Abstract of the introduction to *Statistical Learning*

Siger

February 21, 2023

Si Dieu est infini, alors je suis une partie de Dieu sinon je serai sa limite...

Contents

Chapter 1

Introduction to Statistical Learning

1.1 Statistical Learning

1.1.1 What is statistical Learning?

What is Statistical Learning? Notation:

- **Input variables**: predictors, independent variables features or variables.
- **Output variables**: response, dependent variables.

When we observe a quantitative response Y knowing there are p predictors such as
$X = (X_i)_{1 \leq i \leq p}$ then we write: $\boxed{Y = f(X) + \epsilon}$

$\begin{cases} f \text{ is some fixed but unknown function of } X \text{ and represents the systematics information that } X \text{ provides about } Y \\ \epsilon \text{ is a random error term, independent of X and has mean 0} \end{cases}$

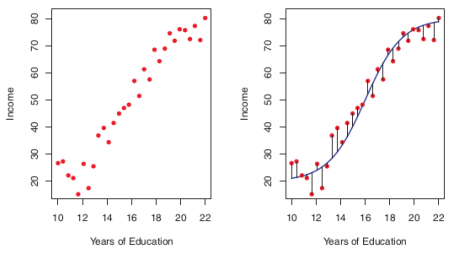


Figure 1.1: Estimation of f

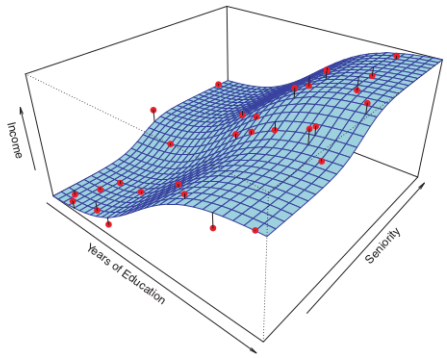


Figure 1.2: Estimation of f in $2-D$

In essence **Statistical Learning** refers to a set of approaches for estimating f.

Why estimate f

Prediction $\boxed{\widehat{Y}=\widehat{f}(X)}$ $\begin{cases}\widehat{f} \text{ represents our estimating for } f\\ \widehat{Y} \text{ represents the resulting prediction for } Y\end{cases}$

$\widehat{f}$ is often treated as a *black box* since we pay more attention to its prediction accuracy than its exact form.
The accuracy of $\widehat{Y}$ depends on 2 quantities
$\begin{cases}\textit{reductible error}, \text{ can be improved by using most appropriate statistical learning technique}\\ \textit{irreductible error}, \text{ cannot be changed and have external causes which are out of control}\end{cases}$
simple calculus shows that:

$$\begin{aligned}\mathbb{E}\left(\left[Y-\widehat{Y}\right]^2\right) &= \mathbb{E}\left(\left[f(X)+\epsilon-\widehat{f}(X)\right]^2\right)\\ &= \underbrace{\mathbb{E}\left(\left[f(X)-\widehat{f}(X)\right]^2\right)}_{Reducible} + \underbrace{\mathbb{V}(\epsilon)}_{Irreducible} \quad \text{think that } \mathbb{E}(\epsilon)=0\end{aligned}$$

Inference Now we want to know how Y evolves when X changes, so we cannot considerate anymore f as a black box.

- *Which predictors are associated with the response?* (to discover variables which have the most important influence therewith to reduce number of considered variables)
- *What is the relationship between the response and each predictor?*(Which components increase Y value and which decrease it)
- *Can the relationship between Y and each predictor be adequately summarized using linear equation or is the relationship more complicated?*

How do we estimate f

Aim Let $(i,j) \in [\![1,n]\!] \times [\![1,p]\!]$ and x_{ij} represent the value of the j^{th} predictor for i^{th} observation. Correspondingly let y_i represent the response variable for the i^{th} observation.

Then our training data consists of $\left\{(x_i, y_i)_{1 \leq i \leq n}\right\}$ where $x_i = \begin{pmatrix} x_{i1} \\ . \\ . \\ . \\ x_{ip} \end{pmatrix}$.

We want to find a function $\widehat{f}$ such that $Y \approx \widehat{f}(X)$ for any observation (X, Y)

Parametric methods

1. *Assumption about functional form of f*

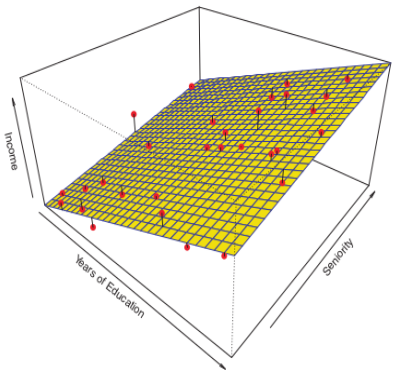
   For example: $f(X) = \beta_0 + \sum_{i=1}^{p} \beta_i X_i$ in this case instead to estimate entirely p-dimensional function f we only need to estimate $(\beta_i)_{0 \leq i \leq p}$

2. After the model selection, we need a procedure which uses *data training* to *fit* or *train* the model.

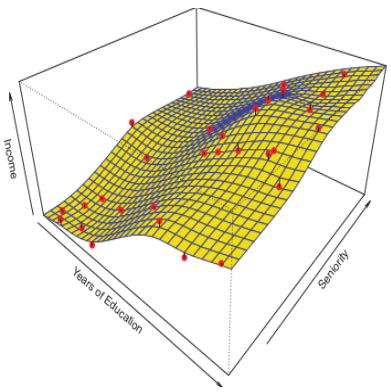
   For example we need to estimate $(\beta_i)_{0 \leq i \leq p}$ such that: $Y \approx \beta_0 + \sum_{i=1}^{p} \beta_i X_i$

   The most common approach to fit the model is *least squares*.

The parametric methods allow to reduce the problem of estimating f down to one of estimating a set of parame



(a) Linear estimation: $income = \beta_0 + \beta_1 \times education + \beta_2 \times seniority$



(b) Best estimation of f for $income \approx f(income, seniority)$

Figure 1.3: Estimation of f with 2 degrees of precision

Non-Parametric methods Any parametric approaches brings with it the possibility that the functional form used to estimate f is very different from the true f. In contrast non-parametric approaches completely avoid this danger since no assumption about the form of f is made. But non parametric approaches do suffer from non-reducing problem, and they need a very greater number of observation than with parametric approaches.

The trade-off between Prediction Accuracy and model Interpretability When inference is the goal, there are clear advantages to using simple and relatively inflexible statistical learning methods.

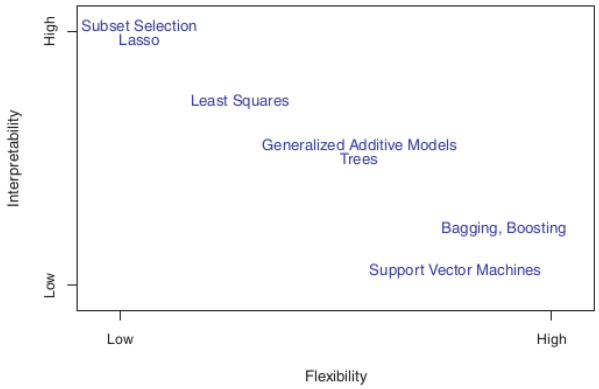


Figure 1.4: To know which methods to use.

Supervised versus unsupervised learning We speak about *supervised problems* when for each predictors x_i there is an associated response measurement y_i.
Whereas in *unsupervised problems* we observe a vector of measurements x_i but not associated response y_i. Then we can seek to understand the relationship between the variables or between observations.
One statistical tool that we may use in this setting is *clustering*

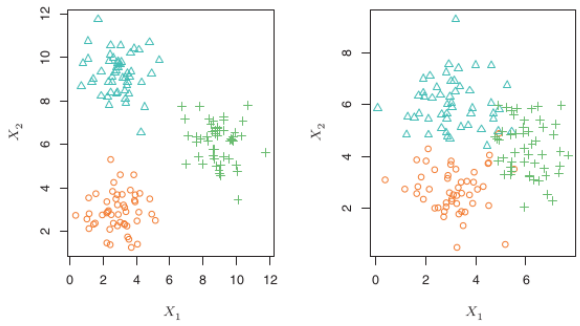


Figure 1.5: Clustering methods is used for *unsupervised problems*.

Regression versus Classification problems In general Regression is used for quantitative variables whereas classification is used for qualitative variables but we can find several counter-examples.

1.1.2 Assessing Model Accuracy

Measuring the quality of a fit In regression setting, the most commonly-used measure of model accuracy is the *Mean Squared Error (MSE)*: $MSE = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - \widehat{f}\,(x_i)\right)^2$

We do not really care about whether for all $i \in [\![1, n]\!]\, \widehat{f}(x_i) \approx y_i$, instead we want to know whether a previously unseen observation not used to train the statistical learning method (x_0, y_0) is such that $\widehat{f}(x_0) \approx y_0$.
In other words if we had a large number of test observations, we could compute

$$Ave\left(\left(y_0 - \widehat{f}(x_0)\right)^2\right)$$

We want to choose the method that gives the lowest *test MSE*, as opposed to the lowest *training MSE*.
Many statistical methods estimate coefficients so as to minimize the training set MSE, then training set MSE can be quite small, but test MSE is often much larger.

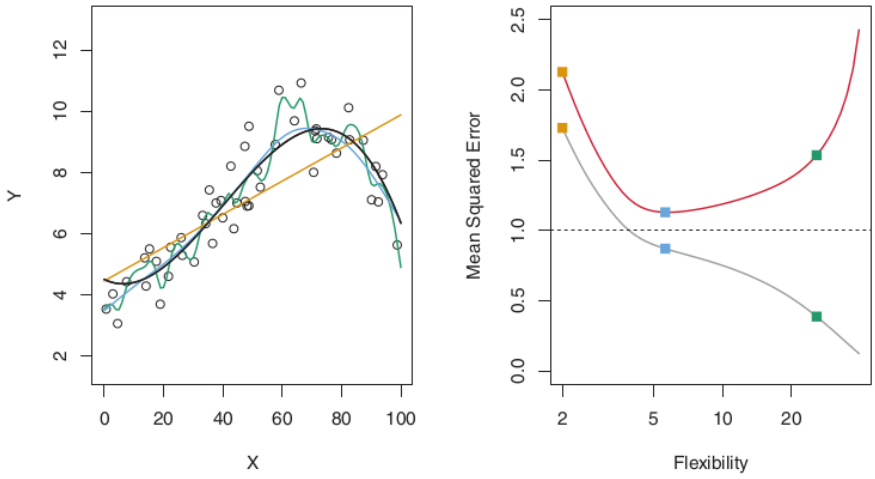


Figure 1.6: Left:Data simulated from f, shown in black and 3 estimates of f
Right: Training MSE in grey and test MSE in red.
Squares represent the training and test MSEs for the 3 fits shown in the left-hand pannel

The blue curve minimizes the test MSE, and visually appears to estimate f the best in the left-hand panel.
The horizontal dashed line indicates $\mathbb{V}(\epsilon)$ the irreducible error in which corresponds to the lowest achievable test MSE among all possible methods.
When a given method yields a high training MSE but a low test MSE we say that data are *overfitting*.

The Bias-Variance Trade-Off

$$\mathbb{E}\left(\left(y_0 - \widehat{f}(x_0)\right)^2\right) = \mathbb{V}\left(\widehat{f}(x_0)\right) + \left[Bias\left(\widehat{f}(x_0)\right)\right]^2 + \mathbb{V}(\epsilon)$$

$\begin{cases} \mathbb{E}\left(\left(y_0 - \widehat{f}(x_0)\right)^2\right) & \text{defines the } \textit{expected test MSE} \text{ and refers to the average test MSE.} \\ \mathbb{V}\left(\hat{f}(x_0)\right) & \text{the amount by which } \widehat{f} \text{ would change if we estimating it using a different training data.} \\ \left[Bias\left(\hat{f}(x_0)\right)\right]^2 & \text{refers to the error that is introduced by approximating a real-life problem.} \end{cases}$

As we increase the flexibility of a class methods, the bias tends to initially decrease faster than the variance increases.

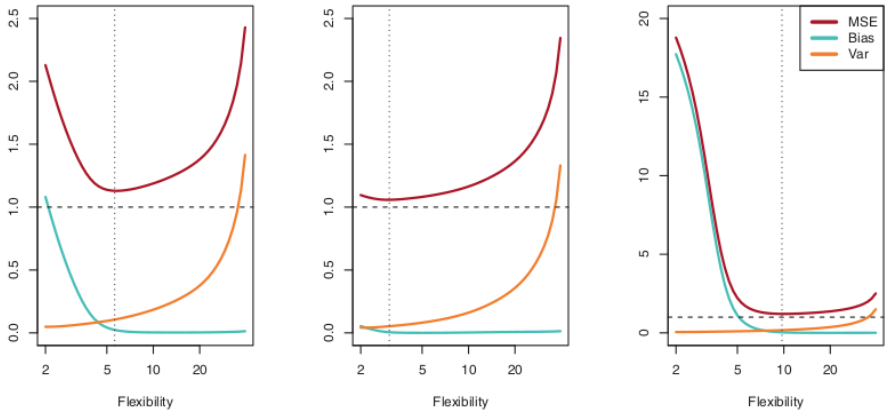


Figure 1.7: Squared bias, variance and test MSE.
$\mathbb{V}(\epsilon)$ indicates by the dashed line.

1.1.3 Bias, Variance and Model Complexity

The Loss function for measuring errors between Y and $\hat{f}(X)$ is denoted by $L\left(Y, \hat{f}(X)\right)$. Typical choices are:

$$L\left(Y, \hat{f}(X)\right) = \begin{cases} \left(Y - \hat{f}(X)\right)^2 & \text{squared error} \\ \left|Y - f(\hat{X})\right| & \text{absolute error} \end{cases}$$

There are 3 important quantities:

$$\begin{cases} \overline{err} = \dfrac{1}{N}\displaystyle\sum_{i=1}^{N} L\left(y_i, \hat{f}(x_i)\right) \\ Err_{\mathcal{T}} = \mathbb{E}\left(L\left(Y, \hat{f}(X)\right)|\mathcal{T}\right) \quad \textit{Test error} \\ Err = \mathbb{E}\left(L\left(Y, \hat{f}(X)\right)\right) = \mathbb{E}\left(Err_{\mathcal{T}}\right) \quad \textit{Test error} \end{cases}$$

The classification setting Suppose that we seek to estimate f on the basis of training observations $\{(x_i, y_i)\}_{1 \le i \le n}$ where now $(y_i)_{1 \le i \le n}$ are qualitative. The most common approach for quantifying the accuracy of our estimate $\widehat{f}$ is the training *error rate*:

$$\frac{1}{n}\sum_{i=1}^{n} I_{y_i \neq \widehat{y_i}} \text{ with } I_{y_i \neq \widehat{y_i}} \begin{cases} 1 \Leftarrow y_i \neq \widehat{y}_i \\ 0 \Leftarrow y_i = \widehat{y}_i \end{cases}.$$

The *test error rate* associated with a set of test observation of the form (x_0, y_0) is given by: $Ave\left(I_{y_0 \neq \widehat{y_0}}\right)$

1.2 Linear Regression

Here are a few important questions that we might seek to address: Linear model is a model of the form:

$$p(y|\boldsymbol{x}, \boldsymbol{\theta}) = \mathcal{N}\left(\boldsymbol{\beta}^T \boldsymbol{x}, \sigma^2\right)$$

. Linear regression can be made to model non-linear reltationships by replacing $\boldsymbol{x}$ with some non-linear function of the inputs ϕ

$$p(y|\boldsymbol{x}, \boldsymbol{\theta}) = \mathcal{N}\left(\boldsymbol{\beta}^T \phi(\boldsymbol{x}), \sigma^2\right)$$

.

1.2.1 Ordinary Least Squares Regression

Estimating the Regression Coefficients A common way to estimate the parameters of a statistical model is to compute the MLE(Maximum Likelihood Estimation) defined as

$$\hat{\boldsymbol{\theta}} \triangleq \arg\max_{\theta} \log\left(p(\mathcal{D}|\boldsymbol{\theta})\right)$$

$$\begin{aligned} l(\boldsymbol{\theta}) &\triangleq \log\left(p(\mathcal{D}|\boldsymbol{\theta})\right) \\ &= \sum_{i=1}^{n} \log\left(p(y_i|\boldsymbol{x_i}, \boldsymbol{\theta})\right) \\ &= \sum_{i=1}^{n} \log\left(\left[\frac{1}{2\pi\sigma^2}\right]^{\frac{1}{2}} \exp\left(-\frac{1}{2\sigma^2}\left[y_i - \boldsymbol{\beta}^T \boldsymbol{x_i}\right]\right]^2\right)\right) \\ &= \frac{1}{2\sigma^2} RSS(\boldsymbol{\beta}) + \frac{n}{2}\log(2\pi\sigma^2) \end{aligned}$$

Then the Residual Sum of Squares (RSS) is equal to $\sum_{i=1}^{n}\left(y_i - \beta^T x_i\right)^2$ Instead of maximizing the log-likelihood we can equivalently minimize the Negative Log Likelihood (NLL)

$$NLL(\beta) \triangleq l(\beta)$$

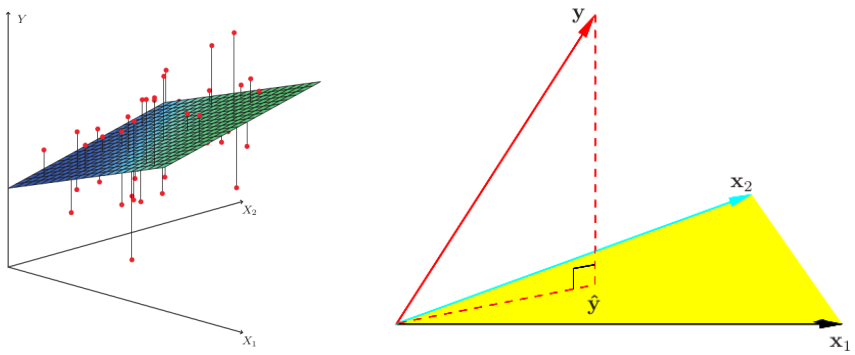
Considering $\boldsymbol{X}$ the $N \times (p+1)$ matrix with each row an input vector and y be the $N - vector$ of outputs in the training set.

$$RSS(\beta) = (y - \boldsymbol{X}\beta)^T (y - \boldsymbol{X}\beta)$$

Differentiating with respect to β we obtain: $\begin{cases} \dfrac{\partial RSS}{\partial \beta} = -2\boldsymbol{X}^T(y - \boldsymbol{X}\beta) \\ \dfrac{\partial^2 RSS}{\partial \beta \partial \beta^T} = 2\boldsymbol{X}^T\boldsymbol{X} \end{cases}$

Assuming that $\boldsymbol{X}$ has full column rank, we set the first derivative to 0:
$\boldsymbol{X}^T(y - \boldsymbol{X}\beta) = 0$ to obtain the unique solution:

$$\boxed{\hat{\beta} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y}}$$



(a) n observations (b) 1 observation

Figure 1.8: Least squares for a linear model with 2 predictors of p-dimensions.

$$\begin{aligned}\hat{\boldsymbol{y}} &= \boldsymbol{X}\hat{\beta} \\ &= \boldsymbol{X}\left(\boldsymbol{X}^T\boldsymbol{X}\right)^{-1}\boldsymbol{X}^T\boldsymbol{y} \\ &= \boldsymbol{H}\boldsymbol{y}\end{aligned}$$

$\boldsymbol{H}$ is called "hat" matrix because it puts the hat on $\boldsymbol{y}$.

Hat Matrix Residuals can also be expressed as a function of $\boldsymbol{H}$, $\hat{\boldsymbol{e}} = \boldsymbol{y} - \hat{\boldsymbol{y}} = \boldsymbol{y} - \boldsymbol{H}\boldsymbol{y} = (\boldsymbol{I} - \boldsymbol{H})\boldsymbol{y}$. It can be shown that $\boldsymbol{H}$ and $\boldsymbol{I} - \boldsymbol{H}$ are orthogonal projections.
One can easily show that $\boldsymbol{H}\boldsymbol{H} = \boldsymbol{H}$ and $(\boldsymbol{I} - \boldsymbol{H})(\boldsymbol{I} - \boldsymbol{H}) = \boldsymbol{I} - \boldsymbol{H}$

Range and Kernel of the Hat Matrix $rank(\boldsymbol{X}) = rank(\boldsymbol{X}^T\boldsymbol{X}) = p^*$

Residual and Fitted Values $\boldsymbol{H}(\boldsymbol{I}-\boldsymbol{H}) = \boldsymbol{H} - \boldsymbol{H}\boldsymbol{H} = 0$, hence $\langle \hat{\boldsymbol{y}} | \hat{\boldsymbol{e}} \rangle = 0$. Therefore $\hat{\boldsymbol{y}}$ and $\boldsymbol{e}$ are orthogonal in $\mathbb{R}^n$.

Geometric interpretation The degrees of freedom associated with $\hat{\boldsymbol{y}}$ and $\hat{\boldsymbol{e}}$ can be seen to simply be the dimensions of the respective vector subspace in which these 2 vectors have been projected.
The vectors $\boldsymbol{y}, \hat{\boldsymbol{y}}$ and $\hat{\boldsymbol{e}}$ determine 3 points in $\mathbb{R}^n$ which form a right-angled triangle, we can see the decomposition of total sum of squares into estimated sum of squares and residual sum of squares as a special case of *Pythagoras* theorem.

Further information It might happen that the columns of $\boldsymbol{X}$ are not linearly independent, then $\boldsymbol{X}^T\boldsymbol{X}$ is singular and the least squares coefficients $\hat{\beta}$ are not uniquely defined.
Knowing that $\mathbb{V}(\boldsymbol{A}\boldsymbol{y}) = \boldsymbol{A}\mathbb{V}(\boldsymbol{y})\boldsymbol{A}^T$:

$$\boxed{\mathbb{V}\left(\hat{\beta}\right) = \left(\boldsymbol{X}^T\boldsymbol{X}\right)^{-1}\sigma^2}$$

a estimate of σ^2:$\hat{\sigma}^2 = \frac{1}{N-p-1}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$ The $n-p-1$ rather than n makes $\hat{\sigma}^2$ an unbiased estimate.
$\begin{cases} \hat{\beta} \hookrightarrow \mathcal{N}\left(\beta, (\boldsymbol{X}^T\boldsymbol{X})^{-1}\sigma^2\right) \\ (n-p-1)\hat{\sigma}^2 \hookrightarrow \sigma^2\chi^2_{n-p-1} \end{cases}$

Convexity Functions having a bowl shape with a unique minimum, more precisely:

$$\forall(\boldsymbol{\theta}, \boldsymbol{\theta'}, \lambda) \in \mathcal{S} \times \mathcal{S} \times [0,1],\ \lambda\boldsymbol{\theta} + (1-\lambda)\boldsymbol{\theta'} \in \mathcal{S} \Rightarrow \mathcal{S} \text{ is } \textbf{convex}$$

1.2.2 Robust Regression

Relevancy of using robust regression model In the common way to model the noise, with a Gaussian distribution, maximizing likelihood is equivalent to minimizing the sum of squared residuals, however squared error penalizes deviation quadratically, then a few outliers can provoke poor model fitting.

Potential approach to handle bypass outlier presence Replace Gaussian distribution with **heavy tail** such it will assign higher likelihood to outliers without impacting the main hyperspace explaining them. Like Laplace distribution of which probability function is:
$\begin{cases} \mathbb{R} & \longrightarrow \mathbb{R} \\ x & \longmapsto \frac{1}{2b}e^{\frac{|x-\mu|}{b}} \end{cases}$ The robustness comes from the use of absolute value error instead of quadratic error.
If for $i \in [\![1,n]\!]$, $r_i \triangleq y_i - \beta^T x_i$ then NLL has this form: $l(\beta) = \sum_{i=1}^{n}|r_i(\beta)|$
Unfortunately this function is hard to optimize, however we can convert the NLL to a linear objective.
Let set $r_i \triangleq r_i^+ - r_i^-$, with $r_i^+ \geq 0 \wedge r_i^+ \geq 0$. Then the constrained objectives becomes $\min_{r^+, r^-}\sum_{i=1}^{n}(r_i^+ - r_i^-)$
An alternative to using NLL under a Laplace likelihood is to minimize the **Huber Looss** defined as follow: $\begin{cases} \frac{r^2}{2} & \Rightarrow |r| \leq \delta \\ \delta|r| - \frac{\delta^2}{2} & \Rightarrow |r| > \delta \end{cases}$ This is equivalent to l_2 for smaller errors than δ, and is equivalent to l_1 for larger errors.
This function is differentiable everywhere considering that $r \neq\Rightarrow \frac{d}{dr}|r| = sign(r)$, also this function is continuous since the gradients of the 2 parts of the functcion match at $r = \pm\delta$ Consequently optimizing the Huber loss is much faster than using teh Laplace likelihood.

1.2.3 Rigde Regression

Likelihood	Prior	Name
Gaussian	Uniform	Least Squares
Gaussian	Gaussian	Ridge
Gaussian	Laplace	Lasso
Laplace	Uniform	Robust Regression
Student	Uniform	Robust Regression

The likelihood refers to $p(y|\boldsymbol{x}, \boldsymbol{\beta}, \boldsymbol{\sigma^2})$ and the prior refer to $p(\boldsymbol{\beta})$

Purpose The MLE can overfit as it is picking the parameter values that are the best for modleing the training data, however if the data is noisy such parameters result in complex functions.

Assumption

Functionning Having a too complex model will perfectly fit the training data but the parameters will signifciantly change if we fit on another data set. We can then encourage the parameters to be small by using a prior zero-mean value:
$p(\boldsymbol{\beta}) = \prod_{j=1}^{p} \mathcal{N}(\beta_j|0, \tau^2)$

To avoid to create

Limitation

1.3 Classification

To predict qualitative response is know as *classifying*

1.3.1 Why not linear regression ?

In general there is no natural way to convert a qualitative response variable with more than 2 levels into the quantitative response that is ready for linear regression.

1.3.2 Logistic regression

The logistic model *How should we model the relationship between* $\mathbb{P}_{\{X\}}(\{Y = 1\})$ *and* X*?*

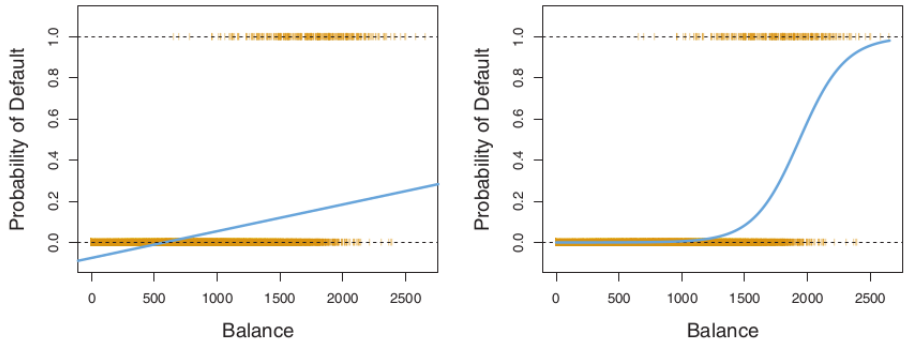


Figure 1.9: Left:Estimated probability of default using linear regression. Some of estimated probability are negative! The orange ticks indicate the 0/1 values coded for default (NO/YES)
Right: Predicted probabilities of default using logistic regression.
All probabilities lie between 0 and 1.

The logistic model

Assumptions

1. Logistic regerssion must be binary or ordinal.
2. Mutual independence of observations.
3. No collinearity
4. Linearity of qualitative independent variables and log odds.

Requirement

1. Logistic regression requires quite large sample size.

Formula The logistic regression:

$$\boxed{p(X) = \frac{e^{\beta_0+\beta_1 X}}{1+e^{\beta_0+\beta_1 X}}}$$

$$\boxed{\log\left(\frac{p(X)}{1-p(X)}\right) = \beta_0 + \beta_1 X}$$

The $\dfrac{p(X)}{1-p(X)}$ quantity is called *odds*.

Estimating the Regression Coefficients To fit a logistic model we use the maximum likelihood method rather that the least squares method (which is a specific case of the former method). Then we seek estimates $\widehat{\beta_0}$ and $\widehat{\beta_1}$ such that the predicted probability $\widehat{p}(x_i)$ of default for each individual corresponds closely as possible to the individual default status.

$$\mathcal{L}\left(\widehat{\beta_0}, \widehat{\beta_1}\right) = \prod_{i=1}^{n} p(x_i)^{y_i} \left(1 - p(x_i)\right)^{1-y_i}$$

$\widehat{\beta_0}$ and $\widehat{\beta_1}$ are chosen to maximize this likelihood function.

The *z-statistics* plays the same role as *t-statistic* in the linear regression output, $z = \frac{\widehat{\beta_1}}{SE\left(\widehat{\beta_1}\right)}$.
The estimated intercept is typically not of interest, its main purpose is to adjust the average fitted probabilities to the proportion of ones in the data.

Marketing Plan $\begin{cases} \mathbb{P}_{\{student=YES\}}\left(\left\{\widehat{default} = YES\right\}\right) = \frac{e^{-3.5041+0.4049\times 1}}{1+e^{-3.5041+0.4049\times 1}} = 0.0431 \\ \mathbb{P}_{\{student=NO\}}\left(\left\{\widehat{default} = YES\right\}\right) = \frac{e^{-3.5041+0.4049\times 0}}{1+e^{-3.5041+0.4049\times 0}} = 0.0292 \end{cases}$
This indicates that students tend to have higher default probabilities than non-student.

Multiple Logistic Regression

$$log\left(\frac{p\left(X\right)}{1-p\left(X\right)}\right) = \beta_0 + \sum_{j=1}^{p} \beta_j X_j$$

Logistic regression for p>2 response classes We use *discriminant analysis*

Fitting Logistic Regression Models Logistic regression models are usually fit by maximum of likelihood, using the conditional likelihood of G given X. The log-likelihood for N observations is:

$$l(\theta) = \sum_{i=1}^{N} log\left(p_g(x_{i;\theta})\right)$$

where $p_g(x_i;\theta) = \mathbb{P}_{\{X=x_i\}}\left(\{G = g;\theta\}\right)$.
It is convenient to code the two-class g_i via a 0/1 response y_i, where $y_i = 1$ when $g_i = 1$, and $y_i = 0$ when $g_2 = 2$. Let $p_1(x;\theta) = p(x;\theta)$, and $p_2(x;\theta) = 1 - p(x;\theta)$. Then the log-likelihood can be written:

$$\begin{aligned} l(\beta) =& \sum_{i=1}^{N} \left\{y_i \log\left(p(x_i;\beta)\right) + \left(1 - y_i\right) \log\left(1 - p(x_i;\beta)\right)\right\} \\ =& \sum_{i=1}^{N} \left\{y_i \beta^T x_i - \log\left(1 + e^{\beta^T x_i}\right)\right\} \end{aligned}$$

To maximize the log-likelihood, we set its derivatives to zero:

$$\frac{\partial l(\beta)}{\partial \beta} = \sum_{i=1}^{N} x_i \left(y_i - p(x_i;\beta)\right) = 0$$

which are $p+1$ equations nonlinear in β. To solve the score, we use the Newton-Raphson algorithm, which requires the second-derivative or Hessian matrix:

$$\frac{\partial^2 l(\beta)}{\partial\beta\partial\beta^T} = \sum_{i=1}^{N} x_i x_i^T p(x_i;\beta)(1-p(x_i;\beta))$$

The aim of the Newton-Raphson algorithm is to find the roots of a given real values function, here $\frac{\partial l(\beta)}{\partial\beta}$ Starting with β^{old}, a single Newton update is :

$$\beta^{new} = \beta^{old} - \left(\frac{\partial^2 l(\beta)}{\partial\beta\partial\beta^T}\right)^{-1} \frac{\partial l(\beta)}{\partial\beta}$$

where the derivatives are evaluated atβ^{old}

Let $\boldsymbol{p}$ denote the vector of fitted probabilities with i^{th} element $p(x_i;\beta^{old})$ and $\boldsymbol{W}$ a $N \times N$ diagonal matrix of weights with i^{th} diagonal element $p(x_i;\beta^{old})(1 - p(x_i;\beta^{old}))$. Then we have:

$$\begin{aligned}\frac{\partial l(\beta)}{\partial\beta} =& \boldsymbol{X}^T(\boldsymbol{y}-\boldsymbol{p})\\ \frac{\partial^2 l(\beta)}{\partial\beta\partial\beta^T} =& -\boldsymbol{X}^T(\boldsymbol{W}-\boldsymbol{X})\end{aligned}$$

The Newton step is thus:

$$\begin{aligned}\beta^{new} =& \beta^{old} + \left(\boldsymbol{X}^T\boldsymbol{W}\boldsymbol{X}\right)^{-1}\boldsymbol{X}^T(\boldsymbol{y}-\boldsymbol{p})\\ =& \left(\boldsymbol{X}^T\boldsymbol{W}\boldsymbol{X}\right)^{-1}\boldsymbol{X}^T\boldsymbol{W}\left(\boldsymbol{X}\beta^{old} + \boldsymbol{W}^{-1}(\boldsymbol{y}-\boldsymbol{p})\right)\\ =& \left(\boldsymbol{X}^T\boldsymbol{W}\boldsymbol{X}\right)^{-1}\boldsymbol{X}^T\boldsymbol{W}\boldsymbol{z}\end{aligned}$$

In the second and third line we have re-expressed the Newton step as a weighted least squares step with the response: $bmz = \boldsymbol{X}\beta^{old} + \boldsymbol{W}^{-1}(\boldsymbol{y}-\boldsymbol{p})$ sometimes known as the **adjusted response**.
This algorithm is referred to as **Iteratively Reweighted Least Squares (IRLS)** since each iteration solves the weighted least squares problem:

$$\beta^{new} \leftarrow \min_{\beta} (\boldsymbol{z}-\boldsymbol{X}\beta)^T \boldsymbol{W}(\boldsymbol{z}-\boldsymbol{X}\beta)$$

It seems that $\beta = 0$ is a good starting value for the iterative procedure although convergence is never guaranteed. Typically the algorithm does converge, since the log-likelihood is concave, but overshooting can occur. Logistic regression models are used mostly as a data analysis and inference tool, where the goal is to understand the role of the input variables in explaining outcome.
Python code

```
import pandas as pd
import sklearn
from sklearn.linear_model import LogisticRegression

y, X = df.iloc[:, 0], df.iloc[:, 1:]
clf = LogisticRegression(random_state=0)
clf.fit(X, y)
print(clf.score(X, y))
```

R code

```
model.logistic <- glm(y ~ ., data=df, family=binomial)
print(summary(model.logistic))
```

Quadratic Approximations and Inference The maximum-likelihood parameter estimates $\hat{\beta}$ satisfy a self-consistency relationship: they are the coefficients of a weighted least squares fit, where the responses are:

$$z_i = x_i^T \hat{\beta} + \frac{(y_i - \hat{p}_i)}{\hat{p}_i(1_i - \hat{p}_i)}$$

- The weighted residual sum-of-squares is the familiar Pearson χ^2 statistic:

$$\sum_{i=1}^{N} \frac{(y_i - \hat{p}_i)}{\hat{p}_i(1 - \hat{p}_i)}$$

  a quadratic approximation of the deviance.

- Asymptotic likelihood theory says that if the model is correct, then $\hat{\beta}$ is consistent.

- A central limit theorem then shows that the distribution of $\hat{\beta} \hookrightarrow \mathcal{N}\left(\beta, \left(\boldsymbol{X}^T \boldsymbol{W} \boldsymbol{X}\right)^{-1}\right)$

- Popular shortcuts are the Rao Score test which tests for inclusion of a term, and the Wald test which can be used to test exclusion of a term.

1.3.3 Linear discriminant analysis

Why do we need another method that logistic regression in the case of several response classes?

- When classes are well-separated, the parameter estimates for the logistic regression model are unstable

- If n is small and the distribution of the predictors is approximately normal in each classe, the *linear discriminant model* is more stable than the logistic.

- Linear discriminant analysis is popular when we have more than 2 response classes.

Using Bayes Theorem for classification Let π_k represents the prior probability that a randomly chosen observation comes from the k^{th} class. Let f_k such that $f_k(x) \equiv \mathbb{P}_{\{X=x\}}(\{Y = k\})$ the density function of X for an observation that comes from k^{th} class:

$$\mathbb{P}_{\{X=x\}}(\{Y = k\}) = \frac{\pi_k f_k(x)}{\sum_{j=1}^{K} \pi_j f_j(x)}$$

Linear Discriminant Analysis for p=1 Aims:

1. To obtain an estimate for $f_k(x)$ that we can plug into
$$\mathbb{P}_{\{X=x\}}(\{Y=k\}) = \frac{\pi_k f_k(x)}{\sum_{j=1}^{K} \pi_j f_j(x)}$$
2. To classify an observation to the class for which $p_k(x)$ is greatest

Assumption: $f_k(x)$ is *Gaussian* that is $f_k(x) = \frac{1}{\sqrt{2\pi\sigma_k}} e^{-\frac{1}{2\sigma_k^2}(x-\mu_k)^2}$

Knowing that $\sigma_1^2 = \cdots = \sigma_K^2$ and taking the log of $p_k(x) = \frac{\pi_k \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{1}{2\sigma^2}(x-\mu_k)^2}}{\sum_{j=1}^{K} \pi_j \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{1}{2\sigma^2}(x-\mu_j)^2}}$

It is equivalent to assigning the observation to the classer for which:

$\delta_k(x) = x\frac{\mu_k}{\sigma^2} - \frac{\mu_k^2}{2\sigma^2} + ln(\pi_k)$ is the largest.

The *linear discriminant analysis* (LDA) method approximates the Bayes classifier by plugging estimates for π_k, μ_k and σ^2:

$$\begin{cases} \hat{\pi}_k = \frac{n_k}{n} \\ \hat{\mu}_k = \frac{1}{n_k} \sum_{i:y_i=k} x_i \\ \hat{\sigma}^2 = \frac{1}{n-K} \sum_{k=1}^{K} \sum_{i:y_i=k} (x_i - \hat{\mu}_k)^2 \end{cases}$$

Linear Discriminant Analysis for p>1

Assumptions

1. Multivariate normality: independent variables are normal for each level of the grouping variable.
2. Homoscedasticity: variances among group variables are the same accross levels of predictors.
3. Non-colinearity
4. Observation are independent

Formulas Now we assume that $X = (X_i)_{1\leq i\leq n} \hookrightarrow \mathcal{N}(\mu, \Sigma)$ Here $\mathbb{E}(X) = \mu = \begin{pmatrix} \mu_1 \\ . \\ . \\ . \\ \mu_p \end{pmatrix}$ and $\Sigma = \begin{pmatrix} Cov(X_1,X_1) & \cdots & Cov(X_1,X_j) & \cdots & Cov(X_1,X_n) \\ & & \cdots & & \\ & & \cdots & & \\ & & \cdots & & \\ Cov(X_i,X_1) & \cdots & Cov(X_i,X_j) & \cdots & Cov(X_i,X_n) \\ & & \cdots & & \\ & & \cdots & & \\ & & \cdots & & \\ Cov(X_n,X_1) & \cdots & Cov(X_n,X_j) & \cdots & Cov(X_n,X_n) \end{pmatrix}$

and $f(x) = \frac{1}{(2\pi)^{\frac{p}{2}}|\Sigma|^{\frac{1}{2}}} exp\left(\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right)$

then

$$\delta_k(x) = x^T\Sigma^{-1}\mu_k - \frac{1}{2}\mu_k^T\Sigma^{-1}\mu_k + ln(\pi_k)$$

Python code

```
import sklearn
from sklearn.discriminant_analysis import
    LinearDiscriminantAnalysis

clf = LinearDiscriminantAnalysis()
clf.fit(X, y)
print(clf.score(X, y))
```

R code

```
library(MASS)

model.lda <- lda(Direction ~ ., data=df)
model.lda
```

Quadratic Discriminant Analysis unlike LDA, QDA assumes that each class has its own covariance matrix. That is for an observation from the k^{th} class $X \hookrightarrow N(\mu_k, \Sigma^k)$

$$\begin{aligned}\delta_k(x) &= -\frac{1}{2}(x-\mu_k)^T\Sigma_k^{-1}(x-\mu_k) - \frac{1}{2}\ln|\Sigma_k| + \ln(\pi_k) \\ &= -\frac{1}{2}x^T\Sigma_k^{-1}x + x^T\Sigma_k^{-1}\mu_k - \frac{1}{2}\mu_k^T\Sigma_k^{-1}\mu_k - \frac{1}{2}\ln|\Sigma_k| + \ln(\pi_k)\end{aligned}$$

Python code

```
import sklearn
from sklearn.discriminant_analysis import
    QuadraticDiscriminantAnalysis

clf = QuadrasticDiscriminantAnalysis()
clf.fit(X, y)
print(clf.score(X, y))
```

R code

```
library(MASS)

model.lda <- lda(y ~ ., data=df)
model.lda
```

Regularized Discriminant Analysis The regularize covariance matrices have the form:

$$\boxed{\hat{\mathbf{\Sigma}}_k(\lambda) = \lambda\hat{\mathbf{\Sigma}}_k + (1-\lambda)\hat{\mathbf{\Sigma}}}$$

where $\hat{\mathbf{\Sigma}}$ is the pooled covariance matrix as used in LDA. Here $\lambda \in [0,1]$ allows a continuum of models between LDA and QDA, λ can be choosed by cross-validation.
Similar modifications allow $\hat{\mathbf{\Sigma}}$ itself to be shrunk toward the scalar covariance:

$$\hat{\mathbf{\Sigma}}(\gamma) = \gamma\hat{\mathbf{\Sigma}} + (1-\gamma)\hat{\sigma}^2\boldsymbol{I}$$

for $\gamma \in [0,1]$ *R code*

```
library(klaR)
model.rda <- rda(y ~ ., data=df, gamma=0.05, lambda=0.2)
```

With rda we have : $\hat{\Sigma}_k(\lambda,\gamma) = (1-\gamma)\hat{\Sigma}_k(\lambda) + \gamma\frac{1}{p}tr\left(\hat{\Sigma}_k(\lambda)\right) I$

$(\gamma = 0, \lambda = 0)$: QDA - individual covariance for each group

$(\gamma = 0, \lambda = 1)$: LDA - a common covariance matrix

$(\gamma = 1, \lambda = 0)$: Conditional independent variables - similar to Naive Bayes, but variable variance whithin group (main diagonal elements) are all equal.

$(\gamma = 1, \lambda = 1)$: Classification using euclidian distance - as in previous case, but variances are the same for all groups. Objects are assigned to group with nearest mean.

Computations for LDA The computations of LDA and QDA are simplified by diagonalizing $\hat{\mathbf{\Sigma}}$ or $\hat{\mathbf{\Sigma}}_k$ for the latter, suppose we compute the eigendecomposistion for each $\hat{\mathbf{\Sigma}}_k = \boldsymbol{U}_k\boldsymbol{D}_k\boldsymbol{U}_k^T$ where $\boldsymbol{U}_k$ is $p \times p$ orthonormal and $\boldsymbol{D}_k$ a diagonal matrix of positive eigenvalues d_{kl}.

- $(x-\hat{\mu}_k)^T\hat{\mathbf{\Sigma}}_k^{-1}(x-\hat{\mu}_k) = \left[\boldsymbol{U}_k^T(x-\hat{\mu}_k)\right]^T \boldsymbol{D}_k^{-1}\left[\boldsymbol{U}_k^T(x-\hat{\mu}_k)\right]$
- $\log\left(\hat{\mathbf{\Sigma}}_k\right) = \sum_l \log(d_{kl})$

LDA classifier can be implemented by the following pair of steps:

- **Sphere** the data with respect to the common covariance estimate $\hat{\mathbf{\Sigma}}$: $X^* \leftarrow \boldsymbol{D}^{\frac{1}{2}}\boldsymbol{U}^T X$ where $\hat{\mathbf{\Sigma}} = \boldsymbol{U}\boldsymbol{D}\boldsymbol{U}^T$. The common covariance estimate of X^* will now be the identity.
- Classify to the closet class centroid in the transformed space, modulo the effect of the class prior probabilities π_k.

Reduced-Rank Discriminant Analysis The K centroids in $p-$dimensional input space lie in an affine subspace of dimension $\leq K-1$, and if p is much larger than K this will be a considerable drop in dimension.
Moreover in locating the closet centroid we can ignore distances orthogonal to this subspace, since they will contribute equally to each class. Thus we might just as well project the X^* onto centroid-spanning subspace H_{K-1} and make distance comparisons there.

Finding the sequences of optimal subspaces for LDA involves the following steps:

- compute the $K \times p$ matrix of class centroids $\boldsymbol{M}$ and the common covariance matrix $\boldsymbol{W}$
- compute $\boldsymbol{M}^* = \boldsymbol{M}\boldsymbol{W}^{-\frac{1}{2}}$ using the eigen-decomposition of $\boldsymbol{W}$
- compute $\boldsymbol{B}^*$, the covariance matrix of $\boldsymbol{M}^*$ and its eigen-decomposition $\boldsymbol{B}^* = \boldsymbol{V}^*\boldsymbol{D}_B(\boldsymbol{V}^*)^T$. The columns v_l^* of $\boldsymbol{V}^*$ in sequence from first to last define the coordinates of the optimal subspaces.

Combining all these operations the l^{th} discriminant variable is given by $Z_l = v_l^T X$ with $v_l = \boldsymbol{W}^{-\frac{1}{2}} v_l^*$

Fisher arrived at this decomposition via a different route, without referring to Gaussian distribution at all.

Find the linear combination $Z = a^T X$ such that the between class variance is maximized relative to the within-class variance.

The between-class variance of Z is a $a^T\boldsymbol{B}a$ and the within-class variance $a^T\boldsymbol{W}a$, where $\boldsymbol{W}$ is defined earlier, and $\boldsymbol{B}$ is the covariance matrix of the class centroid matrix $\boldsymbol{M}$.
Fisher's problem therefore amounts to maximizing the Rayleigh quotient:

$$\max_a \frac{a^T\boldsymbol{B}a}{a^T\boldsymbol{W}a}$$

or equivalently:

$$\max_a a^T\boldsymbol{B}a \text{ subject to } a^T\boldsymbol{W}a = 1$$

This is a generalized eigenvalue problem, with a given by the largest eigenvalue of $\boldsymbol{W}^{-1}\boldsymbol{B}$.

- Gaussian classification with common covariances leads to linear decision boundaries
- One can confine the data to the subspace spanned by the centroids in the sphered space.
- This subspace can be further decomposed into successively optimal subspaces in term of centroid separation. This decomposition is identical to the decomposition due to Fisher.

1.3.4 A comparison of classification methods

Logistic regression, LDA, QDA and KNN

Logistic regression VS LDA The only difference between the 2 approaches lies in fact that β_0 and β_1 are estimated using maximum likelihood, whereas c_0 and c_1 are computed using the estimated mean and variance from a normal distribution.

For $p = 1$ *predictor* we have :

$$ln\left(\frac{p_1(x)}{1-p_1(x)}\right) = c_0 + c_1 x \text{ LDA}$$
$$ln\left(\frac{p_1(x)}{1-p_1(x)}\right) = \beta_0 + \beta_1 x \text{ Logistic regression}$$

It is generally felt that logistic regression is a safer, more robust bet than the LDA model, relying on fewer assumptions.

KNN KNN is a completely non-parametric approach: no assumptions are made about the shape of the decision boundary. Therefore, we can expect this approach to dominate LDA and Logistic regression when th decision boundary is highly non-linear.

QDA It is a compromise between the non-parametric KNN method and the linear LDA and Logistic regression approaches.

1.3.5 Separating Hyperplanes

Separating hyperplane classifiers are procedures that construct linear decision boundaries that explicitly try to separate the data into different classes as well as possible.
Classifier such as $\left\{x : \hat{\beta}_0 + \hat{\beta}_1 x_1 + \hat{\beta}_2 x_2 = 0\right\}$ that compute a linear combination of the input features and return the sign, were called *perceptrons* in the engineering literature in the late $1950's$

Perceptrons set te foundations for the neural network models.
For a surface L defined by the equation: $f(x) = \beta_0 + \beta^T x = 0$
Some properties:

1. $\forall (x_1, x_2) \in L^2,\ \beta^T(x_1 - x_2) = 0$ and hence $\beta^* = \frac{\beta}{\|\beta\|}$ is the vector normal to the surface L

2. $\forall x_0 \in L, \beta^T x_0 = -\beta_0$

3. The signed distance of any point x to L is given by:

$$\begin{aligned}\beta^{*T}(x - x_0) =& \frac{1}{\|\beta\|}\left(\beta^T x + \beta_0\right) \\ =& \frac{1}{\|f'(x)\|} f(x)\end{aligned}$$

Hence $f(x)$ is proportional to the signed distance from x to the hyperplane defined by $f(x) = 0$

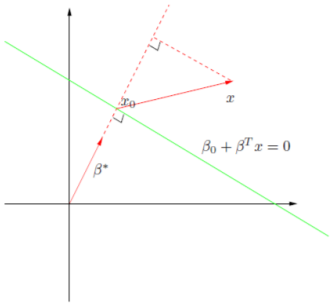


Figure 1.10: The linear algebra of a hyperplane (affine set)

Rosenblatt's Perceptron Learning Algorithm It tries to find a separating hyperplane by minimizing the distance of misclassified points to the decision boundary. The goal is to minimize:

$$D(\beta, \beta_0) = \sum_{i \in \mathcal{M}} y_i \left(x_i^T \beta + \beta_0\right)$$

where $\mathcal{M}$ indexes the set of misclassified points.

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

The algorithm in fact uses *stochastic gradient descent* to minimize this piecewise linear criterion.
The misclassified observations are visited in some sequence, and parameters β are updated via:

$$\begin{pmatrix} \beta \\ \beta_0 \end{pmatrix} \leftarrow \begin{pmatrix} \beta \\ \beta_0 \end{pmatrix} + \rho \begin{pmatrix} y_i x_i \\ y_i \end{pmatrix}$$

ρ is the learning rate.

Optimal Separating Hyperplanes The *optimal separating hyperplanes* separates the 2 classes and maximizes the distance to the closet point from either class. Consider the optimization problem:

$$\max_{\beta, \beta_0, \|\beta\|=1} M$$
$$\text{subject to } y_i \left(x_i^T \beta + \beta_0\right) \geq M, i \in [\![1, N]\!]$$

The set of conditions ensure that all the points are at least a signed distance M from the decision boundary defined by β and β_0

1.4 Resampling Methods

1.4.1 Cross-validation

The Validation Set Approach It involves randomly dividing the available set of observations into 2 parts:

1. *training set*
2. *validation set*

Pitfall:

- the validation estimate of the test error rate can be highly variable.
- Only a subset of the observations are used to fit the model.

```
import sklearn
from sklearn.model_selectioon import LeaveOneOut,\
KFold
```

Leave-One-Out Cross-Validation Instead of creating 2 subsets of comparable size, a single observations (x_1, y_1) is used for the validation set and the remaining observations $\{x_i, y_i\}_{2 \leq i \leq n}$ make up the training set.
The LOOCV estimate for the test MSE is the average of these n test error estimates:

$$CV_n = \frac{1}{n} \sum_{i=1}^{n} MSE_i$$

LOOCV approach tends not to overestimate the test error rate as much as the validation set approach does.
With least squares linear or polynomial regression, an amazing shortcut makes the cost of LOOCV the same as that of a single model fit!

$$CV_n = \frac{1}{n} \sum_{i=1}^{n} \left(\frac{y_i - \overline{y}_i}{1 - h_i} \right)^2$$

h_i is the leverage (that reflects the amount that an observations influence its own fit).
LOOCV is a very general method, and can be used with any kind of predictive modeling.

```
X = [1, 2, 3, 4]
loo = LeaveOneOut()
for train, test in loo.split(X):
    print(train, test)
```

k-Fold Cross-Validation This approach involves randomly dividing the set of observations into k groups (folds), of approximately equal size. The first fold is treated as a validation set, and the method is fit on the remaining $k-1$ folds.

$$CV_n = \frac{1}{k}\sum_{i=1}^{k} MSE_i$$

When we examine real data, we do not know the True test MSE, and so it is difficult to determine the accuracy of the cross-validation estimate.

When we perform cross-validation, our goal might to be determine how well a given statistical learning procedure can be expected to perform on independent data; in this case, the actual estimate of the test MSE is of interest. But other times we are interested only in the location of the *minimum point in the estimated test* MSE curve.

Despite the fact that they sometimes underestimate the true test MSE, all of the CV curves come close to identifying, the flexibility level corresponding to the smallest test MSE.

```
X = [1, 2, 3, 4]
kf = KFold(n_splits=2)
for train, test in kf.split(X):
    print(train, test)
```

Biais-Variance Trade-Off for k-Fold Cross-Validation Validation set approach can lead to overestimates of the test error rate, since the training set used to fit the response contains only half the observations of the entire data set. Likewise LOOCV will give approximately unbiased estimates of the test error.

And performing k-fold CV for, say, $k = 5$ or $k = 10$ will lead to an intermediate level of biais, since each training set contains $\frac{(k-1)n}{k}$ observations fewer than in the LOOCV approach, but substantially more than in the validation set approach.

It turns out that LOOCV has higher variance than does k-fold CV with k $<$ n. Why ? We are averaging the outputs of n fitted models, each of which is trained on an almost identical set of observations; therefore, these outputs are highly (positively) correlated with each other. In contrast, when we perform k-fold CV with $k < n$ we are averaging the outputs of k fitted models that are somewhat less correlated with each other, since the overlap between the training sets in each model is smaller.

Since the mean of many highly correlated quantities has high variance than does the mean of many quantities that are not as highly correlated, the test error estimate resulting from LOOCV tends to have higher variance than does the test error estimate resulting from k-fold CV.

Cross-Validation on Classification Problems Instead to use the MSE to quantify test error, we rather use the number of misclassified observations.

$$CV_n = \frac{1}{n}\sum_{i=1}^{n} Err_i$$

where $Err_i = I(y_i \neq \hat{y}_i)$

1.4.2 The Bootstrap

It can be used to quantify the uncertainity associated with a given estimator or statistical learning method.

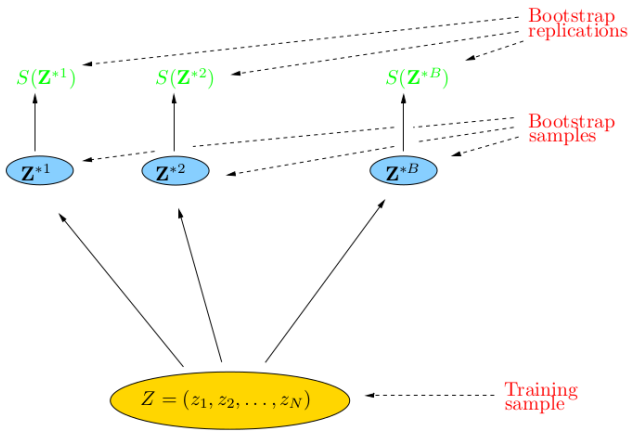


Figure 1.11: Schematic of the bootstrap process

The basic idea is to randomly draw datasets with replacement from the training data, each sample the same size as the original training set. This is done B times, producing B bootstrap datasets. $S(\boldsymbol{Z})$ is any quantity computed from the data $\boldsymbol{Z}$. From the bootstrap sampling we can estimate any aspect of the distribution of $S(\boldsymbol{Z})$, for example its variance:

$$\mathbb{V}\left(\hat{S(\boldsymbol{Z})}\right) = \frac{1}{B-1}\sum_{b=1}^{B}\left(S(\boldsymbol{Z}^{*b}) - \overline{S}^{*}\right)^2$$

with $\overline{S}^{*} = \frac{1}{B}\sum_{b=1}^{B} S(\boldsymbol{Z}^{*b})$

1.5 Linear model selection and regularization

1.5.1 Subset selection

Best Subset Selection We fit a separate least squares regression for each possible combination of the p predictors.

Algorithm

1. Let $\mathcal{M}_0$ denote the *null model*, which contains no predictors. This model model simply predicts the sample mean for each observations.
2. For $k \in [\![1, p]\!]$:
   a Fit all $\binom{p}{k}$ models that contain exactly k predictors
   b Pick the best among these $\binom{p}{k}$ models, and call it $\mathcal{M}_k$
   Here best is defined as having the small RSS or equivalently largest R^2
3. Select a single best model from among $(\mathcal{M}_i)_{0 \leq i \leq p}$ using cross-validation prediction error, $C_p(AIC), BIC$, or adjusted R^2

This task must be performed with care, because the RSS of these $p+1$ models decreases monotonically, and the R^2 increases monotonically as the number of features included in the models increases.

This problem is that a low RSS or a high R^2 indicates a model with a low *training error*, whereas we wish to choose a model that has a model that has a low test error.

Instead use RSS, we use the *deviance*, a measure that plays the role of RSS for a broader class class of models. The deviance is negative 2 times the maximized log-likelihood; the smaller the deviance the better the fit.

R code:

```
library(dplyr)

df <- df %>%
          select_if(is.numeric) # selecting only numerical
    columns
names(df) <- gsub(names(df), '-', '_') # replace all '-' in
    headers
regfit.full <- regsubsets(y ~ ., df, nvmax=dim(df)[2]-1)  #
    best subset
# -1 because response y is excluded from data frame
reg.summary <- summary(regfit.full)

# Plotting several scores
par(mfrow=c(2, 2)) # making a 2 X 2 table of graphs
plot(reg.summary$rss, xlab="Number of Variables", ylab="RSS"
    , type="l")
min.rss <- which.min(reg.summary$rss)
points(min.rss, reg.summary$rss[min.rss], col='red')
plot(reg.summary$adjr2, xlab="Number of Variables", ylab="
    Adjusted RSq", type="l")
max.adjr2 <- which.max(reg.summary$adjr2)
points(max.adjr2, reg.summary$adjr2[max.adjr2], col='red')
plot(reg.summary$cp, xlab="Number of Variables", ylab="Cp",
    type="l")
min.cp <- which.min(reg.summary$cp)
points(min.cp, reg.summary$cp[min.cp], col="red")
```

```
plot(reg.summary$bic, xlab="Number of Variables", ylab="BIC"
    , type="l")
min.bic <- which.min(reg.summary$bic)
points(min.bic, reg.summary$bic[min.bic], col="red") #$

df_mod <- df[, reg.summary$which[max.adjr2, -1]] # -1 to
    exclude intercept
```

Stepwise Selection

Forward Stepwise Selection Algorithm

1. Let $\mathcal{M}_0$ denote the *null* model, which contains no predictors.
2. For $k \in [\![0, p-1]\!]$
   a Consider all $p-k$ models that argument the predictors in $\mathcal{M}_k$ with one additional predictor.
   b Choose the *best* among these $p-k$ models, and call it $\mathcal{M}_{k+1}$. Here *best* is defined as having smallest RSS or highest R^2
3. Select a single best model from among $(\mathcal{M}_i)_{1\leq i\leq 0} p$ using cross-validated prediction error, $C_p(AIC), BIC$, or adjusted R^2

Unlike best subset selection, which involved fitting 2^p models, forward stepwise selection involves fitting one null model, along with $p-k$ models in the kth iteration, for $k \in [\![0, p-1]\!]$.

This amounts to a total of $1 + \sum_{k=0}^{p-1}(p-k) = 1 + \frac{p(p+1)}{2}$ models Python code:

```
import mlxtend # library for model selection
from mlxtend.feature_selection import
    SequentialFeatureSelector
import numpy as np
import matplotlib.pyplot as plt
import sklearn
import sklearn import linear_model

y, X = df.iloc[:, 0], df.iloc[:, 1:]
reg = linear_model.LinearRegression()
sfsl = SFS(reg, # sickit-learn classifier/regressor
    k_features=X.shape[1], # Number of features to select
    forward=True, # Forward if True otherwise False
    floating=False, # Adds a conditional exclusion/inclusion
     if True
    verbose=2, # Level of verbosity to use in logging [0,2]
    scoring='r2', # 'accuracy' for classifiers, 'r2' for
    regressors
    cv=5 # Stratified k-fold for classifier
    )
sfsl.model = sfsl.fit(X, y) # processing selection model
dict_sfsl = sfs1_model.subsets_ # dictionary containing
    scores depending on features
```

```
20
21 plt.figure()
22 x = np.array(list(dict_sfsl))
23 y = np.array([dict_sfsl[k]['avg_score'] for k in list(
       dict_sfsl)])
24 plt.plot(x, y)
25 ind_min_y = np.where(y==y.max())[0][0]
26 plt.plot(ind_min_y, dict_sfsl[ind_min_y]['avg_score'], 'ro')
27 plt.xlabel('Number of features')
28 plt.ylabel(r'$R^{2}$')
29 plt.titel('Score Evolution')
30 plt.show()
31
32 df_mod = df.loc[:, dict_sfsl[ind_min_y]['feature_names']].
       copy()
```

R code:

```
1 regfit.fwd = regsubsets(y ~ ., data=df, nvmax=dim(df)[2],
      method='forward')
```

Backward Stepwise Selection Algorithm

1. Let $\mathcal{M}_p$ denote the *full* model, which contains all p predictors.
2. For $k \in [\![p-1, 0]\!]$
   a Consider all k models that contain all but one of the predictors in $\mathcal{M}_k$ (for a total of) $k-1$ predictors.
   b Choose the *best* among these k models, and call it $\mathcal{M}_{k-1}$.
   Here *best* is defined as having smallest RSS or highest R^2
3. Select a single best model from among $(\mathcal{M}_i)_{0\leq i\leq p}$ using cross-validated prediction error, $C_p(AIC), BIC$, or adjusted R^2

Hybrid approach Such an approach attempts to more closely mimic best subset selection while retaining the computational advantages of forward and backward stepwise selection.

Optimism of the Training Error Rate Given a training set $\mathcal{T} = \{(x_i, y_i : i \in [\![1, N]\!])\}$, the generalization error of a model $\hat{f}$ is: $Err_{\mathcal{T}} = \mathbb{E}_{X^0,Y^0}\left(L(Y^0, \hat{f}(X^0))|\mathcal{T}\right)$
Typically the training error:

$$\overline{err} = \frac{1}{n}\sum_{i=1}^{N} L\left(y_i, \hat{f}(x_i)\right)$$

The *in-sample* error:

$$Err_{in} = \frac{1}{N}\sum_{i=1}^{N} \mathbb{E}_{Y^0}\left(L\left(Y_i^0, \hat{f}(x_i)\right)|\mathcal{T}\right)$$

The Y^0 notation indicates that we observe N new response values at each of the training point x_i with $i \in [\![1, N]\!]$. We define the ***optimism*** as :

$$\begin{cases} op \equiv Err_{in} - \overline{err} \\ \omega \equiv E_y(op) \end{cases}$$

We can usually estimate only the expected error ω rather than op, in the same way that we can estimate the expected error Err rather than the conditional error $Err_{\mathcal{T}}$
For squared error, one can show quite generally that:

$$\omega = \frac{2}{N}\sum_{i=1}^{N} Cov(\hat{y}_i, y_i)$$

In summary we have the important relation:

$$\mathbb{E}_{\boldsymbol{y}}\left(Err_{in}\right) = \mathbb{E}_{\boldsymbol{y}}(\overline{err}) + \frac{2}{N}\sum_{i=1}^{N} Cov(\hat{y}_i, y_i)$$

The effective number of parameters Let $\boldsymbol{S}$ be the hat matrix for linear fitting methods not for only linear regression ($\boldsymbol{H}$) The effective number of parameters is

$$df(\boldsymbol{S}) = trace(\boldsymbol{S})$$

If $\boldsymbol{y}$ arises from an additive-error model $Y = f(X) + \epsilon$ with $\mathbb{V}(\epsilon) = \sigma^2$, then one can show that $\sum_{i=1}^{N} Cov(\hat{y}_i, y_i) = trace(\boldsymbol{S})\sigma^2$ which motivates the more general definition:

$$df(\hat{\boldsymbol{y}}) = \frac{\sum_{i=1}^{N} Cov(\boldsymbol{y}_i, y_i)}{\sigma^2}$$

Choosing the Optimal Model Now rather to choose the model with the smallest RSS and the largest R^2, since these quantities are related to training error, we wish to choose the model with a low test error.
Then we can

1. indirectly estimate test error, by making an *adjustment* to the training error to account for the bias due to overfitting.

2. directly estimate test error, using either a validation set approach or a cross-validation approach.

C_p, AIC, BIC, and adjusted R^2 C_p

For a fitted least squares model containing d predictors, the C_p estimate of test MSE is computed using the equation

$$\begin{cases} C_p = \frac{1}{n}\left(RSS + 2d\hat{\sigma}^2\right) \\ C_p = \overline{err} + 2\frac{d}{N}\hat{\sigma}^2 \end{cases}$$

where $\hat{\sigma}^2$ is an estimate of the variance of the error ϵ associated with each response measurement.
Using this criterion we adjust the training error by a factor proportional to the number of basis function used.

AIC

It is defined for a large class of models fit by maximum likelihood. The *Akaike information criterion* is a similar but more generally applicable but more applicable estimate of Err_{in} when a log-likelihood loss function is used. It relies on a relationship:

$$-2\mathbb{E}\left(\log\left(\mathbb{P}_{\hat{\theta}}(Y)\right)\right) \approx -\frac{2}{N}\mathbb{E}\left(loglik\right) + 2\frac{d}{N}$$

Here $\mathbb{P}_{\hat{\theta}}(Y)$ is a family of densities for Y (containing the "true" density), $\hat{\theta}$ is the maximum-likelihood estimate of θ, and "logik"is the maximized log-likelihood:

$$loglik = \sum_{i=1}^{N} \log\left(\mathbb{P}_{\hat{\theta}}(y_i)\right)$$

Given a set of $f_\alpha(x)$ indexed by a tuning parameter α, denote by $\overline{err}(\alpha)$ and $d(\alpha)$ the training error and number of parameters for each model.

$$\begin{cases} AIC = \frac{1}{n\hat{\sigma}^2}\left(RSS + 2d\hat{\sigma}^2\right) \\ AIC = \overline{err}(\alpha) + 2\frac{d(\alpha)}{N}\hat{\sigma}_\epsilon^2 \end{cases}$$

BIC

It is derived from a Bayesian point of view, like AIC it is applicable in setting where the fitting is carried out by maximization of a log-likelihood:

$$\begin{cases} BIC = \frac{1}{n\hat{\sigma}^2}\left(RSS + \ln(n)d\hat{\sigma}^2\right) \\ BIC = \frac{N}{\sigma^2}\left[\overline{err} + \log(N)\frac{d}{N}\sigma^2\right] \end{cases}$$

Therefore BIC is proportional to AIC, with the factor 2 replaced by $\log(N)$. Assuming $N > e^2 \approx 7.4$, BIC tends to penalize complex models more heavily, giving preference to simpler models in selection.

Adjusted R^2

For a least squares model with d variables it is calculated as:

$$\text{Adjusted } R^2 = 1 - \frac{\frac{RSS}{n-d-1}}{\frac{TSS}{n-1}}$$

Given a family of models, including the true model, the probability that BIC will select the correct model approaches one as the sample size $N \to \infty$. This

is not the case for AIC, which tends to choose models which are too complex as $N \to \infty$.
On the other hand, for finite samples BIC often chooses models that are too simple, because of its penalty on complexity.

Vapnik-Chervonenkis Dimension Altough the effective number of parameters is useful for some non-linear models, it is not fully general. The *Vapnik-Chervonenkis* theory provides such a general measure of complexity and gives associated bounds on the optimism.
The *Vapnik-Chervonenkis* dimension is a way of measuring the complexity of a class of function by assessing how wiggly its members can be.
If we fit N training points using a class of function $\{f(x, \alpha)\}$ (class of functions indexed by a parameter vector α, with $x \in \mathbb{R}^p$) having VC dimensions (that is defined as the largest number of points that can be shattered by members of $\{f(x, \alpha)\}$) noted h. Then with probability at least $1 - \eta$ over training sets:

$$\begin{cases} Err_{\mathcal{T}} \leq \overline{err} + \dfrac{\epsilon}{2}\left(1 + \sqrt{1 + \dfrac{4\overline{err}}{\epsilon}}\right) & \text{(binary classification)} \\ Err_{\mathcal{T}} \leq \dfrac{\overline{err}}{(1 - c\sqrt{\epsilon})_+} & \text{(regression)} \end{cases}$$

where $\epsilon = a_1 \dfrac{h\left[\log\left(a_2 \frac{N}{h}\right) + 1\right] - \log\left(\frac{\eta}{4}\right)}{N}$ and $(a_1, a_2) \in]0, 4] \times]0, 2]$.
Cherkassky and Mulier recommend the value of $c = 1$,

- For regression they suggest $(a_1, a_2) = (1, 1)$
- For classification they suggest $(a_1, a_2) = (4, 2)$

Validation and Cross-Validation We can compute the validation set error or the cross-validation error for each model under consideration, and then select the model for which the resulting estimated test error is smallest.

Bootstrap Methods The bootstrap is a general tool for assessing statistical accuracy.
We denote the training set by $\boldsymbol{Z} = (z_1, \cdots, z_N)$ where $z_i = (x_i, y_i)$. The basic idea is to randomly draw datasets with replacement from the training data, each sample the same size as the original training set. This done B times, producing B bootstrap datasets.

The leave-one-out boostrap estimate of prediction error is defined by:

$$\hat{Err} = \frac{1}{N}\sum_{i=1}^{N} \frac{1}{|C^{-i}|} \sum_{b \in C^{-i}} L\left(y_i, \hat{f}^{*b}(x_i)\right)$$

Here C^{-i} is the set of indices of the bootstrap samples b that do not contain observation i, and $|C^{-i}|$ is the number of such samples. First we define γ to be the *no-information rate* this is the error rate of our prediction rule if the inputs and class labels were independent.

$$\hat{\gamma} = \frac{1}{N^2}\sum_{i=1}^{N}\sum_{j=1}^{N} L\left(y_i, \hat{f}(x_j)\right)$$

The *relative overfitting rate* is defined to be:

$$\hat{R} = \frac{\hat{Err} - \overline{err}}{\hat{\gamma} - \overline{err}}$$

a quantity that ranges from 0 if there is no overfitting to 1 if the overfitting equals the no-information value $(\hat{\gamma} - \overline{err})$.

We conclude that estimation of test error for a particular training set is not easy in general, given just the data from that same training set. Instead cross-validation and related methods may provide reasonable estimate of the *expected* error Err.

1.5.2 Shrinkage methods

Ridge Regression

Definition The ridge regression coefficient estimates $\hat{\beta}^R$ are the values that minimize:

$$\sum_{i=1}^{n}\left(y_i - \beta_0 - \sum_{j=1}^{p}\beta_j x_{ij}\right)^2 + \lambda\sum_{j=1}^{p}\beta_j^2 = RSS + \lambda\sum_{j=1}^{p}\beta_j^2$$

where $\lambda > 0$ is a *turning parameter* to be determined separately.
$RSS(\lambda) = (\boldsymbol{y} - \boldsymbol{X}\beta)^T(y - \boldsymbol{X}\beta) + \lambda\beta^T\beta$

$$\hat{\beta}^{ridge} = (\boldsymbol{X}^T\boldsymbol{X} + \lambda\boldsymbol{I})^{-1}\boldsymbol{X}^T\boldsymbol{y}$$

In the case of orthonormal inputs, the ridge estimates are just a scaled version of the least squares estimates, that is $\hat{\beta}^{ridge} = \frac{1}{1+\lambda}\hat{\beta}$
The Singular Value Decomposition (SVD) of the centered input matrix $\boldsymbol{X}$ gives us some additional insight into the nature of ridge regression: The SVD of the $N \times p$ matrix $\boldsymbol{X}$ has the form:

$$\boldsymbol{X} = \boldsymbol{U}\boldsymbol{D}\boldsymbol{V}^T$$

Here $\boldsymbol{U}$ and $\boldsymbol{V}$ are $N \times p$ and $p \times p$ orthogonal matrices, with the columns of $\boldsymbol{U}$ spanning the columns space of $\boldsymbol{X}$, and the columns of $\boldsymbol{V}$ spanning the row space.
$\boldsymbol{D}$ is a $p \times p$ diagonal matrix, containing singular values of $\boldsymbol{X}$ Using the singular value decomposition:

$$\begin{aligned}\boldsymbol{X}\hat{\beta}^{ls} &= \boldsymbol{X}(\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y} \\ &= \boldsymbol{U}\boldsymbol{U}^T\boldsymbol{y}\end{aligned}$$

For ridge regression:

$$\begin{aligned}\boldsymbol{X}\hat{\beta}^{ridge} &= \boldsymbol{X}(\boldsymbol{X}^T\boldsymbol{X} + \lambda\boldsymbol{I})^{-1}\boldsymbol{X}^T\boldsymbol{y} \\ &= \boldsymbol{U}\boldsymbol{D}(\boldsymbol{D}^2 + \lambda\boldsymbol{I})^{-1}\boldsymbol{D}\boldsymbol{U}^T\boldsymbol{y} \\ &= \sum_{j=1}^{p} u_j \frac{d_j^2}{d_j^2 + \lambda} u_j^T\boldsymbol{y}\end{aligned}$$

Like linear regression, ridge regression computes the coordinates of $\boldsymbol{y}$ with respect to the orthonormal basis $\boldsymbol{U}$, It shrinks these coordinates by $\frac{d_j^2}{d_j^2+\lambda}$, a greater amount of shrinkage is applied to the coordinates of basis vectors with smaller d_j^2
The SVD of the centered matrix X is another way to express the principal components of the variables in $\boldsymbol{X}$. The sample covariance matrix is given by $\boldsymbol{S} = \frac{\boldsymbol{X}^T\boldsymbol{X}}{N}$

$$\boldsymbol{X}\boldsymbol{X}^T = \boldsymbol{V}\boldsymbol{D}^2\boldsymbol{V}^T$$

which is the eigen decomposition of $\boldsymbol{X}^T\boldsymbol{X}$
The eigenvectors v_j are also called the principal components directions of $\boldsymbol{X}$. The first principal component direction v_1 has the property that $z_1 = \boldsymbol{X}v_1$ has the largest sample variance amongst all normalized linear combinations of the columns of $\boldsymbol{X}$:

$$\mathbb{V}(z_1) = \mathbb{V}(\boldsymbol{X}v_1) = \frac{d_1^2}{N}$$

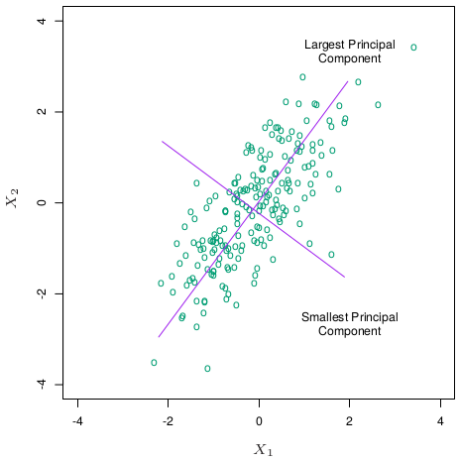


Figure 1.12: The largest principal components is the direction that maximizes the variance of the projected data and the smallest principal component minimizes that variance. Ridge regression projects $\boldsymbol{y}$ onto these components, and then shrinks the coefficients of the low-variance components more than the high-variance components

$$\begin{aligned} df(\lambda) &= tr\left(\boldsymbol{X}(\boldsymbol{X}^T\boldsymbol{X}+\lambda\boldsymbol{I})^{-1}\boldsymbol{X}^T\right) \\ &= tr(\boldsymbol{H}_\lambda) \\ &= \sum_{j=1}^{p}\frac{d_j^2}{d_j^2+\lambda} \end{aligned}$$

This monotone decreasing function of λ is the *effective degrees of freedom* of the ridge regression fit.
Note that $df(\lambda) = p$ when $\lambda = 0$ (no regularization) and $df(\lambda) \to 0$ as $\lambda \to \infty$

Why does Ridge Regression improve over Least Squares? Ridge regression's advantage over least squares is rooted in the *bias-variance trade-off*. As λ increases, the flexibility of the ridge regression fit decreases, leading to decreased variance but increased bias.

```
reg = linear_model.Ridge(alpha=0.5)
reg.fit([[0, 0], [0, 0], [1, 1]], [0, 1, 1])
print(reg.coef_)
```

The Lasso It is a relatively recent alternative to ridge regression.
The Lasso coefficients, $\hat{\beta}_\lambda$ minimize the quantity:

$$\sum_{i=1}^{n}\left(y_i - \beta_0 - \sum_{j=1}^{p}\beta_j x_{ij}\right)^2 + \lambda\sum_{j=1}^{p}|\beta_j| = RSS + \lambda\sum_{j=1}^{p}|\beta_j|$$

Models generated from the lasso are generally much easier to interpret than those produced by ridge regression.

```
reg = linear_model.Lasso(alpha=0.1)
reg.fit([[0, 0], [1, 1]], [0, 1])
print(reg.coef_)
```

Another formulation for Ridge Regression and the Lasso

$$\begin{cases} \min\limits_{(\beta_j)_{0\le j\le p}} \left\{\sum\limits_{i=1}^{n}\left(y_{i-} - \beta_0 - \sum\limits_{j=1}^{p}\beta_j x_{ij}\right)^2\right\} \text{ subject to } \sum\limits_{j=1}^{p}|\beta_j| \le s \text{ Ridge Regression} \\ \min\limits_{(\beta_j)_{0\le j\le p}} \left\{\sum\limits_{i=1}^{n}\left(y_{i-} - \beta_0 - \sum\limits_{j=1}^{p}\beta_j x_{ij}\right)^2\right\} \text{ subject to } \sum\limits_{j=1}^{p}\beta_j^2 \le s \text{ Lasso} \end{cases}$$

The variable selection property of the Lasso The least squares solution is marked as $\hat{\beta}$.

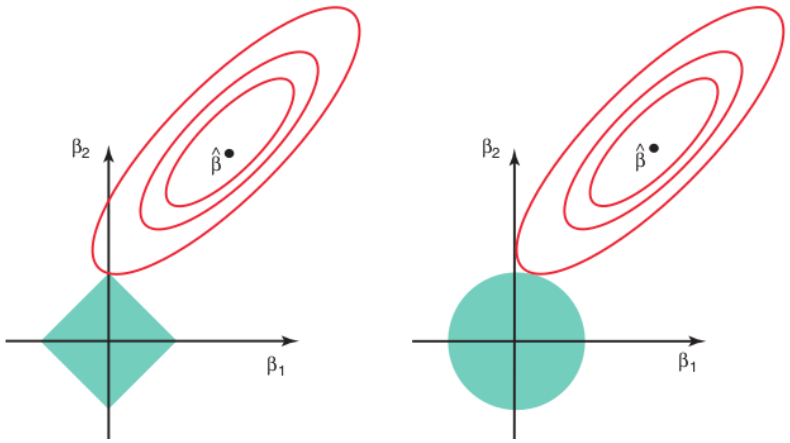


Figure 1.13: Contours of the error and constraint functions for the lasso (left) and ridge regression (right). The solid blue areas are the constraint regressions, $|\beta_1| + |\beta_2| \leq s$ and $\beta_1^2 + \beta_2^2 \leq s$, while the red ellipses are the contours of the RSS (all of the points on a given ellipse share a common value of the RSS).

Since ridge regression has a circular contraint with no sharp points, this intersection will not generally occur on an axis, and so the ridge regression coefficient estimates will be exclusively non-zero.
However, the lasso constraint has *corners* at each of the axes, and so the ellipse will often intersect the constraint region at an axis.

Comparing the Lasso and Ridge Regression In general, one might expect the lasso to perform better in a setting where a relatively small number of predictors have a substantial coefficients, and the remaining predictors have coefficients that are very small or that equal zero.
Ridge regression will perform better when the response is a function of many predictors, all with coefficients of roughly equal size.

A technique such as cross-validation can be used in order to determine which approach is better on a particular data set.

Bayesian Interpretation It assumes that the coefficient vector β has some *prior* distribution, say $p(\beta)$, where $\beta = \begin{pmatrix} \beta_0 \\ . \\ . \\ . \\ \beta_p \end{pmatrix}$ The likelihood of the data can be written as $f(Y|X,\beta)$, where $X = \begin{pmatrix} X_1 \\ . \\ . \\ . \\ X_p \end{pmatrix}$ Multiplying the prior distribution by the likelihood gives us (up to a proportionality constant) the posterior distribution which takes the form:

$p(\beta|X,Y) \propto f(Y|X,\beta)p(\beta|X) = f(Y|X,\beta)p(\beta)$
We assume that $Y = \beta_0 + \sum_{i=1}^{p} X_i\beta_i + \epsilon$ and: $p(\beta) = \prod_{j=1}^{p} g(\beta_j)$, for some density function g.
$g \hookrightarrow \mathcal{N}(0, f(\lambda)^2) \Rightarrow$ the most likely value for β is given by the *Ridge Regression* solution
$g \hookrightarrow Laplace(0, f(\lambda)) \Rightarrow$ the posterior mode for β is the *Lasso Regression* solution.

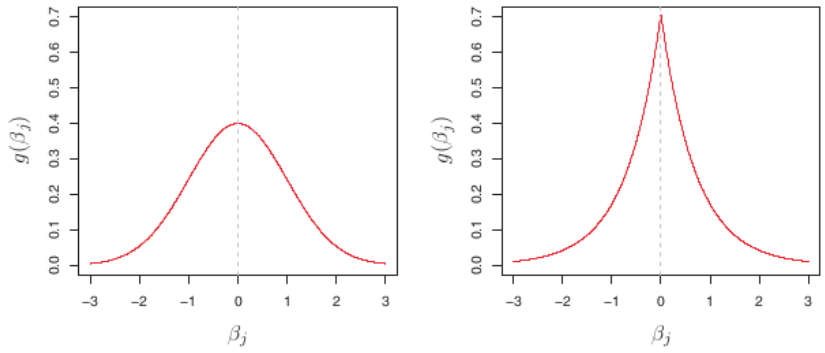


Figure 1.14: Left: Ridge regression is the posterior mode for β under a Gaussian prior.
Right: The lasso is the posterior mode for β under a double-exponential prior.

We can generalize ridge regression and the lasso, and view them as Bayes estimates:

$$\tilde{\beta} = \min_{\beta} \left\{ \sum_{i=1}^{N} \left(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j \right)^2 + \lambda \sum_{j=1}^{p} |\beta_j|^q \right\}$$

for $q \geq 0$ The value: $\begin{cases} q = 0: & \textit{subset selection} \text{ the penalty counts the number of nonzero parameters} \\ q = 1: & \text{the } \textit{lasso} \\ q = 2: & \text{the } \textit{ridge regression} \end{cases}$
The case $q = 1$ (lasso) is the smallest q such that the constraint region is convex: non-convex constraint regions make the optimization problem more difficult.

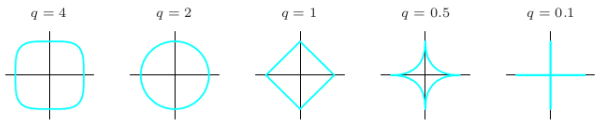


Figure 1.15: Contours of constant value of $\sum_j |\beta_j|^q$

Elastic-net

$$\lambda \sum_{j=1}^{p} \left(\alpha \beta_j^2 + (1-\alpha)|\beta_j|\right)$$

```
reg = linear_model.ElasticNet(random_state=0)
reg.fit([[0, 0], [1, 1]], [0, 1])
print(reg.coef_)
```

Selecting the Tuning Parameter Cross-validation provides a simple way to select λ or equivalently s:

- We choose a grid of λ values
- We compute the cross-validation for each value of λ
- We select the tuning parameter value for which the cross-validation error is smallest.

Finally the model is re-fit using all of the available observations and the selected value of the tuning parameter.

```
X, y = load_diabetes(return_X_y=True)
clf = RidgeCV(cv = 5, random_state=0).fit(X, y)
print(clf.alpha_)

clf = LassoCV(cv = 5, random_state=0).fit(X, y)
print(clf.alpha_)

clf = ElaticNet(cv = 5, random_state=0).fit(X, y)
print(clf.alpha_) # corresponds to lambda
print(clf.l1_ratio_) # corresponds to alpha
```

Least Angle Regression

Least Angle Regression

1. Standardize the predictors to have mean zero an unit norm. Start with the residual $\boldsymbol{r} = \boldsymbol{y} - \overline{\boldsymbol{y}}$, and for all $i \in [\![1, p]\!]$ $\beta_i = 0$
2. Find the predictor x_j most correlated with $\boldsymbol{r}$
3. Move β_j from 0 to its least-squares coefficient $\frac{\langle \boldsymbol{x}_j | \boldsymbol{r} \rangle}{\langle \boldsymbol{x}_j | \boldsymbol{x}_j \rangle}$, then recompute $\boldsymbol{r}$. Find the predictor x_k most correlated with the new $\boldsymbol{r}$
4. Move β_k from 0 to its least-squares coefficient $\frac{\langle \boldsymbol{x}_k | \boldsymbol{r} \rangle}{\langle \boldsymbol{x}_k | \boldsymbol{x}_k \rangle}$, then recompute $\boldsymbol{r}$. Find the predictor x_l most correlated with the new $\boldsymbol{r}$...
5. Continue in this way until all p predictors have been entered. After $\min(N-1, p)$ steps, we arrive at the full least-squares solution.

Suppose $\mathcal{A}_k$ is the active set of variables at the beginning of the k^{th} step and let $\beta_{\mathcal{A}_k}$ be the coefficient vector for these variables at this step, there will be $k-1$ nonzero values and the one just entered will be zero.
If $\boldsymbol{r}_k = \boldsymbol{y} - \boldsymbol{X}_{\mathcal{A}_k}\beta_{\mathcal{A}_k}$ is the current residual, then the direction for this step is:

$$\delta_k = (\boldsymbol{X}_{\mathcal{A}_k^T}\boldsymbol{X}_{\mathcal{A}_k^T})^{-1}\boldsymbol{X}_{\mathcal{A}_k^T}^T\boldsymbol{r}_k$$

The name "least angle" arises from a geometrical interpretation of this process; $\boldsymbol{u}_k$ makes the smallest (and equal) angle with each of the predictors in $\mathcal{A}_k$
Advantages and Disadvantages:

+ Numerically efficient when $p \gg n$
+ As fast as forward selection and has the same order of complexity as OLS
+ Produces a full piecewise linear solution path
+ Stable
+ Esealy modified to produce for other estimators like the Lasso
- Because LARS is based upon iterative refitting of the residuals, it would appears to be especially sensitive to the effect noise.

Least Angle Regression: Lasso Modification

4a If a non-zero coefficient hits zero, drop its variable from the active set of varibalbes and recompute the current joint least squares direction

```
reg = Lars()
reg.fit([[ -1, 1], [0, 0], [1, 1]], [-1.1111, 0, -1.1111])
print(reg.coef_)
```

1.5.3 Dimension reduction methods

Principal Component Analysis

Intuitive PCA can be thought of as fitting a $p-dimensional$ to the data, which each axis of the ellipsoid represents a principal component. If some axis of the ellipsoid is small, then the variance along that axis is also small, and by omitting that axis and its corresponding principal component from our representation of the dataset.

Methods To find the axes ellipsoid, we must first subtract the mean of each variable from the dataset to center the data around the origin.

1. Subtract the mean of each variable from the dataset to center the data around the origin.
2. Computing the covariance matrix of the data and calculate the eigenvalues and corresponding eigenvectors.
3. Normalize each of the orthogonal eigenvectors to turn into unit vectors.

Then each of the mutually orthogonal, unit eigenvectors can be interpreted as an axis of the ellipsoid fitted to the data.
Our covariance matrix will be transformed into a diagonalised form with the diagonal elements representing the variance of each axis.
The proportion of the variance that each eigenvector represents can be calculated by dividing the eigenvalue corresponding to that eigenvector by the sum of all eigenvalues.

Mathematically PCA is defined as an orthogonal linear transformation that transforms the data to a new coordinate system such that the greatest variance by some scalar projection of the data comes to lie on the first coordinate (the first principal component), the second greatest variance on the second coordinate.

The transformation is defined by a set of size l of $p-dimensional$ vectors of weights $\boldsymbol{\omega}_{(k)} = (\omega_1, \cdots, \omega_p)_{(k)}$, with $k \in [\![1, l]\!]$, that map each row vector $\boldsymbol{x}_{(i)}$ of $\boldsymbol{X}$ to a new vector of principal component scores $t_{(i)} = (t_1, \cdots, t_l)_{(i)}$ given by: $\forall (i,k) \in [\![1,n]\!] \times [\![1,l]\!]\, t_{k\,(i)} = \langle \boldsymbol{x}_i | \boldsymbol{\omega}_{(k)} \rangle$ with $l \leq p$

In order to maximize variance, the first weight vector $\boldsymbol{\omega}_{(1)}$ thus has to satisfy:

$$\boldsymbol{\omega}_{(1)} = \max_{\|\boldsymbol{\omega}\|=1} \|\boldsymbol{X}\boldsymbol{\omega}\|^2 = \max_{\|\boldsymbol{\omega}\|=1} (\boldsymbol{X}\boldsymbol{\omega})^T \boldsymbol{X}\boldsymbol{\omega}$$

Since $\boldsymbol{\omega}_{(1)}$ has been defined to be a unit vector, it equivalently also satisfies:

$$\boldsymbol{\omega}_{(1)} = \max_{\|\boldsymbol{\omega}\|=1} \frac{\boldsymbol{\omega}^T \boldsymbol{X}^T \boldsymbol{X} \boldsymbol{\omega}}{\boldsymbol{\omega}^T \boldsymbol{\omega}}$$

The quantity to be maximised can be recognized as a Rayleigh quotient. A standard result for a positive semidefinite matrix such as $\boldsymbol{X}^T\boldsymbol{X}$ is that the Rayleigh quotient's maximum possible value is the largest eigenvalue of the matrix, which occurs when $\boldsymbol{\omega}$ is the corresponding eigenvector.

Further components:
The k^{th} component can be found by subtracting the first $k-1$ principal components from $\boldsymbol{X}$: $\hat{\boldsymbol{X}}_k = \boldsymbol{X} - \sum_{s=1}^{k-1} \boldsymbol{X}\boldsymbol{\omega}_{(s)}\boldsymbol{\omega}_{(s)}^T$ and then finding the weight vector which extracts the maximum variance from this new data matrix:

$$\boldsymbol{\omega}_{(k)} = \max_{\|\boldsymbol{\omega}\|=1} \left\| \hat{\boldsymbol{X}}_k \boldsymbol{\omega} \right\|^2 = \max_{\|\boldsymbol{\omega}\|=1} \frac{\boldsymbol{\omega}^T \hat{\boldsymbol{X}}_k^T \hat{\boldsymbol{X}}_k \boldsymbol{\omega}}{\boldsymbol{\omega}^T \boldsymbol{\omega}}$$

It turns out that this gives the remaining eigenvectors of $\boldsymbol{X}^T\boldsymbol{X}$, with the maximum values for the quantity in brackets given by their corresponding eigenvalues. Thus the weight vectors are eigenvectors of $\boldsymbol{X}\boldsymbol{X}^T$.
The full principal components decomposition of $\boldsymbol{X}$ can therefore be given as: $\boldsymbol{T} = \boldsymbol{X}\boldsymbol{\Omega}$ where $\boldsymbol{\Omega}$ is a $p \times p$ of weights whose columns are the eigenvectors of $\boldsymbol{X}^T\boldsymbol{X}$.

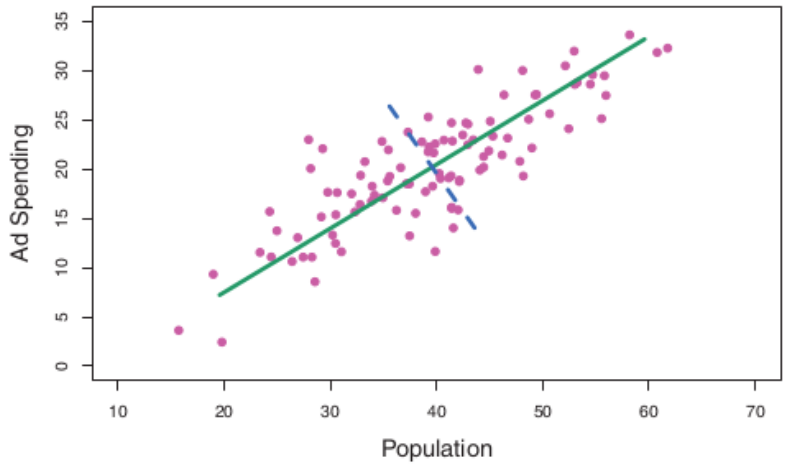


Figure 1.16: The population size and ad spending for 100 different cities are shown as purple circles. The green solid line indicates the first principal component, and the blue dashed line indicates the second principal compoment.

Principal Components Regression and Ridge Regression The *Principal Components Regression* (PCR) approach involves constructing the first M principal components $(Z_i)_{1\leq i\leq M}$ and then using these components as the predictors in a linear regression model that is fit using least squares.

One can show that PCR and Ridge Regression are very closely related, one can even think of ridge regression as a continuous version of PCR.
When performing PCR we generally recommend *standardizing* each predictor, using, prior to generating the principal components.

Ridge regression shrinks the coefficients of the principal components, shrinking more depending on the size of the corresponding eigenvalue; principal components regression discard the $p - M$ smallest eigenvalue components.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import statsmodels.api as sm
from statsmodels.multivariate.pca import PCA
import sklearn
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA as PCA_sk

df = pd.read_csv('myfile.csv', sep=';')
X, y = df.iloc[:, 1:], df.iloc[:, 0]
pca_model = PCA(X,
    standardize=False, # mean 0 and unit variance
    demean=True # to subtract columns by their mean
```

```
)
# Standardize the Data
fig = pca_model.plot_scree(log_scale=False) # to plot the
    explained
# variance of components.
print(pca_model.loadings.iloc[:, :2]).argsort() # to
# estimate the implication of a given variable in a the
# 2 first components.

# Computation
X_rm_mean = StandardScaler(with_std=False).fit_transform(X)
pca = PCA_sk(n_components=2) # 2 assuming that the 2 first
# principal components explained the quasi-totality of the
    variance
principalCompnents = pca.fit_transform(X_rm_mean)
df_pca = pd.DataFrame(data= principalCompnents,
    columns=['pc1', 'pc2'])
df_transf = df.iloc[:, 0].join(df_pca)

# Visualize
fig = plt.figure()
ax.set_xlabel('Principal Component 1', fontsize=15)
ax.set_ylabel('Principal Component 2', fontsize=15)
ax.set_title('2 component PCA', fontsize=20)
ax.scatter(df_transf.loc[:, 'pc1'], df_transf.loc[:, 'pc2'])

# Explained Variance
print(pca.explained_variance_ratio_)
```

Partial Least Squares It is a *supervised* alternative to PCR, PLS approach attempts to find directions that help explain both the response and the predictors.

- To standardize the p predictors
- Computation of the first direction Z_1 by setting each ϕ_{j1} equal to the coefficient from the simple linear regression of Y onto X_j.

Hence in computing $Z_1 = \sum_{j=1}^{p} \phi_{j1} X_j$ places the highest weight on the variables that are most strongly related to the response.

Partial Least Squares

- Standardize each x_j to have mean zero and variance one.
  Set $\hat{\boldsymbol{y}}^{(0)} = \overline{\boldsymbol{y}}$ and for all $j \in [\![1, p]\!]$ $x_j^{(0)} = x_j$
- For $m \in [\![1, p]\!]$

  (a) $\boldsymbol{z}_m = \sum_{j=1}^{p} \frac{\langle \boldsymbol{x}_j^{(m-1)} \,\middle|\, \boldsymbol{y}^{(m-1)} \rangle}{\langle \boldsymbol{x}_j^{(m-1)} \,\middle|\, \boldsymbol{x}_j^{(m-1)} \rangle} \boldsymbol{x}_j^{(m-1)}$

(b) $\hat{\boldsymbol{y}}^{(m)} = \hat{\boldsymbol{y}}^{(m-1)} + \frac{\langle \boldsymbol{z}_m | \boldsymbol{y}^{(m-1)} \rangle}{\langle \boldsymbol{z}_m | \boldsymbol{z}_m \rangle} \boldsymbol{z}_m$

(c) Orthogonalize each $\boldsymbol{x}_j^{m-1}$ with respect to $\boldsymbol{z}_m$:

$$\boldsymbol{x}_j^{(m)} = \boldsymbol{x}_j^{(m-1)} - \frac{\langle \boldsymbol{z}_m | \boldsymbol{x}_j^{(m-1)} \rangle}{\langle \boldsymbol{z}_m | \boldsymbol{z}_m \rangle} \boldsymbol{z}_m \text{ for } j \in [\![1, p]\!]$$

PLS seeks directions that havae high variance and have high correlation with response, in contrast to PCR which keys only on high variance. Particularly the m^{th} principal component direction v_m solves:

$$\max_{\alpha} \mathbb{V}(\boldsymbol{X}\alpha) \text{ such that } \begin{cases} \|\alpha\| = 1 \\ \forall l \in [\![1, m-1]\!] \, \alpha^T \boldsymbol{S} v_l = 0 \end{cases}$$

$\boldsymbol{S}$ is the sample covariance matrix of the $\boldsymbol{x}_j$ The conditions $\alpha^T \boldsymbol{S} v_l = 0$ ensures that $z_m = \boldsymbol{X}\alpha$ is uncorrelated with all the previous linear combinations $z_f = \boldsymbol{X} v_l$
The m^{th} PLS direction $\hat{\Phi}_m$ solves

$$\max_{\alpha} Corr^2(\boldsymbol{y}, \boldsymbol{X}\boldsymbol{\alpha}) \mathbb{V}(\boldsymbol{X}\alpha) \text{ such that } \begin{cases} \|\alpha\| = 1 \\ \forall l \in [\![1, m-1]\!] \, \alpha^T \boldsymbol{S} \hat{\Phi}_l = 0 \end{cases}$$

PLS, PCR and Ridge Regression tend to behave similarly, Ridge Regression may be preferred because it shrinks smoothly, rather than in discrete steps. Lasso falls somewhere between ridge regression and best subset regression, and enjoys some of the properties of each.

```
X = [[0., 0., 1.], [1., 0., 0.], [2., 2., 2.], [2., 5., 4.]]
y = [0.1, 0.9, 12.3]

pls1 = PLSRegression(n_components=2)
pls1.fit(X, y)
print(pls1.score(X, y))
```

1.5.4 Multiple Outcome Shrinkage and Selection

Canonical correlation analysis (CCA) is a data reduction technique developped for the multiple output case. Similar to PCA, CCA finds a sequence of uncorrelated linear combinations $\boldsymbol{X}v_m \text{for} m \in [\![1, M]\!]$ of the $\boldsymbol{x}_j$ and a corresponding sequence of uncorrelated linear combinations $\boldsymbol{Y}u_m$ of the respons $\boldsymbol{y}_k$ such that the correlations :

$$Corr^2(\boldsymbol{Y}u_m, \boldsymbol{X}v_m)$$

are succesively maximized.

Reduced-rank regression Given an error covariance $Cov(\epsilon) = \Sigma$, we solve the following:

$$\hat{\boldsymbol{B}}^{rr}(m) = \min_{rank(\boldsymbol{B})=m} \sum_{i=1}^{N} \left(y_i - \boldsymbol{B}^T x_i\right)^T \Sigma^{-1} \left(y_i - \boldsymbol{B}^T x_i\right)$$

With Σ replaced by the estimate $\frac{\boldsymbol{Y}^T\boldsymbol{Y}}{N}$ that the solution is given by a CCA of $\boldsymbol{Y}$ and $\boldsymbol{X}$:

$$\hat{\boldsymbol{B}}^{rr}(m) = \hat{\boldsymbol{B}}\boldsymbol{U}_m\boldsymbol{U}_m^-$$

where $\boldsymbol{U}_m$ is the $K \times m$ sub-matrix of $\boldsymbol{U}$ consisting of the first m columns and $\boldsymbol{U}$ is the $K \times M$ matrix of left canonical vectors $u_1, u_2 \cdots u_M$. $\boldsymbol{U}_m^-$

1.5.5 More on the Lasso and Related Path algorithms

Incremental Forward Stagewise Regression

Incremental Forward Stagewise Regression - FS_ϵ

1. Start with the residual $\boldsymbol{r}$ equal to $\boldsymbol{y}$ and $\beta_1, \beta_2, \cdots, \beta_p = 0$. All the predictors are standardized to have mean zero and unit norm.
2. Find the predictor x_j most correlated with $\boldsymbol{r}$
3. Update $\beta_j \leftarrow \beta_j + \delta_j$, where $\delta_j = \epsilon \times \text{sign}[\langle \boldsymbol{x}_j | \boldsymbol{r} \rangle]$ and $\epsilon > 0$ is a small step size, and set $\boldsymbol{r} \leftarrow \boldsymbol{r} - \delta_j \boldsymbol{x}_j$
4. Repeat steps 2 and 3 many times, until the residuals are uncorrelated with all the predictors.

The linear regression version of the forward-stagewise generates a coefficient profile by repeatedly updating (by a small amount ϵ the coefficient of the variable most correlated with the current residual.
If $\delta_j = \langle \boldsymbol{x}_j | \boldsymbol{r} \rangle$ (the least squares coefficient of the residual on j^{th} predictor), then this is exactly the usual forward stagewise procedure (FS).
Letting $\epsilon \to 0$ gives the right panel, which in this case is identical to the lasso path. We call this limiting procedure infinitesimal forward stagewise regression or FS_0

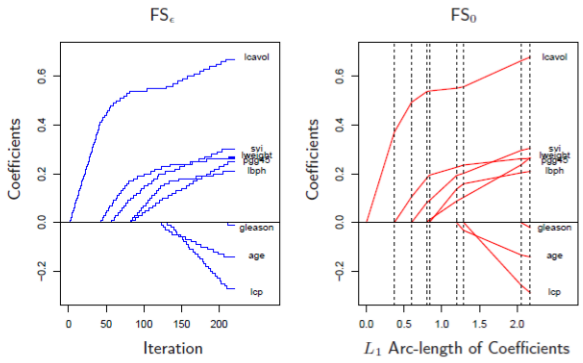


Figure 1.17: The left pannel shows incremental forward stagewise regression with step size $\epsilon = 0.01$. The right panel shows the infinitesimal version FS_0 obtained letting $\epsilon \to 0$

Least Angle Regression FS_0

4 Find the new direction by solving the constrained least squares problem:

$$\min_b \|\boldsymbol{r} - \boldsymbol{X}_{\mathcal{A}} b\|_2^2 \text{ subject to } b_j s_j \geq 0, j \in \mathcal{A}$$

where s_j is the sign of $\langle \boldsymbol{x}_j | \boldsymbol{r} \rangle$

While the Lasso makes optimal progress in terms of reducing the residual sum-of-squares per unit increase in L_1-norm of the coefficient vector β. FS_0 is optimal per unit increase in L_1 arc-length traveled along the coefficient path. Hence its coefficient path is discouraged from changing direction too often.

Piecewise-Linear Path Algorithms

$$\hat{\boldsymbol{\beta}}(\lambda) = \min_{\beta} (\boldsymbol{R}(\beta) + \lambda \boldsymbol{J}(\beta)) \text{ with} \boldsymbol{R}(\beta) = \sum_{i=1}^{N} \boldsymbol{L} \left(y_i, \beta_0 + \sum_{j=1}^{p} x_{ij} \beta_j \right)$$

with both the loss function $\boldsymbol{L}$ and the penalty function $\boldsymbol{J}$ are convex.
Then the following are sufficient conditions for the solution path $\beta(\lambda)$ to be piecewise linear.

1. R is quadratic or piecewise-quadratic as a function of β
2. J is a piecewise linear in β

The Dantzig Selector

$$\min_{\beta} \left\| \boldsymbol{X}^T (\boldsymbol{y} - \boldsymbol{X}\beta) \right\|_{\infty} \text{ subject to } \|\beta\|_1$$

Here $\|.\|_{\infty}$ denotes the L_∞ norm, the maximum absolute value of the components of the vector.

The Grouped Lasso Suppose that the p predictors are divised in L groups, with p_l the number in group l.
$\boldsymbol{X}_l$ represents the predictors corresponding to the l^{th} group, with corresponding coefficient vector β_l.
The grouped-lasso minimizes the convex criterion

$$\min_{\beta \in \mathbb{R}^p} \left(\left\| \boldsymbol{y} - \beta_0 \boldsymbol{1} - \sum_{l=1}^{L} \boldsymbol{X}_l \beta_l \right\|_2^2 + \lambda \sum_{l=1}^{L} \sqrt{p_l} \|\beta_l\|_2 \right)$$

where the $\sqrt{p_l}$ terms accounts for the varying group sizes, $\|.\|_2$ is the Euclidean norm (not squared). Since the Euclidean norm of a vector β_l is zero only if all its components are zero, this procedure encourages sparsity at both the group and individual levels.

1.5.6 Consideration in high dimensions

High-Dimensional Data Data sets containing more features than observations are often referred to as *high-dimensional*. Classical approaches such as least squares linear regression are not appropriate in this setting.

Regression in High Dimensions Ridge regression, the lasso and principal components regression, are particularly useful for performing regression in the high-dimensional setting. Essentially these approaches avoid overfitting by using a less flexible fitting approach than least squares.

1.6 Moving beyond linearity

1.6.1 Polynomial Regression

Definition Historically when the relationship between the response and the predictors is non-linear we used to replace the standard linear model

$$y_i = \beta_0 + \beta_1 x_i + \epsilon_i$$

with a polynomial function:

$$\boxed{y_i = \sum_{r=0}^{d} \beta_r x_i^r}$$

Generally speaking, it is unusual to use d greater than 3 or 4.

```
import pandas as pd
import sklearn
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.pipeline import Pipeline

df = pd.read_csv('myFile.csv', sep=';')
y, X = df.iloc[:, 0].values, df.iloc[:, 1].values.reshape(-1
    , 1)

model = Pipeline([
  ('poly', PolynomialFeatures(degree=3)),
  ('linear', LinearRegression(fit_intercept=False))
  ])
model = model.fit(X, y)
print(model.score(X, y))
```

1.6.2 Regression splines

Piecewise Polynomials It involves fitting separate low-degree polynomials over different regions of X.
The points where the coefficients change are called *knots*. It is a function $f(X)$ that is obtained by dividing the domain of X into contiguous intervals, and representing f by a separate polynomial in each interval. More generally, an

order$-M$ spline with knots $\zeta_j, j \in [\![1, K]\!]$ is a piecewise-polynomial of order M, and has continuous derivatives up to order $M-2$. Likewise the general form for the truncated-power basis set would be:

$$\begin{aligned} h_j(X) &= X^{j-1}, & j &\in [\![1, M]\!] \\ h_{M+j} &= (X-\zeta_l)_+^{M-1}, & l &\in [\![1, K]\!] \end{aligned}$$

The spline Basis Representation A cubic spline with K knots can be modeled as:

$$y_i = \beta_0 + \sum_{r=1}^{K+3} \beta_r b_r(x_i)$$

for an appropriate choice of *basis functions* $(b_i)_{1 \leq i \leq K+3}$
The most direct way to represent cubic spline is to start off with a basis for a cubic polynomial -namely, x, x^2, x^3 -and then add one *truncated power basis* function per knot. A truncated power basis function, for a cubic polynomial, is defined as:

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

where ζ is the knot.
In order to fit a cubic spline to a data set with K knots, we perform least squares regression with an intercept and $3+K$ predictors of the form $X, X^2, X^3, h(X, \zeta_1), h(X, \zeta_2), , .., h(X, \zeta_K)$ where $(\zeta_i)_{1 \leq i \leq K}$ are the knots. This amounts to estimating a total of $K+4$ regression coefficients.
Cubic splines are popular because most human eyes cannot detect discontinuity at the knots.

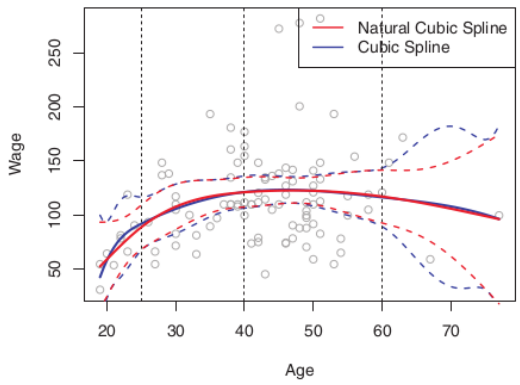


Figure 1.18: A cubic spline and a natural cubic spline, with 3 knots, fit to a subset of the Wage data, confidence interval as dashed lines.

A natural spline is a regression spline with additional boundary constraints: the function is required to be linear at the boundary (in the region where X is smaller than the smallest knot, or larger than the largest knot).

This frees up 4 degrees of freedom (2 constraints each in both boundary regions), which can be spent more profitably by sprinkling more knots in the interior region. A natural cubic spline with K knots is represented by K basis functions.

$$\begin{cases} N_1(X) = 1 \\ N_2(X) = X \end{cases}, \; N_{k+2}(X) = d_k(X) - d_{K-1}(X)$$

where $d_k(X) = \dfrac{(X-\zeta_k)_+^3 - (X-\zeta_K)_+^3}{\zeta_K - \zeta_k}$

Choosing the Number and Locations of the Knots One option is to place more knots in places where we feel the function might vary most rapidly, and to place fewer knots where it seems more stable.
For the number of knots we can use cross-validation.

```
import pandas
import statsmodels.api as sm
import sklearn
from sklearn.model_selection import KFold
import patsy
from patsy import dmatrix

# Choosing the good number of knots
y, x = df.iloc[:, 0], df.iloc[:, 1]
kf = KFold(n_splits=5)
n = 52
cv_set = {str(i):0 for i in range(1, n)}
for i in range(1, n):
    n_knots = i
    knot_list = np.quantile(x, np.linspace(0, 1, n_knots + 2
    ))[1:-1]
    mse_list = []
    for train, test in kf.split(x):
        x_natural = dmatrix('cr(x, knots=knot_list)', {'x':x
    [train]})
        fit_natural = sm.GLM(y[train], x_natural).fit()
        yhat = fit_natural.predict(dmatrix('cr(x, knots=
    knot_list)', {'x':x[test]}))
        mse = ((yhat-y[test])**2).mean()
        # mse_list.append(mse)
        mse_list.append(mse)
        # print(mse)
    cv_set[str(i)] = np.array(mse_list).mean()
cv_list = list({k:v for k,v in sorted(cv_set.items(), key=
    lambda item: item[1])})


n_knots = int(cv_list[0])
knot_list = np.quantile(x, np.linspace(0, 1, n_knots + 2))[1
    :-1]
# Natural
x_natural = dmatrix('cr(x, knots = knot_list)', {'x':x})
fit_natural = sm.GLM(y, x_natural).fit()
```

```
34
35 # Create spline
36 xp = np.linspace(x.min(), x.max(), 50)
37 line_natural = fit_natural.predict(dmatrix('cr(xp,knots=
       knot_list)',
38                                                 {'xp':xp}))
39
40 plt.figure()
41 plt.scatter(df.iloc[:, 0], df.iloc[:, 1])
42 plt.plot(xp, line_natural, color='red', label="Natural
       Spline Regression")
43 for k in knot_list:
44     plt.axvline(k, color='gray', ls='--')
45 plt.legend()
46 plt.show()
```

Comparison to Polynomial Regression Regression splines give superior results to polynomial regression.
This is because unlike polynomials, which must use a higher degree to produce flexible fits, splines introduce flexibility by increasing the number of knots but keeping the degree fixed.
Splines produce also more stable estimates.

1.6.3 Smoothing splines

Definition We want a function g that makes $RSS = \sum_{i=1}^{n} (y_i - g(x_i))^2$ small but also with constraint on $g(x_i)$ therewith to avoid overfitting and get smooth curve.
A natural approach is to find the function g that minimizes:

$$\underbrace{\sum_{i=1}^{n} (y_i - g(x_i))^2}_{\text{Loss function}} + \lambda \underbrace{\int g''(t)^2 dt}_{\text{Penalty term}}$$

where λ is a nonnegative *tunning parameter*, and g is a *smoothing spline*.

The first derivative $g'(t)$ measures the slope of a function at t, and the second derivative corresponds to the amount by which the slope is changing, if $g(t)$ is very wiggly near t $g''(t)$ is large, otherwise it is close to zero.
$\int g''(t)^2 dt$ is a measure of the total change in the function $g'(t)$, over its entire range. Since the solution is a natural spline, we can write is as:

$$f(x) = \sum_{j=1}^{N} N_j(x)\theta_j$$

where the $N_j(x)$ are a $N-$dimensional set of basis functions for representing this family of natural splines:

$$RSS(\theta, \lambda) = (\boldsymbol{y} - \boldsymbol{N}\theta)^T (\boldsymbol{y} - \boldsymbol{N}\theta) + \lambda \theta^T \Omega_N \theta$$

where $\begin{cases} \{N\}_{ij} = N_j(x_i) \\ \{\Omega_N\}_{jk} = \int N_j^{''}(t) N_k^{''}(t) dt \end{cases}$ The solution is easily seen to be:

$$\boxed{\hat{\theta} = \left(\boldsymbol{N}^T \boldsymbol{N} + \lambda \Omega_N\right)^{-1} \boldsymbol{N}^T}$$

The fitted smoothing spline is given by:

$$\hat{f}(x) = \sum_{j=1}^{N} \boldsymbol{N}_j(x) \hat{\theta}_j$$

Choosing the smoothing parameter λ The vector of fitted values, when applying a smoothing spline to the data, can be written as $n \times n$ matrix S_λ times the response y:

$$\hat{g}_\lambda = S_\lambda y$$

where $\hat{g}$ is a smoothing spline for a particular choice of λ, that is it is a n-vector containing the fitted values of the smoothing spline at the training points $(x_i)_{1 \leq i \leq n}$

```
import csaps
from csaps import csaps

kf = KFold(n_splits=5)
df2 = df.iloc[:, [0, 1]].groupby(df.columns[1]).agg(
    np.median).reset_index()
cv_dict = {str(i):0 for i in range(0, 100+1, 1)}
for k in range(0, 100+1, 1):
    mse_list = []
    for train, test in kf.split(df2.budget_std):
        smoothing_spline = csaps(df2.iloc[train, 1],
                                 df2.iloc[train, 0],
                                 smooth=round(k/100, 2))
        mse = ((smoothing_spline(
            df2.iloc[test, 0])-df2.iloc[test, 1])**2).mean()
        mse_list.append(mse)
    cv_dict[str(k)] = np.array(mse_list).mean()
cv_list = list({k:v for k,v in sorted(cv_dict.items(), key=
    lambda item: item[1])})

smoothing_spline = csaps(df2.iloc[:, 1], df2.iloc[:, 0],
                         smooth=round(float(cv_list[0])/100,
    2))
# Create spline line for 50 evenly spaced values of age
xp = np.linspace(df2.iloc[:, 1].min(), df2.iloc[:, 1].max(),
    50)
yp = smoothing_spline(xp)
```

Degrees of freedom refer to the number of free parameters such as the number of coefficients fit in a polynomial or cubic spline. Denote by $\hat{f}$ the $N-$ vector of fitted values $\hat{f}(x_i)$ at the training predictors x_i:

$$\begin{aligned}\hat{f} =& \boldsymbol{N}\left(\boldsymbol{N}^T\boldsymbol{N}+\lambda\boldsymbol{\Omega}_N\right)^{-1}\boldsymbol{N}^T\boldsymbol{y}\\ =& \boldsymbol{S}_\lambda\boldsymbol{y}\end{aligned}$$

The finite linear operator $\boldsymbol{S}_\lambda$ is known as the *smoother matrix*
Linear operator are familiar in more traditional least squares fitting as well, suppose B_ζ is a $N \times M$ matrix of M cubic-spline basis functions evaluated at the N training points x_i, with knot sequence ζ and $M \ll N$. Then the vector of fitted spline values is given by:

$$\begin{aligned}\hat{f} =& \boldsymbol{B}_\zeta\left(\boldsymbol{B}_\zeta^T\boldsymbol{B}_\zeta\right)^{-1}\boldsymbol{B}_\zeta^T\boldsymbol{y}\\ =& \boldsymbol{H}_\zeta\boldsymbol{y}\end{aligned}$$

The linear operator $\boldsymbol{H}_\zeta$ is a projection operator, also known as the *hat matrix*.

- Both $\boldsymbol{H}_\zeta$ and $\boldsymbol{S}_\lambda$ are symmetric, positive semi-definite
- $\boldsymbol{H}_\zeta\boldsymbol{H}_\zeta = \boldsymbol{H}_\zeta$ (*idempotent*)
  $\boldsymbol{S}_\lambda\boldsymbol{S}_\lambda \preceq \boldsymbol{S}_\lambda$ meaning that the right-hand side exceeds the left-hand by a positive semi-definite matrix. This is a consequence of the shrinking nature of $\boldsymbol{S}_\lambda$
- $\boldsymbol{H}_\zeta$ has rank M, while $\boldsymbol{S}_\lambda$ has rank N.

The expression $M = trace\,(\boldsymbol{H}_\zeta)$ gives the dimension of the projection space, which is also the number of basis functions, and hence the number of parameters involved in the fit.
By analogy we define the *effective degrees of freedom* of a smoothing spline to be:

It is possible to show that as λ increases from 0 to ∞, the effective degrees of freedom, which we write df_{lambda}, decreases from n to 2.
Hence df_λ is a measure of the flexibility.

$$\boxed{df_\lambda = trace(\boldsymbol{S}_\lambda) = \sum_{i=1}^{n}\{S_\lambda\}_{ii}}$$

the sum of the diagonal elements of the matrix S_λ
Since $\boldsymbol{S}_\lambda$ is symmetric (and positive semi-definite), it has a real eigen-decomposition. Before we proceed, it is convenient to rewrite $\boldsymbol{S}_\lambda$ in the *Reinsch* form: $\boldsymbol{S}_\lambda = (\boldsymbol{I}+\lambda\boldsymbol{K})^{-1}$ where K does not depend on λ. Since $\hat{\boldsymbol{f}} = \boldsymbol{S}_\lambda\boldsymbol{y}$ solves: $\min_{f}(\boldsymbol{y}-\boldsymbol{f})^T(\boldsymbol{y}-\boldsymbol{f})+\lambda\boldsymbol{f}^T\boldsymbol{K}\boldsymbol{f}$ The eigen-decomposition of $\boldsymbol{S}_\lambda$ is:

$$\boldsymbol{S}_\lambda = \sum_{k=1}^{N}\rho_k(\lambda)\boldsymbol{u}_k\boldsymbol{u}_k^T$$

with $\rho_k(\lambda) = \dfrac{1}{1+\lambda d_k}$ and d_k the corresponding eigenvalue of K.

It turns out that the *leave-cross-out* cross-validation error (LOOCV) can be computed very efficiently for smoothing splines, with essentially the same cost as computing a single fit using the following formula:

$$RSS_{cv}(\lambda) = \sum_{i=1}^{n} \left(y_i - \hat{g}_{\lambda}^{(-i)}(x_i) \right)^2 = \sum_{i=1}^{n} \left[\frac{y_i - \hat{g}_{\lambda}(x_i)}{1 - \{S_{\lambda}\}_{ii}} \right]^2$$

The notation $\hat{g}_{\lambda}^{(-i)}(x_i)$ indicates the fitted value for this smoothing spline evaluated at x_i, where the fit uses all of the training observations except for the i^{th} observation (x_i, y_i). In contrast $\hat{g}_{\lambda}(x_i)$ indicates the smoothing spline function fit to all of the training observations and evaluated in x_i.

Automatic Selection of the Smoothing Parameters

The Bias-Variance Tradeoff

$$\begin{aligned} Cov\left(\hat{f}\right) =& Cov\left(\boldsymbol{S}_{\lambda}\boldsymbol{y}\right) \\ =& \boldsymbol{S}_{\lambda} Cov\left(\boldsymbol{y}\right) \boldsymbol{S}_{\lambda}^{T} \\ =& \boldsymbol{S}_{\lambda}\boldsymbol{S}_{\lambda}^{T} \end{aligned}$$

The diagonal contains the pointwise variances at the training x_i. The bias is given by:

$$\begin{aligned} \text{Bias}(\hat{\boldsymbol{f}}) =& \boldsymbol{f} - \mathbb{E}\left(\hat{\boldsymbol{f}}\right) \\ =& \boldsymbol{f} - \boldsymbol{S}_{\lambda}\boldsymbol{f} \end{aligned}$$

The integrated squared prediction error (EPE) combines both bias and variance in a single summary:

$$\begin{aligned} EPE\left(\hat{f}_{\lambda}\right) =& \mathbb{E}\left(Y - \hat{f}_{\lambda}(X)\right)^2 \\ =& \mathbb{V}\left(Y\right) + \mathbb{E}\left(\text{Bias}^2\left(\hat{f}_{\lambda}(X)\right) + \mathbb{V}\left(\hat{f}_{\lambda}\left(X\right)\right)\right) \\ =& \sigma^2 + \text{MSE}\left(\hat{f}_{\lambda}\right) \end{aligned}$$

1.6.4 Multidimensional Splines

Suppose $X \in \mathbb{R}^2$ and we have a basis of functions $h_{1k}(X_1)$ with $k \in [\![1, M_1]\!]$ for representing functions of coordinate X_1, and likewise a set of M_2 functions $h_{2k}(X_2)$ for coordinate X_2. Then the $M_1 \times M_2$ dimensional *tensor product basis* defined by:

$$g_{jk}(X) = h_{1j}(X_1)h_{2k}(X_2) \text{ with } (j,k) \in [\![1, M_1]\!] \times [\![1, M_1]\!]$$

can be used for representing a $2-$dimensional function:

$$g(X) = \sum_{j=1}^{M_1}\sum_{j=1}^{M_2}\theta_{jk}g_{jk}(X)$$

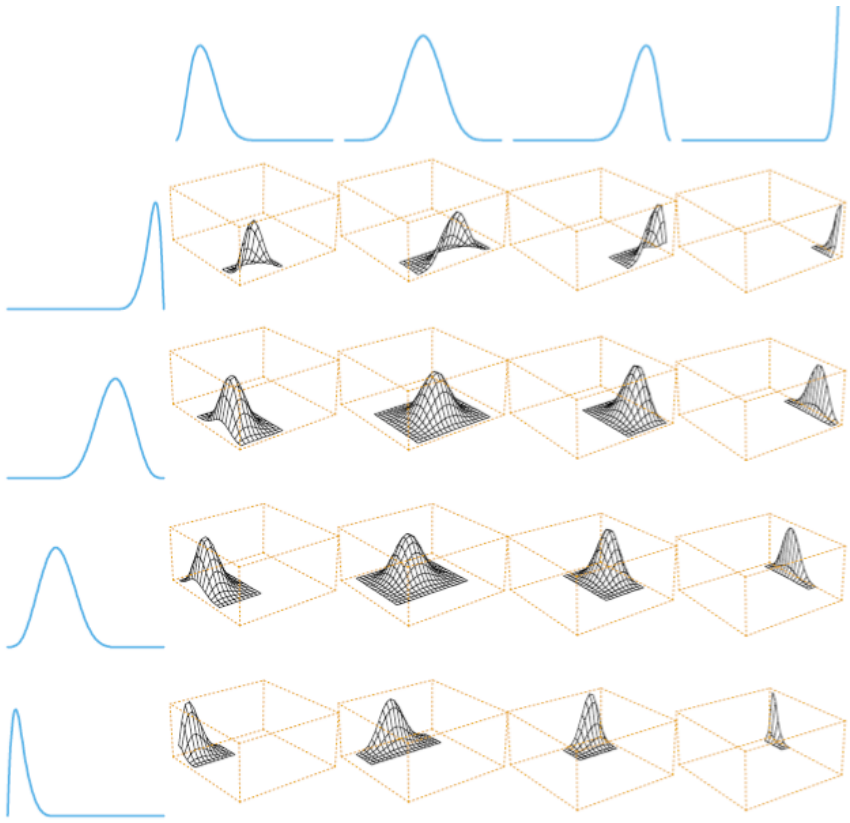


Figure 1.19: A tensor product basis of $B-$splines, showing some selected pairs. Each $2-$dimensionnal function is the tensor product of the corresponding one dimension marginals

1.6.5 Wavelet Smoothing

Wavelet typically use a complete orthonormal basis to represent functions, bu then shrink and select the coefficients toward a sparse representation.

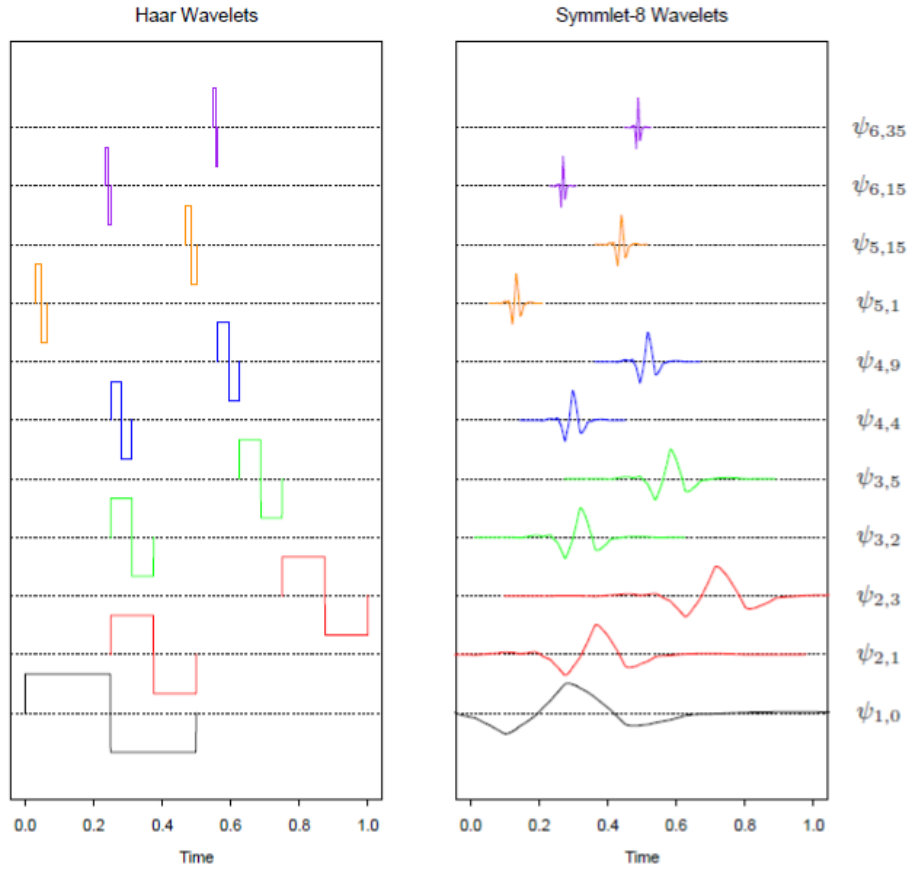


Figure 1.20: Some selected wavelets at different translations and dilatations for the Haar and symmlet families. The functions have been scaled to suit the display.

Wavelet Bases and the Wavelet Transform Wavelet bases are generated by translations and dilatations of a single scaling function $\phi(x)$ (also known as the *father*).
The *Haar* basis produces a piecewise-constant representation, thus if $\phi(x) = I(x \in [0,1])$, then $\phi_{0,k}(x) = \phi(x-k)$, k an integer, generates an orthonormal basis for functions with jumps at the integrs. Call this reference space V_0
The dilatations $\phi_{1,k}(x) = \sqrt{2}(2x-k)$ form an orthonormal basis for a space $V_1 \supset V_0$. More generally we have $\cdots \supset V_1 \supset V_1 \supset V_1 \supset \cdots$ where each V_j is spanned by $\phi_{j,k} = 2^{\frac{j}{2}}\phi(2^j x - k)$
We might represent a function in V_{j+1} by a component in V_{j+1} plus the component in the orthogonal complement W_j of V_j to V_{j+1} written as $V_{j+1} = V_j \bigoplus W_j$
The component in W_j represents detail, and we might wish to set some elements of this components to zero. It is easy to see that the functions $\psi(x-k)$ generated by the *mother wavelet* $\psi(x) = \phi(2x) - \phi 2x - 1$ form an orthonormal basis for W_0 for the Haar family. Likewise $\psi_{j,k} = 2^{\frac{j}{2}}\psi(2^j x - k)$ form a basis for W_j

Now $V_{j+1} = V_j \bigoplus W_j = V_{j-1} \bigoplus W_{j-1} \bigoplus W_j$, more generally: $V_j = V_0 \bigoplus W_0 \bigoplus W_1 \cdots W_{j-1}$.

Notice that since these spaces are orthogonal, all the basis functions are orthonormal. In fact, if the domain is discrete with $N = 2^J$ (time) points, this is as far as we can go.

Adaptive Wavelet Filtering Suppose $\boldsymbol{y}$ is the response vector, and $\boldsymbol{W}$ is the $N \times N$ orthonormal wavelet basis matrix evaluated at the N uniformly spaced observations. Then $\boldsymbol{y}^* = \boldsymbol{W}^T\boldsymbol{y}$ is called the *wavelet transform* of $\boldsymbol{y}$
A popular method for adaptive wavelet fitting is known as SURE(Stein Unbiased Risk Estimation) shrinkage, we start with the criterion:

$$\min_{\theta} \|\boldsymbol{y} - \boldsymbol{W}\boldsymbol{\theta}\|_2^2 + 2\lambda \|\theta\|_1$$

Because $\boldsymbol{W}$ is orthonormal, this leads to the simple solution:

$$\hat{\theta}_j = sign(y_i^*)(|y_j^*| - \lambda)_+$$

The least squares coefficient are translated toward zero, and truncated at zero. The fitted function is then given by the *inverse wavelet transform* $\hat{\boldsymbol{f}} = \boldsymbol{W}\hat{\boldsymbol{\theta}}$

1.6.6 Kernel Smoothing Methods

We will describe a class of regression techniques that achieve flexibility in estimating the regression function $f(X)$ over the domain $\mathbb{R}^p$ by fitting a different but simple model separately at each query point x_0. This is done by using only those observations close to the target point x_0 to fit the simple model, and in such a way that the resulting estimated function $\hat{f}(X)$ is *smooth* in $\mathbb{R}^p$
This localization is achieved via a weighting function or kernel $K_\lambda(x_0, x_i)$, which assigns a weight to x_i based on its distance from x_0.

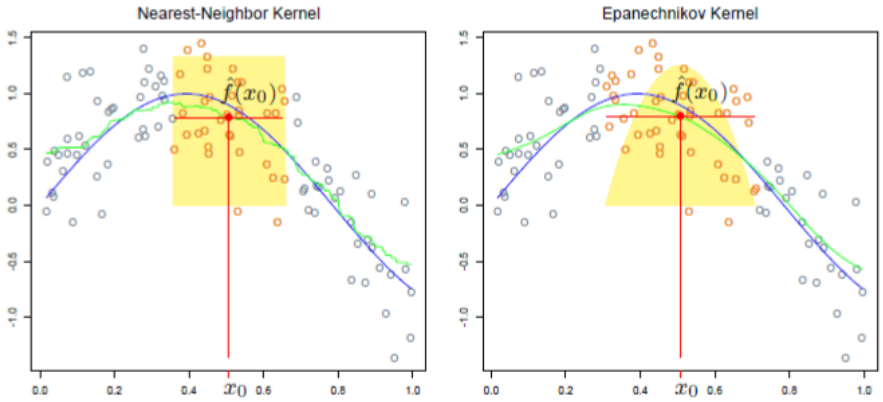


Figure 1.21: Left pannel is the result of a 30-nearest-neighbor running-mean smoother. The right panel is the result of a kernel-weighted average using *Epanechnikov* kernel with (half) window width $\lambda = 0.2$

One-Dimensional Kernel Smoother The green curve is bumpy, since $\hat{f}(x)$ is discontinuous in x. As we move x_0 from left to the right, the k-nearest

neighborhood remains constant, until a point x_i to the right of x_0 becomes closer than the furthest point $x_{i'}$ in the neighborhood to the left of x_0 at which time x_i replaces $x_{i'}$.
Rather than give all the points in the neighborhood equal weight, we can assign weights that die off smoothly with distance from the target point as the so-called Nadaraya-Watson kernel-weighted average:

$$\hat{f}(x_0) = \frac{\sum_{i=1}^{N} K_\lambda(x_0, x_i) y_i}{\sum_{i=1}^{N} K_\lambda(x_0, x_i)}$$

with the *Epanechenikov* quadratic kernel:

$$K_\lambda(x_0, x) = D\left(\frac{|x - x_0|}{\lambda}\right), \text{ with } D(t) = \begin{cases} \frac{3}{4}(1 - t^2) \text{ if } |t| \leq 1 \\ 0 \text{ otherwise} \end{cases}$$

We can use such adaptive neighborhoods with kernels, more generally:

$$K_\lambda(x_0, x) = D\left(\frac{|x - x_0|}{h_\lambda(x_0)}\right)$$

Local Linear Regression Locally-weighted averages can be badly biased on the boundaries of the domain.

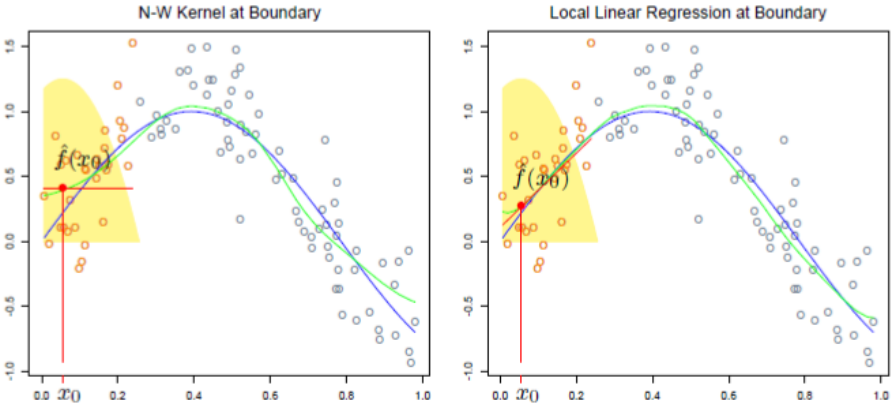


Figure 1.22: The true function is approximately linear, but most of the observations in the neighborhood have a higher mean than the target point, so despite weighting, their mean will be biased upwards. By fitting a locally weighted linear regression (right panel), this bias is removed to first order.

Locally weighted regression solves a separate weighted least squares problem at each target point x_0:

$$\min_{\alpha(x_0), \beta(x_0)} \sum_{i=1}^{N} K_\lambda(x_0, x_i)[y_i - \alpha(x_0) - \beta(x_0) x_i]^2$$

The estimate is then $\hat{f}(x_0) = \hat{\alpha}(x_0) + \hat{\beta}(x_0)x_0$.
Define the vector-valued function $b(x)^T = (1, x)$. Let $\boldsymbol{B}$ be the $N \times 2$ regression matrix with i^{th} row $b(x_i)^T$, and $\boldsymbol{W}(x_0)$ the $N \times N$ diagonal matrix with i^{th} diagonal element $K_\lambda(x_0, x_i)$ then:

$$\begin{aligned}\hat{f}(x_0) &= b(x_0)^T \left(\boldsymbol{B}^T \boldsymbol{W}(x_0) \boldsymbol{B}\right)^{-1} \boldsymbol{B}^T \boldsymbol{W}(x_0) \boldsymbol{y} \\ &= \sum_{i=1}^{N} l_i(x_0) y_i\end{aligned}$$

These weights $l_i(x_0)$ combine the weighting kernel $K_\lambda(x_0, \cdot)$ and the least squares operations and are sometimes referred to as the *equivalent kernel*.
Local linear regression *automatically* modifies the kernel to correct the bias *exactly* to first order, a phenomenon dubbed as automatic kernel carpentry.

$$\begin{aligned}\mathbb{E}\left(\hat{f}(x_0)\right) =& \sum_{i=1}^{N} l_i(x_0) f(x_i) \\ =& f(x_0) \sum_{i=1}^{N} l_i(x_0) + f'(x_0) \sum_{i=1}^{N} (x_i - x_0) l_i(x_0) + \frac{f''(x_0)}{2} \sum_{i=1}^{N} (x_i - x_0)^2 l_i(x_0) + R\end{aligned}$$

where the remainder term R involves third and higher order derivatives of f and is typically small under suitable smoothness assumptions.

Local Polynomial Regression

$$\min_{\alpha(x_0), \beta_j(x_0) | j \in [\![1,d]\!]} \sum_{i=1}^{N} K_\lambda(x_0, x_i) \left[y_i - \alpha(x_0) - \sum_{j=1}^{d} \beta_j(x_0)(x_0) x_i^j \right]^2$$

with the solution $\hat{f}(x_0) = \hat{\alpha}(x_0) + \sum_{j=1}^{d} \hat{\beta}_j(x_0) x_0^j$. In fact, an expansion such as the equation of $\mathbb{E}\left(\hat{f}(x_0)\right)$ for local linear regression will tell us that the bias will only have components of degree $d+1$ and higher.

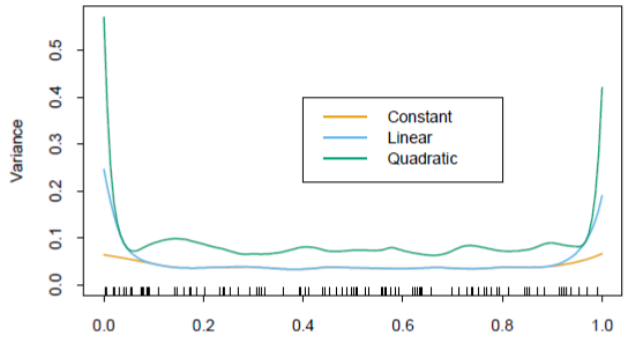


Figure 1.23: The variances functions $\|l(x)\|^2$ for local constant, linear and quadratic regression, for a metric bandwidth ($\lambda = 0.2$) tri-cube kernel

Local linear fits tend to be biased in regions of curvature of the true function, a phenomenon referred to as *trimming the hills* and *filling the valleys*. Local quadratic regression is generally able to correct this bias.

- Local linear fits can help bias dramatically at the boundaries at a mode cost in variance. Local quadratic fits do little at the boundaries for bias, but increase the variance a lot.
- Local quadratic fits tend to be most helpful in reducing bias due to curvature in the interior of the domain.

It is not recommended to move from local linear fits at the boundary to local quadratic fits in the interior. But rather to choose the degree of the fit in function of the application, if we are interested in extrapolation then the boundary is of more interest and local linear fits are probably more reliable.

Selecting the Width of the Kernel

- For the *Epanechnikov* or *tri-cube* kernel with metric width, λ is the radius of the support region
- For the *Gaussian* kernel, λ is the standard deviation
- For the *k-nearest* neighborhoods, λ is the number k of nearest neighbors, often expressed as a fraction or span $\frac{k}{N}$ of the total training sample.

There is a natural bias-variance tradeoff as we change the width of the averaging window:

- If the window is narrow, $\hat{f}(x_0)$ is an average of a small number of y_i close to x_0, and its variance will be relatively large-close that of an individual y_i.
- If the window is wide, the variance of $\hat{f}(x_0)$ will be small relative to the variance of any y_i, because of the effect of averaging.

Local Regression in $\mathbb{R}^R$ Local linear regression will fit a hyperplane locally in X, by weighted least squares, with weights supplied by a p-dimensional kernel.

Let $b(X)$ be a vector of polynomial terms in X, of maximum degree d, $\begin{cases} d = 0 \Leftarrow b(x) = 1 \\ (d,p) = (1,2) \Leftarrow b(x) = (1, X_1, X_2) \\ (d,p) = (2,2) \Leftarrow b(x) = (1, X_1, X_2, X_1^2, X_2^2, \end{cases}$

At each $x_0 \in \mathbb{R}^p$ we solve:

$$\min_{\beta(x_0)} \sum_{i=1}^{N} K_\lambda(x_0, x_1) \left(y_i - b(x_i)^T \beta(x_0)\right)^2$$

with $K_\lambda(x_0, x_1) = D\left(\frac{\|x - x_0\|}{\lambda}\right)$

Structured Local Regression Models in $\mathbb{R}^p$

Structured Kernels Let be $\boldsymbol{A}$ a semi-definite matrix to weigh the different coordinates:

$$K_{\lambda,A}(x_0, x) = D\left(\frac{(x - x_0)^T \boldsymbol{A}(x - x_0)}{\lambda}\right)$$

If $\boldsymbol{A}$ is diagonal, then we can increase or decrease the influence of individual predictors X_j by increasing or decreasing A_{jj}

Structured Regression Functions We are trying to fit a regression function $\mathbb{E}(Y|X) = f(X_1, X_2, \cdots, X_p)$ in $\mathbb{R}^p$. It is natural to consider ANOVA decomposition of the form:

$$f(X_1, X_2, \cdots, X_p) = \alpha + \sum_j g_j(X_j) + \sum_{k<l} g_{kl}(X_k, X_l) + \cdots$$

and then introduce by eliminating some of the higher-order terms. We then assume the conditionally linear model:

$$f(X) = \alpha(Z) + \beta_1(Z)X_1 + \cdots + \beta_q(Z)X_q$$

For given Z, this is a linear model, but each of the coefficient can vary with Z.

$$\min_{\alpha(Z_0),\beta(z_0)} \sum_{i=1}^{N} K_\lambda(z_0, z_i)\left(y_i - \alpha(z_0) - x_{1i}\beta_1(z_0) - \cdots - x_{qi}\beta_q(z_0)\right)$$

1.6.7 Kernel Density Estimation and Classification

Kernel density estimation is an unsupervised learning procedure.

Kernel Density Estimation Arguing as before, a natural local estimate has the form : $\hat{f}_X(x_0) = \frac{\#x_i \in \mathcal{N}(x_0)}{N\lambda}$ where $\mathcal{N}(x_0)$ is a small metric neighborhood around x_0 of width λ. This estimate is bumpy, and the smooth *Parzen* estimate is preferred:

$$\hat{f}_X(x_0) = \frac{1}{N\lambda}\sum_{i=1}^{N} K_\lambda(x_0, x_i)$$

because it counts observations close to x_0 with weights that decrease with distance from x_0. In this case a popular choice for K_λ is the Gaussian kernel $K_\lambda(x_0, x) = \phi\left(\frac{|x-x_0|}{\lambda}\right)$

Python Code

```
import sklearn
from sklearn.neighbors import KernelDensity

kde = KernelDensity(
  brandwith=0.2
  kernel='gaussian') # or 'tophat','epanechnikov','
    exponential','linear','cosine'
kde_log_density = kde.score_samples(y)
```

Kernel Density Classification Suppose for a J class problem we fit non-parametric density estimates $\hat{f}_j(X), j \in [\![1, J]\!]$ separately in each the classes, and we also have estimates of the class priors $\hat{\pi}_j$ (usually the sample propositions). Then:

$$\hat{\mathbb{P}}_{\{X=x_0\}}(G = j) = \frac{\hat{\pi}_j \hat{f}_j(x_0)}{\sum_{k=1}^{j} \hat{\pi}_k \hat{f}_k(x_0)}$$

1.6.8 Local regression

Span It plays a role like that of the tuning parameter λ in smoothing splines: it controls the flexibility of the non-linear fit.
The smaller the value of s, the more *local* will be our fit.

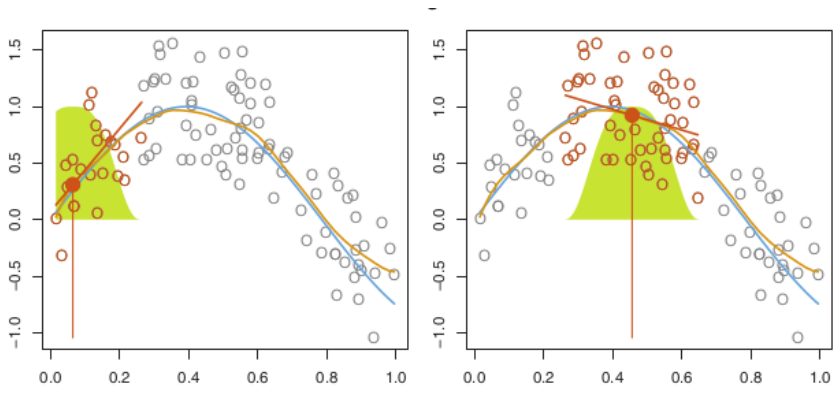


Figure 1.24: Blue curve represents $f(x)$ from which the data were generated.
Orange curve corresponds to the local regression estimate $f(x)$.
Orange colored points are local to the target point x_0.
Yellow-bell-shape superimposed on the plot indicates weights assigned to each point, decreasing to zero with distance from the target point.

Algorithm: *Local Regression* at $X = x_0$

1. Gather the fraction $s = \frac{k}{n}$ of training points whose x_i are closet to x_0
2. Assign a weight $k_{i0} = K(x_i, x_0)$ so that the point furthest from x_0 has weight zero, and the closet has the highest weight.
   All but these k nearest neighbors get weight 0
3. Fit a weighted least squares regression of the y_i on the x_i, using the aforementioned weights, by finding β_0 and β_1 that minimize:

$$\sum_{i=1}^{n} K_{i0}(y_i - \beta_0 - \beta_1 x_i)^2$$

4. The fitted value at x_0 is given by $\hat{f}(x_0) = \hat{\beta}_0 + \hat{\beta}_1 x_0$

For p-dimensional neighborhoods, local regression can perform poorly if p is much larger than 3 or 4, because there will generally be very few training observations close to x_0

1.6.9 Generalized additive models

Generalized Additive Models (GAMs) provide a genral framework for extending a standard linear model by allowing non-linear functions of each of the varialbes, while maintaining additivity.

Principle

Definition Considering we have $y_i = \beta_0 + \sum_{r=1}^{p} \beta_r x_{ir} + \epsilon$

We replace each linear component $\beta_j x_{ij}$ with a (smooth) non-linear function $f_j(x_{ij})$:

$$y_i = \beta_0 + \sum_{j=1}^{p} f_j(x_{ij}) + \epsilon_i$$

It is called an *additive model* because we calculate a separate f_i for each X_j, and then add together all of their contributions.

In the regression setting, a generalized additive model has the form: $\mathbb{E}\left(Y | (X_j)_{1\leq j\leq p}\right) = \alpha + \sum_{j=1}^{p} f_j(X_j)$, the f_j's are unspecified smooth ("non-parametric") functions.

Fitting Additive Models The additive model has the form:

$$Y = \alpha + \sum_{j=1}^{p} f_j(X_j) + \epsilon$$

where $\mathbb{E}(\epsilon) = 0$. Given observations x_i, y_i a criterion like the penalized sum of squares

$$PRSS\left(\alpha, (f_j)_{1\leq j\leq p}\right) = \sum_{i=1}^{N} \left(y_i - \alpha - \sum_{j=1}^{p} f_j(x_{ij})\right)^2 + \sum_{j=1}^{p} \lambda_j \sum f_j''(t_j)^2 dt_j$$

where the $\lambda_j \geq 0$ are the tuning parameters.

The Backfitting Algorithm for Additive Models

1. Initialize: $\forall (i,j) \in [\![1,N]\!] \times [\![1,p]\!], \hat{\alpha} = \frac{1}{N}\sum_{1}^{N} y_i, \hat{f}_j \equiv 0$

2. Cycle: $j = 1, \ldots, p, 1, \ldots, p, \ldots$

$$\hat{f}_j \leftarrow \boldsymbol{S}_j \left[\left\{ y_i - \hat{\alpha} - \sum_{k \neq} \hat{f}_k(x_{ik}) \right\}_1^N \right]$$

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

until the function $\hat{f}_j$ change less than a pre-specified threshold.

We set $\hat{\alpha} = ave(y_i)$ and it never changes. We apply a cubic smoothing spline S_j to the targets $\left\{ y_i - \hat{\alpha} - \sum_{k \neq j} \hat{f}_k(x_{ik}) \right\}_1^N$. The process is continued unitil the estimates $\hat{f}_j$ stabilize.
The backfitting procedure allow one to choose a fitting method appropriate for each input variable however it fits all predictors which is not feasible or desirable when a large number are available

```
import pygam
from pygam import LinearGAM, LogisticGAM
from pygam import GAM, l, s, f, te, intercept


# REGRESSION
df_reg = pd.read_csv('myFile_reg.txt', sep=';')
y, X = df.iloc[:, 0], df.iloc[:, 1:]
p = len(X)
lgam = LinearGAM(
    eval(' + '.join(['f({})'.format(i)
        for i in range(p-3)] +
        ['l(p-3)', 'l(p-2)', 'l(p-1)'])
        )
)

lgam_grid = lgam.gridsearch(X, y)
lgam_grid.summary()

fig, ax = plt.subplots()
ax.scatter(X.budget_std, y)
ax.plot(X.budget_std, lgam_grid.predict(X), color='red')
ax.plot(X.budget_std, lgam_grid.prediction_intervals(X)[:, 0
    ])
ax.plot(X.budget_std, lgam_grid.prediction_intervals(X)[:, 1
    ])
plt.show()



# CLASSIFICATION
# Replace LinearGAM by LogisticGAM!!
```

Pros and Cons of GAM's

+ We do not need to manually try out many different transformations on each variable individually.

+ The non-linear fits can potentially make more accurate predictions for the response Y

+ Because the model is additive, we can examine the effect of each X_j on Y individually while holding all of the other variables fixed.

+ The smoothness of the function f_j for the variable X_j can be summarized via degrees of freedom.

- The model is restricted to be additive

The Naive Bayes Classifier It is especially appropriate when dimension p of the feature space is high, making density estimation unattractive. The naive Bayes model assumes that given a class $G = j$, the features X_k are independent. While this assumption is generally not true, it does simplify the estimation dramatically, using class J as the base we can derive:

- The individual class-conditional marginal densities f_{jk} can each be estimated separately using one-dimensional kernel density estimates.

- If a component X_j of X is discrete, then an appropriate histogram estimate can be used.

Despite these rather optimistic assumptions, naive Bayes classifiers often outperform far more sophisticated alternatives. Because although the individual class density estimates may be biased this bias might not hurt the posterior probabilities as much especially near the decision regions.

$$
\begin{aligned}
\log\left(\frac{\mathbb{P}_{\{X\}}(\{G=l\})}{\mathbb{P}_{\{X\}}(\{G=J\})}\right) &= \log\left(\frac{\pi_l f_l(X)}{\pi_J f_J(X)}\right) \\
&= \log \frac{\pi_l \prod_{k=1}^{p} f_{lk}(X_k)}{\pi_J \prod_{k=1}^{p} f_{Jk}(X_k)} \quad \text{(independence assumption)} \\
&= \log\left(\frac{\pi_l}{\pi_J}\right) + \sum_{k=1}^{p} \log\left(\frac{f_{lk}(X_k)}{f_{Jk}(X_k)}\right) \\
&= \alpha_l + \sum_{k=1}^{p} g_{lk}(X_k)
\end{aligned}
$$

```
import sklearn
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB

df = pd.read_csv('myFile.csv', sep=';')
y, X = df.iloc[:, 0], df.iloc[:, 1:]

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.5, random_state=0)
gnb = GaussianNB()
y_pred = gnb.fit(X_train, y_train).predict(X_test)
print('Number of mislabeled points out of a total\
%d points: %d'
% (X_test.shape[0], (y_test != y_pred).sum()))
```

1.6.10 Radial Basis Functions and Kernels

Kernel method achieve flexibility by fitting simple models in a region local to the target point x_0. Localization is achieved via a weighting kernel K_λ, and individual observations receive weights $K_\lambda(x_0, x_1)$.
Radial basis functions combine these ideas by treating the kernel function $K_\lambda(\xi, x)$. This leads to the model:

$$\begin{aligned} f(x) =& \sum_{j=1}^{M} K_\lambda(\xi_j, x)\beta_j \\ =& \sum_{j=1}^{M} D\left(\frac{\|x - \xi_j\|}{\lambda_j}\right)\beta_j \end{aligned}$$

where each basis elements is indexed bya a location or *prototype* parameters ϵ_j and a scale parameter λ_j.

1.6.11 Model Inference and Averaging

Maximum Likelihood Inference The likelihood function can be used to assess the precision of $\hat{\theta}$. We need few more definitions.

- Score function: $\dot{l}(\theta, \boldsymbol{Z}) = \sum_{i=1}^{N} \dot{l}(\theta, z_i) = \sum_{i=1}^{N} \frac{\partial l(\theta; z_i)}{\partial \theta}$
- The *information matrix* is: $\boldsymbol{I} = -\frac{i=1}{N} \frac{\partial^2 l(\theta; z_i)}{\partial \theta \partial \theta^T}$
- *Fisher information*: $\boldsymbol{i}(\theta) = \mathbb{E}_\theta\left(\boldsymbol{I}(\theta)\right)$

Confidence points for θ_j can be constructed from either approximation: $\hat{\theta} - z^{(1-\alpha)}\sqrt{\boldsymbol{I}(\hat{\theta})^{-1}_{jj}}$ More accurate confidence intervals can be derived from the

likelihood function, by using the chi-squared approximation

$$2\left[l(\hat{\theta}) - l(\theta_0)\right] \hookrightarrow \chi^2_p$$

where p is the number of components in θ.
The maximum likelihood estimate is obtained by setting $\frac{\partial l}{\partial \beta} = \frac{\partial^2 l}{\partial \sigma^2} = 0$ giving

$$\begin{cases} \hat{\beta} = \left(\boldsymbol{H}^T \boldsymbol{H}\right)^{-1} \boldsymbol{H}^T \boldsymbol{y} \\ \sigma^2 = \frac{1}{N} \sum \left(y_i - \hat{\mu}(x_i)\right)^2 \end{cases}$$

The advantage of the bootstrap over the maximum of likelihood formula is that it allows us to compute maximum likelihood estimates of standard errors and other quantities in setting where no formulas are available.

The EM Algorithm We would like to model the density of the data points, and due to the apparent bi-modality, a Gaussian distribution would not be appropriate. $\begin{cases} Y_1 \hookrightarrow \mathcal{N}(\mu_1, \sigma_1^2) \\ Y_2 \hookrightarrow \mathcal{N}(\mu_2, \sigma_2^2) \\ Y = (1-\Delta)Y_1 + \Delta Y_2 \end{cases}$ where $\delta \in \{0, 1\}$ with $\mathbb{P}(\{\Delta = 1\}) = \pi$

$$l_0(\theta; \boldsymbol{Z}, \boldsymbol{\Delta}) = \sum_{i=1}^{N} \left[(1 - \boldsymbol{\Delta}_i) \log\left(\phi_{\theta_1}(y_i)\right) + \boldsymbol{\Delta}_i \log\left(\phi_{\theta_2}(y_i)\right)\right] + \sum_{i=1}^{N} \left[(1 - \boldsymbol{\Delta}_i) \log(1 - \pi) + \boldsymbol{\Delta}_i \log(\pi)\right]$$

Since the values of the Δ_i's are actually unknown, we proceed in an iterative fashion substituting for each Δ_i in the previous equation : $\gamma_i(\theta) = \mathbb{E}_{\theta, \boldsymbol{Z}}(\Delta_i) = \mathbb{P}_{\{\theta, \boldsymbol{Z}\}}(\{\Delta_i = 1\})$ also called the *responsibility* We use the following EM algorithm for special case of the Gaussian mixtures.

- Expectation Step: we do a soft assignment of each observation to each model: the current estimates of the parameters are used to assign responsibilities according to the relative density of the training points under each model.

- Maximization Step: These responsibilities are used in weighted maximum-likelihood fits to update the estimates of the parameters.

The EM algorithm is a popular tool for simplifying difficult maximum likelihood problems.

EM Algorithm for 2-Component Gaussian Mixture

1. Take initial guesses for the parameters $\hat{\mu}_1, \hat{\sigma}_1, \hat{\mu}_2, \hat{\sigma}_2, \hat{\pi}$

2. Excpectation Step: compute the responsibilities: $\hat{\gamma}_i = \dfrac{\hat{\pi}\phi_{\hat{\theta}_2}(y_i)}{(1-\hat{\pi})\phi_{\hat{\theta}_1}(y_i) + \hat{\pi}\phi_{\hat{\theta}_2}(y_i)}$ for $i \, [\![1, N]\!]$

3. Maximization Step: compute the weighted means and variances

$$\hat{\mu}_1 = \frac{\sum_{i=1}^{N}(1-\hat{\gamma}_i)y_i}{\sum_{i=1}^{N}(1-\hat{\gamma}_i)}, \qquad \hat{\sigma}_1 = \frac{\sum_{i=1}^{N}(1-\hat{\gamma}_i)(y_i-\hat{\mu}_1)^2}{\sum_{i=1}^{N}(1-\hat{\gamma}_i)}$$

$$\hat{\mu}_2 = \frac{\sum_{i=1}^{N}\hat{\gamma}_i y_i}{\sum_{i=1}^{N}\hat{\gamma}_i}, \qquad \hat{\sigma}_2 = \frac{\sum_{i=1}^{N}\hat{\gamma}_i(y_i-\hat{\mu}_2)^2}{\sum_{i=1}^{N}\hat{\gamma}_i}$$

4. Iterate steps 2 and 3 until convergence

and the mixing probability $\hat{\pi} = \sum_{i=1}^{N}\frac{\hat{\gamma}_i}{N}$

```
import sklearn
from sklearn.mixture import GaussianMixture

df = pd.read_csv('myFile.csv', sep=';')
y, X = df.iloc[:, 0], df.iloc[:, 1:]

model = GaussianMixture(n_components=2,
    init_params='random')
model.fit()
y_pred = model.predict(X)
```

The EM Algorithm in General Our observed data is $\boldsymbol{Z}$ having log-likelihood $l(\theta, \boldsymbol{Z})$. The latent or missing data is $\boldsymbol{Z}^m$, so that the complete data is $\boldsymbol{T} = (\boldsymbol{Z}, \boldsymbol{Z}^m)$ with log-likelihood $l_0(\theta, \boldsymbol{T})$. In the mixture problem $(\boldsymbol{Z}, \boldsymbol{Z}^m) = (\boldsymbol{y}, \boldsymbol{\Delta})$ Since $\mathbb{P}_{\{\boldsymbol{Z},\theta'\}}(\{\boldsymbol{Z}^m\}) = \frac{\mathbb{P}_{\{\theta'\}}(\{\boldsymbol{Z}^m, \boldsymbol{Z}\})}{\mathbb{P}_{\{\theta'\}}(\{\boldsymbol{Z}\})}$ we can write: $\mathbb{P}_{\{\theta'\}}(\{\boldsymbol{Z}\}) = \frac{\mathbb{P}_{\{\theta'\}}(\{\boldsymbol{T}\})}{\mathbb{P}_{\{\boldsymbol{Z},\theta'\}}(\{\boldsymbol{Z}^m\})}$. In terms of log-likelihoods, we have:

$$l(\theta'; \boldsymbol{Z}) = l_0(\theta'; \boldsymbol{T}) - l_1(\theta'; \boldsymbol{Z}^m | bmZ)$$

where l_1 is based on the conditional density: $\mathbb{P}_{\{\boldsymbol{Z},\theta'\}}(\{\boldsymbol{Z}^m\})$

1. Start with initial guesses for the parameters $\hat{\theta}^{(0)}$
2. Expectation Step: at the j^{th} step, compute

$$Q(\theta', \hat{\theta}^{(j)}) = \mathbb{E}\left(l_0(\theta' : \boldsymbol{T}) | \boldsymbol{Z}, \hat{\theta}^{j)}\right)$$

3. Maximization Step: determine the new estimate $\hat{\theta}^{(j+1)}$ as the maximizer of $Q\left(\theta', \hat{\theta}^{(j)}\right)$ over θ'
4. Iterate steps 2 and 3 until convergence

In the M step, the EM algorithm maximizes $Q(\theta', \theta)$ over θ' rather than actual objective function $l(\theta' : \boldsymbol{Z})$

EM as a Maximization Procedure Here is a different view of the EM procedure, as a joint maximization algorithm. Consider function:

$$F(\theta', \tilde{P}) = \mathbb{E}_{\tilde{P}}\left(l_0(\theta' : \boldsymbol{T})\right) - \mathbb{E}_{\tilde{P}}\left(\tilde{P}(\boldsymbol{Z}^m)\right)$$

Here $\tilde{P}(\boldsymbol{Z}^m)$ is any distribution over the latent data $\boldsymbol{Z}^m$

The function F expands the domain of the log-likelihood, to facilitate its maximization. The maximizer over

$$\tilde{P}(\boldsymbol{Z}) = \mathbb{P}_{\{\boldsymbol{Z},\theta'\}}\left(\{\boldsymbol{Z}\}\right)$$

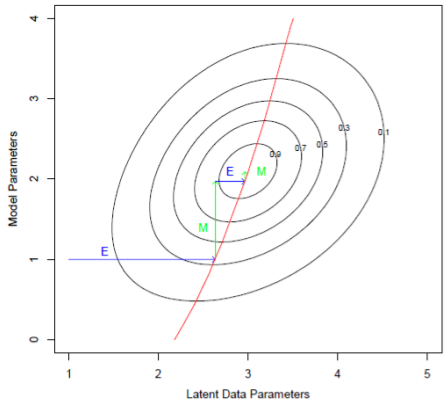


Figure 1.25: The contours of the (augmented) observed data log-likelihood $F(\theta, \tilde{P})$ The E step is equivalent to maximizing the log-likelihood over the parameters of the latent data distribution. The M step maximizes it over the parameters of the log-likelihood. The red curve corresponds to the observed data log-likelihood a profile obtained by maximizing $F(\theta', \tilde{P})$ for each value of θ'

Gibbs Sampler

1. Take some initial values $U_k^{(0)}$ for $k \in [\![1, K]\!]$
2. Repeat for $t \in [\![1, ?]\!]$: For $k \in [\![1, K]\!]$ generate $U_k^{(t)}$ from
$$\Pr(U_k^{(t)} | U_1^{(t)}, \cdots U_{k-1}^{(t)}, U_{k+1}^{(t-1)}, \cdots U_K^{(t-1)})$$
3. Continue step 2 until the joint distribution of $\left(U_j^{(t)}\right)_{1 \leq j \leq K}$

MCMC (Markov chain Monte Carlo) for Sampling from the Posterior Gibbs sampling is an MCMC procedure that is closely related to the EM Algorithm: the main difference is that is it samples from the conditional distributions rather than maximizing over them.
More formally, Gibbs sampling produces a Markov chain whose stationary distribution is the true joint distribution, and hence the term of "Markov Chain Monte Carlo"

Gibbs sampling for mixtures

1. Take some initial values: $\theta^{(0)} = (\mu_1^{(0)}, \mu_2^{(0)})$
2. Repeat for $t \in [\![1, ?]\!]$:
   (a) For $i \in [\![1, N]\!]$ generate $\mathbf{\Delta}_i^{(t)} \in \{0, 1\}$ with $\mathbb{P}\left(\left\{\mathbf{\Delta}_i^{(t)} = 1\right\}\right) = \hat{\gamma}_i(\theta^{(t)})$
   (b) Set

$$\hat{\mu}_1 = \frac{\sum_{i=1}^{N}\left(1 - \mathbf{\Delta}_i^{(t)}\right) y_i}{\sum_{i=1}^{N}\left(1 - \mathbf{\Delta}_i^{(t)}\right)}$$

$$\hat{\mu}_2 = \frac{\sum_{i=1}^{N}\mathbf{\Delta}_i^{(t)} y_i}{\sum_{i=1}^{N}\mathbf{\Delta}_i^{(t)}}$$

   and generate $\mu_1^{(t)} \hookrightarrow \mathcal{N}(\hat{\mu}_1, \hat{\sigma}_1^2)$ and $\mu_1^{(t)} \hookrightarrow \mathcal{N}(\hat{\mu}_2, \hat{\sigma}_2^2)$
   (c) Continue until the joint distribution of $\left(\Delta^{(t)}, \mu_1^{(t)}, \mu_2^{(t)}\right)$ does not change.

Bagging We show how to use the bootstrap to improve the estimate or prediction itself.
For each bootstrap $\boldsymbol{Z}^{*b}$ for $b \in [\![1, B]\!]$ we fit our model, giving prediction $\hat{f}^{*b}(x)$. The bagging estimate is defined by:

$$\hat{f}_{bag}(x) = \frac{1}{B}\sum_{b=1}^{B}\hat{f}^{*b}(x)$$

It is a Monte Carlo estimate of the true bagging estimate, approaching it as $B \to \infty$.o Suppose ξ

Model Averaging and Stacking The posterior distribution of ξ is

$$\mathbb{P}_{\{\boldsymbol{Z}\}}(\{\xi\}) = \sum_{m=1}^{M}\mathbb{P}_{\{M_m, \boldsymbol{Z}\}}(\{\xi\})\,\mathbb{P}_{\{\boldsymbol{Z}\}}(\{M_m\})$$

with posterior $\mathbb{E}(\xi|\boldsymbol{Z}) = \sum_{m=1}^{M}\mathbb{E}(\xi|M_m, \boldsymbol{Z})\,\mathbb{P}_{\{\boldsymbol{Z}\}}(\{M_m\})$. This Bayesian prediction is a weighted average of the individual predictions with weights proportional to the posterior probability of each model. This formulation leads to a number of different model-averaging strategies.

Stochastic Search: Bumping *Bumping* uses bootstrap sampling to move randomly through model space. In detail we draw bootstrap samples $\left(\boldsymbol{Z}^{*j}\right)_{1\leq j\leq B}$ and fit our model to each giving predictions $\hat{f}^{*b}(x), b \in [\![1, B]\!]$. We then choose the model that produces the smallest prediction error, averaged over the *original training set*.

$$\hat{b} = \min_b \sum_{i=1}^{N} \left[y_i - \hat{f}^{*b}(x_i)\right]^2$$

1.7 Tree-based-methods

1.7.1 The basics of decision trees

Regression Trees

Predication via Stratification of the feature space

- We divide the predictor space (the set of possible values for $(X_j)_{1\leq j\leq p}$) into J distinct and non overlapping regions $(R_k)_{1\leq k\leq J}$.
- For every observation that falls into the region R_j we make the same prediction, which is simply the mean of the response values for the training observation in R_j

Theoretically the regions could have any shape, however we choose to divide the predictor space into high-dimensional rectangles (boxes). The goal is to find $(R_k)_{1\leq k\leq J}$ that minimize the RSS given by :

$$\sum_{k=1}^{J}\sum_{i\in R_k} (y_i - \hat{y}_{R_k})^2$$

where $\hat{y}_R$, is the mean response for the training observations within the k^{th} box. It is computationally infeasible to consider every possible partition of the feature space into J boxes.
For this reason, we take an approach which is:

- ***top-down***: it begins at the top of the tree (at which point all observations belong to a single region) and then successively splits the predictor space.
- ***greedy***: at each step of the tree-building process, the best split is made at that particular step

It is known as *recursive binary splitting*

We select the predictor X_j and the cutpoint s such that splitting the predictor space into the regions $\{X|X_j < s\}$ ("*the region of predictor space in which X_j takes on a value less than s*") and $\{X|X_j > s\}$ leads to the greatest possible reduction in RSS. For any j and s we define the pair of half-planes:

$$\begin{cases} R_1(j,s) = \{X|X_j < s\} \\ R_2(j,s) = \{X|X_j > s\} \end{cases}$$

and we seek the value of j and s that minimize:

$$\sum_{i:x_i \in R_1(j,s)} (y_i - \hat{y}_{R_1})^2 + \sum_{i:x_i \in R_2(j,s)} (y_i - \hat{y}_{R_2})^2$$

where $\hat{y}_{R_1} = ave\,(y_i | x_i \in R_1(j,s))$ is the mean response for the training observations in $R_1(j,s)$

Tree Pruning A smaller tree with fewer splits (fewer regions $(R_i)_{1 \leq i \leq J}$) might lead to lower variance and better interpretation at the cost of a little bias. A better strategy is to grow a very large tree T_0, and then *prune* it back in order to obtain a *subtree*.
Rather than considering every possible subtree, we consider a sequence of trees indexed by a nonnegative tuning parameter α.

Algorithm: Building a Regression Tree

1. Use recursive binary splitting to grow a large tree on the training data, stopping only when each terminal node has fewer than some minimum number of observations
2. Apply cost complexity pruning to the large tree in order to obtain a sequence of subtrees, as a function of α
3. Use K-fold cross-validation to choose α. That is divide the training observations into K-folds.
   A Repeat steps 1 and 2 on all but k^{th} fold of the training data.
   B Evaluate the mean squared prediction error on the data in the left-out k^{th}, as a function of α

   Average the results for each value of α, and pick α to minimize the average error.
4. Return the subtree from Step 2 that correspond to the chosen value of α

For each value of α there corresponds a subtree $T \subset T_0$ such that:

$$\sum_{m=1}^{|T|} \sum_{i:x_i \in R_m} (y_i - \hat{y}_{Rm})^2 + \alpha |T|$$

is as small as possible. $|T|$ indicates the number of terminal nodes of the tree T.

```
import sklearn
from sklearn import tree

X = [[0, 0], [2, 2]]
y = [0.5, 2.5]
reg = tree.DecisionTreeRegressor()
reg = reg.fit(X, y)
print(reg.score(X, y))
```

Classification Trees The classification error rate is simply the fraction of the training observations in that region that do not belong to the most common class: $E = 1 - \max_k(\hat{p}_{mk})$ Here $\hat{p}_{mk}$ represents the proportion of training observation in the m^{th} region are from the k^{th} class, $\hat{p}_{mk} = \frac{1}{N_m} \sum_{x_i \in R_m} I(y_i = k)$.

It turns out that classification error is not sufficiently sensitive for tree-growing, and in practice 2 other measures are preferable.
The Gini index is defined by

$$G = \sum_{k=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk})$$

The entropy given by: $D = -\sum_{k=1}^{K} \hat{p}_{mk} \ln(\hat{p}_{mk})$

The entropy will take on a value near zero if the $\hat{p}_{mk}$'s are all near zero or near one.
Therefore, like the Gini index, the entropy will take on a small value if the m^{th} node is pure.

Any of these three approaches might be used when *pruning* the tree, but the classification error rate is preferable if prediction accuracy of the final pruned tree is the goal.

```
import sklearn
from sklearn import tree

X, y = load_iris(return_X_y=True)
clf = tree.DecisionTreeClassifier()
clf = clf.fit(X, y)
print(clf.score(X, y))
tree.plot_tree(clf)
```

Advantages and Disadvantages of Trees

+ They are very easy to explain to people. easily interpreted.
+ Trees can be displayed graphically, and are easily interpreted
+ Trees can easily handle qualitative predictors without the need to create dummy variables.
- Trees generally do not have the same level of predictive accuracy as some of the other regression and classification approaches.
- Trees can be non-robust, a small change in the data can cause a large change in the final estimated tree.

PRIM: Bump Hunting Tree-based methods (for regression) partition the feature space into box-shaped regions, to try to make the response averages in each box as different as possible.
The Patient Rule Introduction Method (PRIM) also finds boxes in the feature space, but seeks boxes in which the response average is high. Hence it looks for maxima in the target function. PRIM also differs from tree-based partitioning methods in that the box definition are not described by a binary tree. This makes interpretation of the collection of rules more difficult; however, by removing the binary tree constraint, the individual rules are often simpler.

Example There are 200 data points uniformely distributed over the unit square. The color-coded plot indicates the response $Y = \begin{cases} 1 \text{ (red)}, & 0.5 < X_1 < 0.8 \text{ and } 0.4 < X_2 < 0.6 \\ 0 \text{ (blue)}, & \text{otherwise} \end{cases}$
The panel shows the successive boxes found by the top-down peeling procedure, peeling off a proportion $\alpha = 0.1$ of the remaining data points at each stage.
After the top-down sequence is computed, PRIM reverses the process, expanding along any edges, if such an expansion increases the box mean, this called *pasting*.

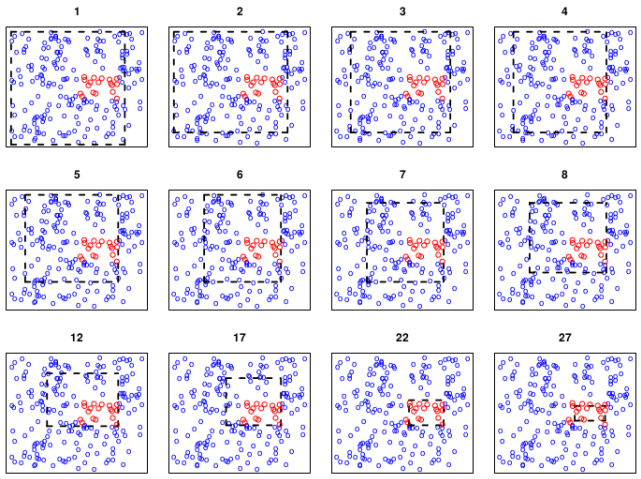


Figure 1.26: The procedure starts with a rectangle (broken black lines) surrounding all of the data, and then peels away points along one edge by a prespecified amout in order to maximize the mean of the points remaining in the box. The iteration number is indicated at the top of each panel.

Patient Rule Induction Method

1. Start with all of the training data, and a maximal box containing all of the data

2. Consider shrinking the box by compressing one face, so as to peel off the proportion α of observations having either the highest values of a predictor X_j or the lowest. Choose the peeling that produces the highest response mean in the remaining box. (Typically $\alpha = 0.05$ or 0.10)

3. Repeat step 2 until some minimal number of observation remain in the box, (say 10).
4. Expand the box along any face, as long as the resulting box mean increases.
5. Step 1-4 give a sequence of boxes, with different numbers of observation in each box. Use cross-validation to choose a member of the sequence. Call the box B_1
6. Remove the data in box B_1 from the dataset and repeat steps $2-5$ to obtain a second box, and continue to get as many boxes as desired.

This produces a sequence of boxes $(1_l)_{1\leq l\leq k}$, each box is defined by a set of rules involving a subset of predictors like: $\begin{cases} a_1 \leq X_1 \leq b_1 \\ a_3 \leq X_3 \leq b_3 \end{cases}$

MARS: Multivariate Adaptive Regression Splines MARS is an adaptive procedure for regression, and is well suited for high-dimensional problems. MARS uses expansions in piecewise linear basis functions of the form $(x-t)_+$, $(t-x)_+$

$$(x-t)_+ = \begin{cases} x-t \Leftarrow x > t \\ 0 \Leftarrow x \leq t \end{cases} \quad \text{and } (t-x)_+ = \begin{cases} t-x \Leftarrow x < t \\ 0 \Leftarrow x \geq t \end{cases}$$

Therefore, the collection of basis function is:

$$\mathcal{C} = \left\{ (X_j - t)_+, (t - X_j)_+ | \in \{x_{ij}\}_{1\leq i\leq N} \right\}_{1\leq j\leq p}$$

Thus the model has the form

$$f(X) = \beta_0 + \sum_{m=1}^{M} \beta_m h_m(X)$$

where each $h_m(X)$ is a function in $\mathcal{C}$ or a product of 2 or more such functions. Given a choice for the h_m, the coefficient β_m are estimated by minimizing the residual sum-of-squares, that is, by standard linear regression.
At each stage we consider as a new basis function pari all products of a function h_m in the model set $\mathcal{M}$ with one of the reflected pairs in $\mathcal{C}$
We add to the model $\mathcal{M}$ the term of the form:

$$\hat{\beta}_{M+1} h_l(X) \times (X_j - t)_+ + \hat{\beta}_{M+2} h_l(X) \times (t - X_j)_+, h_l \in \mathcal{M}$$

that produces the largest decrease in training error. Here $\hat{\beta}_{M+1}$ and $\hat{\beta}_{M+2}$ are coefficients estimated by least squares, along with all other $M+1$ coefficient in the model.

```
import pyearth
from pyearth import Earth

model_mars = Earth()
model_mars.fit(X, y)
```

Example At the first stage we consider adding to the model a function of the form $\beta_1 (X_j - t)_+ + \beta_2 (t - X_j)_+ ; t \in \{x_{ijj}\}$ since multiplication by the constant function just produces the functdion itself.
Suppose the best choice is $\hat{\beta}_1 (X_2 - x_{72})_+ + \hat{\beta}_2 (x_{72} - X_2)_+$.
Then this pair of basis functions is added to the set $\mathcal{M}$ and the next stage we consider including a pair of products the form: $h_m(X) \times (X_j - t)_+$ and $h_m(X) (t - X_j)_+ , t \in \{x_{ij}\}$ where we have the choices:

$$\begin{aligned} h_0(X) &= 1 \\ h_1(X) &= (X_2 - x_{72})_+ \\ h_0(X) &= (x_{72} - X_2) \end{aligned}$$

Cross-validation One could use cross-validation to estimate the optimal value of λ, but for computational savings the MARS procedure instead uses generalized cross-validation:

$$GCV(\lambda) = \frac{\sum_{i=1}^{N} \left(y_i - \hat{f}_\lambda(x_i) \right)^2}{\left(1 - \frac{M(\lambda)}{N} \right)^2}$$

The $M(\lambda)$ is the effective number of parameters in the model: this accounts both for the number of parameters used in selecting the optimal positions of the knots.

1.7.2 Bagging, RandomForest and Boosting

Bagging

Bootstrap aggregation (bagging) definition It is a general-purpose procedure for reducing the variance of a statistical learning method; we introduce it here because it is particularly useful and frequently used in the context of decision trees.
In other words for estimating $\hat{f}(x)$, the prediction at input x, we could calculate $\left(f^i(x) \right)_{1 \leq i \leq B}$ using B separate training sets, and average them in order to obtain a single low-variance statistical learning model.
We then train our method on the b^{th} bootstrapped training set in order to get $\hat{f}^{*b}(x)$, and finally average all the predictions to obtain:

$$\hat{f}_{bag}(x) = \frac{1}{B} \sum_{b=1}^{B} \hat{f}^{*b}(x)$$

This is called bagging.

Python Code

```
import pandas as pd
import sklearn
from sklearn import tree
from sklearn.ensemble import BaggingClassifier,
```

```
    BaggingRegressor

y, X = df.iloc[:, 0], df.iloc[:, 1:]
bagging = BaggingClassifier(tree.DecisionTreeClassifier(),
    max_samples=0.5, max_features=0.5)
```

Out-of-Bag Error Estimation The key to bagging is that trees are repeatedly fit to bootstrapped subsets of the observations.

Random Forest When building these decision trees, each time a split in a tree is considered, a random sample of m predictors is chosen as split candidates from the full set of p predictors.
In building a random forest, at each split in the tree the algorithm is *not even allowed to consider* a majority of the available predictors.

Random Forest for Regression or Classification

1. For $b \in [\![1, B]\!]$
   (a) Draw a bootstrap sample $\boldsymbol{Z}^*$ of size N from the training data
   (b) Grow a random-forest tree T_b to the bootstrapped data, by recursively repeating the following steps for each terminal mode of the tree, until the minimum node size n_{min} is reached.
      i. Select m variables at random from the p variables
      ii. Pick the best variable/split-point among the m
      iii. Split the node into 2 daughter nodes.
2. Output the ensemble of trees $\{T_b\}_1^B$

Regression: $\hat{f}_{rf}^B(x) = \frac{1}{B}\sum_{b=1}^{B} T_b(x)$

Classification: Let $\hat{C}_b(x)$ be the class prediction of the b^{th} random-forest tree, then $\hat{C}_{rf}^B(x) = majority\_vote\left\{\hat{C}_b(x)\right\}_1^B$

```
import pandas as pd
import sklearn
from sklearn.ensemble import RandomForestClassifier,
    RandomForestRegressor

y, X = df.iloc[:, 0], df.iloc[:, 1:]
clf = RandomForestClassifier(n_estimator=10)
```

Boosting

Boosting for Regression Trees The motivation for boosting was a procedure that combines the outputs of many "weak" classifiers to produce a powerful "committee"'. A weak classifier is one whose error rate is only slightly better than random guessing.

1. Set $\hat{f}(x) = 0$ and $r_i = y_i$ for all i in the training set.
2. For $b \in [\![1, B]\!]$ repeat:
   a Fit a tree $\hat{f}^b$ with d splits ($d+1$ terminal nodes) to the training data (X,r)
   b Update $\hat{f}$ by adding in a shrunken version of the new tree:
   $$\hat{f}(x) \leftarrow \hat{f}(x) + \lambda \hat{f}^b(x)$$
   c Update the residuals, $r_i \leftarrow r_i - \lambda \hat{f}^b(x_i)$
3. Output the boosted model, $\hat{f}(x) = \sum_{b=1}^{B} \lambda \hat{f}^b(x)$

Boosting methods Consider a two-class problem, with the output variable coded as $Y \in \{-1, 1\}$. Given a vector of predictor variables X, a classifier $G(X)$ produces a prediction taking one of the 2 values $\{-1, 1\}$

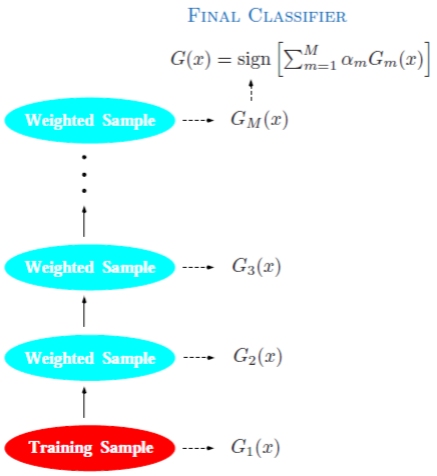


Figure 1.27: Schematic of AdaBoost. Classifiers are trained on weighted versions of the dataset, and then combined to produce a final prediction.

$$G(x) = sign\left(\sum_{m=1}^{M} \alpha_m G_m(x)\right)$$

Here α_m's are computed by the boosting algorithm and weight the contribution of each respective $G_m(x)$. Their effect is to give higher influence to the more accurate classifiers in the sequence.

AdaBoost.M1

1. Initialize the observation weights $w_i = \frac{1}{N}, i \in [\![1, N]\!]$
2. For $m \in [\![1, M]\!]$:
   a Fit a classifier $G_m(x)$ to the training data using weights w_i
   b Compute:
   $$err_m = \frac{\sum_{i=1}^{N} \omega_i I\left(y_i \neq G_m(x_i)\right)}{\sum_{i=1}^{N} \omega_i}$$
   c Compute $\alpha_m = \log\left(\frac{1 - err_m}{err_m}\right)$
   d Set $\omega_i \leftarrow \omega_i e^{\alpha_m I(y_i \neq G_m(x_i))}$ for $i \in [\![1, N]\!]$
3. Output $G(x) = sign\left[\sum_{m=1}^{M} \alpha_m G_m(x)\right]$

```
import pandas as pd
import sklearn
from sklearn.ensemble import AdaBoostClassifier,
    AdaboostRegressor
from sklearn.model_selection import corss_val_score

y, X = df.iloc[:, 0], df.iloc[:, 1:]
clf = AdaBoostClassifier(n_estimator=100)
scores = corss_val_score(clf, X, y, cv=5)
```

Boosting Fits an Additive Model

$$f(x) = \sum_{m=1}^{M} \beta_m b(x; \gamma_m)$$

where β_m's are the expansion coefficients and $b(x; \gamma) \in \mathbb{R}$ are usually simple functions of the multivariate argument x, characterized by a set of parameters γ

Forward Stagewise Additive Modeling

1. Initialize $f_0(x) = 0$
2. For $m \in [\![1, M]\!]$
   a Compute $(\beta_m, \gamma_m) = \min_{\beta, \gamma} \sum_{i=1}^{N} L(y_i, f_{m-1}(x_i) + \beta b(x_i; \gamma))$
   b Set $f_m(x) = f_{m-1}(x) + \beta_m b(x; \gamma_m)$

Forward Stagewise Additive Modeling At each iteration m, one solves the optimal basis function $b(x;\gamma_m)$ and corresponding coefficient β_m to add the current expansion $f_{m-1}(x)$. This procedure $f_m(x)$, and the process is repeated. For the squared-error loss: $L(y, f(x)) = (y - f(x))^2$ one has

$$\begin{aligned} L(y_i, f_{m-1}(x_i) + \beta b(x_i;\gamma)) &= (y_i - f_{m-1}(x) - \beta b(x_i;\gamma)) \\ &= (r_{im} - \beta b(x_i,\gamma))^2 \end{aligned}$$

where $r_{im} = y_i - f_{m-1}(x_i)$ is simply the residual of the current model on the i^{th} observation.

Loss Functions and Robustness

Robust Loss Functions for Classification The minimizer of the corresponding risk on the population is :

$$f^*(x) = \min_{f(x)} \mathbb{E}_{Y|x} (Y - f(x))^2 = \mathbb{E}(Y|X) = 2\mathbb{P}_{\{x\}} (\{Y = 1\}) - 1$$

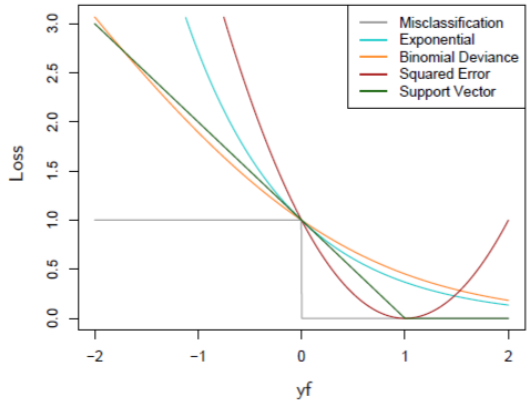


Figure 1.28: The response is $y = \pm 1$ the prediction is f, with class prediction $sign(f)$. The losses are misclassification: $I(sign(f) \neq y)$; exponential: $exp(-yf)$; binomial deviance $\log(1 + e^{-2yf})$; squared error $(y - f)^2$; and support vector: $(1 - yf)_+$

```
import pandas as pd
import sklearn
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.model_selection import train_test_split

y, X = df.iloc[:, 0], df.iloc[:, 1:]
X_train, X_test, y_train, y_test = train_test_split(
    X, y, random_state=0)
clf = GradientBoostingClassifier(
   loss = 'deviance', # {'deviance', 'exponential'}
```

```
    random_state = 0
)
print(clf.score(X_test, y_test))
```

Robust Loss Functions for Regression One such criterion is the Huber loss criterion used for M-regression

$$L(y, f(x)) = \begin{cases} (y - f(x))^2 \Leftarrow |y - f(x)| \leq \delta \\ 2\delta|y - f(x)| - \delta^2 \Leftarrow |y - f(x)| > \delta \end{cases}$$

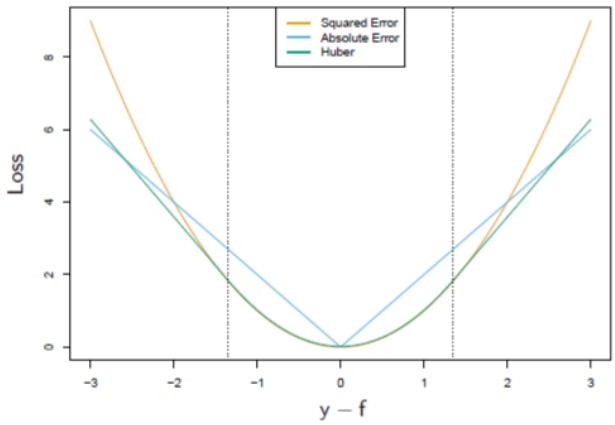


Figure 1.29: The Huber loss function combines the good properties of squared-error loss near zero and absolute error loss when $|y - f|$ is large

```
import pandas as pd
import sklearn
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.model_selection import train_test_split

y, X = df.iloc[:, 0], df.iloc[:, 1:]
X_train, X_test, y_train, y_test = train_test_split(
    X, y, random_state=0)
reg = GradientBoostingRegressor(
    loss = 'huber', # {'ls', 'lad', 'huber'}
    random_state = 0
)
print(reg.score(X_test, y_test))
```

Characteristic	Neural Nets	SVM	Trees	MARS	k-NN, Kernels
Natural handling of data of "mixed" type	▼	▼	▲	▲	▼
Handling of missing values	▼	▼	▲	▲	▲
Robustness to outliers in input space	▼	▼	▲	▼	▲
Insensitive to monotone transformations of inputs	▼	▼	▲	▼	▼
Computational scalability (large N)	▼	▼	▲	▲	▼
Ability to deal with irrelevant inputs	▼	▼	▲	▲	▼
Ability to extract linear combinations of features	▲	▲	▼	▼	◆
Interpretability	▼	▼	◆	▲	▼
Predictive power	▲	▲	▼	◆	▲

Figure 1.30: Some characteristics of different learning methods.Green=good, Yellow=fair, Red=poor

Procedures for Data Mining

Boosting Trees Regression and classification trees partition the space of all joint predictor variable values into disjoint regions $R_j, j \in [\![1, J]\!]$ as represented by the terminal nodes of the tree.
A constant γ_j is assigned to each such region and the predictive rule is : $x \in R_j \Rightarrow f(x) = \gamma_j$. Thus a tree can be formally expressed as:

$$T(x; \Theta) = \sum_{j=1}^{J} \gamma_j I(x \in R_J)$$

with parameters $\Theta = \{R_j, \gamma_j\}_1^J$ The parameters are found by minimizing the empirical risk: $\Theta = \min_{\Theta} \sum_{j=1}^{J} \sum_{x_i \in R_j} L(y_i, \gamma_j)$

Numerical Optimization via Gradient Boosting The goal is to minimize $L(f) = \sum_{i=1}^{N} L(y_i, f(x_i))$ with respect to f, where here $f(x)$ is constrained to be a sum of trees. Ignoring this constraint, minimizing $L(f)$ can be viewed as a numerical optimization:

$$\hat{\boldsymbol{f}} = \min_{f} L(\boldsymbol{f})$$

where the parameters $\boldsymbol{f} \in \mathbb{R}^N$ are the values of the approximating function $f(x_i)$ such as $\boldsymbol{f} = (f(x_i))_{1 \leq i \leq N}^T$ Numerical optimization procedures solves the previous equation as a sum of component vectors:

$$\boldsymbol{f}_M = \sum_{m=0}^{M} \boldsymbol{h}_m, \boldsymbol{h}_m \in \mathbb{R}^N$$

Steepest Descent Steepest descent chooses $\boldsymbol{h}_m = -\rho_m \boldsymbol{g}_m, (\rho_m, \boldsymbol{g}_m) \in \mathbb{R} \times \mathbb{R}^N$, $\boldsymbol{g}_m$ is the gradient of $L(\boldsymbol{f})$ evaluated at $\boldsymbol{f} = \boldsymbol{f}_{m-1}$. The components of the gradient $\boldsymbol{g}_m$ are:

$$g_{im} = \left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f(x_i = f_{m-1}(x_1))}$$

The step length ρ_m is the solution to $\rho_m = \min_{\rho} L\left(\boldsymbol{f}_{m-1} - \rho \boldsymbol{g}_m\right)$, the current solution is then updated

$$\boldsymbol{f}_m = \boldsymbol{f}_{m-1} - \rho_m \boldsymbol{g}_m$$

Setting	Loss Function	$-\partial L(y_i, f(x_i))/\partial f(x_i)$
Regression	$\frac{1}{2}[y_i - f(x_i)]^2$	$y_i - f(x_i)$
Regression	$\lvert y_i - f(x_i)\rvert$	$\text{sign}[y_i - f(x_i)]$
Regression	Huber	$y_i - f(x_i)$ for $\lvert y_i - f(x_i)\rvert \leq \delta_m$ $\delta_m \text{sign}[y_i - f(x_i)]$ for $\lvert y_i - f(x_i)\rvert > \delta_m$ where $\delta_m = \alpha$th-quantile$\{\lvert y_i - f(x_i)\rvert\}$
Classification	Deviance	kth component: $I(y_i = \mathcal{G}_k) - p_k(x_i)$

Figure 1.31: Gradients for commonly used loss functions

Gradient Tree Boosting Algorithm

1. Initialize $f_0(x) = \min_{\gamma} \sum_{i=1}^{N} L(y_i, \gamma)$
2. For $m \in [\![1, M]\!]$:
   (a) For $i \in [\![1, N]\!]$ compute: $r_{im} = -\left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f=f_{m-1}}$
   (b) Fit a regression tree to the targets r_{im} giving terminal regions $R_{jm}, j \in [\![1, J_m]\!]$
   (c) For $j \in [\![1, J_m]\!]$ compute: $\gamma_{jm} = \min_{\gamma} \sum_{x_i \in R_{jm}} L(y_i, f_{m-1}(x_i) + \gamma)$

(d) Update $f_m(x) = f_{m-1}(x) + \sum_{j=1}^{J_m} \gamma_{jm} I(x \in \mathbb{R}_{jm})$

3. Output $\hat{f}(x) = f_M(x)$

1.8 Support vector machines

1.8.1 Maximal margin classifier

Definition It is the separating hyperplane that is farthest from the training observations. We can compute the (perpendicular) distance from each training observation to a given separating hyperplane; the smallest such distance is the minimal distance from the observations to the hyperplane, and is known as the margin.
In a sense, the maximal margin hyperplane represents the mid-line of the widest "slab" that we can insert between 2 classes.

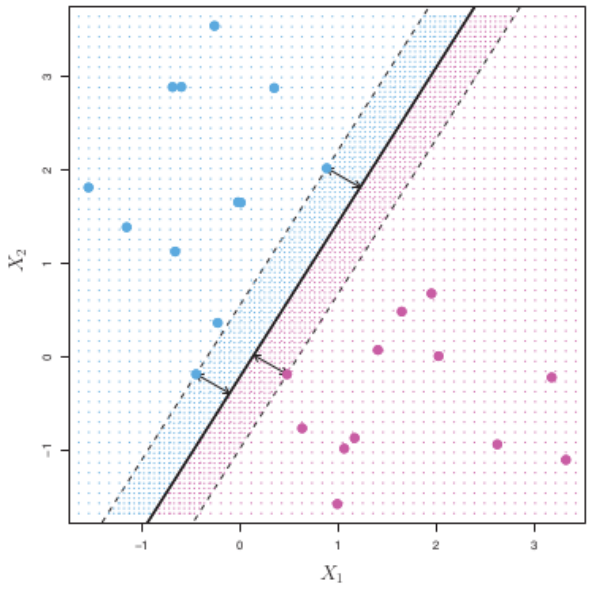


Figure 1.32: The margin is distance from the solid line to either of the dashed lines. The 2 points and the purple point that lie on the dashed lines are the support vectors

Construction of the Maximal Margin Classifier Set of n observations $(x_1 ... x_p) \in \mathbb{R}^p$ and associated class labels $(y_i)_{1 \leq i \leq n} \in \{-1, 1\}$ The maximal margin hyperplane is the solution to the optimization problem:

$$\max_{(\beta_i)_{0\leq i\leq p},M} M \text{ subject to: } \begin{cases} \sum_{j=1}^{p}\beta_j^2 = 1 \ (1) \\ \forall i \in [\![1,n]\!],\ y_i(\beta_0 + \sum_{j=1}^{p}\beta_j x_{ij}) > M \ (2) \end{cases}$$

(2) guarantees that each observation will be on the correct side of the hyperplane, provided that $M \geq 0$

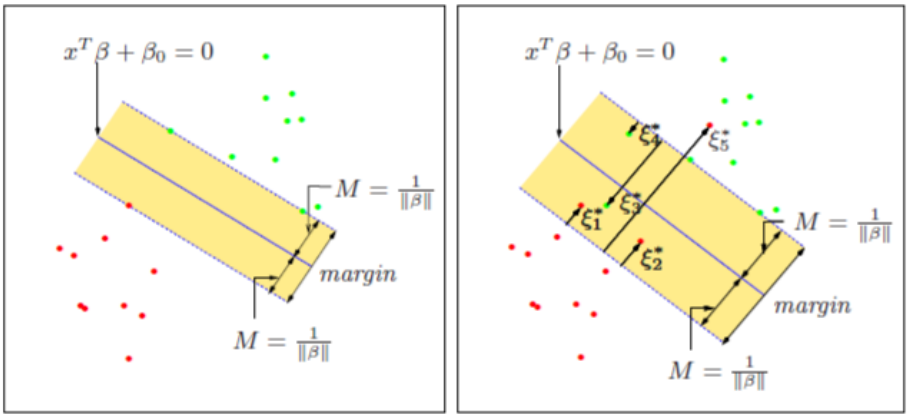


Figure 1.33: The left panel shows the separable case. The decision boundary is the solid line, while broken lines bound the shaded maximal margin of width $2M = \frac{2}{\|\beta\|}$. The right panel shows the *non-separable* case. The point labeled ξ_j^* are on the wrong side of their margin by an amount $\xi_j^* = M\xi_j$; points on the correct side have $\xi_j^* = 0$. The margin is maximized subject to a total budget $\sum \xi_i \leq constant$. Hence $\sum \xi_i^*$ is the total distance of points on the wrong side.

We can drop the norm constraint on β, and instead define $M = \frac{1}{\|\beta\|}$:

$$\min \|\beta\| \text{ subject to } \begin{cases} \forall i, y_i\left(x_i^T\beta + \beta_0\right) \geq M\left(1 - \xi_i\right) \\ \xi_i \geq 0, \sum \xi_i \leq constant \end{cases}$$

This is the usual way the support vector classifier is defined from the non-separable case.

Computing the Support Vector Classifier Computationally it is convenient to re-express previous equation:

$$\min_{\beta,\beta_0} \frac{1}{2}\|\beta\|^2 + C\sum_{i=1}^{N}\xi_i \text{ subject to } \forall i, \xi_i \geq 0, y_i\left(x_i^T\beta + \beta_0\right) \leq M\left(1 - \xi_i\right)$$

where the "cost" parameter C replaces the constraint; the separable case corresponds to $C = \infty$

We describe a quadratic programming solution using Lagrange multipliers. The Lagrange (primal) function is:

$$L_p = \frac{1}{2}\|\beta\|^2 + C\sum_{i=1}^{N}\xi_i - \sum_{i=1}^{N}\alpha_i\left[y_i^T\beta + \beta_0 - (1-\xi_i)\right] - \sum_{i=1}^{N}\mu_i\xi_i$$

Setting the respective derivatives to zero we get : $\begin{cases} \beta = \sum_{i=1}^{N}\alpha_i y_i x_i \\ 0 = \sum_{i=1}^{N}\alpha_i y_i \\ \alpha_i = C - \mu_i, \forall i \end{cases}$

By inserting the equation system in the Lagrange function we obtain: $L_D = \sum_{i=1}^{N}\alpha_i - \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_j y_i y_j x_i^T x_j$ which gives a lower bound on the objective function (computational re-expression) for any feasible point.

1.8.2 Support vectors classifiers

Aim Considering that the maximal margin hyperplane is extremely sensitive to a change in a single observation, it may have overfit the training data.
In this case we might be willing to consider a classifier based on a hyperplane that does not perfectly separate the 2 classes therewith to get

- Greater robustness to individual observation
- Better classification of most of the training observations

And the *Support Vector Classifier* does exactly this.

Details The support vector classifier classifies a test observation depending on which side of the hyperplane it lies.

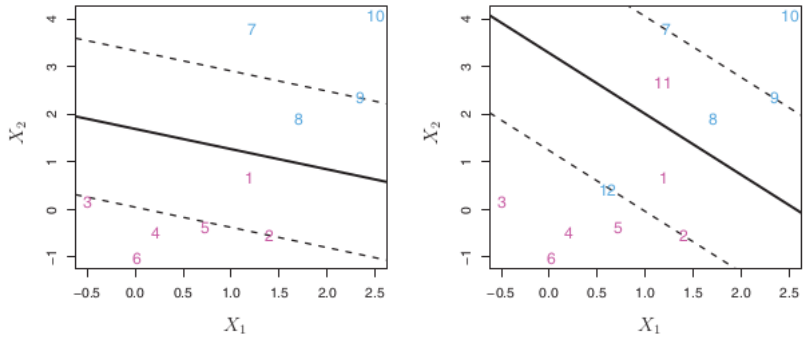


Figure 1.34: Left:Purple observations: 3, 4, 5 and 6 are on the correct side, 2 is on the margin and 1 is on the wrong side.
Blue observations: 7 and 10 are on the correct side, 9 on the margin and 8 on the wrong side
Right: same as left panel with two additional points, 11 and 12 which are on the wrong side

It is the solution to the optimization problem

$$\max_{(\beta_i)_{0\leq i\leq p}(\epsilon_i)_{1\leq i\leq n},M} M \quad \text{subject to} \begin{cases} \sum_{j=1}^{p}\beta_j^2 = 1 \\ y_i\left(\beta_0 + \sum_{j=1}^{p}\beta_j x_{ij}\right) > M(1-\epsilon_i) \\ \epsilon_i \geq 0, \sum_{i=1}^{n}\epsilon_i \leq C \end{cases}$$

where C is a nonnegative tuning parameter, M is the width of the margin. $(\epsilon_i)_{1\leq i\leq n}$ are *slack variables* that allow individual observations to be on the wrong side of the margin or the hyperplane. We have:

$$\begin{cases} \epsilon_i = 0 \Rightarrow i^{th} \text{ observation is on the correct side of the margin} \\ \epsilon_i > 0 \Rightarrow i^{th} \text{ observation is on the wrong side of the margin} \\ \epsilon_i > 1 \Rightarrow i^{th} \text{ observation is on the wrong side of the hyperplane} \end{cases}$$

C determines the number and severity of the violations to the margin that we will tolerate.

In practice C is treated as a tuning parameter that is generally chosen via cross-validation.

It turns out an observation that only observations that lies strictly on the correct side of the margin does not affect the support vector classifier.

1.8.3 Support vectors machines

Classification with non-linear decision boundaries Rather than fitting a support vector classifier using p features we could instead fit a support vector

classifier using $2p$ features, then we get for $\left(X_i, X_i^2\right)_{0\leq i\leq p}$:

$$\max_{(\beta_{i1})_{0\leq i\leq p}(\beta_{i2})_{1\leq i\leq p}(\epsilon_i)_{1\leq i\leq n},M} M \text{ subject to } \begin{cases} \sum_{j=1}^{p}\sum_{k=1}^{2}\beta_{jk}^2 = 1 \\ y_i\left(\beta_0 + \sum_{j=1}^{p}\beta_{j1}x_{ij} + \sum_{j=1}^{p}\beta_{j2}x_{ij}^2\right) > M(1-\epsilon_i) \\ \epsilon_i \geq 0, \sum_{i=1}^{n}\epsilon_i \leq C \end{cases}$$

Support Vector Machine The linear support vector classifier can be represented as :

$$f(x) = \beta_0 + \sum_{i\in\mathcal{S}}\alpha_i\langle x_i|\, x_{i'}\rangle$$

where $(\alpha_i)_{1\leq i\leq n}$ are parameters.
It turns out that α_i is nonzero only for the support vectors in the solution. So $\mathcal{S}$ is the collection of indices of support vectors in the solution.

We replace inner product with a generalization of the inner product of the form: $K(x_i, x_{i'})$ Where K is some function that we will refer to as a *kernel*. A kernel is a function that quantifies the similarity of 2 observations

Linear Kernel $K(x_i, x_{i'}) = \sum_{j=1}^{p} x_{ij}x_{i'j}$ essentially quantifies the similarity of a pair of observations using Pearson (standard) correlation.

Polynomial Kernel $K(x_i, x_{i'}) = \left(1 + \sum_{j=1}^{p} x_{ij}x_{i'j}\right)^d$ instead of the standard linear kernel in the support vector classifier algorithm leads to a much more flexible decision boundary.

Radial Kernel $K(x_i, x_{i'}) = \exp\left(-\gamma\sum_{j=1}^{p}(x_{ij}x_{i'j})^2\right)$ with $\gamma \geq 0$, the radial kernel has very local behavior, in the sens that only nearby training observations have an effect on the class label of a test observation.

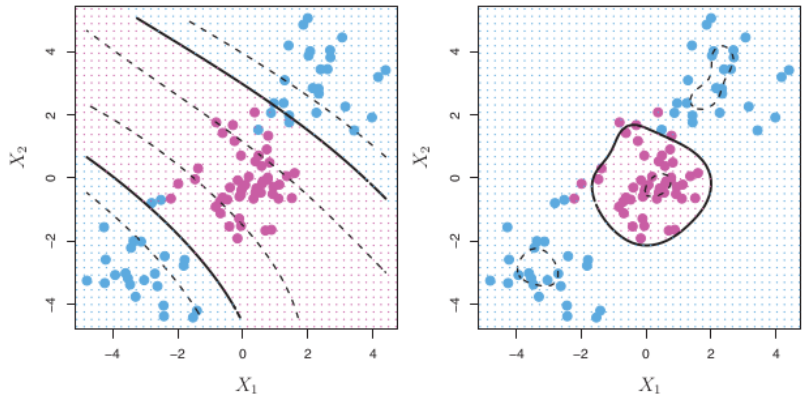


Figure 1.35: Left: a SVM with a polynomial kernel of degree 3 is applied to non-linear data.
Right: a SVM with a radial kernel is applied.

```python
import pandas as pd
import sklearn
from sklearn import tree
from sklearn import svm

y, X = df.iloc[:, 0], df.iloc[:, 1:]
clf = svm.SVC(
    C = 1.0, # Regularization parameter
    kernel = 'linear', # {'linear', 'poly', 'rbf'}
)
clf.fit(X, y)
```

1.8.4 SVMs with more than 2 classes

One-Versus-One Classification Suppose that we would like to perform classification using SVMs, and there are $K > 2$ classes.
We classify a test observation using each of the $\binom{K}{2}$ classifier and we tally the number of times that the test observation is assigned to each of the K classes. The final classification is performed by assigning the test observation to the class to which it was most frequently assigned in these $\binom{K}{2}$ pairwise classifications.

One-Versus-All Classification We fit K SVMs, each time comparing one of the K classes to the remaining $K - 1$ classes.

1.8.5 Relationship to Ridge Regression

It turns out that one can rewrite solution for fitting the support vector classifier

$$f(X) = \beta_0 + \sum_{i=1}^{p} \beta_i X_i$$

$$\min_{(\beta_i)_{1\leq i\leq p}} \left\{ \sum_{i=1}^{n} max[0, \quad 1 - y_i f(x_i)] + \lambda \sum_{j=1}^{p} \beta_j^2 \right\}$$

where λ is a nonnegative tuning parameter, and $P(\beta) = \sum_{j=1}^{p} \beta_j^2$ (ridge regression) and $P(\beta) = \sum_{j=1}^{p} |\beta_j|$ (lasso) Recall that the "Loss + Penality" form is:
$\min_{(\beta_i)_{1\leq i\leq p}} \quad L(X, y, \beta) + \lambda P(\beta)$
For the ridge regression and the lasso both loss functions take this form with:
$L(X, y, \beta) = \sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j \right)^2$
For the SVM the loss function istead takes the form:
$L(X, y, \beta) = \sum_{i=1}^{n} max[0, \quad 1 - y_i \left(\beta_0 + \sum_{j=1}^{p} \beta_j x_{ij} \right)]$

Regression and Kernels Suppose we consider approximation of the regression function in terms of a set of basis functions $\{h_m(x)|m \in [\![1, M]\!]\}$: $f(x) = \sum_{m=1}^{M} \beta_m h_m(x) + \beta_0$ To estimate β and β_0 we minimize:

$$H(\beta, \beta_0) = \sum_{i=1}^{N} V(y_i - f(x_i)) + \frac{\lambda}{2} \sum \beta_m^2$$

for some general error measure $V(r)$.
For any choice of $V(r)$, the solution $\hat{f}(x) = \sum \hat{\beta}_m h_m(x) + \hat{\beta}_0$ has the form :

$$\hat{f}(x) = \sum_{i=1}^{N} \hat{\alpha}_i K(x, x_i)$$

with $K(x, y) = \sum_{m=1}^{M} h_m(x) h_m(y)$
We estimate β by minimizing the penalized least squares criterion:

$$H(\beta) = (\boldsymbol{y} - \boldsymbol{H}\beta)^T (\boldsymbol{y} - \boldsymbol{H}\beta) + \lambda \|\beta\|^2$$

The solution is $\hat{\boldsymbol{y}} = \boldsymbol{H}\hat{\beta}$ with $\hat{\beta}$ determined by: $-\boldsymbol{H}^T(\boldsymbol{y} - \boldsymbol{H}\hat{\beta}) + \lambda\hat{\beta} = 0$ We can pre-multiply by $\boldsymbol{H}$ to give:

$$\boldsymbol{H}\hat{\beta} = \left(\boldsymbol{H}\boldsymbol{H}^T + \lambda \boldsymbol{I}\right)^{-1} \boldsymbol{H}\boldsymbol{H}^T \boldsymbol{y}$$

```
import pandas as pd
import sklearn
from sklearn.kernel_ridge import KernelRidge

y, X = df.iloc[:, 0], df.iloc[:, 1:]
clf = KernelRidge(
    alpha=0.5,
    kernel='linear')
clf.fit(X, y)
```

1.8.6 Linear Discriminant Analysis

There is a class of techniques that produce better classifiers than LDA by directly generalizing LDA.

Flexible Discriminant Analysis This is a method for performing LDA using linear regression on derived responses.
We assume that we have observations with a quantitative response G falling into one K classes $\mathcal{G} = \{k\}_{1 \leq k \leq K}$ each having measured features X.
Suppose $\theta : \mathcal{G} \mapsto \mathbb{R}$ is a function that assigns scores to the classes, such that the transformed class labels are optimally predicted by linear regression on X: If our training has the form (g, x_i) for $i \in [\![1, N]\!]$ we solve:

$$\min_{\beta,\theta} \sum_{i=1}^{N} \left(\theta(g_i) - x_i^T \beta\right)^2$$

with restriction to θ to avoid a trivial solution (mean zero and unit variance over the training data).
More generally, we can find up to $L \leq K - 1$ sets of independent scorings for the class labels $(theta_l)_{1 \leq l \leq L}$ and L corresponding linear maps $\eta_l(X) = X^T \beta_l$ with$l \in [\![1, L]\!]$ chosen to be optimal for multiple regression in $\mathbb{R}^p$. The scores $\theta_l(g)$ and the maps β_l are chosen to minimize the averages squared residual:

$$ASR = \frac{1}{N} \sum_{l=1}^{L} \left[\sum_{i=1}^{N} \left(\theta_l(g_i) - x_i^T \beta_l\right)^2 \right]$$

Moreover, the Mahalanobis distance of a test point x to the k^{th} class centroid $\hat{\mu}_k$ is given by:

$$\delta_J(x, \hat{\mu}_k) = \sum_{l=1}^{K-1} \omega_l \left(\hat{eta}_l(x) - \overline{\eta}_l^k\right)^2 + D(x)$$

where $\overline{\eta}_l^k$ is the mean of the $\hat{\eta}_l(x_i)$ in the k^{th} class, and $D(x)$ does not depend on k. Here ω_l are coordinate weights that are defined in terms of the mean squared residual r_l^2 of the l^{th} optimally scored fit:

$$\omega_l = \frac{1}{r_l^2(1 - r_l^2)}$$

To summarize LDA can be performed by a sequence of linear regressions, followed by classification to the closest class centroid in the space of fits. The analogy applies both to the reduced rank version or the full rank case when $L = K - 1$.
The real power of this result is in the generalizations that it invites, we can replace the linear regression fits $\eta_l(x) = x^T\beta$ by far more flexible non-parametric fits, and by analogy achieve a more flexible classifier than LDA. In this more general form the regression problems are defined via the criterion:

$$ASR\left(\{\theta_l, \eta_l\}_{l=1}^{L}\right) = \frac{1}{N}\sum_{l=1}^{L}\left[\sum_{i=1}^{N}\left(\theta_l(g_i) - \eta_l(x_i)\right)^2 + \lambda J(\eta_l)\right]$$

where J is a regularized appropriate for some forms of non-parametric regression, such as smoothing splines, additive splines and lower-order ANOVA spline models.

Penalized Discriminant Analysis Although FDA is motivated by generalized optimal scoring, it can also be viewed directly as a form of regularized discriminant analysis. Suppose the regression procedure used in FDA amounts to a linear regression onto a basis expansion $h(X)$, with a quatrat

The steps in FDA can be viewed as generalized form of LDA which we call penalized discriminant analysis or PDA:

- Enlarge the set of predictors X via basis expansion $h(X)$
- Use penalized LDA in the enlarged space, where the penalized Mahalanobis distance is given by: i

  $$D(x, \mu) = \left(h(x) - h(\mu)\right)^T \left(\Sigma_W + \lambda\Omega\right)^{-1} \left(h(x) - h(\mu)\right)$$

  where Σ_W is the within-class covariance matrix of the derived variables $h(x_i)$
- Decompose the classification subspace using a penalized metric:

  $$\max u^T \sum \beta u \text{ subject to } u^T\left(\Sigma_W + \lambda\Omega\right)u = 1$$

1.9 Unsupervised learning

1.9.1 The challenge of unsupervised learning

Unsupervised learning is often performed as part of an *explaratory data analysis.*

1.9.2 Principal compoments analysis

Refers to the process by which principal components are computed, and the subsequent use of these components in understanding the data.

Definition of principal components PCA finds a low-dimensional representation of a data set that contains as much as possible of the variation. PCA seeks a small number of dimensions that are interesting as possible.

The *first principal component* of a set of features $(X_i)_{1\leq i\leq p}$ is the *normalized* linear combination of the features:

$$Z_1 = \sum_{i=1}^{p} \phi_{i1} X_i$$

that has the largest variance. By normalized, we mean that $\sum_{j=1}^{p} \phi_{j1}^2 = 1$

The first principal component loading vector solves the optimization problem:

$$\max (\phi_{i1})_{1\leq i\leq n} \left\{ \frac{1}{n} \sum_{i=1}^{n} \left(\sum_{j=1}^{p} \phi_{j1} x_{ij} \right)^2 \right\} \text{ subject to } \sum_{j=1}^{p} \phi_{j1}^2 = 1$$

It turns out that constructing Z_2 to be uncorrelated with Z_1 is equivalent to constraining the direction ϕ_2 to be orthogonal to the direction ϕ_1

The principal components of a set of data in $\mathbb{R}^p$ provide a sequence of best linear approximations t that data, of all ranks $q \leq p$
Consider the rank-q linear model:

$$f(\lambda) = \mu + \boldsymbol{V}_q \lambda$$

where μ is a location vector in $\mathbb{R}^p$, $\boldsymbol{V}_q$ is a $p \times q$ orthogonal unit vectors, and λ is a q vector of parameters. Fitting such a model to the data by least squares amounts to minimizing the *reconstruction error*:

$$\min_{\mu, \{\lambda_i\}, \boldsymbol{V}_q} \sum_{i=1}^{N} \|x_i - \mu - \boldsymbol{V}_q \lambda_i\|^2$$

Another interpretation of principal components An alternative interpretation for principal components can also be useful: principal components provide low-dimensional linear surfaces that are *closet* to the observations.
With the first principal component we seek a single dimension of the data that lies as close as possible to all of the data points.

More on PCA

Scaling the variables The results obtained when we perform PCA will also depend on whether the variables have been individually scaled.

Uniqueness of the principal component The sign has no effects as the direction does not change.

The proportion of variance explained We are interested in knowing the *Proportion of Variance Explained* PVE by each principal component.
The total variance present in a daa set (assuming that the variables have been centered to have mean zero) is defined as:

$$\sum_{j=1}^{p} \mathbb{V}(X_j) = \sum_{j=1}^{p} \frac{1}{n} \sum_{i=1}^{n} x_{ij}^2$$

The variance explained by the m^{th} principal component is:

$$\frac{1}{n} \sum_{i=1}^{n} z_{im}^2 = \frac{1}{n} \sum_{i=1}^{n} \left(\sum_{j=1}^{p} \phi_{jm} x_{ij} \right)^2$$

PVE:

$$\frac{\sum_{i=1}^{n} \left(\sum_{j=1}^{p} \phi_{jm} x_{ij} \right)^2}{\sum_{j=1}^{p} \sum_{i=1}^{n} x_{ij}^2}$$

Deciding how many principal component to use We tipycally decide on the number of principal components required to visualize the data by examining a *scree pot*

```
import pandas as pd
import sklearn
from sklearn.decomposition import PCA

y, X = df.iloc[:, 0], df.iloc[:, 1:]
pca = PCA(
    n_components=2,
    svd_solver='full' # {'auto', 'full', 'arpack', '
     randomized'}
    )
pca.fit(X)
print(pca.explained_variance_ratio_)
```

1.9.3 Clustering methods

K-Means Clustering It is a method for finding clusters and cluster centers in a set of unlabeled data.

Definition The principle lays on the thought that a good clustering is one for which the within cluster variation is as small as possible. The within-cluster variation for cluster $C_k$is a measure $W(C_k)$:

$$\min_{(C_i)_{1\leq i\leq K}} \left\{ \sum_{k=1}^{K} W(C_k) \right\}$$

We need to define the within-cluster variation, the most common choice is:

$$W(C_k) = \frac{1}{|C_k|} \sum_{i,i' \in C_k} \sum_{j=1}^{p} (x_{ij} - x_{i'j})^2$$

where $|C_k|$ denotes the cardinal of k^{th} cluster.

Algorithm K-Means Clustering

1. Randomly assign a number in $[\![1, K]\!]$ to each of the observations.
2. Iterate until the cluster assignment stop changing:
   (a) For each K clusters, compute the cluster *centroid*. The k^{th} cluster centroid is the vector of the p feature means for the observations in the k^{th} cluster.
   (b) Assign each observation to the cluster whose centroid is closet.

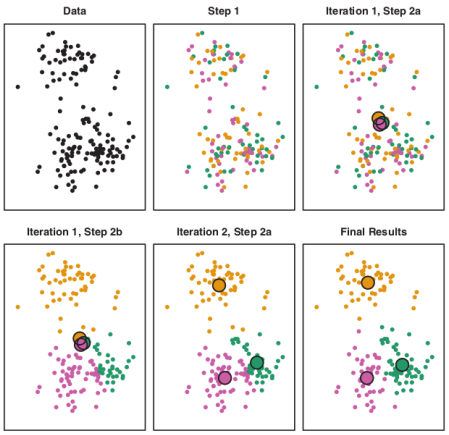


Figure 1.36: Step1: Each observation is randomly assigned
Step(2a): The clusters are computed
Step(2b): Each observation is assigned to the nearest centroid.

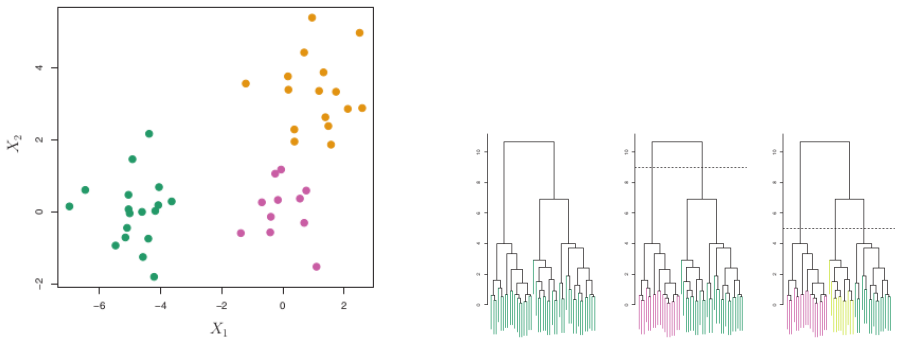
```
import pandas as pd
import sklearn
from sklearn.cluster import KMeans

X = df.copy()
kmeans = KMeans(n_clusters=2,
    random_state=0).fit(X)
print(kmeans.labels_)
print(kmeans.cluster_centers_)
```

Learning Vector Quantization In this technique, prototypes are placed strategically with respect to the decision boundaries in an ad-hoc way. LVQ is an online algorithm. The idea is that the training points attract prototypes of the correct class and repel other prototypes.

1. Choose R initial prototypes for each class: $\{m_r(k)|(r,k) \in [\![1,R]\!] \times [\![1,K]\!]\}$ by sampling R training points at random from each class.
2. Sample a training point x_i randomly (with replacement):
   a if $g_i = k$ (i.e. they are in the same class), move the prototype towards the training point:
   $$m_j(k) \leftarrow m_j(k) + \epsilon(x_i - m_j(k))$$
   where ϵ is the learning rate
   b If $g_i \neq k$ (i.e, they are in different classes), move the prototype away from the training point:
   $$m_j(k) \leftarrow m_j(k) - \epsilon(x_i - m_j(k))$$
3. Repeat step 2, decreasing the learning rate ϵ with each iteration towards zero.

Hierarchical Clustering It is an alternative approach which does not require the we commit to a particular choice of K.
It results in an attractive tree-based representation of the observations, called a *dendrogram*.



(a) Original Data Set (b) Dendrogram

Figure 1.37: There are in reality 3 classes, but we will ignore them and seek to cluster the observations.
In the right figure:
Left: Dendogram obtained from hierarchically clustering the data of of the left figure.
Center: The Dendrogram from the left-hand panel cut at a heigh of 9. This cut results in 2 distinct clusters.
Right: The Dendrogram from the left-hand panel cut at a heigh of 5. This cut results in 3 distincts clusters. .

Interpreting a Dendogram

Hierarchical Clustering

1. Begin with n observations and a measure (such as Euclidean distance) of all the $\binom{n}{2}$ pairwise dissimilarities.
   Treat each observations as its own cluster:
2. $\forall i \in [\![2, n]\!]$:
   (a) Examine all pairwise inter-cluster dissimilarities among the i cluster and identify the pair of clusters that are least dissimilar.
   Fuse these 2 cluster, the dissimilarity between these 2 clusters indicates the height of the dendrogram at which the fusion should be placed.
   (b) Compute the new pairwise inter-cluster dissimilarities among the $i-1$ remaining clusters.

```python
import pandas as pd
import sklearn
from sklearn.cluster import AgglomerativeClustering

X = df.copy()
clustering = AgglomerativeClustering().fit(X)
print(clustering.labels_)
```

Adaptive Nearest-Neighbor Methods When nearest-neighbor classification is carried out in a high-dimensional feature space, the nearest neighbors of a point can be very far away, causing bias.
To quantify this, consider N data points uniformly distributed in the unit cube $\left[-\frac{1}{2}, \frac{1}{2}\right]^p$. Let R be the radius of a 1-nearest-neighborhood centered at the origin. Then:

$$median(R) = v_p^{-\frac{1}{p}} \left(1 - \frac{1}{2}^{\frac{1}{N}}\right)^{\frac{1}{p}}$$

where $v_p r^p$ is the volume of the sphere of radius r in p dimensions.

The *Discriminant Adaptive Nearest-Neighbor* (DANN) metric at a query point x_0 is defined by :

$$D(x, x_0) = (x - x_0)^T \mathbf{\Sigma} (x - x_0)$$

where

$$\begin{aligned}\mathbf{\Sigma} =& \boldsymbol{W}^{-\frac{1}{2}} \left[\boldsymbol{W}^{-\frac{1}{2}} \boldsymbol{B} \boldsymbol{W}^{-\frac{1}{2}} + \epsilon \boldsymbol{I}\right] \boldsymbol{W}^{-\frac{1}{2}} \\ =& \boldsymbol{W}^{-\frac{1}{2}} \left[\boldsymbol{B}^* + \epsilon \boldsymbol{I}\right] \boldsymbol{W}^{-\frac{1}{2}}\end{aligned}$$

Here $\boldsymbol{W}$ is the pooled within-class covariance matrix $\sum_{k=1}^{K}\pi_k \boldsymbol{W}_k$ and $\boldsymbol{W}$ is the between class covariance matrix $\sum_{k=1}^{K}\pi_k\left(\overline{x}_k-\overline{x}\right)\left(\overline{x}_k-\overline{x}\right)$ with $\boldsymbol{W}$ and $\boldsymbol{B}$ computed using only the 50 nearest neighbors around x_0.

1.10 Neural Networks

1.10.1 Projection Pursuit Regression

Let ω_m with $m \in [\![1, M]\!]$ be unit p-vectors of unknown parameters. The projection pursuit regression (PPR):

$$f(X) = \sum_{m=1}^{M} g_m(\omega_m^T X)$$

This is an additive model, but in the derived features $V_m = \omega_m^T X$ rather than the inputs themeselves. The functions g_m are unspecified and are estimated along with the directions ω_m using some flexible smoothing method. The function $g_m(\omega_m^T X)$ is called a ridge function in $\mathbb{R}^p$ The scalar variable $V_m = \omega_m X$ is the projection of X onto the unit vector ω_m an we seek ω_m so that the model fits well, hence the name "projection pursuit".

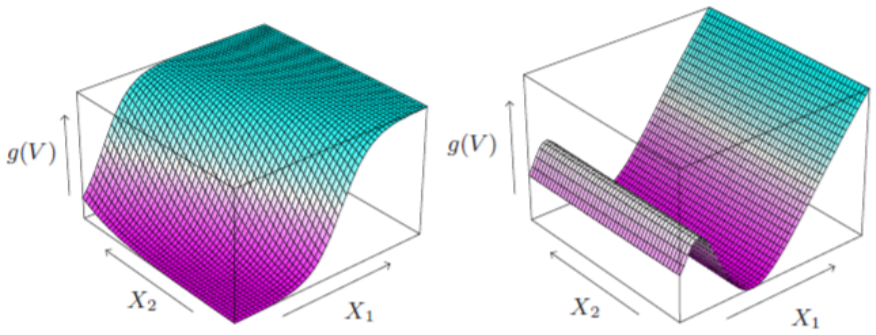


Figure 1.38: Perspective plots of 2 ridge functions.
Left: $g(V) = \dfrac{1}{1+e^{-5(V-0.5)}}$, where $V = \frac{X_1+X_2}{\sqrt{(2)}}$
Right: $g(V) = (V+0.1)\sin\left(\dfrac{1}{\frac{1}{3}+0.1}\right)$, where $V = X_1$

We seek to approximate minimzers of the error function:

$$\sum_{i=1}^{N}\left[y_i - \sum_{m=1}^{M} g_m(\omega_m^T x_i)\right]^2$$

over functions g_m and direction vectors ω_m.
Let ω_{old} be the current estimate for ω.

$$g(\omega^T x_i) \approx g(\omega_{old} x_i) + g'(\omega_{old}^T x_i)(\omega - \omega_{old})^T x_i$$

1.10.2 Neural Networks

They are a large class of nonlinear statistical models much like the projection pursuit regression model.
A neural network is a two-stage regression or classification model, represented by a network diagram:

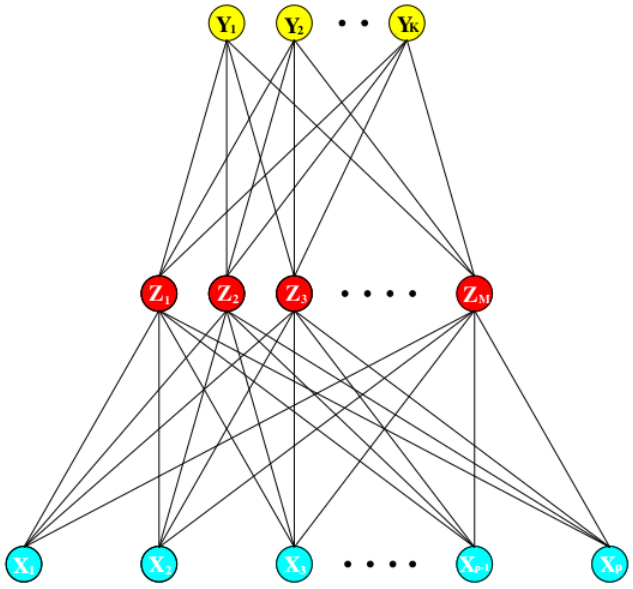


Figure 1.39: Schematic of a single hidden layer, feed-forward neural network

For K-class classification, there are K units at the top, with k^{th} unit modeling the probability of class k. There are K target measurements $\{Y_k : k \in [\![1, k]\!]\} \in 0, 1^K$.
Derived features Z_m are created from linear combinations of the Z_m:

$$\begin{cases} \forall m \in [\![1, M]\!], \ \boldsymbol{Z}_m = \sigma\left(\alpha_{0m} + \alpha_m^T \boldsymbol{X}\right) \\ \forall k \in [\![1, K]\!], \ \boldsymbol{T}_k = \beta_{0k} + \beta_k^T \boldsymbol{Z} \\ \forall k \in [\![1, K]\!], \ f_k(\boldsymbol{X}) = g_k(\boldsymbol{T}) \end{cases}$$

where $Z = \begin{pmatrix} Z_1 \\ \vdots \\ Z_M \end{pmatrix}$ and $T = \begin{pmatrix} T_1 \\ \vdots \\ T_K \end{pmatrix}$ The activation function is usually chosen to be the sigmoid $\boxed{\sigma(v) = \dfrac{1}{1 + e^{-v}}}$, the softmax function is $g_k(T) = \dfrac{e^{T_k}}{\sum_{l=1}^{K} e^{T_l}}$

1.10.3 Fitting Neural Networks

The neural network model has unknown parameters often called *weights* and we seek values for them that make the model fit the training data well. We denote the complete set of of weights by

$$\theta = \begin{cases} \{(\alpha_{0m}, \alpha_m) | m \in [\![1, M]\!]\} \\ \{(\beta_{0k}, \beta_k) | k \in [\![1, K]\!]\} \end{cases}$$

A aour measure of fit we use:

$$\begin{cases} R(\theta) = \sum_{k=1}^{K} \sum_{i=1}^{N} (y_{ik} - f_k(x_i))^2 \text{ regression} \\ R(\theta) = -\sum_{k=1}^{K} \sum_{i=1}^{N} y_{ik} \log (f_k(x_i)) \text{ classification} \end{cases}$$

The generic approach to minimizing $R(\theta)$ is by gradient descent called *back-propagation* in this setting.
Here is back-propagation in detail for squared error loss.
Let $z_{mi} = \sigma(\alpha_{0m} + \alpha_m^T x_i)$ and let $z_i = (z_{1i}, \cdots, z_{Mi})$ with derivatives:

$$\begin{cases} \dfrac{\partial R_i}{\partial \beta_{km}} = -2 (y_{ik} - f_k(x_i)) g_k^{'} (\beta_k^T z_i) z_{mi} \\ \dfrac{\partial R_i}{\partial \alpha_{ml}} = -\sum_{k=1}^{K} 2 (y_{ik} - f_k(x_i)) g_k^{'} (\beta_k^T z_i) \beta_{km} \sigma^{'} (\alpha_m^T x_i) x_{il} \end{cases}$$

Given these derivatives, a gradient descent update at the $(r+1)^{st}$ iteration has the form:

$$\boxed{\begin{cases} \beta_{km}^{(r+1)} = \beta_{km}^{(r)} - \gamma_r \sum_{i=1}^{N} \dfrac{\partial R_i}{\partial \beta_{km}^{(r)}} \\ \alpha_{lm}^{(r+1)} = \alpha_{lm}^{(r)} - \gamma_r \sum_{i=1}^{N} \dfrac{\partial R_i}{\partial \alpha_{lm}^{(r)}} \end{cases}}$$

where γ_r is the *learning rate*. Now we write derivatives as:

$$\begin{cases} \dfrac{\partial R_i}{\partial \beta_{km}} = \delta_{ki} z_{mi} \\ \dfrac{\partial R_i}{\partial \alpha_{ml}} = s_{mi} x_{il} \end{cases}$$

The quantities δ_{ki} and s_{mi} are "errors" from the current model at the output and hidden layer units, respectively. These errors satisfy:

$$\boxed{s_{mi} = \sigma^{'} (\alpha_m^T x_i) \sum_{k=1}^{K} \beta_{km} \delta_{ki}}$$

known as the *back-propagation equations*.

```
import pandas as pd
import sklearn
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier,
    MLPRegressor

y, X = df.iloc[:, 0], df.iloc[:, 1:]
clf = MLPClassifier(
   random_state=1,
   max_iter=300)
clf.fit(X_train, y_train)
print(clf.score(X_test, y_test))
```

Chapter 2

Deep Learning

Consider $p_{data}(\boldsymbol{x})$ the distribution by which examples $\{x_1, \cdots, x_n\}$ are drawn. Let $p_{model}(\boldsymbol{x}, \boldsymbol{\theta})$ be a parametric family of probability distribution over the same space indexed by *theta*. from maps any configuration

2.1 Deep Forward Networks

2.1.1 Gradient based learning

Cost functions In most cases our parametric model defines a distribution $p(x|x; \boldsymbol{\theta})$ and we simply use the principle of maximum likelihood, namely the *cross-entropy* between the training data and the model's prediction as the cost function.

Learning conditional distributions with Maximum Likelihood The cost function is simply the negative log-likelihood:

$$J(\theta) = -\mathbb{E}_{\boldsymbol{X},\boldsymbol{y}\sim\hat{p}_{data}} \left(\log \left(p_{model}(\boldsymbol{y}|\boldsymbol{x})\right)\right)$$

2.2 Regularization for Deep Learning

2.3 Optimization for training Deep Learning

2.4 Convolution Networks

2.5 Recurrent and Recursive Nets