irrelevant terms} \\
\text{where} \quad \tilde{w}_{i,k} &= \frac{\tilde{\pi}_k \cdot \mathcal{N}\left(x_i \mid \tilde{\mu}_k, \sigma^2 \cdot I_p\right)}{\sum_{\ell\in[K]} \tilde{\pi}_\ell \cdot \mathcal{N}\left(x_i \mid \tilde{\mu}_\ell, \sigma^2 \cdot I_p\right)},
\end{aligned}$$

and 'irrelevant terms' denotes terms which are constant with respect to $\{\pi_k\}_{k\in[K]}, \{\mu_k\}_{k\in[K]}$.

Summing over the full data, we thus seek to minimise

$$\sum_{i\in[N]} \sum_{k\in[K]} \tilde{w}_{i,k} \cdot \left(\log \pi_k - \frac{1}{2\sigma^2}\left\|x_i - \mu_k\right\|_2^2\right)$$

with respect to $\{\pi_k\}_{k\in[K]}, \{\mu_k\}_{k\in[K]}$. Taking partial derivatives, one can solve in closed form that

$$\begin{aligned}
\hat{\pi}_k &= \frac{\sum_{i\in[N]} \tilde{w}_{i,k}}{\sum_{i\in[N]} \sum_{k\in[K]} \tilde{w}_{i,k}} = \frac{1}{N} \sum_{i\in[N]} \tilde{w}_{i,k} \\
\hat{\mu}_k &= \frac{\sum_{i\in[N]} \tilde{w}_{i,k} x_i}{\sum_{i\in[N]} \tilde{w}_{i,k}}.
\end{aligned}$$

This gives the EM Algorithm for spherical Gaussian mixture models as follows.

Algorithm 11: EM Algorithm for spherical Gaussian mixture with data $\{x_i\}_{i=1}^N$ and K clusters.

1. Initialise $\mu_1, \ldots, \mu_K$ with distinct values, and $\pi = K^{-1}\mathbf{1}_K$.
2. Until convergence, do
   (a) For $i = 1, \ldots, N$, and for $k = 1, \ldots, K$, set

$$w_{i,k} = \frac{\pi_k \cdot \mathcal{N}\left(x_i \mid \mu_k, \sigma^2 \cdot I_p\right)}{\sum_{\ell \in [K]} \pi_\ell \cdot \mathcal{N}\left(x_i \mid \mu_\ell, \sigma^2 \cdot I_p\right)} \tag{3.4}$$

   (b) For $k = 1, \ldots, K$, set

$$\pi_k = \frac{1}{N} \sum_{i \in [N]} w_{i,k} \tag{3.5}$$

$$\mu_k = \frac{\sum_{i \in [N]} w_{i,k} x_i}{\sum_{i \in [N]} w_{i,k}}. \tag{3.6}$$

Something to think about: informally speaking, what happens to the algorithm here as $\sigma^2 \to 0^+$?

Discussion

Some key advantages of the GMM formulation of clustering and its solution by the EM Algorithm are the following:

- Not only is there a valid objective function which is being optimised, but there is even a genuine model for the data involved in the GMM. As such, if one really believes the GMM as the data-generating process, then parameter estimation by maximum likelihood is certainly a well-founded approach.
- The EM Algorithm leads to non-increasing values of the objective function, and so running the Algorithm for more iterations can never make things worse.
- Each iteration of the EM Algorithm has cost which is of polynomial size in each of (N, p, K). As such, the algorithm has a bearable computational cost, assuming that not too many iterations are required for convergence.

Some potential drawbacks and limitations include:

- By analogy with K-Means, the Gaussian model for the data suggests that in the face of outliers, one should expect difficulties.

- Algorithmically, it is again seen that initialisation can play a substantial role in the eventual quality of clusterings which are obtained. Similarly, guarantees on the convergence of the EM algorithm to global minima has not been known until recently, and it is again common practice to run multiple independent instances of the algorithm to diagnose the robustness of its findings.

Some simple extensions of this approach which could be discussed:
[Not examinable!]

- Given the likelihood-based formulation of this approach, it is comparatively straightforward to extend this approach beyond the spherical Gaussian setting. Which aspects of the algorithm do you expect to change?

- Similarly, how might the likelihood interpretation of the loss function suggest approaches to selecting K, the number of clusters?

- Latent variable models arise in various other contexts, e.g. heavy-tailed distributions are often naturally expressed as scale mixtures of Gaussian distributions, zero-inflated Poisson distributions admit a simple mixture decomposition which facilitates their estimation, and hidden Markov models are directly expressed as marginals of some extended process involving unobserved states. Roughly speaking, how might the EM algorithm apply in these settings?

- In more complex models, it may be the case that one cannot solve the minimisation problem in θ in closed form. Still, it will often be the case that by using a suitable numerical method, one can at least guarantee to find a value of θ which increases the value of the objective. This could be termed an Expectation-'Approximate Maximisation' algorithm (more often 'Generalised EM', though this is arguably less informative), and might be expected to perform well. Indeed, sometimes even when the exact minimisation in θ is feasible, it can be considerably more expensive to implement than an approximate minimisation, and so approximate steps are certainly of interest.

3.4 Clustering Outlook

[Not examinable!]

We comment briefly here on some cutting-edge research topics which involving clustering.

- There are ways of fitting the parameters of a GMM which are based not on maximum likelihood, but on the method of moments. The idea is to identify parameters $\theta = \left\{(\pi_k, \mu_k, \Sigma_k)\right\}_{k\in[K]}$ so that p_θ has some of the same polynomial moments of the data (i.e. the empirical averages of X^ℓ for $\ell = 1, 2, \cdots, L$ for some $L \in \mathbf{N}$), up to a cheap pre-processing

step. It turns out that this can be formulated as a semidefinite program (a particular structured form of convex optimisation), and hence solved efficiently, at least in principle.

- Even in the large-data limit, it has not always been clear that the log-likelihood function $\theta \mapsto \mathbf{E}_{\text{data}} \left[\log p_\theta (X) \right]$ should be favourable for optimisation, even locally. Recent work has established that under appropriate assumptions on the data distribution, things might not be so bad. These conclusions also have positive implications for finite-sample properties.

- When working with data for which a finite GMM might be an inappropriate model, it can be worthwhile to consider working with GMMs with an *infinite* number of components. It turns out that this formulation has some very favourable properties. In particular, maximum likelihood estimation for nonparametric GMMs admits an infinite-dimensional formulation which is convex in its argument. This has led to several exciting innovations. One observation has been that in this infinite-dimensional formulation, for natural data, one can use only quite a small number of components (e.g. in one dimension, one can take $K \sim \log N$, where N is the size of the data set) and obtain a fit of essentially-optimal quality.

Chapter 4

Dimension reduction

In this section, we will explore dimensionality reduction. We are often dealing with high volumes of data and this section aims to motivate why, when and how we might reduce the dimension of this data.

We define our **data** to be a set of d-dimensional vectors. That is, we have d *features* per observation, with n observations (sometimes referred to as samples). We can denote our raw data as an $n \times d$ matrix X and throughout these notes, will refer to this matrix and the vectors interchangeably.

If there are significantly fewer features than observations, that is $d \ll n$, then working with the raw data directly is much faster. Often, this is not the case and we need to work with a subset of the raw data. Dimension reduction refers to techniques and methods for extracting information with a lower dimension which is still representative of our original dataset. The overarching idea is that the high dimensional data lies on some lower dimensional manifold which we can use to represent our dataset. Consider the data shown in Figures 4.1 and 4.2. These are both in two dimensional space but can they be described in a smaller space? Figure 4.1 indeed lies on a manifold and can be described well by a 1-dimensional space. Figure 4.2 however, is randomly generated in 2-dimensional space, so reducing to fewer dimensions would result in a loss of information.

4.0 Motivation

There are a number of reasons to reduce the dimension of our data. A commonly cited reason is to avoid *overfitting*. This is important when choosing the right model to use, which will be covered in Section **??**.

Overfitting occurs when a model performs well on the training data, but poorly on unseen data as the model may have memorised the input/output pairs for the training data. The moral of the story here is, just because we have available data, does not always mean that it is helpful. Often, using fewer features to obtain a simpler model, and letting the ML algorithm infer the rest

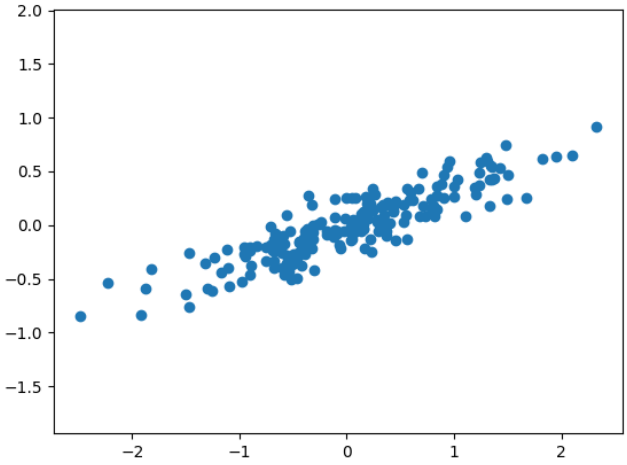


Figure 4.1: 200 data points in 2D.

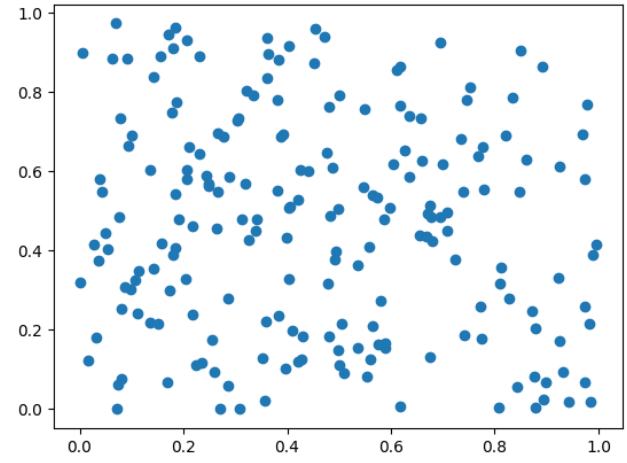


Figure 4.2: Randomly generated data.

is a better approach than using all available data. There is a balance to be found here, using too little data can create a poor model too.

Overfitting is commonly cited when we think of high dimensional data, but there are other reasons to apply dimension reduction to our data.

With less data we have **more model choices** available, **training is faster**, and less computational power will be required. There is **less storage required** for a reduced amount of data; according to the National Human Genome Institute, the DNA data for just one person could require up to 200 gigabytes to store. As discussed with overfitting, not all data is useful data, by reducing our dimension we can **remove redundancy** such as noise.

Now that we have discussed *why*, we need to consider *how*. It is useful to think of our data matrix X as having some representation in a lower dimensional latent space. The question of how then becomes how to find an appropriate representation, which will depend on several factors (often to do with the application and the data itself).

The important thing is that this lower dimensional representation contains enough *useful* information for the representation to be meaningful. There are two main categories of techniques for dimensionality reduction: ***feature selection*** and ***feature extraction***.

4.0.1 Feature Selection

In **feature selection** we choose a subset of the original features from our data. A simple example could be a dataset with features `height`, `weight` and `hair colour`, where we wish to predict height. It could be argued that hair colour is not useful here, and thus we eliminate this feature and have reduced the dimension of our data. As always, context is key, and in this example it is obvious what we may wish to eliminate. In reality, the choice of redundant features may not be so apparent.

The output from a feature selection process is a subset of the existing features in the data. Some methods filter out features based on a measure of information gain, others use a search of the features and some metric to select features. Some methods are embedded in the model building, and thus feature selection is often thought of as a part of model selection (covered later in Section 5).

4.0.2 Feature Extraction

While feature selection chooses the 'best' features from the original data, feature extraction creates a new set of features. Typically this new set of features is some combination of the original features, extracted in a way which captures as much 'useful' information as possible.

The goal with feature selection is to find a lower dimension space and project the original data down into this space.

4.1 Singular value decomposition

For a real $m \times n$ matrix A, the SVD is a factorisation of the form

$$A = U\Sigma V^T \tag{4.1}$$

$$= \sum_{i=1}^{\min\{m,n\}} \sigma_i u_i v_i^T, \tag{4.2}$$

where U is an $m \times m$ orthogonal matrix ($U^T U = UU^T = I$, Σ is an $m \times n$ (rectangular) diagonal matrix and V is an $n \times n$ orthogonal matrix.

Properties of the SVD:

- The SVD always exists, for any real matrix A.
- The singular values are always non negative.
- Non-zero elements of Σ are the singular values in decreasing order.
- The columns of U and V are the *left-singular* and *right-singular* vectors associated with A.

In this course we will not cover how to perform the decomposition to find the SVD. The factorisation is useful for dimensionality reduction, as we will see in the next section.

4.2 Low rank matrix approximation

Low rank matrix approximation is an application of the SVD for dimension reduction. The goal is to represent a matrix approximately as a matrix of lower rank. Equation (4.2) tells us that any matrix A can be written as a linear combination of rank-1 matrices. This is something which we can make use of.

If A is an $m \times n$ matrix, then it is represented by mn entries. The goal of low-rank matrix approximation is to decompose A into an $m \times k$ matrix, multiplied by a $k \times n$ matrix where k is the lower rank desired. Informally, a 'tall' rectangular matrix multiplied by a short, wide matrix. The product of these matrices can be described by $k(m+n)$ entries. When k is small, relative to both m and n, this is smaller than mn, and so we have compressed the matrix.

Let B be the rank k matrix approximating A. Then the quality of our approximation can be judged by the Frobenius norm between A and B:

$$\|A - B\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}(a_{ij} - b_{ij})^2}. \tag{4.3}$$

The Frobenius norm is orthogonally invariant, so

$$\|A\|_F = \|U^T A V\|_F \tag{4.4}$$

$$= \|\Sigma\|_F = \sqrt{\sum_{i=1}^{\min\{m,n\}} \sigma_i^2}. \tag{4.5}$$

Theorem 11 (Schmidt [1908])**.** *The rank k matrix, B, which minimises the Frobenius norm in equation* (4.3) *is*

$$\tilde{A}_k = U_k \Sigma_k V_k^T, \tag{4.6}$$

where U_k is the $m \times k$ matrix formed by taking the first k columns of U, Σ_k is the diagonal matrix of the first k singular values and V_k is the $n \times k$ matrix formed by taking the first k columns of V.

Theorem 11 tells us that an optimal choice for our low-rank approximation is the truncated SVD. Further, the residual $\|A - \tilde{A}_k\|_F = \sqrt{\sum_{i=k+1}^{\min\{m,n\}} \sigma_i^2}$.

Therefore to perform low rank approximation, we begin with the SVD and simply 'keep' the first k entries of the sum (assuming that the singular values have been sorted so that $s_1 \geq s_2 \geq \cdots \geq 0$).

Figure 4.3 shows the results of an image being compressed using this method, for different values of k.

4.3 Principal components analysis

Principal components analysis (PCA) is a dimensionality reduction technique which is widely used due to its simplicity. PCA was proposed by Pearson in 1901 Pearson [1901] and is useful for exploratory data analysis. It is an unsupervised, linear transformation technique. PCA works by finding patterns in the data, and extracting useful information in a lower dimension, based on the correlation between each feature.

We start with d-dimensional vectors, where d is the number of features in our data, and reduce these to a q-dimensional subspace. The q-dimensional subspace will be a *summary* of the larger space and a *projection* of the original vectors onto the *principal components*. This projection is a linear map, and is done in a way which maximises the variance of the data.

PCA works by calculating eigenvectors from the covariance matrix of the data. The eigenvectors which correspond to the largest eigenvalues (these are the principal components) are then used to reconstruct an approximation of the variance of the original data.

4.3.1 Principal components

The principal components of a data set in $\mathbb{R}^d$ give a sequence of linear approximations to the original data. There are several equivalent ways to obtain the principal components, we will only cover one in this course.

(a) $k = 1$ (b) $k = 2$ (c) $k = 3$

(d) $k = 4$ (e) $k = 5$ (f) $k = 10$

(g) $k = 20$ (h) $k = 50$ (i) $k = 100$

Figure 4.3: Florence Nightingale's image compressed using Low rank matrix approximation for different values of rank, k.

Problem formulation

To formulate the problem, we will first consider finding a 1-dimension representation. The goal is as follows: find a direction vector to project our data onto so that we can minimise the projection error, also called the *projection residuals*.

We begin d-dimensional vectors $x_1, \ldots, x_n$ that are centred, i.e. $\sum_{i=1}^{n} x_i = 0$. The one dimensional projection is a line through the origin which can be specified by a unit vector, w. The projection of x_i on to the line is $x_i \cdot w$ (this is a scalar value).

The 'best' projection is the one which is closest to the true data vectors. To find this, we consider the residuals. For one vector, the residual of the projection is

$$\begin{aligned} |x_i - (w \cdot x_i)w|^2 &= (x_i - (w \cdot x_i)w) \cdot (x_i - (w \cdot x_i)w) && (4.7)\\ &= |x_i|^2 - 2(w \cdot x_i)^2 + (w \cdot x_i)^2 w \cdot w && (4.8)\\ &= |x_i|^2 - (w \cdot x_i)^2, && (4.9) \end{aligned}$$

since the dot product is distributive and w is a unit vector. Summing these residuals across all n vectors gives the mean squared error (MSE)

$$\begin{aligned} MSE(w) &= \frac{1}{n} \sum_{i=1}^{n} |x_i|^2 - (w \cdot x_i)^2 && (4.10)\\ &= \frac{1}{n} \left(\sum_{i=1}^{n} |x_i|^2 - \sum_{i=1}^{n} (w \cdot x_i)^2 \right). && (4.11) \end{aligned}$$

The first sum in Equation (4.11) does not depend on the projection, so to minimise the MSE, we must maximise the second term:

$$\frac{1}{n} \sum_{i=1}^{n} (w \cdot x_i)^2. \tag{4.12}$$

This is also the *sample mean* of $(w \cdot x_i)^2$.

Recall that if Y is a random variable, then $\mathbb{E}(Y^2) = \mathbb{E}(Y)^2 + \mathrm{Var}(Y)$. Therefore, Equation (4.12) can be written as

$$\frac{1}{n} \sum_{i=1}^{n} (w \cdot x_i)^2 = \left(\frac{1}{n} \sum_{i=1}^{n} w \cdot x_i \right)^2 + \mathrm{Var}(w \cdot x_i), \tag{4.13}$$

where the first argument is the mean of the projections, which is 0 (since $\frac{1}{n} \sum_{i=1}^{n} x_i = 0$).

We have so far seen that *minimising* the mean squared error (Equation (4.12)) is equivalent to *maximising* Equation (4.13) - the variance of the projections.

However, this is for projecting down to just one vector (or one principal component). We often want to project to multiple principal components (the figure below has two principal components, for example). The image of data vector x_i projected onto the space spanned by k unit vectors, $w_1, \ldots, w_k$, which are orthogonal to one another is

$$\sum_{j=1}^{k} (x_i \cdot w_j) w_j. \tag{4.14}$$

Following the same procedure to find the mean squared error, we are left to maximise the sum of the variances of the projections onto the components.

Maximising the variance

We work with the projected variance. That is, $Var(XW)$, which we can expand as follows:

$$\begin{aligned} \frac{1}{n}(XW)^T(XW) =& W^T \frac{X^T X}{n} W & (4.15)\\ =& W^T C W & (4.16) \end{aligned}$$

where C is the covariance of X.

This is a variance-covariance matrix, and the diagonal entries are the variances of the projections. The diagonal entries are given by terms $w^T C w$ (where w is a vector representing a column of W), and the w are unit vectors. We can perform the optimization using a Lagrange multiplier, λ.

The unit vector constraint can be written as $w^T w = 1$ (equivalently $w^T w - 1 = 0$), so our Lagrangian for the maximisation is

$$L(w, \lambda) = w^T C w - \lambda(w^T w - 1). \tag{4.17}$$

Which we differentiate with respect to λ and w to obtain

$$\begin{aligned} \frac{\partial L(w,\lambda)}{\partial \lambda} &= w^T w - 1 & (4.18)\\ \frac{\partial L(w,\lambda)}{\partial w} &= 2Cw - 2\lambda w. & (4.19) \end{aligned}$$

This means that our equation is maximised when

$$\begin{aligned} w^T w = 1, \text{ and} & & (4.20)\\ Cw = \lambda w. & & (4.21) \end{aligned}$$

By taking w such that $Cw = \lambda w$ and $w^T w = 1$ we see that

$$w^T C w = \lambda w^T w = \lambda,$$

and so the choice of w that maximizes the variance of the corresponding projection is the eigenvector of C with the largest eigenvalue. So the first principal component is the eigenvector of C with the largest eigenvalue.

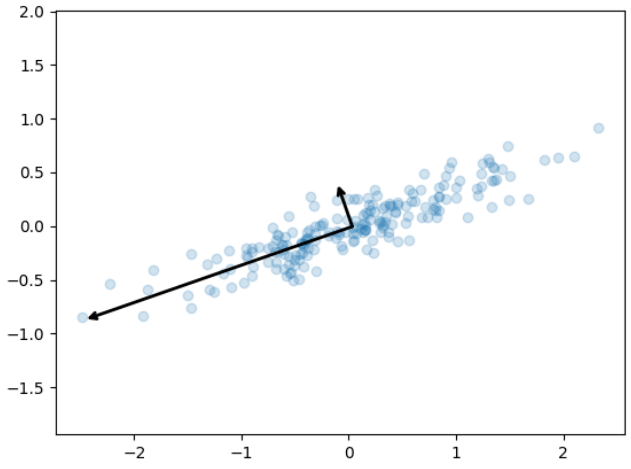


Figure 4.4: The data from Figure 4.1 with the first 2 principal components marked on.

To find the second (and further) principal components we can similarly perform constrained optimization with the additional constraint that the vector w should be orthogonal to the first (all previous) principal components. By adding the appropriate Lagrange multipliers we again obtain that the solution is that w should be an eigenvector of C and hence the principal components are given in order by the eigenvectors of C with decreasing eigenvalues.

Remark 6. *The principal components are constructed such that the first principal component maximises the variance of the projected data, the second principal component has the second most variance. Figure 4.4 shows the first 2 principal components, with their lengths proportional to their explained variance.*

Step-by-step PCA

We showed in sections 4.3.1 and 4.3.1 that the principal components which maximise the variance are the eigenvalues of the data covariance matrix. To recap, the key steps to finding the principal components are:

1. Mean center the data, so that the mean is 0.
2. Calculate the covariance matrix for the scaled variables.
3. Calculate the eigenvalues of the covariance matrix.

The largest eigenvector which corresponds to the largest eigenvalue is the first principal component.

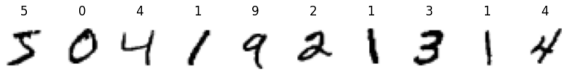


Figure 4.5: Handwritten digits sampled from the MNIST dataset.

4.3.2 How many principal components?

We tend to keep principal components which have eigenvalues that are 'large enough' - how do we decide? It is useful to understand how much information is contained in each principal component first.

Discarded eigenvectors will contain some useful information, even if they also contain some noise but we can't keep them all, otherwise we aren't reducing our dimension!

Definition 4.3.1. *The* explained variance *of a principal component with corresponding eigenvalue λ_i is*

$$\frac{\lambda_i}{\sum_{j=1}^{n} \lambda_j}. \tag{4.22}$$

There is no agreed number of principal components to keep, however those with a 'large' explained variance are good to keep. For exploratory data analysis (EDA), we are interested in visualising the data, so often only the first 2 or 3 are kept to visualise.

4.3.3 Example: MNIST handwritten digits

The MNIST database is a dataset of handwritten digits 0-9. Each sample in the original database is a black and white image which has been normalised to fit into a 28x28 pixel box - seen in Chapter 2 in Figure 2.2. As a reminder, a sample of the digits are presented in Figure 4.5. For this example, we work with a more pixelated version of the digits which is 8x8 (so 64 dimensions). We will reduce the dimension from 64 to 2.

Performing PCA on 1797 of the digits, and plotting them on the axes of the first two principal components gives the result in Figure 4.6. The digits have been colour coded to categorise them. We can see that even with only two principal components, we obtain reasonable separation between the digits.

Next, we consider the explained variance (Definition 4.3.1). Figure 4.7 shows the cumulative explained variance as the number of principal components used increases. To retain 80% explained variance, we would need to keep 12 principal components. To retain 90%, we would need 20 components. Despite this, we still see useful information displayed in just 2 components. This is why PCA is often used to explore high dimensional datasets and understand the redundancy present.

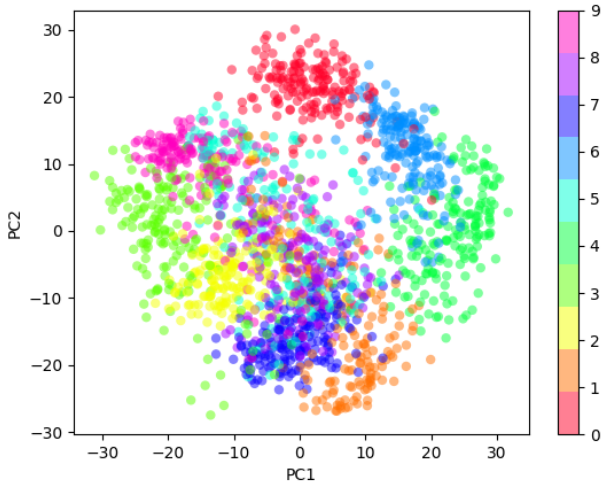


Figure 4.6: 1797 handwritten digits from the MNIST dataset, plotted on their first 2 principal components.

4.3.4 Kernel PCA

Consider applying PCA as above to the data in Figure 4.8. Where would this go wrong? The data still lies on a 1-dimensional manifold using polar coordinates. If we use PCA on this, the suggested dimension will be 2 (as it will find two components, both accounting for half the variance).

Although we didn't say this explicitly, the act of computing the eigenvectors is a linear transformation. In finding the principal components, we are taking linear combinations of the data in a principled way to bring out patterns which are not immediately obvious. This works well for linear data, but how do we deal with non-linear data? Kernel principal components was designed to be a non-linear version of principal components.

In Kernel PCA the ideas from standard PCA are expanded to consider non-linear transformations of the features. As the number of variable combinations explodes with non-linear transformations, this becomes computationally difficult. To mitigate this, we can use the *kernel trick*.

We introduce a function $\phi(x_i)$, which maps the data points to some nonlinear space, which we will call the *feature space*. One example of such a ϕ could be using polar coordinates instead of cartesian coordinates. Rather than mean-centering the data, we now assume the data in the feature space is centered as follows

$$\sum_{i=1}^{n} \phi(x_i) = 0. \tag{4.23}$$

Now the kernel matrix, $\kappa(x) = \phi(x)\phi^T(x)$ enables us to acquire the eigenval-

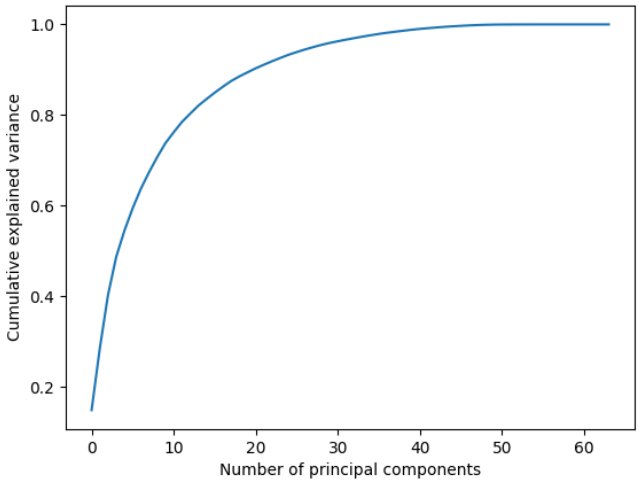


Figure 4.7: Cumulative explained variance for the 64 principal components

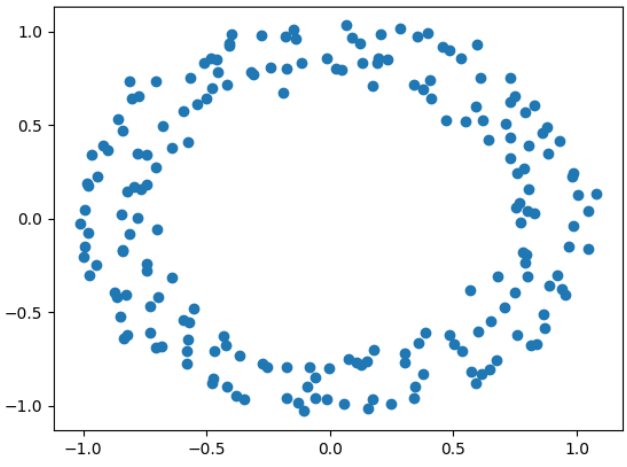


Figure 4.8: 200 data points in 2D.

ues and eigenvectors needed for kernel PCA without calculating $\phi(x)$ directly.

The process for Kernel PCA is then largely the same as the standard PCA. Rather than using the covariance matrix, the kernel matrix is used.

$$\frac{1}{n}\sum_{i=1}^{n}\phi(x_i)\phi(x_i)^T, \tag{4.24}$$

Remark 7. *The kernel matrix is $n \times n$, so kernel PCA may have difficulties for large numbers of data points. In this case, a subset of data would be chosen using dictionary methods (which are beyond the scope of this course).*

4.4 Topic models

Topic models are a type of algorithm used to discover themes in unstructured collections of text. They are useful as they require no training sets, prior information or labelling before outputting an estimate (so they are an *unsupervised*).

There is an underlying assumption that each text document is comprised of a statistical mixture of topics. That is, the mixture of topics are modelled as a statistical distribution which may be derived as the sum of the distributions of the individual topics.

But what is a topic? A topic is a subject covered within the text, or a set of words which co-occur. Topic models give a probability distribution over a fixed vocabulary. That is, a set of words of interest.

A topic model would aim to infer the presence of the words in the vocabulary and their strength of that presence with regards to a particular topic. Consider the vocabulary: matrix, data, dimension, cat, dog, collar. When talking about the topic *data science*, the words *matrix*, *data* and *dimension* will be used frequently whereas *cat*, *dog* and *collar* will be used infrequently. When talking about the topic *pet shops*, these will be reversed. However, no words will have a zero probability under either topic.

4.4.1 Bag of words models

Both of the topic models covered in this course use an input of text data in the ***bag-of-words*** format. The bag-of-words disregards punctuation and order, keeping all of the words themselves and their multiplicity. Much like putting all of the constituent words of a dataset into a bag!

- **Word**: a unit of discrete data.
- **Dictionary**: the set of all words which appear across the entire corpus.
- **Document**: a sequence of words (can be represented as a vector of words) which are the individual observations in our dataset. This is represented by a feature vector under the model.

- **Corpus**: the collection of documents (the entire dataset, sometimes called the body).
- **Topic**: a collection of words that occur together.

Example 4. *We will use a snippet from Macbeth, taken from Project Gutenberg, to be our corpus:*

Double, double, toil and trouble;
Fire burn and cauldron bubble.

Each individual line will be treated as a document. So in this example, we have two documents. The dictionary is of size 8 and consists of the unique terms appearing across the corpus:

"double", "toil", "and", "trouble", "fire", "burn", "cauldron", "bubble",

therefore each document can be represented by a fixed-length vector of length 8.

We can 'score' the first document ("double, double, toil and trouble") according to this fixed-length vector to give

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

This is scored using Boolean logic, where a 1 means that the word is present in the document, and a 0 means it is not. Under this regime, a bag-of-words matrix representation of our corpus is:

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}. \tag{4.25}$$

The transpose of the matrix in Equation4.25 is called the ***Term Document Matrix*** for the corpus.

Remark 8. *Note that in Example 4 we ignored case and punctuation. This process referred to as* text cleaning *and is required to reduce the dimension of the dictionary. This example was only two lines, but if our corpus is a series of books the dictionary can become very large. Each document may contain few of the words from the dictionary, resulting in a sparse bag-of-words matrix.*

Some simple text cleaning techniques which are often applied include:

- *Fixing spelling errors.*
- *Ignoring 'stop words', these are words such as "a" and "of" which do not contain much information.*
- *Reducing words to their stem (eg "barking" becomes "bark") using a stemming algorithm. Stemming algorithms are not covered in this course.*

Other common ways to clean text include fixing spelling errors, reducing words to their stem and ignoring frequent words which don't contain much information, known as stop words, such as "a" and "of".

The method used to score in Example 4 only counts the presence of words, not their frequency or count. It is also possible to score by counting the instances of each word. Under this scoring system, the representation from Equation (4.25) becomes

$$\begin{pmatrix} 2 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}. \tag{4.26}$$

Exercise: what does the matrix representation become when using a frequency score?

4.4.2 Term Frequency-Inverse Document Frequency (TF-IDF)

Scoring systems based on word frequencies or counts mean that high frequency words may dominate, but these may not necessarily contain a large amount of useful information when compared to specific, but rarer, words.

Words such as "the" may appear frequently across all documents. Term Frequency - Inverse Document Frequency (TF-IDF) is an approach which tries to combat this by scaling the frequency across over all documents.

- **Term Frequency** scores the frequency of the word in the current document, using any frequency metric (as we saw in Example 4).
- **Inverse Document Frequency** scores how rare the word is across all documents. A high IDF is likely to point to a rare word, a low IDF is likely to point to a frequent word.

A commonly used frequency metric for term t in document d is:

$$\mathrm{tf(t,d)} = \frac{\mathrm{X_d(t)}}{\sum_{\mathrm{t=1}}^{\mathrm{T}} \mathrm{X_d(t)}}, \tag{4.27}$$

where T is the total number of words (in Example 4, $T = 8$). The (log) inverse document frequency is:

$$\mathrm{idf(t,d)} = \log\left(\frac{\mathrm{D}}{1 + \mathrm{n_d(t)}}\right), \tag{4.28}$$

where D is the total number of documents (in Example 4, $D = 2$) and $n_d(t)$ is the number of documents which contain term t. Here, $X_d(t)$ denotes some score for term t in document d. In Example 4,

$$X_d(t) = \begin{cases} 1 \text{ if term } t \in d \\ 0 \text{ otherwise.} \end{cases}$$

Commonly, we will use the count of term t in document d. That is, how many times the term appears in the document. The 1 in Equation (4.28) comes from a Bayesian smoothing procedure which we will not cover in this course.

So the TF-IDF equation takes the form

$$\mathrm{tf - idf(t,d)} = \frac{\mathrm{X_d(t)}}{\sum_{\mathrm{t=1}}^{\mathrm{T}} \mathrm{X_d(t)}} \log\left(\frac{\mathrm{D}}{1 + \mathrm{n_d(t)}}\right). \tag{4.29}$$

4.5 Latent Semantic Indexing (LSI)

Latent Semantic Indexing (LSI), sometimes called Latent Semantic Analysis, is a process for approximating the term-document matrix, X, using the SVD (Section 4.1). It is an unsupervised technique. The word *semantic* refers to trying to find the relationships between words. The goal in LSI is to find combinations of topics which are close together in latent space by querying the corpus with multiple topics and finding the documents which are close.

LSI assumes that words which are close in meaning will occur together in similar pieces of text. We use the document vectors obtained from Section 4.4.2 and approximate it using Low rank matrix approximation (Section 4.2). So we have that

$$\tilde{X}_k \simeq U_k \Sigma_k V_k^T, \tag{4.30}$$

where U, Σ and V are defined as in Section 4.1 and the subscript k denotes the low rank approximation of order k.

We consider querying a piece of text, and aim to find documents in our corpus that are close to our query. Let the query be represented as a vector q. We find the lower dimensional representation of our terms in the latent space (the rows of $U_k\Sigma_k$) and represent our query as the centre of the latent terms.

We can then rank the documents in the latent space (the columns of $\Sigma_k V_k^T$) relatively to the vector representing the query, q. For document i, this can be computed using the cosine similarity:

$$\frac{d_i \cdot q}{\mid d_i \mid\mid q \mid}.$$

Interpreting the SVD of a term-document matrix.

Our original term-document matrix was $m \times n$, where m is the number of terms and n is the number of documents.

U is an $m \times k$ matrix, where k is the number of topics and m is the number of terms. This can be interpreted as a term-topic matrix.

The non-zero entries of Σ represent how much each latent concept explains the data variance. A large entry would correspond to a latent concept with a large amount of explained variance.

The matrix V is $n \times k$, so can be interpreted as a document-concept matrix.

We will represent our documents as columns of ΣV^T and our terms as rows of $U\Sigma$.

Considerations for LSI

The computational cost of performing the SVD for large amounts of data is significant. This means that in practice, it is often performed on a sampled subset of the data.

4.5.1 LSI example: Romeo and Juliet

The following example is taken from Thomo, they provide more results than are presented here.

We will use the documents
d_1: Romeo and Juliet.
d_2: Juliet: O happy dagger!
d_3: Romeo died by dagger.
d_4: 'Live free or die', that's the New-Hampshire's motto.
d_5: Did you know, New-Hampshire is in New-England.

Consider the search query *dies*, *daggers*. We hope that d_3 would rank highly, since it contains both query terms. Then d_2 and d_4 should follow as each contain one word from the query. If you are familiar with the plot of Romeo and Juliet, you may know that d_1 is also related, but d_5 is not related to the query.

The aim of LSI is to detect the above relations. We want to see that d_3 is closer to the query than d_4 which in turn is closer to the query than d_5.

The term document matrix, X, is:

	d_1	d_2	d_3	d_4	d_5
romeo	1	0	1	0	0
juliet	1	1	0	0	0
happy	0	1	0	0	0
dagger	0	1	1	0	0
live	0	0	0	1	0
die	0	0	1	1	0
free	0	0	0	1	0
new-hampshire	0	0	0	1	1

The matrix Σ is

$$\Sigma = \begin{bmatrix} 2.285 & 0 & 0 & 0 & 0 \\ 0 & 2.010 & 0 & 0 & 0 \\ 0 & 0 & 1.361 & 0 & 0 \\ 0 & 0 & 0 & 1.118 & 0 \\ 0 & 0 & 0 & 0 & 0.797 \end{bmatrix}.$$

We will perform a low-rank matrix approximation, letting $k = 2$.

Our query (*dies*, *dagger*) is now represented as the centre of the vectors for its terms.

$$\Sigma_2 = \begin{bmatrix} 2.285 & 0 \\ 0 & 2.010 \end{bmatrix}.$$

$$U_2 = \begin{bmatrix} -0.396 & 0.280 \\ -0.314 & 0.450 \\ -0.178 & 0.269 \\ -0.438 & 0.369 \\ -0.264 & -0.346 \\ -0.524 & -0.246 \\ -0.264 & -0.346 \\ -0.326 & -0.460 \end{bmatrix}$$

$$V_2^T = \begin{bmatrix} -0.311 & -0.407 & -0.594 & -0.603 & -0.143 \\ 0.363 & 0.541 & 0.200 & -0.695 & -0.229 \end{bmatrix}.$$

The terms in the latent space are now represented by the row vectors of $U_2\Sigma_2$. The documents are represented by the columns of $\Sigma_2 V_2^T$. For example:

$$\text{dagger} = \begin{bmatrix} -1.001 \\ 0.742 \end{bmatrix}, \text{die} = \begin{bmatrix} -1.197 \\ -0.494 \end{bmatrix},$$

and the query is represented as the centre of the vectors for *dagger* and *die*:

$$q = \frac{\begin{bmatrix} -1.197 \\ -0.494 \end{bmatrix} + \begin{bmatrix} -1.001 \\ 0.742 \end{bmatrix}}{2} = \begin{bmatrix} -1.099 \\ 0.124 \end{bmatrix}.$$

and

$$d_1 = \begin{bmatrix} -0.711 \\ 0.730 \end{bmatrix}, d_2 = \begin{bmatrix} -0.930 \\ 1.087 \end{bmatrix}, d_3 = \begin{bmatrix} -1.357 \\ 0.402 \end{bmatrix}, d_4 = \begin{bmatrix} -1.378 \\ -1.397 \end{bmatrix}, d_5 = \begin{bmatrix} -0.327 \\ -0.460 \end{bmatrix}.$$

Figure 4.9 shows the vectors above. We can see that document d_3 is indeed closest to our query, as expected.

4.5.2 Latent Dirichlet Allocation (LDA)

Remark 9. *The acronym LDA used in this section is different to Linear Discriminant Analysis in Section 2.2.1. Apologies for any confusion!*

Another way of comparing documents is using Latent Dirichlet Allocation (LDA) Blei et al. [2003]. We use the vector representation of our set of documents under the Bag-of-Words model and the term document matrix.

The LDA model is a Bayesian mixture model which assumes that topics are uncorrelated. It is generative, but because the words are independent the simulated documents are rarely useful. It is more useful for generating keywords and to categorise text.

- Each document is modelled as a mixture of topics under a Bayesian mixture model.
- Each topic is modelled as a distribution over the words.

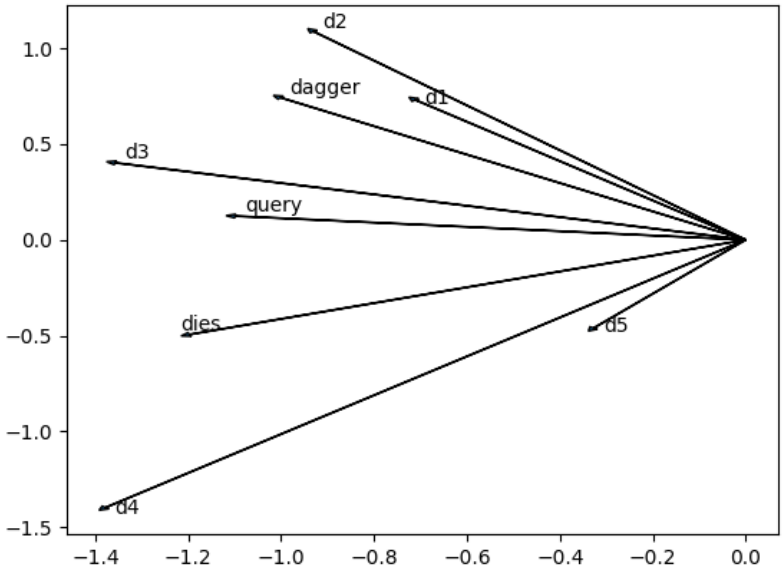


Figure 4.9: 2-dimensional vectors representing the data from Example reference.

- Only the number of topics is specified in advance.

We have N words and D documents and K topics. Let η be the word distribution (a length N vector), and α be the topic distribution (a length K vector). We already know the words that belong to each document, and would like to calculate the distribution of words belonging to a topic. Each topic k of the K is described by a word frequency vector which is drawn from a Dirichlet distribution,

$$\beta_k \sim \text{Dirichlet}(\eta). \tag{4.31}$$

Each document d of D is described by a topic frequency vector

$$\theta_d \sim \text{Dirichlet}(\alpha). \tag{4.32}$$

We then use this θ_d to, for each word, draw a topic assignment

$$t_{d,n} \sim \text{Multinomial}(\theta_{\text{d}}), \tag{4.33}$$

and draw a word using this topic

$$w_{d,n} \sim \text{Multinomial}(\beta_{\text{t}_{\text{d,n}}}). \tag{4.34}$$

The inference then consists of estimating a posterior distribution over the parameters from a combination of what we observe (words in documents) and latent variables (topic and word parameters).

4.5.3 LDA: the generative model

The generative process for each document d in a corpus is as follows:

1. Choose a $\theta_d \sim \text{Dir}(\alpha)$.
2. Choose a $\beta_k \sim \text{Dir}(\eta)$,
3. For each of the N words, w_n, choose:
   (a) A topic $t_{d,n} \sim \text{Multinomial}(\theta)$ (this will give an integer, indicating the topic of the nth word in the dth document).
   (b) A word $w_{d,n} \sim \text{Multinomial}(\beta_{\text{t}_{\text{d,n}}})$ (this will give an integer which indexes over all the possible words).

In this process, the dimensionality k of the Dirichlet distribution (and thus the topic variable t) is assumed to be known, and it is fixed. The word probabilities are parameterised by β where $\beta_{ij} = p(w_j = 1 \mid t_i = 1)$.

In reality, this generative model will produce something which is unlikely to make sense. However there is a nonzero probability it will also produce the original text. We would like to finetune the parameters to maximise this probability.

LDA iterates over each document and each word, computing two probabilities for every topic in the process. Using these probabilities, LDA can estimate a new posterior.

The joint distribution across the words, topics, θ and β is:

$$p(w, t, \theta, \beta \mid \alpha, \eta) = \prod_k p(\beta_k \mid \eta) \prod_d p(\theta_d \mid \alpha) \prod_n p(t_{d,n} \mid \theta_d) p(w_{d,n} \mid t_{d,n}, \beta).$$

This is a product over the topics, documents and words. Given the D documents, the goal is now to find the parameters which best maximise this joint probability. This can be done, for example, using a variational EM algorithm, which will not be covered in this course.

Chapter 5

Model selection and aggregation

We have seen that there are many techniques to build models from data, but as Breiman [2001] wrote: *"Suppose two statisticians, each one with a different approach to data modelling, fit a model to the same data set. Assume that each one applies standard goodness-of-fit tests, looks at residuals, etc., and is convinced that their model fits the data. Yet the two models give different pictures of nature's mechanism and lead to different conclusions."*

The question of which model(s) most accurately reflects the data is not straightforward to resolve. This leads to the question of how do we choose a 'good' or 'right' model? In the first part of this chapter, we will consider various criteria and methods for choosing a model. We start by describing various kinds of errors in Section 5.2, then introduce the key concepts of bias and variance 5.2.2, and build on these ideas to describe properties of the widely used model selection criteria AIC and BIC in Sections 5.2.4 and 5.2.5.

In the later parts of this chapter (Sections 5.4 and 5.4.4), we will take a look at a class of models - tree-based models - that provide conceptually straightforward but powerful and popular techniques for handling data where the relationship between features and outcome is non-linear, or where features interact with one another.

Key references for this chapter are Hastie et al. [2008], James et al. [2021], and Johari [2016].

5.1 Motivating example

In Figure 5.1 (vanGerwen [2021]) four models (columns) are visualised for five datasets (rows) for binary classification. For each of the four models: logistic regression (Section 2.3), support vector machine (Section 2.4), decision tree (Section 5.4) and a random forest (Section 5.4.4), from the shading of the decision boundaries in the background we can see how the different models perform

the classification differently.

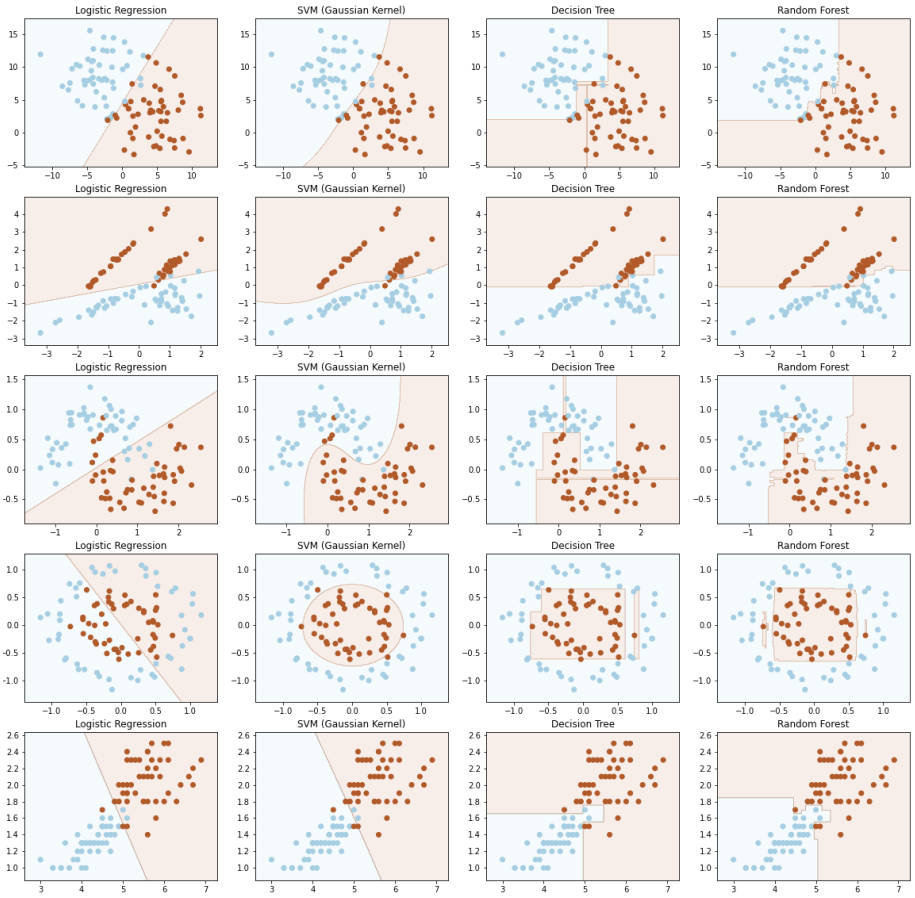


Figure 5.1: Classifier comparison

Without going into further detail on this illustration (i.e., the distributions of each dataset), by visual inspection, and from earlier sections of this course, we can see that there are limitations and challenges for each algorithms.

The performance of a learning method relates to its prediction capability on independent test data. Assessment of this performance is extremely important in practice, since it guides the choice of model, and gives us a measure of the quality of the chosen model. A major question is how to best avoid overfitting a model we build to the training data, while leveraging as much information as we can to make useful predictions for the test data.

5.2 Test error

5.2.1 Types of errors

Suppose we observe sample data $\boldsymbol{X}$, a matrix of covariates consisting of n rows and p variables, and corresponding outcomes $\boldsymbol{Y}$, a vector of n responses. We assume that for any observation X, the corresponding response Y can be modelled as $Y = f(X) + \varepsilon$, where ε is a noise term with mean $\mathbb{E}[\varepsilon|X] = 0$, and variance $\text{Var}(\varepsilon|X) = \sigma^2$, which does not depend on the data $\boldsymbol{X}$.

The problem we will focus on here is: given data $\boldsymbol{X}$ and $\boldsymbol{Y}$, fit a model $\hat{f}$ to approximate the true f so a prediction can be made about a new covariate vector X. The model $\hat{f}$ is chosen so the **training error** between $\hat{f}(\boldsymbol{X})$ and $\boldsymbol{Y}$ is minimised. Measurement of the test error depends on the loss function $L(Y, \hat{f}(X))$, for example, in regression: squared error loss, absolute error and in classification: zero-one loss. We will focus on regression with squared error as our measure of prediction error. The training error is defined as the average the loss over training sample:

$$\text{Err}_{\text{train}} = \frac{1}{n} \sum_{i=1}^{n} \left(Y_i - \hat{f}(\boldsymbol{X}_i) \right)^2 . \tag{5.1}$$

The **test error** (also called prediction error and generalisation error) for a new test data point X, Y is

$$\text{Err}_{\text{test}} = \mathbb{E} \left[\left(Y - \hat{f}(X) \right)^2 | \boldsymbol{X}, \boldsymbol{Y} \right], \tag{5.2}$$

where the only randomness is in the new sample X and Y. Hence the training set is fixed and the test error is specific to this training set. A related quantity is the **expected test error**, which is obtained by averaging over training sets. If we are in a data-rich situation, the best approach is

1. Separate the data into three parts: a **training set**, a **validation set**, and a **test set**
2. Use the training data to fit different models $\hat{f}$.
3. **Model selection:** Use validation data to estimate test error of the different models, and pick the best one.
4. **Model assessment:** Use test data to assess the performance of the chosen model.

Therefore, when constructing model $\hat{f}$ we need to balance explaining our training data with fitting unseen test data. It is difficult to give a general rule on how to choose the number of observations in each of the three parts, as this depends on the signal-to-noise ratio in the data and the training sample size. A typical split is to take 50% of the data for training, 25% for validation and 25% for testing.

Validation estimates the test error of the different models, and chooses the best one. Suppose the validation set is made up of samples $(\tilde{\boldsymbol{X}}_1, \tilde{Y}_1), \ldots, (\tilde{\boldsymbol{X}}_k, \tilde{Y}_k)$. Then for each fitted model $\hat{f}$, we can estimate the test error as

$$\frac{1}{k}\sum_{i=1}^{k}\left(\tilde{Y}_i - \hat{f}(\tilde{\boldsymbol{X}}_i)\right)^2, \tag{5.3}$$

and choose the model with the smallest test error. The validation error of the best model in the validation step is typically an underestimate of the true test error.

To obtain an accurate (i.e., unbiased) estimate of the test error of the selected model, we use another holdout set, called the test set. Suppose the test set is made up of samples $(\tilde{\boldsymbol{X}}_{k+1}, \tilde{Y}_{k+1}), \ldots, (\tilde{\boldsymbol{X}}_\ell, \tilde{Y}_\ell)$. Let $\hat{f}^*$ be the chosen model, then an unbiased estimate of test error is:

$$\frac{1}{\ell - k}\sum_{i=k+1}^{\ell}\left(\tilde{Y}_i - \hat{f}^*(\tilde{\boldsymbol{X}}_i)\right)^2. \tag{5.4}$$

So far we have seen how we can select and evaluate predictive models using the train-validate-test methodology. This approach works well if we have 'enough' data. What if we don't have enough data to blindly train and validate models? We have to understand the behavior of test error well enough to intelligently explore the space of models.

5.2.2 Bias-variance decomposition

We will characterize exactly how the test error behaves through the ideas of bias and variance. More formally, the bias-variance decomposition describes how sensitive prediction error is to changes in the **training data** (in this case, $\boldsymbol{Y}$) as a sum of three components.

The bias-variance decomposition is:

$$\mathbb{E}\left[\left(Y - \hat{f}(X)\right)^2 | \boldsymbol{X}, X\right] = \underbrace{\left(\mathbb{E}\left[\hat{f}(X)|\boldsymbol{X}, X\right] - f(X)\right)^2}_{\text{Bias}^2} + \underbrace{\mathbb{E}\left[\left(\hat{f}(X) - \mathbb{E}[\hat{f}(X)|\boldsymbol{X}, X]\right)^2 | \boldsymbol{X}, X\right]}_{\text{Variance}} + \sigma^2 \tag{5.5}$$

Each term has the following interpretation:

- The first term is the amount by which the average of our estimates differs from the true mean (squared). **If there are systematic errors in prediction made regardless of the training data, then there is high bias**. This is mainly due to having a model too rigid (too simple).

- The second term is the expected squared deviation of $\hat{f}(x)$ around its means. **If the fitted model is very sensitive to the choice of training data, then there is high variance**. This is mainly due to sampling and having a structure too flexible, so the estimated model $\hat{f}$ changes a lot with different training samples (changes on the training set lead to very different solutions),
- The third term is the **irreducible error** which is the variance of the target Y around its true mean $f(x)$. This cannot be avoided no matter how well we estimate $f(x)$ (unless $\sigma^2 = 0$), due to the noise ϵ.

The justification for bias-variance decomposition can be obtained in a few steps and we present the proof of Equation (5.5).

Proof. We have

$$\begin{aligned}
&\mathbb{E}\left[\left(Y - \hat{f}(X)\right)^2 |\boldsymbol{X}, X\right] = \mathbb{E}\left[\left(f(X) - \hat{f}(X) + \varepsilon\right)^2 |\boldsymbol{X}, X\right] \\
&= \mathbb{E}\left[\left(f(X) - \hat{f}(X)\right)^2 |\boldsymbol{X}, X\right] + 2\mathbb{E}\left[\left(f(X) - \hat{f}(X)\right)\varepsilon |\boldsymbol{X}, X\right] + \mathbb{E}\left[\varepsilon^2 |\boldsymbol{X}, X\right] \\
&= \mathbb{E}\left[\left(f(X) - \hat{f}(X)\right)^2 |\boldsymbol{X}, X\right] + \sigma^2.
\end{aligned}$$

Adding and subtracting $\mathbb{E}[\hat{f}(X)|\boldsymbol{X}, X]$ in the first term

$$\mathbb{E}\left[\left(\hat{f}(X) - f(X)\right)^2 |\boldsymbol{X}, X\right] = \mathbb{E}\left[\left(\hat{f}(X) - \mathbb{E}[\hat{f}(X)|\boldsymbol{X}, X] + \mathbb{E}[\hat{f}(X)|\boldsymbol{X}, X] - f(X)\right)^2 |\boldsymbol{X}, X\right],$$

and multiplying out:

$$\begin{aligned}
\mathbb{E}\left[\left(\hat{f}(X) - f(X)\right)^2 |\boldsymbol{X}, X\right] = \mathbb{E}\left[\left(\hat{f}(X) - \mathbb{E}[\hat{f}(X)|\boldsymbol{X}, X]\right)^2 |\boldsymbol{X}, X\right] + \left(\mathbb{E}\left[\hat{f}(X)|\boldsymbol{X}, X\right] - f(X)\right)^2 \\
+ 2\mathbb{E}\left[\left(\hat{f}(X) - \mathbb{E}[\hat{f}(X)|\boldsymbol{X}, X]\right)\left(\mathbb{E}[\hat{f}(X)|\boldsymbol{X}, X] - f(X)\right) |\boldsymbol{X}, X\right].
\end{aligned}$$

The conditional expectation drops out on the middle term, because given $\boldsymbol{X}$ and X, it is no longer random. The third term is zero, which can be seen using the tower property of conditional expectation:

$$\begin{aligned}
&\mathbb{E}\left[\left(\hat{f}(X) - \mathbb{E}[\hat{f}(X)|\boldsymbol{X}, X]\right)\left(\mathbb{E}[\hat{f}(X)|\boldsymbol{X}, X] - f(X)\right) |\boldsymbol{X}, X\right] \\
&= \left(\mathbb{E}[\hat{f}(X)|\boldsymbol{X}, X] - f(X)\right)\mathbb{E}\left[\left(\hat{f}(X) - \mathbb{E}[\hat{f}(X)|\boldsymbol{X}, X]\right) |\boldsymbol{X}, X\right] = 0. \quad \square
\end{aligned}$$

Note: The proof given in the slides of lecture 1 has further details.

- From the bias-variance decomposition, we can see that even if our prediction is unbiased,

$$\mathbb{E}\left[\hat{f}(X)|\boldsymbol{X}, X\right] = f(X),$$

we can still incur a large error if it is highly variable. Meanwhile, even when our prediction is stable and not variable, we can incur a large error if it is badly biased

- Typically, underfitting means high bias and low variance, overfitting means a lower the bias but a higher the variance. In other words, the more complex we make the model $\hat{f}(x)$, the lower the bias but the higher the variance. So there is a bias-variance **tradeoff**.

5.2.3 Training error vs. test error

Typically, the training error $\text{Err}_{\text{train}}$ will be less than the test error Err_{test} because the fitting method adapts to the training data, and will hence be an overly **optimistic** estimate of the test error.

Part of this discrepancy is due to where the evaluation points occur. *The model is trained to minimise the error at those evaluation points so we want to know how badly it could possibly do in this best case.* So to make it easier to compare our training error with our test error, we define **in-sample test error**. This is the test error if we receive new samples of Y corresponding to each covariate vector in our existing data. Define

$$\text{Err}_{\text{in-sample}} = \frac{1}{n}\sum_{i=1}^{n}\mathbb{E}\left[\left(Y - \hat{f}(X)\right)^2 |\boldsymbol{X}, \boldsymbol{Y}, X = \boldsymbol{X}_i\right] \tag{5.6}$$

Note that the only randomness in the preceding expression is in the new observations Y.

Intuitively, if we can adjust $\text{Err}_{\text{train}}$ to behave more like $\text{Err}_{\text{in-sample}}$, we would have a way to estimate the test error on new data (at least, for covariates $\boldsymbol{X}_i$ we have already seen).

For squared error:

$$\mathbb{E}\left[\text{Err}_{\text{in-sample}}|\boldsymbol{X}\right] = \mathbb{E}\left[\text{Err}_{\text{train}}|\boldsymbol{X}\right] + \frac{2}{n}\sum_{i=1}^{n}\text{Cov}\left(\hat{f}(\boldsymbol{X}_i), Y_i|\boldsymbol{X}\right). \tag{5.7}$$

where the expectations are taken over the random outcomes $\boldsymbol{Y}$.

The proof of Equation (5.7) is given below to demonstrate the construction of this important result. *The proof itself will not be examinable.*

Proof. If we expand the definitions of $\text{Err}_{\text{train}}$ and $\text{Err}_{\text{in-sample}}$ we have:

$$\text{Err}_{\text{train}} - \text{Err}_{\text{in-sample}} = \frac{1}{n}\sum_{i=1}^{n}\left(\mathbb{E}[Y^2|X = \boldsymbol{X}_i] - Y_i^2 - 2(\mathbb{E}[Y|X = \boldsymbol{X}_i] - Y_i)\hat{f}(\boldsymbol{X}_i)\right),$$

taking expectations over $\boldsymbol{Y}$, we use the fact that

$$\mathbb{E}[Y^2|\boldsymbol{X}, X = \boldsymbol{X}_i] = \mathbb{E}[Y_i^2|\boldsymbol{X}]$$

since both are the expectation of the square of a random outcome with associated covariate $\boldsymbol{X}_i$, so we have

$$\mathbb{E}\left[\text{Err}_{\text{train}} - \text{Err}_{\text{in-sample}}|\boldsymbol{X}\right] = -\frac{2}{n}\sum_{i=1}^{n}\mathbb{E}\left[(\mathbb{E}[Y|X = \boldsymbol{X}_i] - Y_i)\hat{f}(\boldsymbol{X}_i)|\boldsymbol{X}\right]$$

also for the same reason

$$\mathbb{E}[Y|X = \boldsymbol{X}_i] = \mathbb{E}[Y_i|X = \boldsymbol{X}].$$

Finally, since

$$\mathbb{E}\left[Y_i - \mathbb{E}[Y_i|\boldsymbol{X}]|\ \boldsymbol{X}\right] = 0,$$

we get

$$\begin{aligned}\mathbb{E}\left[\text{Err}_{\text{train}} - \text{Err}_{\text{in-sample}}|\boldsymbol{X}\right] = -\frac{2}{n}\sum_{i=1}^{n}\Bigg(&\mathbb{E}\left[(Y_i - \mathbb{E}[Y|X = \boldsymbol{X}_i])\hat{f}(\boldsymbol{X}_i)\big|\boldsymbol{X}\right] \\ &+ \mathbb{E}[Y_i - \mathbb{E}[Y_i|\boldsymbol{X}]|\boldsymbol{X}]\mathbb{E}[\hat{f}(\boldsymbol{X}_i)\big|\boldsymbol{X}]\Bigg)\end{aligned}$$

which reduces to

$$\frac{2}{n}\sum_{i=1}^{n}\text{Cov}\left(\hat{f}(\boldsymbol{X}_i), Y_i|\boldsymbol{X}\right),$$

and the result is obtained.

□

The second term in the result in Equation (5.7) is what Hastie et al. [2008] calls the **average optimism**. It is the amount by which the training error underestimates the true error depends on how strongly Y_i affects its own prediction. **The harder we fit the data, the greater $\text{Cov}(\hat{f}(\boldsymbol{X}_i), Y_i|\boldsymbol{X})$ will be, thereby increasing the optimism.** In practice, for any reasonable model, we should expect our predictions to be positively correlated with our outcome. In particular, if $\text{Cov}(\hat{f}(\boldsymbol{X}_i), Y_i|\boldsymbol{X}) > 0$ then the training error underestimates test error. This result holds more generally for other measures of test error, for example, zero-one loss in classification, and other loss functions.

Example 1. Linear regression

For a model $\hat{f}$ with linear fit with p inputs, we have

$$\text{Cov}\left(\hat{f}(\boldsymbol{X}_i), Y_i|\boldsymbol{X}\right) = p\sigma^2. \tag{5.8}$$

Then by Equation (5.7)

$$\mathbb{E}\left[\text{Err}_{\text{in-sample}}|\boldsymbol{X}\right] = \mathbb{E}\left[\text{Err}_{\text{train}}|\boldsymbol{X}\right] + \frac{2p\sigma^2}{n}. \tag{5.9}$$

Consider the optimism described by the second term, we can see that it

- increases with σ^2: more noise gives the model more opportunities to seem to fit well by capitalizing on chance,
- decreases with n: at any fixed level of noise, more data makes it harder to pretend the fit is better than it really is,
- and increases with p: every extra parameter is another control which can be adjusted to fit to the noise.

This leads to **Mallow's C_p statistic** (1973) defined as:

$$C_p = \text{Err}_{\text{train}} + \frac{2p}{n}\hat{\sigma}^2, \tag{5.10}$$

where $\hat{\sigma}^2$ is an estimate of the noise variance σ^2 obtained from the mean squared error, and can be seen as an approximation to Equation (5.9). The first term can be interpreted as a measure of the fit to the existing data, and the second term as a penalty for model complexity. So the C_p statistic balances underfitting and overfitting the data (bias and variance). **The selection rule is to pick the model which minimizes C_p.**

Remark 10. *In this example with linear regression,* ***model complexity*** *is described by the number of covariates. In the next sections we present model selection criteria AIC and BIC that apply to general models, beyond linear regression, but capture model complexity. We omit details in their general construction, we will see mimic C_p by estimating the optimism and add it to the training error. In contrast, cross-validation and bootstrap methods, described in Section 5.3, are direct estimates of the* ***expected test error*** *$\mathbb{E}[Err_{test}]$.*

5.2.4 AIC

Given a set of models $f_\alpha(x)$ indexed by a tuning parameter α, define the AIC (**Akaike information criterion**):

$$\text{AIC}(\alpha) = \text{Err}_{\text{train}}(\alpha) + 2\frac{d(\alpha)}{n}\hat{\sigma}^2, \tag{5.11}$$

where $\text{Err}_{train}(\alpha)$ and $d(\alpha)$ is the training error and the number of parameters for each model.

As can be recognised from the second term in Equation (5.7), C_p, AIC (and later BIC) can be described as using **covariance penalties**. For nonlinear and other complex models, we need to replace $d(\alpha)$ by some measure of model complexity, akin to the number of parameters. The function $\text{AIC}(\alpha)$ provides an estimate of the test error curve, and we find the tuning parameter α^* that minimizes it. **We choose the model giving smallest AIC over the set of models considered**.

As James et al. [2021] noted, that even for this case of AIC for least squares regression, detailed derivations of the formulae are difficult, so not included in that text. Further details are in [Hastie et al., 2008, Chap. 7].

5.2.5 BIC

BIC (**Bayesian Information Criteria**) is a variant of AIC with a stronger penalty for including additional variables to the model. BIC is derived from a Bayesian point of view, but ends up looking similar to C_p (and AIC) as well. The BIC is given by

$$\text{BIC}(\alpha) = \text{Err}_{\text{train}}(\alpha) + \log(n)\frac{d(\alpha)}{n}\hat{\sigma}^2 \tag{5.12}$$

It can be seen that BIC replaces the 2 in AIC with $\log(n)$ for BIC. Since $\log(n) > 2$ for any $n > 7$, BIC generally places a heavier penalty on models with many variables and hence selects smaller models than AIC.

Remark 11. *Examples using AIC and BIC will be provided in lectures. See also [James et al., 2021, Chap. 6].*

5.2.6 The Bayesian approach

In model selection the data are used to select one of the models under consideration. An alternative to selecting one model and basing all further work on this one model is that of **model averaging** [Claeskens and Hjort, 2008, Chap 7]. Bayesian model averaging computes posterior probabilities for each of the models and uses those probabilities as weights.

We define the Bayesian approach to model selection. Suppose we have a set of candidate models $\mathcal{M}_m$, $m = 1, \ldots, M$, with corresponding model parameters θ_m and we wish to choose a best model from among them. Assuming we have a prior distribution $P[\theta_m|\mathcal{M}_m]$ for the parameters of each model $\mathcal{M}_m$, the posterior probability of a given model is

$$P[\mathcal{M}_m|\boldsymbol{X}, \boldsymbol{Y}] \propto P[\mathcal{M}_m]P[\boldsymbol{X}, \boldsymbol{Y}|\mathcal{M}_m], \tag{5.13}$$

$$\propto P[\mathcal{M}_m] \int P[\boldsymbol{X}, \boldsymbol{Y}|\theta_m, \mathcal{M}_m]P[\theta_m|\mathcal{M}_m]d\theta_m. \tag{5.14}$$

To compare two models $\mathcal{M}_m$ and $\mathcal{M}_\ell$ we can write the **posterior odds** as:

$$\frac{P[\mathcal{M}_m|\boldsymbol{X},\boldsymbol{Y}]}{P[\mathcal{M}_\ell|\boldsymbol{X},\boldsymbol{Y}]} = \frac{P[\mathcal{M}_m]}{P[\mathcal{M}_\ell|}\frac{P[\boldsymbol{X},\boldsymbol{Y}|\mathcal{M}_m]}{P[\boldsymbol{X},\boldsymbol{Y}|\mathcal{M}_\ell]}. \tag{5.15}$$

If the odds are greater than one we choose model m, otherwise we choose model ℓ. The rightmost quantity is called the **Bayes factor**:

$$\frac{P[\boldsymbol{X},\boldsymbol{Y}|\mathcal{M}_m]}{P[\boldsymbol{X},\boldsymbol{Y}|\mathcal{M}_\ell]}, \tag{5.16}$$

which is the contribution of the data toward the posterior odds. Typically it is assumes that the prior over models is uniform, so that $P[\mathcal{M}_m]$ is constant.

Approximations are often employed for the integral in Equation (5.14), for example, using a **Laplace approximation**.

5.3 Resampling methods

In an ideal situation, as discussed in Section 5.2.1, if we had enough data, we would set aside a validation set and use it to assess the performance of our model. But in practice this is not usually possible. We present two widely used resampling techniques: cross-validation and bootstrap, to directly estimate of the **test error** Err_{test}.

5.3.1 Cross-validation

Given a sufficient number of samples, we can set aside some data for a validation set to estimate the test error. When we feel we do not have a sufficient number of samples cross validation; this still uses part of the data to fit our model, and another part of the data to test it, and this process is repeated.

Divide the data into K parts (folds) $F_1, \ldots, F_K$, so

$$F_1 \cup \ldots F_K = \{1, \ldots, n\}. \tag{5.17}$$

For each $k = 1, \ldots, K$, we fit our model to all points but those in the k^{th} fold, i.e., to $K-1$ parts of the data, and calculate the error of the fitted model when predicting the k^{th} part of the data, denoted by $\hat{f}^{-(k)}$ and evaluate error on the points in the k^{th} fold:

$$\text{CV}_k(\hat{f}^{-(k)}) = \frac{1}{n_k}\sum_{i\in F_k}\left(Y_i - \hat{f}^{-(k)}(X_i)\right)^2, \tag{5.18}$$

where n_k is the number of points the k^{th} fold, $n_k = |F_k|$. We then average these fold-based errors to yield an estimate of the **test error** Err_{test}:

$$\text{CVErr}(\hat{f}) = \frac{1}{K}\sum_{i=1}^{K}\text{CV}_k(\hat{f}^{-(k)}), \tag{5.19}$$

this is known as the K**-fold cross-validation**.

After running a K-fold cross validation, we build a model $\hat{f}$ from all the training data and the idea is that $\text{CVErr}(\hat{f})$ should be a good estimate of Err_{test} on new samples. This brings us to the question of what value of K to choose? Bias and variance play a role.

- If $K = n$, the resulting method is called leave-one-out cross-validation (LOOCV). In this case, the cross-validation estimator $\text{CVErr}(\hat{f})$ is approximately unbiased for the true test error, but can have high variance because the n training sets are so similar to one another. The computational burden may also be considerable, requiring n applications of the learning method.

- In comparison, If $K = 5$ or 10, (common choices), the mean of our estimate $\text{CVErr}(\hat{f})$ is now a little bit farther off Err_{test}. But the variance of the error has decreased, since when calculating $\text{CVErr}(\hat{f})$, the quantities it is averaging over are less correlated (than they were for $K = n$), because the fits $\hat{f}^{-(k)}$, $k = 1, \ldots, K$ are not dependent on as much overlapping data. Note that the variance of the sum of highly correlated quantities is larger than that with midly correlated quantities

For K-fold cross-validation, it can be helpful to assign a quantitative notion of variability to the cross-validation error estimate. Consider

$$\text{Var}\left(\text{CVErr}(\hat{f})\right) = \text{Var}\left(\frac{1}{K}\sum_{i=1}^{K}\text{CV}_k(\hat{f}^{-(k)})\right) \tag{5.20}$$

$$= \frac{1}{K^2}\text{Var}\left(\sum_{i=1}^{K}\text{CV}_k(\hat{f}^{-(k)})\right) \tag{5.21}$$

$$\approx \frac{1}{K}\text{Var}\left(\text{CV}_1(\hat{f}^{-(1)})\right) \tag{5.22}$$

Why is this an approximation? This would hold exactly if $\text{CV}_1(\hat{f}^{-(1)}), \ldots, \text{CV}_K(\hat{f}^{-(K)})$ were i.i.d., but they're not. This approximation is valid for small $K = 5$ or 10 but not really for large K (e.g., $K = n$).

For small K, we employ the approximation in Equation 5.20 to obtain an estimate of the **cross-validation error estimate**:

$$\frac{1}{K}\text{var}\left\{\text{CV}_1(\hat{f}^{-(1)}), \ldots, \text{CV}_K(\hat{f}^{-(K)})\right\} \tag{5.23}$$

where $\text{var}(\cdot)$) denotes the sample variance operator, and hence the **standard error (deviation) of the cross-validation estimate** is

$$\frac{1}{\sqrt{K}}\text{sd}\left\{\text{CV}_1(\hat{f}^{-(1)}), \ldots, \text{CV}_K(\hat{f}^{-(K)})\right\}, \tag{5.24}$$

where sd denotes the sample standard deviation.

To choose between models in practice, we simply compute cross-validated errors for each, and then take the model with the minimum cross-validated error.

For tuning parameter selection (for example, the k in k-nearest neighbors), write $\hat{f}_\alpha$ to denote that our fitting mechanism depends on some parameter α, then for a range of parameters $\alpha_1, \ldots, \alpha_m$ of interest we compute:

$$\text{CVErr}(\hat{f}_{\alpha_1}), \ldots, \text{CVErr}(\hat{f}_{\alpha_m}), \tag{5.25}$$

and choose the value of minimizing the cross-validation error curve (a curve over α):

$$\hat{\alpha} = \underset{\alpha_1,\ldots,\alpha_m}{\arg\min}\, \text{CVErr}(\hat{f}_\alpha) \tag{5.26}$$

The correct way to do cross-validation

We present Hastie et al. [2008]'s example based on a classification example, but note the same implications stand for regression problems. Consider a classification problem with a large number of predictors, where a typical strategy for analysis might be as follows:

1. Screen the predictors: find a subset of 'good' predictors that show fairly strong (univariate) correlation with the class labels
2. Using just this subset of predictors, build a multivariate classifier.
3. Use cross-validation to estimate the unknown tuning parameters and to estimate the prediction error of the final model.

Is this a correct application of cross-validation? Consider a scenario with $n = 50$ samples in two equal size classes and $p = 5000$ quantitative predictions (standard Gaussian) that are independent of the class labels. The true error rate of any classifier is 50%. Using the above three steps, choosing in step (1) the 100 predictors having highest correlation with the class labels, and then using a 1-nearest neighbor classifier, based on just these 100 predictors, in step (2). Over 50 simulations from this setting, the average cross validation error rate was 3%, a value far lower than the true error rate of 50%.

The problem is that the predictors have an unfair advantage, as they were chosen in step (1) on the basis of all of the samples. Leaving samples out after the variables have been selected does not correctly mimic the application of the classifier to a completely independent test set, since these predictors ?have already seen? the left out samples. In this example, the correct way to carry out cross-validation is:

1. Divide the samples into K cross-validation folds at random.
2. For each fold $k = 1, \ldots, K$,

(a) Find a subset of 'good' predictors that show fairly strong (univariate) correlation with the class labels, using all of the samples except those in fold k.

(b) Using just this subset of predictors, build a multivariate classifier, using all of the samples except those in fold k.

(c) Use the classifier to predict the class labels for the samples in fold k

In general, with a multistep modelling procedure, cross-validation must be applied to the entire sequence of modelling steps to get unbiased results.

5.3.2 Bootstrap

The bootstrap is one of the most general and the most widely used tools to estimate measures of uncertainty associated with a given statistical method. Some common bootstrap applications are: estimating the bias or variance of a particular statistical estimator, or constructing approximate confidence intervals for parameters of interest [Tibshirani, 2014].

Suppose that we have independent samples $\boldsymbol{Y} = \{Y_1, \ldots, Y_n\}$, where the samples are drawn from a population, where we are interested in a population level parameter θ (noting that it is parameter of the population, not a property of the sample). For example, θ could be the mean of the distribution, the variance, or something more complicated. Let $\hat{\theta}$ be an estimate for θ that we can compute from the samples $Y_1, \ldots, Y_n$.

Suppose we are interested in the variance $\mathrm{Var}(\hat{\theta})$ of the statistic $\hat{\theta}$. If we had access to the population, we could just draw another fresh n samples, recompute the statistic, and repeat say, 1000 times, we would have computed 1000 statistics and could just use the sample variance of these statistics.

However, without access to the population, **the idea** behind the bootstrap is to use the observed samples $Y_1, \ldots, Y_n$ to generate n "new" samples, as if they came from the population. Denoting the new samples by $\tilde{Y}_1, \ldots, \tilde{Y}_n$, we draw these according to

$$\tilde{Y}_j \sim U\{Y_1, \ldots, Y_n\}, \quad i = 1, \ldots, n, \tag{5.27}$$

in other words, each $\tilde{Y}_j$ is independent and drawn uniformly from $Y_1, \ldots, Y_n$. This is called *sampling with replacement* and each $\tilde{Y}_1, \ldots, \tilde{Y}_n$ is called a **bootstrap sample**. We can write this as an algorithm:

Algorithm 12: Bootstrap algorithm

Given sample $\boldsymbol{Y}$ of n observations.

For $b = 1, \ldots, B$:

1. Draw n samples uniformly at random, with replacement, from $\boldsymbol{Y}$. Denote the i^{th} observation in the b^{th} sample by $\tilde{Y}_i^{(b)}$
2. Compute the value of the statistic of interest $\tilde{\theta}^{(b)}$ from the b^{th} sample

Note: This notation in this section is slightly varied from those given in the lecture slides.

The histogram (that is, the empirical distribution) of $\{\tilde{\theta}^{(b)}, b = 1, \ldots, B\}$ is an estimate of the sampling distribution and it is called the **bootstrap distribution**.

To get an estimate for $\mathrm{Var}(\hat{\theta})$, just draw a bootstrap sample, recompute our statistic, repeat many times, and finally compute the sample variance over the statistics, as we would have done with population samples directly, if we had access to the population.

Example

We consider an example from James et al. [2021], to demonstrate how estimates are constructed *(for further details and visualisations see the reference)*.

Suppose that we have two random variables $X, Y \in \mathbb{R}$ which represent the yields of two financial assets. We will invest a fraction of our money θ in X, and the remaining fraction $1 - \theta$ in Y. The yield will then be the random quantity:

$$\theta X + (1 - \theta)Y,$$

from which we may want to choose θ to minimize the variance of our investment. One can show that the value of θ minimizing $\mathrm{Var}(\theta X + (1 - \theta)Y)$ is

$$\theta = \frac{\sigma_Y^2 - \sigma_{XY}}{\sigma_X^2 + \sigma_Y^2 - 2\sigma_{XY}},$$

where $\sigma_X^2 = \mathrm{Var}(X)$, $\sigma_Y^2 = \mathrm{Var}(Y)$ and $\sigma_{XY} = \mathrm{Cov}(X, Y)$. In reality, $\sigma_X^2, \sigma_Y^2, \sigma_{XY}^2$ are all unknown. Given $n = 100$ samples of past (paired) measurements we can compute estimates $\hat{\sigma}_X^2$, $\hat{\sigma}_Y^2$ and $\hat{\sigma}_{XY}$. Therefore our estimate will be

$$\hat{\theta} = \frac{\hat{\sigma}_Y^2 - \hat{\sigma}_{XY}}{\hat{\sigma}_X^2 + \hat{\sigma}_Y^2 - 2\hat{\sigma}_{XY}},$$

It is natural to wish to quantify the accuracy of our estimate of θ. In this toy example, if the parameters are set to $\sigma_X^2 = 1, \sigma_Y^2 = 1.25, \sigma_{XY}^2 = 0.5$, so the true value of $\theta = 0.6$. Further, 1000 simulations can be generated to estimate the sample standard deviation to be $\mathrm{SE}(\hat{\theta}) = 0.083$ [James et al., 2021]. So roughly speaking, for a random sample from the population, we would expect $\hat{\theta}$ to differ from θ by approximately 0.08, on average.

In practice, the above procedure for estimating $\mathrm{SE}(\hat{\theta})$ cannot be applied, because for real data we cannot generate new samples. Picking a large number, $B = 1000$, using the bootstrap algorithm to estimate the standard error of our estimator $\hat{\theta}$, will be the sample standard deviation of the bootstrap statistics $\tilde{\theta}^{(1)}, \ldots, \tilde{\theta}^{(B)}$:

$$\mathrm{SE}(\hat{\theta}) \approx \sqrt{\frac{1}{B}\sum_{b=1}^{B}\left(\tilde{\theta}^{(b)} - \frac{1}{B}\sum_{r=1}^{B}\tilde{\theta}^{(r)}\right)^2} \tag{5.28}$$

The bootstrap estimate SE($\hat{\theta}$) from Equation (5.28) is 0.087, very close to the estimate of 0.083 obtained using 1,000 simulated data sets. The results are shown in Figure 5.2, where inspecting the bootstrap distribution (right panel) compared to the simulated datasets (left panel), indicates that here the bootstrap approach effectively estimates the variability associated with $\hat{\theta}$.

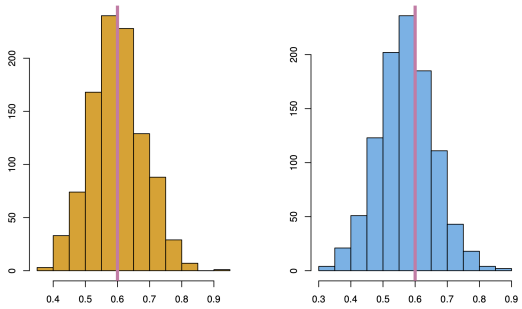


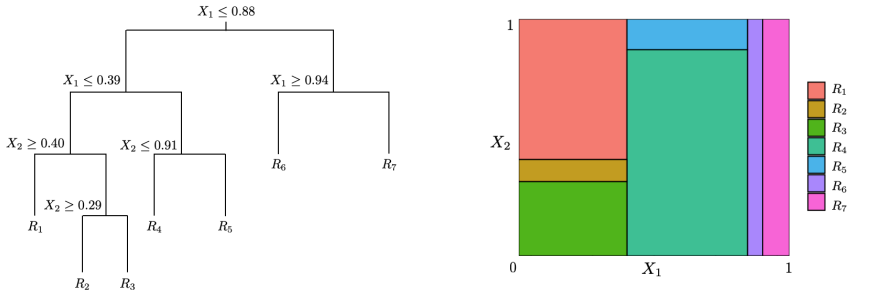
Figure 5.2: Left: A histogram of the estimates of θ obtained by generating 1000 simulated data sets from the true population. Right: A histogram of the estimates of θ obtained from 1000 bootstrap samples from a single data set.[James et al., 2021].

Why does the Bootstrap Work? *(This is not examinable!).*
We refer the reader to: `https://www.stat.cmu.edu/~larry/=sml/Boot.pdf`

5.4 Tree-based models

We now examine some additional statistical machine learning methods. Tree-based methods, specifically random forests represent the current state of the art for predictive performance of a single model on so-called tabular data. Tabular data is where we have observations on variables often arranged into a design matrix. Data which is not tabular is called unstructuref (e.g., images and video). The methods in this section still perform well on unstructured data, but deep learning (in the next section of the course), typically achieves the highest performance in unstructured settings.[Aslett, 2022]. Tree-based models are conceptually straightforward but powerful techniques for handling data where the relationship between features and outcome is non-linear, or where features interact with one another James et al. [2021].

A tree is a data structure representing a partition of the predictor space into distinct regions. In a statistical machine learning context, these regions are then treated as having a constant fixed prediction value for any observation whose feature vector lies within each given region, (that is, a constant in regression and a class in classification). There are different ways to create such partitions, but for tractability reasons it is very common to see only binary trees considered in the literature.



(a) Example tree

(b) Partition of the predictor space

Figure 5.3: Adapted from Aslett [2022]

Consider a regression problem with continuous response Y and inputs X_1 and X_2, each taking values in the unit interval. For this problem consider the tree in Figure 5.3a. The full dataset sits at the top of the tree, called the **root node**. Observations satisfying the condition at each junction are assigned to the left branch, and the others to the right branch. This root node is then divided into two branches representing one half of a partition of the whole space into two components. This process is then recursively applied to each child node to create a successive partition until some termination rule, resulting in nodes that are no longer further subdivided, these are called **leaf nodes** (also called a terminal nodes) and defines a region of the space R_i. The corresponding region R_i of the predictor space represented by each leaf is illustrated in Figure 5.3b.

For a tree T, we define **tree size** $|T|$ as the number of leaf nodes in T, as we will see, this is a tuning parameter that governs the model complexity and will be adaptively chosen from the data. Define the **depth of a node** to be the length of the path from the root to the node, so the node "$X_2 \leq 0.91$" is at depth 2, and define the **height of the tree** to be the maximum over all depths, so the tree above has height 4. In Figure 5.3a, the first split is at $X_1 = 0.88$, and the process continues. The result of this process is a partition into 7 regions $R_1, \ldots, R_M$ shown in Figure 5.3b with $M = 7$. The corresponding regression model predicts Y as a constant c_m in region R_m, that is

$$\hat{f}(X) = \sum_{m=1}^{M} c_m I\left(X \in R_m\right). \tag{5.29}$$

5.4.1 Regression Trees

Given data (X_i, Y_i), for $i = 1, 2, \ldots, n$ and where $X_i = (X_{i1}, X_{i2}, \ldots, X_{ip}$, ($p$ inputs for each of the n observations and the corresponding response). The problem of constructing a tree on the basis of observed data, amounts to learning the split indices and values, and also what shape the tree should have.

We note that there are many, many algorithms that can grow a tree [Loh, 2014]. They differ in the possible structure of the tree (e.g. number of splits per node), the criteria how to find the splits, when to stop splitting and how to estimate the simple models within the leaf nodes. We introduce a well-known class of algorithms called CART (Classification And Regression Tree) following Hastie et al. [2008].

How to construct the regions $R_1, \ldots, R_M$? In theory, the regions could have any shape. However, we choose to divide the predictor space into high-dimensional rectangles, for simplicity and for ease of interpretation of the resulting predictive model. A goal is to find boxes $R_1, \ldots, R_M$ that minimise square error, given by:

$$\sum_{m=1}^{M} \sum_{i \in R_j} \left(Y_i - \hat{Y}_{R_j} \right)^2, \tag{5.30}$$

where $\hat{Y}_{R_j}$ is the mean response for the training observations within the j^{th} box. But it is computationally infeasible to consider every possible partition of the predictor space into J boxes.

For this reason, we take greedy approach that is known as **recursive binary splitting**. The approach begins at the top of the tree (at which point all observations belong to a single region) and then successively splits the predictor space. It is **greedy** because at each step of the tree-building process, the best split is made at that particular step, rather than looking ahead and picking a split that will lead to a better tree in some future step.

Starting with all of the data, consider a splitting variable j and split point s, then define half-planes:

$$R_1(j, s) = \left\{ X | X_j \leq s \right\} \quad \text{and} \quad R_2(j, s) = \left\{ X | X_j > s \right\}. \tag{5.31}$$

Then we seek the splitting variable j and split point s that minimise

$$\sum_{X_i \in R_1(j,s)} (Y_i - \hat{c}_1)^2 + \sum_{X_i \in R_2(j,s)} (Y_i - \hat{c}_2)^2, \tag{5.32}$$

where $\hat{c}_1$ is the mean response for the training observations in $R_1(j, s)$, and similarly for $\hat{c}_2$ with $R_2(j, s)$, that is,

$$\hat{c}_1 = \text{ave}\left(Y_i | X_i \in R_1(j, s) \right) \quad \text{and} \quad \hat{c}_2 = \text{ave}\left(Y_i | X_i \in R_2(j, s) \right) \tag{5.33}$$

For each splitting variable, the determination of the split point s can be done quickly and hence by scanning through all of the inputs, determination of the best pair (j, s) is feasible. Having found the best split, we partition the data into the two resulting regions and repeat the splitting process on each of the two regions. Then this process is repeated on all of the resulting regions.

How large should we grow the tree? A very large tree might overfit the data, while a small tree might not capture the important structure. Tree size is a tuning parameter governing the model's complexity, and the optimal

tree size should be adaptively chosen from the data. The strategy presented here is to grow a large tree T_0 (the full tree), stopping the splitting process only when some minimum node size (say 5) is reached. Then this large tree is pruned using **cost-complexity pruning** (also known as weakest-link pruning), which we now describe.

Define a subtree $T \subset T_0$, to be any tree that can be obtained by pruning T_0, that is, collapsing any number of its internal nodes. Indexing leaf nodes by m, representing region R_m.

Let the number of observations in a region R_m be

$$N_m = \left|\{X_i \in R_m\}\right|, \tag{5.34}$$

and define the prediction in region R_m (calculated by the mean of observations in that region) be

$$\hat{c}_m = \frac{1}{N_m} \sum_{X_i \in R_m} Y_i. \tag{5.35}$$

Then the **node impurity** function, here measured using squared error, is defined as

$$Q_m(T) = \frac{1}{N_m} \sum_{X_i \in R_m} (Y_i - \hat{c}_m)^2 \tag{5.36}$$

The concept of node impurity is a measure of how badly the observations at a given node fit the model. (This is easier to understand in the classification tree, where a 'pure' node only contains the data from a single class.)

For a tuning parameter α, we define the cost-complexity criterion $C_\alpha(T)$ for a tree T as

$$C_\alpha(T) = \sum_{m=1}^{|T|} N_m Q_m(T) + \alpha|T|. \tag{5.37}$$

Our goal is to find the subtree T_α for a given α which minimises $C_\alpha(T)$. If we set $\alpha = 0$, then we will attain the tree T_0, by Equation (5.37) just measures the training error. As α increases, there is a cost for having a tree with many terminal nodes, so $C_\alpha(T)$ will tend to be minimized for a smaller subtree. *Further pictorial details and examples are given in the lecture slides.*

It turns out that as we increase α from zero, branches get pruned from the tree in a nested and predictable fashion, so obtaining the whole sequence of subtrees is easy. We can select a value of α using cross-validation. We then return to the full data set and obtain the subtree corresponding to α. This process is summarized in Algorithm 13.

Algorithm 13: Building a regression tree

1. Use recursive binary splitting to grow a large tree on the training data, stopping when each terminal node has fewer than some minimum number of observations
2. Apply cost complexity pruning to a large tree in order to obtain a sequence of best subtrees, as a function of α
3. Use k-fold CV to choose α. That is, divide the training obserations into K folds. For each $k = 1, \ldots, K$
   (a) Repeat steps 1 and 2 on all but the k^{th} fold of the training data
   (b) Evaluate the mean squared error on the data in the left out k^{th} fold (we called this $CV_k(\hat{f}^{(-k)})$), as a function of α
   (c) Average the results for each α, and choose α to minimise the average error
4. Return the subtree from step 2 that corresponds to the chosen value of α

Given a tree, performing prediction is quite straightforward and the decisions which lead to a particular prediction are clear. Further, a key advantage of the binary tree is its interpretability. But trees are also quite unstable, that is, a few changes in the training dataset can create a completely different tree. This is because each split depends on the parent split, and if a different feature is selected as the first split feature, the entire tree structure changes. It does not create confidence in the model if the structure changes so easily. This forms the motivation for Bagging, that we will introduce in Section 5.4.3

5.4.2 Classification trees

If the target is a classification outcome taking values $1, 2, \ldots, , K$, the only changes needed in the tree algorithm relate to the criteria for splitting nodes and pruning the tree. For regression we used the squared-error node impurity measure $Q_m(T)$ in Equation (5.36).

For a node m representing R_m, with N_m observations, define **the proportion of class k observations in node m**:

$$\hat{p}_{mk} = \frac{1}{N_m} \sum_{X_i \in R_m} I\left(Y_i = k\right). \tag{5.38}$$

We classify the observations in node m to class

$$k(m) = \text{argmax}_k \hat{p}_{mk}, \tag{5.39}$$

the majority class in node m. Using $\hat{p}_{mk}$ and $k(m)$, we introduce some classification impurity measures:

- **Misclassification error**: For region R_m,

$$\frac{1}{N_m} \sum_{X_i \in R_m} \mathbb{1}\left\{Y_i \neq k(m)\right\} = 1 - \hat{p}_{mk(m)}.$$

- **Gini index**: For region R_m,

$$\sum_{k \neq k'} \hat{p}_{mk} \hat{p}_{mk'} = \sum_{k=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk}).$$

- **Cross-entropy/Deviance**: For region R_m,

$$-\sum_{k=1}^{K} \hat{p}_{mk} \log \hat{p}_{mk}.$$

For the case of $K = 2$, with p_2 the proportion of points in the second class (in region m), we have:

$$\begin{aligned} \text{Misclassification} &= 1 - \max\left(p_2, 1 - p_2\right), \\ \text{Gini} &= 2p_2(1 - p_2), \\ \text{Cross-entropy} &= -p_2 \log p_2 - (1 - p_2) \log(1 - p_2). \end{aligned}$$

We illustrate the value of these three impurity measures as a function of p_2 in Figure 5.4. All three are similar, but cross-entropy and the Gini index are differentiable, and hence more amenable to numerical optimization. Cross-entropy and the Gini index are more sensitive to changes in the node probabilities than the misclassification rate, *see example given in the lecture slides*. The Gini index or cross-entropy should be used when growing the tree. To guide cost-complexity pruning, any of the three measures can be used, but typically it is the misclassification rate.

5.4.3 Bagging

Decision trees suffer from high variance, meaning that if we split the training data into two parts at random, and fit a decision tree to both halves, the results that we get could be quite different. A method to reduce the variance is Bagging (**B**ootstrap **Agg**regat**ing**).

Earlier in this chapter, we introduced the bootstrap in Section **??**) to study the variability of predictions. Now, we will use the average of these predictions as an estimator with reduced variance. Think about the following: given a set of n observations $X_1, \ldots, X_n$, each with variance σ^2, the variance of the mean of the observations is σ^2/n.

Hence a natural way to reduce the variance and increase the test set accuracy of a statistical learning method is to take many training sets from the population, build a separate prediction model using each training set, and average the

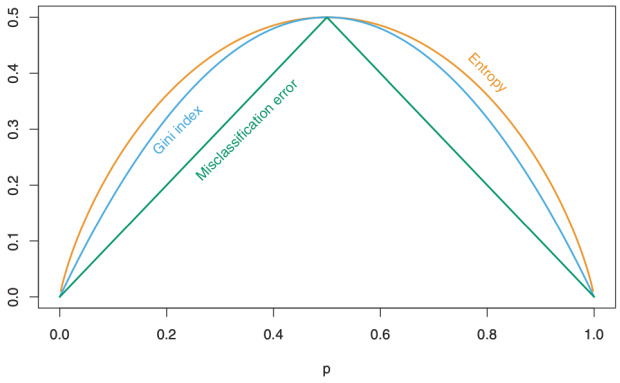


Figure 5.4: Node impurity measures for two-class classification, as a function of the proportion p_2 in class 2. Cross-entropy has been scaled to pass through (0.5, 0.5) [Hastie et al., 2008].

resulting predictions. Of course, this is not practical because we do not have access to multiple training sets. Instead, we can bootstrap, by taking repeated samples from the (single) training data set.

Consider the regression problem, where we fit a model to our training data. For each bootstrap sample $\tilde{\boldsymbol{Y}}^{(b)}$ for $b = 1, \ldots, B$, we fit our tree, that gives prediction $\hat{f}^{(b)}(X)$ at input vector X. Then the **bagging estimate** is

$$\hat{f}_{\text{bag}}(X) = \frac{1}{B} \sum_{b=1}^{B} \hat{f}^{(b)}(X) \tag{5.40}$$

Each bootstrap tree will typically involve different features than the original and might have a different number of leaf nodes. The bagging estimate is the average prediction at X from these B trees.

For a classification tree, the bagged classifier selects the class with the majority votes from the B trees. That is, for a K-class response, then the bagging estimate $\hat{f}_{\text{bag}}(X)$ is a K-vector $[p_1(X), p_2(X), \ldots, p_K(X)]$ with $p_k(X)$ equal to the proportion of trees predicting class k at X.

We note that in bagging, we do not prune the trees as we did in Section 5.4. The pruning (Step 2) of the CART algorithm was used to lower the variance after growing 'deep trees'. Now each individual tree has high variance, but low bias - and we use aggregation to lower the variance.

There is a straightforward way to estimate the test error of a bagged model, without the need to perform cross-validation or the validation set approach. Consider a draw within a single bootstrap sample (of n observations): the probability for one observation *not* to be drawn *in a single draw* is $1 - \frac{1}{n}$. Then the probability that one observation is not to be drawn in any one of the n bootstrap draws is approximately: $\left(1 - \frac{1}{n}\right)^n \approx e^{-1} = 0.368 = 1 - 0.632$. Thus 36.8% of

observations are not used when fitting any single bootstrap resample. Likewise, for a single observation, it does not appear in about 36.8% of the fitted models Each tree is grown on a training sample of about 63%, Thus each tree brings its own test (out-of-bag) (OOB) sample.

For each sample X_i, find the prediction $\hat{f}^{(b)}(X_i)$ for all bootstrap samples b which do not contain X_i. Average these predictions to obtain $\hat{f}^{(b)}_{\text{OOB}}(X_i)$. The error averaging over all observations $i, \ldots, n$ is

$$\frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{f}^{(b)}_{\text{OOB}}(X_i))^2, \tag{5.41}$$

this is the **OOB error**. Similarly for classification, we have

$$\frac{1}{n}\sum_{i=1}^{n} I(Y_i \neq \hat{f}^{(b)}_{\text{OOB}}(X_i)). \tag{5.42}$$

Note: slightly different OOB notation was used in the lecture slides.

5.4.4 Random Forests

We introduced bagged trees in the previous section to reduce the variance of a single tree. A **random forest** is a bagged tree with modified splitting criteria to further reduce the variance.

Before we present the random forest algorithm, consider random variables $X_1, \ldots, X_B$ that are *identically distributed, but not necessarily independent*, with mean μ, variance σ^2 and pairwise correlation ρ. Given the inevitable overlap between the bootstrap samples used to grow each tree, the trees are not independent. The variance of the average of these random variables is

$$Var\left(\frac{1}{B}\sum_{i=1}^{B} X_i\right) = \rho\sigma^2 + \frac{1-\rho}{B}\sigma^2. \tag{5.43}$$

This result can be derived in a few steps, consider the left-hand-side of Equation (5.43):

$$Var\left(\frac{1}{B}\sum_{i=1}^{B} X_i\right) = \frac{1}{B^2}Var\left(\sum_{j=1}^{B} X_i\right), \tag{5.44}$$

using the property

$$Cov\left(\sum_{i=1}^{n} X_i, \sum_{j=1}^{m} Y_j\right) = \sum_{i=1}^{n}\sum_{j=1}^{m} Cov(X_i, Y_j), \tag{5.45}$$

we have

$$Var\left(\frac{1}{B}\sum_{i=1}^{B}X_i\right) = \frac{1}{B^2}\left(\sum_{j=1}^{B}Var(X_i) + \sum\sum_{i\neq j}Cov(X_i, X_j)\right). \qquad (5.46)$$

Using the definition $\rho = \frac{Cov(X,Y)}{\sigma_X \sigma_Y}$, the result in Equation (5.43) can be obtained.

As B increases, the second term disappears, and hence the size of the correlation of pairs of bagged trees limits the benefits of averaging.

The idea of random forests is to improve the variance reduction of bagging by reducing the correlation between the trees, without increasing the variance too much. This is achieved in the tree-growing process through random selection of the input variables:

1. Build B decision trees on bootstrapped samples and at each node of the tree
   - Randomly select m variables at random out of all p possible variables (independently for each node)
   - Find the best split from the randomly selected m variables
2. Grow the trees to maximum depth without pruning, use majority vote (classification) or average (regression) to get predictions for new data

The rationale: suppose that there is one very strong predictor in the data set. Then in the collection of bagged trees, most of the trees will use this predictor in the top split. On average $(p-m)/p$ of the splits will not consider the strong predictor, and so other predictors will have more of a chance. We can think of this process as decorrelating the trees, thereby making the average of the resulting trees less variable and hence more reliable.

There remains the question of how we should select the number m. A typical choice is $m = \sqrt{p}$, but is this beyond the course. For additional reading see [Hastie et al., 2008, Chapter 15].

Chapter 6

Deep learning

6.1 Lecture 1

One of the greatest revolutions of the past decade (which is still continuing now!) is the advent of *deep learning*. At a high level, deep learning concerns statistical learning (e.g. supervised learning: regression, classification or unsupervised learning) using massively overparameterized (complex) functions, which are learnt directly from data using gradient-based numerical methods.

Some general references for this part of the course: Goodfellow et al. [2016], Feizi [2020], Zhang et al. [2021].

6.1.1 Examples

Some fun examples here.

6.1.2 Motivation

Try to imagine writing a function which takes as input an image with 256×256 pixels, and outputs whether or not the image is a 'cat' (label 0) or a 'dog' (label 1). Let us write $\mathcal{X}$ for the set of such images. Given access to nothing more than a (very large) vector of raw pixel values, it is almost impossible for a human to manually write such a function, given the cat or dog could be in any orientation, pose, colour, breed...

So instead, one might consider a very large class of functions, which are described by a very large number of parameters $\theta \in \mathbb{R}^d$. (Think d is in the millions.) We hope that this class of possible functions is so large and rich, it could describe almost any function from $\mathcal{X} \to \{0, 1\}$.

Then, rather than hand-tuning the millions of parameters θ by hand, we use training data – a large quantity of labelled images of cats and dogs – and an optimization algorithm so that searching for the parameters is done automatically.

The first question, therefore, is how to construct such a class of highly expressive functions.

6.1.3 Definition of feedforward networks

The most fundamental example of such a class of functions are *feedforward neural networks* (FNNs), also known as the *multilayer perception* (MLP), sometimes referred to as an *artificial neural network*, *deep neural network*, or just generically as a *neural network*.

Let's suppose we are interested in the problem of regression for a p-dimensional input $x \in \mathbb{R}^p$ to predict some real-valued response $y \in \mathbb{R}$. We will write the output of the neural network for a given x as $f(x;\theta)$, where $\theta \in \mathbb{R}^m$ is the (very large) vector of parameters, which we will define shortly. In other words, we will construct a (highly nonlinear) function

$$f(\cdot;\theta) : \mathbb{R}^p \to \mathbb{R},$$

where $f(x;\theta)$ is our estimate for $\mathbb{E}[Y|X = x]$.

A feedforward neural network consists of a number of layers $L \geq 1$, where L referred to as the *depth* of the network. Each layer $l \in [L]$ consists of a number of neurons $n_l \geq 1$, each of which you should just think of as a box which can store a single real number. Since we are interested in predicting a real-valued response Y, we only need to output a single real number so our final (top) layer has only a single neuron $n_L = 1$. Extensions to multivariate response or classification when we need multiple output values is a straightforward extension to $n_L > 1$.

We denote the values of the neurons in layer l for a given input x as

$$\alpha^{(l)}(x;\theta) \in \mathbb{R}^{n_l}.$$

The neurons in a layer l are connected to the neurons in the layer above $l+1$ and the layer below, except of course for the bottom layer and the top layer. The simplest neural network architecture is the *fully connected* feedforward neural network, where each neuron is connected to every other neuron in the layers above and below. The values of the neurons are calculated *recursively*, given the values of the neurons in the layer below.

The bottom layer (which you can think of as layer $l = 0$) is known as the *input layer*, and will consist of p neurons, since our input is p-dimensional, and each neuron will hold one component of the input vector; we set $n_0 = p$ and

$$\alpha^{(0)}(x;\theta) := x \in \mathbb{R}^p.$$

For each other layer $l \in [L]$, the neurons are calculated in two steps, as

$$\begin{aligned}\tilde{\alpha}^{(l)}(x;\theta) &:= W^{(l)}\alpha^{(l-1)}(x;\theta) + b^{(l)} \in \mathbb{R}^{n_l},\\ \alpha^{(l)}(x;\theta) &:= \sigma(\tilde{\alpha}^{(l)}(x;\theta)) \in \mathbb{R}^{n_l}.\end{aligned}$$

Here, for each l, $W^{(l)}$ is an $n_l \times n_{l-1}$ matrix of *weights* and $b^{(l)} \in \mathbb{R}^{n_l}$ is a vector of *biases*. $W^{(l)}\alpha^{(l-1)}(x;\theta)$ simply denotes the matrix multiplication of

the matrix $W^{(l)}$ with the vector $\alpha^{(l-1)}(x;\theta)$ of values from the previous layer. Thus $\tilde{\alpha}^{(l)}(x;\theta)$ is simply an affine transformation of the previous layer's values.

We then apply a nonlinear *activation function*

$$\sigma : \mathbb{R} \to \mathbb{R}.$$

In a slight abuse of notation, we write $\sigma(\tilde{\alpha}^{(l)}(x;\theta))$ to mean the vector in $\mathbb{R}^{n_l}$ where the ith component is given by applying σ componentwise;

$$\left(\sigma(\tilde{\alpha}^{(l)}(x;\theta))\right)_i = \sigma(\tilde{\alpha}^{(l)}(x;\theta)_i).$$

The choice of the activation function is a subtle and delicate art. Some common choices are the following:

- $\sigma = \mathrm{Id}$, the identity function, also referred to as a *linear* activation. In this case, the network becomes a *linear* function of the inputs. This is not used in practice of course, but still technically a neural network.
- $\sigma = \mathrm{ReLU}$ for *rectified linear unit*, given by

  $$\mathrm{ReLU}(z) := (z)_+,$$

  the positive part of z.
- $\sigma = \mathrm{sigmoid}$, where

  $$\mathrm{sigmoid}(z) = \frac{1}{1+\exp(-z)}.$$

- $\sigma = \tanh$, the hyperbolic tangent function.
- $\sigma = \arctan$, the inverse tangent function.
- And many, many more!

Note that these typically have an increasing S-shaped curve. The biological motivation for this fact was that if the input to a neuron i in layer l was too low, namely if $\tilde{\alpha}^{(l)}(x;\theta)_i \ll 0$, the neuron should 'turn off' and return 0.

Finally, the output $f(x;\theta) \in \mathbb{R}$ is given by $\alpha^L(x;\theta)$.

The vector of parameters, $\theta \in \mathbb{R}^m$ consists of all entries of the weight matrices $W^{(1)}, \ldots, W^{(L)}$ and all bias vectors $b^{(1)}, \ldots, b^{(L)}$. Thus the total number of parameters is

$$m = \sum_{l=1}^{L} \left(n_{l-1} n_l + n_l\right),$$

which can be incredibly large if the depth L is large or the widths n_l are large. In typical modern applications, m could be in the order of millions or billions (or more).

Note that the structure of a feedforward neural network is *compositional*. We could alternatively have written it as

$$f(x;\theta) = \sigma\left(W^{(L)}\big(\sigma(W^{(L-1)}\sigma(\cdots\sigma(W^{(1)}x + b^{1)})\dots) + b^{(L-1)}\big) + b^{(L)}\right).$$

Intuitively, this large composition of functions enables the resulting neural network to be highly *expressive* as the depth increases. This is what motivates *deep* learning: such networks with $L > 1$ have the capacity to represent (almost) any given function to an arbitrary degree of accuracy.

Other architectures

Fully connected feedforward neural networks are the basic building blocks of modern deep learning architecture. There are many, many other kinds of neural network architecture which we will only mention briefly

This concludes our basic introduction to neural networks. The next lecture will discuss how one learns the millions (or billions) of parameters $\theta \in \mathbb{R}^m$ from data.

6.1.4 Topics not covered

Since deep learning has exploded over the past decade, there is necessarily a very long list of topics we will not cover, but are also important aspects of deep learning. The interested student is encouraged to independently look into such topics.

- Reinforcement learning, meta-learning.
- Natural language processing and transformers.
- Normalizing flows.
- Domain adaptation.
- Semi-supervised learning.
- (Adversarial) robustness.
- ..

6.2 Lecture 2

6.2.1 Formulating the Optimisation Problem

In this section, we will discuss the problem of training a neural network to solve a given problem. The focus will be on statistical problems, e.g. regression ($y^i = f(x^i) + \xi^i$), classification ($P(y = 1|x) = f(x)$), etc., where f is given by a neural network in some class.

To pose our problem, we usually first specify an architecture (number of layers, widths of layers, choice of activation function) for the network, and then parametrise a function with these characteristic in terms of its weights and biases, hereafter denoted by $\theta \in \Theta$.

With this class specified, we want to find a value of $\theta_* \in \Theta$ such that the function $f_* = f(\cdot, \theta_*)$ is a good predictor on unseen data. That is, we want the risk

$$\mathrm{L}(\theta) = \mathbf{E}_{X,Y}[\ell(y, f(x, \theta))] \tag{6.1}$$

to be small, where ℓ is some loss function which measures the mismatch between our predictions and the observations. Common examples of loss functions include $\ell(y, f) = |y - f|^p$ for continuous data (with $p = 1, 2$ the most common), and $\ell(y, f) = -\sum_{k\in[K]} \mathbf{I}[y_i = k] \log f_k$ for classification with K classes.

Of course, we don't have direct access to the full data-generating process, so we cannot write down $\mathcal{L}(\theta)$, and hence it is challenging to minimise it, even in principle. In practice, we just have the training data. Under the assumption that the training data come from the same distribution as future data, we might reasonably then make the approximation

$$\mathrm{L}(\theta) \approx \mathcal{L}(\theta) = \frac{1}{N} \sum_{i\in[N]} \ell(y_i, f(x_i, \theta)). \tag{6.2}$$

Crucially, we **can** write down this loss function concretely, and it is thus a valid candidate for minimisation.

some discussion of training error, test error, risk (or pointer to earlier reference in notes)

Classically, it is common to regularise the optimisation problem somewhat, in order to promote solutions f_* which are more well-behaved, e.g. smooth, piecewise-constant, etc. The idea is that if one focuses purely on minimising $\mathcal{L}$, one could end up with a high-complexity function f which matches the training data well, but behaves badly away from it. Thus, one can consider modifications of the ERM problem such as

$$\min_{\theta\in\Theta} \mathcal{L}(\theta) + \lambda \cdot \Omega(\theta) \tag{6.3}$$

$$\text{or} \quad \min_{\theta\in\Theta} \mathcal{L}(\theta) \quad \text{such that } \Omega(\theta) \leqslant R \tag{6.4}$$

where Ω is a regularisation functional which takes large values when the function $f(\cdot, \theta)$ is irregular. Common examples of regularisation functionals for vector-valued θ include $\Omega(\theta) = \|\theta\|_2^2, \|\theta\|_1, \|\theta\|_\infty$. To some extent, this form of

regularisation carries on in modern deep learning, though it should be noted that regularisation now takes many other forms as well, some less direct than others.

contextualise necessity of regularisation for over-parametrised / unidentifiable problems

6.2.2 Challenges of ERM in Deep Learning

Relative to previous settings, solving the ERM problem for deep neural networks poses some distinct problems.

1. In classical statistics, estimators can often be computed by hand, in closed form. There is no hope for this to happen here.
2. In modern statistics, estimators can often be formulated as the solution to a convex program, which can be solved efficiently using the methods of convex optimisation. However, for typical formulations of the deep learning problem, convexity is off the table as well.

Having asserted this, some discussion is merited on the topic of how this non-convexity arises.

Well, to begin with, for generic x, the mapping $\theta \mapsto f(x, \theta)$ is nonlinear. As such, even when the loss function ℓ is convex in its usual arguments, we cannot expect the composite mapping $\theta \mapsto \ell(y, f(x, \theta))$ to be convex.

give a simple example of this

Moreover, standard parametrisations of neural networks are not identifiable, in the sense that one can locate $\theta_1 \neq \theta_2$ such that $f(x, \theta_1) = f(x, \theta_2)$ for all x. Hence, even if there is a unique optimal function f in our class of neural networks, its representation in parameters θ will typically be non-unique, again precluding convexity for our practical optimisation problem.

Now, if we believe that the problem is non-convex, we should also understand why this might be a problem.

To begin with, non-convexity puts us at risk of getting stuck at so-called 'bad local minima' of the loss function (though 'bad stationary point' might be more accurate). This corresponds to landing at a parameter θ such that $\nabla \mathcal{L}(\theta) = 0$, but with $\mathcal{L}(\theta)$ being strictly greater than the optimal value. Essentially, the concern is that the optimisation landscape is too 'weird' to guarantee that local search methods will find anything close to an optimal solution of the problem.

Another phenomenon of non-convexity which is well-recognised in over-parametrised neural networks is the existence of many interpolating *global* minima, i.e. an entire manifold $\mathcal{M} \subset \Theta$ of parameters θ such that $y_i = f(x_i, \theta)$ for all i. A byproduct of this is that not all of these interpolating solutions will generalise equally well, and so we run the risk that our optimisation routine terminates at a *bad global minimum*. Note that it may be hard to diagnose this while training a network, as it pertains to out-of-sample performance.

6.2.3 Regularisation for Neural Networks

Here, we will discuss some regularisation techniques which one can consider for neural networks. I will informally write $\theta = \{\theta_\ell\}_{\ell=1}^{L}$, where θ_ℓ correspond to the parameters in the ℓ^{th} layer.

Firstly, generic regularisation functionals (largely in the spirit of modern high-dimensional statistics):

1. ℓ^2 regularisation (Ridge-like): $\Omega(\theta) = \sum_{\ell=1}^{L} \alpha_\ell \|\theta_\ell\|_2^2$, for some nonnegative vector α.
2. ℓ^1 regularisation (LASSO-like): $\Omega(\theta) = \sum_{\ell=1}^{L} \alpha_\ell \|\theta_\ell\|_1$, for some nonnegative vector α.
3. ℓ^2 constraints: $\Omega(\theta) = \sum_{\ell=1}^{L} \chi\left(\|\theta_\ell\|_2 \leqslant R_\ell\right)$, for some nonnegative vector α.

Writing $\theta_\ell = (W_\ell, b_\ell)$ for the weights and biases respectively, one might sometimes penalise the weight matrices to be well-behaved in operator norm, e.g. $\Omega(W) = \|W\|_{\mathrm{op}}$ or $\|W - I\|_{\mathrm{op}}$.

There also exist a number of deep learning-specific regularisation strategies, which do not really have analogues in shallow learning. Some examples of this include Batch Normalisation, Group Normalisation, and Layer Normalisation, the latter of which I will outline here.

The idea behind these methods is to guarantee that the size of the inputs and outputs of each layer of the network remain relatively stable with ℓ. For example, if layer ℓ has N_ℓ neurons, then it might be desirable for them to satisfy

$$\frac{1}{N_\ell} \sum_{i \in [N_\ell]} x_\ell^i \approx 0 \tag{6.5}$$

$$\frac{1}{N_\ell} \sum_{i \in [N_\ell]} \left(x_\ell^i\right)^2 \approx 1 \tag{6.6}$$

for all layers ℓ. In principle, this could improve the stability and conditioning of the optimisation process. The impact of this regularisation on generalisation performance is less clear.

Layer Normalisation accomplishes this task as follows: given $x_\ell = \sigma\left(W_\ell x_{\ell-1} + b_\ell\right)$, write

$$\mu_\ell(x) = \frac{1}{N_\ell} \sum_{i \in [N_\ell]} x_\ell^i \tag{6.7}$$

$$\sigma_\ell^2(x) = \frac{1}{N_\ell} \sum_{i \in [N_\ell]} \left(x_\ell^i - \mu_\ell(x)\right)^2 \tag{6.8}$$

One then defines the normalised quantity $z_\ell(x)$ by

$$z_\ell^i(x) = \frac{x_\ell^i - \mu_l(x)}{\sigma_\ell(x)}, \tag{6.9}$$

which is fed into the next layer via

$$x_{\ell+1} = \sigma\left(W_{\ell+1} z_\ell + b_{\ell+1}\right). \tag{6.10}$$

Thus, instead of the usual cascade of "linear map, nonlinear activation", we have "linear map, nonlinear activation, nonlinear normalisation".

So, at this stage, we have talked about

1. How to specify a class of neural networks.
2. How to parametrise a neural network within that class.
3. How to define a loss function, via ERM and some amount of regularisation.

Given this information, one can write down a training objective, and run an optimisation routine to minimise this objective (the specifics of which will be covered in the next lecture).

6.3 Lecture 3

Optimization.

Backpropagation.

1. Emphasise that the task is empirical risk minimisation. In this context, that means non-convexity, large parameter spaces, and often large data sets. As a result, it is hard to obtain theoretical guarantees, and many standard methods (Newton, Batch GD) need not apply. The practical focus is on gradient-based methods with data subsampling.

2. Stochastic Gradient Method, Robbins-Monro, Unbiased Estimator. Not a descent method.

3. Variants: minibatching, importance sampling, random reshuffling, variance reduction.

4. Step sizes in theory and practice, Polyak-Ruppert Averaging, Tail Averaging.

5. Proof of one-step descent in expectation for sufficiently small step-sizes, Lipschitz (?) objectives.

6. Adagrad-Norm, Full Adagrad. Prove that effective step size will stabilise.

7. Variants: Adadelta, RMSProp, Adam, Momentum SGD.

8. Some overview of what theory is available, and under which conditions.

6.4 Deep Generative Models

The opening statistical problems we looked at in this course were regression and classification. In either case, we relied on having *labelled data*:

- For regression, we assumed we had access to pairs $(\mathbf{x}_i, y_i)_{i=1}^N$, where each $\mathbf{x}_i \in \mathbb{R}^p$ was some vector of covariates, and each $y_i \in \mathbb{R}$ was some real-valued response.
- For classification, we also assumed access to pairs $(\mathbf{x}_i, y_i)_{i=1}^n$, where again each $\mathbf{x}_i \in \mathbb{R}^p$ was a vector of covariates, but where now each $y_i \in \mathcal{G}$, where $\mathcal{G}$ was some discrete set of labels.

In machine learning, this setting is referred to as *supervised learning*, because some 'supervisor' has appropriately labelled each covariate $\mathbf{x}_i$. In this section, we will consider a different setting, where we are simply presented with a set of training data of the form $\mathcal{D} = (\mathbf{x}_i)_{i=1}^N$, with no accompanying labels. For example, we could be presented with a large collection of images, say, of people, with no further information.

How can we make sense of such an unlabelled data set? This is the problem of *unsupervised learning*.

One approach to unsupervised learning, which we will focus on during the next two lectures, is *generative modelling*. The idea of generative modelling is as follows. Suppose that each observed data point $\mathbf{x}_i \overset{\text{i.i.d.}}{\sim} p_{\text{data}}(\mathbf{x})$. In other words, each data point $\mathbf{x}_i$ is drawn i.i.d. from some unknown underlying probability distribution $p_{\text{data}}(\mathbf{x})$. Then, if we were able to create new samples, which appear to have been drawn from $p_{\text{data}}(\mathbf{x})$, then this would be an indication that we have understood our data set. In other words, if we are able to produce synthetic data points $\tilde{\mathbf{x}}_j$, which look like they are also drawn from $p_{\text{data}}(\mathbf{x})$, e.g., they are indistinguishable to a human observer from our actual samples $\mathbf{x}_i$, then we would say we have solved the generative modelling problem.

We will be interested in *parametric* approximations to the data distribution, which summarise all of the information about the dataset at hand in a finite set of parameters. In comparison to *non-parametric* models, parametric models scale much more efficiently with very large datasets, but are limited in the class of distributions that they can represent.

The way we will approach this task is to try to learn a model p_θ, parameterised by some parameter $\theta \in \Theta$, which is in some sense close to p_{data}. Mathematically speaking, we can specify our goal as trying to find the solution of the following optimisation problem

$$\min_{\theta \in \Theta} d(p_{\text{data}}, p_\theta), \tag{6.11}$$

where $d(\cdot)$ is some notion of distance between probability distributions. Once we have learned a generative model, we can use it for a huge range of downstream tasks. Among these, there are two which we will highlight as particularly important:

- *Density Estimation.* What is the probability $p_\theta(\mathbf{x})$ assigned to a given data point $\mathbf{x}$?
- *Sampling.* How can we generate new data $\mathbf{x}_{\text{new}} \sim p_\theta(\mathbf{x})$ from the generative model?

6.4.1 Variational Auto-Encoders

One popular approach to generative modelling is to use a Variational Auto-Encoder (VAE), first introduced by Kingma and Welling [2014]. For some popular subsequent extensions, see Higgins et al. [2017], van den Oord et al. [2017], Razavi et al. [2019].

Overview

Let's imagine, as described above, that we have some unlabelled i.i.d. training data $\mathcal{D} = (\mathbf{x}_i)_{i=1}^N$, with each $\mathbf{x}_i \in \mathbb{R}^d$, where d could be quite large. We now propose a *latent variable model* that generated this data. Formally, a latent variable model is a distribution over two sets of variables $\mathbf{x}$ and $\mathbf{z}$:

$$p_\theta(\mathbf{x}, \mathbf{z}) \tag{6.12}$$

where $\mathbf{x} \in \mathbb{R}^d$ are the observed variables, and $\mathbf{z} \in \mathbb{R}^r$ are the latent (or hidden) variables. For example, $\mathbf{x}$ could be images of people, while $\mathbf{z}$ could be high level of features such as eye colour, hair colour, etc. Typically, the dimension r of the latent space will be much smaller than the dimension d of the data space. This is related to the distribution of interest via

$$p_\theta(\mathbf{x}) = \int p_\theta(\mathbf{x}, \mathbf{z}) \mathrm{d}\mathbf{z}. \tag{6.13}$$

Using Bayes' rule, we can rewrite the joint distribution above in the following way

$$p_\theta(\mathbf{x}, \mathbf{z}) = p_\theta(\mathbf{x}|\mathbf{z})p(\mathbf{z}). \tag{6.14}$$

From a generative modeling perspective, this model describes a generative process for data $\mathbf{x}$ according to the following procedure. First generate a latent variable $\mathbf{z} \sim p(\mathbf{z})$. We can think of this as a latent *code* which generates an output, and sometimes refer to $p(\mathbf{z})$ as the *code* distribution. Given this $\mathbf{z}$, we then generate $\mathbf{x} \sim p_\theta(\mathbf{x}|\mathbf{z})$. We will refer to the conditional distribution $p_\theta(\mathbf{x}|\mathbf{z})$ as the probabilistic *decoder*, since it maps our latent codes to points in the data space.

We will suppose that $p(\mathbf{z}) \in \mathcal{P}_{\mathbf{z}}$, where $\mathcal{P}_{\mathbf{z}}$ denotes some family of distributions defined on the latent space. Similarly, we will suppose that $p_\theta(\mathbf{x}|\mathbf{z}) \in \mathcal{P}_{\mathbf{x}|\mathbf{z}}$. Our hypothetical class of generative models is then given by

$$\mathcal{P}_{\mathbf{x},\mathbf{z}} = \left\{ p(\mathbf{x}, \mathbf{z}) | p(\mathbf{z}) \in \mathcal{P}_{\mathbf{z}} \ , \ p_\theta(\mathbf{x}|\mathbf{z}) \in \mathcal{P}_{\mathbf{x}|\mathbf{z}} \right\}. \tag{6.15}$$

Given the dataset $\mathcal{D} = (\mathbf{x}_i)_{i=1}^N$, we are then interesting the following tasks:

- What is the distribution $p_\theta(\mathbf{x}, \mathbf{z}) \in \mathcal{P}_{\mathbf{x},\mathbf{z}}$ which best fits the observed data $\mathcal{D}$?
- Given a sample $\mathbf{x}$ and a model $p_\theta(\mathbf{x}, \mathbf{z})$, what is the posterior distribution over the latent variables $p_\theta(\mathbf{z}|\mathbf{x})$?

We can measure how closely $p_\theta(\mathbf{x}, \mathbf{z})$ fits the observed dataset $\mathcal{D} = (\mathbf{x}_i)_{i=1}^N$ using the Kullback-Leibler (KL) divergence between $p_{\text{data}}(\mathbf{x})$ and the marginal distribution of the model, $p_\theta(\mathbf{x})$. In particular, we would like to find the parameters $\theta \in \Theta$ which minimises

$$\begin{aligned} \mathrm{D}_{\mathrm{KL}}(p_{\text{data}}(\mathbf{x})|p_\theta(\mathbf{x})) &= \mathbb{E}_{p_{\text{data}}(\mathbf{x})}\left[\log \frac{p_{\text{data}}(\mathbf{x})}{p_\theta(\mathbf{x})}\right] && (6.16)\\ &= \mathbb{E}_{p_{\text{data}}(\mathbf{x})}\left[\log p_{\text{data}}(\mathbf{x})\right] - \mathbb{E}_{p_{\text{data}}(\mathbf{x})}\left[\log p_\theta(\mathbf{x})\right] && (6.17)\\ &\approx \frac{1}{N}\sum_{i=1}^N \log p_{\text{data}}(\mathbf{x}_i) - \frac{1}{N}\sum_{i=1}^N \log p_\theta(\mathbf{x}_i) && (6.18) \end{aligned}$$

Since the first term is independent of the parameter, minimising this quantity over θ is precisely equivalent to maximising the *marginal log-likelihood* of the observed data

$$\mathcal{L}(\theta; (\mathbf{x}_i)_{i=1}^N) = \log \prod_{i=1}^N p_\theta(\mathbf{x}_i) = \sum_{i=1}^N \log p_\theta(\mathbf{x}_i).$$

In general, evaluating this function, let alone maximising it, is very difficult. In particular, evaluating

$$p_\theta(\mathbf{x}) = \int p_\theta(\mathbf{x}, \mathbf{z}) \mathrm{d}\mathbf{z} \tag{6.19}$$

is intractable for high-dimensional $\mathbf{z}$ as it involves an integral (or sum in the case that $\mathbf{z}$ only takes discrete values) over the latent variables $\mathbf{z}$.

One idea would be to approximate this objective using a *Monte Carlo* estimate. In particular, for any given data point $\mathbf{x}$, we can estimate its marginal log-likelihood as

$$\log p(\mathbf{x}) = \log \int p_\theta(\mathbf{x}, \mathbf{z}) \mathrm{d}\mathbf{z} = \log \int p_\theta(\mathbf{x}|\mathbf{z}) p(\mathbf{z}) \mathrm{d}\mathbf{z} \approx \frac{1}{k}\sum_{j=1}^k p_\theta(\mathbf{x}|\mathbf{z}_j) \tag{6.20}$$

where $\mathbf{z}_j \overset{\text{i.i.d.}}{\sim} p(\mathbf{z})$. In practice, however, optimising this approximation leads to gradient estimates with very high variance.

The Evidence Lower Bound

Rather than maximising the log-likelihood directly, an alternative approach is is to construct a lower bound which is more amenable to optimisation. We will then *maximise this lower bound* directly. The hope is that the set of parameters which optimise the lower bound will still lead to an effective generative model.

We begin by using Bayes' rule. In particular, for any $\mathbf{z} \in \mathcal{Z}$, we have that $p(\mathbf{z})p_\theta(\mathbf{x}|\mathbf{z}) = p_\theta(\mathbf{x})p_\theta(\mathbf{z}|\mathbf{x})$. Taking logarithms, it follows that

$$\log p_\theta(\mathbf{x}) = \log p(\mathbf{z}) + \log p_\theta(\mathbf{x}|\mathbf{z}) - \log p_\theta(\mathbf{z}|\mathbf{x}). \tag{6.21}$$

In order to proceed, we now introduce an auxiliary distribution $q_\lambda(\mathbf{z}|\mathbf{x}) \in \mathcal{Q}_{\mathbf{z}|\mathbf{x}}$, where $\mathcal{Q}_{\mathbf{z}|\mathbf{x}}$ is a *variational family* of distributions which approximate the true (but intractable) posterior $p(\mathbf{z}|\mathbf{x})$. This family of distributions will be parameterised by $\lambda \in \Lambda$. We will refer to the distribution $q_\lambda(\mathbf{z}|\mathbf{x})$ as the probabilistic *encoder* since, given an observed data point $\mathbf{x}$, it gives us a distribution over latent codes $\mathbf{z}$.

We can now go back to (6.21). Since this equation is valid for any $\mathbf{z} \in \mathcal{Z}$, we can integrate both sides over $\mathbf{z}$ with respect to the distribution $q_\lambda(\mathbf{z}|\mathbf{x})$ to obtain

$$\begin{aligned}
\log p_\theta(\mathbf{x}) &= \mathbb{E}_{\mathbf{z}\sim q_\lambda(\mathbf{z}|\mathbf{x})}[\log p(\mathbf{z})p_\theta(\mathbf{x}|\mathbf{z})] - \mathbb{E}_{\mathbf{z}\sim q_\lambda(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{z}|\mathbf{x})] \\
&= \mathbb{E}_{\mathbf{z}\sim q_\lambda(\mathbf{z}|\mathbf{x})}[\log p(\mathbf{z})p_\theta(\mathbf{x}|\mathbf{z})] - \mathbb{E}_{\mathbf{z}\sim q_\lambda(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{z}|\mathbf{x})] \\
&\quad + \mathbb{E}_{\mathbf{z}\sim q_\lambda(\mathbf{z}|\mathbf{x})}[\log q_\lambda(\mathbf{z}|\mathbf{x})] - \mathbb{E}_{\mathbf{z}\sim q_\lambda(\mathbf{z}|\mathbf{x})}[\log q_\lambda(\mathbf{z}|\mathbf{x})] \\
&= \mathbb{E}_{\mathbf{z}\sim q_\lambda(\mathbf{z}|\mathbf{x})}\left[\log \frac{p(\mathbf{z})p_\theta(\mathbf{x}|\mathbf{z})}{q_\lambda(\mathbf{z}|\mathbf{x})}\right] + \mathbb{E}_{\mathbf{z}\sim q_\lambda(\mathbf{z}|\mathbf{x})}\left[\log \frac{q_\lambda(\mathbf{z}|\mathbf{x})}{p_\theta(\mathbf{z}|\mathbf{x})}\right]
\end{aligned}$$

where in the second equality we have added and subtracted $\mathbb{E}_{\mathbf{z}\sim q_\lambda(\mathbf{z}|\mathbf{x})}[\log q_\lambda(\mathbf{z}|\mathbf{x})]$. Using the definition of the Kullback-Leibler divergence, we can rewrite this as

$$\log p_\theta(\mathbf{x}) = \mathbb{E}_{\mathbf{z}\sim q_\lambda(\mathbf{z}|\mathbf{x})}\left[\log \frac{p(\mathbf{z})p_\theta(\mathbf{x}|\mathbf{z})}{q_\lambda(\mathbf{z}|\mathbf{x})}\right] + D_{\mathrm{KL}}(q_\lambda(\mathbf{z}|\mathbf{x})|p_\theta(\mathbf{z}|\mathbf{x})).$$

Finally, using the fact that the second quantity on the RHS is always non-negative, we arrive at the *Evidence Lower Bound* (ELBO):

Proposition 12 (ELBO, Kingma and Welling [2014])**.** *We have the lower bound,*

$$\log p_\theta(\mathbf{x}) \geq \underbrace{\mathbb{E}_{\mathbf{z}\sim q_\lambda(\mathbf{z}|\mathbf{x})}\left[\log \frac{p(\mathbf{z})p_\theta(\mathbf{x}|\mathbf{z})}{q_\lambda(\mathbf{z}|\mathbf{x})}\right]}_{\mathrm{ELBO}}. \tag{6.22}$$

Proof. This follows directly from the previous expression and the non-negativity of the KL divergence (Appendix B). □

Learning VAEs

Instead of trying to maximising log-likelihood directly over the parameters $\theta \in \Theta$, we will instead try to maximise the ELBO, also referred to as the *variational lower bound* or the *variational free energy*. In particular, we want to solve

$$\max_{\theta\in\Theta} \sum_{i=1}^{N} \mathbb{E}_{\mathbf{z}\sim q_\lambda(\mathbf{z}|\mathbf{x}_i)}\left[\log \frac{p(\mathbf{z})p_\theta(\mathbf{x}_i|\mathbf{z})}{q_\lambda(\mathbf{z}|\mathbf{x}_i)}\right]. \tag{6.23}$$

It remains to work out how we should choose the variational approximation $q_\lambda(\mathbf{z}|\mathbf{x})$. First, we definitely need to be be able to evaluate and sample from $q_\lambda(\mathbf{z}|\mathbf{x})$, so that we can approximate the summands in (6.23) using a Monte Carlo estimate:

$$\mathbb{E}_{\mathbf{z}\sim q_\lambda}\left[\log \frac{p(\mathbf{z})p_\theta(\mathbf{x}|\mathbf{z})}{q_\lambda(\mathbf{z}|\mathbf{x})}\right] \approx \frac{1}{k}\sum_{j=1}^{k} \log \frac{p(\mathbf{z}_j)p_\theta(\mathbf{x}|\mathbf{z}_j)}{q_\lambda(\mathbf{z}_j|\mathbf{x})}, \tag{6.24}$$

where $\mathbf{z}_j \overset{\text{i.i.d.}}{\sim} q_\lambda(\mathbf{z}|\mathbf{x})$. But how do we choose the variational parameters λ? Even though the derivation of the ELBO is valid for any choice of λ, the tightness of the lower bound in Proposition 12 depends on the properties of $q_\lambda(\mathbf{z}|\mathbf{x})$. In particular, from (6.4.1), the gap between the marginal log-likelihood $p_\theta(\mathbf{x})$ and the variational lower bound is given by

$$\log p_\theta(\mathbf{x}) - \underbrace{\mathbb{E}_{\mathbf{z}\sim q_\lambda(\mathbf{z}|\mathbf{x})}\left[\log \frac{p(\mathbf{z})p_\theta(\mathbf{x}|\mathbf{z})}{q_\lambda(\mathbf{z}|\mathbf{x})}\right]}_{\text{ELBO}} = D_{\mathrm{KL}}(q_\lambda(\mathbf{z}|\mathbf{x})|p_\theta(\mathbf{z}|\mathbf{x})) \tag{6.25}$$

Thus, the gap is smaller the closer $q_\lambda(\mathbf{z}|\mathbf{x})$ is to the true posterior $p_\theta(\mathbf{z}|\mathbf{x})$. We would like to make this gap as small as possible. In other words, we would like to solve

$$\min_{\lambda\in\Lambda}\left[\log p_\theta(\mathbf{x}) - \mathbb{E}_{\mathbf{z}\sim q_\lambda(\mathbf{z}|\mathbf{x})}\left[\log \frac{p(\mathbf{z})p_\theta(\mathbf{x}|\mathbf{z})}{q_\lambda(\mathbf{z}|\mathbf{x})}\right]\right] := \max_{\lambda\in\Lambda}\mathbb{E}_{\mathbf{z}\sim q_\lambda(\mathbf{z}|\mathbf{x})}\left[\log \frac{p(\mathbf{z})p_\theta(\mathbf{x}|\mathbf{z})}{q_\lambda(\mathbf{z}|\mathbf{x})}\right] \tag{6.26}$$

where to go from the LHS to the RHS we have used the fact that the first term is independent of the λ.

Combining (6.23) and (6.26), we will train our model by jointly optimising over the model parameters and the variational parameters. Thus, in particular, we will aim to solve

$$\max_{\theta\in\Theta}\max_{\lambda\in\Lambda}\sum_{i=1}^{N}\mathbb{E}_{\mathbf{z}\sim q_\lambda(\mathbf{z}|\mathbf{x}_i)}\left[\log \frac{p(\mathbf{z})p_\theta(\mathbf{x}_i|\mathbf{z})}{q_\lambda(\mathbf{z}|\mathbf{x}_i)}\right]. \tag{6.27}$$

This optimisation problem is typically solved iteratively using *stochastic gradient descent*.

Algorithm 14: Learning VAEs using (Stochastic) Gradient Descent

Input: initial parameters $\theta \in \Theta$, $\lambda \in \Lambda$; learning rates $\gamma \in \mathbb{R}_+$, $\eta \in \mathbb{R}_+$.
While not converged:

- Update the variational parameters.

$$\lambda_i \leftarrow \lambda + \eta \tilde{\nabla}_\lambda \sum_{i=1}^{N} \mathbb{E}_{\mathbf{z} \sim q_\lambda(\mathbf{z}|\mathbf{x}_i)} \left[\log \frac{p(\mathbf{z}) p_\theta(\mathbf{x}_i|\mathbf{z})}{q_\lambda(\mathbf{z}|\mathbf{x}_i)} \right]. \tag{6.28}$$

  where $\tilde{\nabla}_{\lambda_i}$ denotes an unbiased estimate of the graident w.r.t λ.

- Update the model parameters.

$$\theta \leftarrow \theta + \gamma \tilde{\nabla}_\theta \sum_{i=1}^{N} \mathbb{E}_{\mathbf{z} \sim q_\lambda(\mathbf{z}|\mathbf{x}_i)} \left[\log \frac{p(\mathbf{z}) p_\theta(\mathbf{x}_i|\mathbf{z})}{q_{\lambda_i}(\mathbf{z}|\mathbf{x}_i)} \right] \tag{6.29}$$

  where $\tilde{\nabla}_\theta$ denotes an unbiased estimate of the gradient w.r.t θ.

Gradient Estimation

In general, it isn't possible to compute the gradients in (6.28) and (6.29) in closed form. Instead, we will approximate them using Monte Carlo estimates. Let's start with the gradient w.r.t θ. In this case, we have

$$\nabla_\theta \mathbb{E}_{\mathbf{z} \sim q_\lambda(\mathbf{z}|\mathbf{x})} \left[\log \frac{p(\mathbf{z}) p_\theta(\mathbf{x}|\mathbf{z})}{q_\lambda(\mathbf{z}|\mathbf{x})} \right] = \mathbb{E}_{\mathbf{z} \sim q_\lambda(\mathbf{z}|\mathbf{x})} \left[\nabla_\theta \log \frac{p(\mathbf{z}) p_\theta(\mathbf{x}|\mathbf{z})}{q_\lambda(\mathbf{z}|\mathbf{x})} \right] \tag{6.30}$$

$$= \mathbb{E}_{\mathbf{z} \sim q_\lambda(\mathbf{z}|\mathbf{x})} \left[\nabla_\theta \log p_\theta(\mathbf{x}|\mathbf{z}) \right] \tag{6.31}$$

Thus, given samples $\mathbf{z}_j \overset{\text{i.i.d.}}{\sim} q_\lambda(\mathbf{z}|\mathbf{x})$, we can approximate

$$\nabla_\theta \mathbb{E}_{\mathbf{z} \sim q_\lambda(\mathbf{z}|\mathbf{x})} \left[\log \frac{p(\mathbf{z}) p_\theta(\mathbf{x}|\mathbf{z})}{q_\lambda(\mathbf{z}|\mathbf{x})} \right] \approx \frac{1}{k} \sum_{j=1}^{k} \nabla_\theta \log p_\theta(\mathbf{x}|\mathbf{z}_j). \tag{6.32}$$

The gradient w.r.t. λ is a little more complicated, since the expectation depends on λ. In this case, we have

$$\nabla_\lambda \mathbb{E}_{\mathbf{z} \sim q_\lambda(\mathbf{z}|\mathbf{x})} \left[\log \frac{p(\mathbf{z}) p_\theta(\mathbf{x}|\mathbf{z})}{q_\lambda(\mathbf{z}|\mathbf{x})} \right] = \nabla_\lambda \int \log \frac{p(\mathbf{z}) p_\theta(\mathbf{x}|\mathbf{z})}{q_\lambda(\mathbf{z}|\mathbf{x})} q_\lambda(\mathbf{z}|\mathbf{x}) \mathrm{d}\mathbf{z} \tag{6.33}$$

$$= \int \nabla_\lambda \left[\log \frac{p(\mathbf{z}) p_\theta(\mathbf{x}|\mathbf{z})}{q_\lambda(\mathbf{z}|\mathbf{x})} q_\lambda(\mathbf{z}|\mathbf{x}) \right] \mathrm{d}\mathbf{z}. \tag{6.34}$$

Let's work out the derivative inside the integral. Using the product rule, we have

$$\nabla_\lambda \left[\log \frac{p(\mathbf{z}) p_\theta(\mathbf{x}|\mathbf{z})}{q_\lambda(\mathbf{z}|\mathbf{x})} q_\lambda(\mathbf{z}|\mathbf{x}) \right] = \nabla_\lambda \left[\log \frac{p(\mathbf{z}) p_\theta(\mathbf{x}|\mathbf{z})}{q_\lambda(\mathbf{z}|\mathbf{x})} \right] q_\lambda(\mathbf{z}|\mathbf{x}) + \log \frac{p(\mathbf{z}) p_\theta(\mathbf{x}|\mathbf{z})}{q_\lambda(\mathbf{z}|\mathbf{x})} \nabla_\lambda q_\lambda(\mathbf{z}|\mathbf{x}). \tag{6.35}$$

The first term can be simplified as follows:

$$\nabla_\lambda \left[\log \frac{p(\mathbf{z})p_\theta(\mathbf{x}|\mathbf{z})}{q_\lambda(\mathbf{z}|\mathbf{x})}\right] q_\lambda(\mathbf{z}|\mathbf{x}) = -\nabla_\lambda \left[\log q_\lambda(\mathbf{z}|\mathbf{x})\right] q_\lambda(\mathbf{z}|\mathbf{x}) \tag{6.36}$$

$$= -\frac{\nabla_\lambda q_\lambda(\mathbf{z}|\mathbf{x})}{q_\lambda(\mathbf{z}|\mathbf{x})} q_\lambda(\mathbf{z}|\mathbf{x}) = -\nabla_\lambda q_\lambda(\mathbf{z}|\mathbf{x}). \tag{6.37}$$

Thus, integrating w.r.t $\mathbf{z}$, we have that

$$\int \nabla_\lambda \left[\log \frac{p(\mathbf{z})p_\theta(\mathbf{x}|\mathbf{z})}{q_\lambda(\mathbf{z}|\mathbf{x})}\right] q_\lambda(\mathbf{z}|\mathbf{x})\mathrm{d}\mathbf{z} = -\int \nabla_\lambda q_\lambda(\mathbf{z}|\mathbf{x})\mathrm{d}\mathbf{z} = -\nabla_\lambda \underbrace{\int q_\lambda(\mathbf{z}|\mathbf{x})\mathrm{d}\mathbf{z}}_{1} = 0. \tag{6.38}$$

Meanwhile, the second term can be written as

$$\log \frac{p(\mathbf{z})p_\theta(\mathbf{x}|\mathbf{z})}{q_\lambda(\mathbf{z}|\mathbf{x})} \nabla_\lambda q_\lambda(\mathbf{z}|\mathbf{x}) = \log \frac{p(\mathbf{z})p_\theta(\mathbf{x}|\mathbf{z})}{q_\lambda(\mathbf{z}|\mathbf{x})} \frac{\nabla_\lambda q_\lambda(\mathbf{z}|\mathbf{x})}{q_\lambda(\mathbf{z}|\mathbf{x})} q_\lambda(\mathbf{z}|\mathbf{x}) \tag{6.39}$$

$$= \log \frac{p(\mathbf{z})p_\theta(\mathbf{x}|\mathbf{z})}{q_\lambda(\mathbf{z}|\mathbf{x})} \nabla_\lambda \log q_\lambda(\mathbf{z}|\mathbf{x}) \; q_\lambda(\mathbf{z}|\mathbf{x}). \tag{6.40}$$

Therefore, again integrating w.r.t. $\mathbf{z}$, we obtain

$$\int \log \frac{p(\mathbf{z})p_\theta(\mathbf{x}|\mathbf{z})}{q_\lambda(\mathbf{z}|\mathbf{x})} \nabla_\lambda q_\lambda(\mathbf{z}|\mathbf{x})\mathrm{d}\mathbf{z} = \int \log \frac{p(\mathbf{z})p_\theta(\mathbf{x}|\mathbf{z})}{q_\lambda(\mathbf{z}|\mathbf{x})} \nabla_\lambda \log q_\lambda(\mathbf{z}|\mathbf{x}) \; q_\lambda(\mathbf{z}|\mathbf{x})\mathrm{d}\mathbf{z} \tag{6.41}$$

$$= \mathbb{E}_{\mathbf{z}\sim q_\lambda(\mathbf{z}|\mathbf{x})}\left[\frac{p(\mathbf{z})p_\theta(\mathbf{x}|\mathbf{z})}{q_\lambda(\mathbf{z}|\mathbf{x})} \nabla_\lambda \log q_\lambda(\mathbf{z}|\mathbf{x})\right] \tag{6.42}$$

Finally, combining (6.38) and (6.42), we finally arrive at

$$\nabla_\lambda \mathbb{E}_{\mathbf{z}\sim q_\lambda(\mathbf{z}|\mathbf{x})}\left[\log \frac{p(\mathbf{z})p_\theta(\mathbf{x}|\mathbf{z})}{q_\lambda(\mathbf{z}|\mathbf{x})}\right] = \mathbb{E}_{\mathbf{z}_\lambda(\mathbf{z}|\mathbf{x})}\left[\frac{p(\mathbf{z})p_\theta(\mathbf{x}|\mathbf{z})}{q_\lambda(\mathbf{z}|\mathbf{x})} \nabla_\lambda \log q_\lambda(\mathbf{z}|\mathbf{x})\right]. \tag{6.43}$$

Thus, given samples $\mathbf{z}_j \overset{\text{i.i.d.}}{\sim} q_\lambda(\mathbf{z}|\mathbf{x})$, we can approximate

$$\nabla_\lambda \mathbb{E}_{\mathbf{z}\sim q_\lambda(\mathbf{z}|\mathbf{x})}\left[\log \frac{p(\mathbf{z})p_\theta(\mathbf{x}|\mathbf{z})}{q_\lambda(\mathbf{z}|\mathbf{x})}\right] \approx \frac{1}{k}\sum_{j=1}^{k} \frac{p(\mathbf{z}_j)p_\theta(\mathbf{x}|\mathbf{z}_j)}{q_\lambda(\mathbf{z}_j|\mathbf{x})} \nabla_\lambda \log q_\lambda(\mathbf{z}_j|\mathbf{x}) \tag{6.44}$$

In practice, this estimator tends to suffer from very high variance, and so isn't often used in practice. One way around this is something known as the *reparameterisation trick*, introduced in Kingma and Welling [2014].

The key idea here is to introduce a fixed, auxiliary distribution $p(\varepsilon)$ and a differentiable function $T_{\varepsilon,\lambda}(\mathbf{x})$ such that sampling $\varepsilon \sim p(\varepsilon)$, and then defining

$\mathbf{z} = T_{\varepsilon,\lambda}(\mathbf{x})$, is equivalent to sampling $\mathbf{z} \sim q_\lambda(\mathbf{z}|\mathbf{x})$. In this case, we simply have

$$\nabla_\lambda \mathbb{E}_{\mathbf{z}\sim q_\lambda(\mathbf{z}|\mathbf{x})}\left[\log \frac{p(\mathbf{z})p_\theta(\mathbf{x}|\mathbf{z})}{q_\lambda(\mathbf{z}|\mathbf{x})}\right] = \nabla_\lambda \mathbb{E}_{\varepsilon\sim p(\varepsilon)}\left[\log \frac{p(T_{\varepsilon,\lambda}(\mathbf{x}))p_\theta(\mathbf{x}|T_{\varepsilon,\lambda}(\mathbf{x}))}{q_\lambda(T_{\varepsilon,\lambda}(\mathbf{x})|\mathbf{x})}\right] \tag{6.45}$$

$$= \mathbb{E}_{\varepsilon\sim p(\varepsilon)}\left[\nabla_\lambda \log \frac{p(T_{\varepsilon,\lambda}(\mathbf{x}))p_\theta(\mathbf{x}|T_{\varepsilon,\lambda}(\mathbf{x}))}{q_\lambda(T_{\varepsilon,\lambda}(\mathbf{x})|\mathbf{x})}\right] \tag{6.46}$$

Thus, given samples $\varepsilon_j \sim p(\varepsilon)$, we can instead approximate

$$\nabla_\lambda \mathbb{E}_{\mathbf{z}\sim q_\lambda(\mathbf{z}|\mathbf{x})}\left[\log \frac{p(\mathbf{z})p_\theta(\mathbf{x}|\mathbf{z})}{q_\lambda(\mathbf{z}|\mathbf{x})}\right] \approx \frac{1}{k}\sum_{j=1}^{k} \nabla_\lambda \log \frac{p(T_{\varepsilon_j,\lambda}(\mathbf{x}))p_\theta(\mathbf{x}|T_{\varepsilon_j,\lambda}(\mathbf{x}))}{q_\lambda(T_{\varepsilon_j,\lambda}(\mathbf{x})|\mathbf{x})}. \tag{6.47}$$

This typically has a much lower variance that our estimate from before.

Parameterisations

Thus far, we have talked about $p(\mathbf{z})$, $p_\theta(\mathbf{x}|\mathbf{z})$, and $q_\lambda(\mathbf{z}|\mathbf{x})$ in the abstract. In this section, we will talk about some popular choices for these distributions.

We begin with the code distribution $p(\mathbf{z})$. A standard choice for $p(\mathbf{z})$ is the unit Gaussian:

$$p(\mathbf{z}) = \mathcal{N}(\mathbf{z}|0, \mathbf{I}) \tag{6.48}$$

where $\mathbf{I} \in \mathbb{R}^{r\times r}$ is the $r \times r$ identity matrix. Another common alternative is a mixture of Gaussians with trainable mean and covariance parameters. In this case, one would replace $p(\mathbf{z})$ by $p_\theta(\mathbf{z})$ by $p(\mathbf{z})$ in all of the previous calculations.

Next we consider the probabilistic decoder $p_\theta(\mathbf{x}|\mathbf{z})$. In this case, a popular choice is

$$p_\theta(\mathbf{x}|\mathbf{z}) = \mathcal{N}(\mathbf{x}|\mu_\theta(\mathbf{z}), \Sigma_\theta(\mathbf{z})) \tag{6.49}$$

where $\mu_\theta : \mathbb{R}^r \to \mathbb{R}^d$ and $\Sigma_\theta : \mathbb{R}^r \to \mathbb{R}^{d\times d}$ are neural networks which specify the mean and the covariance of a Gaussian distribution over $\mathbf{x}$, when conditioned on $\mathbf{z}$.

Finally, we consider the variational approximation $q_\lambda(\mathbf{z}|\mathbf{x})$. We would like to choose this distribution such that the reparametrisation trick (see above) is possible. In practice, a standard choice is once again the Gaussian distribution. In particular,

$$q_\lambda(\mathbf{z}|\mathbf{x}) = \mathcal{N}(\mathbf{z}|\mu_\lambda(\mathbf{x}), \sigma^2_\lambda(\mathbf{x})\mathbf{I}), \tag{6.50}$$

where, similar to above, $\mu_\lambda : \mathbb{R}^d \to \mathbb{R}^r$ and $\sigma^2_\lambda : \mathbb{R}^d \to \mathbb{R}^r$ are neural networks. In this case, the reparametrisation trick is possible with $p(\varepsilon) = \mathcal{N}(\varepsilon|0, \mathbf{I})$ and $T_{\varepsilon,\lambda}(\mathbf{x}) = \mu_\lambda(\mathbf{x}) + \sigma^\top_\lambda(\mathbf{x})\mathbf{I}\varepsilon$. Indeed, using standard rules for transforming Gaussians, if $\boldsymbol{\varepsilon} \sim \mathcal{N}(0, \mathbf{I})$, then $\mathbf{z} = T_{\varepsilon,\lambda}(\mathbf{x}) = \mu_\lambda(\mathbf{x}) + \sigma^\top_\lambda(\mathbf{x})\mathbf{I}\boldsymbol{\varepsilon} \sim \mathcal{N}(\mathbf{z}|\mu_\lambda(\mathbf{x}), \sigma^2_\lambda(\mathbf{x})\mathbf{I}) = q_\lambda(\mathbf{z}|\mathbf{x})$.

Conditional VAEs

In conditional generative models, the goal is to generate samples (e.g., images) conditioned on some additional information. The additional information could be discrete (e.g., a class label) or continuous (e.g., the viewing angle of an image).

Mathematically, we can formulate this problem as follows. Suppose we have access to samples $(\mathbf{x}_i, \mathbf{y}_i) \overset{\text{i.i.d.}}{\sim} p_{\text{data}}(\mathbf{x}, \mathbf{y})$, where $\mathbf{x}$ denotes the data samples and $\mathbf{y}$ denotes the additional information. Our objective is to learn a generative model $p_\theta(\mathbf{x}|\mathbf{y})$ which approximates the data distribution $p_{\text{data}}(\mathbf{x}|\mathbf{y})$.

We can extend the approach described in the previous section to this problem as follow. We begin, similar to before, by introducing the latent variable model

$$p_\theta(\mathbf{x}, \mathbf{z}|\mathbf{y}) \tag{6.51}$$

where once again $\mathbf{z} \in \mathbb{R}^r$ denote a set of latent variables. Using Bayes' rule, we can write this as

$$p_\theta(\mathbf{x}|\mathbf{z}, \mathbf{y})p(\mathbf{z}|\mathbf{y}). \tag{6.52}$$

Typically, we will assume that $\mathbf{z}$ is independent of $\mathbf{y}$, and thus this further simplifies as

$$p_\theta(\mathbf{x}|\mathbf{z}, \mathbf{y})p(\mathbf{z}). \tag{6.53}$$

Like before, this conditional generative model describes a procedure for generating $\mathbf{x} \sim p_\theta(\mathbf{x}|\mathbf{y})$. In particular, we first generate $\mathbf{z} \sim p(\mathbf{z})$. Then, given $\mathbf{y} \sim p(\mathbf{y})$, or just some fixed $\mathbf{y}$ of interest, we then generate $\mathbf{x} \sim p_\theta(\mathbf{x}|\mathbf{y})$.

Similar to before, we would like to maximise the log-likelihood of the observed data, namely,

$$\max_{\theta \in \Theta} \sum_{i=1}^{N} \log p_\theta(\mathbf{x}_i|\mathbf{y}_i). \tag{6.54}$$

Given the challenges associated with doing this directly, we will instead introduce an auxiliary distribution $q_\lambda(\mathbf{z}|\mathbf{x}, \mathbf{y})$, and maximise the variational lower bound over both the model parameters $\theta \in \Theta$ and the variational parameters $\lambda \in \Lambda$, viz

$$\max_{\theta \in \Theta} \max_{\lambda \in \Lambda} \sum_{i=1}^{n} \mathbb{E}_{\mathbf{z} \sim q_\lambda(\mathbf{z}|\mathbf{x}_i, \mathbf{y}_i)} \left[\log \frac{p(\mathbf{z}) p_\theta(\mathbf{x}_i|\mathbf{z}, \mathbf{y}_i)}{q_\phi(\mathbf{z}|\mathbf{x}_i, \mathbf{y}_i)} \right]. \tag{6.55}$$

6.4.2 Generative Adversarial Networks

Another powerful class of generative models are Generative Adversarial Networks (GANs), which were first introduced by Goodfellow et al. [2014]. See also a popular extension, Wasserstein GANs (W-GANs) Arjovsky et al. [2017].

Overview

Similar to VAEs, GANs also perform generative modelling in the unsupervised learning setting: we suppose as before we are given an unlabelled i.i.d. training set $\mathcal{D} = (\mathbf{x}_i)_{i=1}^N$ with each $\mathbf{x}_i \in \mathbb{R}^d$. We imagine, as before, that the $\mathbf{x}_i \overset{\text{i.i.d.}}{\sim} p_{\text{data}}(\mathbf{x})$ are drawn independently from some underlying probability distribution $p_{\text{data}}(\mathbf{x})$. Our goal, once again, is to to generate synthetic but realistic new samples $\tilde{\mathbf{x}}_j$ which look like they are also drawn from $p_{\text{data}}(\mathbf{x})$.

GANs consist of two central components: a *generator* and a *discriminator*. The generator is a deterministic function that aims to generate realistic samples $\mathbf{x} \in \mathbb{R}^d$ from latent codes $\mathbf{z} \in \mathbb{R}^r$. The discriminator is a function which aims to distinguish between samples from the real dataset, and samples from the generator.

Let's make this a little more precise. Similar to before, we begin by sampling a latent variable $\mathbf{z} \in p(\mathbf{z})$, where $p(\mathbf{z})$ is a simple, fixed distribution on $\mathbb{R}^r$, e.g., $p(\mathbf{z}) = \mathcal{N}(\mathbf{z}|0, \mathbf{I}_r)$. The generator is a deep neural network $g_\theta : \mathbb{R}^r \to \mathbb{R}^d$, parameterised by $\theta \in \Theta$, which maps $\mathbf{z} \in \mathbb{R}^r$ to $\mathbf{x} \in \mathbb{R}^d$. The idea is to train this function such that, given $\mathbf{z} \sim p(\mathbf{z})$,

$$\mathbf{x} = g_\theta(\mathbf{z}) \tag{6.56}$$

is approximately distributed according to $p_{\text{data}}(\mathbf{x})$. Thus, in particular, by drawing $\mathbf{z}_1, \ldots, \mathbf{z}_N \overset{\text{i.i.d.}}{\sim} p(\mathbf{z})$, we can generate realistic synthetic samples by computing $\mathbf{x}_1 = g_\theta(\mathbf{z}_1), \ldots, \mathbf{x}_N = g_\theta(\mathbf{z}_N)$.

The discriminator $d_\phi : \mathbb{R}^d \to [0, 1]$ is another neural network, which aims to predict whether a sample is real, $\mathbf{x} \sim p_{\text{data}}(\mathbf{x})$, or fake, $\mathbf{x} \sim p_\theta(\mathbf{x})$. Here, we have introduced the notation $p_\theta(\mathbf{x})$ for the distribution implicitly defined by the generator, viz, $\mathbf{z} \sim p(\mathbf{z})$, $\mathbf{x} = g_\theta(\mathbf{z})$. This is essentially a binary classification task. In particular, suppose that for each sample $\mathbf{x} \in \mathbb{R}^d$, we define the label $y \in \{0, 1\}$ as

$$y = \begin{cases} 1\ , & \mathbf{x} \sim p_{\text{data}}(\mathbf{x}), \\ 0\ , & \mathbf{x} \sim p_\theta(\mathbf{x}). \end{cases} \tag{6.57}$$

Our aim is to learn $d_\phi(\mathbf{x}) \approx p(y = 1|\mathbf{x})$, i.e., the probability that $\mathbf{x}$ is a real sample. Thus, if $d_\phi(\mathbf{x}) \approx 1$, the discriminator is very confident that $\mathbf{x}$ is a real data point. Meanwhile, if $d_\phi(\mathbf{x}) \approx 0$, it is very confident that $\mathbf{x}$ is a fake sample. On the other hand, if $d_\phi(\mathbf{x}) \approx 1/2$, then it is unsure whether or not $\mathbf{x}$ is real or fake: it is either a real sample, or a very convincing fake.

Training GANs

We will train the generator and the discriminator competitively. This is why the term *adversarial* enters into *generative adversarial networks*. Informally, the generator will try to fool the discriminator by generating samples which are indistinguishable from the true samples. Meanwhile, the discriminator will try to distinguish between the true samples and the fake samples.

More precisely, we will train via the generator and the discriminator according to a *two-player minimax game*. In particular, the GAN objective function

is given by

$$\min_{\theta} \max_{\phi} \Bigg[\underbrace{\mathbb{E}_{\mathbf{x}\sim p_{\text{data}}(\mathbf{x})}[\log d_\phi(\mathbf{x})] + \mathbb{E}_{\mathbf{z}\sim p(\mathbf{z})}[\log(1 - d_\phi(g_\theta(\mathbf{z})))]}_{V(\phi,\theta)} \Bigg]. \tag{6.58}$$

Let's try to unpack this loss function. For a fixed generator g_θ, the training objective for the discriminator is

$$\max_{\phi} \mathbb{E}_{\mathbf{x}\sim p_{\text{data}}(\mathbf{x})}[\log d_\phi(\mathbf{x})] + \mathbb{E}_{\mathbf{x}\sim p_\theta(\mathbf{x})}[\log(1 - d_\phi(\mathbf{x}))] \tag{6.59}$$

$$= \min_{\phi} \Big[-\mathbb{E}_{\mathbf{x}\sim p_{\text{data}}(\mathbf{x})}[\log d_\phi(\mathbf{x})] - \mathbb{E}_{\mathbf{x}\sim p_\theta(\mathbf{x})}[\log(1 - d_\phi(\mathbf{x}))] \Big] \tag{6.60}$$

The function in the second line is known as the *cross entropy loss*, and is popular objective used in binary classification problems. For a fixed generator g_θ, the optimal discriminator can actually be obtained in closed form. In particular, it is given by

$$d_{\phi^*}(\mathbf{x}) = \frac{p_{\text{data}}(\mathbf{x})}{p_{\text{data}}(\mathbf{x}) + p_\theta(\mathbf{x})} \tag{6.61}$$

Meanwhile, for a fixed discriminator, the training objective for the generator is

$$\min_{\theta} \Big[\mathbb{E}_{\mathbf{x}\sim p_{\text{data}}(\mathbf{x})}[\log d_\phi(\mathbf{x})] + \mathbb{E}_{\mathbf{x}\sim p_\theta(\mathbf{x})}[\log(1 - d_\phi(\mathbf{x}))] \Big] \tag{6.62}$$

$$= \min_{\theta} \mathbb{E}_{\mathbf{x}\sim p_\theta(\mathbf{x})}[\log(1 - d_\phi(\mathbf{x}))]. \tag{6.63}$$

In other words, the generator tries to minimise the log-probability that the discriminator makes the correct decisions for fake samples. Now, if we substitute in the optimal discriminator, then we have

$$\mathbb{E}_{\mathbf{x}\sim p_{\text{data}}(\mathbf{x})}[\log d_{\phi^*}(\mathbf{x})] + \mathbb{E}_{\mathbf{x}\sim p_\theta(\mathbf{x})}[\log(1 - d_{\phi^*}(\mathbf{x}))] \tag{6.64}$$

$$= \mathbb{E}_{\mathbf{x}\sim p_{\text{data}}(\mathbf{x})}[\log \frac{p_{\text{data}}(\mathbf{x})}{p_{\text{data}}(\mathbf{x}) + p_\theta(\mathbf{x})}] + \mathbb{E}_{\mathbf{x}\sim p_\theta(\mathbf{x})}[\log(1 - \frac{p_{\text{data}}(\mathbf{x})}{p_{\text{data}}(\mathbf{x}) + p_\theta(\mathbf{x})})] \tag{6.65}$$

$$= \mathbb{E}_{\mathbf{x}\sim p_{\text{data}}(\mathbf{x})}[\log \frac{p_{\text{data}}(\mathbf{x})}{p_{\text{data}}(\mathbf{x}) + p_\theta(\mathbf{x})}] + \mathbb{E}_{\mathbf{x}\sim p_\theta(\mathbf{x})}[\log \frac{p_\theta(\mathbf{x})}{p_{\text{data}}(\mathbf{x}) + p_\theta(\mathbf{x})}] \tag{6.66}$$

$$= \mathbb{E}_{\mathbf{x}\sim p_{\text{data}}(\mathbf{x})}[\log \frac{p_{\text{data}}(\mathbf{x})}{\frac{p_{\text{data}}(\mathbf{x}) + p_\theta(\mathbf{x})}{2}}] + \mathbb{E}_{\mathbf{x}\sim p_\theta(\mathbf{x})}[\log \frac{p_\theta(\mathbf{x})}{\frac{p_{\text{data}}(\mathbf{x}) + p_\theta(\mathbf{x})}{2}}] - \log 4 \tag{6.67}$$

$$= \mathrm{D}_{\mathrm{KL}} \left[p_{\text{data}}(\mathbf{x}), \frac{p_{\text{data}}(\mathbf{x}) + p_\theta(\mathbf{x})}{2} \right] + \mathrm{D}_{\mathrm{KL}} \left[p_\theta(\mathbf{x}), \frac{p_{\text{data}}(\mathbf{x}) + p_\theta(\mathbf{x})}{2} \right] - \log 4 \tag{6.68}$$

$$= 2D_{\mathrm{JSD}} \left[p_{\text{data}}(\mathbf{x}), p_\theta(\mathbf{x}) \right] - \log 4 \tag{6.69}$$

where D_{JSD} is the *Jensen-Shannon* or *symmetric KL* divergence, which provides another way to measure the distance between two probability distribution.

This functional has similar properties to the KL divergence. In particular, $D_{\text{JSD}}(p, q) \geq 0$, with equality if and only if $p = q$. It follows that (6.69) is uniquely minimised by θ^* such that $p_{\theta^*}(\mathbf{x}) = p_{\text{data}}(\mathbf{x})$. In other words, the optimal generator $g^*(\mathbf{x}) = g_{\theta^*}(\mathbf{x})$ guarantees that $p_{\theta^*}(\mathbf{x}) = p_{\text{data}}(\mathbf{x})$. Meanwhile, the optimal value of the loss function is given by $V(\phi^*, \theta^*) = -\log 4$.

We will solve the optimisation problem in (6.58) by alternating between a (stochastic) gradient ascent step for the parameters of the discrimininator, and a (stochastic) gradient descent step for the parameters of the generator. This is summarised below.

Algorithm 15: Training GANs using (Stochastic) Gradient Ascent - Descent

---

Input: initial parameters $\theta \in \Theta$, $\phi \in \Phi$, learning rates $\gamma \in \mathbb{R}_+$, $\eta \in \mathbb{R}_+$.
For $t = 1, \ldots, T$:

- Sample a minibatch of data samples $(\mathbf{x}_i)_{i=1}^m \sim p_{\text{data}}(\mathbf{x})$.
- Sample of minibatch of noise samples $\mathbf{z} \sim p(\mathbf{z})$.
- Update the discriminator parameters using a stochastic gradient ascent step

$$\phi \leftarrow \phi + \gamma \frac{1}{m} \nabla_\phi \sum_{j=1}^{m} \left[\log d_\phi(\mathbf{x}_j) + \log(1 - d_\phi(\mathbf{x}_j)) \right] \tag{6.70}$$

- Update the generator parameters using a stochastic gradient descent step

$$\theta \leftarrow \theta - \eta \frac{1}{m} \nabla_\theta \sum_{j=1}^{m} \left[\log(1 - d_\phi(g_\theta(\mathbf{z}_j))) \right] \tag{6.71}$$

An Alternative Loss for the Generator

While the minimax formulation is useful for theoretical analysis, it does not work particularly well in practice. This is due to a problem known as *saturation*. Essentially, when the discriminator is doing very well, or equivalently the generator is doing very badly, the generator's gradient vanishes. This scenario is common near the start of training, and prevents the generator from improving.

In Goodfellow et al. [2014], the authors propose an alternative *non-saturated* objective for the generator, given by

$$\max_\theta \mathbb{E}_{\mathbf{x} \sim p_\theta(\mathbf{x})} \left[\log d_\phi(\mathbf{x}) \right]. \tag{6.72}$$

In comparison to the original generator objective, which minimises the log probability of the discriminator making correct predictions, this alternative objective maximises the log probability of the discriminator making the wrong predictions.

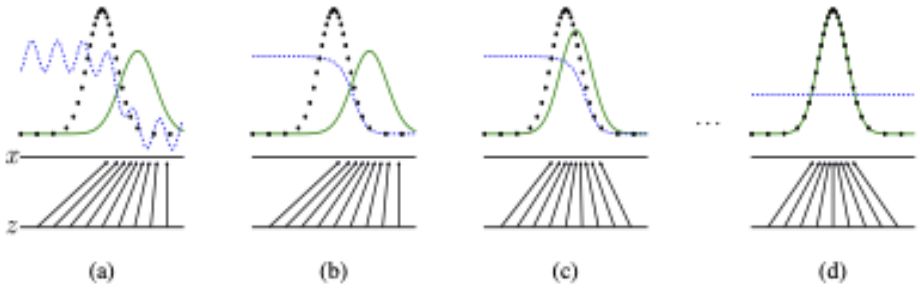


Figure 6.1: **Training a GAN to model a one-dimensional distribution**. The distribution $p_{\text{data}}(\mathbf{x})$, the discriminator $d_\phi(\mathbf{x})$ is shown in blue, and the (implicit) generator distribution $p_\theta(\mathbf{x})$ is shown in green. From Goodfellow et al. [2014].

In this case, the gradient signal is strong when the discriminator confidently rejects generator samples, which resolves the saturation problem.

Conditional GANs

Similar to conditional VAEs, conditional GANs are designed for the task of conditional generative modelling. In particular, similar to before, we suppose that we have access to $(\mathbf{x}_i, \mathbf{y}_i)_{i=1}^N$, where $(\mathbf{x}_i, \mathbf{y}_i) \overset{\text{i.i.d.}}{\sim} p_{\text{data}}(\mathbf{x}, \mathbf{y})$. Once again, we think of $\mathbf{x}_i$ as data samples (e.g., images) and $\mathbf{y}_i$ as additional information (e.g., class labels).

Conditional GANs are much the same as the GANs we have seen so far. Once again, they consist of two components: a generator and a discriminator. In this case, however, the generator is not only a function of the latent component $\mathbf{z}$, but also of the additional variable $\mathbf{y}$. In particular, one now generates samples by first sampling $\mathbf{z} \sim p(\mathbf{z})$, then sampling $\mathbf{y} \sim p_{\text{data}}(\mathbf{y})$, and then setting

$$\mathbf{x} = g_\theta(\mathbf{z}, \mathbf{y}). \tag{6.73}$$

This procedure implicitly defines a conditional distribution $p_\theta(\mathbf{x}|\mathbf{y})$. Similarly, the discriminator $d_\phi(\mathbf{x}, \mathbf{y})$ is now a function of both $\mathbf{x}$ and $\mathbf{y}$.

Similar to before, learning is achieved by optimising an adversarial objective, in this case

$$\min_\theta \max_\phi \Bigg[\underbrace{\mathbb{E}_{\mathbf{x}\sim p_{\text{data}}(\mathbf{x},\mathbf{y})}[\log d_\phi(\mathbf{x})] + \mathbb{E}_{\mathbf{z}\sim p(\mathbf{z}),\mathbf{y}\sim p_{\text{data}}(\mathbf{y})}[\log(1 - d_\phi(g_\theta(\mathbf{z}, \mathbf{y}), \mathbf{y}))]}_{V(\phi,\theta)} \Bigg]. \tag{6.74}$$

In this case, one can show that the optimal discriminator for a fixed generative model with parameter θ is given by

$$d_{\phi^*}(\mathbf{x}, \mathbf{y}) = \frac{p_{\text{data}}(\mathbf{x}|\mathbf{y})}{p_{\text{data}}(\mathbf{x}|\mathbf{y}) + p_\theta(\mathbf{x}|\mathbf{y})} \tag{6.75}$$

Figure 6.2: **Samples from a GAN.** From Shahbazi et al. [2022].

Figure 6.3: **Samples from a conditional GAN.** Each row represents a different class label. From Shahbazi et al. [2022].

and that, given the optimal discriminator, the optimal generator is the generator such that $p_{\text{data}}(\mathbf{x}|\mathbf{y}) = p_\theta(\mathbf{x}|\mathbf{y})$.

6.4.3 VAEs versus GANs

Having discussed VAEs and GANs for the task of generative modelling, it is very natural to compare them. There are some key similarities:

- Both methods make use of a latent space, on which we define a simple distribution (a standard Gaussian).
- Both methods then learn a transformation which maps latent variables $\mathbf{z}$ to (hopefully) convincing fake data points $\mathbf{x}$.

The crucial difference is in how the methods construct this mapping, and how they are subsequently trained.

- For VAEs, we make use of encoder-decoder formalism to define an explicit probabilistic model. We then optimize a *lower bound* of the resulting likelihood. The advantage is that we have a (lower bound on the) likelihood, which can be used as a measure of success.
- For GANs, we no longer have an explicit probabilistic model, and instead compete two neural networks against each other to create an effective generator. Because of this adversarial nature, training GANs can be tricky in practice.

In terms of the output, GANs are generally regarded to be the state of the art in generative modelling. This can be measured in terms of metrics such as the *inception score* or the *Frêchet inception score*. However, they can be much harder to train than VAEs, which also provide a probabilistic model.

Appendix A

Constrained optimization

A.1 First-order conditions

Here, we detail the theory of solving general inequality-constrained optimization problems of the form

$$\min_w f(w) \quad \text{subject to } c_j(w) \geq 0 \quad j \in \mathcal{I}, \tag{A.1}$$

where f and c_j are smooth functions.

Such problems can be solved by defining a *Lagrangian*

$$L_P(w, \lambda) = f(w) - \sum_{j \in \mathcal{I}} \lambda_j c_j(w), \tag{A.2}$$

and applying the following theorem:

Theorem 13. *Suppose that w^* is a sufficiently regular*[1] *solution to the minimization problem* (A.1) *and f and c_j are continuously differentiable. Then there exists λ^* such that*

$$\nabla_w L_P(w^*, \lambda^*) = 0, \tag{A.3}$$

$$c_j(w^*) \geq 0, \qquad j \in \mathcal{I}, \tag{A.4}$$

$$\lambda_j^* \geq 0, \qquad j \in \mathcal{I}, \tag{A.5}$$

$$\lambda_j^* c_j(w^*) = 0, \qquad j \in \mathcal{I}. \tag{A.6}$$

These necessary conditions are known as the *Karush-Kuhn-Tucker (KKT) conditions*. In the case that both f and the $-c_j$ are convex, these conditions are also sufficient for minimality.

[1] In general, w^* must satisfy a condition known as the linear independence constraint qualification (LICQ). However, in the case of the constraint functions c_j being linear (as in SVM), this is unnecessary.

In the case of the support vector classifier described in section 2.4.3, the Lagrangian for the problem is given by

$$L_P(\beta_0, \beta, \xi, \alpha, \mu) = \frac{1}{2}\|\beta\|^2 + \sum_{i=1}^{N}\left[C\xi_i - \alpha_i\left(y_i(x_i^\top\beta + \beta_0) - (1-\xi_i)\right) - \mu_i\xi_i\right], \tag{A.7}$$

and hence the KKT conditions read

$$\sum_{i=1}^{N}\alpha_i y_i = 0, \tag{A.8}$$

$$\beta - \sum_{i=1}^{N}\alpha_i y_i x_i = 0, \tag{A.9}$$

$$C - \alpha_i - \mu_i = 0, \qquad i = 1, \ldots, N, \tag{A.10}$$

$$y_i(x_i^\top\beta + \beta_0) - (1-\xi_i) \geq 0, \qquad i = 1, \ldots, N, \tag{A.11}$$

$$\xi_i \geq 0, \qquad i = 1, \ldots, N, \tag{A.12}$$

$$\alpha_i \geq 0, \qquad i = 1, \ldots, N, \tag{A.13}$$

$$\mu_i \geq 0, \qquad i = 1, \ldots, N, \tag{A.14}$$

$$\alpha_i\left(y_i(x_i^\top\beta + \beta_0) - (1-\xi_i)\right) = 0, \qquad i = 1, \ldots, N, \tag{A.15}$$

$$\mu_i\xi_i = 0, \qquad i = 1, \ldots, N. \tag{A.16}$$

A.2 Duality

Here, we explore an alternative problem known as the (Lagrange) "dual" to the "primal" problem specified above. The dual objective function is given by

$$L_D(\lambda) \coloneqq \inf_w L_P(w, \lambda). \tag{A.17}$$

When f and $-c_j$ are both convex, L_P is also convex and hence any local minimum of L_P is in fact a global minimum, greatly simplifying the computation of L_D. The dual optimization problem to (A.1) is given by:

$$\max_\lambda L_D(\lambda) \quad \text{subject to } \lambda_j \geq 0 \quad j \in \mathcal{I}. \tag{A.18}$$

The first important relation between the primal and dual problems is given by the following theorem:

Theorem 14. *For any feasible point $\bar{w}$ and $\bar{\lambda} \geq 0$, we have $L_D(\bar{\lambda}) \leq f(\bar{w})$.*

That is to say that dual problem gives a lower bound for the value of the objective function in the primal problem. Also, optimal Lagrange multipliers λ for the primal problem give solutions to the dual problem:

Theorem 15. *Let $\bar{w}$ solve* (A.1) *and f, $-c_j$ be convex and continuously differentiable. Then, if $(\bar{w}, \bar{\lambda})$ satisfies the KKT conditions eqs.* (A.3) *to* (A.6)*, $\bar{\lambda}$ solves* (A.17)*.*

Finally, under the additional condition of strict convexity of L_P as a function of w, we also have the converse result, which allows us to solve the primal problem through solutions of the dual problem:

Theorem 16. *Let $\bar{w}$ be a sufficiently regular solution of* (A.1) *and f, $-c_j$ be convex and continuously differentiable. If $\hat{\lambda}$ solves* (A.17) *and $L_P(\cdot, \hat{\lambda})$ is strictly convex with infimum attained at $\hat{w}$, then in fact $\bar{w} = \hat{w}$ and $f(\bar{w}) = L_P(\hat{w}, \hat{\lambda})$.*

Another, slightly different notion of duality is the Wolfe dual problem, which is defined by:

$$\max_{w,\lambda} L_P(w, \lambda), \tag{A.19}$$

$$\text{subject to } \nabla_w L_P(w, \lambda) = 0, \quad \lambda_j \geq 0 \quad j \in \mathcal{I}. \tag{A.20}$$

This is related to the primal problem by the following theorem:

Theorem 17. *Let $\bar{w}$ be a sufficiently regular solution of* (A.1) *and f, $-c_j$ be convex and continuously differentiable. Then, if $(\bar{w}, \bar{\lambda})$ satisfies the KKT conditions eqs.* (A.3) *to* (A.6)*, $\bar{\lambda}$ solves the Wolfe dual problem eqs.* (A.19) *and* (A.20)*.*

Appendix B

Basic facts regarding KL divergence

We briefly review the basic definitions and properties of the Kullback–Leibler (KL) divergence.

Given two probability density functions p, q on a state space $\mathcal{X}$, the KL divergence is defined as

$$D_{\mathrm{KL}}(p|q) = \mathbb{E}_{X\sim p}\left[\log \frac{p(X)}{q(X)}\right] = \int p(x) \log \frac{p(x)}{q(x)}\,\mathrm{d}x.$$

Lemma 18. *For any densities p, q, $D_{\mathrm{KL}}(p|q) \geq 0$.*

Proof. If $D_{\mathrm{KL}}(p|q) = \infty$ then the result is vacuously true. Otherwise, use Jensen's inequality:

$$\begin{aligned} D_{\mathrm{KL}}(p|q) &= \mathbb{E}\left[-\log \frac{q(X)}{p(X)}\right] \\ &\geq -\log \mathbb{E}\left[\frac{q(X)}{p(X)}\right] \\ &= -\log \int \frac{q(x)}{p(x)} p(x)\,\mathrm{d}x \\ &= -\log 1 \\ &= 0. \end{aligned}$$

□

Bibliography

Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein Generative Adversarial Networks. In *Proceedings of the 34th International Conference on Machine Learning*, volume 70, pages 214–223. PMLR, jul 2017. URL `https://proceedings.mlr.press/v70/arjovsky17a.html`.

Louis J.M. Aslett. Statistical Machine Learning, Durham University, 2022. `https://www.louisaslett.com/StatML/notes/`, accessed March 3, 2023.

David M Blei, Andrew Y Ng, and Michael I Jordan. Latent dirichlet allocation. *Journal of machine Learning research*, 3(Jan):993–1022, 2003.

Leo Breiman. Statistical modeling: The two cultures (with comments and a rejoinder by the author). *Statistical science*, 16(3):199–231, 2001.

Gerda Claeskens and Nils Lid Hjort. *Model Selection and Model Averaging*. Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press, 2008. doi: 10.1017/CBO9780511790485.

Soheil Feizi. *Foundations of Deep Learning*. University of Maryland, 2020. URL `http://www.cs.umd.edu/class/fall2020/cmsc828W/`.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

Ian J Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative Adversarial Nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014. URL `http://www.github.com/goodfeli/adversarial`.

Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer, 2 edition, 2008.

Irina Higgins, Loic Matthey, Arka Pal, Christopher Burgess, Xavier Glorot, Matthew Botvinick, Shakir Mohamed, and Alexander Lerchner. B-VAE: Learning basic visual concepts with a constrained variational framework. *International Conference on Learning Representations*, 2017. URL `https://openreview.net/pdf?id=Sy2fzU9gl`.

Gareth James, Daniela Witten, Trevor Hastie, Robert Tibshirani, Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. Statistical learning. *An introduction to statistical learning: with applications in R*, pages 15–57, 2021.

Ramesh Johari. MS&E 226 Notes, 2016. `http://web.stanford.edu/~rjohari/teaching/notes.html`, accessed March 3, 2023.

Diederik P Kingma and Max Welling. Auto-Encoding Variational Bayes. In *ICLR*, 2014. URL `https://arxiv.org/abs/1312.6114`.

Wei-Yin Loh. Fifty years of classification and regression trees. *International Statistical Review*, 82(3):329–348, 2014.

Karl Pearson. On lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin philosophical magazine and journal of science*, 2(11):559–572, 1901.

Ali Razavi, Aaron van den Oord, and Oriol Vinyals. Generating Diverse High-Fidelity Images with VQ-VAE-2. In *Advances in Neural Information Processing Systems*. arXiv, jun 2019. URL `http://arxiv.org/abs/1906.00446`.

Erhard Schmidt. Zur theorie der linearen und nichtlinearen integralgleichungen. iii. teil: Über die auflösung der nichtlinearen integralgleichung und die verzweigung ihrer lösungen. *Mathematische Annalen*, 65(3):370–399, 1908.

Rajen D. Shah. *Modern statistical methods*. 2020. URL `http://www.statslab.cam.ac.uk/~rds37/teaching/modern_stat_methods/notes_MSM.pdf`. Accessed May 2021.

Mohamad Shahbazi, Martina Danelljan, Danda Pani Paudel, and Luc Van Gool. Collapse by Conditioning: Training Class Conditional GANs with Limited Data. In *10th International Conference on Learning Representations (ICLR 2022)*. International Conference on Learning Representations, ICLR, 2022. URL `https://arxiv.org/abs/2201.06578`.

Alex Thomo. Latent semantic analysis. URL `https://www.engr.uvic.ca/~seng474/svd.pdf`.

Ryan Tibshirani. Advanced data analysis from an elementary point of view, 2014. `https://www.stat.cmu.edu/~ryantibs/`, accessed March 20, 2023.

Aaron van den Oord, Oriol Vinyals, and Koray Kavukcuoglu. Neural Discrete Representation Learning. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL `http://arxiv.org/abs/1711.00937`.

Sebastian vanGerwen. The differences between machine learning models, 2021. `https://www.thedataschool.com.au/sebastian-van-gerwen/an-intuitive-guide-to-the-differences-between-machine-learning-models/`, accessed March 10, 2023.

Aston Zhang, Zack C. Lipton, Mu Li, and Alex J. Smola. *Dive into Deep Learning*. 2021. URL `http://d2l.ai/index.html`.