Data Mining Methods

Professor Paul D. Feigin

Dr Darryl Greig

February, 2005

References

[A:1990] A. Agresti, *"Categorical Data Analysis"*, Wiley 1990.

[B:1994] L. Brieman, "*Bagging Predictors*", Technical Report No. 421, Dept of Statistics, UC Berkeley, 1994.

[BFOS:1984] L. Brieman, J. H. Friedman, R. A. Olshen, C. J. Stone, *"Classification and Regression Trees"*, Wadsworth and Brooks/Cole, 1984.

[CHP:2000] S. Chatterjee, A. S. Hadi, B. Price, *"Regression Analysis by Example"*, 3rd Ed, Wiley 2000 (has online datasets at http://www.ilr.cornell.edu/~hadi/RABE).

[ET:1993] B. Efron, R. Tibshirani, *"An Introduction to the Bootstrap"*, Chapman & Hall 1993.

[F:1991] J. Friedman, "*Multivariate Adaptive Regression Splines*", Ann. Stat., Vol. 19, No. 1, pg 1–67, 1991.

[G:1971] J. Gower, *"A General Coefficient of Similarity and Some of its Properties"*, Biometrics 27, 857-74 (1971).

[G:2002] L. Greenfield, http://www.dwinfocenter.org; © LGI Systems Incorporated, 1995-2002.

[GC:2000] M. Gen & R. Cheng, *"Genetic Algorithms and Engineering Optimization"*, Wiley 2000.

[H:1981] D. Hand, *"Discrimination & Classification"*, Wiley 1981.

[HL:1989] D. W. Hosmer & S. Lemeshow, *"Applied Logistic Regression"*, Wiley 1989.

[HMS:2001] D. Hand, H. Mannila, P. Smyth *"Principles of Data Mining"*, DRAFT 2001.

[J:1998] D. Jensen, *"Statistical Challenges to Inductive Inference in Linked Data"*, Proc. 1998 AAAI Fall Symp. Artif. Intel. Link Anal. http://www-eksl.cs.umass.edu/aila/jensen.pdf

[KM:2002] http://www.kernel-machines.org, Multiple tutorials on kernel machines and SVM.

[KR:1990] L. Kaufman & P.J. Rousseeuw, *"Finding Groups in Data: An Introduction to Cluster Analysis"*, Wiley 1990.

[KTRR:1998] R. Kimball, W. Thornthwaite, L. Reeves, M. Ross, *"The Data Warehouse Lifecycle Toolkit"*, Wiley, 1998.

[L:1997] J. S. Long, *"Regression Models for Categorical and Limited Dependent Variables"*, Sage 1997.

[MST:1994] D. Michie, D.J. Spiegelhalter, C.C. Taylor, (eds) *"Machine Learning, Neural and Statistical Classification"*, http://www.amsta.leeds.ac.uk/~charles/statlog/

[R:1986] P. J. Rousseeuw "*A visual display for hierarchical classification*", in E. Diday, Y.Escoufier, L. Lebart, J. Pages, Y. Schektman, R. Tomassone (Eds), *Data Analysis and Informatics 4* (North-Holland, Amsterdam, 1986) 743–748.

[R:1996] B. Ripley, *"Pattern Recognition and Neural Networks"*, Cambridge 1996.

[S:2002] R. Schapire, *"The Boosting Approach to Machine Learning: An Overview"*, MSRI Works. Nonlin. Est. Classif., 2002. http://www.research.att.com/~schapire/boost.html

[SHR:1997] A. Struyf, M. Hubert, P. J. Rousseeuw "*Clustering in an Object-Oriented Environment*" , J. Stat. Soft. Vol 1, No. 4, 1997. http://www.jstatsoft.org/v01/i04/paper/clus.pdf

[SS:1990] A. Sen, M. Srivastava, *"Regression Analysis: Theory, Methods and Applications"*, Springer-Verlag 1990.

[TA:1997] T. M. Therneau & E. J. Atkinson, *"An Introduction to Recursive Partitioning Using the RPART Routines"*, http://www.stats.ox.ac.uk/pub/SWin/rpart2doc.zip

[W:1999] S. Weisburg, University of Minnesota STAT 5163 Course, 1999.

[WI:1998] S. Weiss, N. Indurkhya *"Predictive Data Mining"*; Morgan Kauffman 1998.

1 Introduction to Data Warehousing

[G:2002], [KTRR:1998]

Modern communications, storage media & computational power give rise to the possibility of collecting huge amounts of data. In particular data that would once be read and thrown away (or not even read at all) can now be collected and stored at little or no extra cost. What is more, in some cases legal considerations may require complete electronic records (e.g. credit card transactions). The result is huge databases of information "siphoned off" from regular electronic transactions, or else deliberately created.

There are many definitions of what constitutes a Data Warehouse, one as good as any is:

> "A data warehouse is a copy of transaction data specifically structured for querying and reporting" (above website)

Although this may not be true in every case (e.g., the data may not be transaction data, or the structure may incorporate features other than just querying and reporting) it provides a helpful starting point for our purposes. In particular we should note the following points:

- A data warehouse is NOT generally structured for statistical analysis
- The form of the stored data is not predictable from the outset. It may be normalized, transformed in some other way, stored in date order, etc

To build a data warehouse the following elements are required:

1. The operational transaction system (Source System). Note that this may be distributed over a number of sites, possibly involving different machines and data formats.
2. A mechanism for interacting with the operational system to access the transaction data.
3. A mechanism for cleaning and verifying the data.
4. A mechanism for storing the data for future access.

1.1 Examples

- US government census data (http://www.census.gov)
- All credit card transactions for a particular bank
- Time, number of zones and duration for all telephone calls through a particular exchange
- Contents of individual shopping trolleys at a supermarket

1.2 Statistical Data Collection and Data Warehousing

It is important to emphasize the difference between the kind of data collection associated with traditional statistical experiments, and the data warehousing process. In traditional statistics, data is usually collected under an experimental design that requires certain levels of randomness and accuracy in the collection process. Since data collection can be expensive, usually the number of observations is also specified.

By contrast, a data warehouse is somewhat at the mercy of the operational system. This system may consist of diverse data collecting nodes, which may not use uniform format or methods for collecting data. Furthermore, the collection mechanism will usually not be allowed to interfere with the operating efficiency of the source system, and so may be restricted as to when and how it is allowed to access the data. On the other hand, the number of observations available is normally well above what is needed for specific data mining applications.

1.3 Technical Issues with Data Warehousing

1.3.1 Accessing the Source System

- Source system is supposed to extremely reliable with fast response times for expected queries
- Generally designed for narrow "account based" queries
- Maintenance of historical data and data reporting are often a burden on the system
- Assume that a data warehousing system can access the source data at restricted times, with little opportunity for anything other than sequential reading of the data.

1.3.2 Cleaning the Data

- Examine the data for missing observations, nonsensical values.
- Possibly transform the data so that records from different sites in the source system correspond to the required structure
- Search for and remove duplicate records.
- Format the data for the kind of queries expected by the data warehouse.

1.3.3 Data Access Delays

- Since we are concerned particularly with data mining in this course, we need to understand that we will generally be accessing an existing data warehouse, and will have little or no control over the format or database architecture employed. In general data warehouses are not structured specifically for data mining applications, so simple tasks such as obtaining a random sample of the dataset may prove difficult.
- Along with this, we need to consider the access delays associated with using very large data sets. In this course we assume datasets that can reside on the hard disk of a single machine, which eliminates delays caused by communications or even loading data from static media. However we do NOT assume that the dataset can be loaded into the RAM of a single machine. Our procedures will need to account for this fact and be structured to minimize delays caused by disk access.

2 Introduction to Data Mining

2.1 Utilizing Data Warehouses

Data warehouses offer access to information of a quality and quantity never encountered before. It is only natural that individuals and companies are interested in utilizing this resource for public benefit, or commercial gain. Correctly handled, data mining applications can leverage the resource of data warehouses to pinpoint particular cases (such as detecting criminal activities), or identify general trends in the data (such as telephone usage or spending patterns). In some instances the owner of a data warehouse (such as a bank or supermarket) wishes to maximize the company's benefit from the data collecting and storage process, whereas in other cases, a third party (such as a government agency, or marketing company) wants to access information for tasks that don't directly benefit the warehouse owner.

2.2 Data Types and Distance Measures

[G:1971], [KR:1990]

2.2.1 Tabular Data Format & Distance Measures

Suppose a data warehouse supplies data *observations* of the form $\mathbf{x}_i = (x_{i1}, x_{i2}, ..., x_{ik})$. That is, the observation $\mathbf{x}_i$, which is the i^{th} in some sequence of N observations, is made up of k separate *features* corresponding to the k components. Each of the features are connected in some way to the observation. For example, an observation related to a person might have features describing the gender, height, weight, age, religion, hair color etc of that person. The set of features may be a mix of numerical, binary, ordinal and categorical variables.

The set of observations can be represented as a table (or matrix) with k columns and N rows, corresponding to the features and observations respectively. This *simple tabular data format* is very common in data mining algorithms.

$$\begin{array}{c} \\ \mathbf{x}_1 \rightarrow \\ \mathbf{x}_2 \rightarrow \\ \\ \mathbf{x}_N \rightarrow \end{array} \begin{array}{c} \begin{array}{cccc} \text{feature}1 \downarrow & \text{feature}2 \downarrow & & \text{feature}k \downarrow \end{array} \\ \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1k} \\ x_{21} & x_{22} & \cdots & x_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ x_{N1} & x_{N2} & \cdots & x_{Nk} \end{bmatrix} \end{array}$$

Before we pass this data to a data mining algorithm, we need to consider the format of the data, and what data types are present. Many algorithms presume the data is in a particular format (often continuous, real-valued features), and this may not be a valid assumption for the data we wish to use. In some cases we may also need to supply a *distance* (or *dissimilarity*) between observations, and in other cases a specific distance may be presumed. A distance measure $d(\mathbf{x}, \mathbf{y})$ between observations $\mathbf{x}$ and $\mathbf{y}$ should be zero when the observations are identical, and should increase as they diverge from each other. In particular, a sensible distance measure should satisfy the following:

1. $d(\mathbf{x},\mathbf{y}) \geq 0$ for all observations.
2. $d(\mathbf{x},\mathbf{x}) = 0$.
3. $d(\mathbf{x},\mathbf{y}) = d(\mathbf{y},\mathbf{x})$ (symmetry).
4. $d(\mathbf{x},\mathbf{z}) \leq d(\mathbf{x},\mathbf{y}) + d(\mathbf{y},\mathbf{z})$ (triangle inequality).

In this section we shall consider scaling issues and distance measures for the 5 broad categories of data types – numerical, symmetric binary, asymmetric binary, ordinal and categorical. Finally we shall look at a distance measure that may be used when multiple data types are present in a single observation. Note that there are a great many proposals for distance measures. It is not our intention to supply an exhaustive list, but rather to give a sample of measures that are of interest in data mining applications.

2.2.2 Numerical Data

The main issue we face with numerical data (discrete or continuous) is when we need to compare numerical variables of different type. For example, suppose we fit a model with predictors weight and height – the result may be drastically altered if we measure these in different units, even though the within-feature comparisons will be consistent. Usually it is best to deal with this by scaling the features to either fit within a fixed common interval (uniform scaling), or otherwise to have similar distributional properties (mean zero, standard deviation 1, for example).

There are a number of commonly used distance measures. By far the most common are the *Euclidean distance*:

$$d(\mathbf{x},\mathbf{y}) = \sqrt{\sum_{i=1}^{k}(x_i - y_i)^2} = \sqrt{(\mathbf{x}-\mathbf{y})^T(\mathbf{x}-\mathbf{y})},$$

and the *Manhattan distance*:

$$d(\mathbf{x},\mathbf{y}) = \sum_{i=1}^{k}|x_i - y_i|,$$

both of which are specific cases of the more general distance measure, the L_m or *Minkowski distance*:

$$d(\mathbf{x},\mathbf{y}) = \left(\sum_{i=1}^{k}|x_i - y_i|^m\right)^{\frac{1}{m}}.$$

The Minkowski distance reduces to the Manhattan distance for $m = 1$ and the Euclidean distance for $m = 2$.

In some cases it is known that the features in some observation are correlated with each other. Supposing we are able to obtain or estimate the covariance matrix Σ of the features, then we may compute the *Mahalanobis distance*:

$$d(\mathbf{x},\mathbf{y}) = \sqrt{(\mathbf{x}-\mathbf{y})^T\Sigma^{-1}(\mathbf{x}-\mathbf{y})}.$$

In the case the features are uncorrelated, Σ is the identity matrix and this distance reduces to the Euclidean distance.

2.2.3 Binary Data

Many features encountered in real world datasets are encoded as binary variables, with 1 indicating the presence of the feature, and 0 its absence. Also two case features are often encoded using a binary variable. Now suppose $\mathbf{x}$ and $\mathbf{y}$ are two observations with

binary features. We can trivially compute the following matrix of agreement between the two observations:

$$\begin{array}{c|cc} \mathbf{x} \backslash \mathbf{y} & 1 & 0 \\ \hline 1 & a & b \\ 0 & c & d \end{array}$$

where a indicates the number of features that are 1 in both $\mathbf{x}$ and $\mathbf{y}$, and so on.

The most obvious candidate for a distance is *simple matching*, which amounts to counting up the discrepancies between the two observations:

$$d(\mathbf{x},\mathbf{y}) = b + c = \sum_{i=1}^{k} I(x_i \neq y_i)$$

where $I(\bullet)$ is the indicator function which is 1 if and only if its argument is true. In some cases a normalized distance (i.e. in the range [0,1]) is required, in which case we have:

$$d(\mathbf{x},\mathbf{y}) = \frac{b+c}{a+b+c+d} = \frac{\sum_{i=1}^{k} I(x_i \neq y_i)}{k}.$$

The simple matching distance is *invariant*. That is, if the 0-1 coding for some or all features are reversed, the distance will remain unchanged. This is certainly a desirable quality in cases where the 0 and 1 outcomes are "equivalent" in some sense. For example, if the feature is "gender", then the encodings {0-male, 1-female} and {0-female, 1-male} ought to be interchangeable. This is an example of a *symmetric* binary variable. Symmetric binary variables are essentially 2-class categorical variables.

There are, however, many instances in which this symmetry is not desirable. For example, if the features are characteristics of a plant, then one feature may be that the color of the flower is red. This is a 0-1 feature, but the fact that two plants have a 1-1 match on this feature is qualitatively different than if they have a 0-0 match. In particular, a 1-1 match tells us that the plants have flowers of the same color, and that the color is red. A 0-0 match, on the other hand, tells us only that the flowers of both plants are not red, but provides no further color information. This is an example of an *asymmetric* binary feature. In comparing two asymmetric binary features we are usually interested only in the case where at least 1 of them is 1 – the 0-0 case holds no interest. In this instance we use a measure that is *not* invariant under a different coding. A common candidate is the Jaccard coefficient (see [KR:1990]):

$$d(\mathbf{x},\mathbf{y}) = \frac{b+c}{a+b+c} = \frac{\sum_{i=1}^{k} I(x_i \neq y_i)}{k - \sum_{i=1}^{k} I(x_i = y_i = 0)}$$

2.2.4 Categorical Data

A categorical feature simply records that a particular category is observed or not. For example, the feature might be fruit, and the categories {banana, apple, orange}. Once again it makes no sense to compare this data to other categorical data, or to different data

types. If the different categories are considered to be equivalent, there is little more that can be done as far as comparison than simple matching. The distance is considered to be 1 if the features do not match, or zero otherwise:

$$d(\mathbf{x},\mathbf{y}) = \sum_{i=1}^{k} I(x_i \neq y_i) .$$

Once again, we may wish to normalize this distance over the number of features in an observation of categorical features to get a distance between 0 and 1:

$$d(\mathbf{x},\mathbf{y}) = \frac{\sum_{i=1}^{k} I(x_i \neq y_i)}{k} .$$

However, in many instances a distance between categorical features also involves a concept of different costs incurred. For example, we may determine that an observation of "horse" is closer to an observation of "pony" than "frog". This can be formalized in the concept of a *cost matrix* C, in which the ij^{th} entry gives the cost (or distance) incurred by having an observation of type i and one of type j.

$$d(\mathbf{x},\mathbf{y}) = \sum_{i} C[x_i, y_i] .$$

Some data mining algorithms require numerical input for all features. In this case we must be careful that our numerical transformation doesn't bias one category over another. An effective way to represent a categorical feature with m-categories is as a m-vector of zeros and ones. For a particular observation the vector will be zeros everywhere except for the entry corresponding to the category of the observation.

2.2.5 Ordinal Data

Ordinal data has a feature set that is ordered, but has no intrinsic numerical significance. For example, a variable with values: {worst, bad, good, best}. The only trustworthy information is in the *order* of the variable, and this forms the basis for comparing observations. In the first instance, suppose we have an ordinal feature with m levels. Suppose further that we can treat the levels of this feature as being equally spaced. Then we may substitute observations of the feature for the *rank* of that feature observation. That is, x_i is substituted by $r(x_i) \in \{1,...,m\}$, the rank of x_i. If we wish to normalize the rank to 0-1, we can use, for the i-th feature with m_i levels,

$$z_i(x_i) = \frac{r(x_i) - 1}{m_i - 1} .$$

Then a suitable candidate for the distance between two observations $\mathbf{x} = (x_1,...,x_k)$ and $\mathbf{y}$ with ordinal features is the Manhattan distance, which we encountered above, between the rank transformed values:

$$d(\mathbf{x},\mathbf{y}) = \sum_{i=1}^{k} \left| z_i(x_i) - z_i(y_i) \right|$$

where z_i denotes the rank transform for ordinal feature i. This measure may also be normalized by dividing by the number of features in each observation.

If we may *not* assume that the levels of an ordinal feature are equally spaced (for example, is a "good" observation as different from a "bad" one as a "bad" is from a "worst"?), then an obvious choice is to treat the feature as categorical and define an appropriate cost matrix based distance.

2.2.6 Data with Mixed Feature Types

In many data mining tasks we are faced with observations that contain a mixture of binary, ordinal, categorical and numerical features. Furthermore, even if observations contain features of a single type, they may not be on the same scale (a weight measured in kilograms and another in grams, for example). There are a number of options available for comparing observations of this type.

Firstly, we may simply choose to analyze each type of data separately. That is, we perform a separate analysis on numerical, binary, categorical and ordinal features, and then compare the results at the end. If the results agree, this is fine, but if not we have a problem.

A second approach is to simply treat all the variables as numerical and use one of the general numerical metrics (such as Euclidean or Mahalanobis). This will generally require that categorical and ordinal variables are transformed in some sensible way (for example, represent categorical features as binary vectors, and ordinals using the ranking scheme described above). Many data mining algorithms favor this approach because it is very simple, and often gives quite good results.

Finally, we may attempt to construct some sort of combined distance measure. Depending on the specific characteristics of the data it may be desirable to define a "custom-built" distance measure, although there are general purpose distances available.

A popular and simple combined distance measure is *Gower's General Dissimilarity Metric* ([G:1971]). This distance measure is essentially a sum of four different (normalized) distance measures for the four different types of data. In the case that features are all of one type, the distance reduces to the corresponding distance measure for that type of data. Once again, we use the index $i \in \{1,\ldots,k\}$ to refer to feature number i in the observations. Then Gower's metric is

$$d(\mathbf{x},\mathbf{y}) = \frac{\sum_{i=1}^{k} \delta(x_i, y_i) d^{(i)}(x_i, y_i)}{\sum_{i=1}^{k} \delta(x_i, y_i)}$$

where δ is an indicator function which is 0 if either x_i or y_i are missing values, or if both are 0 and i is an asymmetric binary feature. Otherwise δ is 1. The distance $d^{(i)}(x_i, y_i)$ is a distance that depends based on the type of feature i.

- If feature i is binary or categorical:
  $$d^{(i)}(x_i, y_i) = I(x_i \neq y_i)$$
- If feature i is numerical the let R_i be the *range* of the feature. That is $R_i = \max_i - \min_i$, where the maximum and minimum

are either drawn from domain knowledge, or from the available data. Then

$$d^{(i)}(x_i, y_i) = \frac{|x_i - y_i|}{R_i}$$

- If feature i is ordinal:

$$d^{(i)}(x_i, y_i) = |z_i(x_i) - z_i(y_i)|$$

where z_i is defined as above.

It is easy to show that Gower's distance reduces to the normalized Manhattan distance in the case that all the features are numerical, the simple matching distance in the symmetric binary and categorical cases, the Jaccard coefficient in the asymmetric binary case, and the ordinal Manhattan distance in the ordinal case. What is more, for specific observations the distance always lies between 0 and 1 (presuming the range R for numerical features is not violated).

The concept of the δ function as a "pre-filter" can easily be extended to deal with more general situations. For example, we could include conditions to deal with extreme, spurious or unwanted feature values such as numerical values over some threshold, or categorical values belonging to uninteresting categories. In this way we can define a distance measure suitable to the data at hand that does not require extensive preprocessing of the data, or exclusion of observations with NA values.

We may also choose to exchange some of the component metrics in Gower's distance for others (e.g. we may wish to use a Euclidean rather than a Manhattan metric for numerical features), but this will always require some care to ensure the normalization properties are maintained.

2.3 The Functions of Data Mining

([HMS:2001], 1.3)

Data Mining is about discovering useful structure in a data set. Generally the structure we might want to look for can be characterized as one of two types.

Firstly it may be a *global model* of the data set. It makes statements about any point in the allowable space. For example, a model can assign any point to a cluster, or predict the value of some other variable based on the given point. A simple example of a global model is a linear function. Supposing the linear function is obtained by a regression of Y on $X_1,\ldots,X_k$, $Y = \beta_0 + \beta_1 X_1 + \cdots + \beta_k X_k$, then for any point $\mathbf{x} = (x_1,\ldots,x_k)$ in the input space, the model is able to make some statement about the value of the response variable Y.

Alternatively we may look for *pattern structures* in the data set. These make statements only about restricted regions of the space spanned by the variables. Patterns tend to describe characteristics of parts of the data space that are of particular interest – such as regions containing rare events like credit card fraud, or heart attacks. A pattern structure may take the form of a probabilistic rule, such as $P(Y > y_1 \mid X > x_1) = p_1$.

Patterns and models may be viewed as different sides of the same coin. Patterns tend to be concerned with unusual behavior, but finding unusual behavior requires some notion of what is "usual", which is exactly the purpose of global models. Note that this is only a conceptual distinction – in some cases it is not clear whether the structure we are looking for should be regarded as a model or a pattern. Nevertheless, the two classes do help to sharpen our thinking on the functions of data mining.

2.4 The Tasks of Data Mining

([HMS:2001], 1.4)

It is helpful to understand both the type of structure you are seeking in the data set (global or pattern), and also to clarify the purpose of your data mining exercise. We list below one categorization of the tasks of data mining that gives a broad overview of the kinds of tasks you may undertake.

2.4.1 Exploratory Data Analysis (Data Visualization)

The goal here is simply to explore the data without any clear ideas of what we might be looking for. A bit like a policeman walking the beat, looking out for something unusual. In fact, most data mining begins with some sort of exploratory analysis, even if it is just taking summary statistics of the distribution. With 2 or 3 dimensional data, various plots of the data may be useful, although these quickly become cumbersome in higher dimensional data. In that case we must resort to projections or pair-wise comparisons of the data components.

2.4.2 Descriptive Modeling

The goal of a descriptive model is to describe all of the data (or the process generating the data). Examples of such descriptions include:

- Models for the overall probability distribution of the data (*density estimation).*
- Partitioning of the *p*-dimensional space into groups (*cluster analysis and segmentation)*. Segmentation analysis has been successfully used in marketing to divide customers into homogenous groups based on purchasing patterns and demographic data such as age, income, etc.
- Models describing the relationships between variables (*dependency modeling)*.

Note that the goal here is only to find some way to describe or summarize the given data set. There is no intention to extend the model to unseen data, nor to make predictions based on these models.

2.4.3 Predictive Modeling

For the purposes of this course we will think of predictive modeling as either *regression* or *classification.* The aim here is to build a model that will permit the value of one (or more) variables to be predicted based on the known values of the other variables. The term "prediction" is used in the general sense of finding something unknown from something known, rather than a strictly time-based sense. For example, we may be interested in time dependent processes like the stock market, or the weather patterns, but we may just as

well be interested in whether a patient has heart disease, or if the document we are viewing is a journal article rather than a book.

The key distinction between prediction and description is that prediction has as its objective a unique variable or set of variables (the market's value, the disease class etc), while in descriptive problems, no single variable is central to the model.

2.4.4 Discovering Patterns and Rules

The above tasks are all concerned with model building. Other data mining applications are concerned with pattern detection.

- Detecting fraudulent behavior by detecting regions of the space of credit card transactions that differ significantly from the rest.
- Finding unusual stars or galaxies to discover novel astronomical phenomena.
- Discovering combinations of particular items that occur frequently together, for example in a shopping basket.

The process of discovering patterns is related to what statisticians deal with in the context of outlier detection. The challenge here is to determine what truly constitutes unusual behavior in the context of normal variability, a problem only compounded by higher dimensionality in the data. In this field some prior (or background) knowledge can be invaluable to answering these questions.

2.4.5 Retrieval by Content

Here the user has a pattern of interest and wishes to extract similar patterns from the data set. It is particularly applicable in modern data warehouse contexts where there is no physical way for a user to scan all the data available.

- All www search engines are based on retrieval by content algorithms. The user enters keywords and the system returns pages that contain those words and possibly fit other "relevancy criteria".
- Images or documents in a database may be annotated with keywords (for example: "Lion, Jungle, Africa") to enable keyword based retrieval.
- A much harder problem is to retrieve images based on another image. For example: "I want all photographs in which the following face appears …" There is a lot of research into developing systems like this.

In this course we will focus on the predictive methods of data mining for classification and regression. Many of the techniques will be applicable for descriptive modeling and retrieval by content, however we shall not refer further to these tasks in this course.

2.5 Components of a Data Mining Algorithm
([HMS:2001], 1.5)

Once we have defined the data mining task we are attempting, we need to assemble the various pieces necessary for constructing an algorithm to perform the task at hand. We identify 6 pieces required for a data mining algorithm, briefly outlined in the following subsections.

2.5.1 Model or Pattern Structure

This has already been discussed above. Essentially we need to determine what sort of structure we wish to fit to the data. If we wish to discover a global model of the data we may choose a regression (linear or non-linear) structure. If we are seeking a pattern, we may choose a structure consisting of one or more conditional probabilities.

2.5.2 Score Function

Once we have a class of structures we wish to fit to the data, we need some way to quantify how well a particular realization of that class (i.e., a set of parameters applied to the structure) fits the given data set. Our hope is to choose a score function that reflects accurately the expected benefit of the model. In predictive problems this is difficult because the model is intended to be evaluated on unseen data, although validation and regularization techniques can assist in these cases. Furthermore, our score function must be suitable for the kind of structure and optimization we intend to use. For example, if we have chosen a neural network model which we intend to fit using backpropagation, we are restricted to score functions that are differentiable.

In short, the score function is supposed to tell us whether one set of parameters (and possibly model structure) is better than another. Several widely used score functions should be familiar:

- Sum of squared errors (SSE) score (regression problems)

  $$\sum_{i=1}^{n} (y(i) - \hat{y}(i))^2 ,$$

  where we are predicting "target" values for n cases, $y(i), 1 \le i \le n$ and our predictions are denoted as $\hat{y}(i)$.
- Misclassification rate (classification problems)

  $$\sum_{i=1}^{n} \frac{I[y(i) \neq \hat{y}(i)]}{n} .$$
- Sum of absolute errors score (regression problems)

  $$\sum_{i=1}^{n} | y(i) - \hat{y}(i) | .$$

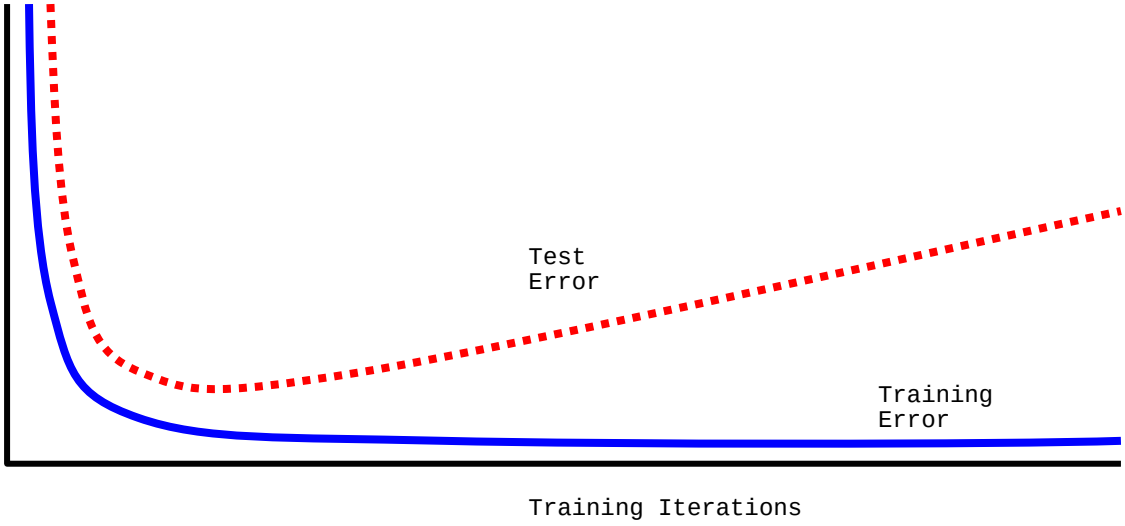
  This score function gives less weight to more extreme outliers than the SSE function.

2.5.3 Regularization & Overfitting

Many model types are very general, and very flexible. It is not unusual to have a model that can fit the data set almost perfectly, particularly when iterative fitting methods are used. However this is often not desirable since the model will tend to fit characteristics that are specific to the training data (such as the inherent noise in the data set), and thus restrict the model's ability to behave well on unseen data. This problem is called *overfitting* and the process of adjusting a model to compensate for overfitting is variously called *regularization* or *generalization*.

The overfitting phenomenon is often thought of in terms of a pair of error curves, as illustrated below. The blue (lower) curve is the reported error on the training set for some iterative training process. Most training schemes ensure that the training error

decreases at each iteration, so the curve tends to follow an initial sharp decline which flattens out to (ideally) asymptote to the minimal training set error. The red (upper) curve represents the performance of the model on an independent set of test data, drawn from the same distribution as the training data. The red curve is generated by stopping training at each iteration of the training algorithm and measuring the test set error on the (partially fitted) model. As the model is fitted to the underlying data characteristics we would expect to see the same sort of rapid initial improvement in the test set error as in the training set error. However, if the model begins to overfit to the training data in later stages of the training, then the test set error will actually increase, which indicates that overfitting is taking place.



In this case, it is desirable to stop training at the point the test error is minimized. Of course, we can't do this directly since monitoring the test set during training violates the independence requirement. Note that this is a very stylized (if popular) representation of the overfitting phenomenon – real training /test curves can be significantly more complicated, involving multiple humps in the test error.

Regularization techniques fall into three general categories, each of which have advantages and disadvantages:

1. Augment the score function with another term that penalizes more complex models, or the use of available flexibility. For example, suppose $\|\theta\| = \sum_j I[\theta_j \neq 0]$ counts the number of non-zero parameters, then we may add a term $\lambda\|\theta\|$ to the error function that contributes a penalty of λ for each parameter used. Alternatively we may allow many parameters, but penalize the size of each parameter $\lambda\sum\theta_i^2$ (this is a form of *weight decay* regularization).
2. Restrict the number of parameters available to the model, either by a hard restriction or a pruning procedure that cuts back parameters as part of the fitting process.
3. Stop the fitting process before the best fit is achieved on the training data set. This amounts to attempting to find the optimal point in the above diagram, and is based on the

assumption that the fitting process will behave in this way. In general, larger models are employed for this kind of regularization scheme to try and guarantee a suitable fit/overfit training curve.

2.5.4 Optimization and Search Method

Once we have chosen a score function with which to compare proposed models or patterns, we require some method of stepping through the space of allowable parameters, or models. The selection of this method is often closely related to the selection of regularization or overfitting containment method, if any is to be used. It also depends on the type of model or pattern structure chosen. For example, a linear regression model is amenable to an algebraic solution, whereas most non-linear regression models must be solved by some form of iterative optimization procedure. Alternatively, the task of finding interesting patterns (such as rules) from a large family of candidates is typically cast as a combinatorial search problem, and often accomplished using heuristic search techniques.

2.5.5 Data Management Technique

The "big data" nature of data mining means that the method used to access, index and store data becomes an important component of a data mining algorithm. Most well-known data analysis algorithms in statistics and machine learning have the implicit assumption that all the data can fit into the RAM of a local machine, with the result that these algorithms rarely include an explicit component dealing with data management.

> ASIDE
> We note in passing that numerical implementations of such algorithms often have to have very efficient data handling properties, particularly involving minimal passes over the data set, and preferably sequential access to the data points.

As noted before, we can assume in this course that the whole data set can reside on a local hard disk (if this assumption is not true in every case, it is rapidly becoming so), but we may not assume that it can reside in RAM.

In our situation, we will need to work with a classic disk access structure, meaning that sequential access to disk addresses is considerably faster than accessing addresses at random. There is research being done on different disk access structures, for example, tree-structured indexing systems, but these are not generally accessible at the moment. Although data management will not be a central topic of this course, it will be referred to from time to time, and the student should keep the issues in mind when structuring algorithms. In particular it is preferable to create algorithms that

1. Minimize passes over the data set.
2. Access the data set in a sequential fashion.
3. Access the data set in a "chunk" fashion – that is, you are able to load a RAM sized chunk of the data set, process that, and then move to the next chunk.

3 Data Summary & Visualization

3.1 Introduction

When confronted with a new data set, it is standard practice to try to summarize the data in some way to begin to understand it. During this stage, new irregularities may become evident, and further data cleaning ensue.

There are a variety of summary statistics that can highlight the broad features of a data set, as well as graphical tools for displaying the data. In an interactive environment these tools enable users to step into the data themselves and use their natural pattern recognition capabilities to try and understand what is happening.

Exploratory data mining primarily uses these tools - summarizing and visualizing the data to see the trends and patterns that come out of their own accord.

The human eye restricts visualization to displays of two or at most three dimensions, which is often not sufficient for the data under consideration. We will consider techniques that allow data to be projected onto two dimensions, or alternatively display the interactions between pairs of higher dimensional datasets.

3.2 Summary Statistics

The first step in analyzing any dataset is usually to produce some summary statistics. These give the basic characteristics of the dataset, but are not usually able to report more sophisticated behavior, such as multimodality. Spurious or outlying observations can only be seen from summaries in the simplest cases (such as very high maximums).

```
> # summary statistics for each variable
> rcdata=read.table("C:\\RData\\Recumbent Cows.txt",header=TRUE)
> summary(rcdata)
```

```
      AST              Urea           Uketone           PCV            Inflamat
 Min.   :  33.0   Min.   :  1.000   Min.   :  1.0   Min.   : 13.00   Min.   :  0.0000
 1st Qu.: 123.0   1st Qu.:  5.625   1st Qu.:  2.0   1st Qu.: 32.00   1st Qu.:  0.0000
 Median : 240.0   Median :  7.600   Median :  2.0   Median : 35.00   Median :  1.0000
 Mean   : 398.4   Mean   :  9.803   Mean   :  2.2   Mean   : 35.56   Mean   :  0.7206
 3rd Qu.: 492.0   3rd Qu.: 10.975   3rd Qu.:  3.0   3rd Qu.: 40.00   3rd Qu.:  1.0000
 Max.   :2533.0   Max.   : 50.000   Max.   :  3.0   Max.   : 61.00   Max.   :  1.0000
 NA's   :   6.0   NA's   :169.000   NA's   :170.0   NA's   :260.00   NA's   :299.0000
    Myopathy          Outcome          Calving          Daysrec            CK
 Min.   :  0.0000   Min.   :0.0000   Min.   :0.0000   Min.   : 0.000   Min.   :   13
 1st Qu.:  0.0000   1st Qu.:0.0000   1st Qu.:1.0000   1st Qu.: 0.000   1st Qu.:  560
 Median :  0.0000   Median :0.0000   Median :1.0000   Median : 1.000   Median : 1760
 Mean   :  0.4279   Mean   :0.3816   Mean   :0.7517   Mean   : 1.947   Mean   : 5352
 3rd Qu.:  1.0000   3rd Qu.:1.0000   3rd Qu.:1.0000   3rd Qu.: 3.000   3rd Qu.: 5467
 Max.   :  1.0000   Max.   :1.0000   Max.   :1.0000   Max.   :20.000   Max.   :71000
 NA's   :213.0000                    NA's   :4.0000   NA's   : 3.000   NA's   :   22
    Unknown           CaseId
 Min.   : 2.565   Min.   : 2788
 1st Qu.: 6.328   1st Qu.:10543
 Median : 7.473   Median :11689
 Mean   : 7.458   Mean   :12011
 3rd Qu.: 8.606   3rd Qu.:13290
 Max.   :11.170   Max.   :18522
 NA's   :22.000
```

```
> # covariance matrix between some variables(NAs omitted)
> cov(rcdata [c("AST","Urea","Uketone","PCV","CK","Unknown")],
+ use="complete.obs")
```

```
                AST         Urea      Uketone          PCV           CK      Unknown
AST      195742.52274  1202.047541  -13.1430460 5.247351e+02 3.206842e+06 4.354904e+02
Urea       1202.04754    67.292459    1.7147541 1.323115e+01 6.043374e+04 7.133026e+00
Uketone     -13.14305     1.714754    0.3778424 3.389741e-01 1.436596e+03 1.877571e-01
```

```
PCV         524.73506    13.231148    0.3389741 3.386039e+01 1.547229e+04 1.960647e+00
CK      3206841.53966 60433.740984 1436.5957166 1.547229e+04 1.372737e+08 1.469647e+04
Unknown     435.49037     7.133026    0.1877571 1.960647e+00 1.469647e+04 2.324824e+00
```

```
> # correlation matrix between some variables(NAs omitted)
> cor(rcdata[c("AST","Urea","Uketone","PCV","CK","Unknown")],
 + use="complete.obs")
```

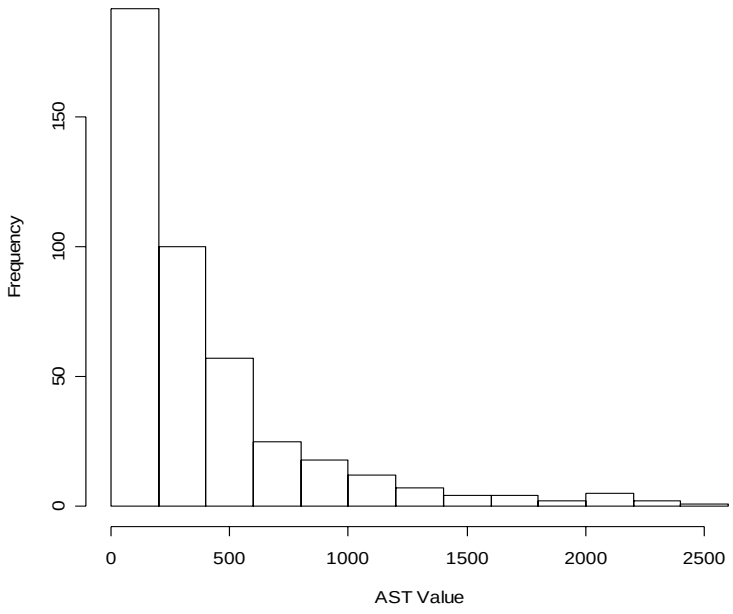
```
               AST      Urea     Uketone        PCV        CK   Unknown
AST      1.00000000 0.3312042 -0.04832792 0.20382221 0.6186446 0.6455664
Urea     0.33120421 1.0000000  0.34006607 0.27718411 0.6287856 0.5702897
Uketone -0.04832792 0.3400661  1.00000000 0.09476875 0.1994736 0.2003300
PCV      0.20382221 0.2771841  0.09476875 1.00000000 0.2269420 0.2209827
CK       0.61864456 0.6287856  0.19947364 0.22694203 1.0000000 0.8226674
Unknown  0.64556640 0.5702897  0.20032997 0.22098274 0.8226674 1.0000000
```

3.3 Density Visualization

3.3.1 Histogram

Simple histograms are only useful for displaying univariate data. A histogram gives the general shape of the distribution, the number of modes, etc. These plots can be useful for determining anomalies in the data, and for grasping some of the properties of the data distribution.

```
> hist(rcdata[,"AST"],main="",xlab="AST Value")
```



3.3.2 Kernel Smoothing

These methods produce what amounts to a smoothed histogram. However they tend to be easier to interpret than a histogram and less subject to noise.

Suppose we have some variable X with observations $x(1),\ldots,x(n)$. The density at any point x is estimated by allowing each of the observed points to contribute according to its distance from x based on some kernel function K. The contribution of each observation is

determined by the distance between the $x(i)$ and the required density point x, thus

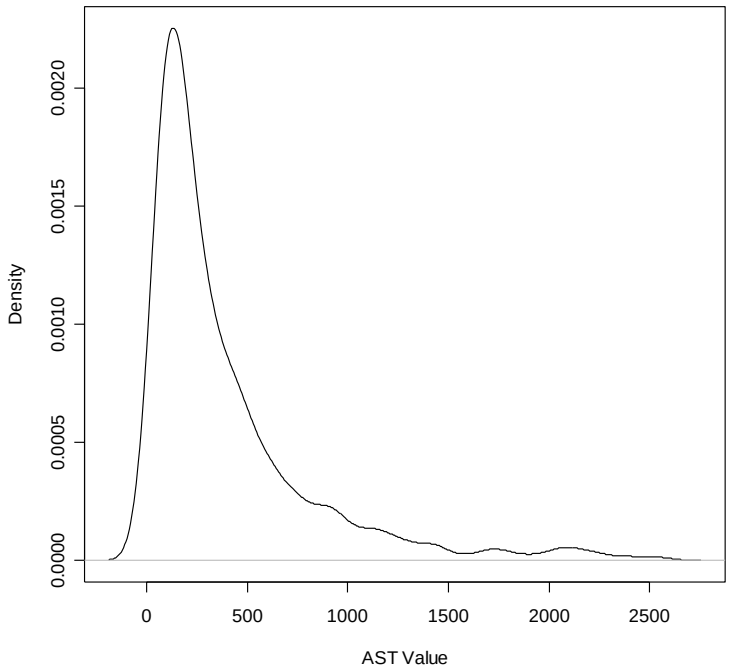
$$f(x) = \frac{1}{nh}\sum_{i=1}^{n} K\left(\frac{(x - x(i))}{h}\right)$$

where $\int K(t)dt = 1$, and where K is usually chosen to be a smooth unimodal function with mode at 0. A common form for K is the Gaussian curve with standard deviation 1 (i.e. $h = 1$):

$$K(t) = \frac{1}{\sqrt{2\pi}} e^{-0.5t^2} .$$

There are methods for choosing an optimal h, however if we are interested in visualizing the data this is best left to the user. A smaller h will give less smoothing, and a larger h more. By adjusting this parameter the user can see more or less "features" of the data.

```
> plot(density(rcdata[,"AST"],kernel="gaussian",na.rm=TRUE), main="",
 + xlab="AST Value")
```

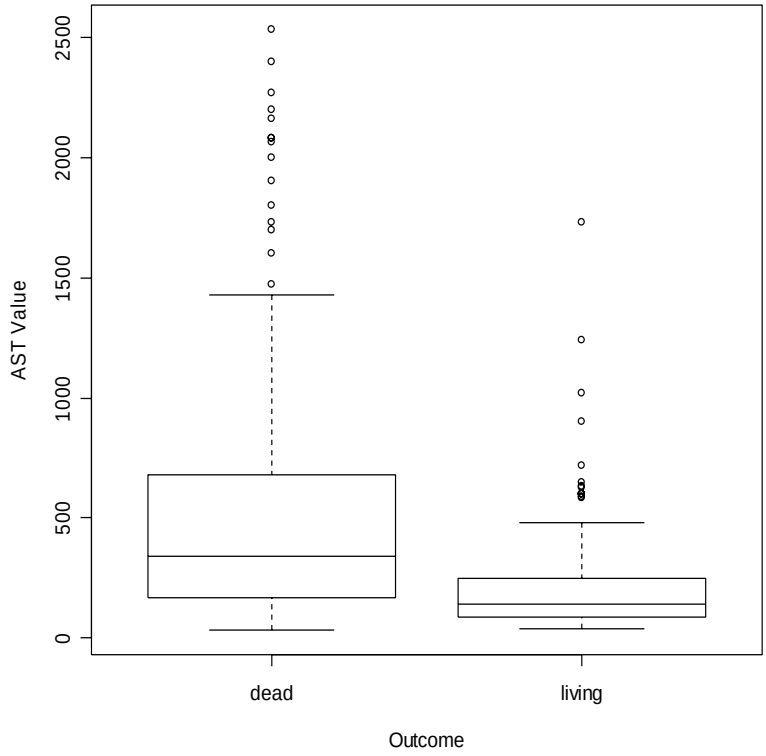


3.3.3 Boxplots

The above techniques are useful for visualizing single variables, however in some cases we may wish to see different groups of scores for a single variable separately. For example, in measuring mathematics grades in school, we may wish to display a summary for the girls' scores and the boys' scores separately, but in some way that the results can be easily compared.

In general, a boxplot consists of a box spanning the bulk of the data range (usually ranging between the first and third quartiles). Some measure of location is given by a line across the box, usually the median of the data. Whiskers projecting from the box tend to illustrate the spread of the empirical distribution.

```
> outcome <- rcdata["Outcome"] # makes the command tidier
> boxplot((rcdata[outcome==0,"AST"]), (rcdata[outcome==1,"AST"]),
 + names=c("dead","living"), xlab="Outcome",ylab="AST Value")
```

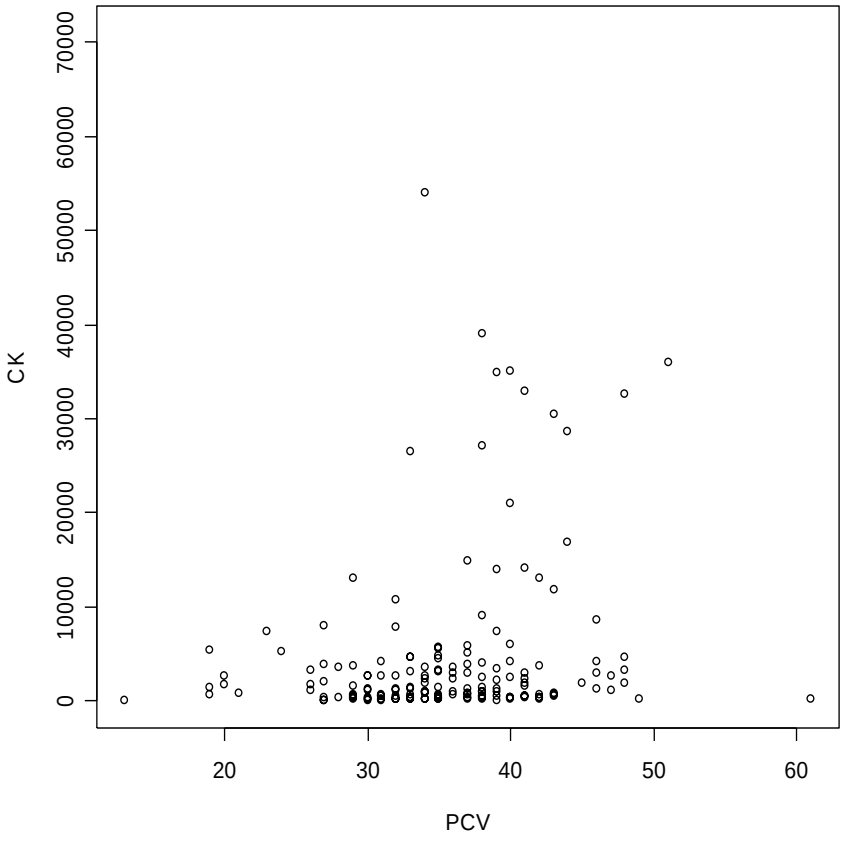


3.4 Relationships between two variables

3.4.1 Scatterplot

The scatterplot is a standard tool for displaying two dimensional data. From a scatterplot we can get some idea of the general trends of the data, and possibly identify outlying values. Unfortunately, the size of the data sets in data mining can limit the usefulness of a scatterplot because there is so much overlap.

```
> plot(rcdata[c("PCV","CK")])
```

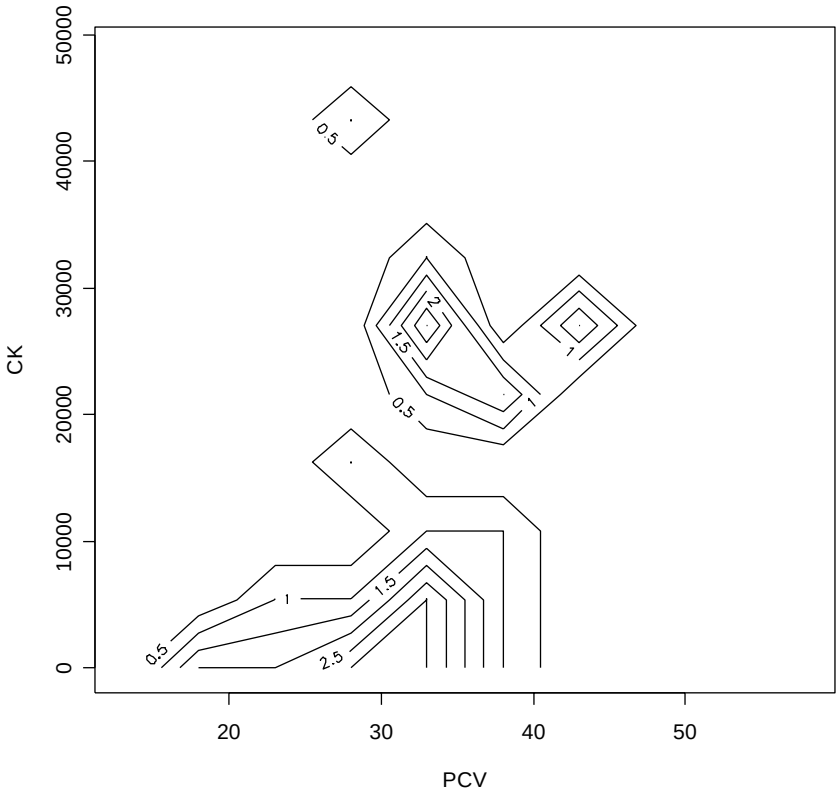


3.4.2 Contour Plots

Contour plots can overcome the issue of point overlap in scatterplots to some extent. A contour plot effectively requires us to construct a two dimensional density estimate using some thing like a two-dimensional generalization of the one-dimensional kernel smoothing already mentioned, and so we are again faced with window-size decisions.

```
# this is a function for taking two variables with overlapping
# scatterplots + generating a contour map of the scatterplot space
mycontour<-function(z,gridx=10,gridy=10,...){
 matz<-as.matrix(na.omit(z))
 rx<-range(matz[,1],na.rm=TRUE)
 ry<-range(matz[,2],na.rm=TRUE)
 winx<-ceiling((rx[2]-rx[1])/gridx)
 winy<-ceiling((ry[2]-ry[1])/gridy)
 gridmat<-matrix(ncol=2,nrow=nrow(matz))
 gridmat[,1]<-floor((matz[,1]-rx[1])/winx)
 gridmat[,2]<-floor((matz[,2]-ry[1])/winy)
 countmat<-matrix(0,ncol=gridy,nrow=gridx)
 dimnames(countmat) <- list(((0:(gridx-1))*winx) + rx[1],\
  ((0:(gridy-1))*winy) + ry[1])

 for(i in 1:nrow(matz)) {
  if(!is.na(gridmat[i,1]) && !is.na(gridmat[i,2])) {
   countmat[gridmat[i,1], gridmat[i,2]] = countmat[gridmat[i,1], \
    gridmat[i,2]] + 1
  }
 }
 contour(x=((0:(gridx-1))*winx)+rx[1],((0:(gridy-1))*winy) + ry[1], \
       countmat,...)
```

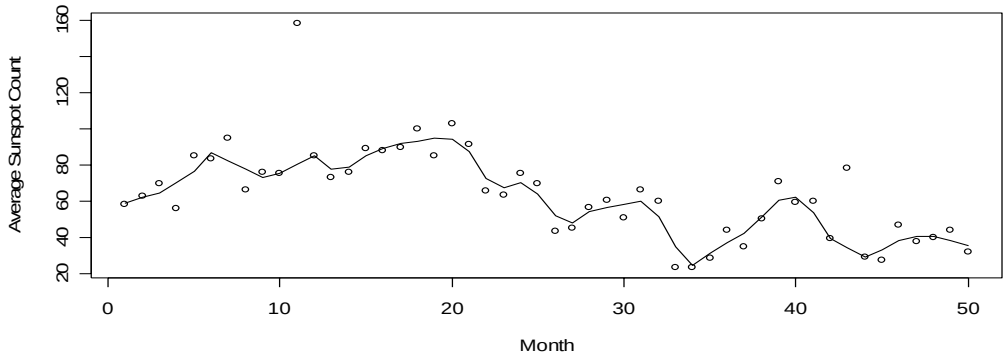
```
}

# now do the contour plot of the same variables
mycontour(rcdata[c("PCV","CK")], xlab="PCV",ylab="CK")
```



3.4.3 Curve Fitting

In the special case that one of the two variables is some sort of index variable (such as time), a scatterplot amounts to showing the value of the other variable as the index increments. The features of the data can be more clearly seen by fitting a smooth curve to the points.

```
> data(sunspots)
> scatter.smooth(1:50,sunspots[1:50], span=0.1, xlab="Month",
 + ylab="Average Sunspot Count")
```

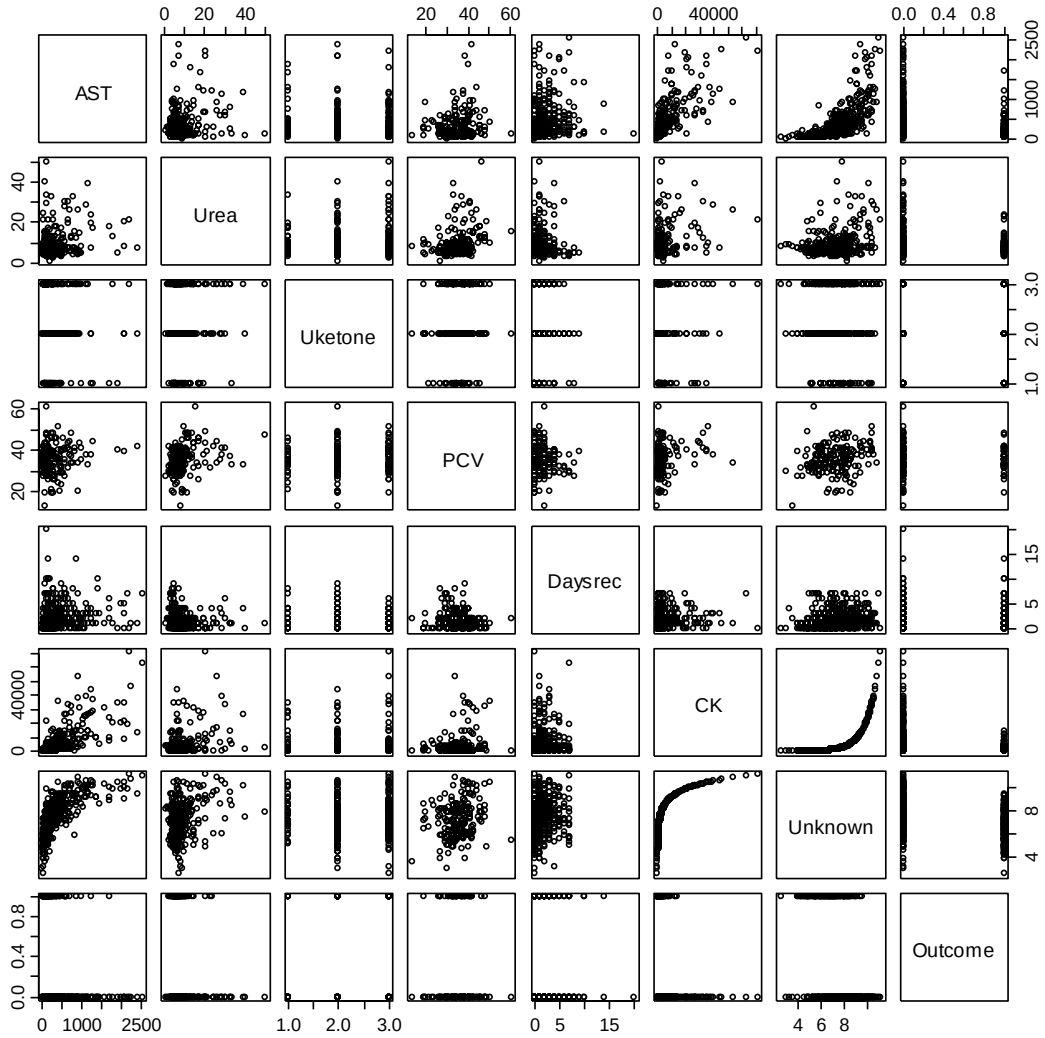


3.5 Relationships between more than two variables

3.5.1 Scatterplot Matrix

A scatterplot matrix is an array of the scatterplots of each pair of variables. This gives a quick visual summary of how the pairs of variables interact - for example highly correlated variables stand out, as do variables with a strong non-linear relationship.

```
> plot(rcdata[c("AST", "Urea", "Uketone", "PCV", "Daysrec", "CK",
 + "Unknown", "Outcome")])
```

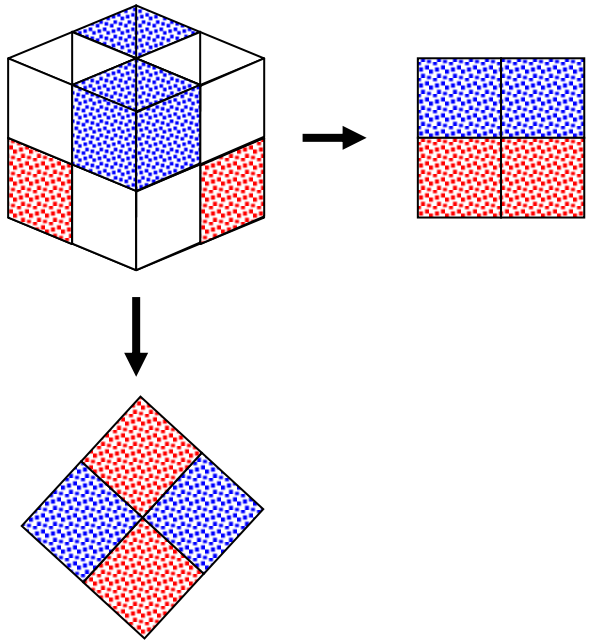


Note the relationship between CK and Unknown. It appears to be an exponential relationship, which can be tested using R.

```
> cor(rcdata[,"CK"], rcdata[,"Unknown"],use="complete.obs")
[1] 0.8226674
> cor(rcdata[,"CK"],exp(rcdata[,"Unknown"]), use="complete.obs")
[1] 1
```

Thus we have the deterministic relationship CK $\propto$ exp(Unknown).

Of course, this is not really a multidimensional technique and may miss entirely higher dimensional structure. For example, consider a cube divided into 8 sub-cubes, which are alternatively left blank or filled with uniform random points. Any two dimensional projection of this data set will reveal no structure whatsoever.



3.5.2 Interactive Projections

Suppose we have p variables $X = (X_1,...,X_p)$, and $F(X,\beta)$ is some projection onto two or three dimensional space. With an interactive graphic system, we can plot the projection for some set of parameters β, and then dynamically alter the parameter set β and see if any interesting features emerge. We could even allow the system to alter β in some random fashion (although a smooth change might be more accessible to the eye), and simply watch and wait. Alternatively we can use some more intelligent techniques (such as projection pursuit) to try and find projections of interest.

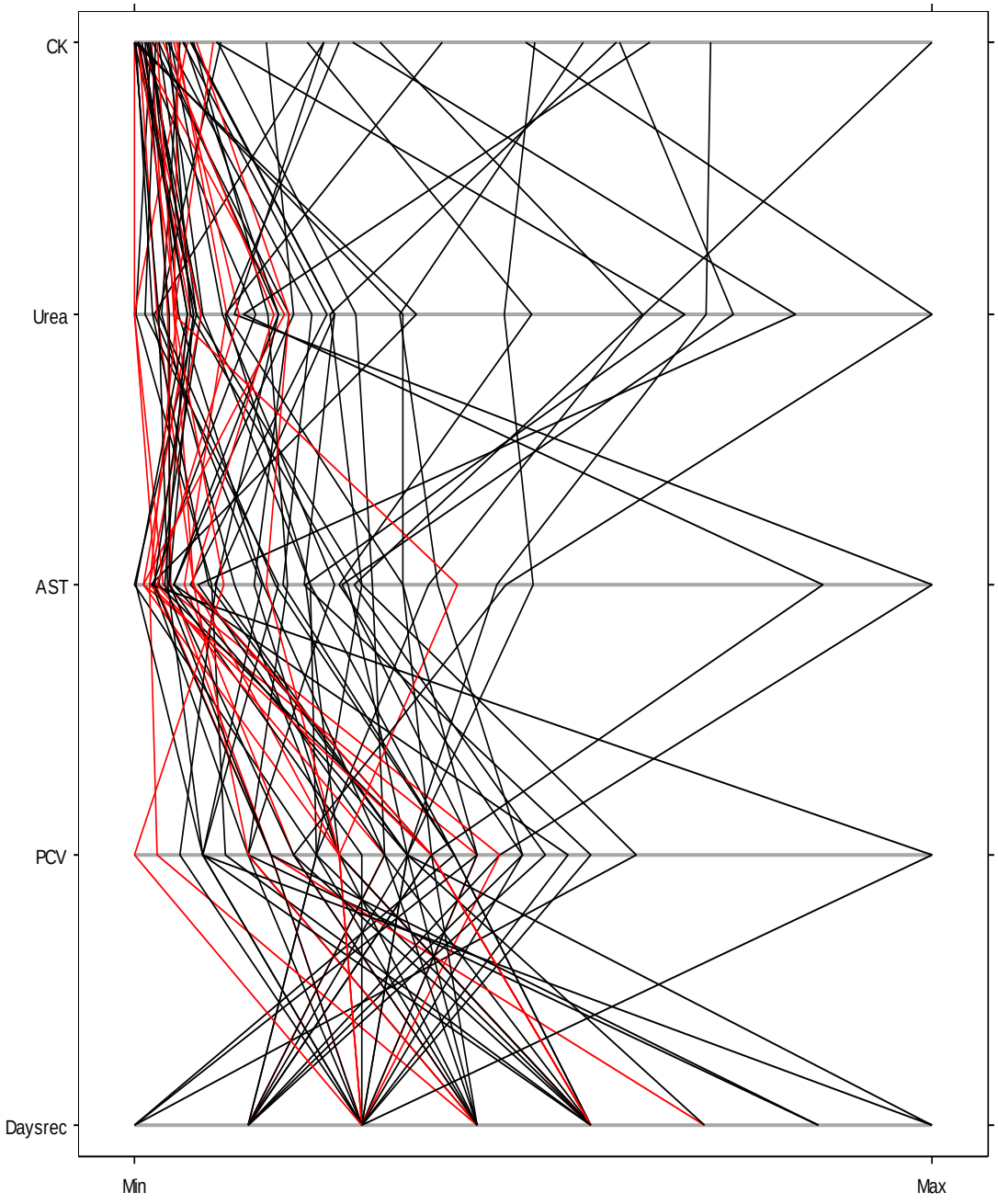
3.5.3 Parallel Coordinates Plot

Parallel coordinates plots show variable values plotted on parallel axes (one for each variable), representing each case as a piecewise linear plot connecting the measured values for that case. For example, we might have a dataset of patient responses to a battery of medical tests. An axis point is given for each test on the vertical axis for that test - the horizontal axis simply marks the position of each test result. A single patient is represented by a line joining up each test result for that patient.

Clearly this graph will look quite different if the order of the vertical axes is switched around, and one may use such a technique to search for hidden associations. Additionally, in investigating a particular pathology, we may color the lines corresponding to those patients and look for a pattern in the data.

```
> library(lattice)
```

```
> rcdata2 <- na.omit(rcdata)
> outcome<-rcdata2[,"Outcome"]+1
> parallel(~rcdata2[,c("Daysrec","PCV","AST","Urea","CK")],
 + col=outcome)
```



3.6 Projections

3.6.1 Principle Components Analysis

Principle components analysis (PCA) is one method of determining a projection that is "interesting" in some sense. In PCA we seek the projection onto k-dimensions that minimizes the sum of the squared differences between the data points and their projections. In the case of a two dimensional projection this is achieved by (1) the linear combination of the variables that has maximum sample variance together with (2) the linear combination that has maximum sample

variance subject to being uncorrelated with the first linear combination.

- k^{th} principle component is the k^{th} eigenvector of the covariance matrix of the data (Derivation HMS 3.6).
- Scree plots for estimating how big k should be to adequately describe the data.
- Advantages: computationally efficient (if data has dimension p and n observations, computing the principle components is $O(np^2 + p^3)$, so scales well as function of data set size n); can be used to reduce dimensionality of data to make it more accessible to further analysis; sometimes the components themselves may give insight to the inherent structure in the data.
- Disadvantages: lose information, which may contain important structure.

4 Market Basket Analysis (Association Analysis)

4.1 Introduction

"Market Basket Analysis", also called "Association Analysis" or "Association Rules Mining", is used to describe the process of discovering association rules from a transactional database. Given a database in which objects are grouped by context we seek sets of objects that tend to associate with each other. For example, the objects may be retail products from a store, and the context (or grouping) the items that appear in a single customer's shopping basket, on a single shopping visit.

4.1.1 Confidence, Support and Lift

To put this more formally, suppose $\Omega = \{\omega_1,\ldots,\omega_k\}$ are the objects, and our observations are a sequence of groupings $C_1,\ldots,C_n$ such that $C_i \subseteq \Omega$. Let $A,B \subseteq \Omega$, then we say that B is *associated* with A if the appearance of A in an observation usually implies that B will occur in that observation also. In this case we say that the rule $A \Rightarrow B$ is *confident* in the database. That is

$$P(B \subseteq C_i \mid A \subseteq C_i) \geq \tau_{conf}$$

where $\tau_{conf} \geq 0$ is a user supplied threshold. In many cases, we will only be interested in an association rule if it occurs in more than a certain percentage of observations. This percentage is called the *support* for the rule, and a rule is said to be *supported* if

$$P(B \subseteq C_i, A \subseteq C_i) \geq \tau_{\sup}$$

where $\tau_{\sup} \geq 0$ is again supplied by the user.

Note that low support doesn't always mean a rule is worthless. For example, if we search a database of credit card transactions for evidence of fraudulent behavior, we would expect any relevant rule to have low support since it is intrinsically a rare event. Never-the-less, if the confidence is high, the association rule may still be useful. An example of such a rule might be

> "Spend more than $200 on fuel in a single day implies fraudulent behavior".

This may pick up fraud from (say) a gas station attendant.

Upon discovering a rule, we may also wish to say something about the predictive power offered by that rule. For example, the rule "milk$\Rightarrow$bread" may have very high confidence and support, but the fact of the matter is that almost everyone buys bread (with or without milk) anyway, so the association rule doesn't tell us much. This concept is known as the *lift* given by a rule, and is the ratio of the probability of discovering B in the presence of A to the probability of discovering B without any prior knowledge. That is:

$$\frac{P(B \subseteq C_i \mid A \subseteq C_i)}{P(B \subseteq C_i)}.$$

In general we seek association rules with lift significantly greater than 1.

The solution to a market basket analysis problem is a list of all association rules satisfying our confidence, support and lift requirements. These are usually either that all of confidence, support and lift are above some thresholds, or that confidence and lift are above some very high thresholds.

4.1.2 Example

Consider the following trivial example of the concepts of support, confidence and lift. Suppose we consider the presence or absence of 6 items labeled A, B, C, D, E & F in the baskets of 4 individuals. The support of all (non-trivial) one and two item sets is then simply the proportion of baskets the item set occurs in. Note that the support can also be computed for three and 4 item sets. For example, the set ABC has support 0.25, and the set BCE has support 0.5.

Customer	Basket Contents
1	A, C, D
2	B, C, E
3	A, B, C, E
4	B, E

Item Set	Item Support
A	0.5
B	0.75
C	0.75
D	0.25
E	0.75
AC	0.5
BC	0.5
BE	0.75
CE	0.5
AB	0.25
AD	0.25
CD	0.25
AE	0.25

If we assume a support threshold of 0.5, then the following sets are supported in the database: A, B, C, E, AC, BC, BE, CE, BCE. Now consider the rule CE⇒B. The confidence of this rule is then the support of BCE divided by the support of CE, which equals 1. The lift of this rule is the confidence divided by the support of B, which equals 4/3. Thus we may conclude that the rule CE⇒B is supported in the set, has high confidence and lift greater than 1.

4.2 Algorithms for Market Basket Analysis

Of course, the probabilities introduced above are rarely known exactly, and must be estimated from the data set. In practical market basket analysis algorithms, it is assumed that probabilities are estimated by simply counting the observations. This estimation is simplified somewhat by the *apriori principle: any subset of a supported itemset must also be supported,* or alternatively *any itemset containing an unsupported subset cannot be supported.* The validity of the apriori principle follows directly from the probability rule $P(A) \geq P(A \cap B)$.

Suppose for the moment we are concerned with rules that have high support. The major computational task we face is locating those sets of items which have support greater than threshold $\tau_{\sup}$. Of course, it is likely to be computationally infeasible to explicitly compute the support for all subsets of objects in the database (k items implies $2^k - 1$ subsets), however some simple observations can reduce our computations significantly. For example, suppose we begin by computing the support of each individual item. This can be done in a single pass over the dataset. Then the apriori principle implies that

any items with support less than τ_{sup} cannot occur in any supported association rule and may be ignored henceforth. On the second iteration we could consider all two-item sets, with items chosen from the set of supported items, and so on. Depending on the structure of the database (i.e. how many and what kind of association rules exist) this approach can reduce the dimensionality of the search very quickly. At the end of this we have a collection of supported subsets of the set of items together with the support of each. With no further passes over the dataset we can compute the confidence and lift for all possible association rules from this set.

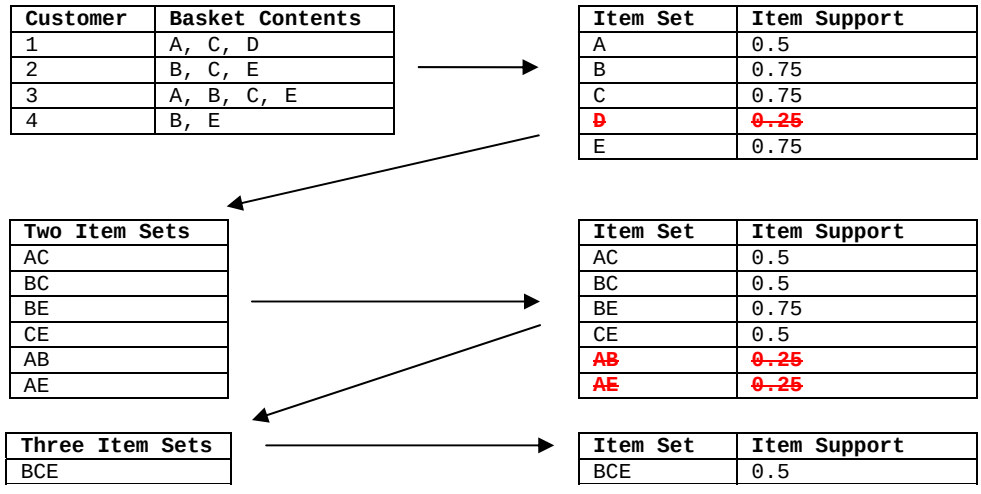
Some algorithms of this sort may be appropriate for a dataset located at a single site, but in the case of multiple site data warehouses we have the added complexity issue of communication time. In fact, we may generally assume that the constraining factor will be the amount of data requested from each site to perform the analysis. By focusing on reducing the amount of communication required between sites we can construct an efficient algorithm. Once again the approach is to try and identify item sets with large support, although now the large support is with respect to the global dataset.

One method is to instruct each site to report all item sets with locally large support. A single report of a locally large item set is enough to trigger the collection of support counts for that item set from all sites. After this we can assess if the item set is globally large, and then proceed to search for association rules.

More sophisticated algorithms have been proposed in which a single report of a locally large item set is not sufficient to trigger a global count. Instead sites perform a negotiation procedure to determine which item sets are globally large and which are not.

4.2.1 Example

The trivial example from the previous subsection is sufficient to demonstrate the process of extracting supported subsets. The algorithm consists of scanning the database and extracting all the supported itemsets of size 1, then using that subset of items to extract all the supported itemsets of size 2, and so on. A strikethrough indicates the itemset is not supported (support threshold is 0.5), and is therefore not considered in any further computations.



Customer	Basket Contents
1	A, C, D
2	B, C, E
3	A, B, C, E
4	B, E

Item Set	Item Support
A	0.5
B	0.75
C	0.75
~~D~~	~~0.25~~
E	0.75

Two Item Sets
AC
BC
BE
CE
AB
AE

Item Set	Item Support
AC	0.5
BC	0.5
BE	0.75
CE	0.5
~~AB~~	~~0.25~~
~~AE~~	~~0.25~~

Three Item Sets
BCE

Item Set	Item Support
BCE	0.5

After computing the support of all one item subsets, the apriori principle requires that only two item subsets made up of supported single item subsets are candidates for supported two item subsets. Similarly, to be a candidate for a supported three item set, all one and two item subsets of the set must be supported.

5 Predictive Data Mining

[R:1996]

5.1 Introduction

Suppose we have a data set with a number of response variables $Y = (Y_1,\ldots,Y_p)$ which we would like to model as some function f of a set of predictor variables, $X = (X_1,\ldots,X_q)$. In particular we seek some function such that $Y = f(X) + \varepsilon$ where $\varepsilon = (\varepsilon_1,\ldots,\varepsilon_p)$ is the random vector describing the error. The data set contains (say) n observations of X and Y, and our objective is to fit the model f in such a way as to minimize the errors (according to some score function).

The model f may serve any of a whole range of purposes. For example, if Y is categorical, then f is a classifier that assigns one or more classes to the predictors X. For the moment we shall interest ourselves in the case that the X are some set of observed values which have associated responses Y that we wish to predict. We have a data set for which n of the X observations have been associated with their correct Y values, and we wish to construct a model that will predict Y values with satisfactory accuracy when evaluated on unseen cases with only X observations (which are assumed to be drawn from the same distribution as our data set).

For example, Y might be the life expectancy of a patient with a particular terminal illness, and X a set of measurable quantities such as age, blood pressure, duration of illness etc. From a given set of observations we may be able to fit a model which will predict with some level of accuracy the life expectancy in new cases.

5.2 General Discussion of Fitting Models to Data

5.2.1 Characteristics of the Sample

Before we attempt to fit a model to our data set, we need to understand what sort of data set we are dealing with. This will assist both in choosing a model and fitting scheme, and also in interpreting the results afterwards. For example, if the response variables are binary, we require a model giving binary output over the range of the inputs.

Many modeling schemes assume the data points are independent and identically distributed – is this true of our data? Remembering the characteristics of data warehouses, it is quite likely that the data points are neither independent, nor identically distributed. Therefore we need to examine what this means for the validity of our results. Can we assume the errors are normally distributed? Can we even assume that the sample is a random subset of the population?

In summary, we should examine the following data set issues before attempting to fit any model.

- Are the observations independent?
- Are the observations a representative of the population such as a random sample from the population?

- Can we make any assumption on the distribution of the output errors (e.g., Normal)?
- Are the data a time series?

5.2.2 Assessing the Fit of a Model – Score Functions

The whole point of fitting a model is to find some good description of the data. Therefore we need some way to determine which models are a good fit, and which are not. In particular, we need some function $S(Y, X, \beta)$, where Y is the response, X is the predictor, and β is the array of parameters for the model, such that S measures the fit of the model. Generally we will attempt to minimize or maximize S over the set of available parameters for our sample. As we shall see in the next section, choosing the score function is not as simple as it first seems. Since we are interested in predicting on unseen data, we are actually looking for a score function that measures the fit of the model on data other than the sample. That is, we want to find a score which will reward fitting the underlying structure in the data, but penalize a model fitting the noise that is specific to the training sample.

5.2.3 Using Data Wisely – Training and Validation Data

Every method of fitting a model to data has different requirements on data usage. Many methods use a two stage or multistage fitting process. In these instances we need to consider carefully how the data is used at each stage. For example, if one stage is fitting parameters to a particular model, and another is validating that model, it would generally be unwise to use the same data for both fitting and validating. A model will tend to fit (or overfit) its training data unusually well, a fact which will lead to spurious conclusions at the validation stage. If there is plenty of data available (as there usually is in a data mining context), it is advisable to separate out disjoint datasets for each stage of the fitting process.

If, after fitting a model, we wish to supply some estimate of the error rate on unseen data, then under no circumstances should we use any of the data used in fitting. It is a very common mistake to give results on the training data set as the expected error rate on unseen data, and this is simply not true. It is good practice to separate out a test data set at the beginning of the whole process, which is used only once, at the end, to provide an estimate of the performance on unseen data.

5.3 Score Functions, Model Validation, Overfitting etc

5.3.1 Score Functions for Predictive Models
([HMS:2001] 7.3.1)

In a prediction task, our training data comes with a "target" value Y, this being a quantitative variable for regression or a categorical variable for classification, and our data set $D = \{(\mathbf{x}(1), y(1)), \ldots, (\mathbf{x}(n), y(n))\}$ consists of pairs of input vectors and target values. Let $\hat{f}(\mathbf{x}(i); \theta)$ be the prediction generated by the model for individual i, $1 \le i \le n$, using parameter values θ. Let $y(i)$ be the actual observed value (or "target") for the i^{th} individual in the training data set.

Score functions should clearly measure the difference between the prediction generated by the model and the known target value. Common error functions in a regression context are:

- Sum of squared errors: $S_{SSE}(\theta) = \sum_{i=1}^{n} (\hat{f}(\mathbf{x}(i);\theta) - y(i))^2$
- Sum of absolute errors (L_1 norm): $S_{L_1}(\theta) = \sum_{i=1}^{n} \left| \hat{f}(\mathbf{x}(i);\theta) - y(i) \right|$

The sum of absolute errors is less sensitive to extreme values than the SSE, but is not differentiable, so may not be suitable for some regression methods.

For categorical Y (classification problems) the most common score function is the misclassification rate:

$$S_{0/1}(\theta) = \frac{1}{n} \sum_{i=1}^{n} I(\hat{f}(\mathbf{x}(i);\theta), y(i))$$

where $I(a,b)$ is 1 if $a \neq b$ and 0 otherwise.

All of these score functions suppose that all individuals in the training set should be treated equally. This is a very common assumption, but must be explicitly considered for each data set before choosing a score function. For example, suppose some data in our training set is known to come from a more accurate measurement process than the remainder – we may wish to weight the scores to emphasize those values. Alternatively we may have time series data for which more recent observations are of greater importance to us. In the classification context, we may consider some kinds of misclassifications worse than others, such as diagnosing cancer or not. In cases such as these we may wish to augment the simpler score functions with a weighting scheme that emphasizes the importance of some observations over others.

In any case, for any given data mining task we should attempt to construct a score function that is as simple as possible, but also reflects the prior knowledge we have about the data. Note that in many software packages there is a restricted set of score functions available, so we cannot always use the score function we would like.

5.3.2 Score Functions for Models of Different Complexity

As noted before, more flexible models can often fit a given training set to whatever accuracy we like. However for many prediction problems a good training set error does not necessarily correspond to a good model, since low training set error may indicate *overfitting*, or fitting the specific characteristics of the training data (noise) rather than the general characteristics of the population. In general, more complex models are more flexible, and so more prone to overfitting.

One of the methods employed to combat this tendency is to add a term to the score function penalizing the complexity of the model. Then we can fit models of different complexities and select based on the *generalized error* (i.e. fit error + complexity penalty). In this section we briefly outline some of the penalty functions commonly used.

The first two complexity penalties also include the *negative log-likelihood* of the model. This term arises from an estimate of the

probability that our training data set was produced by the model under consideration.

$$S_L(\theta) = -\sum_{i=1}^{n} \log \hat{p}(\mathbf{x}(i);\theta)$$

If a given model M has k parameters, and if $\hat{\theta}$ are the fitted parameters for M, then the *Akaike Information Criteria (AIC)* is defined as

$$S_{AIC}(M) = 2S_L(\hat{\theta};M) + 2k \ .$$

An alternative penalty, the *Bayesian Information Criteria (BIC)*, also takes into account the sample size n:

$$S_{BIC}(M) = 2S_L(\hat{\theta};M) + k \log n.$$

In both cases the complexity penalty grows linearly with the number of parameters, which is reasonable. In the BIC case the penalty also grows logarithmically with the size of the training set. For large *n* the effect of this is offset by the fact that the negative log-likelihood term is linear in *n* and thus will dominate. However for smaller *n* it becomes significant, reflecting the intuition that we "trust" the training error less for a small training set.

An alternative approach (called *weight decay* in the neural network literature) adds a penalty term of the form

$$P_{WD}(M) = \lambda \sum_{i=1}^{k} \hat{\theta}_i^2$$

to the raw score function, where $\hat{\theta}_i$ is the value of the i^{th} parameter, and $\lambda > 0$ is a user-specified constant. This penalty doesn't penalize the number of parameters so much as the *use* of those parameters. Clearly the term encourages small parameter values, which can be shown to effectively restrict the degrees of freedom of the model. Using this term is closely related to the *ridge regression* technique, and is very popular in neural network models.

There are many other penalty terms with similar additive forms to those above (see [R:1996]), but the basic idea is the same – we seek an additive complexity penalty that "balances" the training set error.

A different approach, based on the Bayesian framework attempts to estimate the posterior probability of each model directly, and choose the model with the highest probability (see [R:1996],[HMS:2001]).

5.3.3 Score Functions Using External Validation

An alternative method for comparing models is to simply train a sequence of models of differing complexities, and then compare their performance on an independent validation set. As noted above, the validation set should not play any part in the training process before the model selection, and should be drawn from the same distribution as the training set.

If there is a shortage of training data, in which case it may not be practical to set aside an independent validation set, then we can use a cross-validation technique. For *m*-fold cross validation, the

training set is split into m disjoint subsets. For each different model, we iterate through each of the m subsets, training that model on the data remaining after the selected subset is removed. The model is then evaluated on the selected subset. After iterating through all the subsets, an average validation error is taken as a measure of the ability of that model to generalize. This may be compared between the different models and a final selection made.

5.3.4 Evaluating Models

Once we have selected a model, we often want to report an error rate for the model performance on unseen data. Once again, it is very important that the data we use for evaluation has not been used in any stage of the training process (including model selection with a validation set). This is a very common error, and leads to optimistically biased estimates.

In situations where acquiring sufficient data is problematic, we may also use bootstrap or jackknife estimates to estimate the expected error (see below).

5.4 Classic Techniques: Multiple Regression
[SS:1990]

Here we briefly review multiple regression to refresh concepts that should already be familiar from earlier courses.

5.4.1 Problem Formulation and Solutions

Suppose we have a data set consisting of n random realizations of a single response variable Y, which is to be modeled as a function of k independent predictors $X_1,\ldots,X_k$. That is, we seek a model of the form

$$Y = \beta_0 + \beta_1 X_1 + \cdots + \beta_k X_k + \varepsilon$$

where $\beta = (\beta_0,\ldots,\beta_k)^T$ (the $(\ldots)^T$ denotes transpose)is the parameter vector and ε is the vector of residuals (or the error). Then if we let

$$\mathbf{X} = \begin{pmatrix} 1 & x_{11} & \cdots & x_{1k} \\ \vdots & \ddots & & \vdots \\ \vdots & & \ddots & \vdots \\ 1 & x_{n1} & & x_{nk} \end{pmatrix}, \quad \mathbf{y} = \begin{pmatrix} y_1 \\ \vdots \\ \vdots \\ y_n \end{pmatrix} \text{ and } \varepsilon = \begin{pmatrix} \varepsilon_1 \\ \vdots \\ \vdots \\ \varepsilon_n \end{pmatrix}$$

the multiple regression problem can be simply written as

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

If we choose to fit the model using a least squares estimate, this amounts to finding β minimizing the error

$$\begin{aligned} S &= (\mathbf{y} - \mathbf{X}\beta)^T (\mathbf{y} - \mathbf{X}\beta) \\ &= \mathbf{y}^T\mathbf{y} - \beta^T\mathbf{X}^T\mathbf{y} - \mathbf{y}^T\mathbf{X}\beta + \beta^T\mathbf{X}^T\mathbf{X}\beta \\ &= \mathbf{y}^T\mathbf{y} - 2\beta^T(\mathbf{X}^T\mathbf{y}) + \beta^T(\mathbf{X}^T\mathbf{X})\beta \end{aligned}$$

To minimize this we differentiate with respect to β

$$\frac{\partial S}{\partial \beta} = -2\mathbf{X}^T\mathbf{y} + 2\mathbf{X}^T\mathbf{X}\beta$$

and set to 0. Let $\hat{\beta}$ be the least squares estimate of β, then $\hat{\beta}$ satisfies the equation

$$(\mathbf{X}^T\mathbf{X})\hat{\beta} = \mathbf{X}^T\mathbf{y} .$$

In the case $\left(\mathbf{X}^T\mathbf{X}\right)$ is invertible, this has the unique solution

$$\hat{\beta} = \left(\mathbf{X}^T\mathbf{X}\right)^{-1}\mathbf{X}^T\mathbf{y} .$$

If $\left(\mathbf{X}^T\mathbf{X}\right)$ is singular, then generalized inverses may be used to get a non-unique estimate for $\hat{\beta}$:

$$\hat{\beta} = \left(\mathbf{X}^T\mathbf{X}\right)^{-}\mathbf{X}^T\mathbf{y} = \mathbf{X}^{-}\mathbf{y}$$

where $\mathbf{X}^{-}$ is the generalized inverse of $\mathbf{X}$.

Once the model is fitted, we can obtain an estimate $\hat{\mathbf{y}}$ of the target values $\mathbf{y}$ according to the model:

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

in the case that $\left(\mathbf{X}^T\mathbf{X}\right)$ is invertible. The *errors,* or *residuals* of the fit are then given by

$$\begin{aligned}\mathbf{e} &= \mathbf{y} - \hat{\mathbf{y}} \\ &= \mathbf{y} - \mathbf{X}(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} \\ &= (\mathbf{I} - \mathbf{X}(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T)\mathbf{y} \\ &= (\mathbf{I} - \mathbf{H})\mathbf{y}\end{aligned}$$

where $\mathbf{I}$ is the identity matrix, and $\mathbf{H} = \mathbf{X}(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T$ is the *"Hat"* matrix of the predictors.

5.4.2 Interpretation of Model and Output

We have seen that parameters of a linear regression model are fitted in order to minimize the *sum of squared errors,* or the *residual sum of squares*

$$SS_{ERR} = \left\|\mathbf{y} - \hat{\mathbf{y}}\right\|^2 = \mathbf{e}^T\mathbf{e} .$$

Now suppose we fit the data by simply choosing the sample mean $\bar{y}$ of the response variable regardless of the predictor value. This is, in some sense, the simplest model for the response variable. The *total sum of squares* is defined as the sum of squared errors for this simplest model

$$SS_{TOT} = \left\|\mathbf{y} - \mathbf{1}\bar{y}\right\|^2 .$$

The difference between the residual sum of squares and the total sum of squares can be attributed to the regression for that model, and is called the *regression sum of squares*. This is the sum of the squared differences of the predicted values $\hat{\mathbf{y}}$ from the sample mean:

$$SS_{REG} = \left\|\hat{\mathbf{y}} - \mathbf{1}\bar{y}\right\|^2 .$$

The ratio between the regression sum of squares and the total sum of squares is called the *multiple correlation coefficient*, or R^2-value

$$R^2 = \frac{SS_{REG}}{SS_{TOT}} = \frac{\left\|\hat{\mathbf{y}} - \mathbf{1}\bar{y}\right\|^2}{\left\|\mathbf{y} - \mathbf{1}\bar{y}\right\|^2} = 1 - \frac{SS_{ERR}}{SS_{TOT}}.$$

The R^2-value gives a measure of how much of the Y variation is explained by the model. A value near 1 tells us that the model explains most of that variation. As one adds more predictor variables to a regression model, the value of R^2 increases since there is more "opportunity" to explain the variation in the response variable Y.

The number of independent components contributing to each sum of squares is called the number of *degrees of freedom* (df) for that sum of squares. The degrees of freedom for the total sum of squares is $n-1$, for the residual sum of squares it is $n-1-k$, and for the regression sum of squares it is k. As well as sums of squares (SS) we define the *mean square* as the sum of squares divided by its degrees of freedom: $MS = SS/df$ in general; thus $MS_{REG} = SS_{REG}/k$, $MS_{TOT} = SS_{TOT}/(n-1)$ and $MS_{ERR} = SS_{ERR}/(n-1-k)$. In order to compute an *adjusted* R^2-value, one that takes into account (penalizes for) the number of predictor variables (k) used, we use the following formula:

$$R^2_{ADJ} = 1 - \frac{MS_{ERR}}{MS_{TOT}} = 1 - \frac{\left(1-R^2\right)(n-1)}{n-1-k}.$$

Let us suppose for a moment that the response variable is not related to the predictors in any linear fashion at all. In this case, the true regression coefficients ought to be zero. Under this assumption, and further assuming that the ε_i are independently distributed as $N(0,\sigma^2)$, then

$$\frac{SS_{REG}/k}{SS_{ERR}/(n-k-1)} \sim F(k, n-k-1) \quad \text{under } \mathrm{H}_0.$$

This is the basis for a hypothesis test to assess whether the claim that the regression coefficients are significant can be supported.

Note that if the errors (or residuals) ε_i are really drawn independently from a $N(0,\sigma^2)$ distribution, then the *sample quantiles* of the errors should be close to the quantiles of a normal distribution.

ASIDE

Recall that the x^{th}*-percent quantile* of a distribution F is the point q for which $F(q) = x/100$.

Therefore, if we plot the sample quantiles of the errors against the true quantiles of a $N(0,1)$ distribution, we should get a straight line. This is the idea behind a *quantile plot*. A quantile plot is a quick way to assess our assertion that the errors are normal. We shall see an example of a quantile plot later on.

If the errors are in fact independent of the model, then there also should be no clear relationship between the errors and the fitted values of the model. A quick visual assessment of this may be made simply by plotting the residuals against the fitted values. In the case that the errors are indeed independent of the model, we would expect an even cloud of residuals along the range of the fitted values, and centered around zero. We shall see a plot of this type in the example later on.

5.4.3 Stepwise Model Building

The F-statistics introduced in the last subsection also provide the basis for a stepwise model building process, wherein coefficients are added or deleted based on a hypothesis test concerning their significance in the model. More specifically, suppose we have two models with p and $p+q$ parameters respectively. We wish to test the alternative hypothesis that the extra parameters significantly improve the fit of the model against the null hypothesis that they make no significant addition to the model. Under the null hypothesis, and once again assuming that the errors ε_i are independently distributed as $N(0,\sigma^2)$,

$$\frac{\left(SS_{ERR}(p)-SS_{ERR}(p+q)\right)/q}{SS_{ERR}(p+q)/(n-p-q-1)} \sim F(q,n-p-q-1)$$

where $SS_{ERR}(k)$ denotes the sum of squared errors for the model with k parameters. If the null hypothesis is rejected, we conclude that the extra parameters significantly improve the fit and choose the augmented model. There are a number of stepwise implementations based on this test. For example, we may fit a large model, and then test each parameter (or some subsets of parameters) for significance, pruning the model to only the parameters deemed to be statistically significant (*backwards stepwise*). Alternatively we may start with a minimal model and augment the model by adding new parameters and testing the new model against the old using the F-test above (*forwards stepwise*).

Modern regression packages tend to allow both forwards and backwards stepwise model building. In practice a combination of the two is often used. That is, fit the model using the variables that are assumed to be of most significance in the relative domain knowledge. Then prune back the parameters that are deemed not significant. We may then add further parameters, which could be either "secondary" variables whose significance is not yet known, or transformations or combinations of existing variables.

5.4.4 Example

Linear regression, including stepwise model building, is easy to implement in R. For a simple model all that is required are the regression formula and the observations. The model is then fitted using the `lm' function.

```
> # read the data
> rcdata <- read.table("C:\\RData\\Cows\\RC Subset1.txt",header=TRUE)
>
> fmla <- as.formula("AST~Urea+Uketone+Outcome+Daysrec+Calving")
>
> summary(astlm <- lm(fmla, rcdata))
```

```
Call:
lm(formula = fmla, data = rcdata)

Residuals:
    Min      1Q  Median      3Q     Max
-709.78 -296.69  -90.97  166.84 1739.76

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  823.754    209.931   3.924 0.000148 ***
Urea          17.062      5.949   2.868 0.004909 **
Uketone     -142.518     68.576  -2.078 0.039893 *
Outcome     -235.595     97.374  -2.419 0.017096 *
Daysrec      -56.118     27.161  -2.066 0.041044 *
Calving       49.671    133.394   0.372 0.710301
---
Signif. codes:  0 `***' 0.001 `**' 0.01 `*' 0.05 `.' 0.1 ` ' 1

Residual standard error: 451.5 on 116 degrees of freedom
Multiple R-Squared: 0.1944,     Adjusted R-squared: 0.1596
F-statistic: 5.597 on 5 and 116 DF,  p-value: 0.0001174
```

The stepwise model building feature allows both forward and backward steps. The minimal and maximal formulae need to be specified before implementing stepwise regression using the `step' function.

```
> maxfmla <- as.formula("AST ~ Urea + Uketone + Myopathy + Outcome +\
  Calving + Daysrec + CK + Unknown")
>
> minfmla <- as.formula("AST ~ Outcome")
>
> sastlm<-step(astlm,scope=list(lower=minfmla, upper=maxfmla),
 + test="F")

Start:  AIC= 1497.33
 AST ~ Urea + Uketone + Outcome + Daysrec + Calving

           Df Sum of Sq      RSS      AIC F value     Pr(F)
+ Myopathy  1  10150664 13498062     1431 86.4810 1.112e-15 ***
+ Unknown   1   8232592 15416135     1447 61.4128 2.557e-12 ***
+ CK        1   7860064 15788662     1450 57.2504 1.032e-11 ***
- Calving   1     28267 23676993     1495  0.1387  0.710301
<none>                 23648726     1497
- Daysrec   1    870284 24519010     1500  4.2689  0.041044 *
- Uketone   1    880526 24529252     1500  4.3191  0.039893 *
- Urea      1   1676865 25325591     1504  8.2252  0.004909 **
---
Signif. codes:  0 `***' 0.001 `**' 0.01 `*' 0.05 `.' 0.1 ` ' 1

Step:  AIC= 1430.91
 AST ~ Urea + Uketone + Outcome + Daysrec + Calving + Myopathy

           Df Sum of Sq      RSS      AIC F value     Pr(F)
+ CK        1   1759751 11738311     1416 17.0903 6.836e-05 ***
+ Unknown   1   1418687 12079375     1419 13.3890 0.0003848 ***
- Daysrec   1     26172 13524234     1429  0.2230 0.6376727
- Urea      1     40117 13538178     1429  0.3418 0.5599472
- Uketone   1    182097 13680159     1431  1.5514 0.2154574
<none>                 13498062     1431
- Calving   1    242482 13740544     1431  2.0659 0.1533430
- Myopathy  1  10150664 23648726     1497 86.4810 1.112e-15 ***
---
Signif. codes:  0 `***' 0.001 `**' 0.01 `*' 0.05 `.' 0.1 ` ' 1

Step:  AIC= 1415.87
 AST ~ Urea + Uketone + Outcome + Daysrec + Calving + Myopathy +
    CK
```

```
           Df Sum of Sq      RSS      AIC F value     Pr(F)
- Daysrec   1       200 11738511     1414  0.0019   0.96493
- Urea      1     12569 11750880     1414  0.1221   0.72745
+ Unknown   1    298399 11439912     1415  2.9475   0.08875 .
<none>                  11738311     1416
- Uketone   1    194786 11933097     1416  1.8917   0.17170
- Calving   1    226569 11964880     1416  2.2004   0.14074
- CK        1   1759751 13498062     1431 17.0903 6.836e-05 ***
- Myopathy  1   4050351 15788662     1450 39.3362 6.600e-09 ***
---
Signif. codes:  0 `***' 0.001 `**' 0.01 `*' 0.05 `.' 0.1 ` ' 1

Step:  AIC= 1413.87
 AST ~ Urea + Uketone + Outcome + Calving + Myopathy + CK

           Df Sum of Sq      RSS      AIC F value     Pr(F)
- Urea      1     12722 11751232     1412  0.1246   0.72471
+ Unknown   1    292677 11445834     1413  2.9151   0.09048 .
<none>                  11738511     1414
- Uketone   1    203642 11942153     1414  1.9950   0.16052
- Calving   1    234388 11972899     1414  2.2963   0.13243
+ Daysrec   1       200 11738311     1416  0.0019   0.96493
- CK        1   1785723 13524234     1429 17.4944 5.654e-05 ***
- Myopathy  1   4110479 15848990     1449 40.2696 4.532e-09 ***
---
Signif. codes:  0 `***' 0.001 `**' 0.01 `*' 0.05 `.' 0.1 ` ' 1

Step:  AIC= 1412
 AST ~ Uketone + Outcome + Calving + Myopathy + CK

           Df Sum of Sq      RSS      AIC F value     Pr(F)
+ Unknown   1    282503 11468730     1411  2.8327   0.09507 .
<none>                  11751232     1412
- Calving   1    221705 11972937     1412  2.1885   0.14175
- Uketone   1    237685 11988918     1412  2.3463   0.12831
+ Urea      1     12722 11738511     1414  0.1246   0.72471
+ Daysrec   1       352 11750880     1414  0.0034   0.95327
- CK        1   1818325 13569557     1428 17.9492 4.571e-05 ***
- Myopathy  1   4136332 15887564     1447 40.8310 3.586e-09 ***
---
Signif. codes:  0 `***' 0.001 `**' 0.01 `*' 0.05 `.' 0.1 ` ' 1

Step:  AIC= 1411.04
 AST ~ Uketone + Outcome + Calving + Myopathy + CK + Unknown

           Df Sum of Sq      RSS      AIC F value     Pr(F)
<none>                  11468730     1411
- Uketone   1    249025 11717754     1412  2.4970   0.11681
- Calving   1    270685 11739415     1412  2.7142   0.10219
- Unknown   1    282503 11751232     1412  2.8327   0.09507 .
+ Urea      1     22896 11445834     1413  0.2280   0.63389
+ Daysrec   1      6710 11462020     1413  0.0667   0.79662
- CK        1    613082 12081812     1415  6.1475   0.01461 *
- Myopathy  1   2676076 14144805     1435 26.8337 9.557e-07 ***
---
Signif. codes:  0 `***' 0.001 `**' 0.01 `*' 0.05 `.' 0.1 ` ' 1
```

The +/- values signify the feature is included or excluded from the model, respectively. At each step, the F-statistic is computed for the new model (with the relevant feature included or excluded), compared to the old model.

IMPORTANT ASIDE

In the R implementation of stepwise regression, the decision of which parameters to add or drop is not actually made based on the F-statistic, but rather on the AIC criteria. These choices are independent of whether the test="F" option is set or not (test="F" only causes the F-statistics to be reported as part of the fitting process, but does not use them in the analysis). Thus the final model obtained by the step function may not be the same as a model obtained by F-statistic based stepwise regression using (say) a 0.05 confidence level. For example, as

the second last step of the above stepwise regression, the `Unknown' variable is added into the model, despite the fact that the associated F-probability is 0.09507.

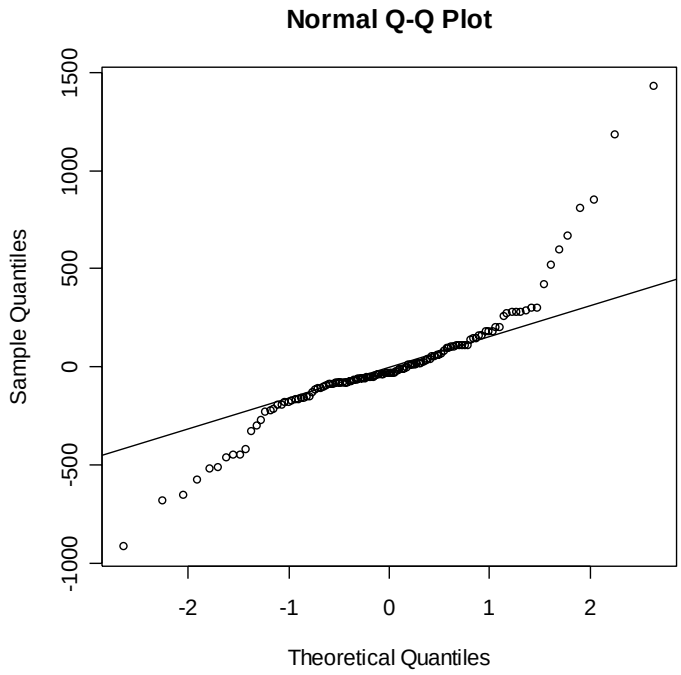
Here is a summary of the model obtained by stepwise regression:

```
> summary(sastlm)

Call:
lm(formula = AST ~ Uketone + Outcome + Calving + Myopathy + CK +
    Unknown, data = rcdata)

Residuals:
   Min     1Q Median     3Q    Max
-920.2 -109.1  -32.9  102.4 1430.6

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept) -1.072e+02  2.645e+02  -0.405   0.6860
Uketone     -7.370e+01  4.664e+01  -1.580   0.1168
Outcome     -1.515e+01  7.052e+01  -0.215   0.8303
Calving      1.496e+02  9.079e+01   1.647   0.1022
Myopathy     4.521e+02  8.728e+01   5.180 9.56e-07 ***
CK           9.441e-03  3.808e-03   2.479   0.0146 *
Unknown      5.274e+01  3.133e+01   1.683   0.0951 .
---
Signif. codes:  0 `***' 0.001 `**' 0.01 `*' 0.05 `.' 0.1 ` ' 1

Residual standard error: 315.8 on 115 degrees of freedom
Multiple R-Squared: 0.6093,     Adjusted R-squared: 0.5889
F-statistic: 29.89 on 6 and 115 DF,  p-value:     0
```

The new model produced by the stepwise procedure is a significantly better fit to the data (R^2_{ADJ}=0.59 as opposed to R^2_{ADJ}=0.16). Two of the features of the original model have been dropped (Urea & Dayesrec), and three new features have been added (Myopathy, CK & Unknown).

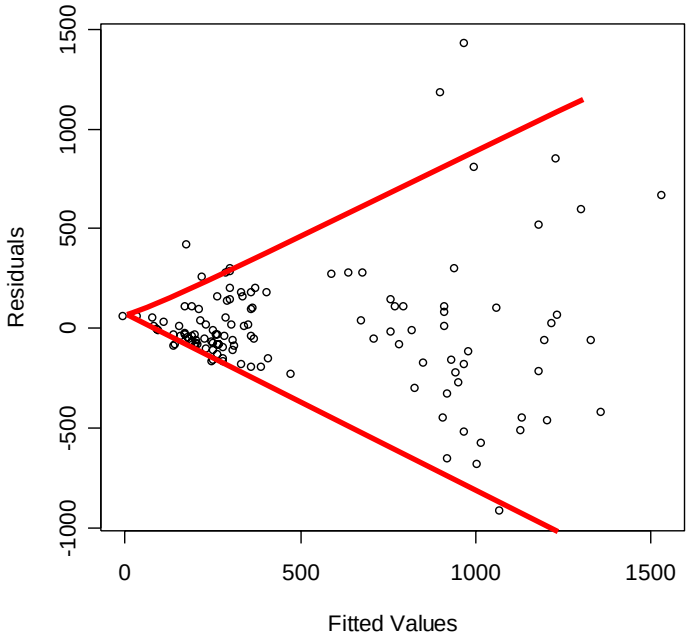
Using the qqnorm and qqline functions we can assess our assumption that the residuals are normal. Note that there are statistical tests that can formally test this hypothesis, however a visual heuristic is sufficient for our purposes.

```
> qqnorm(sastlm$residuals)
> qqline(sastlm$residuals)
```



Below is a plot of the residuals against the fitted values. In the case that the residuals are independent of the model, we would expect there to be a cloud of residuals, centered on zero, along the range of the fitted values. This does not appear to be true for our model – there is a significant positive correlation (0.55) between the magnitude of the residuals and the size of the fitted value. The red lines outline the apparent behavior of the residuals.

```
> plot(sastlm$fitted.values,sastlm$residuals,ylab="Residuals",
+ xlab="Fitted Values")
> cor(sastlm$fitted.values,abs(sastlm$residuals))
[1] 0.5512344
```



6.4.5 Shortcomings of the Multiple Regression Model

The obvious shortcoming of the multiple regression model is that it is a linear model. In practice many data sets encountered exhibit highly non-linear behavior, and may not be fitted well by a linear model. Of course, non-linear variables and interactions can be explicitly introduced (for example, add the square of each predictor, and the products of pairs of predictors), but these tend to burden the stepwise model building procedures, and usually require some deeper domain knowledge. Furthermore, the really important characteristics of the data may not be able to be captured by simple transforms of the inputs.

6 Classification: Logistic Regression

[CHP:2000], [HL:1989], [L:1997], [A:1990], [W:1999]

6.1 Introduction to Logistic Regression Using a Single Predictor

Logistic regression is suitable for data with binary response variables. For example predicting the incidence of heart failure based on certain physiological factors, or the incidence of loan defaults based on account and income information.

Rather than try to predict a 0-1 value, we try to model the *probability* that the response takes one of these values.

Consider the case with a single predictor X. If we used a simple linear function,

$$p(x) = P(Y = 1 \mid X = x) = \beta_0 + \beta_1 x$$

the problem is that $0 \le p(x) \le 1$, whereas the linear equation on the right is unbounded.

6.1.1 The Logistic Function

Instead we represent the relationship between $p(x)$ and X as a *logistic response function:*

$$(1) \qquad p(x) = P(Y = 1 \mid X = x) = \frac{e^{(\beta_0 + \beta_1 x)}}{1 + e^{(\beta_0 + \beta_1 x)}}$$

or equivalently

$$\ln \frac{p(x)}{1 - p(x)} = \text{logit}[p(x)] = \beta_0 + \beta_1 x \,.$$

The function (1) is now bounded between 0 and 1, so satisfying the requirements of a probability function.

> [NOTE: there are other functions that can achieve this behavior, e.g. the cumulative distribution of the normal curve, which gives the *probit model*]

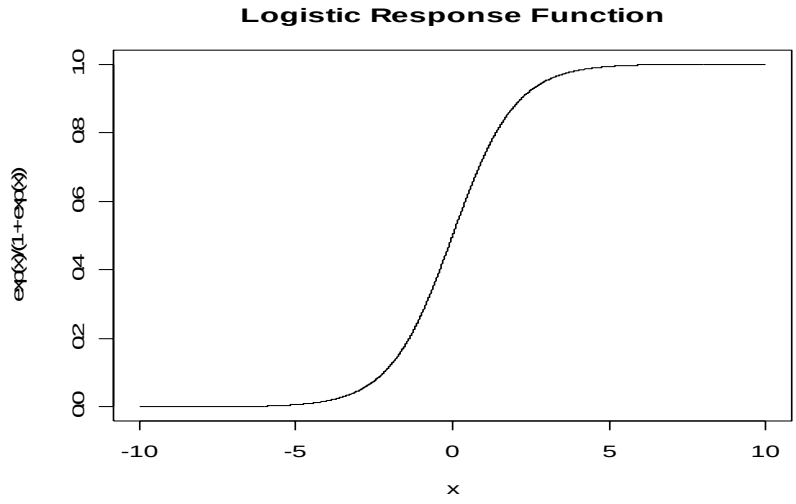
This is an example of a *generalized linear model (GLM).* A linear model takes the form:

$$E(Y \mid X = \mathbf{x}) = \beta_0 + \beta^T \mathbf{x}$$

whereas a generalized linear model transforms the response variable by some *link function* g:

$$g(E(Y \mid X = \mathbf{x})) = \beta_0 + \beta^T \mathbf{x} \,.$$

The logistic regression model is simply a GLM with a logit link function.



In this instance the regression problem becomes that of fitting a logistic function to model the response probabilities. However the model no longer has a closed form solution as in linear regression. Therefore we must re-examine our learning framework.

6.2 Derivation of Maximum Likelihood Estimators

Using the data (x_i, y_i), we want to determine a model $M(\beta_0, \beta_1)$ such that the probability of observing the outcomes $\{y_i\}$, given the model parameters β_0, β_1 and the predictors $\{x_i\}$, is maximized. Thus, assuming independence, we want the product of

$P(Y = y_i \mid X = x_i, M(\beta_0, \beta_1))$, over $i = 1, \ldots, n$, to be maximized. Writing

$$\begin{aligned}\zeta(x_i, y_i) &= P(Y = y_i \mid X = x_i, M(\beta_0, \beta_1)) \\ &= P(Y = 1 \mid X = x_i, M(\beta_0, \beta_1)) I(y_i = 1) + P(Y = 0 \mid X = x_i, M(\beta_0, \beta_1)) I(y_i = 0) \\ &= p(x_i) I(y_i = 1) + (1 - p(x_i)) I(y_i = 0) \\ &= p(x_i)^{y_i} \left(1 - p(x_i)\right)^{(1-y_i)}\end{aligned}$$

then the likelihood of our model given the data is

$$L(M(\beta_0, \beta_1)) = \prod_{i=1}^{n} \zeta(x_i, y_i) .$$

We wish to choose β_0 and β_1 that maximize this likelihood. This can be done simply by taking logs of both sides, in which case

$$\begin{aligned}\ln(L(M(\beta_0, \beta_1))) &= \sum_{i=1}^{n} (y_i \ln(p(x_i)) + (1 - y_i) \ln(1 - p(x_i))) \\ &= \sum_{i=1}^{n} \left[y_i \ln\left(\frac{p(x_i)}{1 - p(x_i)} \right) + \ln(1 - p(x_i)) \right].\end{aligned}$$

Substituting the model (1) and taking partial derivatives for β_0 and β_1 gives:

$$\frac{\partial}{\partial \beta_0} = \sum_{i=1}^{n} (y_i - p(x_i))$$
$$\frac{\partial}{\partial \beta_1} = \sum_{i=1}^{n} x_i (y_i - p(x_i))$$

therefore we may maximize the likelihood by choosing β_0 and β_1 which equate these derivatives to zero. (Question to think about: how do we know the solution really corresponds to a maximum of the (log) likelihood?)

6.3 Extension to Multiple Predictors

All of the above theory may be generalized in the obvious manner to a predictor X of any dimension. In this case β is a vector such that $\beta^T X = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_k X_k$, where we assume $X_0 \equiv 1$ and $X = (X_0, X_1, \ldots, X_k)$. Consider a data set of n observations of the predictors and their corresponding 0-1 response. Let $\mathbf{X}$ be the $n \times k$ matrix of predictor observations, $\mathbf{x}_i$ be the i^{th} predictor observation, and $\mathbf{y}$ the $n \times 1$ vector of responses. Then the logistic regression model is expressed as:

$$p(\mathbf{x}) = P(Y = 1 \mid X = \mathbf{x}) = \frac{e^{\beta^T \mathbf{x}}}{1 + e^{\beta^T \mathbf{x}}},$$

and the log likelihood can be shown to be:

$$\ln(L(M(\beta))) = \mathbf{y}^T \mathbf{X} \beta - \sum_{i=1}^{n} \ln\left(1 + e^{\beta^T \mathbf{x}_i}\right).$$

See the toy example on the following page. As in the single predictor case, the goal of logistic regression is to choose β such that this value is maximized.

Unlike the multiple linear regression, there is no formula giving a closed form solution to this problem. Therefore we must turn to iterative procedures in which an initial solution is iteratively refined until a satisfactory solution (i.e. satisfying some stopping condition) is reached. We shall detail one such method in the next section.

6.4 Fitting a Logistic Regression

Some version of the following algorithm is commonly employed to fit a logistic regression:

1. Obtain an initial guess for β, say $\mathbf{b}_0$. The choice $\mathbf{b}_0 = \mathbf{0}$ is often used in practice.
2. Set the iteration counter $j = 1$.
3. Approximate the log-likelihood close to $\mathbf{b}_{j-1}$ by a quadratic curve.

Example computation of likelihood

Data: 3 cases

i	x_1	x_2	y
1	2	3	1
2	1	1	0
3	–2	2	1

Calculations:

$\beta_0+\beta_1 x_1+\beta_2 x_2$	$p(x)$	$1-p(x)$	y
$\beta_0+2\beta_1+3\beta_2$	$\frac{e^{\beta_0+2\beta_1+3\beta_2}}{1+e^{\beta_0+2\beta_1+3\beta_2}}$	$\frac{1}{1+e^{\beta_0+2\beta_1+3\beta_2}}$	1
$\beta_0+\beta_1+\beta_2$	$\frac{e^{\beta_0+\beta_1+\beta_2}}{1+e^{\beta_0+\beta_1+\beta_2}}$	$\frac{1}{1+e^{\beta_0+\beta_1+\beta_2}}$	0
$\beta_0-2\beta_1+2\beta_2$	$\frac{e^{\beta_0-2\beta_1+2\beta_2}}{1+e^{\beta_0-2\beta_1+2\beta_2}}$	$\frac{1}{1+e^{\beta_0-2\beta_1+2\beta_2}}$	1

Likelihood:

$$\frac{e^{\beta_0+2\beta_1+3\beta_2}}{1+e^{\beta_0+2\beta_1+3\beta_2}} * \frac{1}{1+e^{\beta_0+\beta_1+\beta_2}} * \frac{e^{\beta_0-2\beta_1+2\beta_2}}{1+e^{\beta_0-2\beta_1+2\beta_2}}$$

4. Use the known formula for maximizing the quadratic curve to produce a new guess $\mathbf{b}_j$.
5. Check a stopping criterion. A common criterion is to stop iteration if $\ln\left(L(M(\mathbf{b}_j))\right) - \ln\left(L(M(\mathbf{b}_{j-1}))\right)$ is smaller than a pre-selected tolerance value. If the criterion is satisfied, report $\mathbf{b}_j$ as the estimator, otherwise increment j and go to step 3.

The major difference between implementations of this algorithm is to do with the approximation employed in steps 3 and 4. If this approximation is based on a Taylor series, which requires both the first and second derivatives of the log-likelihood, then the algorithm is exactly the *Newton-Raphson Method*, which is a common technique for solving nonlinear equations.

Alternatively the parameters may be updated by the *Fisher Scoring Method* (see [W:1999], 21.4). This method updates the parameters according to a weighted least squares estimate which depends on the current estimated parameters. In this case the algorithm is called the *Iteratively Reweighted Least Squares* algorithm. The advantage of this second method is it does not require explicit computation of the first and second partial derivatives of the log-likelihood. On the other hand, the Newton-Raphson method is widely used and efficient implementations are readily available.

6.6 Deviance and Pearsonian Residuals

In multiple linear regression, the residual sum of squares provides the basis for tests comparing mean functions. In logistic regression the situation is not so clear cut. There are two common substitutes proposed for the residual sum of squares, both having the same large sample distribution.

The *deviance* of the fit is twice the difference between the log-likelihood evaluated at the parameter values (that is using $p(\mathbf{x}_i)$ for the i^{th} observation), and the log-likelihood when the estimator at each observation is taken to be the 0-1 target values themselves. Thus:

$$\begin{aligned} G^2 &= 2\sum_{i=1}^{n}\left[y_i \ln\left(y_i\right) + (1-y_i)\ln\left(1-y_i\right)\right] - \left[y_i \ln\left(p(\mathbf{x}_i)\right) + (1-y_i)\ln\left(1-p(\mathbf{x}_i)\right)\right] \\ &= 2\sum_{i=1}^{n}\left[y_i \ln\left(\frac{y_i}{p(\mathbf{x}_i)}\right) + (1-y_i)\ln\left(\frac{1-y_i}{1-p(\mathbf{x}_i)}\right)\right] \\ &= \sum_{i=1}^{n}\mathrm{dev}(y_i, p(\mathbf{x}_i)) = -2 * \log\text{-likelihood} \end{aligned}$$

since the term added is $0 = 2\sum_{i=1}^{n}\left[y_i \ln\left(y_i\right) + (1-y_i)\ln\left(1-y_i\right)\right]$. The deviance term is the sum of the squared *deviance residuals,* where the i^{th} deviance residual is $r_i = \mathrm{sign}(y_i - p(\mathbf{x}_i))\sqrt{\mathrm{dev}(y_i, p(\mathbf{x}_i))}$.

Pearson's X^2 is an approximation to the deviance defined for logistic regression by:

$$X^2 = \sum_{i=1}^{n} \frac{(y_i - p(\mathbf{x}_i))^2}{p(\mathbf{x}_i)(1-p(\mathbf{x}_i))} .$$

In general the G^2 is to be preferred for model comparison (see [W:1999]). We shall use this below to introduce stepwise model building for logistic regression models.

6.7 Logistic Regression Output

We shall discuss the output to a logistic regression fit by means of an example in R.

6.7.1 Example

The `iris' dataset consists of 150 observations of iris flowers. Each observation has 4 measurements of the flower, and then the flower species as a dependent variable. We shall consider a logistic regression on a single species type, "versicolor". In R, two class logistic regression is implemented using the glm function with the family option set to binomial(logit).

```
> data(iris)
> tmpdata <- iris
> Versicolor <- as.numeric(tmpdata[,"Species"]=="versicolor")
>
> tmpdata[,"Species"] <- Versicolor
>
> fmla <- as.formula(paste("Species ~ ",paste(names(tmpdata)[1:4],
 + collapse="+")))
>
> ilr <- glm(fmla, data=tmpdata, family=binomial(logit))
> summary(ilr)

Call:
glm(formula = fmla, family = binomial(logit), data = tmpdata)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-2.1281  -0.7668  -0.3818   0.7866   2.1202

Coefficients:
             Estimate Std. Error z value Pr(>|z|)
(Intercept)    7.3785     2.4983   2.953 0.003143 **
Sepal.Length  -0.2454     0.6494  -0.378 0.705567
Sepal.Width   -2.7966     0.7832  -3.571 0.000356 ***
Petal.Length   1.3136     0.6836   1.922 0.054641 .
Petal.Width   -2.7783     1.1728  -2.369 0.017834 *
---
Signif. codes:  0 `***' 0.001 `**' 0.01 `*' 0.05 `.' 0.1 ` ' 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 190.95  on 149  degrees of freedom
Residual deviance: 145.07  on 145  degrees of freedom
AIC: 155.07

Number of Fisher Scoring iterations: 4
```

- The *Deviance Residuals* entry supplies the quartiles and mean for the deviance residuals of the fit.
- The *Coefficients* list gives an estimate of the coefficients for each predictor, and a standard error for that estimate.
- The *Null Deviance* is the G^2 statistic for the null model, i.e. the model in which the response on any observation is just the mean of the sample responses.

- The *Residual Deviance* is the G^2 value for the fitted model.
- *AIC* is the Akaike Information Criteria for the fitted model.
- The number of Fisher Scoring Iterations measures the number of iterations required for the training (or fitting) algorithm to converge.

6.8 Stepwise Model Building

The G^2 statistic provides a basis for stepwise model building. In particular, suppose we have a model with l parameters and a second model with $l+m$ parameters. Then we may test the alternative hypothesis that the extra parameters significantly improve the model. If the null hypothesis is true (that is, the extra parameters don't improve the fit), then it can be shown that

$$G_l^2 - G_{l+m}^2 \sim \chi_m^2,$$

which is also the difference of the "-2*log-likelihoods" of the two models. Thus the difference in G^2 values defines a test statistic for performing the hypothesis test. Stepwise model building can then be undertaken in a way exactly analogous to the multiple linear approach. Note that in multiple linear regression the test statistic has an F distribution, whereas here it has an approximate χ^2 distribution.

Further note that differences of Pearson's X^2 statistic do not have a χ^2 distribution and are not suitable for stepwise model building.

6.8.1 Example

The R package also provides a stepwise option for logistic regression models via the `step' function with the `test="Chisq"' option set. Returning to the previous example of the Iris data set:

```
> silr <- step(ilr,test="Chisq")

Start:  AIC= 155.07
 Species ~ Sepal.Length + Sepal.Width + Petal.Length + Petal.Width

               Df Deviance     AIC     LRT   Pr(Chi)
- Sepal.Length  1  145.213 153.213   0.143   0.70524
<none>             145.070 155.070
- Petal.Length  1  149.021 157.021   3.952   0.04682 *
- Petal.Width   1  151.479 159.479   6.409   0.01135 *
- Sepal.Width   1  160.767 168.767  15.697 7.434e-05 ***
---
Signif. codes:  0 `***' 0.001 `**' 0.01 `*' 0.05 `.' 0.1 ` ' 1

Step:  AIC= 153.21
 Species ~ Sepal.Width + Petal.Length + Petal.Width

               Df Deviance     AIC     LRT   Pr(Chi)
<none>             145.213 153.213
- Petal.Length  1  151.668 157.668   6.456   0.01106 *
- Petal.Width   1  151.836 157.836   6.624   0.01006 *
- Sepal.Width   1  172.523 178.523  27.311 1.733e-07 ***
---
Signif. codes:  0 `***' 0.001 `**' 0.01 `*' 0.05 `.' 0.1 ` ' 1
>
```

The stepwise fitting removed the redundant `Sepal.Length' coefficient, and refit the model using the remaining predictors.

6.9 Goodness of Fit Tests

There are goodness of fit tests available based on the deviance and Pearson's X^2, although they require some further conditions on the training set. For example, both depend on multiple observations at each input observation (see [W:1999]). If we have continuous input parameters this can be difficult to obtain.

This leaves us with more basic techniques for assessing the fit of the regression.

6.9.1 Percentages of Correct Classification

The simplest way to assess fit is to look at the percentage of data points correctly classified by the model. Of course, this opens up the old training/validation set question. Reporting a training set classification rate as the expected accuracy is optimistically biased and not acceptable. If there is sufficient data available, an independent validation set should be selected after training and the classification rate on that set reported. If it is not possible to obtain such a set, then cross-validation techniques, bootstrap or jackknife estimates should be used to estimate the error rate on unseen data.

The output of a logistic regression is actually an estimate of the *probability* that the observation is of the particular class or not. In order to actually classify the observation we must specify some threshold value τ above which the observation will be classified as a "1", and "0" otherwise. As we vary this threshold we will get some feel for the efficacy of the model. For example, if we find that many of the outputs occur around 0.5, then slight variations in τ will bring about significant changes in the model accuracy. Alternatively, if negatives are correctly clustered around 0 and positives around 1, the model is more robust to threshold selection. The term *sensitivity* is used to describe the probability the model gives a true positive (i.e., output is 1 when the observation really has target 1), and the *specificity* is the probability the model gives a true negative. For example, consider the two class results in the following *confusion matrix*. The entries are counts of the number of observations in the class given by the column that are classified according to the row.

	True 0	True 1
Pred 0	a	b
Pred 1	c	d

Then sensitivity = $\frac{d}{b+d}$, and specificity = $\frac{a}{a+c}$.

6.9.2 ROC Curves

ROC (Receiver Operating Characteristic) Curves are a common tool for assessing the model in this way. An ROC Curve is a plot of the true positive rate (sensitivity) of the model against the false positive rate (1-specificity) of the model, for different threshold values. Clearly if we set $\tau = 1$, we will never classify an observation as positive, so the sensitivity will be 0 and the specificity will be 1. Alternatively if we set $\tau = 0$, everything will be classified as a positive, so the sensitivity will be 1 and the specificity will be

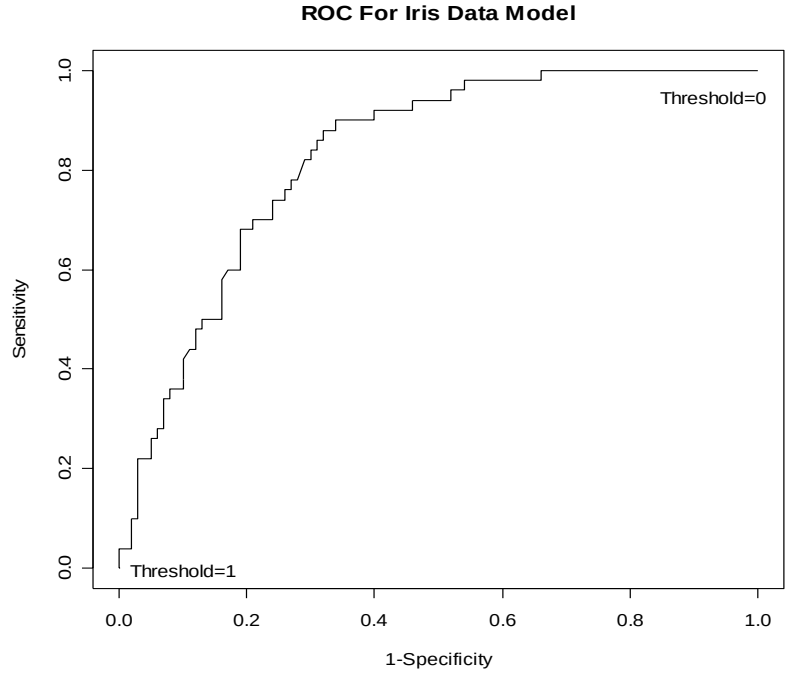
zero. As τ varies between 0 and 1, there is a tradeoff between the sensitivity and the specificity. The perfect test is one which separates the two groups completely, that is, there is some value of τ for which both sensitivity and specificity are 1. This corresponds to the ROC curve hugging the top left corner of the graph. In real life, we try to choose a threshold which gives the largest sensitivity and specificity possible. Note that for different tests, one of sensitivity and specificity might be more important than the other. For example, in detecting a disease we may be more concerned that our test is sensitive rather than specific, and choose τ accordingly.

The closer the ROC curve comes to a 45 degree line, the less value there is in our model. In the extreme, the model predicts nothing and the observations could be classified equally well by a random coin toss.

6.9.3 Example

Continuing the Iris Data example from earlier in this section, we may now construct an ROC curve using the fitted model. This curve is based on threshold increments of 0.001.

```
> # generate the ROC curve
>
> sensit <- NULL
> specif <- NULL
>
> truepos <- sum(tmpclass)
> trueneg <- length(tmpclass)-truepos
>
> coarse <- 1000
> for ( j in 1:coarse ) {
+      fv <- as.numeric(silr$fitted.values>=(j/coarse))
+      st <- fv * tmpclass
+      sp <- (fv==0) * (tmpclass==0)
+      sensit <- c(sensit,sum(st)/truepos)
+      specif <- c(specif,sum(sp)/trueneg)
+ }
>
> plot(1-specif,sensit,type="l",ylab="Sensitivity",
+  xlab="1-Specificity",main="ROC For Iris Data Model")
> text(0.1,0,"Threshold=1")
> text(0.93,0.95,"Threshold=0")
```



An ROC curve demonstrates the following:

1. It shows the tradeoff between sensitivity and specificity as the threshold τ varies.
2. The closer the curve hugs the top left corner of the ROC space the more accurate the model.
3. The closer the curve comes to the 45-degree diagonal of ROC space the less accurate the test.
4. The area under the curve is a measure of test accuracy. This quantity normally has to be estimated by some numerical method.

6.10 Extension to Multiple Responses

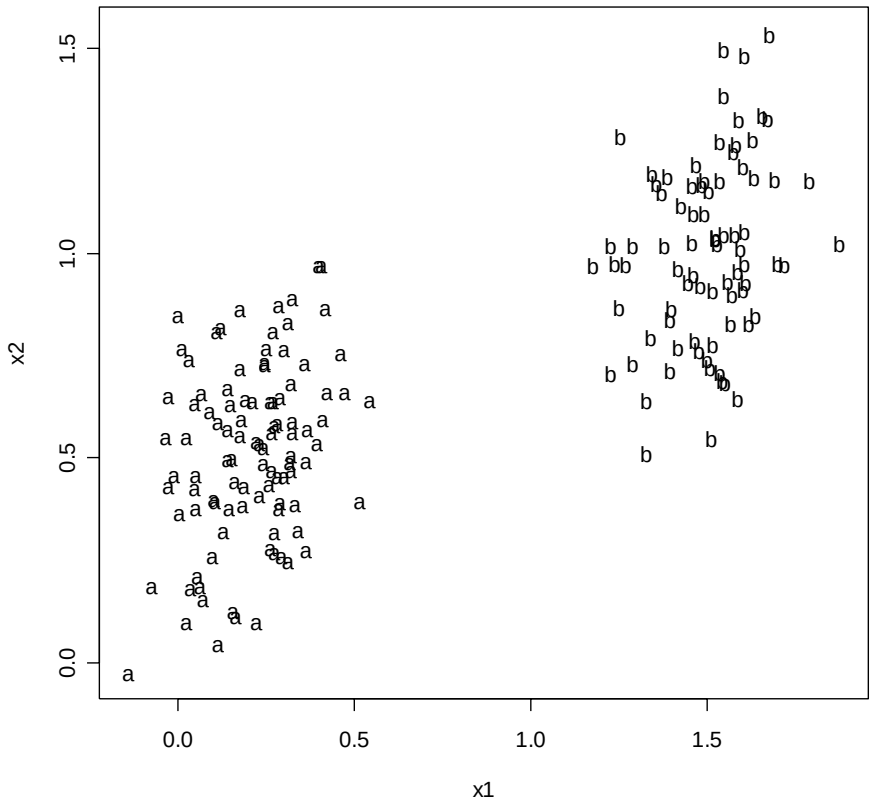
The logistic regression model can be extended to multiple response variables. This is called *polytomous* or *multinomial* logistic regression. For example, suppose each observation is to be classified as one of K classes. This is usually modeled as a logistic regression with K binary outputs, and the output pattern will be of the form $(0,0,\ldots,0,1,0,\ldots,0,0)$ where the 1 occurs at the index of the observation's class. The situation is different again if we have ordinal output. For example, the targets may be from the set {"very good", "good", "OK", "bad", "very bad"}. The order between targets induces a further structure on the logistic regression model. The theory of polytomous logistic regression is quite similar to that which we have already seen for the single binary output case. We shall not investigate this model further in this course.

7 Classification: Discriminant Analysis

7.1 Discriminating General Populations

Consider the task of classifying an observation $\mathbf{x}$ which is known to come from one of two disjoint populations (C_1 or C_2) with density functions given by f_1 and f_2 respectively. For the purposes of this section we shall assume that the misclassification cost is the same for both populations – that is, the penalty for misclassifying an observation from group 1 as group 2 is the same as the other way round. In real life this is not always the case (a missed illness is nearly always more serious than a false positive, for example), however it is a fairly simple exercise to generalize the formulae to this case.

Let π_1 and π_2 be the marginal probabilities of each population in the whole space. These are called the *apriori probabilities* of the populations and correspond to the optimal selection scheme if no other information is available. Namely, each observation is classified as group 1 with probability π_1, and group 2 with probability π_2. For example, if the two distributions (labeled `a' & `b') are quite distinct we may expect something like the following in the 2D case:



Now let R_1 and R_2 be a partition of the space of observations according to our classification rule (i.e. $\mathbf{x} \in R_i$ means we assign $\mathbf{x}$ to group i), and represent the rule itself by $\rho : (R_1 \cup R_2) \mapsto \{1,2\}$.

In the case that the probability distribution of $\mathbf{x}$ is discrete, the process of finding an optimal rule ρ amounts to maximizing, for each $\mathbf{x}$,

$$P(X \in C_i \mid X = \mathbf{x}) = \frac{P(X = \mathbf{x} \mid X \in C_i) P(X \in C_i)}{P(X = \mathbf{x})} = \frac{\pi_i f_i(\mathbf{x})}{\pi_1 f_1(\mathbf{x}) + \pi_2 f_2(\mathbf{x})},$$

with respect to C_i – i.e $\rho(\mathbf{x}) = i$ for which the above is largest. This yields the classification rule $\rho(\mathbf{x}) = 1$ if $\pi_1 f_1(\mathbf{x}) > \pi_2 f_2(\mathbf{x})$.

Of course, the distribution of observations $\mathbf{x}$ is often not discrete, in which case we can arrive at the above classification rule more rigorously. Let P_{err} be the probability of making a classification error. The optimal rule ρ is obtained by choosing the partition $\{R_1, R_2\}$ which minimizes this probability.

$$\begin{aligned} P_{err} &= \pi_1 \int_{R_2} f_1(\mathbf{x}) d\mathbf{x} + \pi_2 \int_{R_1} f_2(\mathbf{x}) d\mathbf{x} \\ &= \pi_1 - \pi_1 \int_{R_1} f_1(\mathbf{x}) d\mathbf{x} + \pi_2 \int_{R_1} f_2(\mathbf{x}) d\mathbf{x} \\ &= \pi_1 - \int_{R_1} \left(\pi_1 f_1(\mathbf{x}) - \pi_2 f_2(\mathbf{x}) \right) d\mathbf{x}. \end{aligned}$$

The set R_1^* minimizing this probability is the set of all observations for which the integral is positive, that is

$$R_1^* = \left\{ \mathbf{x} : \pi_1 f_1(\mathbf{x}) - \pi_2 f_2(\mathbf{x}) > 0 \right\}.$$

This leads to the same optimal classification rule given above. Note that for more than two groups this rule extends naturally to give the general classification rule for multiple groups. In particular, for G groups,

$$\rho(\mathbf{x}) = h \text{ if } \pi_h f_h(\mathbf{x}) = \max_{g \in \{1,\ldots,G\}} \left\{ \pi_g f_g(\mathbf{x}) \right\}.$$

This classification rule is known as the *Bayes Decision Rule*. It is the optimal classification rule for new observations because it is based on complete knowledge of the relevant probability distributions. The points at which the decision is non-unique (i.e., two or more groups achieve the same maximal probability) lie along the *Bayes Decision Boundary.* For some more simple examples we are able to compute and plot these boundaries. Of course, in practice we rarely have such knowledge, so the best we can hope for is to approximate the Bayes decision rule somehow.

7.2 Discriminating Normal Populations

Consider the more specific case in which we wish to discriminate between groups drawn from different multivariate normal populations. Recall that a p-dimensional multivariate normal random variable with mean $\mu = (\mu_1, \ldots, \mu_p)^T$ and covariance matrix Σ (that is, a $N_p(\mu, \Sigma)$ random variable) has distribution function given by

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

where $\left|\Sigma\right|$ is the determinant of Σ.

Now suppose there are G such populations with means and covariances given by $(\mu_{(g)}, \Sigma_{(g)})$ for $g = 1,...,G$. Then using the classification rule described above, we may assign a new observation $\mathbf{x} \in \Re^p$ to the group h such that

$$\pi_h\phi_h(\mathbf{x}) = \max_{g\in\{1,...,G\}}\left\{\pi_g\phi_g(\mathbf{x})\right\}.$$

Since the logarithm is a monotone function, this is equivalent to the condition

$$\ln(\pi_h) + \ln(\phi_h(\mathbf{x})) = \max_{g\in\{1,...,G\}}\left\{\ln(\pi_g) + \ln(\phi_g(\mathbf{x}))\right\}.$$

The logarithm of the normal density function may be expanded as follows:

$$\ln(\phi(\mathbf{x})) = -\frac{p}{2}\ln(2\pi) - \frac{1}{2}\ln\left|\Sigma\right| - \frac{1}{2}(\mathbf{x}-\mu)^T\Sigma^{-1}(\mathbf{x}-\mu).$$

Note that the first term on the left hand side is constant, so it need not be considered in the maximization. Thus a new observation $\mathbf{x}$ is classified as belonging to the group g that maximizes the expression:

$$L(\mathbf{x}, g) = \ln(\pi_g) - \frac{1}{2}\ln\left|\Sigma_{(g)}\right| - \frac{1}{2}(\mathbf{x}-\mu_{(g)})^T\Sigma_{(g)}^{-1}(\mathbf{x}-\mu_{(g)}).$$

7.2.1 Groups with Identical Covariance: Linear Discriminants

Let us assume a further condition on our populations, namely that the population covariances are identical. That is $\Sigma = \Sigma_{(1)} = \cdots = \Sigma_{(G)}$. By expanding $L(\mathbf{x}, g)$ we find

$$L(\mathbf{x}, g) = \ln(\pi_g) - \frac{1}{2}\ln\left|\Sigma\right| - \frac{1}{2}\mathbf{x}^T\Sigma^{-1}\mathbf{x} + \frac{1}{2}\mathbf{x}^T\Sigma^{-1}\mu_{(g)} + \frac{1}{2}\mu_{(g)}{}^T\Sigma^{-1}\mathbf{x} - \frac{1}{2}\mu_{(g)}{}^T\Sigma^{-1}\mu_{(g)}.$$

However, since the covariances are the same across groups, terms 2 and 3 on the left hand side are constant and do not contribute to the maximization problem. Recall that the covariance matrix is symmetric, thus we may simplify the expression to

$$\begin{aligned} L^*(\mathbf{x}, g) &= \ln(\pi_g) + \frac{1}{2}\mathbf{x}^T\Sigma^{-1}\mu_{(g)} + \frac{1}{2}\mu_{(g)}{}^T\Sigma^{-1}\mathbf{x} - \frac{1}{2}\mu_{(g)}{}^T\Sigma^{-1}\mu_{(g)} \\ &= \ln(\pi_g) - \frac{1}{2}\mu_{(g)}{}^T\Sigma^{-1}\mu_{(g)} + \mu_{(g)}{}^T\Sigma^{-1}\mathbf{x} \\ &= b_{(g)} + \mathbf{a}_{(g)}^T\mathbf{x} \end{aligned}$$

where $b_{(g)}$ and $\mathbf{a}_{(g)}$ are constants attached to group g. Thus, in this case, assigning an observation $\mathbf{x}$ to a group simply requires evaluating a linear expression in $\mathbf{x}$ for each group and selecting the maximum. This is called *linear discriminant analysis*, and is an extremely efficient classification technique.

7.2.2 Example

The plot at the beginning of the chapter was generated from two multivariate normal populations with means $\mu_{(a)}=(0.2,0.5)^T$ and $\mu_{(b)}=(1.5,1)^T$ respectively, and common covariance matrix

$$\Sigma=\begin{bmatrix}0.02 & 0.01\\ 0.01 & 0.04\end{bmatrix}.$$

The population proportions are $\pi_{(a)}=0.55$ and $\pi_{(b)}=0.45$. Thus the above formula may be evaluated to give

$$L^*(\mathbf{x},a)=-3.87+(4.29,11.43)\mathbf{x}$$

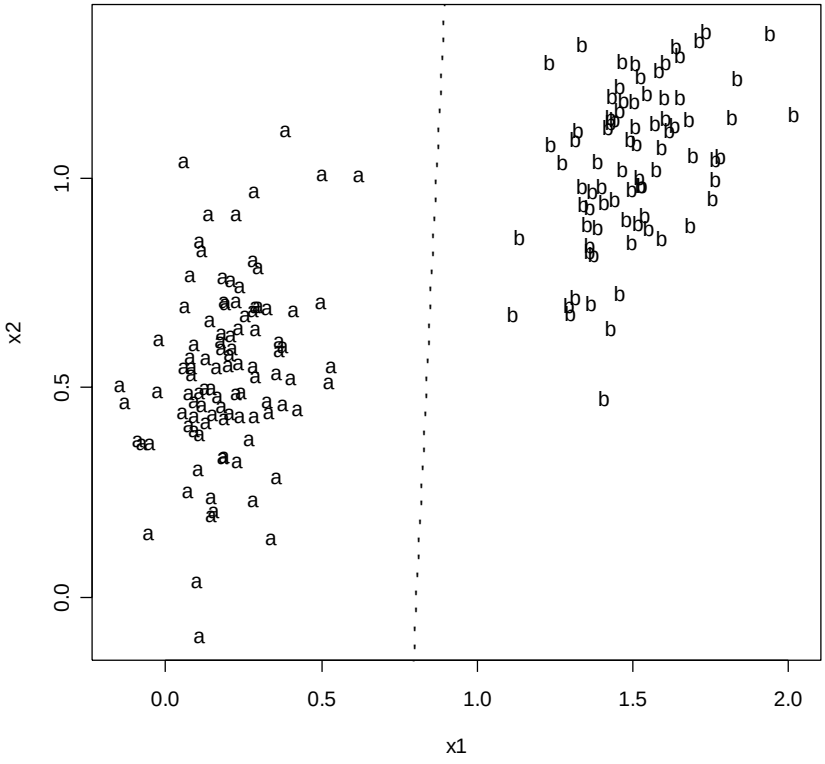
and

$$L^*(\mathbf{x},b)=-57.95+(71.43,7.14)\mathbf{x}.$$

The Bayes Decision Boundary occurs at values of $\mathbf{x}$ such that $L^*(\mathbf{x},a)-L^*(\mathbf{x},b)=0$, which is when

$$x_2=-12.62+15.67x_1.$$

Thus we may plot the populations again, and include the decision boundary.



7.2.3 Groups with Different Covariance: Quadratic Discriminants

In the more general case that we cannot assume the populations have identical covariance matrices, we are left with the original formula for $L(\mathbf{x},g)$. In practice the *generalized distance* measure is often used:

$$D(\mathbf{x}, g) = -2\ln(\pi_g) + \ln\left|\Sigma_{(g)}\right| + (\mathbf{x} - \mu_{(g)})^T \Sigma_{(g)}^{-1} (\mathbf{x} - \mu_{(g)})$$
$$= -2L(\mathbf{x}, g).$$

Clearly the group g maximizing $L(\mathbf{x}, g)$ is also the group that *minimizes* the generalized distance. The term generalized distance comes from the fact that $D(\mathbf{x}, g)$ provides some sort of measure of the distance between the observation $\mathbf{x}$ and the mean of the group g. Thus we may assign $\mathbf{x}$ to the group which it is "closest to" in the sense of D.

Recall that logistic regression returns a probability that a particular observation belongs to a given group. Using the generalized distance we may also obtain the same information from discriminant analysis if the populations may be assumed to be normally distributed. For some group g, the probability that an observation $\mathbf{x}$ belongs to group g is then

$$p(g \mid \mathbf{x}) = \frac{\mathrm{e}^{-\frac{1}{2}D(\mathbf{x},g)}}{\sum_{h=1}^{G} \mathrm{e}^{-\frac{1}{2}D(\mathbf{x},h)}}.$$

7.2.4 Populations with Unknown Parameters

In practice we are rarely presented with distributions that have known parameters. In this situation the mean and covariance of group g is estimated by the sample mean $\bar{\mathbf{x}}_{(g)}$ and sample covariance $S_{(g)}$ of the observations in that group. If we have reason to believe the population covariance matrices are the same, then an estimate of the common covariance matrix is given by a pooling the individual group estimates.

If we are willing to assume that the covariances are identical, then we may use linear discriminant analysis, otherwise we must use quadratic discriminant analysis.

Note that these methods are based on the assumption that the populations are distributed according to a multivariate normal distribution. In practice this is often a good assumption, and even if it is manifestly false, the methods may still provide good classification if the groups are sufficiently separated.

7.3 R Example and Results Reporting

Classification accuracy is normally reported via a *confusion matrix*. The columns and rows represent true groups and predicted groups respectively, with the ij^{th} entry, n_{ij} giving the number of observations predicted as group i when they in fact belong to group j. Thus the diagonal gives the number of correct predictions in each group, and the off-diagonal entries detail the misclassifications.

	Group A	Group B	Group C
Predicted A	n_{AA}	n_{AB}	n_{AC}
Predicted B	n_{BA}	n_{BB}	n_{BC}
Predicted C	n_{CA}	n_{CB}	n_{CC}

7.3.1 Example

We shall demonstrate the R discriminant analysis procedures using the two artificially generated groups (`a' and `b') described above. In addition to these we shall include a third population `c' with $\mu_{(c)}=(0.6,1)^T$ and

$$\Sigma_{(c)} = \begin{bmatrix} 0.07 & 0 \\ 0 & 0.002 \end{bmatrix}.$$

The new population proportions for a, b and c are $\pi_{(a)} = 0.46$, $\pi_{(b)} = 0.36$ and $\pi_{(c)} = 0.18$.

```
> library(MASS) # for multivariate normal generation
>
> ## two multivariate normal distributions with same covariance
> S <- matrix(c(0.02,0.01,0.01,0.04),nrow=2)
> mu1 <- c(0.2,0.5)
> mu2 <- c(1.5,1)
>
> ## and a third with different covariance matrix
> S3 <- matrix(c(0.07,0,0,0.002),nrow=2)
> mu3 <- c(0.6,1)
>
> # generate training observations
> ob1 <- mvrnorm(100,mu1,S)
> ob2 <- mvrnorm(80,mu2,S)
> ob3 <- mvrnorm(40,mu3,S3)
>
> # generate test observations
> tob1 <- mvrnorm(200,mu1,S)
> tob2 <- mvrnorm(160,mu2,S)
> tob3 <- mvrnorm(80,mu3,S3)
>
> # plot the data points
> plot(ob1,xlim=range(range(ob1[,1]),range(ob2[,1])), xlab="x1",
+ ylab="x2", pch="a", ylim=range(range(ob1[,2]),range(ob2[,2])))
> points(ob2,pch="b")
> points(ob3,pch="c")
```

Linear and quadratic discriminant models are fitted using the `lda' and `qda' functions respectively.

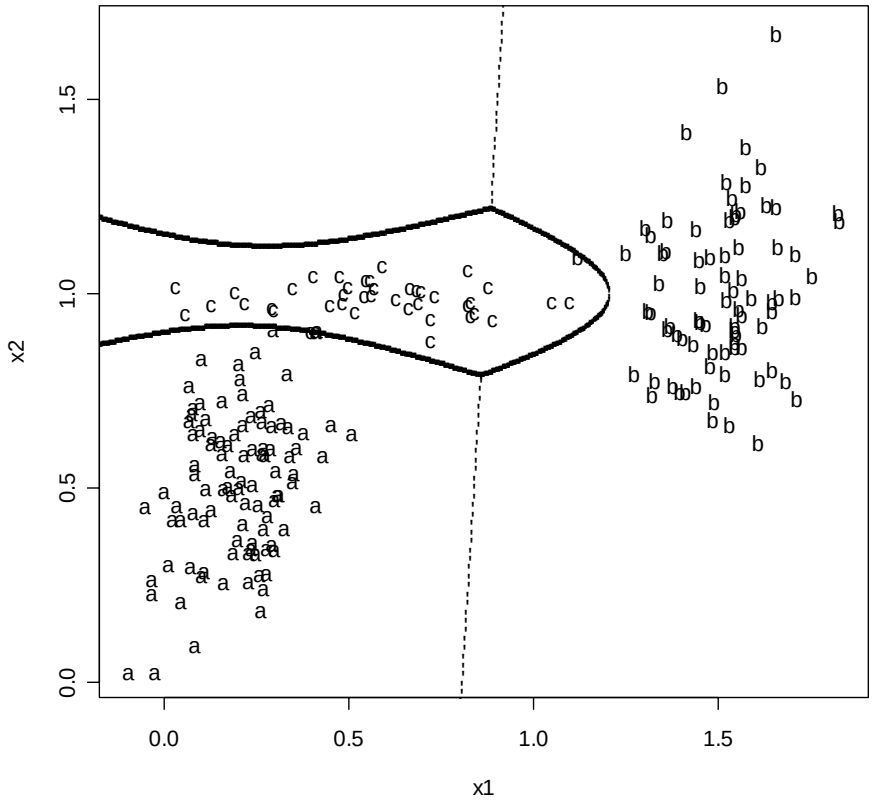
```
> # group the data for discriminant analysis
> grps <- as.factor(c(rep("a",100),rep("b",80),rep("c",40)))
> obs <- rbind(ob1,ob2,ob3)
>
> # test observations
> tgrps <- as.factor(c(rep("a",200),rep("b",160),rep("c",80)))
> tobs <- rbind(tob1,tob2,tob3)
>
> # do lda
> ld <- lda(obs,grps)
>
> # do qda
> qd <- qda(obs,grps)
>
```

Now we can predict the classes of the test observations using these models. The `predict.lda' and `predict.qda' functions provide a prediction of the class ($class attribute) and posterior probabilities ($posterior attribute) of each observation.

```
> # predict on the test set using both models
> qdpred <- predict(qd,tobs)
> ldpred <- predict(ld,tobs)
>
```

To assess these results we use a confusion matrix. This is generated using the `confusion' function (package="mda").

```
> # produce a confusion matrix for each prediction
> library(mda)      # need this for confusion matrix
>
> confusion(ldpred$class,tgrps)  # lda confusion matrix
       true
object   a   b  c
     a 194   0  6
     b   0 160  5
     c   6   0 69
attr(,"error")
[1] 0.03863636
>
> confusion(qdpred$class,tgrps)  # qda confusion matrix
       true
object   a   b  c
     a 196   0  2
     b   0 160  3
     c   4   0 75
attr(,"error")
[1] 0.02045455
```



The prediction from the quadratic model is somewhat better than that from the linear model since the assumption of equal covariance matrices is not true with these three populations. The Bayes decision boundaries are included on the plot for reference.

7.4 Discriminating General Populations Revisited

As noted above, the linear and quadratic discriminant methods are based on the assumption that the data come from multivariate normal

distributions. In many cases this assumption is not valid, although the methods described can often give good separation anyway. A more general approach was proposed by Fisher in 1936. For an observation $\mathbf{x}$ it seeks a linear combination $\mathbf{x}^T\mathbf{a}$ that maximizes the ratio of the *between-group variance* to the *within-group variance* in the population. This concept is intuitively appealing no matter what the distributions of the groups happen to be.

The within-group covariance matrix W is usually taken to be the pooled estimator of the population standard deviation based on the assumption that the groups have identical covariance. This is the same as the estimator used for covariance in the case of distinguishing normal populations with identical covariance.

The between-group covariance matrix B is computed by letting all the observations in a group be equal to the mean of that group, and then computing the global covariance matrix.

Once these two matrices have been computed, we attempt to maximize the expression:

$$\frac{\mathbf{a}^T B\mathbf{a}}{\mathbf{a}^T W\mathbf{a}}$$

which amounts to seeking a linear transformation of the space that separates the groups as much as possible.

The procedure for deriving the linear combination $\mathbf{a}$ is beyond the scope of this course, however the basic concept can be easily grasped. Most methods construct a sequence $(\mathbf{a}_1,\ldots,\mathbf{a}_r)$ of vectors, each one maximizing the ratio between the variances subject to being linearly independent of the previous vectors. This amounts to a linear transformation A of the observation space to an r dimensional space in which the groups are maximally separated according to the variance ratio. An unseen observation $\mathbf{x}$ may then be classified by means of its representation in the transformed space, $\mathbf{x}^T A$. There are a number of possibilities for making this classification, among the simplest is simply to classify the observation according to the group whose mean is closest to the transformed observation.

8 Classification: Decision Trees

[BFOS:1984], [TA:1997]

8.1 Introduction

Although there are more general tree structures, we will concentrate on **binary decision trees** in the following, even if the qualification "binary" is omitted. A decision tree structures the classification task as a series of binary questions put to the input data. Each question divides the space of possible observations into two disjoint subsets, each of which are faced with a further (different) question, and so on until the leaves of the tree are reached. A decision tree is conventionally represented as growing down from the top of the page. The *root* is the top node, and decisions are made at each *node* until a terminal node or *leaf* is reached. A *subtree* of a decision tree *T* is a tree with root at a node of *T*, and it is a *rooted subtree* if its root is the root of *T.* A generic term for many algorithms that fit decision trees is *recursive partitioning*, which describes the process of beginning with a single root node and recursively splitting the node and its children to build the tree.

Decision trees have the advantage of being very easy to interpret and as such are favored for automatic diagnostic systems etc. An expert can verify that the decisions proposed at each node are indeed sensible, and thus promote confidence in the system.

In order to construct a decision tree, we need to address several issues:

- Split assessment: how do we know if a particular split improves the tree or not?
- Splitting rule: how are the decisions to be chosen at each node?
- Stop-splitting rule: at what point do we stop growing the tree?
- Pruning rule: once the tree growth is completed, do we apply any pruning to the model, and if so, based on what criteria?

The answers to these questions constitute the major differences between tree classifiers, or recursive partitioning classifiers.

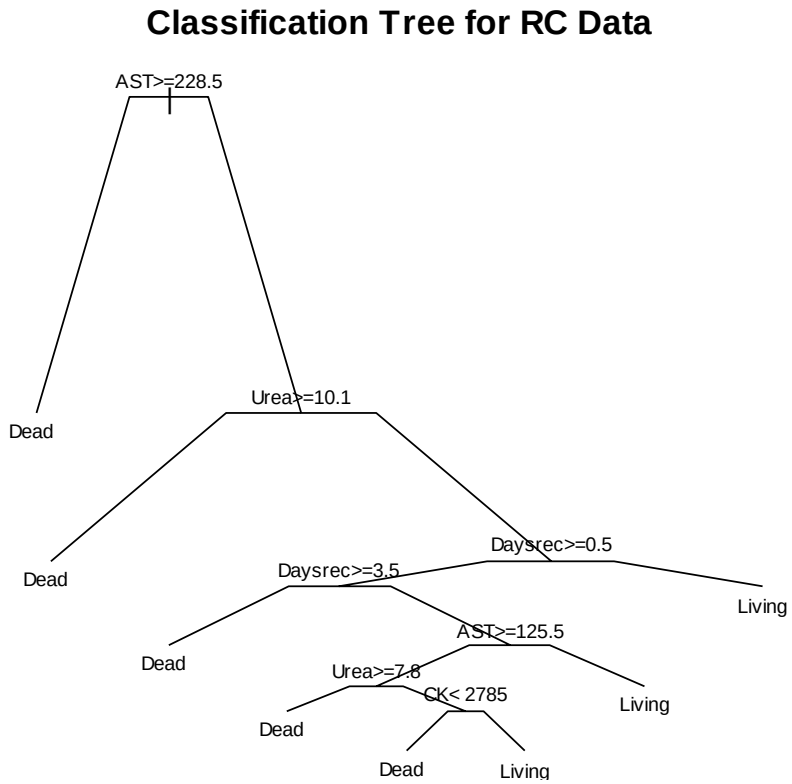
CART is a very popular method for building decision trees. Indeed, the book [BFOS:1984] is largely responsible for the explosion of the field. We shall use the CART methodology throughout this chapter as both an example of decision trees in general, and a useful technique on its own. The current commercial CART package contains many options above the basic system which we do not intend to detail here. Finally note that the name "CART" is actually a trademark and may not be used for implementations of the [BFOS:1984] algorithms. In this course we use the rpart package in the R environment, which implements most of the basic CART functions.

8.1.1 Example

We use a decision tree algorithm (rpart, from the R environment) to create a decision tree on the recumbent cows data set. Recall that an outcome of 1 means the cow is expected to survive.

```
> library(rpart)
>
> rcdata <- read.table("C:\\RData\\Cows\\Recumbent Cows.txt",
+ header=TRUE)
>
> # set up the outcome levels so they mean something
```

```
> rcdata[,"Outcome"] <- as.factor(rcdata[,"Outcome"])
> levels(rcdata[,"Outcome"])<-c("Dead","Living") # 0=Dead, 1=Living
>
> fmla <- as.formula("Outcome~AST+Urea+Uketone+PCV+Inflamat+Myopathy\
 +Calving+Daysrec+CK")
>
> # do the default rpart tree
> rctree <- rpart(fmla,rcdata,method="class")
>
> # now plot it – compress, branch, cex are just formatting options
> plot(rctree,compress=T,branch=.3)
> text(rctree,cex=.7)
```



8.2 Impurity and Node Splitting

Suppose we have a tree consisting of a single root node, t_0, and we are considering a population split at this node that will generate two child nodes, t_1 and t_2. To assess the value of this split, we introduce the concept of *node impurity*. In general terms, a node is considered to be pure if only observations from a single class are found at that node. Conversely a node is impure if a mixture of observations from different classes is found at that node, and the node is maximally impure if this mixture is uniformly distributed over all possible classes. A node split that decreases the overall impurity of the tree, or at least does not increase the overall impurity is considered as a candidate for the next split.

8.2.1 Node & Tree Impurity

To put this more formally, suppose the population we are considering is divided into G groups. Then any node t in the tree will have an associated probability distribution

$$\mathbf{p}(t) = (p_1(t), p_2(t), ..., p_G(t))$$

which gives the probability of an observation found at node *t* being from each of the groups. An *impurity function* $i(\mathbf{p}(t))$ is a real-valued function satisfying the following conditions:

(a) i achieves its global maximum only at $\mathbf{p}(t) = \left(\frac{1}{G}, \frac{1}{G}, \cdots, \frac{1}{G}\right)$.

(b) i achieves its global minimum only at the points $(1,0,...,0)$, $(0,1,0,...,0)$, . . . , $(0,...0,1)$.

(c) $i(\mathbf{p}(t))$ is symmetric in $p_1(t),...,p_G(t)$.

(d) For any pair of nodes t_j, t_k, and numbers *r*, *s* such that *r+s=1*, $i(r\mathbf{p}(t_j) + s\mathbf{p}(t_k)) \geq r(i(\mathbf{p}(t_j))) + s(i(\mathbf{p}(t_k)))$.

The last condition is proposed in [BFOS:1984] as a desirable feature in a "good" impurity function. It basically ensures that a mixture is at least as impure as the average impurity of its components.

There are many possible candidates for the impurity function - we shall focus on two of the most common. The first is called the *entropy*, and is indeed the standard definition for the entropy of a population drawn from a probability distribution $\mathbf{p}$.

$$i(\mathbf{p}(t_i)) = -\sum_{g=1}^{G} p_g(t_i) \log p_g(t_i) .$$

Clearly this satisfies our requirements of an impurity function, and is a well accepted measure of the amount of randomness in a set.

The second candidate is the *Gini index*. This function is simply 1 minus the probability that two independent observations from the node of interest are of the same class:

$$i(\mathbf{p}(t_i)) = 1 - \sum_{g=1}^{G} p_g^2(t_i) .$$

Let $\pi(t)$ be the proportion of the population that is found at node t. The sum of these proportions taken over all the terminal nodes (that is, nodes with no children) will always be 1.

Now let us return to the example at the beginning of this section. Namely a split is proposed which divides the root node t_0 into two child nodes t_1 and t_2. Suppose further that the split siphons a proportion $\pi(t_1)$ of the population to node t_1, and the remaining proportion $\pi(t_2) = 1 - \pi(t_1)$ to node t_2. Then the total change in impurity as a result of this split is given by

$$\Delta i = i\left(\mathbf{p}(t_0)\right) - \left(\pi(t_1) i\left(\mathbf{p}(t_1)\right) + \pi(t_2) i\left(\mathbf{p}(t_2)\right)\right) .$$

The splitting strategy is simply to choose at each split the candidate that supplies the maximum decrease in impurity.

The impurity of a tree T is defined to be

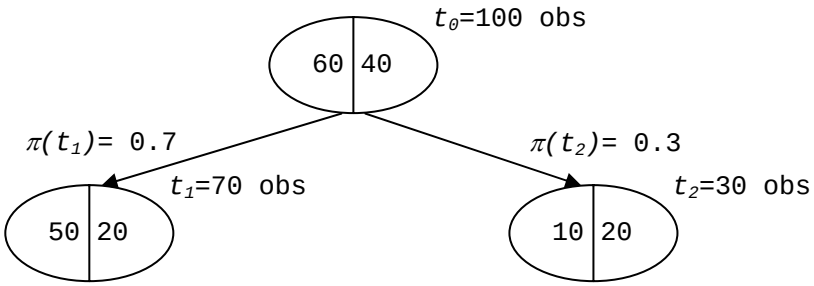
$$I(T) = \sum_{t \in \tilde{T}} \pi(t) i(\mathbf{p}(t))$$

where $\tilde{T}$ is the set of terminal nodes or leaves in T.

In the above we have assumed that we know the probability distribution at each node. Of course, this is rarely true in practice, although in some cases we may have a known prior distribution at the root node. Normally these distributions are estimated from a set of training examples. At each split the set of examples at the parent node is split into two, and estimates of the distributions for the child nodes are generated from the two subsets.

8.2.2 Example

Consider the simple numerical example in the diagram below. Each oval represents a node, the number on the left side of the vertical bar gives the number of examples from group 1, and on the right from group 2. A split is proposed to generate the two child nodes t_1 and t_2 from the root node t_0.



Using the Gini index we have:

$$i(\mathbf{p}(t_0)) = 1-(60/100)^2-(40/100)^2 = 0.48$$

$$i(\mathbf{p}(t_1)) = 1-(50/70)^2-(20/70)^2 = 0.408$$

$$i(\mathbf{p}(t_2)) = 1-(10/30)^2-(20/30)^2 = 0.444$$

Thus the total decrease in impurity is

$$\Delta i = 0.48-(0.7)(0.408)-(0.3)(0.444) = 0.06$$

8.2.3 Splitting Rules

At each node, a splitting rule is determined based on the value of the predictors. In the simplest case, a single attribute (predictor) is chosen, and the node is split based on that attribute. If the attribute is binary, clearly the split will be a simple two-way split. If the attribute is L-valued categorical, we may either split the node L ways, or consider a binary split dividing the categories into two groups (note that there are $2^{L-1}-1$ non-empty pairs of groups, so this will generate many potential attribute splits for large L; further, at each such split we will need to keep track of the new attribute sets for future node splits on this attribute). If the attribute is continuous or ordinal, the split is two-way binary around some breakpoint $x \le x_c$. There have been some more complex splitting rules considered, see [R:1996] for more on this.

The basic CART model uses binary splits on single variables. To find the best candidate it uses a *greedy local search* which is an exhaustive search of the possibilities. For each ordered variable x_m the splitting rule is of the form $I[x_m \le c]$ for some $-\infty \le c \le \infty$, and for each categorical variable x_m with values $\{k_1,\ldots,k_L\}$ the splitting

rule is of the form $I[x_m \in S]$ for all subsets S of $\{k_1,\ldots,k_L\}$. Note that for a finite training set, there are only a finite number of splitting rules available. For each ordered variable, we need only select splitting constants c that separate between training examples, so for each ordered variable there are at most as many rules as there are training examples. For a categorical variable note that $[x_m \in S]$ and $[x_m \notin S]$ generate the same split, therefore there are $2^{L-1}-1$ possible splits for L categories.

CART searches through all the split possibilities to find the best split for each variable, and then chooses the best variable from those. CART splitting is based on the Gini measure of impurity, although the authors claim the algorithm is not sensitive to the impurity measure used, and entropy can be used if requested.

8.2.4 Stop-splitting Rules

Stop-splitting rules generally take the form of some function of the impurity on each of the leaf nodes. Suppose the tree T has leaf nodes $t_1,\ldots,t_L$, and $i_{t_l}(p)$ is the impurity of leaf node t_l, then we seek some function $\varsigma(i_{t_1}(p),\ldots,i_{t_L}(p)) \in \{0,1\}$ that is 0 when the splitting is to continue, and 1 when the splitting is to stop. We may also have rules that apply to individual leaves.

Some examples of stop-splitting rules:

- Stop splitting when the overall impurity of the tree is below some threshold.
- Stop splitting on a particular node when the overall impurity of the node is below some threshold.
- Stop splitting on the tree when the depth of the tree is greater than some parameter.
- Stop splitting on a particular node when the number of training set examples reaching that node gets below some parameter.
- Stop splitting when the decrease in impurity on a split is below some threshold.

Clearly more complicated splitting rules are possible, either as combinations of the above or using different criterion altogether.

In principle CART does not use a stop-splitting rule at all. The growing phase of the algorithm aims to grow a tree with maximal purity on the training set. Thus splitting on a node stops only if that node has impurity 0 or if no improvement in impurity can be achieved, which may occur when the node is reached by only a single training set observation. In practice CART is limited by complexity issues and may stop splitting when the tree gets beyond a certain depth or, for a particular node, less than some threshold numbers of observations reach that node. All of these parameters can be adjusted by the user.

8.3 Tree Pruning

Once a complete decision tree has been grown, we wish to prune the tree back to a minimal "suitable" tree. Once again, there are a number of heuristic methods proposed for pruning, the best known of

which was proposed by [BFOS:1984]. We shall present the basic concept here to illustrate the ideas behind pruning, and develop the actual method in the next section on CART models. Note that some approaches to building decision tree classifiers rely on stopping rules at the splitting stage and don't employ pruning at all.

8.3.1 Loss (Risk) Function

Before we consider any pruning methods, we need to address the question of how to determine if a particular pruning operation has been "successful" or not. That is, we need some cost function for a tree, so we can tell when this function has been reduced, which indicates an improvement in the fit. The obvious cost function is the probability that the tree will misclassify an observation. More formally, if d_T is the classification function from tree T, and (X,Y) is a random observation with features X and classification Y, then the *loss (or risk) function* $R(T)$ of the tree is

$$R(T) = P(d_T(X) \neq Y) .$$

Of course, we don't normally know the distribution of (X,Y), so this probability has to be estimated from a data sample. In the case that we have an abundance of data, it may be estimated by means of a test set, independent of the data used to fit the tree:

$$R(T) \approx R^{ts}(T) = \frac{1}{N_{ts}} \sum_{i=1}^{N_{ts}} I(d_T(\mathbf{x}_i) \neq y_i)$$

where $(\mathbf{x}_1, y_1),...,(\mathbf{x}_{N_{ts}}, y_{N_{ts}})$ is the set of test data. Alternatively we may estimate $R(T)$ by cross-validation.

The loss function defined above is equivalent to the sum of the probabilities of misclassification on the leaves of T. That is,

$$R(T) = \sum_{t \in \tilde{T}} R(t) = \sum_{t \in \tilde{T}} \pi(t) r(t)$$

where $\tilde{T}$ is the set of terminal nodes in T, *r(t)* is the node error rate, and as before $\pi(t)$ is the probability of a case belonging to leaf node *t*.

There are other loss function candidates besides the misclassification probability, although $R(T)$ is generally taken to be a sum of contributions from each leaf. For example, we may take the entropy or deviance of the partition at each leaf. The pruning rules below do not depend on the form of the loss function chosen, although $R(T)$ is evaluated on the *learning set* unless otherwise specified (for example by the `ts' superscript).

9.3.2 Pruning Rules

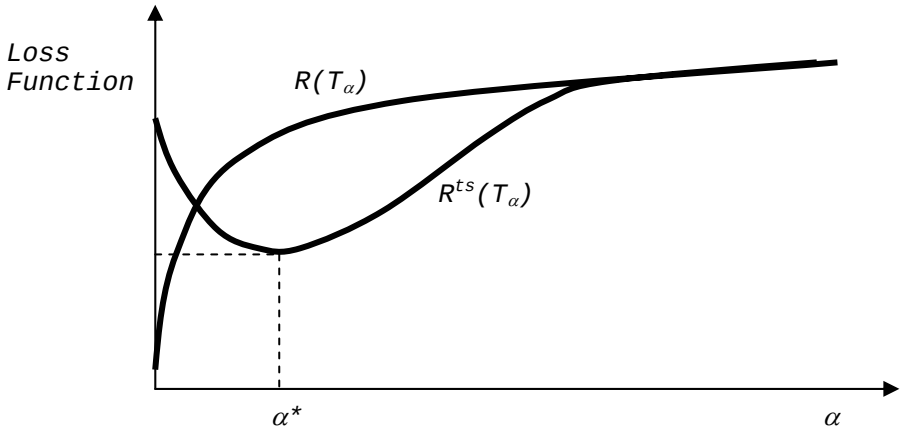
Now suppose we have fit some (possibly quite large) tree T to a data set. Suppose further we have chosen a loss function $R(T)$ for that tree which is evaluated on the training set. We now seek to "prune" T back to a smaller tree that is optimal in some sense. In particular, depending on the stop-splitting rule used when building T, the tree may be heavily overfitted to the training set. By pruning the tree back we hope to obtain a tree that generalizes better.

Let $|\tilde{T}|$ be the number of leaves in T, and let α be a pruning parameter dictating the severity of the pruning. Then for a given α we choose a rooted subtree T_α of the full tree T which minimizes:

$$R_\alpha(T) = R(T) + \alpha|\tilde{T}|.$$

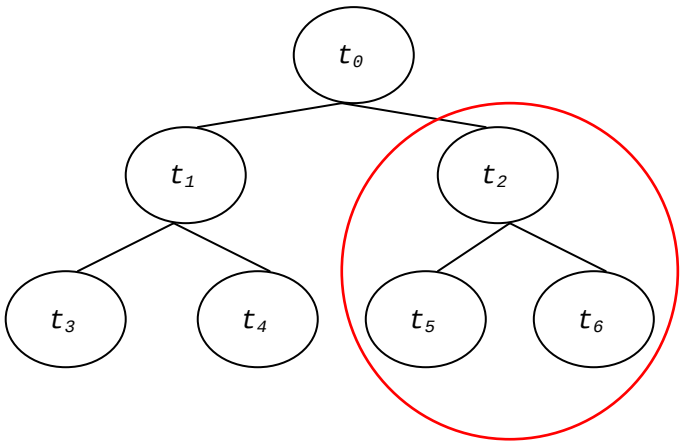
Thus we impose a penalty α on the tree for each leaf node. A larger α will impose a heavier penalty and encourage smaller trees. On the other hand, a smaller value for α will grow larger trees. We shall look at methods for building an optimal tree given a specific α later, but suppose for the moment we are able to find an optimal T_α for each α.

Since the $R(T)$ is evaluated on the learning set it will increase as α increases, because the unpenalized fit will always give better predictions on the learning set. In the case of a heavily overfitted tree, $R(T)$ will be close to 0 for $\alpha = 0$. On the other hand, the test set error should begin to decrease as α increases, until some point where the pruning becomes so heavy that the prediction deteriorates.



The diagram gives a stylized rendition of how we might expect the loss functions to differ as α increases. The optimal value of α is marked as α^*. If training data is scarce, then once we have found α^*, we may choose to combine the training and test sets, refit the tree, and prune using this pruning parameter.

Consider the following tree T:



This is a tree with 4 leaf nodes, t_3, t_4, t_5 and t_6. The red circle indicates the *subtree* rooted at node t_2, which we shall call T_{t_2}. For a given complexity parameter α, the *subtree error* of T_{t_2} is

$$R_\alpha(T_{t_2}) = R(T_{t_2}) + \alpha\left|\tilde{T}_{t_2}\right| = \pi(t_5)r(t_5) + \pi(t_6)r(t_6) + \alpha \bullet 2 .$$

We also define the *node error* of t_2 by

$$R_\alpha(t_2) = R(t_2) + \alpha = \pi(t_2)r(t_2) + \alpha$$

where $R(t_2)$ indicate the loss function evaluated at t_2 if it is treated as a leaf node (i.e. no further splits are allowed). If we are pruning the tree to complexity parameter α, then we will prune subtree T_{t_2} (that is, remove all the child splits and their children, and so on, and treat node t_2 as a leaf node) only in the case

$$R_\alpha(t_2) \le R_\alpha(T_{t_2}) .$$

A simple reordering of the expressions above shows that this condition is satisfied if an only if

$$g(t_2, T) \equiv \frac{R(t_2) - R(T_{t_2})}{\left|\tilde{T}_{t_2}\right| - 1} \le \alpha .$$

Note that the expression $g(t_2, T)$ does not depend on α, and may be computed for each internal node in T in a straightforward manner. Thus, for *any* α, we may prune T simply by cutting off all the subtrees T_{t_i} with $g(t_i, T) \le \alpha$.

In the CART pruning algorithm these observations are used to generate a sequence of trees, and corresponding α values, ranging from the simplest to the most complex. More formally, for a given decision tree T [BFOS:1984] showed that there is a nested family of rooted subtrees T_k of $T_0 = T$ such that each is optimal for a range of α, and so there are values $0 = \alpha_0 < \alpha_1 < \cdots < \alpha_k < \infty$ such that T_i is an optimal tree for $\alpha \in [\alpha_i, \alpha_{i+1})$. The algorithm is given here, the proofs can be found in [BFOS:1984] or [R:1996].

1. Set $k = 0$ and write out $T_0 = T$.
2. Set $\alpha = \infty$.
3. Visit the non-terminal nodes t in bottom-up order and calculate $R(T_t)$ and $\left|\tilde{T}_t\right|$ by summing over the descendants (and including any contribution at t). Set
   $g(t, T)$ and $\alpha = \min(\alpha, g(t, T))$.
4. Visit the nodes in top-down order and prune whenever $g(t, T) = \alpha$.
5. Set $k = k + 1$ and write out $\alpha_k = \alpha$ and $T_k = T$.
6. If T is non-trivial, go to 3.

Note that in Step 4. one may prune whole branches, sub-branches of which would not have been pruned by comparing their top node $g(t, T)$'s with α. Each iteration will prune off some (possibly large) number of nodes, thus the tree can be dramatically reduced in size in a single iteration. After each pruning iteration, when returning to

Step 3. the $g(t,T)$ values have to be re-computed. Note that the tree T_i, which is optimal for complexity parameters in the range $[\alpha_i, \alpha_{i+1})$, is normally associated with the complexity parameter $\alpha_i^* = \sqrt{\alpha_i \alpha_{i+1}}$, the geometric mean of the endpoints.

Having constructed the sequence of nested trees, we now need to choose a particular tree from within this sequence. There are two general approaches employed here, depending on the amount of data available.

Option 1: Test Set Validation
In a data mining context where data is usually abundant, the best approach is to select an independent test set and evaluate the error measure $R^{ts}(T_k)$ for each tree in the sequence. The tree giving the minimal test set error rate will be our candidate. Once again, when we have found the optimal tree and its corresponding α^*, we may wish to pool the training and test set, refit the tree, and prune it back using α^* so as to use the largest training set possible.

Option 2: Cross-Validation
If there is not sufficient data for an independent validation set, [BFOS:1984] propose using a cross-validation technique to select the optimal tree. In this approach we take the whole data set (call it D) and fit a tree with complexity parameter α small or zero (in the case of very complex data, it may be worth stopping the training before the tree is maximal to save computations). From this fit we can generate a sequence of candidate trees and corresponding complexity parameters $\alpha_1^*, \ldots, \alpha_k^*$ using the algorithm above. Now the following algorithm is executed:

1. Split the dataset D into V equal parts $D_1, D_2, \ldots, D_V$ ($V = 10$ is a popular value).
2. Carry out, for each α_i^*, steps 3 and 4 below.
3. For each $v \in \{1, \ldots, V\}$, build a new tree T_v using learning data $D_1 \cup \ldots \cup D_{v-1} \cup D_{v+1} \cup \ldots \cup D_V$ and prune the tree using complexity parameter α_i^*. Then evaluate the pruned $T_v^{(\alpha_i^*)}$ tree using D_v as a test set to get an estimate of the test set error $R^{ts}(T_v^{(\alpha_i^*)})$.
4. Average all the V test set error estimates to estimate the overall test set error of a tree built with complexity parameter α_i^*. Call this error $R^{CV}(\alpha_i^*)$, the *cross-validation error*. Using the V test set error values we may also obtain an estimate of the standard error (the standard error of the mean) of our estimate of the cross-validation error.

Thus, at the end, we have a list of estimates of the cross-validation error, and its standard error, corresponding to our list of complexity parameters. Finally we select α^*, the optimal complexity parameter, based on these cross-validation errors. In practice a rule of thumb is often used, based on the observation that the minimal error is often surrounded by other errors very close to it. Therefore we treat all cross-validation errors within 1 standard error of the minimum as equal, for the purposes of selecting α^*. The complexity

parameter is then chosen to be the parameter corresponding to the smallest tree with cross-validation error within 1 standard error of the minimum. This rule is called the *1-SE rule*.

In the commercial CART package both of these options are implemented, but in the R package, rpart, only cross-validation is available, although test set validation can be implemented with minimal programming effort.

8.4 Interpreting CART Output (rpart)

In this course we shall use the `rpart' package in R. This package follows the theory of [BFOS:1984] quite closely, but is not a commercial package. The commercially available CART package (http://www.salford-systems.com/index.html) has many more options, and is much better suited to large data sets.

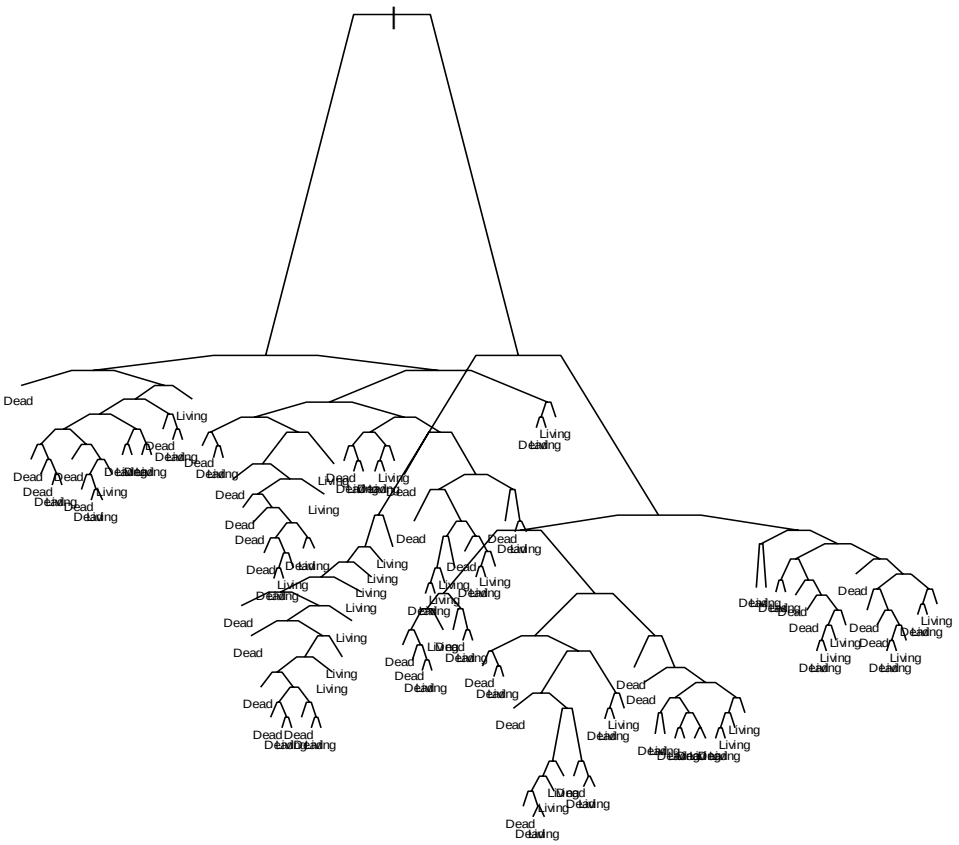
Recall the "Recumbent Cows" tree fitted at the beginning of the section. The rpart function has a number of default arguments set which attempt to restrict the growth of the tree in some sensible way to reduce the burden on the pruning step. The `rpart.control' function is used to change these parameters. For the following example, the default arguments we are interested in are:

minsplit: the minimum number of observations required at a node before a split is considered on that node (default is 20).
minbucket: the minimal number of observations allowed at a node (default is (minsplit/3)).
cp: a complexity parameter. If a split doesn't improve the overall loss by at least cp, it is not undertaken (default is 0.01).

Therefore we may construct an exhaustive tree (which perfectly classifies the training set) in the following way:

```
> rctrl <- rpart.control(minbucket=1,minsplit=2,cp=0)
> rctree2 <- rpart(fmla,rcdata,method="class",control=rctrl)
>plot(rctree2,compress=T,branch=.3,main="Unpruned Tree for RC Data")
> text(rctree2,split=FALSE,cex=.5)
```

Unpruned Tree for RC Data



We have left off the splitting criteria from the plot to improve the readability.

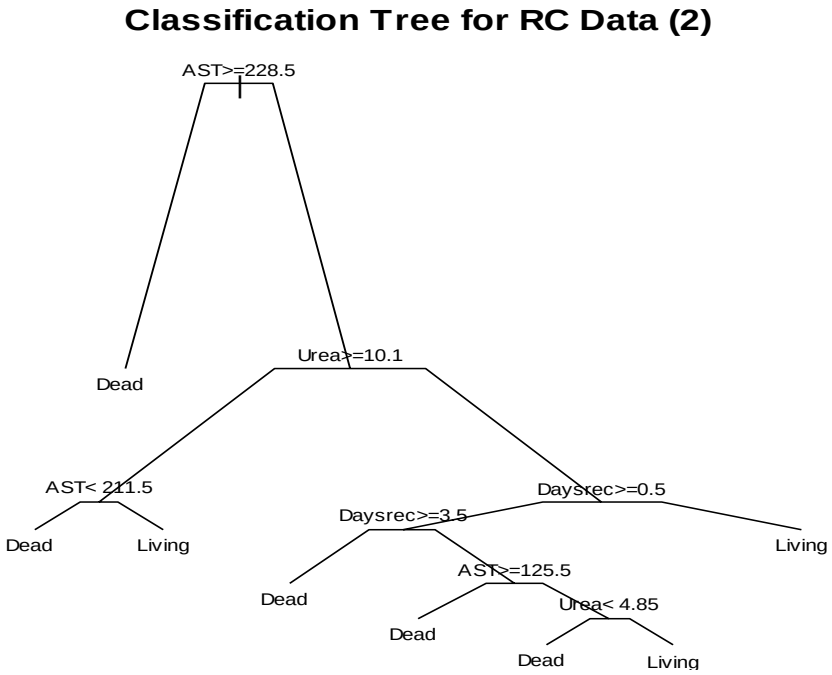
As part of the fitting process, rpart constructs the sequence of nested trees referred to in the pruning section above. Each tree in the sequence is assessed using cross-validation on the training set to get an estimate of the error, and an estimate of the error standard deviation. The relative training set error (`rel error'), the cross validation estimate of risk (`xerror'), the standard deviation of the risk (`xstd'), and the related cp value are reported by the printcp function:

```
> printcp(rctree2)

Classification tree:
rpart(formula = fmla, data = rcdata, method = "class", control =
rctrl)

Variables actually used in tree construction:
[1] AST        Calving   CK         Daysrec   Inflamat Myopathy PCV
Uketone
[9] Urea

Root node error: 166/435 = 0.38161

n= 435

          CP nsplit rel error  xerror     xstd
1  0.1927711      0 1.0000000 1.00000 0.061035
2  0.0903614      1 0.8072289 0.88554 0.059430
3  0.0180723      2 0.7168675 0.86747 0.059126
```

```
4  0.0150602      7 0.6204819 0.78313 0.057513
5  0.0120482     11 0.5602410 0.78313 0.057513
6  0.0090361     13 0.5361446 0.81325 0.058127
7  0.0084337     15 0.5180723 0.86145 0.059021
8  0.0080321     25 0.4156627 0.87952 0.059330
9  0.0060241     28 0.3915663 0.88554 0.059430
10 0.0045181     56 0.2228916 0.89759 0.059624
11 0.0040161     61 0.1987952 0.92169 0.059995
12 0.0030120     64 0.1867470 0.93976 0.060258
13 0.0020080    106 0.0421687 1.04217 0.061492
14 0.0000000    123 0.0060241 1.06627 0.061723
```

The `nsplit' quantity in the table reports the number of splits in the relevant tree. Any of the trees can be extracted using the `prune' function with the appropriate cp value. Of course, we wish to return the optimal tree, which should be the tree with the minimal risk (xerror). In practice a rule of thumb (the 1-SE rule) is usually employed to compensate for the fact that many of the trees are not significantly different in their performance. The 1-SE rule classifies any tree with (xerror) within one standard deviation (the associated xstd) of the minimum as minimal. Then we simply select the simplest of these trees as our candidate. In the table above, tree numbers 4 & 5 both achieve the minimum, but tree number 6 is also classified as a minimal tree. The simplest of these is tree 4 (7 splits), and this is our candidate. We may generate and plot the candidate tree as follows:

```
> rctree3 <- prune(rctree2,cp=0.0150602)
>
> plot(rctree3,compress=T,branch=.3)
> text(rctree3,cex=.7)
```

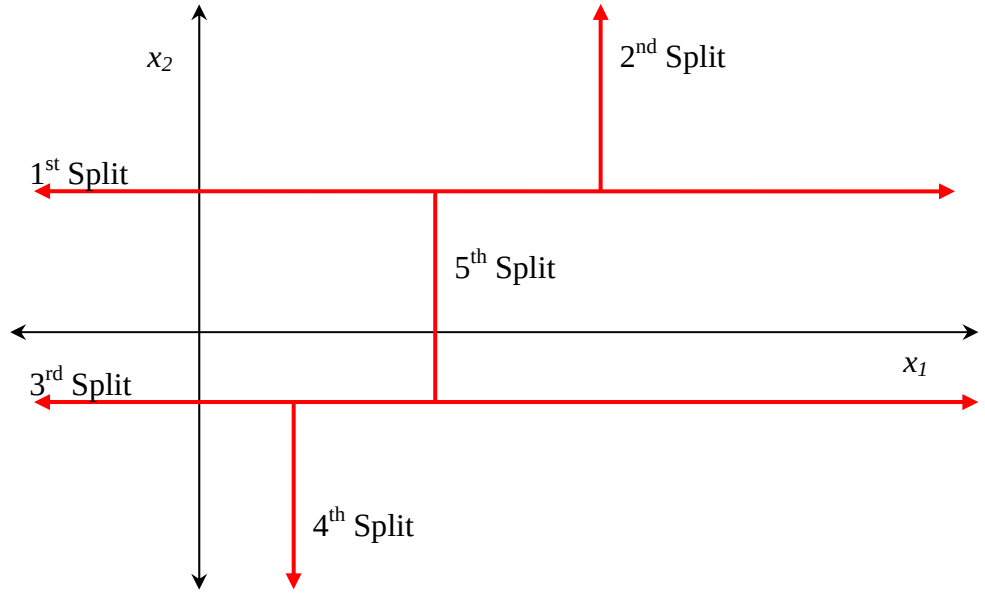


Note that this tree is not the same as the tree we produced using the system defaults, although it has the same number of splits (7), and the earlier splits are the same.

8.5 More on CART

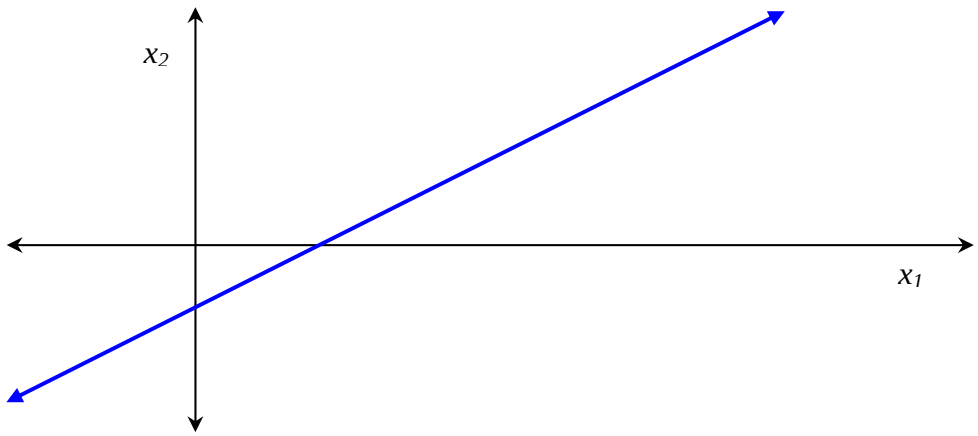
8.5.1 How CART Partitions the Space

Suppose the data set has observations made up of two features, $\mathbf{x}_i = (x_{i1}, x_{i2})$, and an associated group y_i. Then the first split made by the CART algorithm partitions the input space into two. The diagram below gives a stylized rendition. The second split pertains only to one half of the input (in this case, the upper half), and partitions that into two also. Once we have concluded fitting the tree, we have in fact partitioned the space into a collection of disjoint rectangles. Each rectangle has a group associated with it, corresponding to the group of the majority of the training set candidates that fall in that rectangle. Even from this simple example, it is clear that CART (or decision trees in general) can therefore approximate any decision surface simply by adding more and more rectangles (intersecting hyperplanes in higher dimensions).



8.5.2 Affine Splits

Never-the-less, there are simple situations that the basic CART algorithms doesn't cope with well. For example, suppose the true partition of the input space is a straight line as follows:



In this case the basic CART algorithm will fit a very complicated tree as it attempts to approximate the affine line with vertically aligned rectangles. To deal with this sort of problem, CART allows a more advanced option, namely to split on linear combinations of the features. In this case, CART would be able to perfectly partition the space in a single split. However the extra modeling power does come at a significant increase in the computational complexity of the algorithm, since both combinations of features and combinations of their various split levels need to be considered. It is generally not recommended to allow splits that combine more than a small number of features.

8.5.3 Surrogate & Competitive Splits and Variable Importance

A very powerful and significant feature of the CART algorithm is that it supplies a built in facility for handling missing values (NAs). When the algorithm selects a split on a particular node it also looks for alternative splits that can approximate the selected split. These *surrogate splits* are chosen as the splits which route the training examples as closely as possible to the routing of the selected split. CART normally generates a list of surrogate splits for each node, the length of which can be determined by the user. When a new observation is presented to the tree for classification, CART begins routing it according to its primary splits. If, at some node, CART encounters an NA value in the feature on which the primary split is made, CART simply attempts to use the first surrogate split, and so on down the surrogate list until a split is found matching a feature that exists in the observation.

CART also employs the concept of a *competitive split*. This is a split which achieves either the same, or almost the same reduction in impurity as the primary split on a node. Competitive splits are not used in building the tree, but they can be listed at the conclusion of the algorithm so that a domain expert can see what other splits are available in the tree. This can be particularly useful when the decision tree is employed for explanatory purposes.

The list of primary, surrogate and competitive splits can also be used to assign an importance to each of the features in the input space. That is, the fact that a feature doesn't actually appear in the primary tree does not mean it is not a significant feature since it may occur in many surrogate splits, or competitive splits. This *variable importance* can also help in understanding the model.

9 Classification: k-Nearest Neighbor

([HMS:2001],10.6; [R:1996],6.2; [H:1981])

9.1 *k*-NN Classification Rule

The basis of using a k-NN classifier is to assign a class to a case with feature vector $\mathbf{x}$, based on the known classes of the k nearest neighbors to $\mathbf{x}$ in the learning set.

This approach can be justified by showing that it is an approximation to the Bayes rule for classification – which we know is optimal in the sense of minimum error rate. The argument can be explained as follows.

Suppose we have a given input vector $\mathbf{x}$ and we wish to estimate the probability (density) at $\mathbf{x}$ for the distribution of points in class C_i. Let L be a (small) ball in the space of input vectors such that $\mathbf{x} \in L$, then the probability of getting an observation of class C_i in the neighborhood L of $\mathbf{x}$ is

$$\theta = \int_{\zeta \in L} p(\xi \mid C_i) d\xi.$$

If L is sufficiently small, then this probability can be approximated by

$$\theta \approx p(\mathbf{x} \mid C_i) \cdot V \quad \text{, from which we get } \quad p(\mathbf{x} \mid C_i) \approx \frac{\theta}{V}$$

where V is the volume of the ball L.

Now suppose there are n points in the data set, of which k fall in the ball L. Suppose further that of the n points, n_i are of class C_i, and of the k points in the ball, k_i are of class C_i. Then the probability θ can be estimated simply be counting the proportion of points of class C_i that fall in the ball. That is

$$\hat{\theta} = \frac{k_i}{n_i},$$

which gives rise to the density estimator

$$\hat{p}(\mathbf{x} \mid C_i) = \frac{k_i}{n_i V}.$$

From this point we may proceed in one of two ways:

1. Fix the volume V and see how many points fall in the ball. This pathway leads to a kernel estimation of the density.
2. Fix k, the number of points residing in the ball, and choose the smallest ball containing k points around $\mathbf{x}$.

We proceed in the second manner, which leads to *k-nearest neighbor (k-NN)* estimates of the density, and k-NN classification.

> ASIDE
> Consider what happens to this density estimate as n_i varies. On the one hand, our estimator will be subject to less random variation if we take k_i (and correspondingly the ball L) as large as possible: so let k_i increase as n_i increases. On the other hand, errors introduced by the averaging effect will be

small if the volume of L is smaller, and V can be kept small by letting $\frac{k_i}{n_i} \to 0$ as n_i increases. This suggests we enforce two conditions on our selection of k as the number of observations increases

1. $\lim_{n_i \to \infty} k_i = \infty$
2. $\lim_{n_i \to \infty} \frac{k_i}{n_i} = 0.$

These conditions are necessary and sufficient for $\hat{p}(\mathbf{x} \mid c_i)$ to converge in probability to $p(\mathbf{x} \mid c_i)$ at all points of continuity of $p(\mathbf{x} \mid c_i)$.

The following probability estimates are obvious assuming the sample is representative of the population:

$$\hat{P}(c_i) = \frac{n_i}{n} \text{ and } \hat{p}(\mathbf{x}) = \frac{k}{nV},$$

Thus using Baye's rule we arrive at the following:

$$\hat{P}(c_i \mid \mathbf{x}) = \frac{\hat{p}(\mathbf{x} \mid c_i)\hat{P}(c_i)}{\hat{p}(\mathbf{x})} = \frac{k_i}{k}.$$

This leads to the *k-NN classification rule*, namely let a new observation $\mathbf{x}$ belong to class c_i if $k_i = \max_j k_j$.

9.1.1 Selecting k

In the most basic form, we can take $k = 1$, in which case a new point is classified as the class of its nearest neighbor. In highly separated data spaces this classifier can perform well, although it generally makes for a rather unstable classifier (high variance, sensitive to data). The predictions can be made more stable by increasing k, although this may increase the bias of the method since there is more averaging. In the extreme case this will simply return the most populous group, no matter what the observation is.

There is some theoretical work on the best choice of k, but it is somewhat outside the scope of the current discussion ([R:1996] is a good starting point). In practice it is usually best to choose k in some data-adaptive way. Namely, try various values and plot some performance criterion (e.g. misclassification rate) against k. Of course, this evaluation set must be carried out on a data set independent of the training set. Alternatively, if the data set is small, a leave-one-out cross validation approach may be used.

9.1.2 Choosing the Metric

Clearly the choice of metric can greatly influence the effectiveness of the nearest neighbor methods. Please refer back to the extended discussion on distances and metrics in Chapter 2 of these notes to review some of the commonly used metrics. Note that any metric thought to be a reasonable measure of the distance between two

observations in the input space can be used for a k-NN implementation.

In practice the *Euclidean metric* (see 2.2.2) is normally used (this is the metric used in R), but after some suitable scaling (perhaps normalization) of the variables.

Alternatively, if the within-class distributions are roughly normal and of similar covariance matrix, it may make sense to use an (estimated) *Mahalanobis distance* (see 2.2.2).

In the case of mixed variable types we may choose a metric such as *Gower's dissimilarity measure* (see 2.2.6), which can automatically deal with features of different types in the input space.

One of the most important steps in choosing a metric may be to exclude features which have little or no relevance. The features selection by classification tree can be a very useful guide.

9.1.3 Example

We return to the Recumbent Cows dataset and predict the outcome with a k-NN model. In the `R' environment, k-NN predictions can be made using the `knn' function in the `class' package. This implementation uses the Euclidean metric exclusively, and allows the user to specify the number of nearest neighbors. For this example we will use the predictors "AST", "Urea", "Uketone", "Calving", "Daysrec" and "CK", and omit all observations with NA values. This data is found in the "RC Subset1.txt" data file.

```
> rcdata <- read.table("C:\\RData\\Cows\\RC Subset1.txt",header=TRUE)
>
> # set up the outcome levels so they mean something
> rcdata[,"Outcome"] <- as.factor(rcdata[,"Outcome"])
> levels(rcdata[,"Outcome"]) <- c("Dead","Living")
>
>
> input <- rcdata[,1:6]    # predictors
> cl <- rcdata[,7]  # classification of input observations
```

Suppose we wish to predict the class of 1/3 of the dataset based on a 1-NN model from the remaining 2/3.

```
> library(class)
>
> ## classify 1/3 of the set based on the other 2/3 using 1-NN
>
> trainidx <- sample(1:nobs,2*nobs/3,replace=FALSE)
> testidx <- (1:nobs)[-trainidx]
> train <- input[trainidx,]
> test <- input[testidx,]
> trncl <- cl[trainidx]
> tstcl <- cl[testidx]
>
> pred <- knn(train,test,trncl,k=1)
>
> # misclassification rate
> sum(pred!=tstcl)/length(tstcl)
 [1] 0.3809524
>
```

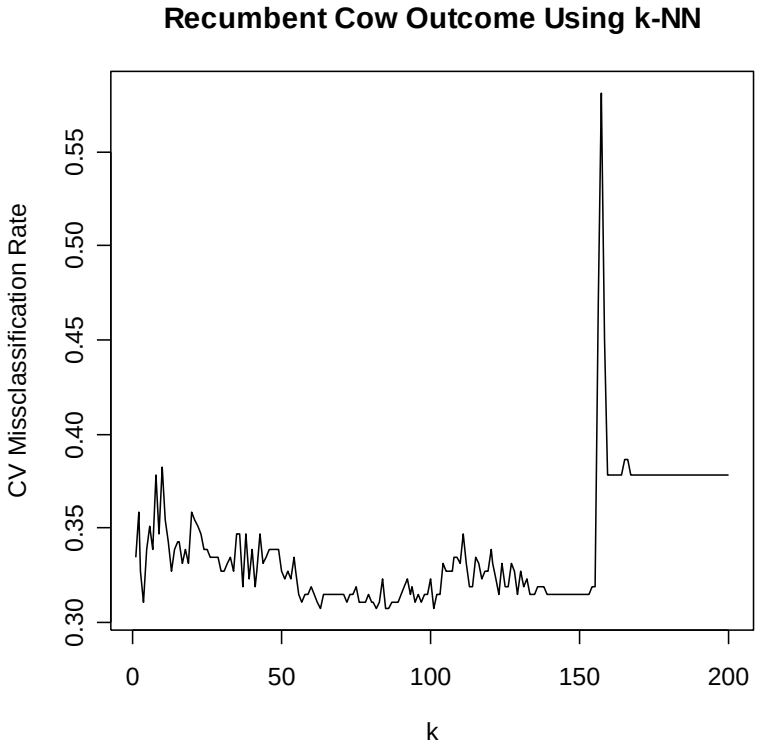
Note that the prediction error here is about 0.38. This is only marginally better than classifying all the test set observations as "Dead":

```
> sum(tstcl!="Dead")/length(tstcl)
[1] 0.3928571
```

This raises the issue of which k to choose. R supplies a function – `knn.cv' that uses leave-one-out cross-validation to predict the

class of each observation in the dataset. That is, each observation is left out of the dataset in turn, and its class is predicted using `knn' on the rest of the dataset. We may use this function to get an idea of which k might be appropriate for this dataset.

```
> ## use knn.cv to find the best k value
>
> xmiss <- NULL
>
> for ( k in 1:200 )
+ {
+      pred <- knn.cv(input,cl,k=k)
+      xmiss <- c(xmiss,sum(pred!=cl)/nobs)
+ }
>
> plot(xmiss, type="l", ylab="CV Missclassification Rate", xlab="k",
+ main="Recumbent Cow Outcome Using k-NN")
>
```



This plot suggest that the misclassification rate is relatively stable at around 0.31 between k=60 and k=100. As k approaches 200, the misclassification error settles to the null error rate (every prediction returned as "Living", which is the most populous response).

9.2 Limitations of *k*-NN Classification

Once the parameter k is chosen and a metric is selected, no further parameters are necessary for *k*-NN classification. This means the process of fitting the model is extremely simple. However the method does have the significant disadvantage that it is expensive both computationally and memory-wise. A naïve implementation requires storing and accessing the entire set of training data, and computing the distance between the input vector and each training point. Supposing the input set consists of n observations, each of dimension p, and that each coordinate is a floating point number (1 byte), then a naïve implementation will require np bytes of storage,

$3np + n$ algebraic operations (if the Euclidean metric is used), and $O(n^2)$ comparisons for each new input.

9.2.1 Data Editing Algorithms

These limitations can be overcome to some extent by *data editing* algorithms. Data editing algorithms attempt to reduce the number of data points in the training set to a smaller number of more significant points. In some cases these methods can also be useful for improving performance by removing outliers.

An example of a data editing algorithm is the *multiedit* algorithm, which requires user-supplied parameters I and V.

1. Put all patterns in the current set.
2. Divide (randomly) the current set more or less equally into $V \geq 3$ sets. (E.g. call the 3 subsets V1, V2 and V3.) Use pairs of subsets cyclically as test and training sets (e.g. V1->V2, V2->V3, V3->V1).
3. For each pair of sets, classify the test set using the k-nn rule from the training set. Mark those patterns in the successive test sets which were incorrectly classified.
4. At the end of the cycle, delete from the current set those patterns which were marked as incorrectly classified.
5. If any patterns were deleted in the last I iterations (cycles), return to step 2.

The edited data set is then used for prediction using the 1-NN rule. In practice this algorithm can perform quite badly if presented with high dimensional data, or large numbers of classes. An alternative algorithm is an example of a *condensing* algorithm, which seeks to retain only the crucial exterior (boundary) points in the clusters.

1. Divide the current patterns into a store and a grabbag. One possible partition is to put the first point in the store, the rest in the grabbag.
2. Classify sequentially each sample in the grabbag by the 1-nn rule using the store as training set. If the sample is incorrectly classified transfer it to the store.
3. Return to 2 unless no transfers occurred in the last pass over the sample or the grabbag is empty.
4. Return the store.

A further refinement of this is the *reduced nearest neighbor rule* which goes back over the condensed training set and drops any patterns (one at a time) which are not needed to correctly classify the remainder of the original training set.

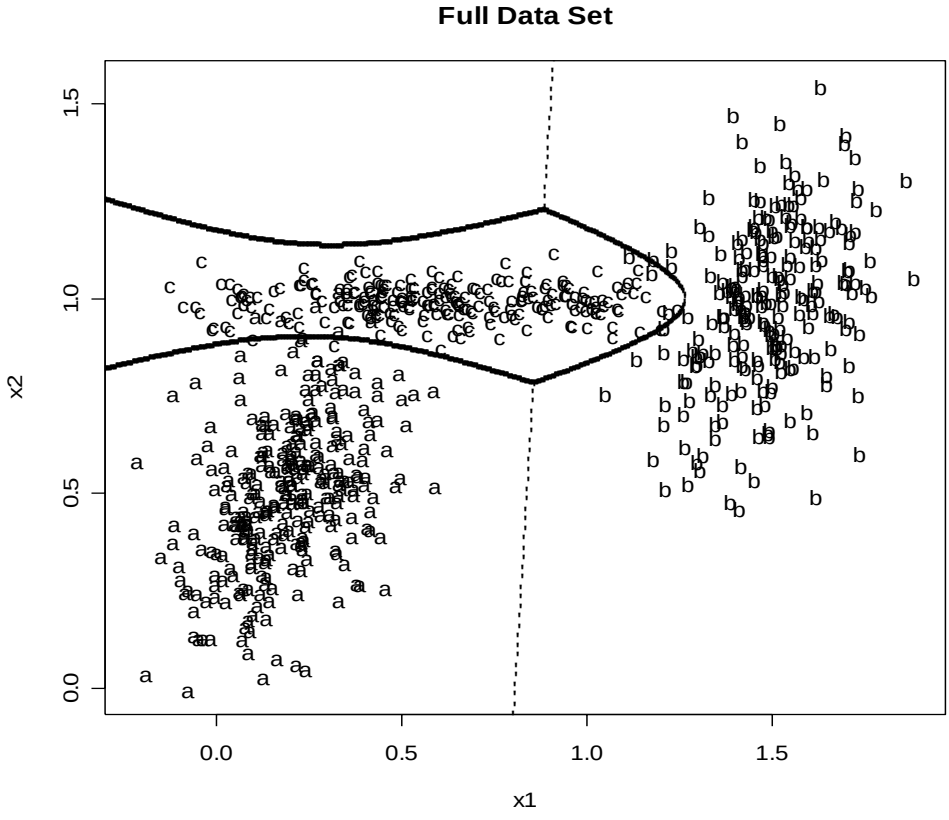
It is probably wise to implement data editing algorithms with caution since all of them can go too far in some circumstances and seriously degrade the accuracy of the classifier.

9.2.2 Example

R has functions for the multiedit, condense and reduced nearest neighbor methods. To demonstrate the effect of these algorithms we shall generate an artificial data set with a 2-D feature space. The set has three data types, `a', `b' and `c', and a total of 780 observations (the observations are multivariate normal with different means and covariance matrices for the different groups). A plot of the full data set is displayed below. The bayes borders (optimal

separation between the groups based on the actual distributions) are also included, although the R code for this is not supplied.

```
> ## generate 3 groups of bivariate normals
>
> library(MASS) # for multivariate normal generation
>
> S <- matrix(c(0.02,0.01,0.01,0.04),nrow=2)
> S3 <- matrix(c(0.1,0,0,0.002),nrow=2)
> mu1 <- c(0.2,0.5)
> mu2 <- c(1.5,1)
> mu3 <- c(0.6,1)
>
> # generate training observations
>
> ob1 <- mvrnorm(300,mu1,S)
> ob2 <- mvrnorm(240,mu2,S)
> ob3 <- mvrnorm(240,mu3,S3)
>
> # generate test observations
> tob1 <- mvrnorm(600,mu1,S)
> tob2 <- mvrnorm(480,mu2,S)
> tob3 <- mvrnorm(480,mu3,S3)
>
> ## now create group vectors for training and test data
>
> grps <- as.factor(c(rep("a",dim(ob1)[1]),rep("b",dim(ob2)[1]),
+ rep("c",dim(ob3)[1])))
> obs <- rbind(ob1,ob2,ob3)
>
> # test observations
> tgrps <- as.factor(c(rep("a",dim(tob1)[1]),rep("b",dim(tob2)[1]),
+ rep("c",dim(tob3)[1])))
> tobs <- rbind(tob1,tob2,tob3)
>
> nobs <- dim(obs)[1]
> ntobs <- dim(tobs)[1]
>
> plot(obs[grps=="a",],xlim=range(obs[,1]),ylim=range(obs[,2]),
+ xlab="x1", ylab="x2",pch="a",main="Full Data Set")
> points(obs[grps=="b",],pch="b")
> points(obs[grps=="c",],pch="c")
```



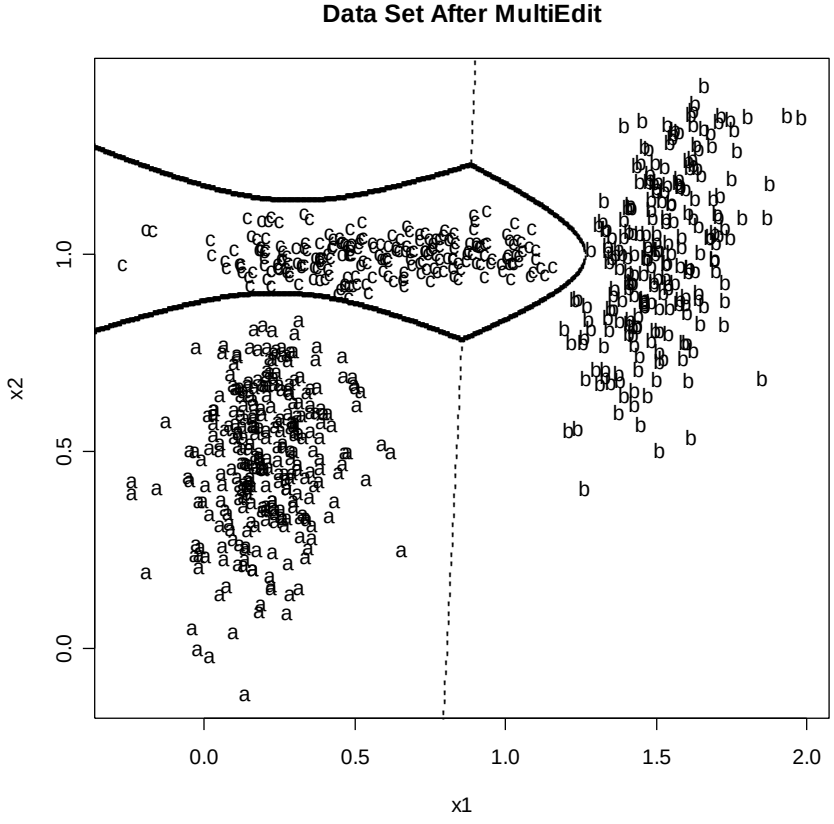
A further 1560 test observations were generated from the same distribution to give an indication of the influence of the various data editing algorithms on classification accuracy. For the full training data, the 1-NN classification rule is as follows:

```
> ## regular knn (k=1)
>
> pred1 <- knn(obs,tobs,grps,k=1)
>
> # misclassification rate
> sum(pred1!=tgrps)/length(tgrps)
[1] 0.02435897
```

This is the error rate for 1-NN with 780 randomly chosen observations.

The multiedit algorithm is executed in R using the command `multiedit'. On our dataset, multiedit reduces the size of the training data from 780 to 748. As the plot shows, the effect is to create more homogenous groups. Bayes borders are once again included.

```
> ## multiedt - refine k-NN rule
>
> mekeep <- multiedit(obs,grps,k=1,V=3,I=5,trace=FALSE)
>
> # number of training observations retained
> length(mekeep)
[1] 748
> plot(obs[grps=="a"& mekv,],xlim=range(obs[,1]),ylim=range(obs[,2]),
+ xlab="x1",ylab="x2",pch="a",main=" Data Set After MultiEdit")
> points(obs[grps=="b"& mekv,],pch="b")
> points(obs[grps=="c"& mekv,],pch="c")
>
```

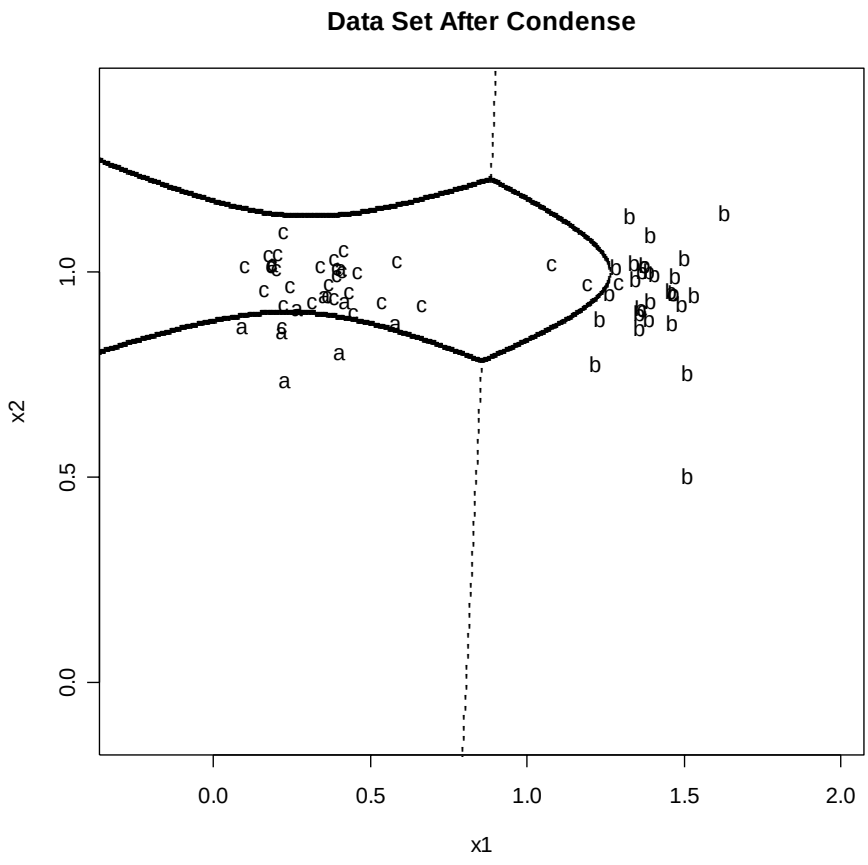


The misclassification rate after multiedit is actually somewhat better than with the original dataset:

```
> mepred <- knn(obs[mekeep,],tobs,grps[mekeep],k=1)
>
> # misclassification rate
> sum(mepred!=tgrps)/length(tgrps)
[1] 0.02051282
```

The condense algorithm is implemented using the R command `condense'. The plot below shows the training data remaining after running condense. Note how the points concentrate around the Bayes borders between the groups. This method does lead to an increase in the misclassification rate on the test set to 0.0308, however the number of points retained in the training set is reduced from 780 to 56.

```
> ## condense
>
> cdkeep <- condense(obs,grps,trace=FALSE)
>
> # number of training observations retained
> length(cdkeep)
[1] 56
>
> # plot command
>
> # keep vector
> cdkv <- !is.na(match(1:nobs,cdkeep))
>
> plot(obs[grps=="a"& cdkv,],xlim=range(obs[,1]),ylim=range(obs[,2]),
+ xlab="x1",ylab="x2",pch="a",main="Data Set After Condense")
> points(obs[grps=="b"& cdkv,],pch="b")
> points(obs[grps=="c"& cdkv,],pch="c")
>
```

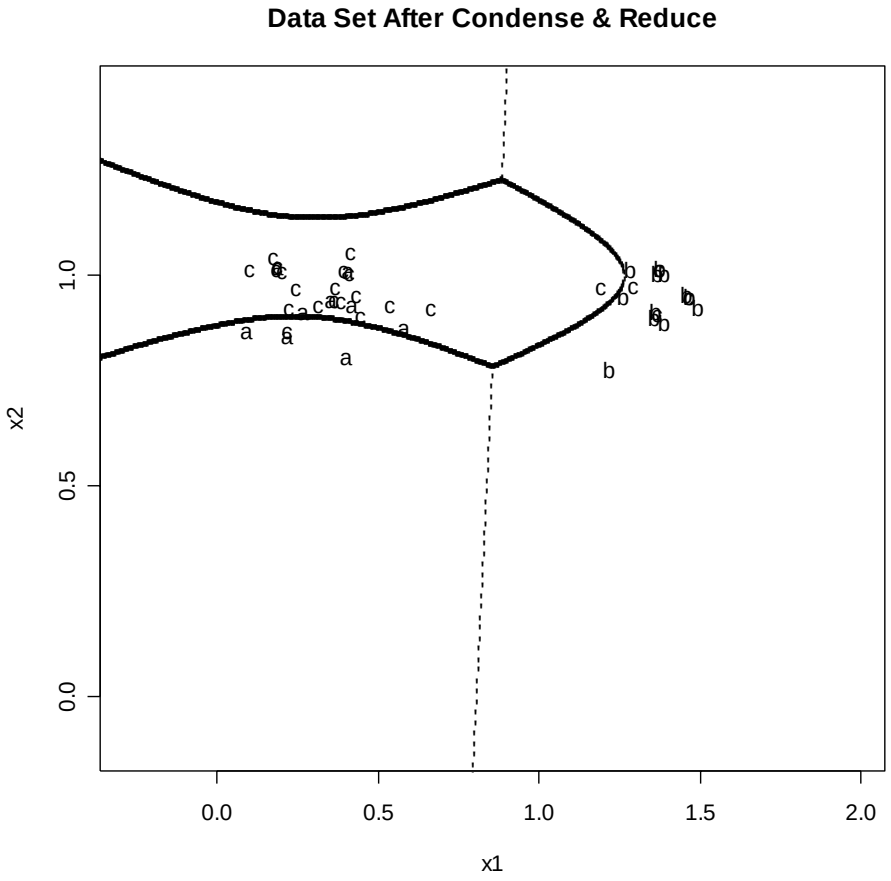



```
> # predict and report the misclassification rate
```

```
> cdpred <- knn(obs[cdkeep,],tobs,grps[cdkeep],k=1)
>
> # misclassification rate
> sum(cdpred!=tgrps)/length(tgrps)
[1] 0.03076923
>
```

Finally the reduce algorithm (command `reduce.nn') can be run after `condense' to further cut down the training set (although that hardly seems necessary in this case). This procedure reduces the training set size further, to 35, and is accompanied by a further increase in the misclassification rate, to 0.0386.

```
> ## condense and reduce.nn
>
> rdkeep <- reduce.nn(obs, cdkeep, grps)
>
> # number of training observations retained
> length(rdkeep)
[1] 35
>
> # keep vector
> rdkv <- !is.na(match(1:nobs,rdkeep))
>
> plot(obs[grps=="a"& rdkv,],xlim=range(obs[,1]),ylim=range(obs[,2]),
+ xlab="x1",ylab="x2", main=" Data Set After Condense & Reduce",
+ pch="a")
> points(obs[grps=="b"& rdkv,],pch="b")
> points(obs[grps=="c"& rdkv,],pch="c")
>
```




```
> # predict and report misclassification rate
> rdpred <- knn(obs[rdkeep,],tobs,grps[rdkeep],k=1)
>
> # misclassification rate
> sum(rdpred!=tgrps)/length(tgrps)
[1] 0.03461538
>
```

9.2.3 Example

Now we shall return to the Recumbent Cows dataset investigated above. Once again we use each of the data editing algorithms to refine the 1-NN rule in the example above. The 1-NN rule gave a misclassification rate of 0.38 and required 167 training observations.

```
> # standard 1-NN rule
>
> pred <- knn(train,test,trncl,k=1)
>
> # number of training observations
> dim(train)[1]
[1] 167
>
> # misclassification rate
> sum(pred!=tstcl)/length(tstcl)
[1] 0.3809524
```

Applying `multiedit' to the training data set provides a large reduction in the number of observations required (from 167 to 71), and an impressive reduction in the misclassification rate (from 0.38 to 0.25).

```
> ## multiedt - refine 1-NN rule
> mekeep <- multiedit(train,trncl,k=1,V=3,I=5,trace=FALSE)
> mepred <- knn(train[mekeep,],test,trncl[mekeep],k=k)
> # number of training observations retained
> length(mekeep)
[1] 71
> # misclassification rate
> sum(mepred!=tstcl)/length(tstcl)
[1] 0.25
```

The gains by the `condense' method are significant, but not as great as with multiedit. The number of observations is reduced from 167 to 92, and the training set misclassification rate is reduced from 0.38 to 0.36

```
> ## condense
> cdkeep <- condense(train,trncl,trace=FALSE)
> cdpred <- knn(train[cdkeep,],test,trncl[cdkeep],k=k)
> # number of training observations retained
> length(cdkeep)
[1] 92
> # misclassification rate
> sum(cdpred!=tstcl)/length(tstcl)
[1] 0.3571429
>
```

Following the condense method with the `reduce.nn' function further improves the condensed training set. From 92 observations, 82 are selected, and the misclassification rate on the test set is reduced to 0.33

```
> ## condense and reduce.nn
> rdkeep <- reduce.nn(train, cdkeep, trncl)
> rdpred <- knn(train[rdkeep,],test,trncl[rdkeep],k=k)
> # number of training observations retained
> length(rdkeep)
[1] 82
> # misclassification rate
> sum(rdpred!=tstcl)/length(tstcl)
[1] 0.3333333
>
```

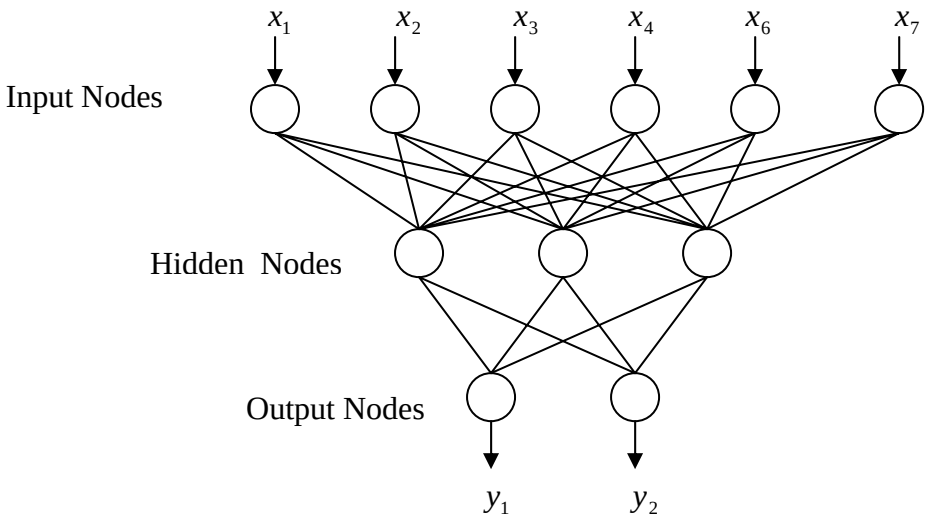
10 Nonparametric Regression: Neural Networks

[R:1996]

10.1 Introduction

Artificial Neural Network (ANN) models have become extremely popular over the previous 15 years or so in a wide variety of regression and classification tasks. Their popularity stems partly from the language usually employed in describing the model and its functions which evoke images of robotic brains. Apart from their intuitive appeal, ANN models also have a very elegant and efficient training method through *backpropagation.*

An ANN can be thought of as a multilayered network, with the nodes of the top layer representing the *input variables* (predictors in multiple regression), and bottom layer nodes representing the *outputs* (response in multiple regression). The input and output layers are linked by a series of one or more *hidden layers*, consisting of a set of nodes performing a simple non-linear function (*activation function)* on the sum of their inputs from the previous layer.



A single hidden layer ANN with input $\mathbf{x} = (x_0 = 1, x_1, \ldots, x_n)$ (the entry $x_0 = 1$ is a simple shorthand for allowing a constant or *bias* term at each hidden node), target values $\mathbf{y} = (y_1, \ldots, y_m)$ and k hidden nodes with activation function $\phi(\cdot)$ can be represented by:

$$f(\mathbf{x}) = (f_1(\mathbf{x}), \ldots, f_m(\mathbf{x}))$$

where

$$f_j(\mathbf{x}) = \phi_o\left(w_{0j} + \sum_{q=1}^{k} w_{qj}^{(2)} \phi\left(\sum_{p=0}^{n} w_{pq}^{(1)} x_p\right)\right),$$

and $w_{pq}^{(1)}, w_{qj}^{(2)}$ represent the real valued *weights* (regression coefficients in multiple regression) from input p to hidden node q in the first layer, and from hidden node q to output j in the second layer, respectively. The extra parameter w_{0b} is a bias term for the output nodes. The function ϕ_o is the *output function* of the

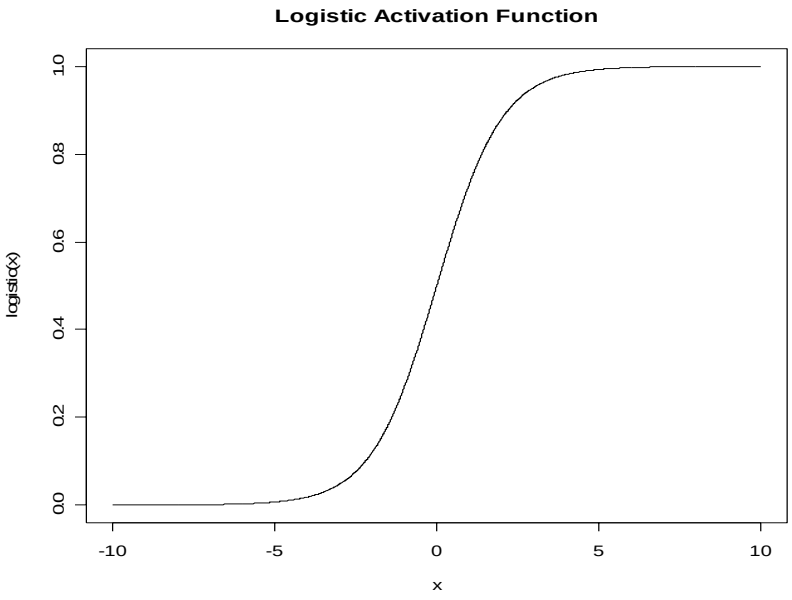
network, and is generally taken to be either the identity function (linear outputs), or the same as the activation function.

Note that if both activation functions are set to the identity function, then the form reduces to a standard multiple regression. The concept of an activation function is borrowed from the neurological idea that neurons in the brain "activate" under certain conditions, and remain dormant otherwise. Whether this is a reasonable assessment of neuron behavior is not important for the current discussion, however it does give some inkling how to proceed in selecting an activation function. The obvious choice would be a threshold function which remains at some initial level (say 0) until the input reaches a threshold level α, at which point the function jumps to some constant "active" level until the input drops below α again.

There is actually no conceptual problem with using threshold activation for the hidden units of a neural network, but it does lead to some computational issues. In particular, neural networks are usually trained using some form of gradient descent optimization (we shall address this more later), in which case the derivatives of the activation function will be required. Of course, the threshold function is not differentiable around its threshold, so we seek candidates that are "like" the threshold function, but have the added property of being differentiable everywhere.

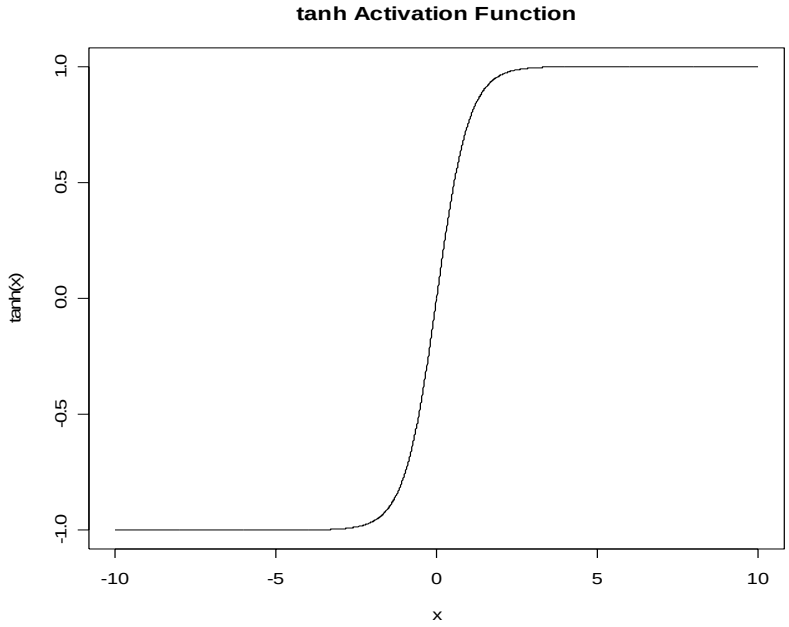
There have been many candidates proposed for the activation function ϕ satisfying this requirement, although the most popular are the logistic function:

$$\text{logistic}(x) = \frac{e^x}{1+e^x}$$



or the hyperbolic tangent function:

$$\tanh(x) = \frac{e^x - 1}{1+e^x}.$$



These functions both share the desirable properties of being smooth, monotone increasing, bounded, and having derivatives expressible entirely in terms of the function value itself. This last property turns out to be extremely useful when it comes to computing the gradients of the parameters for training.

ANN models are able to approximate to arbitrary accuracy any continuous function (see [R:1996] for references). However this flexibility comes at a price – ANN models tend to be highly prone to overfitting. Indeed, much of the research in the field is concerned with finding ways to restrict or reduce the complexity of the ANN models to increase generalization ability.

10.1.1 Other Architecture Options

There have been a number of architecture generalizations to the ANN model proposed, which do not extend the space of functions available, but may improve training time, or make the final model more comprehensible in some sense.

- Multiple hidden layers: Some users claim that multiple layers of hidden units, with fewer nodes in each layer, encourage quicker training.
- Skip layer connections: Some models allow direct linear connections from the input to the output. This allows the model to fit a linear regression through these connections (if the output function is linear) at the same time as fitting the non-linear components using the hidden layers. This has intuitive appeal and may be useful when the linear component of a data set is very strong.
- Generalized Feed Forward: This model orders the hidden and output nodes, and allows connections between any of the input nodes to any of the hidden or output nodes, and between any hidden or output node and any other hidden or output node occurring later in the order.

10.2 ANN Models for Classification and Regression

The ANN model has been adapted successfully for both classification and regression tasks. For classification tasks the target data is usually structured so that each output unit represents a single class, so our target examples are of the form $(0,\ldots,0,1,0,\ldots,0)$ with the 1 corresponding to the index of the target class. Ideally we would like the network outputs to return an approximation of the posterior probability of each class on the input observation, and the class of the observation can be chosen as the class with maximal posterior probability. If we choose to use linear output units we face the same conceptual problem as described in the logistic regression section. Namely, the network is able to return nonsensical results. This is usually addressed by including sigmoid activation on the output nodes, in which case the output is bounded and with appropriate scaling (if tanh activation is used – logistic is already bounded by 0 and 1) the predictions can be easily interpreted.

For regression tasks we may choose to use either sigmoidal or linear outputs (or some other output function). However, it is recommended to do some sort of transformation on the targets if sigmoidal outputs are used so they fall within the active range of the output function. Once again, if the potential targets have a range greater than the output nodes, nonsensical output may result. More elaborate transformation of the targets has also been proposed, although this is usually problem dependent.

10.2.1 Error Functions

There have been a myriad of error functions proposed for the ANN model. We shall only mention some of the most important here. Note that all neural networks using backpropagation based training need differentiable hidden units and a differentiable error function.

By far the most common error function for both classification and regression is the now familiar sum of squared errors. Using the notation above, the sum of squared errors for a model $f(\mathbf{x};\mathbf{w})$ is given by

$$E(\mathbf{w}) = \sum_i \left\| \mathbf{y}^{(i)} - \mathbf{f}(\mathbf{x}^{(i)};\mathbf{w}) \right\|^2 = \sum_i \sum_j \left(y_j{}^{(i)} - f_j(\mathbf{x}^{(i)};\mathbf{w}) \right)^2$$

where i ranges over the set of input-target pairs $(\mathbf{x}^{(i)},\mathbf{y}^{(i)})$. Note that the sum of squares sums over both output units and input-target pairs.

An alternative error function for classification is the *entropy* error, which is related to the log-likelihood and is in fact equal to the deviances from logistic regressions (G^2) summed over all the logistic output nodes. Assume each $y_j^{(i)} = 1 \text{ or } 0 \text{ for each } j = 1,...,m$ representing the presence or absence of some property. Then

$$E(\mathbf{w}) = \sum_{i=1}^{N} \sum_{j=1}^{m} \left[y_j^{(i)} \ln \frac{y_j^{(i)}}{f_j(\mathbf{x}^{(i)};\mathbf{w})} + \left(1 - y_j^{(i)}\right) \ln \frac{1 - y_j^{(i)}}{1 - f_j(\mathbf{x}^{(i)};\mathbf{w})} \right].$$

In this case φ_0 would be chosen as the logistic activation function.

In a general classification problem with *m* classes, only one of the $y_j^{(i)}$ is 1, and all the rest are 0. Then the approach called *softmax* is often used with linear output functions φ_0 . For each j, compute

$$p(j \mid \mathbf{x}^{(i)}) = \frac{\exp(f_j(\mathbf{x}^{(i)}, \mathbf{w}))}{\sum_{s=1}^{m} \exp(f_s(\mathbf{x}^{(i)}, \mathbf{w}))}$$

and assign the case *i* to the class with largest value of $p(j \mid \mathbf{x}^{(i)})$. Then the appropriate negative log-likelihood gives the error function:

$$E(\mathbf{w}) = \sum_{i=1}^{N} \sum_{j=1}^{m} \left[y_j^{(i)} \ln \frac{y_j^{(i)}}{p(j \mid \mathbf{x}^{(i)})} \right].$$

In general, let the error on input observation i be given by $E^{(i)}(\mathbf{w})$, thus $E(\mathbf{w}) = \sum_i E^{(i)}(\mathbf{w})$.

10.3 Derivation of Backpropagation

A neural network model is fit to the data in the following manner. For a given error function $E(\mathbf{w})$ we seek $\mathbf{w}$ that minimize this function. Let $w(l)_{jk}$ represent the weight linking node j in layer $l-1$ to node k in layer l. Then the error function is minimized when

$$\frac{\partial E(\mathbf{w})}{\partial w(l)_{jk}} = 0$$

for all weights $w(l)_{jk}$.

Let $x(l)_j^{(i)}, z(l)_j^{(i)}$ represent respectively the *input to,* and *output from,* the j^{th} node on level l, upon presentation of input pattern i. Thus $\mathbf{x}^{(i)} = (x_1^{(i)}, \ldots, x_n^{(i)}) = (x(0)_1^{(i)}, \ldots, x(0)_n^{(i)})$, $z(l)_j^{(i)} = \phi(x(l)_j^{(i)})$ for all hidden layers l, and $z(o)_j^{(i)} = f_j(\mathbf{x}^{(i)}; \mathbf{w})$, where o is the output layer.

Consider the weight $w(o)_{jk}$ in the output layer. The only term in the error function (presuming the error is additive) that depends on $w(o)_{jk}$ is the output

$$z(o)_k^{(\cdot)} = \phi_o\left(x(o)_k^{(\cdot)}\right) = \phi_o\left(\sum_{j \in o-1} w(o)_{jk}\, z(o-1)_j^{(\cdot)} \right).$$

Thus

$$\frac{\partial E^{(i)}(\mathbf{w})}{\partial w(o)_{jk}} = \frac{\partial z(o)_k^{(i)}}{\partial w(o)_{jk}} = \frac{\partial z(o)_k^{(i)}}{\partial x(o)_k^{(i)}} \frac{\partial x(o)_k^{(i)}}{\partial w(o)_{jk}} = \left(\phi_o'(x(o)_k^{(i)})\right) z(o-1)_j^{(i)}.$$

Note that the derivative of the activation function can be represented entirely in terms of the function itself. For example if ϕ_o is logistic, the derivative is simply

$$\phi_o(x(o)_k^{(i)})\left(1-\phi_o(x(o)_k^{(i)})\right) = z(o)_k^{(i)}(1-z(o)_k^{(i)}) .$$

Therefore we require only the network outputs and the outputs from the last hidden layer before the output layer to compute these derivatives. This suggests an efficient method for computing the derivatives of the error function with respect to the last layer weights:

1. For each input pattern, evaluate the network output, saving the hidden node outputs on the way.
2. At the conclusion of each input evaluation, compute the derivatives of the error function with respect to the last layer weights corresponding to that pattern.

In this way, obtaining the derivatives with respect to the last layer weights requires only marginally more time than computing the output on each pattern.

In more general terms, the derivative of the error function with respect to weight $w(l)_{jk}$ is given by

$$\frac{\partial E^{(i)}(\mathbf{w})}{\partial w(l)_{jk}} = \frac{\partial E^{(i)}(\mathbf{w})}{\partial x(l)_k^{(i)}}\frac{\partial x(l)_k^{(i)}}{\partial w(l)_{jk}} = z(l-1)_j^{(i)}\frac{\partial E^{(i)}(\mathbf{w})}{\partial x(l)_k^{(i)}} = z(l-1)_j^{(i)}\frac{\partial E^{(i)}(\mathbf{w})}{\partial z(l)_k^{(i)}}\frac{\partial z(l)_k^{(i)}}{\partial x(l)_k^{(i)}}$$

and the last term can be shown to be equal to

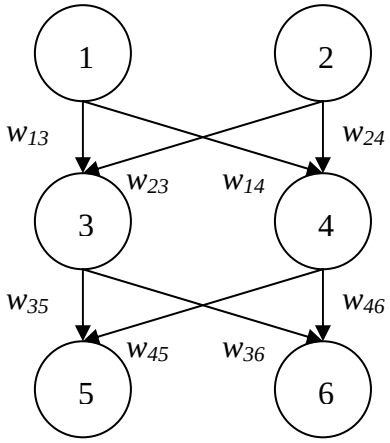
$$z(l-1)_j^{(i)}\phi'(x(l)_k^{(i)})\sum_{p\in l+1} w(l+1)_{kp}\frac{\partial E^{(i)}(\mathbf{w})}{\partial x(l+1)_p^{(i)}} .$$

Thus the derivatives of the error function with respect to the weights on each layer can be represented in terms of only the outputs on the previous layer, the derivatives of the activation functions on the current layer, and the derivatives of the error function on later layers.

These observations are central to the *backpropagation* algorithm, which is central to many neural network training schemes. Backpropagation is essentially an efficient method for computing the first derivatives (and second, with a bit more notational spaghetti) of the error function. As we alluded to before, the algorithm involves a forward pass, in which the input patterns are presented to the network and the partial results at each hidden node are recorded. This is followed by a backward pass, in which the data gathered in the forward pass is used to progressively build up a vector of partial derivatives of the error function.

10.3.1 Simple Backpropagation Example

To illustrate the backpropagation concept, consider the following 2 input, 2 hidden node, 2 output neural network. The network function is then $f(\mathbf{x}) = (f_1(\mathbf{x}), f_2(\mathbf{x}))$. Suppose the network is presented with a pattern $\mathbf{x} = (x_1, x_2)$ which has a corresponding target value $\mathbf{y} = (y_1, y_2)$.



To simplify the notation on this example, we dispense with the more general $x(l)_j^{(i)}$ notation and represent each node only by its index. So x_i is the input to node i (so (x_1, x_2) is the input to the network, as it should be), and let z_i is the output from node i. Thus $z_1 = x_1$, $z_2 = x_2$, $z_5 = f_1(\mathbf{x})$ and $z_6 = f_2(\mathbf{x})$. Furthermore, we assume the network has linear output units, so $x_5 = z_5$ and $x_6 = z_6$. For the purposes of this example we shall use E equal to the sum of squared errors.

To train the network using a gradient based training scheme, we need to compute the error $E(f(\mathbf{x}), \mathbf{y})$, and the derivative of the error with respect to the weight vector $\frac{\partial E}{\partial \mathbf{w}} = \left(\frac{\partial E}{\partial w_{ij}} \right)_{ij}$.

Then the *forward pass* to compute the error, and the *backward pass* to compute the derivatives, can be computed as follows:

Forward Pass

In the forward pass we compute the error E as well as the x_i and z_i values.

- Firstly compute the input and output values connected with the weights between the input and the hidden layers:

$$x_3 = w_{13} z_1 + w_{23} z_2$$
$$x_4 = w_{14} z_1 + w_{24} z_2$$
$$z_3 = \phi(x_3)$$
$$z_4 = \phi(x_4).$$

- Then the input and output values connected with the weights between the hidden and output layers:

$$x_5 = w_{35} z_3 + w_{45} z_4$$
$$x_6 = w_{36} z_3 + w_{46} z_4.$$

- Finally we may compute the error:

$$E(f(\mathbf{x}), \mathbf{y}) = (z_5 - y_1)^2 + (z_6 - y_2)^2 .$$

Backward Pass

In the backward pass we wish to compute $\frac{\partial E}{\partial w_{ij}}$ for all the network weights. Note that in this kind of network architecture, the error depends on weight w_{ij} only through the destination node j. That is,

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial x_j}\frac{\partial x_j}{\partial w_{ij}},$$

and the term $\frac{\partial x_j}{\partial w_{ij}}$ can be easily seen to be z_i. Therefore

$$\frac{\partial E}{\partial w_{ij}} = z_i \frac{\partial E}{\partial x_j},$$

and the problem reduces to computing the derivatives $\frac{\partial E}{\partial x_j}$.

- Firstly we compute the derivatives of the error function with respect to the output nodes. In this case, since the network has linear outputs and E is the sum of squared errors, then:

$$\frac{\partial E}{\partial x_5} = \frac{\partial E}{\partial z_5} = 2(z_5 - y_1)$$

$$\frac{\partial E}{\partial x_6} = 2(z_6 - y_2).$$

- Then we compute the derivatives with respect to the hidden layer nodes:

$$\frac{\partial E}{\partial x_3} = \frac{\partial E}{\partial z_3}\frac{\partial z_3}{\partial x_3} = \phi'(x_3)\left\{ w_{35}\frac{\partial E}{\partial x_5} + w_{36}\frac{\partial E}{\partial x_6} \right\}$$

and similarly

$$\frac{\partial E}{\partial x_4} = \phi'(x_4)\left\{ w_{45}\frac{\partial E}{\partial x_5} + w_{46}\frac{\partial E}{\partial x_6} \right\}.$$

- For standard activation functions ϕ (logistic or tanh), the derivative $\phi'(x_i)$ can be represented in terms of $\phi(x_i)$. For example, for logistic activation,

$$\phi'(x_i) = \phi(x_i)(1-\phi(x_i)) = z_i(1-z_i).$$

In this case the hidden layer derivatives simplify to:

$$\frac{\partial E}{\partial x_3} = z_3(1-z_3)\left\{ w_{35}\frac{\partial E}{\partial x_5} + w_{36}\frac{\partial E}{\partial x_6} \right\}$$

and

$$\frac{\partial E}{\partial x_4} = z_4(1-z_4)\left\{ w_{45}\frac{\partial E}{\partial x_5} + w_{46}\frac{\partial E}{\partial x_6} \right\}.$$

- Thus we have the derivatives expressed only in terms that we have already computed.

This process can continue for multiple hidden layers, the derivatives of the error with respect to the unit inputs being computed only from the quantities computed on the forward pass and the derivatives *propagated back* from the layer below the current one.

10.4 Optimization Algorithms

10.4.1 Backpropagation and Derivatives

We can think of fitting a neural network as a general function minimization problem. Furthermore, at relatively low cost we can compute both the first and second derivatives of the error function with respect to the parameter set (the weights). The original training algorithm for neural networks was a form of steepest descent, and required only the first derivatives:

$$w(l)_{jk} \leftarrow w(l)_{jk} - \eta \frac{\partial E(\mathbf{w})}{\partial w(l)_{jk}}$$

where the η is a user specified training parameter (the *step*) of the update. This training algorithm is referred to by the same name as the derivatives computation – *back-propagation training.* Note that the weights are updated once for every pass of the whole training set. This is an example of a *batch training* algorithm.

The backpropagation algorithm has been modified in many ways. One of the earliest modifications was to add a "momentum" term, which is equivalent to exponential smoothing:

$$w(l)_{jk} \leftarrow w(l)_{jk} - \eta \left[(1-\alpha) \frac{\partial E(\mathbf{w})}{\partial w(l)_{jk}} + \alpha \left(\Delta w(l)_{jk} \right)_{\text{previous}} \right]$$

where we now have two user-defined parameters α, η, and $\left(\Delta w(l)_{jk} \right)_{\text{previous}}$ denotes the change in $w(l)_{jk}$ at the previous iteration. Thus if a large step was taken previously, the algorithm encourages a large step this time also. This is also a batch training procedure.

There is also an "online" version of backpropagation with momentum that updates the weights after the presentation of every training example:

$$w(l)_{jk} \leftarrow w(l)_{jk} - \eta' \frac{\partial E^{(i)}(\mathbf{w})}{\partial w(l)_{jk}} + \alpha' \left(\Delta w(l)_{jk} \right)_{\text{previous}} .$$

Online training only makes sense if the examples are presented in a random or unstructured order.

Three motivations for online training:

1. Biological – brains appear to learn from (almost) every experience.

2. Can converge faster than batch training. Suppose the training set contains a large number of identical examples, then even a small number of examples will provide a good approximation to the overall error and online backpropagation with a small momentum should do quite well.

3. Can be used with very large training sets that are not feasible with batch training.

4. There is a belief that introducing "noise" into the training (in this case by presenting the examples in random order) local minima in the optimization can be avoided.

In some cases we may wish to use a batch training procedure (particularly with the more general optimization methods in the next section), but are restricted because of the size of the training set. In this situation it is possible to use an "online-batch" training algorithm, which partitions the training set into chunks that are digestible by the batch training algorithm. The chunks are presented to the batch algorithm in a random order, which executes some training iterations.

There have been a number of algorithms based on backpropagation that attempt to choose the training parameters α, η in some adaptive way for each weight. Among these is the well known *Quickprop* algorithm which uses a crude line search over η for each parameter.

10.4.2 General Optimization Approach

It is very common to approach neural network training as a general optimization problem, and employ the vast knowledge of that field. The choice of algorithm is largely determined by the constraints of the data set and personal taste. For NN models with some reasonable numbers of weights (up to thousands), quasi-Newton methods work well. These methods generally require a function to compute first derivatives (for which backpropagation is suitable), and sufficient memory to store an approximation to the inverse of the Hessian ($M(M+1)/2$ where M is the number of weights). For larger problems, conjugate gradient methods or limited memory quasi-Newton methods can be used. All of these methods are tried and true, and have very efficient, readily available implementations.

In all of the above methods there remain the issues of how to choose the training parameters, and when to stop the training. Often a training algorithm addresses these problems directly (for example, early stopping specifies stopping criteria). Furthermore, for the generic optimization algorithms such as conjugate gradient and quasi-Newton there is a whole literature dealing with optimal parameter selection and stopping criteria. For backpropagation and related algorithms we are usually left with finding a suitable set of training parameters and stopping criteria ourselves. Stopping criteria normally take the form of some error threshold ("When the training error falls below this number, stop"), or a threshold on the gradient of the error function, or alternatively a restriction on the number of iterations allowed. This last criterion is more of an acknowledgement of the real-world situation in which we simply cannot keep training for more than some specific time, although some training algorithms can get stuck in a "flat" area of the training space, in which case training will continue with minute improvements more or less forever. Optimal selection of training parameters is usually done by a cross-validation scheme.

10.4.3 Example

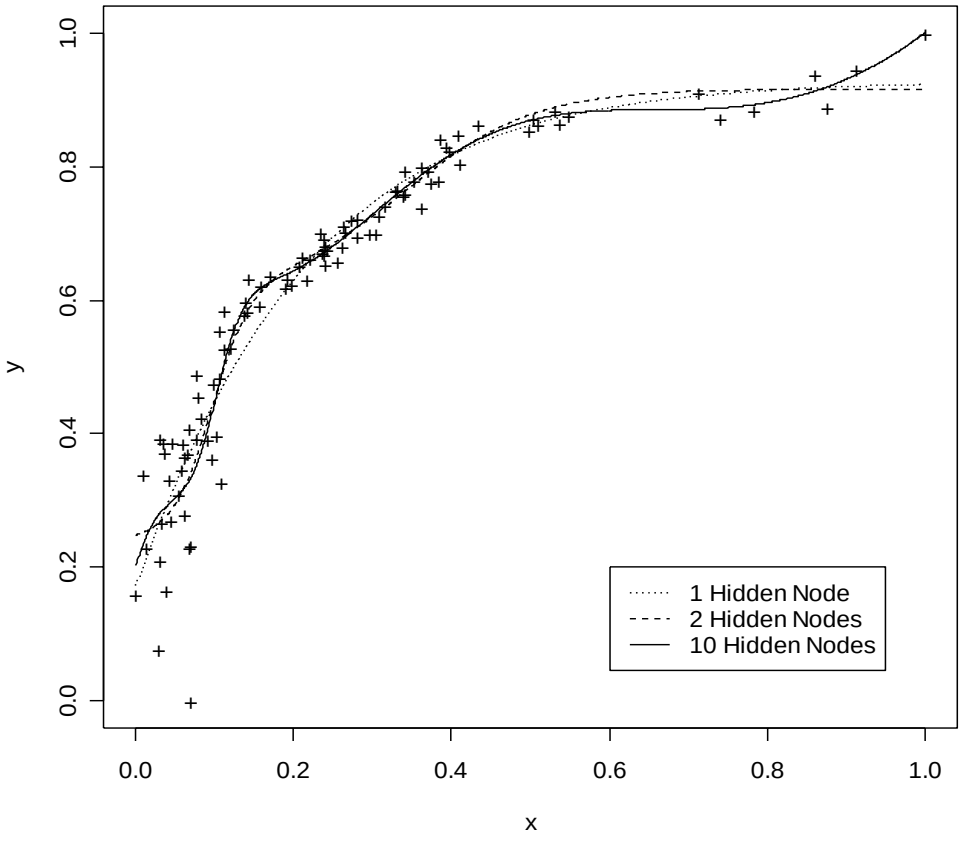
The `nnet' package in R allows fitting of single hidden layer neural network models using logistic activation functions. The models can have either logistic or linear output units, and are fit using a quasi-Newton optimization scheme. It is generally good practice to scale the data so that input and output variables are compatible with

each other. There are many scaling schemes suggested and most sensible ones will do – in our case we shall scale inputs and outputs to fit within the range $[0,1]$. Note that scaling is not required theoretically (the network should adjust its weights to suit the input and output values), but in practice training algorithms tend to get stuck in bad local minima with unscaled data. Of course, if logistic output units are used (we shall use linear in this example), then the output values, or targets, should certainly be scaled to be within $[0,1]$ or the network will produce meaningless output.

We consider an artificial dataset with a single response `y' and a single predictor `x'. This means we can plot the network function against the data and see what sort of fit is obtained. We fit networks with 1, 2 and 10 hidden nodes to show how the increase in flexibility is reflected in the fit. Here is the R code for scaling this data, fitting the networks, and producing a plot of the network functions over the data itself.

```
> # read in the data
> rdat1 <- read.table("C:\\RData\\Artificial\\RDat1.txt",header=TRUE)
>
> # scale the data set to 0-1
> minv <- apply(rdat1,2,min)
> maxv <- apply(rdat1,2,max)
> rdat1 <- sweep(rdat1,2,minv)
> rdat1 <- sweep(rdat1,2,(maxv-minv),FUN="/")
>
> # fit the networks - 1,2 and 10 hidden nodes, linear outputs
> library(nnet)
> rnn1 <- nnet(y~x,rdat1,size=1,linout=TRUE,maxit=2000,trace=FALSE)
> rnn2 <- nnet(y~x,rdat1,size=2,linout=TRUE,maxit=2000,trace=FALSE)
> rnn10 <- nnet(y~x,rdat1,size=10,linout=TRUE,maxit=2000,trace=FALSE)
>
> # fit the predicted curves for each network
> prx <- matrix((1:1000)/1000,ncol=1,dimnames=list(NULL,"x"))
>
> pry1 <- predict(rnn1,prx)
> pry2 <- predict(rnn2,prx)
> pry10 <- predict(rnn10,prx)
>
> # plot the data points and overlay the predicted curves of the nets
> plot(rdat1,pch="+")
> lines(prx,pry1,lty=3)
> lines(prx,pry2,lty=2)
> lines(prx,pry10,lty=1)
>
> legend(0.6,0.2,legend=c("1 Hidden Node","2 Hidden Nodes",
+ "10 Hidden Nodes"),lty=c(3,2,1))
```

As can be seen from the plot below, the functions produced by the neural networks become more flexible and fit the training data more closely as network size grows. Of course, with a sufficient number of hidden units, the training data can be fit perfectly, but the network function will be a very complex curve. In many cases this corresponds to *overfitting* the training data (see below).



10.4.4 Example

Let us return to the Recumbent Cow dataset introduced previously. The data may be scaled as follows:

```
> # read the data
> rcdata <- read.table("C:\\RData\\Cows\\RC Subset1.txt",header=TRUE)
>
> ## now need to scale the data to [0,1]
> minv <- apply(rcdata,2,min)
> maxv <- apply(rcdata,2,max)
> rcdata <- sweep(rcdata,2,minv,FUN="-")
> rcdata <- sweep(rcdata,2,maxv-minv,FUN="/")
```

For the sake of comparison, fit a linear model to this data and compute the SSE.

```
> # define the formula: fit AST on the other variables
> fmla <- as.formula("AST~Uketone+Outcome+Calving+CK+Unknown")
>
> # do linear regression first
> rclm <- lm(fmla,rcdata)
>
> # save the sse of the linear model
> sse <- sum(rclm$residuals^2)
> sse
[1] 3.079341
```

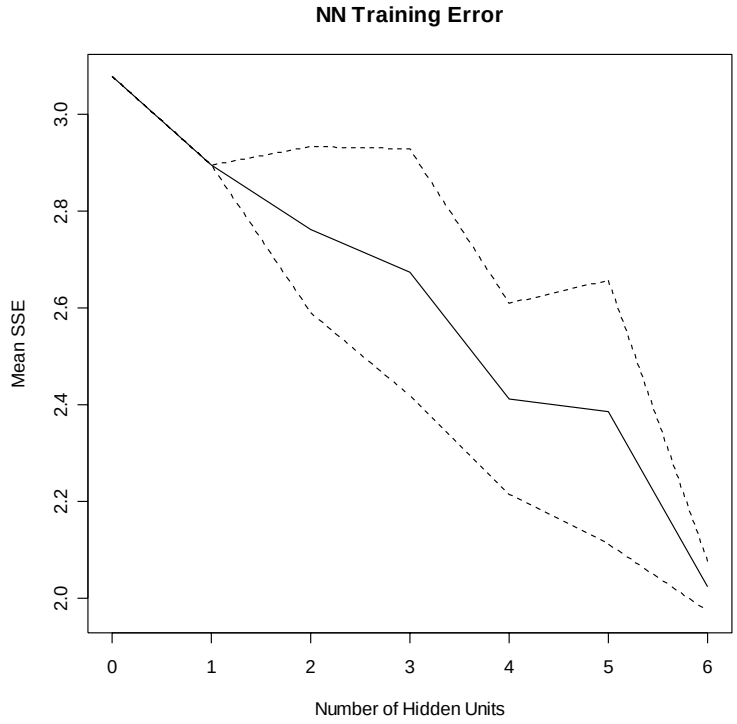
Since linear regression has a closed form solution, we will always get the same answer for this model on this data. However neural networks can converge to many different local minima, depending on the configuration of initial weights, and the parameters of the

optimization algorithm. To get an idea of both the expected SSE for different network sizes (number of hidden units), and the variation within networks of a specific size, we will fit 50 networks to each network size between 1 and 6.

```
> reps <- 50
> maxsize <- 6
>
> sdev <- 0 # no standard dev in the linear model error
>
> for ( s in 1:maxsize )
+ {
+      tmp <- NULL
+      for ( r in 1:reps ){
+            tmpnn <- nnet(fmla,data=rcdata,skip=FALSE,size=s,
+                          linout=TRUE, maxit=2000, trace=FALSE)
+            tmp <- c(tmp,sum(tmpnn$residuals^2))
+      }
+      sse <- c(sse,mean(tmp))
+      sdev <- c(sdev,sd(tmp))
+ }
```

Now we may plot the progress of the mean SSE as the network size increases. The upper and lower dotted lines correspond to a confidence interval of twice the standard error of the mean SSE. Note that the 0 hidden units entry corresponds to linear regression.

```
> plot(0:maxsize,sse,xlab="Number of Hidden Units",ylab="Mean SSE",
+ type="l", ylim=c(min(sse-2*(sdev/sqrt(reps))),
+ max(sse+2*(sdev/sqrt(reps)))),
+ main="NN Training Error")
> lines(0:maxsize,sse-2*(sdev/sqrt(reps)),lty=2)
> lines(0:maxsize,sse+2*(sdev/sqrt(reps)),lty=2)
```



The mean SSE of the networks decreases steadily as the number of hidden units increases. Note that for 3 and 4 hidden units, there is a greater variability in the SSE, which leads to a higher standard deviation of the mean. This suggests that one or more local minima appear in these architectures.

10.5 Training and Generalization Issues

There are a number of issues that need to be addressed when training NN models. For different training methods these may be resolved in different ways, although we shall attempt to outline some general principles. In particular, when we fit a NN model for some kind of predictive task (classification or regression), we are concerned that the model should *generalize* well. That is, it should perform well on unseen examples. The process of fitting or adjusting a model in such a way that it generalizes well is call *regularization.*

10.5.1 Network Architecture

The most common cause of networks not generalizing well is simply that they are heavily over-parameterized. In fact, a model with an appropriate number of parameters (generally hidden units in neural networks, although partially connected architectures can allow more hidden units) will often fit the data well using a very simple error function and training scheme. Having said this it is extremely difficult to choose the size of the parameter space correctly first time, and there is generally some kind of preliminary search before the final training occurs.

In selecting network size we can either start big and prune back, or start small and grow the network. Both approaches have their advantages. With an effective pruning (or complexity penalty) scheme, we may need to train only a single large network, whereas if we grow from a small network, we may need to train multiple networks in a sequence. However smaller networks are much cheaper to train, so there is a balance to be struck. Once again we often resort to cross-validation techniques to establish a convenient network size.

10.5.2 Overfitting

NN models can be extremely flexible, which makes them prone to overfitting. There are three general approaches to overfitting:

1. Complexity Penalty Functions. These usually take the form of some additive term in the error function, designed to penalize the training algorithm for using the available degrees of freedom. Ideally we would like the size of the penalty to balance the size of the training set error in such a way as to allow accurate fitting of the data, but to penalize heavily when overfitting begins. One of the simplest and most widely used penalty is known as *weight decay*: $\lambda \sum_{l,j,k} w(l)_{jk}^2$. Weight decay discourages the network from using large weights, and can be shown to effectively restrict the number of degrees of freedom available to the network. The selection of λ is a training issue, and is often chosen by some sort of cross-validation technique. There are more elaborate penalty techniques that attempt to send some weights to zero, which effectively eliminates that parameter.

2. Post-fitting Pruning Algorithms. The basic idea here is to train a network that you would expect to overfit the data (for example, fit a heavily over-parameterized network), and then remove some subset of the weights according to some measure of "usefulness". A common method is to check the network performance against a validation set, and remove parameters

according to criteria on the validation set error. In an overfitted model, removing some parameters may in fact decrease the validation set error, or at least not significantly increase it. Depending on the training algorithm chosen, the pruned network may be re-trained until convergence.

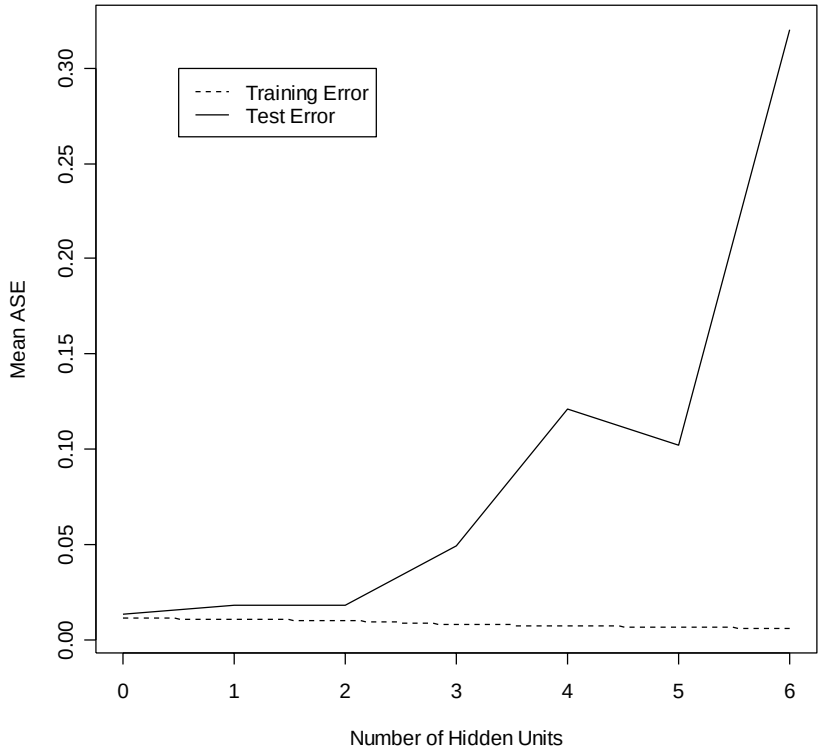
3. Overfitting Stopping Criteria. These methods are based on the idea that the "global" characteristics of a data set are generally stronger than the "local", or noise, characteristics. Thus, when training, the important characteristics are fitted first, and only when no great improvement can be obtained through modeling these characteristics does the training algorithm begin to overfit to local characteristics. The most popular algorithm of this form is called *early stopping*. Early stopping tracks the network error on a second validation set while the network is being fit to the training set. Once the validation set error begins to rise, the training is stopped under the assumption that the global characteristics are now fitted. There are many assumptions in this method, not the least of which is that the validation set error will necessarily follow the expected curve. In practice the validation set error can sometimes have multiple humps and a method that ceases training at the first valley may miss the lowest validation set error. This can be combated by allowing the training to continue to convergence, but recording the validation set error at fixed intervals and selecting the point in the training curve which yields the lowest validation set error.

10.5.3 Example

To show how neural network models can overfit, we repeated the example above, but this time set aside 1/3 of the data as an independent test set. The errors reported here are averages over 50 networks trained with the specified number of hidden units. Both the average error on the training set and the average error on the independent test set are reported for each network size. Note that the ASE is reported in this case, which is equal to the SSE divided by the number of observations in the respective data set.

```
> nobs <- dim(rcdata)[1]
> trainidx <- sample(1:nobs,2*nobs/3,replace=FALSE)
> testidx <- (1:nobs)[-trainidx]
> trndat <- rcdata[trainidx,]
> tstdat <- rcdata[testidx,]
> tsttarg <- tstdat[,"AST"]
>
> # do linear regression first
> rclm <- lm(fmla,trndat)
>
> # save the sse of the linear model prediction on the test set
> psse <- sum((predict(rclm,tstdat) - tsttarg)^2)
> sse <- sum(rclm$residuals^2)
>
> # now train some neural networks
>
> for ( s in 1:6 )
+ {
+      ptmp <- NULL
+      tmp <- NULL
+      for ( r in 1:50 ){
+             tmpnn <- nnet(fmla,data=trndat,size=s,linout=T,
+                    maxit=3000,trace=F)
+             pred <- predict(tmpnn,tstdat)
+             ptmp <- c(ptmp,sum((pred-tsttarg)^2))
+             tmp <- c(tmp,sum(tmpnn$residuals^2))
+      }
```

```
+      sse <- c(sse,mean(tmp))
+      psse <- c(psse,mean(ptmp))
+ }
>
> plot(0:maxsize,psse,xlab="Number of Hidden Units", ylab="Mean ASE",
+  type="l", ylim=c(min(sse,psse),max(sse,psse)))
> lines(0:maxsize,sse,lty=2)
> legend(0.5,0.3,c("Training Error","Test Error"),lty=c(2,1))
>
```

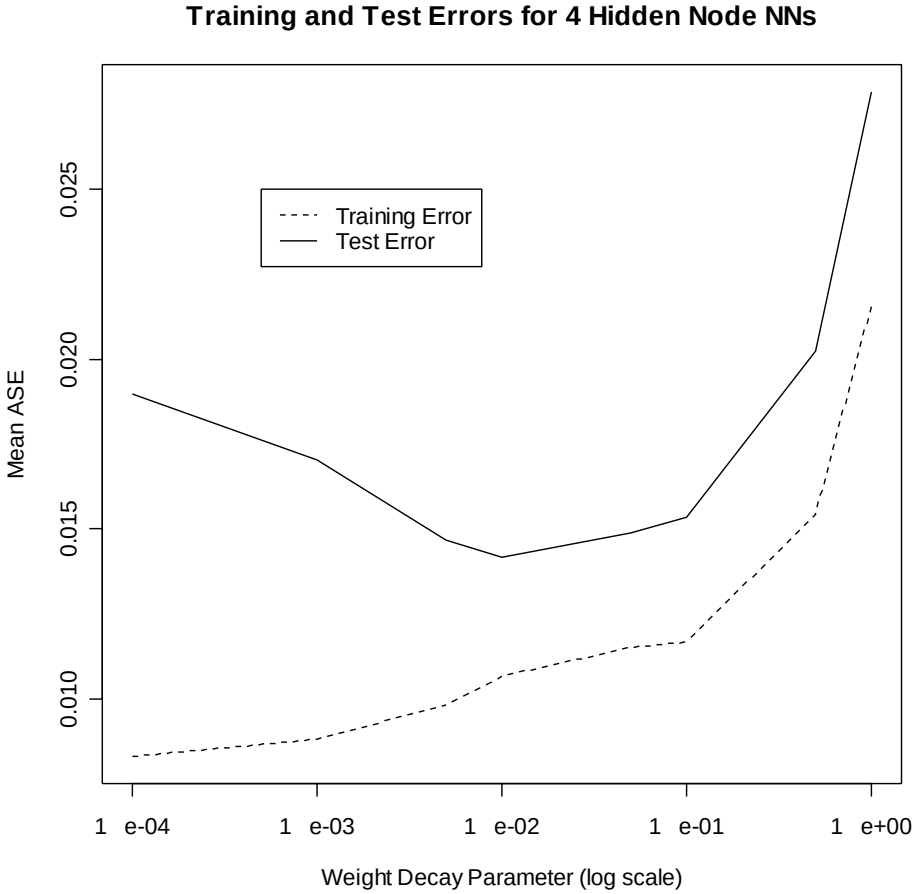


The learning set error decreases steadily as we observed before, however the test set error also increases dramatically as the size of the network increases. This is an example of flexible models overfitting the training data. In fact, the best model in terms of test set error is the linear model, which also has the highest training set error.

The weight decay option in nnet can regularize overfitting. As an example, we fit 4 hidden unit neural networks to the training data above using different values for the weight decay parameter `decay'.

```
> sse <- NULL
> psse <- NULL
>
> for ( d in c(0.0001,0.001,0.005,0.01,0.05,0.1,0.5,1) )
+ {
+      ptmp <- NULL
+      tmp <- NULL
+      for ( r in 1:50 ){
+            tmpnn <-nnet(fmla, data=rcdata[trainidx,],skip=FALSE,
+                         size=4, decay=d, linout=TRUE,
+                          maxit=3000, trace=FALSE)
+            pred <- predict(tmpnn,rcdata[testidx,])
+            ptmp <- c(ptmp,sum((pred-rcdata[testidx,"AST"])^2))
+            tmp <- c(tmp,sum(tmpnn$residuals^2))
+      }
+      sse <- c(sse,mean(tmp))
```

```
+      psse <- c(psse,mean(ptmp))
+ }
> sse <- sse/length(trainidx)
> psse <- psse/length(testidx)
>
> plot(wd,psse,log="x",type="l",ylim=range(c(psse,sse)),
+ ylab="Mean ASE", xlab="Weight Decay Parameter (log scale)",
+ main="Training and Test Errors for 4 Hidden Node NNs")
> lines(wd,sse,lty=2)
> legend(5e-4,0.025,c("Training Error","Test Error"),lty=c(2,1))
```



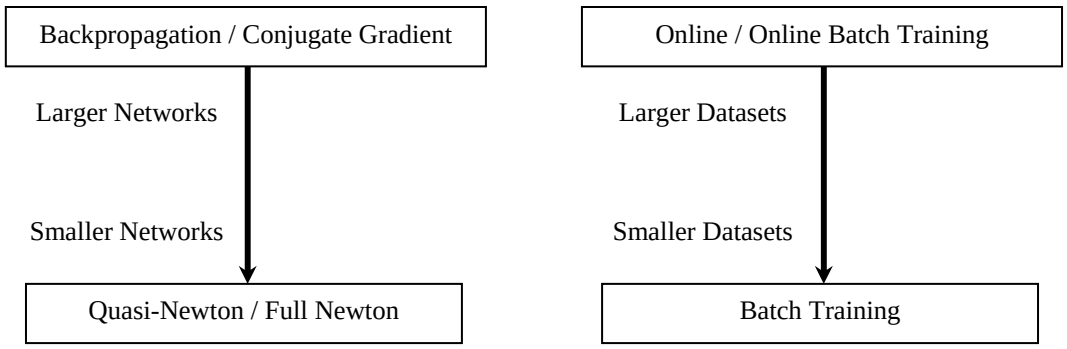
The training set error steadily increases as the weight decay parameter increases, indicating that the penalty reduces the accuracy of the training set fit. Hopefully this signifies a reduction in over-fitting to the training set. The test set error demonstrates that this is true, for the smaller decay parameter values. The weight decay penalty improves the generalization ability of the networks until it reaches an optimal size of about 0.01. As the penalty increases beyond this value the fit becomes increasingly bad in terms of both training error and test error.

10.6 Memory Management Issues

Before applying a particular neural network training algorithm to a dataset, we need to consider the memory management aspect of the problem. In particular, standard batch training algorithms are likely to pose a problem for data mining applications because they presume the entire dataset resides in memory. If this is not possible, then every iteration will require multiple memory swaps and will slow the training down considerably. The solution in this case is to use some sort of online training, or a mixed batch/online method that restricts batch sizes to that which the local machine can handle.

Some online methods use a "see-once" methodology which adjusts the weights at the presentation of every example, but presumes that an example will not be seen again. This can be useful if the training set is a data stream that is not, or cannot be stored in its entirety.

Optimization algorithms also have memory requirements based on the number of network parameters. For example, if the number of weights in the network is W, then the simpler algorithms (backpropagation, conjugate gradients) require only that the first derivatives of the error function are stored, which corresponds to W floating point numbers. However the quasi-Newton and full Newton methods compute and store the (inverse) Hessian matrix (second derivatives) of the error function, which is $W(W+1)/2$ floating point numbers. Note that we don't need to store all W^2 second derivatives because the Hessian is symmetric. For a large network this requirement may become prohibitive in terms of machine RAM. In some cases, even a network with a small number of hidden nodes will have a large number of weights simply because the input/output dimension is so large. Either way, we need to consider these issues when choosing an optimization algorithm.



The diagram gives a rough summary of the training possibilities under different conditions. With very large networks often the only (computationally) feasible optimization algorithms are the simplest – backpropagation, conjugate gradients. As previously mentioned, there are also limited memory implementations of more sophisticated algorithms (like quasi-Newton) that may also be appropriate in these cases. As the network size decreases, the more powerful quasi-Newton and full-Newton training methods become available to us.

Similarly, with very large datasets that we can't possibly hope to hold in RAM, online or online batch training algorithms are usually appropriate. As the dataset size shrinks, full batch training is possible, which tends to have better defined convergence properties.

11 Comparison of CART and Neural Networks

11.1 Introduction

The purpose of this chapter is to consider some of the benefits and drawbacks of two very popular data mining algorithms, CART and Neural Networks. This is *not* intended to be an exhaustive study, nor is it intended to recommend one method over the other, rather to give the student some concept of comparing methods for a particular task.

11.2 Comparison Method

CART models are always fitted using the R function `rpart'. A tree of maximal depth is first fitted to the entire data set. Then the optimal complexity parameter is chosen by running the printcp command on the maximal tree and applying the 1-SE rule. Note that rpart uses 10-fold cross validation on the entire dataset to create the cp table.

Neural Network models are fitted using the `nnet' package in R. The networks all have 10 hidden nodes and are fitted using weight decay. Since all the datasets represent classification problems, the networks have sigmoidal output nodes and use entropy fitting. The choice of 10 hidden nodes is admittedly somewhat arbitrary – it is a number large enough to allow sufficient flexibility for all the problems considered, but usually so large that unregularized fitting would lead to overfitting. The training time could be reduced somewhat (and possibly the accuracy increased) by investing further computations into a cross-validated choice of the number of hidden units. The weight decay parameter is one of (0, 0.0001, 0.001, 0.01, 0.1), and the selection is made by 10-fold cross validation on the entire dataset.

Once we have selected the regularization parameters for rpart and nnet (the cp and weight decay parameters respectively), we then use those parameters to compare the methods. The comparison uses a 10-fold cross validation on the whole dataset. That is, for each $1/10^{th}$ slice of the data, a single rpart tree and a single nnet network are fitted to the remaining $9/10^{th}$ of the data using the optimal parameters chosen above. The two models then predict on the separated $1/10^{th}$ and the number of mismatches is recorded. At the conclusion of the cross-validation procedure, the 10 mismatch counts are summed up and divided by the number of observations to give an overall error rate.

We note at the outset that this is not really an "honest" usage of the data, since the regularization parameters were estimated from the same data used to compare the methods. Ideally we should use independent datasets to estimate the parameters and compare the methods.

Neural network algorithms do not generally cope with NAs in the training or test data. For this reason, NA observations are removed from the training set when fitting nnet models (although rpart is still allowed to use the observations with NA entries for its own fit). Observations with NA entries in the test sets are removed for both the nnet and rpart models before giving a mismatch count. One of the datasets has a large number of observations with NA entries – we shall make further comment on this in the next section.

We are interested in 4 statistics from each of the two methods. The first three are computation time measurements – namely, the time taken to select the optimal regularization parameter, the time taken to fit the models, and the time taken to predict on the test data. The dimensions of the reported times are not important; they are only

intended to give a comparison between the two methods in terms of computational requirements. The final statistic is the overall mismatch rate of the two methods.

11.3 Datasets & Results

We evaluated rpart and nnet on 4 classification datasets. The datasets were all comparatively small, and, with the exception of the now familiar recumbent cow dataset, complete. All non-binary input features were scaled to fall between 0 and 1 so as to be suitable for nnet training. We shall make brief comments on each dataset and the corresponding results, and present the overall results as a table at the end of the section.

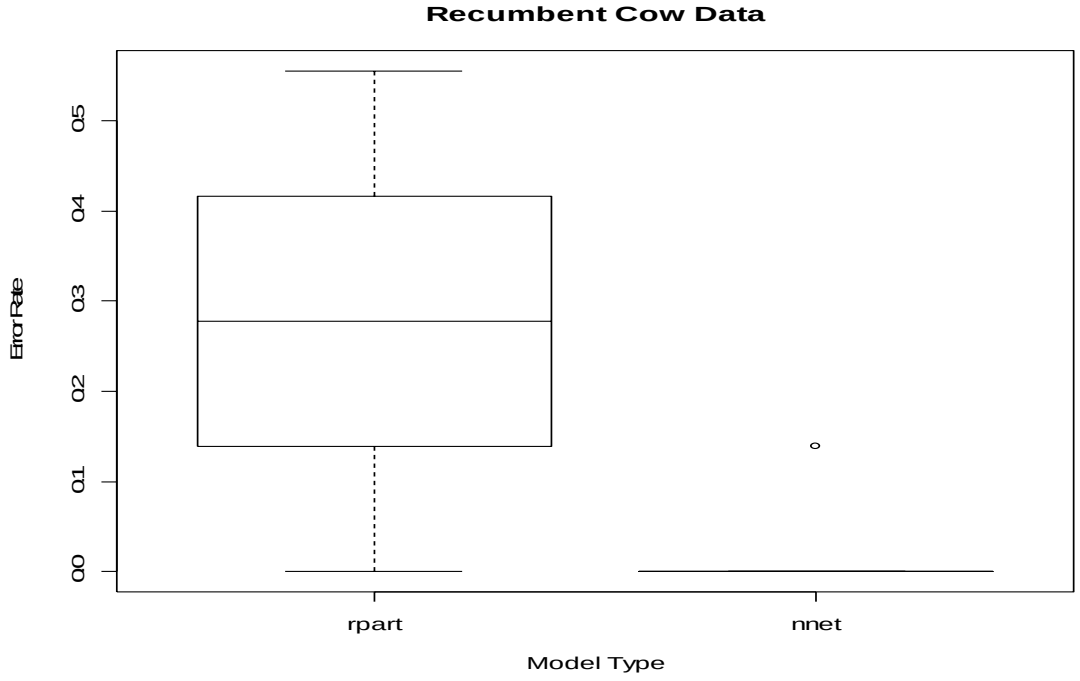
11.3.1 Recumbent Cows Data

We have encountered the Recumbent Cow dataset a number of times in the course so far. Recall that the task is to predict if a recumbent (or lying down) cow is expected to live or die based on a number of physiological measurements.

Of the datasets considered, only the Recumbent Cow dataset has observations with NA entries, but in this case the problem is quite extreme (372 observations out of 435 have at least one NA entries). Despite this, the rpart models are still able to make predictions. This is a significant advantage that CART holds over many other classification algorithms (in particular, neural networks). In fact, the rpart model achieved an overall misclassification rate on *all* the data (including observations with NA entries) only slightly worse than it did on the "cleaned" data reported below (0.22 as opposed to 0.16). Therefore, the standard rpart model is able to make meaningful predictions on 372 observations that the nnet model could not process without a substantial data cleaning exercise.

The comparison between rpart and nnet yields three major points of difference from this dataset. The first is the NA issue discussed above, on which rpart clearly holds an advantage of nnet. The second is the parameter selection / training and prediction times. Once again, rpart requires far less computation time for parameter selection and training than nnet, although the prediction times are comparable. We note at this juncture that the rpart function in R contains built in (presumably optimized) code for parameter selection using cross validation. For the nnet models, on the other hand, we had to settle for far less efficient code. Nevertheless, the training times (with the optimal parameters) highly favor rpart. The last point of comparison is the error rates themselves (on the data with NA observations omitted). Neural network models establish themselves here as significantly superior to the decision trees on this dataset, with no prediction errors at all for the cross-validation test sets.

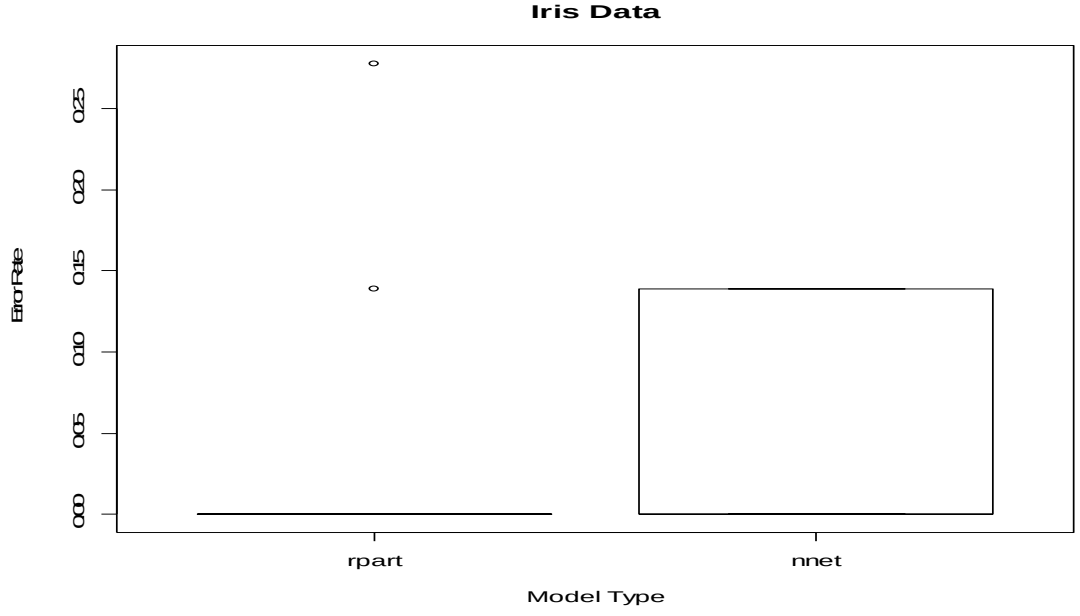
The boxplot below gives a somewhat uninformative comparision of the error rate distribution over the 10 cross-validation test sets because the error rate for neural networks, except in one case, is equal to 0.



11.3.2 Iris Data

The iris data set is one of the datasets included in the R distribution. The task here is to classify an iris flower as one of three different species based on 4 measurements. It is something of a classic classification dataset.

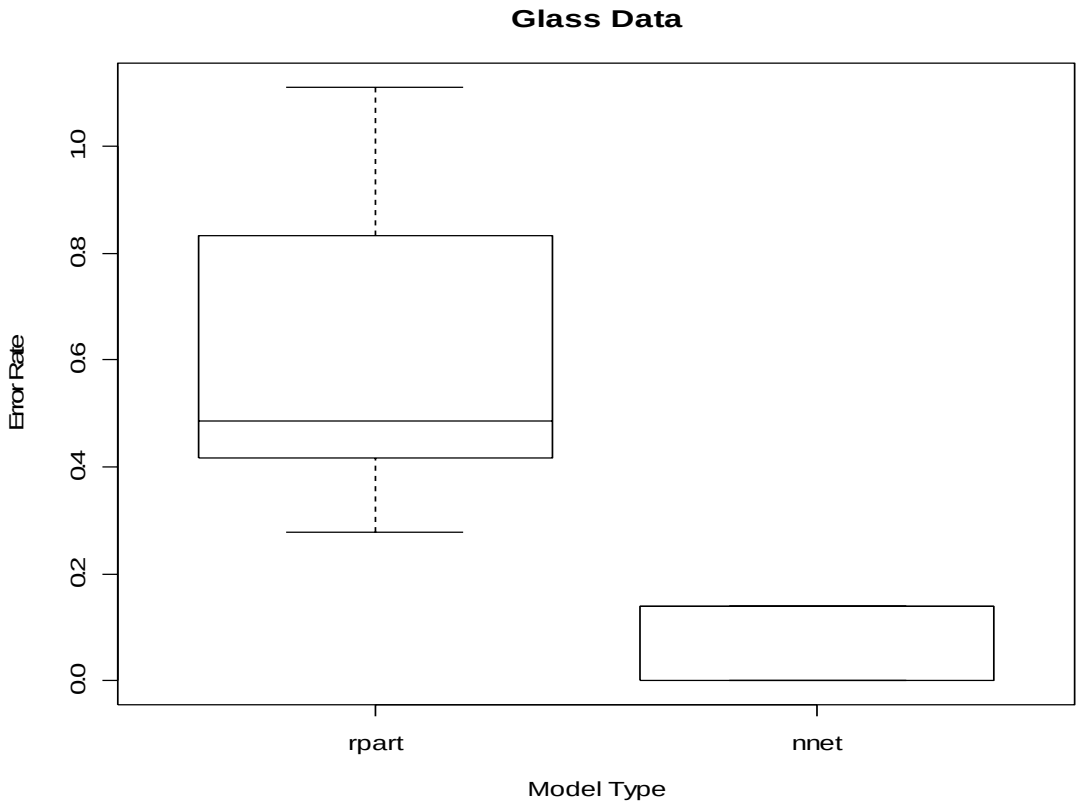
The comparison between rpart and nnet yields virtually the same results as for the Recumbent Cow dataset in terms of training time. That is, the parameter selection and training times for nnet are much greater than for rpart. But in this instance the overall error rate is excellent for both methods, although slightly lower for nnet. From the boxplot below we see that in fact both models make errors in only one of the 10 CV test sets.



11.3.3 Glass Data

The glass dataset has observations of glass fragments from 7 different types of glass. For each fragment some of the chemical properties of the glass are recorded, along with the refractive index. The task is to determine the type of the glass based on these properties.

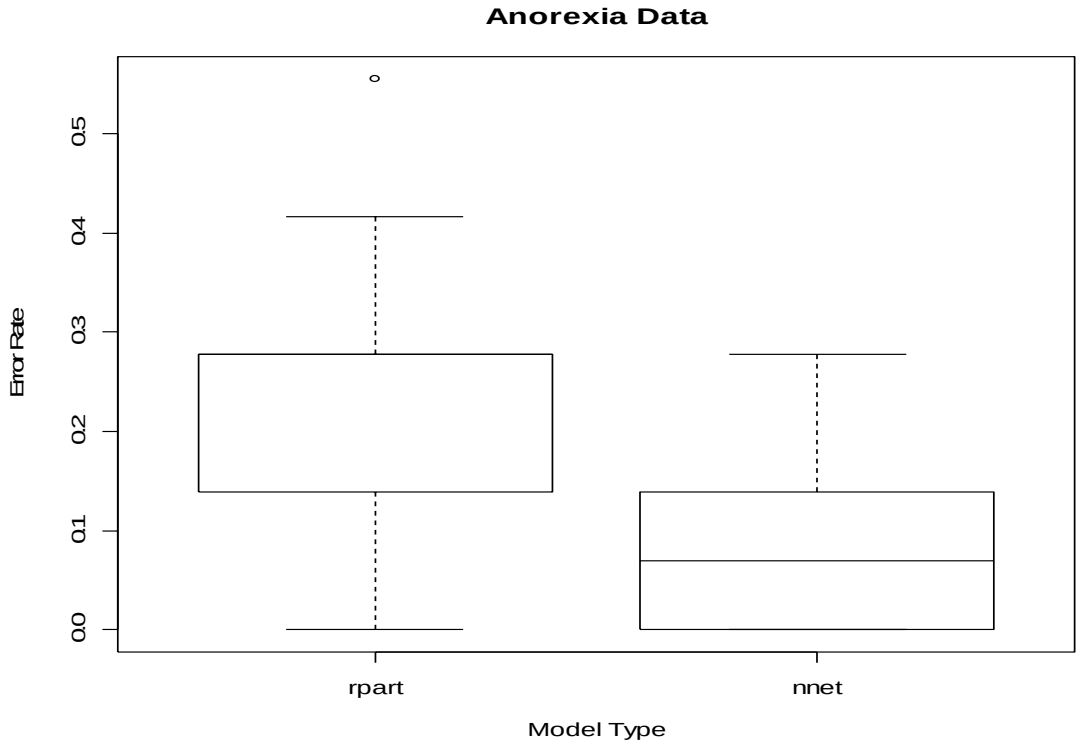
The comparison again reflects the results on the previous two datasets. In this case the difference in parameter selection time and training time is particularly extreme. In this case the nnet models yield a much lower error rate (about 20% of the rpart error rate).



11.3.4 Anorexia Data

The classification task in the anorexia dataset is to identify the type of treatment anorexic girls underwent based on the pre-treatment and post-treatment weight of the girl. In this case we need to be careful with scaling the numeric variables (pre and post treatment weight), since the relative difference between the two is significant. In this instance we simply computed the maximum and minimum weights to be the maximum and minimum over the two variables, and then scaled as normal.

The results tell the same story as the other three datasets – a large benefit in the error rate of nnet models at the cost of much longer training and parameter selection times.

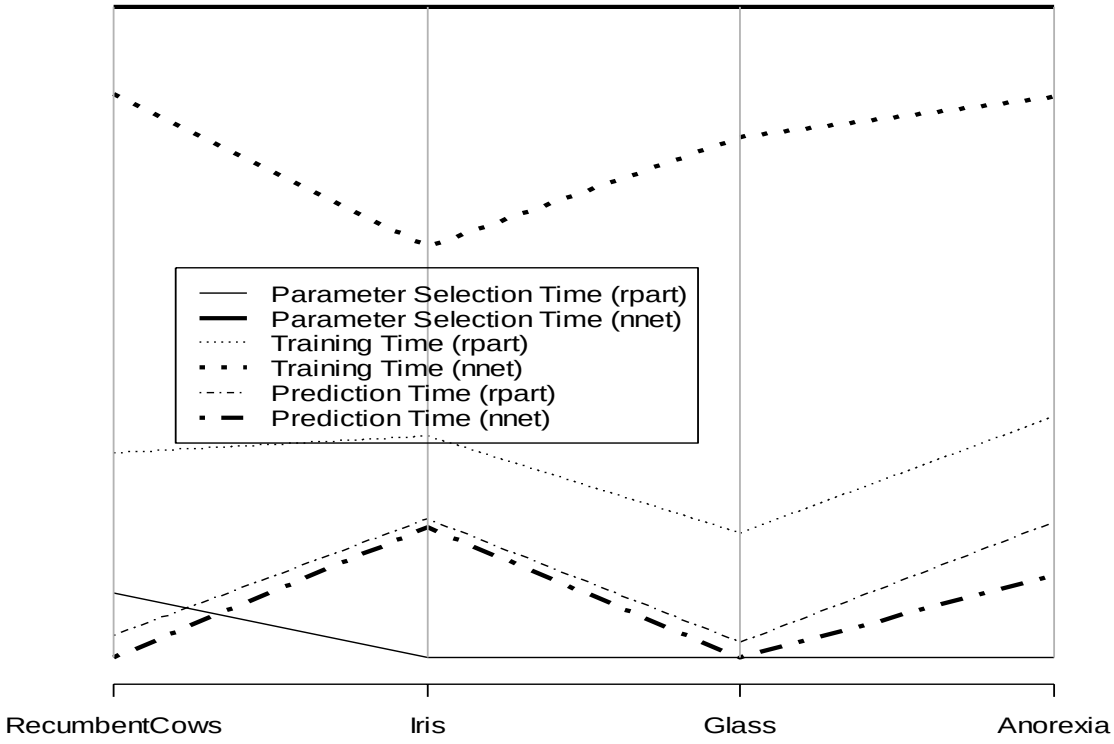


11.3.5 Results and Conclusions

The results for the four data sets are summarized in the following table:

	RecumbentCows	Iris	Glass	Anorexia
rpart par. sel. time	0.17	0.02	0.10	0.03
nnet par. sel. time	21.45	36.66	154.45	22.62
rpart training time	0.54	0.26	0.41	0.35
nnet training time	10.44	2.31	35.36	9.02
rpart prediction time	0.12	0.10	0.12	0.12
nnet prediction time	0.10	0.09	0.10	0.07
rpart error rate	0.29	0.02	0.20	0.26
nnet error rate	0.02	0.02	0.03	0.08

A more pictorial comparison of the 3 different time measurements can be obtained via a parallel coordinates plot. In this case we use a log scale to get a more meaningful plot. The parameter selection time for nnet is always the most significant expense in terms of computation time, whereas the corresponding operation in rpart comes at very little cost. We note again that the parameter selection is part of the inbuilt (optimized) rpart routine, whereas the same operation was implemented in nnet using the R commands detailed below. Nevertheless, it is clear that the cost of parameter selection highly favors the rpart algorithm. The training time, given the optimal parameter, also heavily favors the rpart algorithm. It is possible that the nnet training time could be reduced by a more careful selection of network size, although this would add further to the parameter selection time. Finally there doesn't appear to be a significant difference in prediction time between the two models.



This comparison between the rpart algorithm and the nnet algorithm is very consistent across the four datasets considered. We may summarize the conclusions as follows:

- rpart enjoys a significant advantage over nnet in that observations with NA entries are dealt with automatically as part of the algorithm.
- nnet requires significantly more computation time to select parameters and train models than rpart on all the datasets.
- The cross-validation error rate for nnet models is smaller on all datasets than the comparable rpart models.

11.4 R Code

The R code for implementing this comparison is included below. In theory this code can be easily extended to any number of datasets simply by including the extra datasets in the top section (together with scaling and formula name), and then adding the new datasets to the `dlist' dataset list.

```
###### Comparison of CART and Neural Networks (classification)

library(rpart)
library(nnet)

# use this function to scale data to 0-1
scale01 <- function(dat)
{
       x <- dat
       minv <- apply(x,2,min,na.rm=TRUE)
       maxv <- apply(x,2,max,na.rm=TRUE)
       x <- sweep(x,2,minv,'-')
       sweep(x,2,maxv-minv,'/')
}
```

```
############## Datasets to analyse ##################################################

### Recumbent Cow Data

rcdata <- read.table("C:\\RData\\Cows\\Recumbent Cows.txt",header=TRUE)
rcdata[,"Outcome"] <- as.factor(rcdata[,"Outcome"])
levels(rcdata[,"Outcome"]) <- c("Dead","Living")
targidx <- charmatch("Outcome",names(rcdata))
delidx <- charmatch("CaseId",names(rcdata))  # want to remove this variable
rcfmla <- as.formula(paste("Outcome~",paste(names(rcdata)[-c(targidx,delidx)],
 collapse="+")))

# scale inputs to 0-1
predictors <- attr(terms(rcfmla),"term.labels")
rcdata[,predictors] <- scale01(rcdata[,predictors])

### Iris data
data(iris)
targidx <- charmatch("Species",names(iris))
irisfmla <- as.formula(paste("Species~",paste(names(iris)[-targidx],collapse="+")))

# scale inputs to 0-1
predictors <- attr(terms(irisfmla),"term.labels")
iris[,predictors] <- scale01(iris[,predictors])

### Glass data
library(mda)
data(glass)
targidx <- charmatch("Type",names(glass))
glass[,targidx] <- as.factor(glass[,targidx])
glassfmla <- as.formula(paste("Type~",paste(names(glass)[-targidx],collapse="+")))

# scale inputs to 0-1
predictors <- attr(terms(glassfmla),"term.labels")
glass[,predictors] <- scale01(glass[,predictors])

### Anorexia data
library(MASS)
data(anorexia)
anorfmla <- as.formula("Treat~Prewt+Postwt")

# scale weights to 0-1 (need to scale both columns on the same scale)
maxwt <- max(anorexia[,-1])
minwt <- min(anorexia[,-1])
anorexia[,-1] <- (anorexia[,-1] - minwt)/(maxwt-minwt)

## data list
dlist <- list()
dlist$RecumbentCows <- list(data=rcdata,target="Outcome",formula=rcfmla,
                            name="RecumbentCows")
dlist$Iris <- list(data=iris,target="Species",formula=irisfmla,name="Iris")
dlist$Glass <- list(data=glass,target="Type",formula=glassfmla,name="Glass")
dlist$Anorexia <- list(data=anorexia,target="Treat",formula=anorfmla,name="Anorexia")

## results list
results <- list()
length(results) <- length(dlist)

# Number of cross validation steps we use throughout
cv <- 10

################### loop over all the datasets ########################################

for ( dlidx in 1:length(dlist) ) {

dl <- dlist[[dlidx]]
data <- dl$data
target <- dl$target
fmla <- dl$formula
name <- dl$name

#### 1 determine model parameters:

## CART

res.pst.rpart <- system.time(NULL)    # parameter selection time

ctrl <- rpart.control(minbucket=1,minsplit=2,cp=0,xv=cv) # full tree

res.pst.rpart <- res.pst.rpart + system.time(
 rctree <- rpart(fmla,data,control=ctrl)
)
```

```
# find the optimal complexity parameter according to the SE1 rule
cpmat <- rctree$cptable
xerr <- cpmat[,"xerror"]
se1rule <- min(xerr) + cpmat[order(xerr)[1],"xstd"]

cartcp <- max(cpmat[xerr <= se1rule,"CP"])   # optimal rpart complexity parameter

## NN

res.pst.nnet <- system.time(NULL)    # parameter selection time

# fit 10 hidden node networks

sz <- 10

obs <- sample(dim(data)[1],dim(data)[1])
nobs <- length(obs)

decay <- c(0,0.0001,0.001,0.01,0.1)
res <- NULL

for ( d in decay)
{

tmpres <- 0
for ( i in 1:cv )
{
       tstidx <- obs[((((i-1)*nobs)/cv)+1):(((i+1)*nobs)/cv)]      # choose test set
       tridx <- obs[-tstidx]                                       # training set
       res.pst.nnet <- res.pst.nnet + system.time(
        tmpnet <- nnet(fmla,data[tridx,],size=sz,na.action=na.omit,decay=d,
                      maxit=1000, trace=F)                         # train
       ) + system.time(
        pred <- predict(tmpnet,na.omit(data[tstidx,]),type="class") # predict
       )
       tmpres <- tmpres + sum(na.omit(data[tstidx,])[,target]!=pred)
}

res <- c(res,tmpres/cv)
}

nndecay <- decay[order(res)[1]]      # candidate for optimal weight decay parameter


#### now compare the two models using cv-fold cross validation

# re-mix the input patterns
obs <- sample(dim(data)[1],dim(data)[1])
nobs <- length(obs)

res.rpart <- NULL
res.nnet <- NULL
res.rpart.x <- NULL

res.tt.rpart <- system.time(NULL)
res.pt.rpart <- system.time(NULL)
res.tt.nnet <- system.time(NULL)
res.pt.nnet <- system.time(NULL)

# create a new control vector with no cross-validations & then use optimal cp
ctrl2 <- rpart.control(minbucket=1,minsplit=2,cp=0,xv=0) # full tree

for ( i in 1:cv )
{
       # get the next CV section
       tstidx <- obs[((((i-1)*nobs)/cv)+1):((i*nobs)/cv)]
       trnidx <- obs[-tstidx]
       trndat <- data[trnidx,]
       tstdat <- data[tstidx,]

       # training
       res.tt.rpart <- res.tt.rpart + system.time(
        tmptree <- prune.rpart(rpart(fmla,trndat,control=ctrl2,method="class"),cp=cartcp)
       )
       res.tt.nnet <- res.tt.nnet + system.time(
        tmpnet <- nnet(fmla,trndat,size=sz,na.action=na.omit,decay=nndecay,
                      maxit=1000,trace=F)
       )

       # prediction
       res.pt.rpart <- res.pt.rpart + system.time(
        predcart <- predict(tmptree,na.omit(tstdat),type="class")
       )
       res.pt.nnet <- res.pt.nnet + system.time(
```

```
         prednn <- predict(tmpnet,na.omit(tstdat),type="class")
        )

        # write the number of mismatches
        res.nnet <- c(res.nnet,sum(na.omit(tstdat)[,target]!=prednn))
        res.rpart <- c(res.rpart,sum(na.omit(tstdat)[,target]!=predcart))

        # let rpart predict on the whole test set (including na's)
        res.rpart.x <- c(res.rpart.x,
                sum(tstdat[,target]!=predict(tmptree,tstdat,type="class"))
        )
}

results[[dlidx]] <- list(rpart=list(pst=res.pst.rpart,traintime=res.tt.rpart,
                                predtime=res.pt.rpart, err=res.rpart,errx=res.rpart.x),
                         nnet=list(pst=res.pst.nnet,traintime=res.tt.nnet,
                                predtime=res.pt.nnet,err=res.nnet))
}

names(results) <- names(dlist)

######################## generate a nice table ########################################

resmat <- matrix(ncol=length(results),nrow=8)
dimnames(resmat) <- list(c("rpart par. sel. time","nnet  par. sel. time",
                        "rpart training time", "nnet  training time",
                        "rpart prediction time","nnet  prediction time",
                        "rpart error rate","nnet  error rate"), names(results))

for ( i in 1:length(results) )
{
        res <- results[[i]]
        dat <- dlist[[i]]
        nobs <- dim(na.omit(dat$data))[1]
        resmat[1,i] <- res$rpart$pst[3]; resmat[2,i] <- res$nnet$pst[3]
        resmat[3,i] <- res$rpart$traintime[3]; resmat[4,i] <- res$nnet$traintime[3]
        resmat[5,i] <- res$rpart$predtime[3]; resmat[6,i] <- res$nnet$predtime[3]
        resmat[7,i] <- sum(res$rpart$err)/nobs; resmat[8,i] <- sum(res$nnet$err)/nobs
}
```

12 Nonparametric Regression: Other Techniques

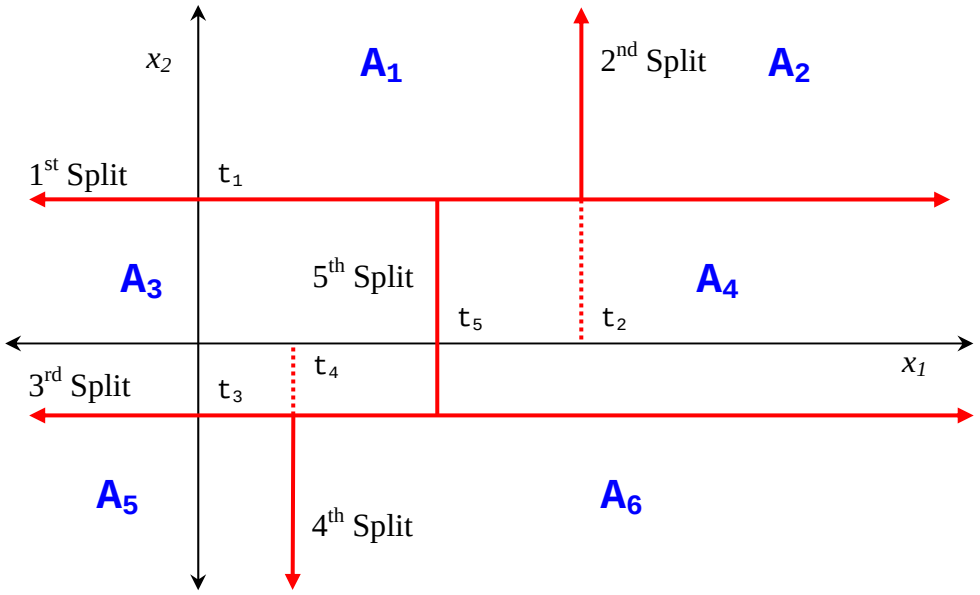
12.1 MARS

([WI:1998] 5.2.3; [R:1996] 4.1; [F:1991])

12.1.1 Motivation From CART

MARS (Multiple Additive Regression Splines) is an advanced and effective nonlinear statistical method for regression problems. It is derived from a natural generalization of the CART methodology to better suit the requirements of nonparametric regression problems. To motivate the method, let us return briefly to CART and propose an alternative representation.

Recall that a CART model (with 2-D feature space) generates a partition of the feature space similar to the following:



In this case the model has 5 splits, two on the variable x_2, and three on the variable x_1, resulting in 6 different classification spaces. In the case that CART is used for a regression problem, each of these "classes" corresponds to a single regression value allotted to each observation that falls in that area. Thus, when used for regression, CART models approximate the function by a series of hyperplanes at orthogonal orientations to the axes.

Recall that the *Heaviside Step Function,* $H(z)$, is 0 when z is negative, and 1 when z is positive. Consider the first split, which cuts the feature space in two around the line $x_2 - t_1 = 0$. For a random observation $\mathbf{x}$ this split can be represented by the two functions $H[(x_2 - t_1)]$ and $H[-(x_2 - t_1)]$. If the first is 1, this indicates the observation belongs to the top half of the feature space, that is, in groups A_1 or A_2. If the second function is 1, the observation belongs in the lower half of the space and is in one of the remaining groups.

The second split only affects observations in the top half of the space, and can be represented by the two functions $H[(x_1 - t_2)]$ and

$H[-(x_1 - t_2)]$. If the first function is 1, then the observation belongs to group $\mathbf{A_2}$, and if the second is 1, the observation is classified as group $\mathbf{A_1}$. Thus the function

$$\mathbf{A}_1 H[(x_2 - t_1)]H[-(x_1 - t_2)]$$

will be $\mathbf{A_1}$ if the observation is to be classified as $\mathbf{A_1}$, and 0 otherwise. Similarly

$$\mathbf{A}_2 H[(x_2 - t_1)]H[(x_1 - t_2)]$$

gives the correct classification of observations into class $\mathbf{A_2}$. Thus the function

$$\mathbf{A}_1 H[(x_2 - t_1)]H[-(x_1 - t_2)] + \mathbf{A}_2 H[(x_2 - t_1)]H[(x_1 - t_2)]$$

is an algebraic representation of the full branch of the CART tree corresponding to the top half of the feature space. Clearly we can build a similar representation for the lower half of the space, and generate a complete algebraic representation of the tree of the form:

$$f(\mathbf{x}) = \sum_{c=1}^{6} \mathbf{A}_c \prod_{k(c)} H[s_{k(c)}(x_{k(c)} - t_{k(c)})]$$

where $k(c)$ indexes the splits relevant to classifying class $\mathbf{A}_c$, $s_{k(c)}$ is the sign of the required split function, $x_{k(c)}$ is the feature on which the split is made, and $t_{k(c)}$ is the split threshold.

There are two major deficiencies with this function with regards to regression problems (in which the "classes" are actually predicted values). As previously mentioned, the function is not continuous, which means that many splits will be required to approximate general, smooth surfaces. Also the function is not even suitable for the simplest surface, the affine line (or affine hyperplane in higher dimensions) because all splits are orthogonal to feature axes.

12.1.2 The MARS Approach

MARS generalizes the above CART formula by using component (or *basis*) functions that are more general than the heaviside step function. In particular, the MARS model uses the *order 1 truncated power spline* functions $\tau(\pm(x-t)) = [\pm(x-t)]_+$, where t is the *split point* and $[\bullet]_+$ is the operator taking the positive part of the contents of the bracket, and returning 0 when the argument is negative.

The MARS fitting algorithm builds up a set of basis functions $\{B_1(\mathbf{x}), \ldots, B_M(\mathbf{x})\}$ used to fit a function to the data of the form

$$f(\mathbf{x}) = \sum_{m=1}^{M} \alpha_m B_m(\mathbf{x})$$

where the α_m are real valued coefficients. The basis functions are analogous to the products of heaviside functions in the CART functions. Once we have a set of basis functions, then fitting the MARS model becomes a simple multiple regression problem in order to compute the coefficients $\alpha_1, \ldots, \alpha_M$. We may summarize the MARS algorithm for building an optimal set of basis functions as follows:

- The algorithm begins with a single basis function $B_1(\mathbf{x}) = 1$.

- At the M^{th} iteration we have $2M-1$ basis functions $B_1(\mathbf{x}),\ldots,B_{2M-1}(\mathbf{x})$ and the current model is $f_M(\mathbf{x}) = \sum_{i=1}^{2M-1}\alpha_i B_i(\mathbf{x})$ where the α_i are real-valued coefficients.
- At the $M+1^{th}$ iteration MARS seeks to add two new basis functions of the form $B_m(\mathbf{x})[+(x_\nu - t)]_+$ and $B_m(\mathbf{x})[-(x_\nu - t)]_+$ where x_ν is one of the features of the input vector $\mathbf{x}$ and t is an observation (or realization) of x_ν taken from the training data.

The iterative step involves three nested loops, the outer one loops over the $2M-1$ basis functions (variable m), the middle loops runs over all the features of $\mathbf{x}$ that are not yet represented in the current basis function (variable ν), and the inner loop runs over all the values of the current feature in the training set (variable t). At each 3-tuple (m,ν,t) we construct the two new candidate basis functions as above, and optimize all the model coefficients over the model including the new candidates. At the conclusion of all three nested loops, the candidate pair offering the greatest improvement in the fit of the model is admitted to the basis. The process terminates when the basis reaches some maximal size $M_{\max}$. Note that the MARS algorithm does *not* allow a basis function to contain multiple contributions from a single feature (analogous to multiple splits on the same variable in a branch of a CART-based decision tree). The reason for this prohibition is beyond the scope of this treatment, but is basically to do with some desirable function approximation properties that are lost if multiple contributions are allowed (see [F:1991]). One can also limit the *degree* of the basis function – that is, the number of factors that can appear in a single basis function.

12.1.3 Backwards-Stepwise Algorithm

MARS employs a backwards-stepwise algorithm which removes basis functions one-by-one, starting with the least significant function in terms of the fit. (The basis function $B_1(\mathbf{x})=1$ is never removed.) Thus, at the conclusion of the MARS algorithm we have a series of $M_{\max}$ candidate base sets:

$$\Theta_0 = \{1\}, \Theta_1 = \left\{1, B_{2*}(\mathbf{x})\right\}, \ldots, \Theta_{M_{\max}-1} = \left\{1, B_2(\mathbf{x}), \ldots, B_{M_{\max}}(\mathbf{x})\right\}$$

where $B_{2*}(\mathbf{x})$ is the most significant single basis function according to the backwards-stepwise algorithm. Recall that once we have a set of basis functions $\{B_1(\mathbf{x})\equiv 1, B_{2*}(\mathbf{x}),\ldots,B_{M*}(\mathbf{x})\}$, the problem of computing the corresponding coefficients $\{\alpha_1,\alpha_{2*},\ldots,\alpha_{M*}\}$ reduces to linear regression. In particular, we may compute the "new" predictors $\{z_1,z_2,\ldots,z_M\}$ where $z_1=1$, $z_j = B_{j*}(\mathbf{x})$, and solve the standard linear equations to find the coefficients that provide the best fit to the training set. Clearly we cannot increase the sum of squared errors on the training set by adding more predictors (basis functions) since we can at worst set the extra coefficients to zero. However we do expect that as the complexity of the model increases the likelihood of overfitting to the training data also increases, and the ability of the model to generalize to new data is harmed.

In linear regression, one of the methods of dealing with this is to use a modified error function that takes into account the degrees of freedom of the model. In particular, for a model with M regressor parameters fitted to n data points, the *generalized cross validation (GCV) lack of fit* measure is:

$$LOF = \frac{n\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{(n-M-1)^2} = \frac{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{(1-(M+1)/n)^2}\,.$$

As the number of predictors increases, the sum of squared errors in the numerator will decrease, but the denominator will also decrease since M is increasing. Thus if a predictor makes a large improvement in the SSE, the *LOF* will still decrease, but otherwise adding a new parameter will result in an overall increase.

This concept is adapted to the MARS framework in the *generalized cross validation error.* In MARS, the "new" predictors (or basis functions) themselves contain parameters that are optimized to fit the training data. Therefore one would expect each new basis (apart from $B_1(\mathbf{x}) = 1$) to incur a greater penalty in the lack of fit measure. In particular, the denominator is taken to be $(n - M - 1 - dM)^2$, where the parameter d estimates how many "extra" degrees of freedom should be penalized for using the MARS basis functions. There is some discussion in the literature about how to choose d, although at least in the discussion following Friedman's original article, [F:1991], $d = 2$ was raised as a reasonable candidate. In the R implementation, if the limiting degree is 2 or more than the penalty *d* is set at 2.

Using this generalized cross validation (GCV) term, we can now build the sequence of bases which include the most significant basis functions in terms of GCV. Furthermore, the basis that yields the lowest GCV can be thought of as optimal in some sense and makes a reasonable candidate for the final model.

12.1.4 MARS Functional Form

From the outline of the algorithm above it is clear that the basis functions have the form

$$B_m(\mathbf{x}) = \prod_{k=1}^{K_m}\left[s_{km}\left(x_{v(k,m)} - t_{km}\right)\right]_+$$

where K_m is the number of features of $\mathbf{x}$ included in $B_m(\mathbf{x})$, s_{km} is ± 1, $v(k,m)$ is the index of the k^{th} feature selected in the construction of $B_m(\mathbf{x})$, and t_{km} is the corresponding realization (or *split point*).

Thus MARS functions have the form:

$$f(\mathbf{x}) = \sum_{m=1}^{M}\alpha_m B_m(\mathbf{x}) = \sum_{m=1}^{M}\alpha_m \prod_{k=1}^{K_m}\left[s_{km}\left(x_{v(k,m)} - t_{km}\right)\right]_+\,.$$

12.1.5 Comments

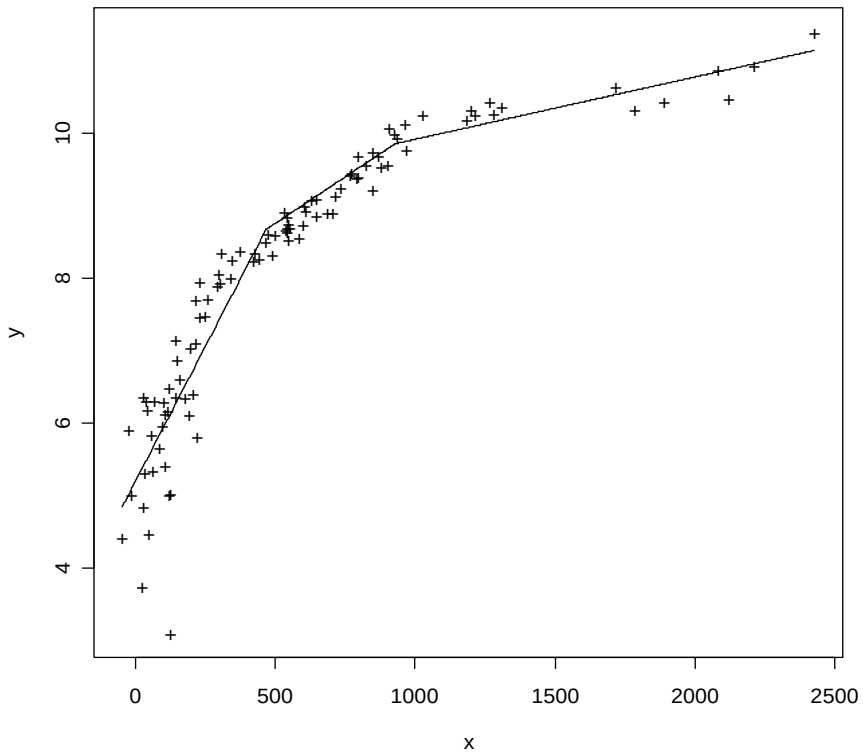
- MARS is a computationally intensive procedure when compared to traditional linear methods, but it compares favorably to training a fully connected neural network.

- It is well worthwhile doing some prior analysis on the data to reduce the feature space a much as possible.
- We can constrain the training time (and model flexibility) by means of the maximum number of basis functions $M_{\max}$. We may also restrict the allowable degree of the basis functions by means of a parameter $K_{\max}$. For example, if we set $K_{\max} = 2$, then we admit only basis functions with one or two features.
- Implementations of MARS tend to appear as a black box to the user. This may be because the form of the MARS functions are not usually very intuitive.

12.1.6 Example

The R environment implements MARS through the `mda' package. The following code fits a MARS model to the artificial regression data set introduced in the previous chapter.

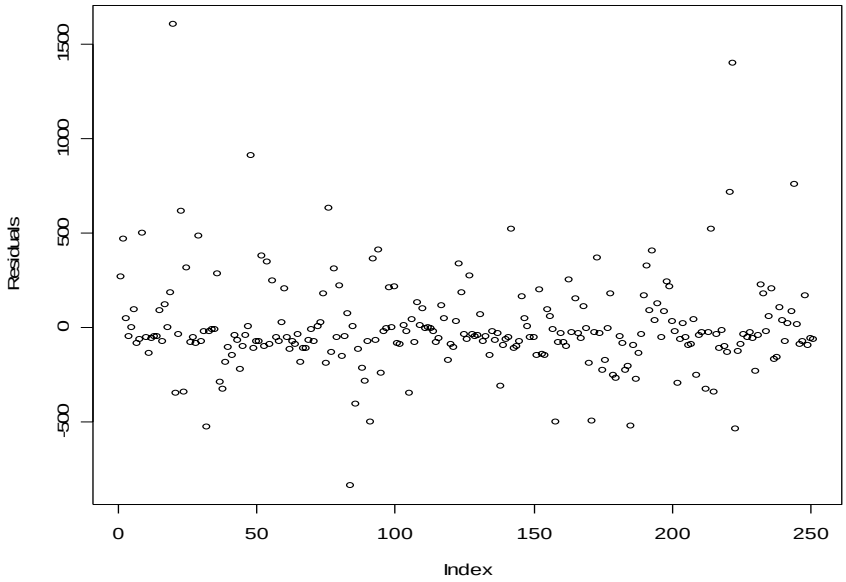
```
> # read in the data
> rdat1 <- read.table("C:\\RData\\Artificial\\RDat1.txt",header=TRUE)
> attach(rdat1)
>
> library(mda)
>
> # don't need degree > 1 with only one variable
> rmars2 <- mars(x,y,degree=1,nk=50)
>
> # generate x values that span the range
> prx <- matrix(((0:1000) * ((max(x)-min(x))/1000)) +
+ min(x),ncol=1,dimnames=list(NULL,"x"))
>
> #  fit the predicted curve
> pry2 <- predict(rmars2,prx)
>
> plot(rdat1,pch="+")
> lines(prx,pry2,lty=1)
> detach(rdat1)
```



12.1.7 Example

We return once again to the Recumbent Cow dataset to demonstrate the details of using MARS models in R.

```
> # read the data
> rcdata <- read.table("C:\\RData\\Cows\\RC Subset1.txt",header=TRUE)
>
> # predict on these variables
> predictors <- c("Uketone","Outcome","Calving","CK","Unknown")
>
> # want to predict the AST value
> response <- "AST"
>
> library(mda)      # need this package
>
> # fit a mars model with basis functions of degree 2, max of 10.
> rcmars <- mars(rcdata[,predictors],rcdata[,response],degree=2,
+ nk=10)
>
> # plot the residuals of this fit
> plot(rcmars$residuals,ylab="Residuals")
```

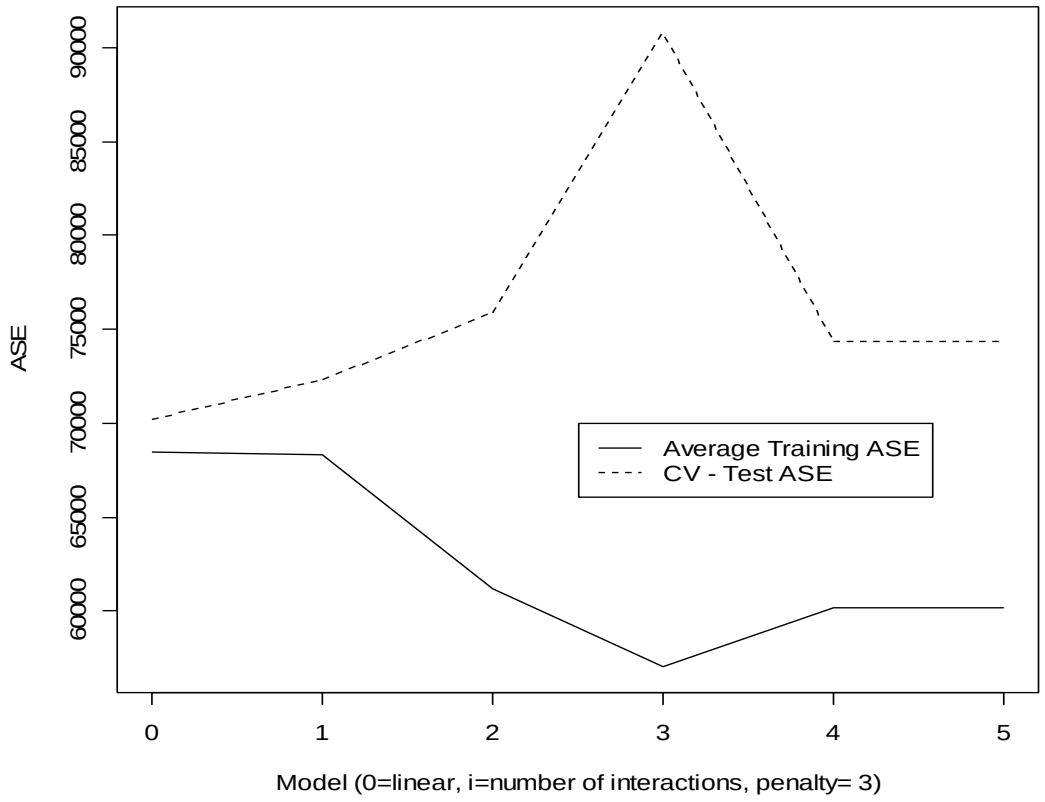


Now we make a brief comparison of MARS with linear regression for this data set. We demonstrate the use of both the penalty for computing the GCV criterion, as well as (3-fold) cross-validation. The sample is first divided into three (almost equal) thirds. Each third serves in turn as a test set while its (2/3) complement serves as the corresponding training set. For each model, the error sum of squares – both on the training portion of the data, as well as for predicting the current test set – is computed. These error sums of squares are averaged over the 3 cross-validation runs – recall that each case appears in **two** training samples, and in one test set. A linear model is fitted (model 0), followed by 4 MARS models with different $K_{\max}$ parameters (`degree'in the `mars' argument list). The $M_{\max}$ ('nk' parameter) is set to 50, and penalty parameter is first set to *d*=3 and then to d=4 (using `penalty=pen` where `pen<- 3` or `4`).

```
# now compare various MARS models with linear regression
# use 3-fold Cross Validation

nobs <- dim(rcdata)[1]

# divide cases into 3 (nearly equal) test sets for cross-validation

test1 <- sample(1:nobs,nobs/3,replace=FALSE)
test2 <- sample((1:nobs)[-test1],nobs/3,replace=FALSE)
test3 <- (1:nobs)[-c(test1,test2)] # what is left

ssetrn <- rep(0,6) # for accumulating training error sums of squares for 6 models
ssetst <- rep(0,6) # for accumulating test set error sums of squares for 6 models

for (idx in (1:3)) {

# define indices for test set and training set for each of 3 runs

bottom <- (1+(idx-1)*length(test1))
top    <- ((idx<3)*idx*length(test1)+(idx==3)*nobs)
testidx  <- c(test1,test2,test3)[bottom:top]
trainidx <- (1:nobs)[-testidx]

# do linear regression first - the 0 model
# ------------------------------------------------------------
fmla <- as.formula(paste(response,"~",paste(predictors,collapse="+")))
rclm <- lm(fmla,rcdata[trainidx,])
pred0 <- predict(rclm,rcdata[testidx,])

ssetrn[1] <- ssetrn[1]+sum(rclm$residuals^2)
ssetst[1] <- ssetst[1]+sum((pred0-rcdata[testidx,response])^2)

# fit a MARS models
pen<- 3 # setting the penalty equal to 3 extra degrees of freedom for each basis function

# MARS with no interactions -                              model 1
# ------------------------------------------------------------
rcmars1 <- mars(rcdata[trainidx,predictors],rcdata[trainidx,response],degree=1,nk=50,penalty=pen)
pred1 <- predict(rcmars1,rcdata[testidx,predictors])
ssetrn[2] <- ssetrn[2]+ sum(rcmars1$residuals^2)
ssetst[2] <- ssetst[2]+ sum((pred1-rcdata[testidx,response])^2)

# MARS with 2 interactions & at most 50 basis functions - model 2
# ------------------------------------------------------------
rcmars2 <- mars(rcdata[trainidx,predictors],rcdata[trainidx,response],degree=2,nk=50,penalty=pen)
pred2 <- predict(rcmars2,rcdata[testidx,predictors])
ssetrn[3] <- ssetrn[3]+sum(rcmars2$residuals^2)
ssetst[3] <- ssetst[3]+sum((pred2-rcdata[testidx,response])^2)

# MARS with 3 interactions & at most 50 basis functions - model 3
# ------------------------------------------------------------
rcmars3 <- mars(rcdata[trainidx,predictors],rcdata[trainidx,response],degree=3,nk=50,penalty=pen)
pred3 <- predict(rcmars3,rcdata[testidx,predictors])
ssetrn[4] <- ssetrn[4]+sum(rcmars3$residuals^2)
ssetst[4] <- ssetst[4]+sum((pred3-rcdata[testidx,response])^2)

# MARS with 4 interactions & at most 50 basis functions - model 4
# ------------------------------------------------------------
rcmars4 <- mars(rcdata[trainidx,predictors],rcdata[trainidx,response],degree=4,nk=50,penalty=pen)
pred4 <- predict(rcmars4,rcdata[testidx,predictors])
ssetrn[5] <- ssetrn[5]+sum(rcmars4$residuals^2)
ssetst[5] <- ssetst[5]+sum((pred4-rcdata[testidx,response])^2)

# MARS with 5 interactions & at most 50 basis functions - model 5
# ------------------------------------------------------------
rcmars5 <- mars(rcdata[trainidx,predictors],rcdata[trainidx,response],degree=5,nk=50,penalty=pen)
pred5 <- predict(rcmars5,rcdata[testidx,predictors])
ssetrn[6] <- ssetrn[6]+sum(rcmars5$residuals^2)
ssetst[6] <- ssetst[6]+sum((pred5-rcdata[testidx,response])^2)
}

ssetrn <- ssetrn/(2*nobs) # Note that each observation has appeared in 2 training sets
ssetst <- ssetst/(nobs)
```

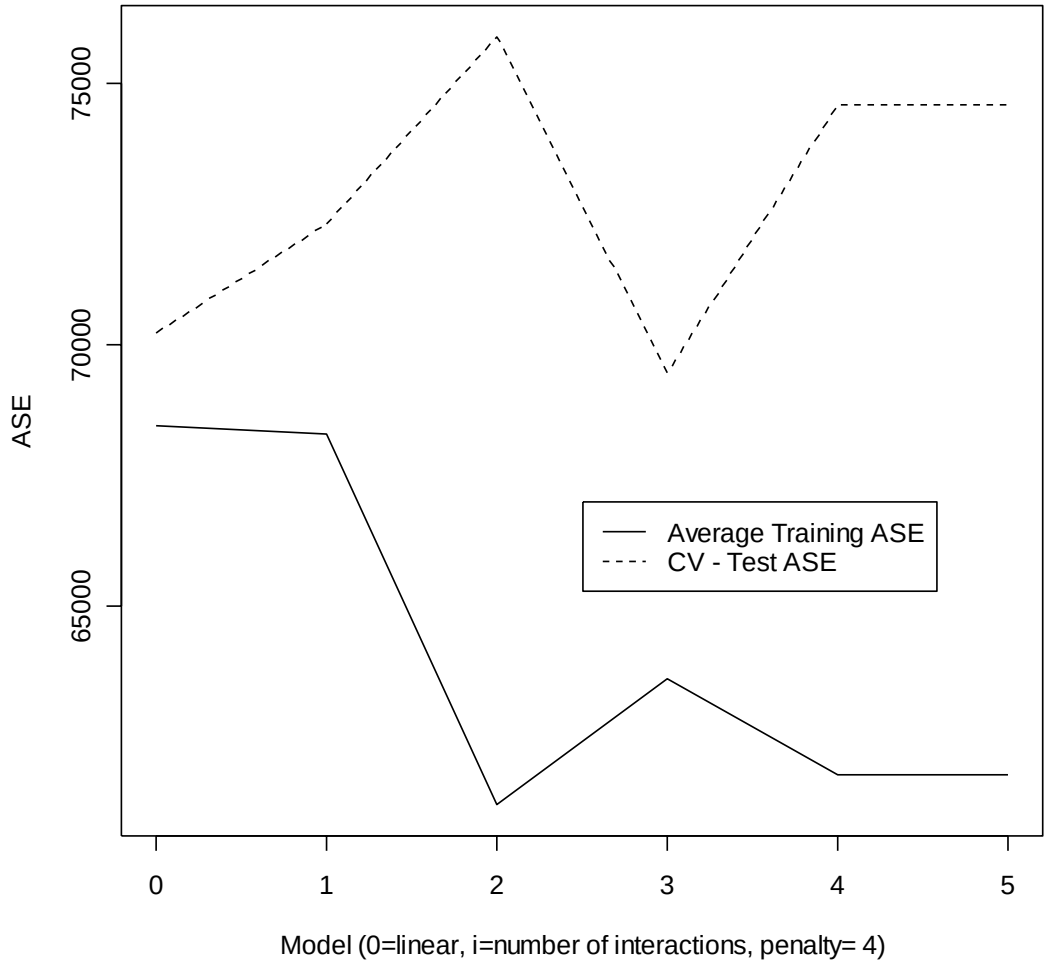
If we plot the average squared error for the training and test set (SSE divided by number of observations) we get the following:

```
plot(0:5,ssetrn,type="l",ylim=c(min(ssetrn,ssetst),max(ssetrn,ssetst)),
     xlab="Model (0=linear, i=number of interactions, penalty= 3)",ylab="ASE")
lines(0:5,ssetst,lty=2)
legend(x=2.5,y=67000,legend=c("Average Training ASE","CV - Test ASE"),lty=c(1,2))
```



This demonstrates classic overfitting behavior. The first model (0) is linear regression, which gives both the highest training ASE, and the lowest cross-validated test ASE. As the MARS models grow more complex, the training error decreases, until the penalty starts to "prevent" the model from getting too complex. However the CV test error increases – and we can conclude that the linear model is really the best for prediction for this data.

Raising the penalty to *d=4*, produces the following plot. Increasing the penalty for complexity does seem to suggest, based on the CV test criterion, that one could consider a model with 3 term interactions in the basis functions. However, playing around in this way with different penalty values, could easily add another dimension of potential overfitting.



12.2 Radial Basis Functions

The MARS approach of expressing a model as a linear combination of basis functions can be extended to more general classes of basis functions. A popular choice is *radial basis functions (RBFs)*, which propose an approximation of the form

$$f(\mathbf{x}) = \alpha + \sum_j \beta_j G\left(\left\|\mathbf{x} - \mathbf{c}_j\right\|\right)$$

for centers $\mathbf{c}_j$. Examples of G proposed include the Gaussian $G(r) = \exp\left(-r^2/2\right)$, the multiquadric $G(r) = \sqrt{\left(\gamma^2 + r^2\right)}$, and the thin plate spline function $G(r) = r^2 \log r$.

When G is Gaussian, RBF models can be seen as extending the notion of approximating a probability density by a mixture of known densities (we will look at this again in clustering).

The RBF model has been considered much in the neural network literature, where the model is treated simply as a single hidden layer neural network with radial basis activation functions.

12.3 GAM (Generalized Additive Models)

([R:1996] 4.1)

Both MARS and radial basis functions consider models which are a linear function of a basis of non-linear functions of the input space. An alternative is to consider models which are a linear function of a basis of non-linear functions of the *variables.* An *additive model* has the form $f(\mathbf{x}) = \alpha + \sum_{j=1}^{p} g_j(x_j)$ for smooth but unknown functions g_j. One choice of the smooth functions $g(x)$ of a single feature is to use splines. We have already encountered splines in our treatment of MARS. For GAM models, we choose splines $g(x)$ of the form:

$$g(x) = \sum_{i=1}^{d+1} \alpha_{i-1} x^{i-1} + \sum_{j=1}^{M} \beta_j \left[x - \xi_j \right]_+^d$$

where d is the degree of the spline, $\xi_1,\ldots,\xi_M$ are the M *knots*, and $\alpha_0,\ldots,\alpha_d,\beta_1,\ldots,\beta_M$ are real-valued parameters. The parameters are chosen either by least squares (*regression splines)* or by a solution to a particular minimization problem (*smoothing splines*).

In the case of regression splines the number and placement (say equally spaced) of the knots is given and the problem reduces to a least squares (possibly constrained) problem. In the implementation mentioned below regression splines are obtained by using the "**fx=TRUE**" option – that is, by fixing the number of knots and degrees of freedom. In the case of smoothing splines, usually a larger number of knots is offered, and the algorithm, using a Generalized Cross Validation methodology, estimates a smoothing parameter which controls how much the *wiggliness* of the fitted function should be penalized. For smoothing splines, use "**fx=FALSE**" in the **R** (**mgcv** library) implementation below, and set **k=** a larger value.

A general procedure to fit additive models is known as *back-fitting*. This holds all but one of the additive terms (features) constant, subtracts the sum of those terms from the corresponding *y* values to compute *residuals*, and fits a smooth term to the residuals against that feature. So for a particular observation $(\mathbf{x}, y)$, the model is

$$y - \alpha - \sum_{j \neq l} g_j(x_j) = g_l(x_l)$$

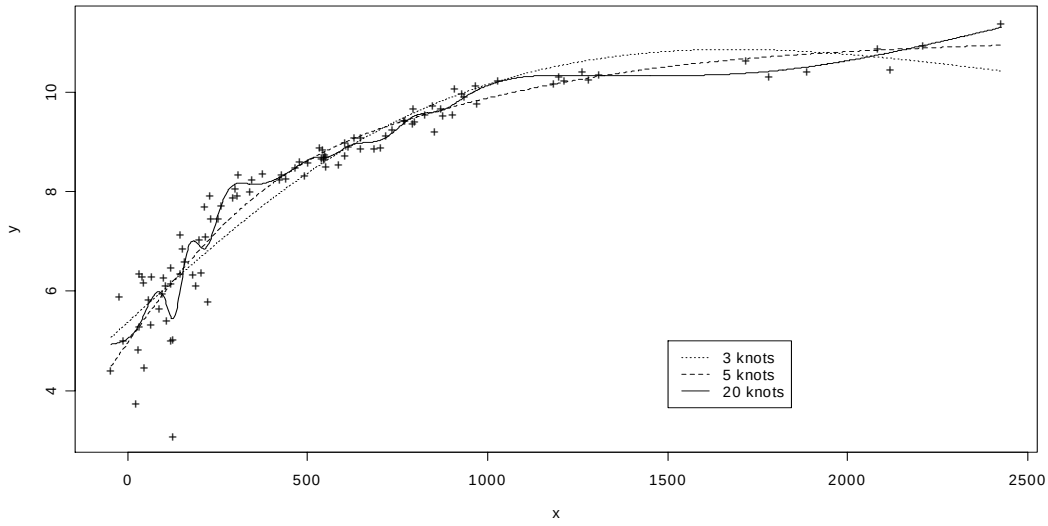
and any smoothing algorithm (which dictates the form of the functions g_i) can be applied to the left hand side. Smoothing is applied a feature at a time until the process converges.

12.3.1 Example

GAM is modeled in R using the `mgcv' package. Using the artificial regression dataset we may see how the models increase in flexibility as more degrees of freedom are allowed. Below, we determine, with no flexibility (**"fx=TRUE"**), exactly how many knots are to be used.

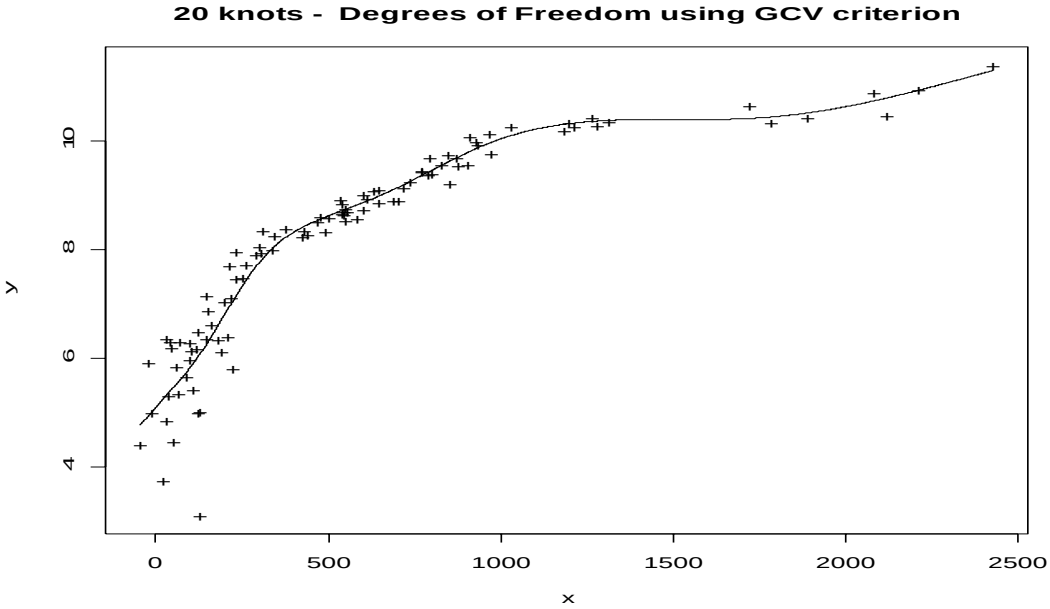
```
> rdat1 <- read.table("C:\\RDat1.txt",header=TRUE)
> attach(rdat1)
>
> library(mgcv)   # need this package
>
> # cubic splines with 3 knots
> rgam1 <- gam(y~s(x,k=3,fx=TRUE,bs="cr"),family=gaussian(),rdat1)
> # quadratic terms and cubic splines with 5 knots
> rgam2 <- gam(y~s(x,k=5,fx=TRUE,bs="cr"),family=gaussian(), rdat1)
> # quadratic terms and cubic splines with 20 knots
```

```
> rgam3 <- gam(y~s(x,k=20,fx=TRUE,bs="cr"),family=gaussian(),rdat1)
>
> # fit the predicted curves
> prx <- matrix(((0:1000)*((max(x)-min(x))/1000))+min(x))
> pry1 <- predict(rgam1,newdata=as.data.frame(prx))
> pry2 <- predict(rgam2,newdata=as.data.frame(prx))
> pry3 <- predict(rgam3,newdata=as.data.frame(prx))
>
> plot(rdat1,pch="+"); lines(prx,pry1,lty=3);
> lines(prx,pry2,lty=2); lines(prx,pry3,lty=1)
> detach(rdat1)
```



Once again we see that the model flexibility increases as the number of knots (number of parameters in the model) increases.

If we allow the GAM function to automatically choose the optimal degree of smoothing (degrees of freedom of the smoothing function), by setting "**fx=FALSE**", then we obtain the following smooth of the same data.



12.4 Loess Smoothing

The *loess smoothing* algorithm uses robustly fitted locally weighted polynomials to fit a smooth curve within the range of the training set. Suppose we have training data $\{(\mathbf{x}_1, y_1),\ldots,(\mathbf{x}_n, y_n)\}$ and we wish to fit a smooth curve to each input value $\mathbf{x}$ in the input space. For each $\mathbf{x}_i$ the loess smoother assigns a weight to the new pattern $\mathbf{x}$

$$W_i(\mathbf{x}) = \left[1 - \left(\frac{\rho(\mathbf{x},\mathbf{x}_i)}{\tau(\mathbf{x})}\right)^3\right]_+^3$$

where $\rho(\mathbf{x},\mathbf{x}_i)$ is some distance measure between the two points, and $\tau(\mathbf{x})$ is a smoothness parameter. This could be simply a constant value, but it is often taken to be some multiple of the distance between $\mathbf{x}$ and the q^{th}-farthest point from $\mathbf{x}$ where q is a user supplied parameter. Let $h(\mathbf{x};\phi)$ be a polynomial of degree d (usually 1 or 2). Then the fitted value $g(\mathbf{x})$ at $\mathbf{x}$ is found by fitting $h(\mathbf{x};\phi)$ according to the weights and reporting the fitted value at $\mathbf{x}$. Specifically, $\hat{\phi}_{\mathbf{x}}$ is chosen to minimize

$$\sum_{i=1}^{n} W_i(\mathbf{x})(y_i - h(\mathbf{x}_i;\phi))^2$$

and $\hat{g}(\mathbf{x}) = h(\mathbf{x};\hat{\phi}_{\mathbf{x}})$. Note that the term $(y_i - h(\mathbf{x}_i;\phi))^2$ is called the *loss function,* and may be replaced with a more general expression, perhaps to penalize large departures less severely.

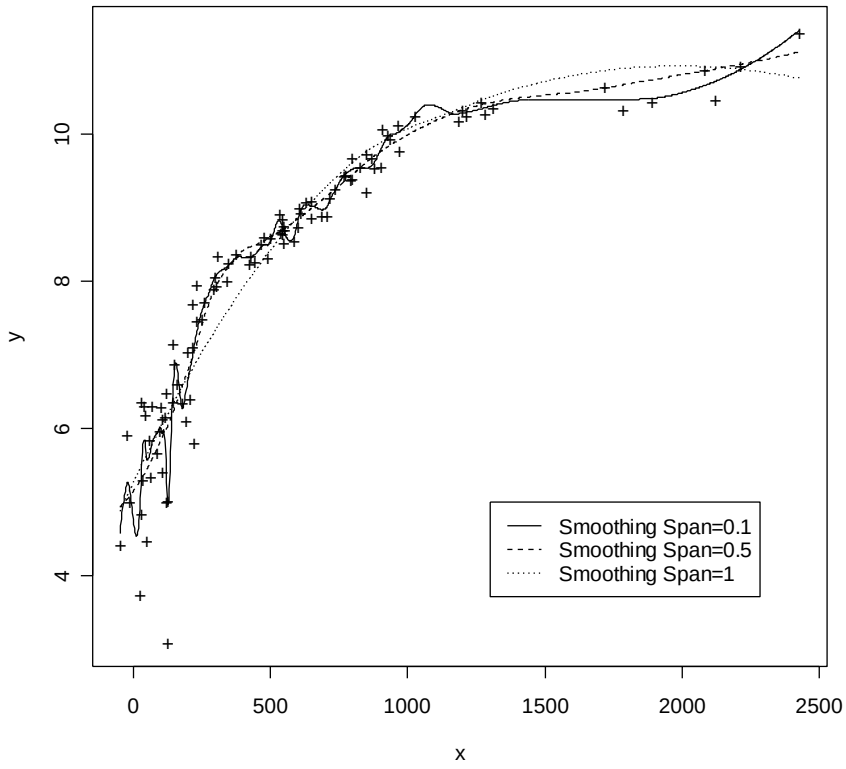
Loess smoothing has similar constraints to *k*-NN classification. For each new prediction the whole training set must be retained and the distance between every point in the training set and the new observation needs to be computed (unless the training set has been partitioned in some clever way). On top of that, a polynomial has to be fitted for each new prediction, although depending on the size of the smoothing window this may not be a large computational load.

12.4.1 Example

There are a number of implementations of loess in R. We shall demonstrate the implementation in the `modreg' package. The $\tau(\mathbf{x})$ parameter in this implementation is controlled by the argument `span'. This argument determines the proportion of points in the dataset used to compute the smoothed value at $\mathbf{x}$, so span=0.2 dictates that the 20% of the data nearest to $\mathbf{x}$ is used to compute the smoothed value. In the following R code we fit the artificial regression dataset using three loess models with smoothness parameters 0.1,0.5 and 1 respectively.

```
> # read in the data
> rdat1 <- read.table("C:\\RData\\Artificial\\RDat1.txt",header=TRUE)
> attach(rdat1)
>
> library(modreg)
>
> rloess1 <- loess(y~x,span=0.1)
> rloess2 <- loess(y~x,span=0.5)
> rloess3 <- loess(y~x,span=1)
>
> # fit the predicted curves
> prx <- matrix( ((0:1000) * ((max(x)-min(x))/1000)) + min(x),ncol=1,
+ dimnames=list(NULL,"x") )
>
> pry1 <- predict(rloess1,prx)
```

```
> pry2 <- predict(rloess2,prx)
> pry3 <- predict(rloess3,prx)
>
> plot(rdat1,pch="+")
> lines(prx,pry1,lty=1)
> lines(prx,pry2,lty=2)
> lines(prx,pry3,lty=3)
> > legend(1300,5,legend=c("Smoothing Span=0.1","Smoothing Span=0.5",
+ "Smoothing Span=1"), lty=c(1,2,3))
>
> detach(dat1)
```



We see clearly that the smaller the span, the more flexible the model.

12.5 Kernel Methods

Finally we consider the class of *kernel methods.* A *kernel* K is a bounded function on the input space that integrates to 1, for example, a probability density function. We assume that K is in some sense peaked around 0, and then use $K(\mathbf{x}_1 - \mathbf{x}_2)$ as a measure of the proximity of $\mathbf{x}_1$ and $\mathbf{x}_2$. This does suggest we should take $K(\mathbf{x}) = K(-\mathbf{x})$, and indeed, this condition is often imposed. The intuition is that a kernel gives a sort of local weighting around each point in the training set which can be used to predict the response of unseen examples. To summarize, a kernel K fulfills the following requirements:

1. $K(\mathbf{x}) \geq 0$ for all $\mathbf{x}$ in the feature space.
2. $\int K(\mathbf{x}) d\mathbf{x} = 1$.
3. $K(\mathbf{x}) = K(-\mathbf{x})$ for all $\mathbf{x}$.

Thus we may think of the kernel function K as probability density function (from conditions 1 and 2) that is symmetric around 0 (condition 3). We discuss kernel candidates below.

For a given training set $\{(\mathbf{x}_1, y_1),\ldots,(\mathbf{x}_n, y_n)\}$, and kernel K, the *Nadaraya-Watson non-parametric kernel regression estimate* is given by:

$$\hat{y}(\mathbf{x}) = \frac{\sum_i y_i K\left(\frac{\mathbf{x}-\mathbf{x}_i}{b}\right)}{\sum_i K\left(\frac{\mathbf{x}-\mathbf{x}_i}{b}\right)}$$

where b is the *bandwidth parameter.* Since the kernel function has most of its density around 0, then by increasing b we can include more observations $\mathbf{x}_i$ in the kernel estimate of $y(\mathbf{x})$. In the limit, all observations in the existing dataset will be included and the smoothest possible estimate will be obtained. Alternatively, as $b \to 0$, only points very near $\mathbf{x}$ are strongly weighted.

In the special case of a classification problem we can use the above formula to estimate $f_k(\mathbf{x}) = p(k \mid \mathbf{x})$. In this case y is the indicator for class k and we have

$$\hat{p}(k \mid \mathbf{x}) = \frac{\sum_i I[y_i = k] K\left(\frac{\mathbf{x}-\mathbf{x}_i}{b}\right)}{\sum_i K\left(\frac{\mathbf{x}-\mathbf{x}_i}{b}\right)}.$$

The R function implementing Nadaraya-Watson kernel regression is `ksmooth' in the `modreg' package. This function is implemented only for 1-D feature space, so the available kernels are functions on the real line. There are two kernels options available, the *box* kernel and the *normal* kernel. The box kernel

$$K_{box}(t) = \begin{cases} 1, & |t| \le 0.5 \\ 0, & |t| > 0.5 \end{cases}$$

gives equal weight to all observations in the "box" $[-0.5b, 0.5b]$. All observations outside this box are weighted to 0. The normal kernel fits a normal density function centered around 0:

$$K_{normal}(t) = \frac{1}{(0.37)\sqrt{2\pi}} e^{-t^2/2(0.37)^2}.$$

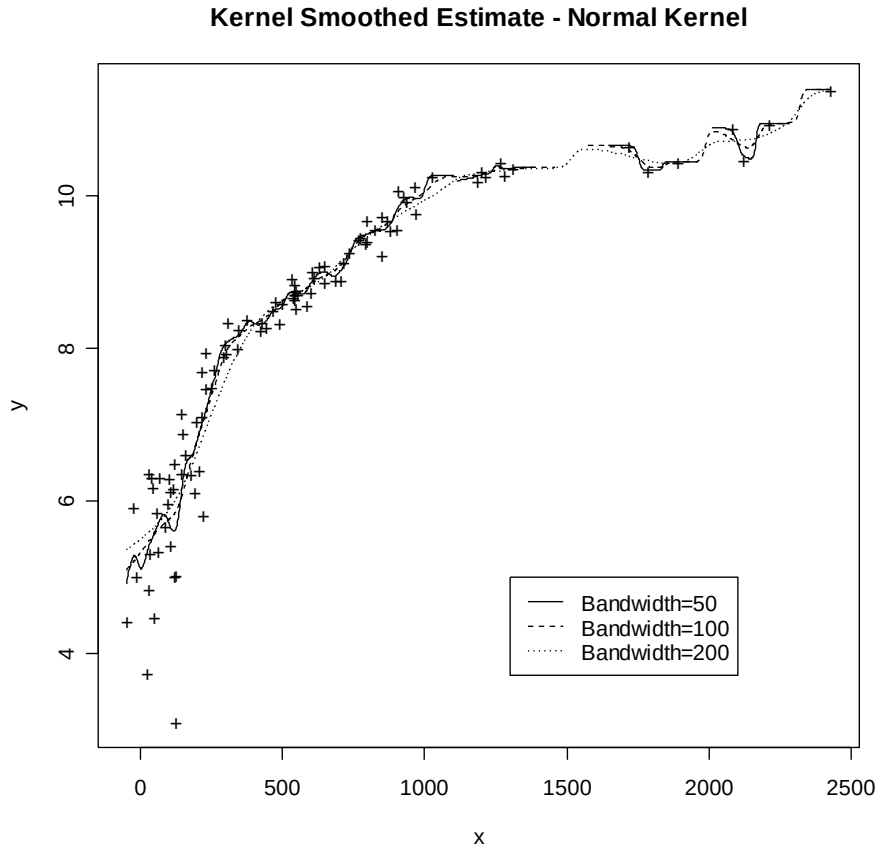
In both cases the kernel function are chosen so the upper and lower quartiles of the corresponding distribution are ± 0.25.

12.5.1 Example

The ksmooth function takes a predictor and response variable, and returns the kernel-smoothed estimate for n.points points evenly spanning the range of the input observations. For this example we will only use the normal kernel. The changes for using the box kernel are trivial. Once again we shall smooth our artificial dataset:

```
> # read in the data
> rdat1 <- read.table("C:\\RData\\Artificial\\RDat1.txt",header=TRUE)
> attach(rdat1)
>
> library(modreg)
>
```

```
> rksn1 <- ksmooth(x,y,kernel="normal",bandwidth=200,n.points=1000)
> rksn2 <- ksmooth(x,y,kernel="normal",bandwidth=100,n.points=1000)
> rksn3 <- ksmooth(x,y,kernel="normal",bandwidth=50,n.points=1000)
>
> plot(rdat1,pch="+",main="Kernel Smoothed Estimate - Normal Kernel")
> lines(rksn1,lty=3)
> lines(rksn2,lty=2)
> lines(rksn3,lty=1)
>
> legend(1300,5,legend=c("Bandwidth=50","Bandwidth=100",
+ "Bandwidth=200"),lty=c(1,2,3))
> detach(rdat1)
```



It is clear from the plot that as the bandwidth increases, the estimate is smoother. However, note that some of the curves (in particular, the bandwidth 50 curve), there are discontinuities. These correspond to sections in which the data is so sparse that the kernel estimate generates an N/A value since the denominator in the estimate is zero. This problem is even more acute when the box kernel is used

13 Clustering Methods

[KR:1990],[SHR:1997]

13.1 Introduction

Clustering algorithms are methods to divide a set of n observations into k groups so that the members of each group are more similar to each other than they are to members of other groups. Clustering methods are generally viewed as *investigative* techniques designed to reveal structure in the data set. Alternatively clustering may impose an external structure on a data set for some other purpose (e.g. we may wish to cluster telephone customers into exchange districts such that each exchange has a relatively equal load and the customers are not too far away). Of course, once a clustering structure has been revealed (or imposed) the clustering algorithms can sometimes be extended to allow classification of new observations into existing clusters.

There can be many reasons why we might want to find clusters in our data set. Given the investigative nature of the task we could think of clustering as a search for (hopefully) natural subsets in the data. These subsets can then be examined to see if some further information of interest can be found. For example, we may cluster a data set of credit card transaction data, and find upon further examination that one or more of the clusters contain a high proportion of fraudulent transactions. This, in turn, can provide a basis for a detection system.

A huge number of clustering algorithms exist in the current literature and we shall not attempt to cover them all here. Rather we shall address three general approaches to clustering and outline some algorithms from each. Most of the methods described below are implemented in the `cluster' package in R, which is well documented in [SHR:1997].

Fundamental to any clustering algorithm is the ability to compute the distance (or *dissimilarity*) between any pair of cases. In fact, many algorithms will work if given only the matrix of pair-wise distances, without having direct knowledge of the actual feature values for each case. If distances are not provided explicitly, the algorithms will typically assume data in the data-matrix lie in a Euclidean space, and therefore compute Euclidean distances in order to proceed. As a result, the user must be aware of the need to properly scale the features a-priori. See also Section **2.2** concerning computing general distances.

13.2 Partitioning Methods

Partitioning methods divide the data set into a pre-assigned number of groups. The requirements of a partitioning method are that

- Each group contains at least one object
- Each object is contained in exactly one group.

Therefore the number of clusters k must be less than or equal to the number of observations in the data set.

Note that in these algorithms the number of clusters is assigned by the user. In some cases (such as the telephone exchanges) this is fine, because a clustering of specific dimension must be applied to the data. However in most cases we seek a clustering that is somehow "natural" to the data. Unless there is some sort of prior knowledge,

the best approach is to partition for several values of k and select the value that gives the most natural partition according to some distance measure, or visual plot. Alternatively, if a measure akin to the purity in decision trees can be provided, we may simply choose to optimize the result over k by an iterative search.

13.2.1 Silhouette Value & Plot

An alternative method for finding k was proposed by Rousseeuw ([R:1986]). In this method a simple index, the *silhouette value*, is computed for each clustering of a dataset. The size of the silhouette value gives a rough indicator as to the quality of the clustering. Suppose we have some clustering $C_1, C_2, ..., C_k$ of our data, using a dissimilarity measure $d(i,j)$. Then for an observation i in cluster C_g, we define the *average dissimilarity* of i from the other objects in C_g as:

$$a(i) = \frac{1}{|C_g| - 1} \sum_{j \in C_g, j \neq i} d(i,j) .$$

For any other cluster, C_l, $l \neq g$, we define the average dissimilarity of i to the objects in C_l by:

$$d(i, C_l) = \frac{1}{|C_l|} \sum_{j \in C_l} d(i,j) .$$

The cluster C_h (other than C_g) satisfying the condition

$$d(i, C_h) \leq d(i, C_l) \qquad \forall l \neq g$$

is called the *neighbor* of observation i, and we let

$$b(i) = d(i, C_h) .$$

Then the silhouette value for observation i is

$$s(i) = \frac{b(i) - a(i)}{\max\{a(i), b(i)\}} .$$

Clearly this value always lies between -1 and 1. Clearly if $b(i)$ is large compared to $a(i)$, $s(i)$ will be close to 1. This indicates that i is well classified in its current cluster, since its neighbor is comparatively far away. To summarize, the value $s(i)$ may be interpreted as follows:

- $s(i) \approx 1 \Rightarrow i$ is well classified in cluster C_g.
- $s(i) \approx 0 \Rightarrow i$ lies on the border between C_g and C_h.
- $s(i) \approx -1 \Rightarrow i$ is badly classified in cluster C_g (closer to C_h than to C_g).

For a given clustering, we may plot the silhouette values for each object. The values are grouped into their clusters and sorted from largest to smallest. This gives a quick summary of the quality of a clustering, and associated problematic areas. We shall see examples of silhouette plots in the section on k-medoids.

The average of all the silhouette values (*average silhouette width*) gives a measure of the overall quality of a clustering. Thus, we may search for an optimal k simply by taking the k value corresponding to the largest average silhouette width. This quantity is called the *silhouette coefficient* (SC), and gives a measure of the clustering

structure that exists in a particular dataset. In practice we cannot normally search over all possible values of k, rather we take a few values and perform some kind of line-search heuristic to find a good candidate for k.

As a general rule of thumb, we may interpret the SC as follows ([SHR:1997]:

- SC $\leq$ 0.25 $\Rightarrow$ no significant clustering structure was found.
- SC $\in$ (0.25,0.5] $\Rightarrow$ structure is weak and may be artificial, try additional clustering methods.
- SC $\in$ (0.5,0.7] $\Rightarrow$ a reasonable structure has been found.
- SC > 0.7 $\Rightarrow$ strong clustering structure has been found.

The SC and the silhouette values provide a simple, understandable measure of the effectiveness of a clustering. In the R environment, silhouette functions are only implemented for the partitioning methods in the `cluster' package, however it is a relatively trivial exercise to write a general function for any algorithm that generates clusters from a dataset and has a corresponding dissimilarity measure between observations.

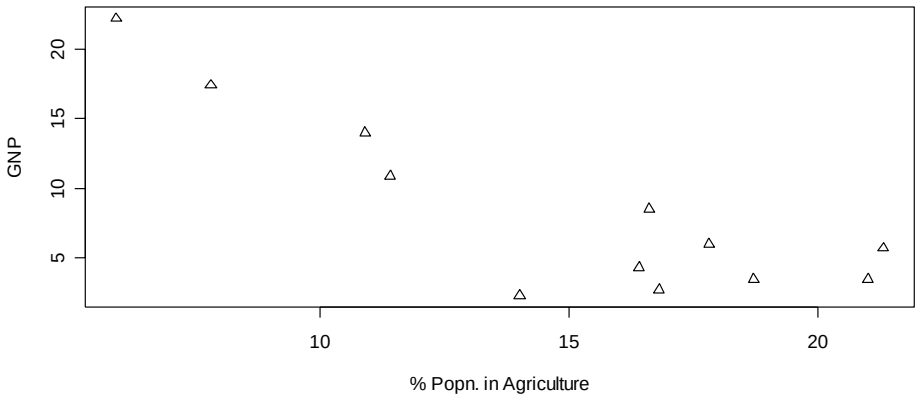
13.2.2 Partitioning Example Datasets

For the purposes of instruction, it is best to test partitioning methods on two dimensional data so as to facilitate easy plotting and visualization of the results. We shall use three datasets for examples in the partitioning methods. Each dataset actually has quite good spatial separation, so are not necessarily appropriate for testing between methods. All the datasets are found in the `cluster' package in R.

```
>library(cluster)
```

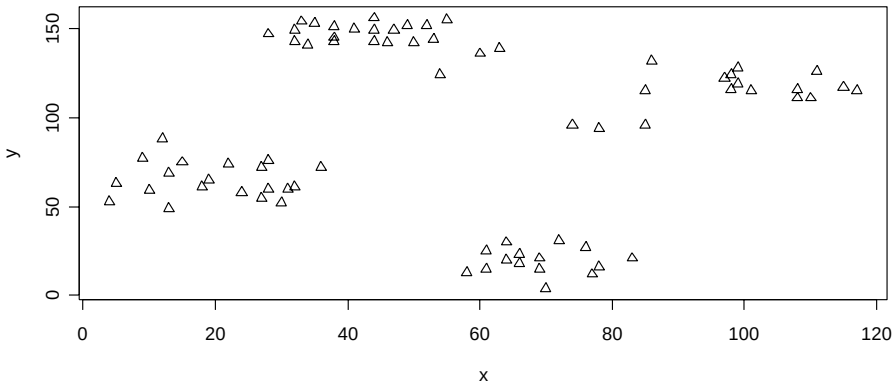
The first data set, `agriculture' gives the GNP of a number of European countries, together with the percentage of the population in those countries working in agriculture.

```
> data(agriculture)
> plot(agriculture, pch=2, xlab="% Popn. in Agriculture",
+ ylab="GNP")
```



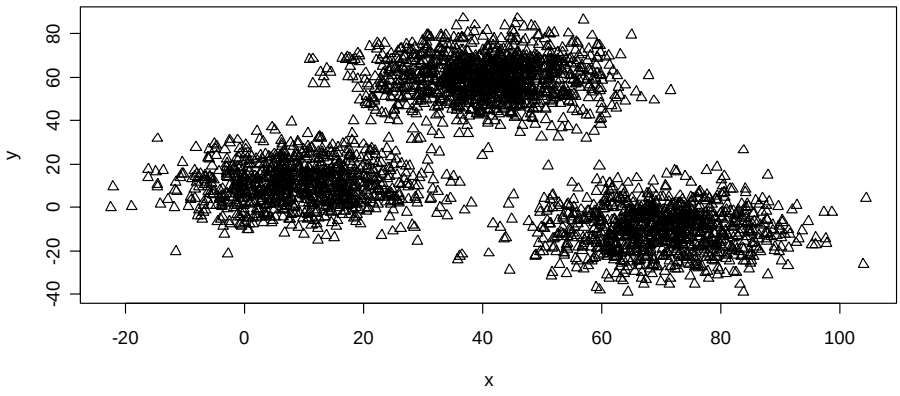
The second data set, `ruspini' consists of 75 data points in 4 groups. It is a very commonly used dataset for demonstrating clustering techniques.

```
> data(ruspini)
> plot(ruspini, pch=2, xlab="x",ylab="y")
```



The third dataset, `xclara' is an artificial dataset with 3000 points in 3 well defined clusters of 1000.

```
> data(xclara)
> plot(xclara,pch=2,xlab="x",ylab="y")
```



13.2.3 K-Means Clustering

Suppose the data consists of continuous features (i.e. resides in $\Re^p$ where p is the dimension of the feature space). Then we can assign a cluster center $\mathbf{m}$ to each group, and then choose both the centers and the groups to minimize the sum of squared distances from each example to its cluster center. That is, if $c:\{1,2,\ldots,n\}\mapsto\{1,2,\ldots,k\}$ is a function assigning each data point to one of the k clusters, and $\mathbf{m}_j$ is the center (component-wise mean) of the j^{th} group, we seek to minimize

$$\sum_{i=1}^{n} \left\| \mathbf{x}_i - \mathbf{m}_{c(i)} \right\|^2$$

over all partitions c. Clearly this method can be extended to more general distance measures (such as Gower's dissimilarity, introduced in Chapter 2), but classical k-means is based on the Euclidean distance (squared).

For a given cluster, it is easy to assign the center – we simply use the mean of all the observations in the cluster. It is somewhat more difficult to deal with the combinatorial task of minimizing over all clusterings. Once again there have been many algorithms proposed for this task. The algorithms generally begin with some division of the examples into k groups, or some assignment of k cluster centers. From the initialization step, some sort of iterative update rule is used, for example:

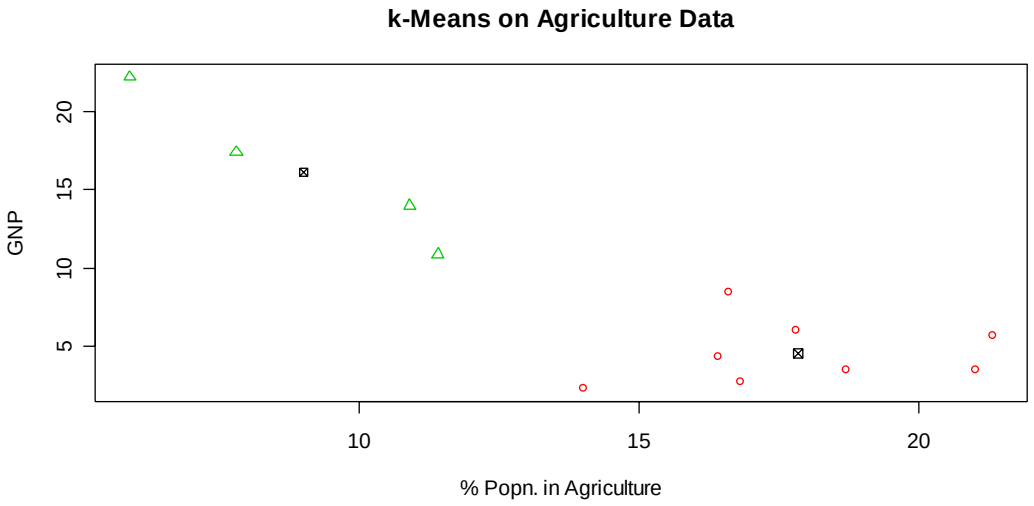
- Re-assign all examples to their nearest cluster center, then recomputed the cluster mean and repeat the process. Note that this process may empty a group, in which case less than k groups may result.
- Read each example in turn and assign that example to the nearest cluster center. Recompute the cluster means for both the source group and the destination group.
- Some algorithms also allow the splitting and merging of clusters, so k changes dynamically as part of the optimization process.

The k-means algorithms extend in the obvious fashion to classification. Any new observation is measured against each of the cluster centers and assigned to the nearest.

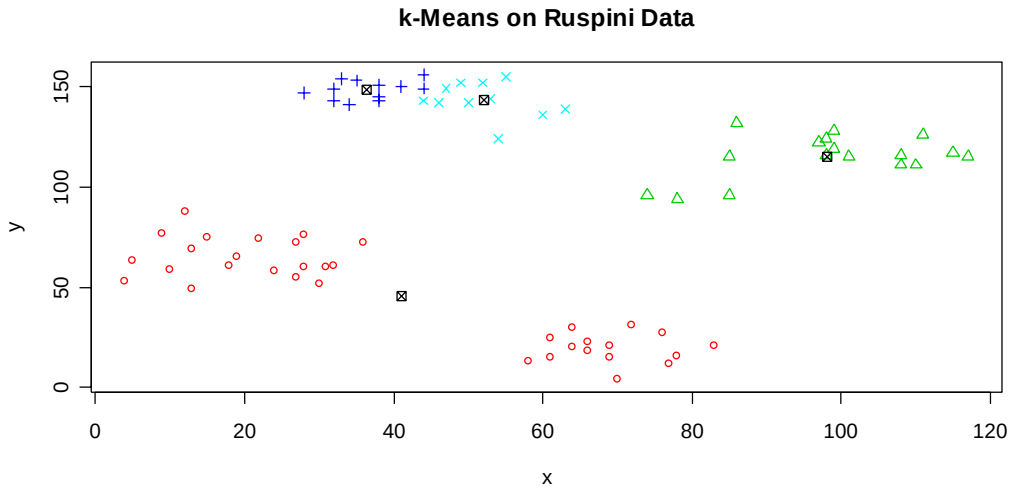
13.2.4 Examples

The `mva' package in R has an implementation of *k*-means clustering. We shall apply this function to the three datasets above, using 2, 4 and 3 clusters respectively. The centers of each cluster are marked on the plots.

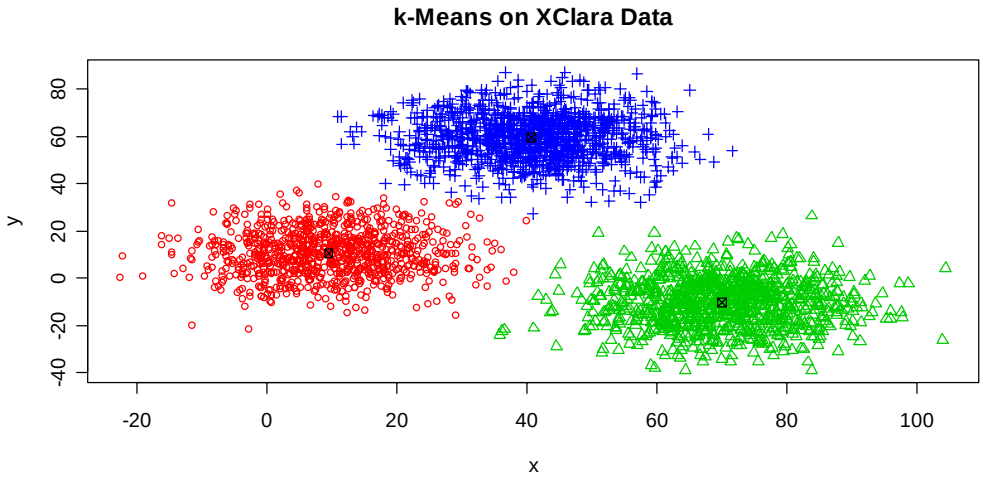
```
> library(mva)
>
> # cluster the agriculture data using 2-means
> kma2 <- kmeans(agriculture,centers=2)
>
> # plot the data with different color / char for each cluster
> plot(agriculture, pch=kma2$cluster,col=kma2$cluster+1, ylab="GNP",
+ xlab="% Popn. in Agriculture", main="k-Means on Agriculture Data")
>
> # now plot the centers of each cluster.
> points(kma2$centers,pch=7)
```




```
> # cluster the ruspini data using 4-means
> kmr4 <- kmeans(ruspini,centers=4)
>
> # plot the data with different color / char for each cluster
> plot(ruspini, pch=kmr4$cluster,col=kmr4$cluster+1,xlab="x",
+ ylab="y", main="k-Means on Ruspini Data")
>
> # now plot the centers of each cluster.
> points(kmr4$centers,pch=7)
```




```
> # cluster the xclara data using 3-means
> kmx3 <- kmeans(xclara,centers=3)
>
> # plot the data with different color / char for each cluster
> plot(xclara, pch=kmx3$cluster,col=kmx3$cluster+1,xlab="x",ylab="y",
+ main="k-Means on XClara Data")
>
> # now plot the centers of each cluster.
> points(kmx3$centers,pch=7)
```



The k-means method classifies agriculture and xclara very well, but is not able to find good clusters for the ruspini data. It is possible that a different initialization will give better results on this dataset.

13.2.5 K-Medoid Clustering

The k-means algorithm is restricted to choosing cluster centers in $\Re^p$, so is only useful when the dataset consists of continuous features, and the whole dataset is accessible for computing the means. These restrictions can be overcome by requiring that the cluster centers be examples. Then we seek a partition c and cluster centers $\mathbf{x}^{(g)}$ such that

$$\sum_{i=1}^{n} d\left(\mathbf{x}_i, \mathbf{x}^{(c(i))}\right)^2$$

is minimized for some dissimilarity measure d. If d is the Euclidean distance, then this is essentially k-means with, say, the median $\mathbf{x}^{(g)}$ of each group chosen for the cluster center instead of computing the mean. The squared dissimilarity may be dominated by outliers, so it is usual to use dissimilarities without squaring. This is known as *k-median* or *k-mediod clustering.* The algorithm arbitrarily chooses k examples as the initial centers, and then considers swapping a center with an example which is not a center, selecting the swap that reduces the dissimilarity measure the most.

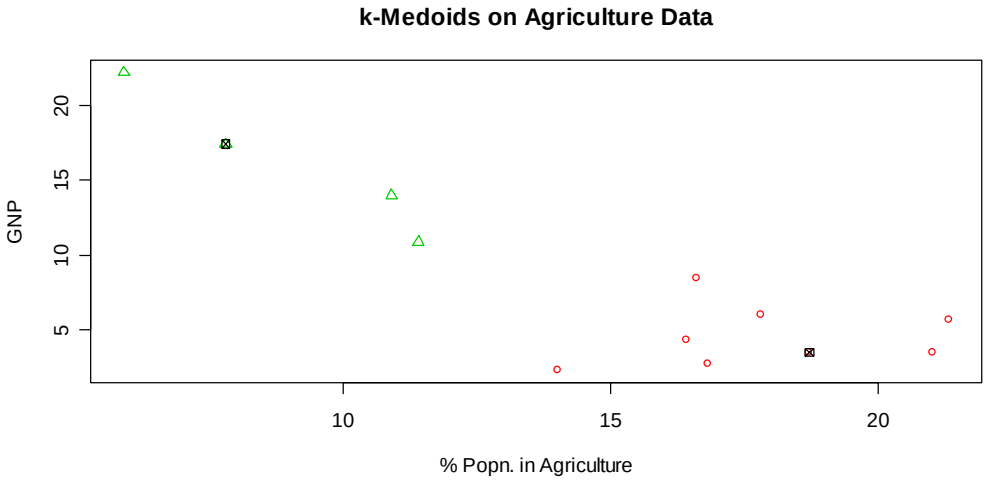
Once again, the choice of d is dictated by the data-types in the observations. k-medoid clustering is implemented in R using the `pam' ("**p**artitioning **a**round **m**edoids") function, from the `cluster' package. This function uses the Euclidean distance by default, but also allows the Manhattan metric by a command line argument, and general metrics via a dissimilarity matrix argument.

13.2.6 Examples

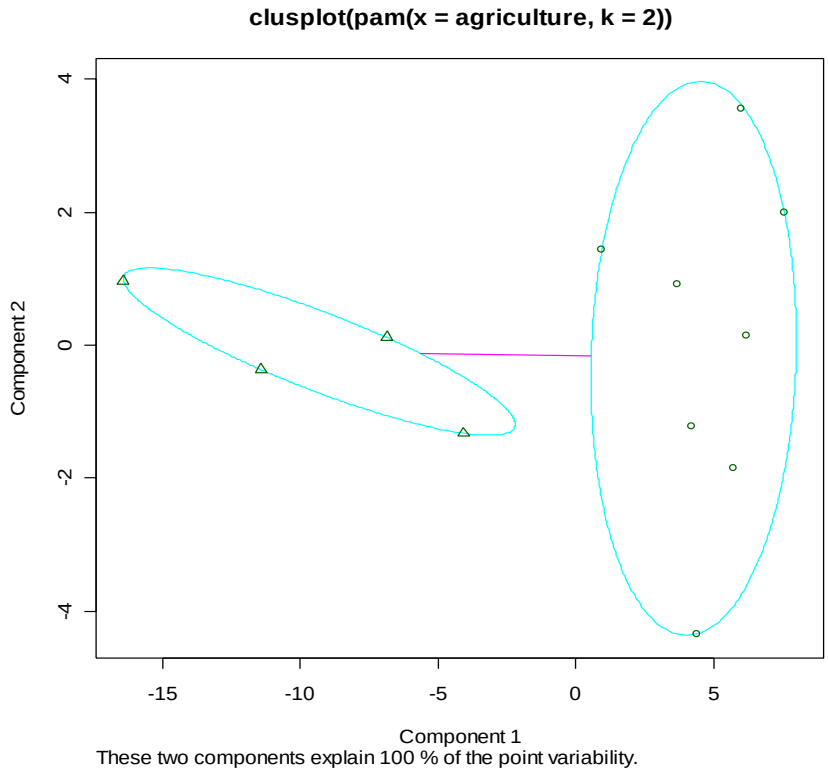
In these examples we shall cluster using several different k values to demonstrate the silhouette plot and silhouette coefficient.

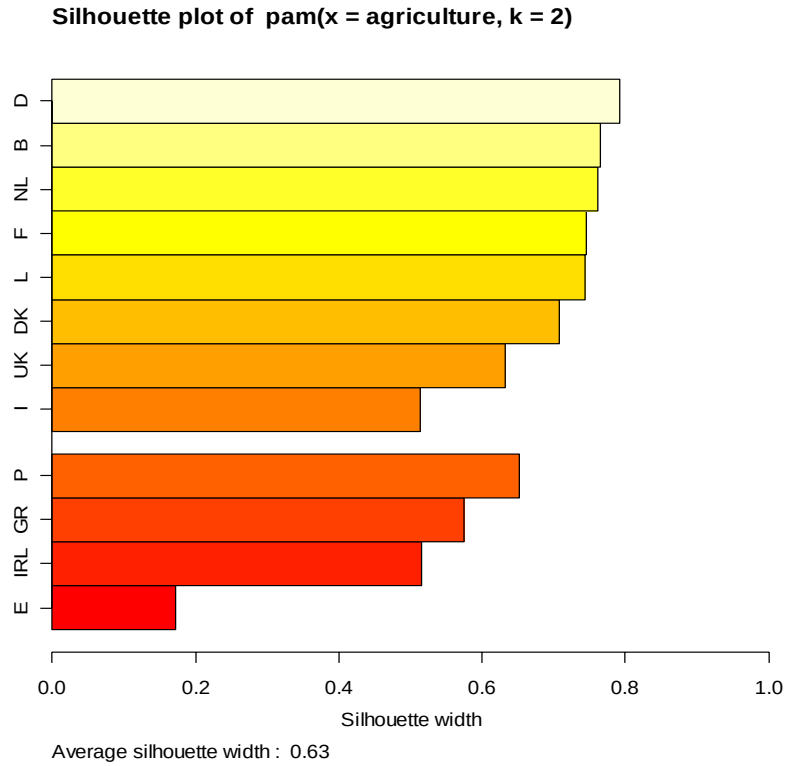
```
> # cluster the agriculture data using 2-medoids
> kda2 <- pam(agriculture,k=2)
```

```
>
> # plot the data with different color / char for each cluster
> plot(agriculture, pch=kda2$clustering,col=kda2$clustering+1,
+ xlab="% Popn. in Agriculture",ylab="GNP",
+ main="k-Medoids on Agriculture Data")
>
> # now plot the mediods of each cluster.
> points(kda2$medoids,pch=7)
```



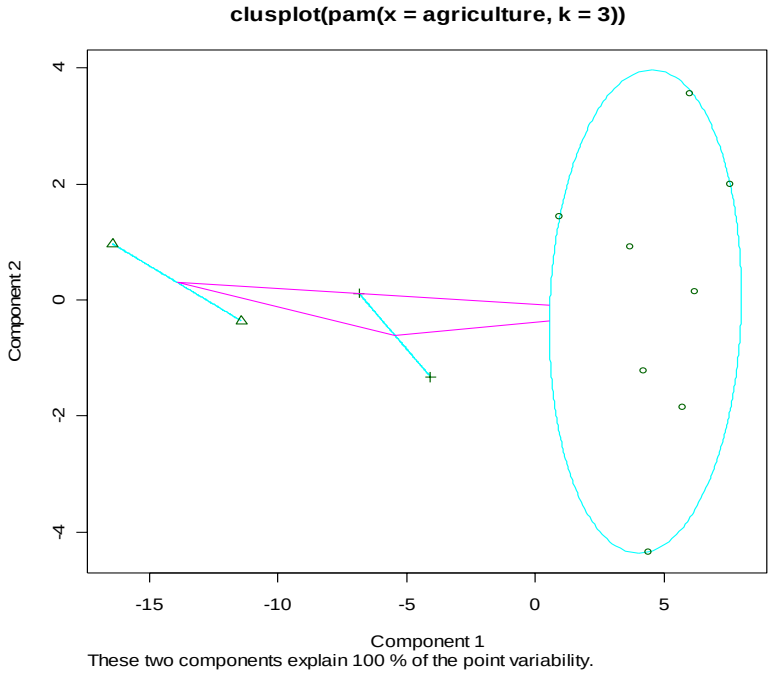

```
> # use the inbuilt plot option for cluster and silhouette plots
> plot(kda2)
Hit <Return> to see next plot:
Hit <Return> to see next plot:
>
```



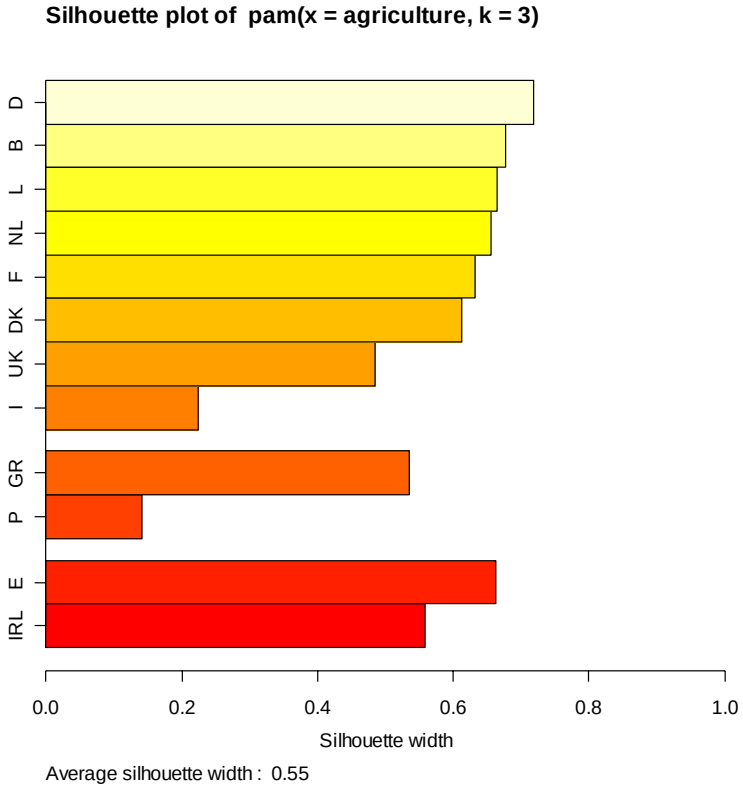


Note that the first plot (clusplot) is a two-dimensional projection of the points in a plane defined by the first two principal components. In this sense, for a high dimensional **x** vector of features, clusters may not seem well-separated in such a two-dimensional projection, especially if the explained variability is low.

```
> # now fit 3-mediods to the agriculture data
> kda3 <- pam(agriculture,k=3)
> plot(kda3)
Hit <Return> to see next plot:
Hit <Return> to see next plot:
```

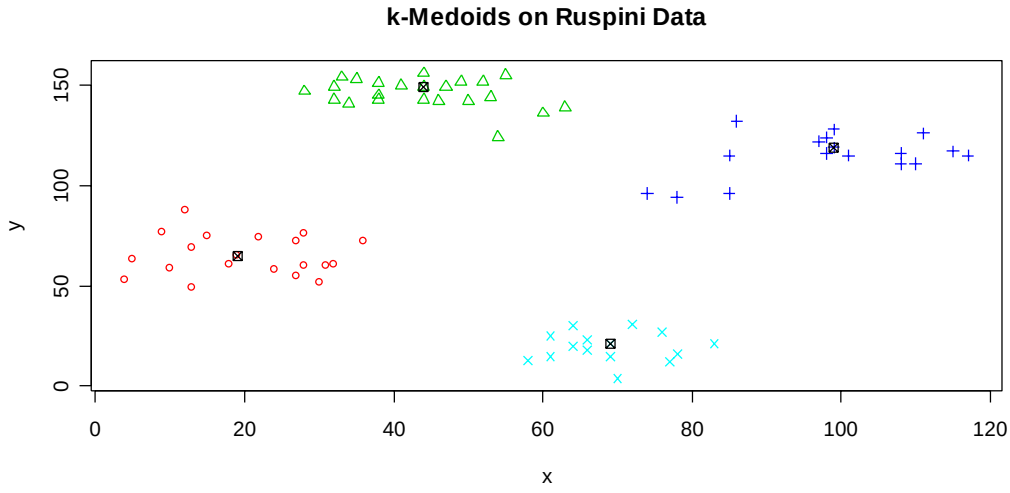


>



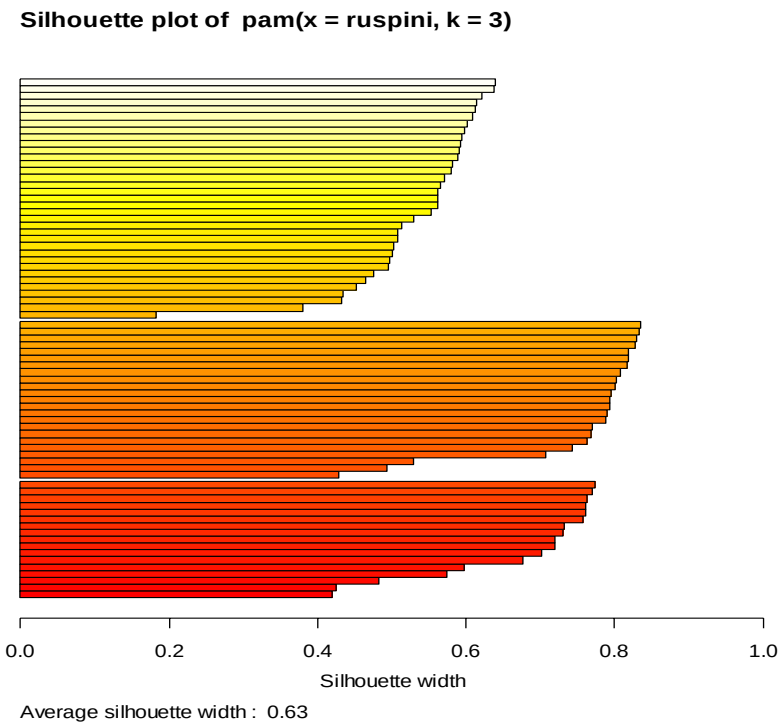
The average silhouette width is smaller with 3 clusters than with 2, suggesting that the **k=2** offers a better clustering than the **k=3**.

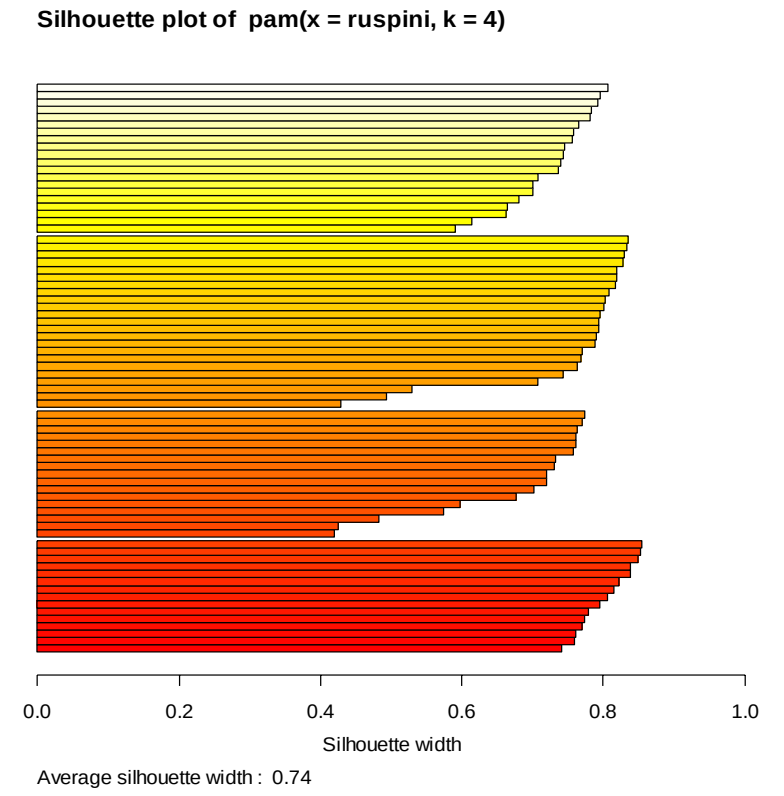
```
> # cluster the ruspini data using 4-medoids
> kdr4 <- pam(ruspini,k=4)
>
> # plot the data with different color / char for each cluster
> plot(ruspini, pch=kdr4$clustering,col=kdr4$clustering+1,xlab="x",
+ ylab="y",main="k-Medoids on Ruspini Data")
>
> # now plot the medoids of each cluster.
> points(kdr4$medoids,pch=7)
```

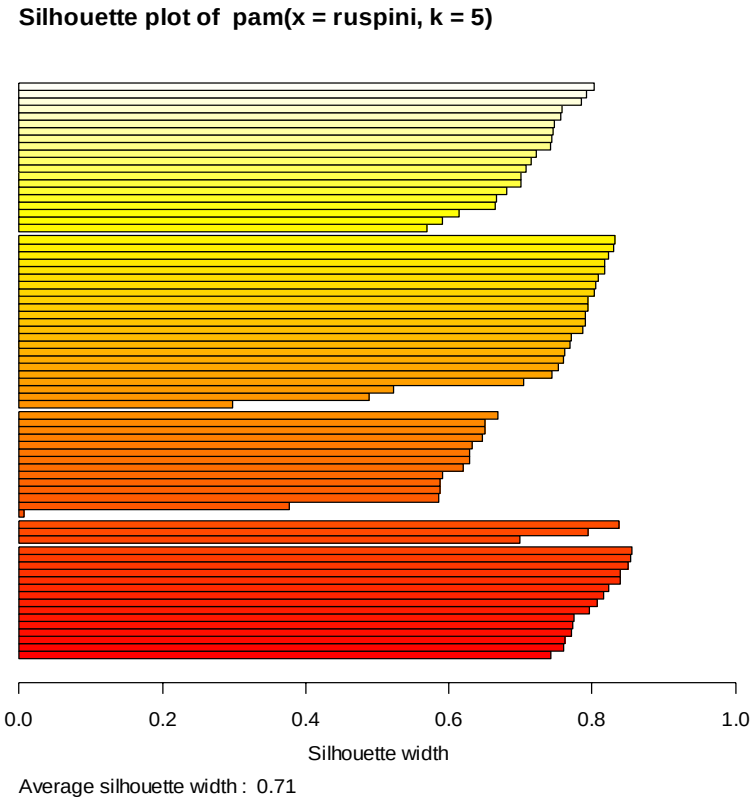



```
> # fit 3 and 5 mediods to the ruspini data
> kdr3 <- pam(ruspini,k=3)
> kdr5 <- pam(ruspini,k=5)
>
> # silhouette plots (which.plots=2) only for the three models
```

```
> plot(kdr3,which.plots=2)
> plot(kdr4,which.plots=2)
> plot(kdr5,which.plots=2)
```



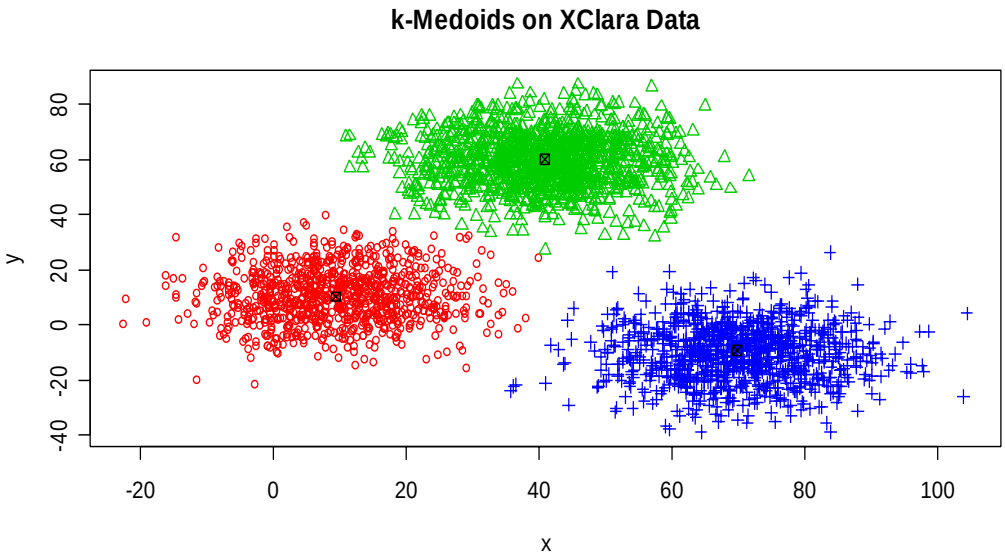




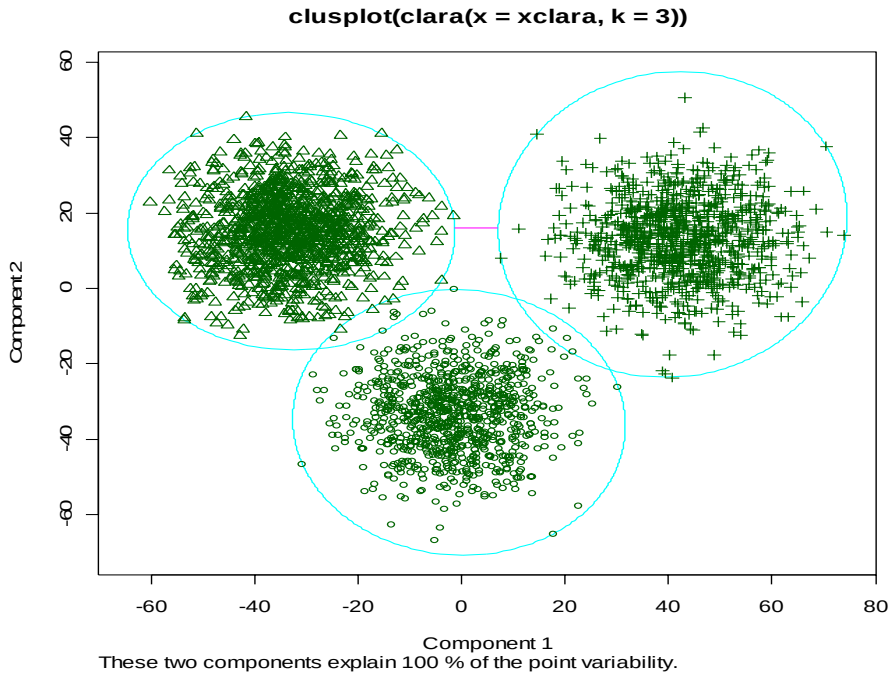
In this case, the average silhouette width is maximized for k=4, as expected. Furthermore, for k=3 and k=5, there are a number of observations with low silhouette values (< 0.4), and no such observations for k=4.

The k-medoids method clustered both of these datasets very well. Note that the pam function actually has quite a large memory requirement since it stores the whole dissimilarity matrix, which requires $O(n^2)$ memory in the number n of observations. There is a limited memory method for k-medoid clustering in R, `clara' which is probably more suitable for data mining applications. We shall apply clara to the xclara dataset.

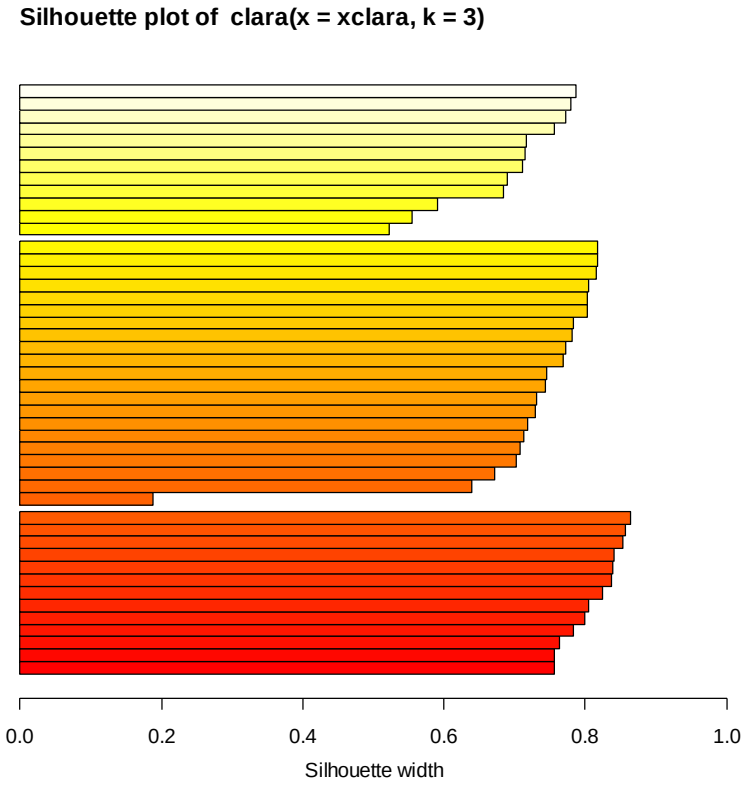
```
> kdx3 <- clara(xclara,k=3)
> kdx4 <- clara(xclara,k=4)
>
> # plot the data with different color / char for each cluster
>                                                    plot(xclara,
pch=kdx3$clustering,col=kdx3$clustering+1,xlab="x",ylab="y",main="k-
Medoids on XClara Data")
>
> # now plot the medoids of each cluster.
> points(kdx3$medoids,pch=7)
```



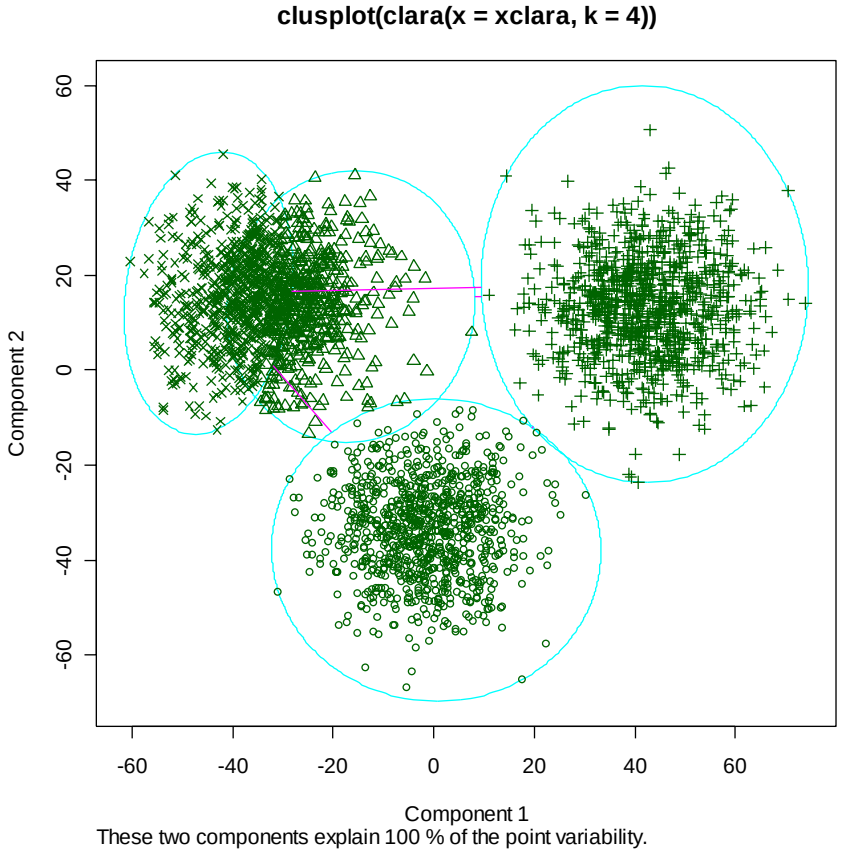

```
> # show plots using native plot methods
> plot(kdx3)
Hit <Return> to see next plot:
Hit <Return> to see next plot:
```

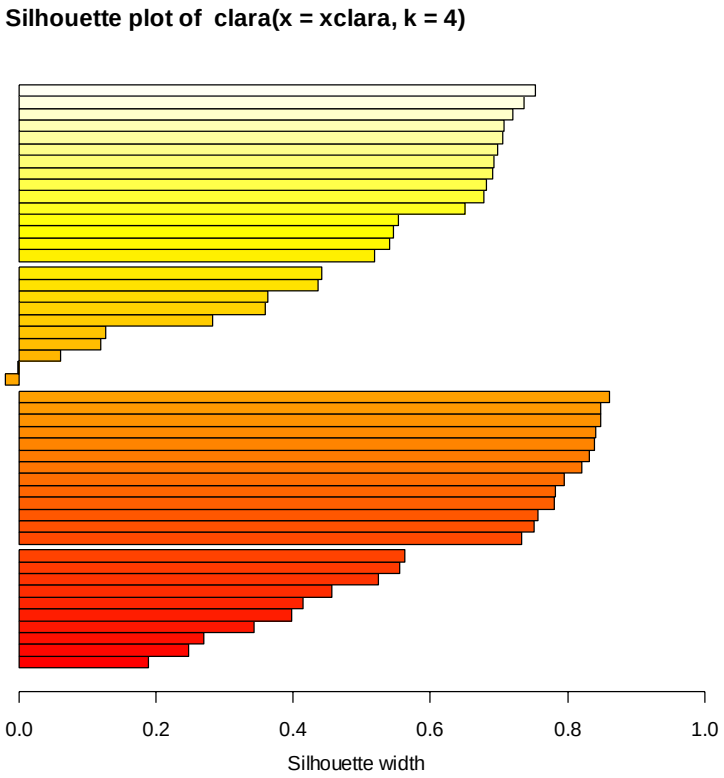


These two components explain 100 % of the point variability.




```
> plot(kdx4)
Hit <Return> to see next plot:
Hit <Return> to see next plot:
```





Once again the average silhouette width decreases with the larger number of clusters, which indicates that $k=3$ provides a better clustering structure. In both plots, the observations are summarized into histogram-style blocks rather than giving a single horizontal bar for each observation as with the smaller datasets. Note that in the second silhouette plot, there are some observations that are "misclassified" according to the silhouette value, resulting in some small negative silhouette values.

13.2.7 Fuzzy Methods

Fuzzy logic allows degrees of membership of sets. If we relax the requirement that each observation is exclusively assigned to one cluster, we can use a fuzzy clustering algorithm that divides the membership of observation i between all k clusters. The proportional membership of observation i to group v is given by u_{iv}, where $u_{iv} \geq 0$ and $\sum_{v=1}^{k} u_{iv} = 1$. Applying the relaxed conditions to the k-means algorithm leads to the *fuzzy k-means* method in which

$$\sum_{i=1}^{n}\sum_{v=1}^{k} u_{iv}^2 \left\| \mathbf{x}_i - \mathbf{m}_v \right\|^2$$

is minimized over all fuzzy partitions defined by the u. The cluster centers $\mathbf{m}_v$ are given by

$$\mathbf{m}_v = \frac{\sum_{i=1}^{n} u_{iv}^2 \mathbf{x}_i}{\sum_{i=1}^{n} u_{iv}^2}$$

Note that the use of u_{iv}^2 instead of u_{iv} as weights is not intuitive, but is nevertheless quoted in the literature. Some authors even suggest use of general weights u_{iv}^m, where $m \geq 1$ is called the *fuzzifier*.

A fuzzy equivalent to k-medoids also exists, in which

$$\sum_{v=1}^{k} \frac{\sum_{i,j} u_{iv}^2 u_{jv}^2 d(\mathbf{x}_i, \mathbf{x}_j)}{2\sum_i u_{iv}^2}$$

is minimized over fuzzy partitions determined by the u_{iv}.

Fuzzy k-medoids is implemented in R via the `fanny' function, which is in the `cluster' package. The call to fanny is identical to the call to pam, but the returned object has "fuzzy" membership to clusters. The cluster package also includes a silhouette plot method for summarizing this sort of clustering, but we shall not provide examples here. See the help documentation for examples and more details.

13.3 Methods Based on Mixtures

Suppose we believe that the examples come from a mixture of sources, and each has a parameterized density $f_g(\mathbf{x};\theta_g)$. The proportions w_g of the mixture are also unknown. For estimators $\hat{w}_g$ and $\hat{\theta}_g$ the posterior probability that observation $\mathbf{x}$ is from distribution h is simply

$$p(h \mid \mathbf{x}) = \frac{\hat{w}_h f_h(\mathbf{x};\hat{\theta}_h)}{\sum \hat{w}_g f_g(\mathbf{x};\hat{\theta}_g)},$$

and the probability of observing $\mathbf{x}$ from that mixture distribution is just

$$p(\mathbf{x}) = \sum \hat{w}_g f_g(\mathbf{x};\hat{\theta}_g).$$

Thus, for a given training set of observations, we seek to maximize the likelihood of that set over the mixture parameters:

$$\prod_{i=1}^{n} p(\mathbf{x}_i) = \prod_{i=1}^{n} \left(\sum_g \hat{w}_g f_g(\mathbf{x}_i, \hat{\theta}_g) \right).$$

This is exactly the approach we saw used in discriminant analysis, except that the distributions of the groups were assumed to be multivariate normal, and the number of groups and mixture proportions were known in advance.

In the general clustering case there are a number of methods available for finding the optimal set of parameters, including the popular *EM Algorithm.* This essentially assigns each observation to its most likely cluster with the current set of parameters. The

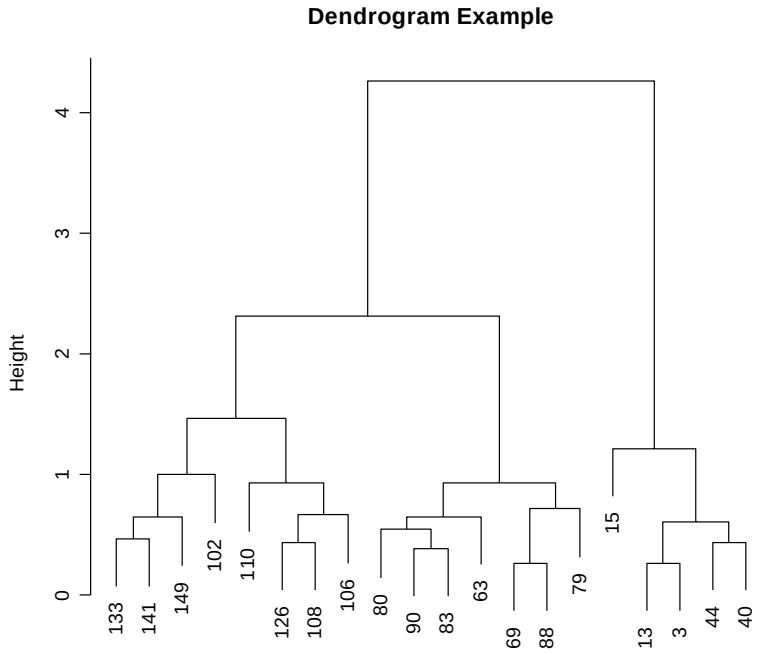
parameters for each distribution are then re-estimated from these assigned observations, and the process is iterated.

The user must choose what sorts of density functions are to be allowed (for example, Gaussian), and how to initialize the process. Initialization involves selecting the number of distributions within the mixture (the process may set some weights to 0, thus removing the distribution from the mixture), and an initial set of parameters.

Clearly this process also yields a natural classification algorithm for future observations.

13.4 Hierarchical Clustering

There are many clustering problems that can be thought of as hierarchical taxonomies. That is, the clustering falls into natural layers or hierarchies. For example, a corporate conglomerate is formed of individual companies, which are formed of sections, made up of departments, and so on. Hierarchical clustering produces a tree known as a *dendrogram*. A dendrogram can be viewed either in a top-down fashion (i.e. coarsest clusters to finer distinction), which is known as a *divisive* view. In this view clusters are seen as being split into finer subclusters. Alternatively, the bottom-up view sees clusters being merged to form coarser groups. This is the *agglomerative* view.



The example dendrogram above demonstrates how the data is presented after hierarchical clustering. Each "leaf" at the bottom of the tree represents a single observation in the dataset. In this case, the data appears to fall into 3 clusters (indeed, the dendrogram comes from clustering a random sample of 20 points of the iris dataset,

which has three classes). If the tree is constructed "bottom-up" (start with leaves and work towards the root), the algorithm is agglomerative, and if the tree is constructed "top-down" the algorithm is divisive. The height at which a split occurs is a based on the distance between the clusters (or observations). Thus the split between observations 3 and 13 in the dendrogram above is set at the lowest height in the graph. This indicates that these two observations are the closest together out of all the pairs of observations.

Hierarchical methods side-step the issue of specifying the number of clusters by presenting the user with many different partitions. These partitions are generated by cutting the tree (dendrogram) at different levels (heights or cluster distances). The tree has usually somewhat less than n levels where n is the size of the data set. Each level differs from the previous by either the agglomeration of two clusters, or the division of one cluster.

One might be tempted to think that hierarchical methods supercede partitioning methods since partitions are effectively presented for all values of k. However hierarchical methods suffer from the problem that optimal decisions (in terms of finding the best overall tree) may not be made at each level, and an erroneous decision cannot be corrected at a later stage. Essentially these methods sacrifice the benefit of more exhaustive search techniques in favor of rapid computation time.

13.4.1 Measures of Dissimilarity

To implement a hierarchical algorithm we need some notion of the dissimilarity between clusters (presuming we already have some measure $d(\mathbf{x}_i, \mathbf{x}_j)$ of the dissimilarity between observations). As usual, there are a large number of proposals for this measure, and little agreement. Some candidates are:

Average dissimilarity: $$d(R,Q) = \frac{1}{\|R\|\|Q\|} \sum_{\mathbf{x}_i \in R, \mathbf{x}_j \in Q} d(\mathbf{x}_i, \mathbf{x}_j)$$

This measure tends to be effective for data where the clusters are roughly ball or oval shaped, with reasonable distance between clusters.

Nearest neighbor (single linkage): $$d(R,Q) = \min_{\mathbf{x}_i \in R, \mathbf{x}_j \in Q} d(\mathbf{x}_i, \mathbf{x}_j)$$

This measure classifies clusters as similar if any two points in the clusters are close. This can lead to a "chaining" effect, where a string of clusters close to each other only by a single pair of observations are agglomerated to an elongated "sausage" shaped class.

Furthest neighbor (complete linkage): $$d(R,Q) = \max_{\mathbf{x}_i \in R, \mathbf{x}_j \in Q} d(\mathbf{x}_i, \mathbf{x}_j)$$

This measure tends to produce smaller, compact clusters, and is good for data that has quite compact groups which are relatively close together.

The definition of the dissimilarity of a point $\mathbf{x}$ to a cluster R extends naturally from the above by simply considering $\mathbf{x}$ as a one point cluster.

13.4.2 Agglomerative and Divisive Algorithms

Once we have a concept of dissimilarity between clusters, implementing an agglomerative algorithm is simple: choose the two clusters that are the least dissimilar at each level and merge them.

Implementing a divisive algorithm is not quite so trivial because at the root level we need to consider $2^{n-1}-1$ possible partitions of the n examples into two non-empty sets. For n of any reasonable size this quickly becomes problematic. However we may try to approximate the optimal split by any one of the partitioning algorithms mentioned previously (for example, 2-means, or 2-medoids), and continuing in a recursive fashion. The divisive algorithm used in R (see [SHR:1997]) splits the largest available cluster into two smaller clusters at each step until all clusters consist of a single point. Thus the algorithm requires $n-1$ splits on a dataset of n points. At each step, the cluster C with the largest diameter:

$$\text{diam}(C) \equiv \max_{i,j \in C} d(i,j)$$

is selected. If this diameter is not zero, then C is split into two child clusters A and B using the following algorithm described in [SHR:1997]:

1. Initially $A = C$ and $B = \varnothing$.
2. Try to move a single object from A to B. This is done by taking each object $i \in A$ and computing the average dissimilarity $a(i)$ from all other objects in A, and the average dissimilarity to the cluster B, $d(i,B)$. Select the object $h \in A$ maximizing the difference $a(h) - d(h,B)$. If this difference is positive, move h from A to B, otherwise the algorithm stops. Alternatively the algorithm stops when A has only one object remaining.

Rousseeuw ([R:1986]) introduced measures of how much clustering structure is found in the data when clustered using agglomerative or divisive clustering. The *agglomerative coefficient* is computed as follows. For each object i, let $\alpha(i)$ be its dissimilarity to the first cluster it is merged with, divided by the dissimilarity of the merger in the last step of the algorithm. The agglomerative coefficient (AC) is then defined as the average of all $1-\alpha(i)$. A large AC (close to 1) indicates that on average, observations merged with clusters (or points) near to them at the first stage, and were finally merged with a cluster far away at the last stage. This can be taken as evidence for a "clumping" structure in the data, which in turn yields good clustering. On the other hand, a small AC indicates that the distance of a point to the first cluster it is merged with is not (on average) much less than the distance to the last cluster it is merged with. This indicates a more even distribution of points and poor clustering structure.

The *divisive coefficient* (DC) is computed as follows. For each object i, let $\delta(i)$ be the diameter of the last cluster to which i belongs (before it is split off as a single object), divided by the diameter of the whole dataset. The divisive coefficient is then the average of all the $1-\delta(i)$. Once again, if points are, on average, in relatively compact clusters (compared to the diameter of the whole dataset) just

before being split off, this indicates a strong clustering structure, and will give a high DC value.

13.4.3 Examples

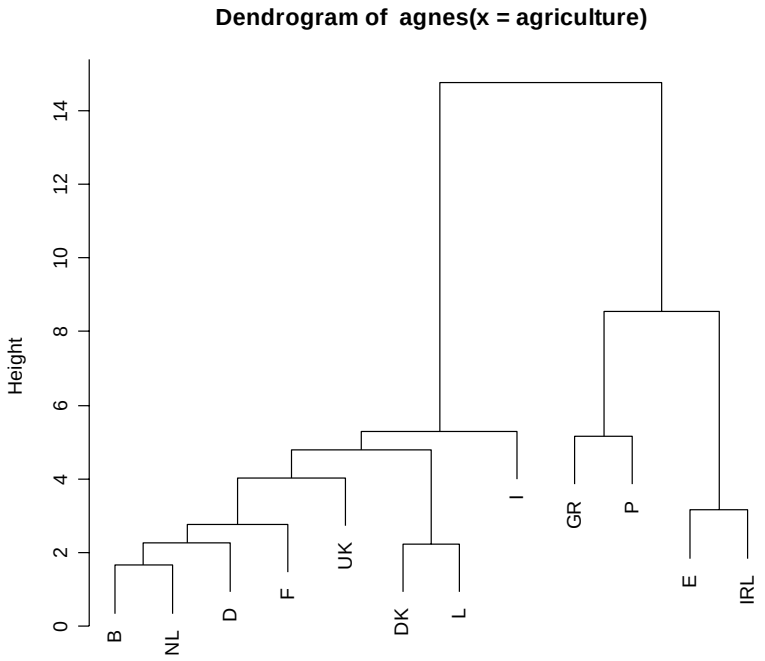
The R package `cluster' contains functions implementing agglomerative and divisive clustering. There are a number of accompanying plots and diagnostic tools to make interpretation of the fits easier. We shall simply demonstrate here some examples of each method, and leave the reader to investigate further using the help documentation and [SHR:1997]. By default these functions use a Euclidean distance and average dissimilarity. There are options for general distance measures (via a dissimilarity matrix), and for nearest and furthest neighbor methods for between cluster dissimilarity (called single and complete linkage respectively).

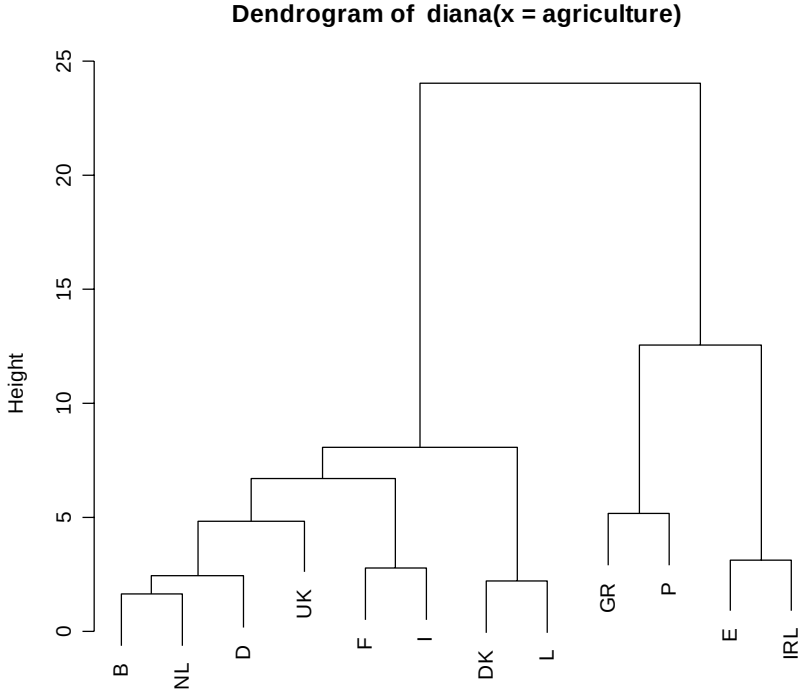
Recall that the agriculture dataset gives GNP and population percentage working in agriculture for a number of European countries. We may use both agglomerative and divisive clustering on this dataset as follows:

```
> ## agglomerative clustering (agnes)
> agagr <- agnes(agriculture)
> # plot the dendrogram
> pltree(agagr)
> # report the AC
> print(agagr$ac)
[1] 0.7818932


> ## divisive clustering (diana)
> diagr <- diana(agriculture)
> # plot the dendrogram
> pltree(diagr)
> # report the DC
> print(diagr$dc)
[1] 0.8711062
```

The dendrograms on the following page show that both methods discovered essentially the same structure. There is a group of 8 countries clustered on the left branch that are comparatively wealthy and have smaller percentages involved in agriculture. The remaining 4 countries are poorer and have a higher percentage of the population involved in agriculture. Both the AC and the DC are quite high, indicating the dataset has a strong clustering structure.

B	NL	D	UK	F	I	DK	L	GR	P	E	IRL
Belgium	Netherlands	Germany	United Kingdom	Finland	Iceland	Denmark	Luxemburg	Greece	Poland	Spain	Ireland


Dendrogram of agnes(x = agriculture)
Height
0
2
4
6
8
10
12
14
B
NL
D
F
UK
DK
L
I
GR
P
E
IRL


Dendrogram of diana(x = agriculture)
Height
0
5
10
15
20
25
B
NL
D
UK
F
I
DK
L
GR
P
E
IRL

13.4.4 Monothetic Algorithms

A special case of a divisive hierarchical algorithm is when splits are confined to binary splits on a single variable. These are called *monothetic* methods, and are very similar to the decision tree methods we previously investigated. Of course, we need to find an analogous concept to the purity measures introduced there. The obvious candidate is the dissimilarity between the two children. In this case we select the feature and the split point that maximizes the dissimilarity between the children.

If we restrict our attention to only binary variables, then monothetic methods have a further interpretation. At each split we generate clusters that differ only on the remaining variables. Thus we seek one variable whose difference most accurately reflects the difference of all. More specifically, suppose we have features f and g. Then we may draw up the corresponding comparison matrix:

	f TRUE	f FALSE
g TRUE	*a*	*b*
g FALSE	*c*	*d*

If the two features are strongly associated, then *a* and *d* will be comparatively large, and *b* and *c* will be small, (or the opposite if they are strongly negatively associated). Therefore we can summarize the *association* between f and g by means of the following association measure:

$$\text{assoc}(f,g) = \left| a(f,g)d(f,g) - b(f,g)c(f,g) \right| .$$

Clearly this will be large for strongly associated variables, and small for disassociated variables. By summing up the associations between f and all the other variables we can get an overall measure for the strength of association between f and the remaining variables. Using this measure we can choose a split variable at each stage. If some observation has an NA in variable f (say), then the NA is dealt with by simply replacing f with the observation value in the variable (say g) that has the largest association with f.

Splitting on the feature that provides the strongest association is also the goal of association analysis (market basket analysis). The connection can be seen by considering a simple market basket problem. Suppose our observations are the shopping baskets of individuals, then a monothetic hierarchical clustering will split the population on the basis of the absence or presence of a single grocery item. For example, it may be that the most significant 2-group clustering is into male and female shoppers. Then one can easily imagine that a number of products exist almost exclusively in the basket of one sex, but not in that of the other. For example, beer might prove to be an effective split product, or a makeup item. The dendrogram will then give us different levels of customer clusters with increasingly homogenous baskets.

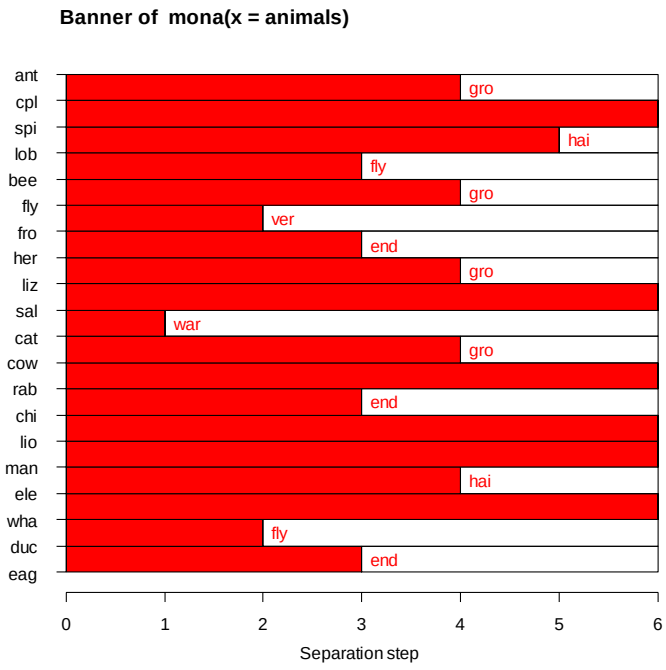
13.4.5 Example

In R monothetic hierarchical clustering is implemented via the `mona' algorithm in the `cluster' package. We shall apply this function to a

dataset of types of animals with binary features such as "warm-blooded", "flying" etc. See help(animals) for more information.

```
A data frame with 20 observations on 6 variables:
      [ , 1]  war  warm-blooded
      [ , 2]  fly  can fly
      [ , 3]  ver  vertebrate
      [ , 4]  end  endangered
      [ , 5]  gro  live in groups
      [ , 6]  hai  have hair
```

```
> data(animals)
> ## monothetic clustering (mona)
> mnan <- mona(animals)
> plot(mnan)
```



The banner plot presents the binary splits as they occur, and orders the observations so they form clusters on the vertical axis. For example, the first split feature is `war' (warm-blooded), and this splits the dataset into cold blooded animals (top half of the diagram) and warm blooded animals (bottom half of the diagram). The second split is on the variable `ver' in the cold-blooded animals, and `fly' in the warm-blooded animals, and so on. In some cases, such as `chi', `lio', 'man' there are no features to distinguish the observations so they form a cluster of three observations.

14 Combining Models

14.1 Introduction

Suppose we have an independent and identically distributed sample $\Im = \{(\mathbf{x}_n, y_n), n = 1,...,N\}$ from a distribution $P_{(\mathbf{X},Y)}$, and let $f(\mathbf{x}, \Im)$ be a model fitted to the sample. Suppose further that the model f is unbiased; that is, if $(\mathbf{X},Y)$ is a random variable with distribution $P_{(\mathbf{X},Y)}$, then $E(f(\mathbf{X}, \Im)) = E(Y)$. The techniques we have examined so far in this course have concentrated on finding a single, "optimal" model f for the training data given. In many cases, it is beneficial to consider *multiple* models, and seek a combination of these that will give a better, or at least more stable, solution.

14.1.1 Multiple Independent Training Sets

If we have an abundance of data (as is often the case in data mining applications), then we may find that our algorithms are simply not able to train on the whole dataset, or even a sizeable portion of the dataset. Either the memory requirement, or computation requirements (or both) make it infeasible to fit a model in reasonable time. On the other hand, we could draw an independent sequence of training sets of some feasible size N, and fit a model to each training set. That is, for independent training sets $\Im_1,...,\Im_T$ drawn from $P_{(\mathbf{X},Y)}$, we fit models $f_1(\mathbf{x}),..., f_T(\mathbf{x})$. Can we use this multiplicity of models to create a new model that will be better (at least on average) than the individual models?

If the response y is numerical, then the obvious thing to do is simply average over the predictions of each model. That is, for a given input $\mathbf{x}$, the prediction of the aggregated model, F is given by:

$$F(\mathbf{x}) = \frac{1}{T}\sum_{t=1}^{T} f_t(\mathbf{x}) .$$

This quantity clearly converges to the expectation over the distribution $P_{\Im}$ of the iid sample $\Im$ drawn from $P_{(\mathbf{X},Y)}$:

$$F(\mathbf{x}, P) = E_{\Im} f(\mathbf{x}, \Im) .$$

On the other hand, if y is categorical, then the combined prediction can simply be the category with the largest vote among the constituent models.

Let $(\mathbf{X},Y)$ be a random variable with distribution $P_{(\mathbf{X},Y)}$, and independent from the training set $\Im$. If we define prediction error in terms of the average squared error, then the prediction error e for a single model is given by:

$$\begin{aligned} e &= E_{\Im}(E_{(\mathbf{X},Y)}(Y - f(\mathbf{X}, \Im))^2) \\ &= E_{(\mathbf{X},Y)}(E_{\Im}(Y - f(\mathbf{X}, \Im))^2) \\ &= EY^2 - 2(EY)E_{(\mathbf{X},Y)}E_{\Im} f(\mathbf{X}, \Im) + E_{(\mathbf{X},Y)}E_{\Im}(f^2(\mathbf{X}, \Im)). \end{aligned}$$

On the other hand, the average prediction error e_A for the aggregated model is given by:

$$e_A = E_{(\mathbf{X},Y)}(Y - F(\mathbf{X},P))^2 .$$

Using the inequality $(EZ)^2 \le EZ^2$, we can simplify the final term in the expression for e to:

$$E_{(\mathbf{X},Y)}E_{\Im}(f^2(\mathbf{X},\Im)) \ge E_{(\mathbf{X},Y)}(E_{\Im}f(\mathbf{X},\Im))^2 = E_{(\mathbf{X},Y)}F^2(\mathbf{X},P)$$

and thus $e_A \le e$.

Therefore the aggregated model always has lower mean squared error than that ***expected*** of a single model. How much lower will depend on how coarse the inequality above is. This is essentially a measure of how stable the models are. If the individual models f are quite stable over different training sets $\Im$, then the variance over the population of training sets will be small. That is, the difference

$$E_{\Im}(f^2(\mathbf{X},\Im)) - (E_{\Im}f(\mathbf{X},\Im))^2$$

is small, and hence the aggregated model will not contribute much improvement over the individual models. On the other hand, if the above variance is large (implying less stable models), then a correspondingly large improvement will be seen.

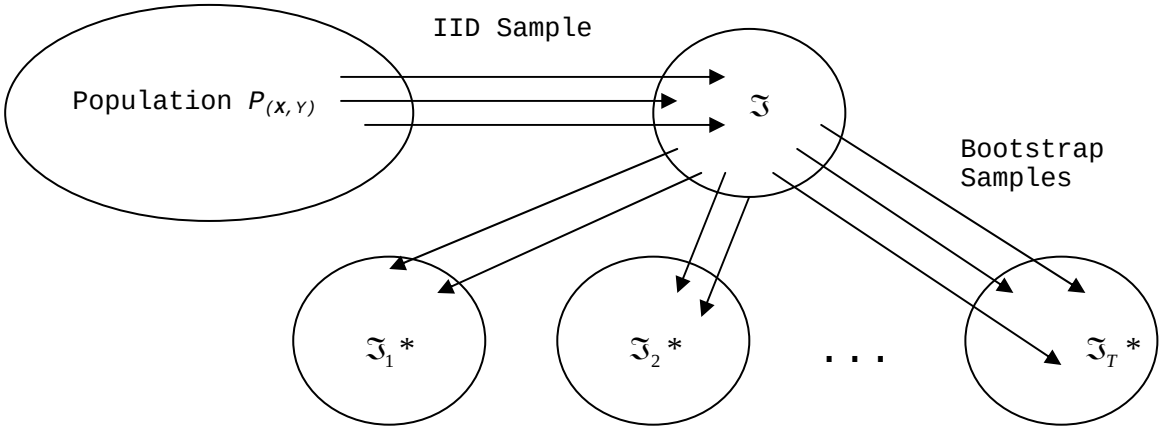
14.2 Bootstrap Samples and Bagging

[ET:1993], [B:1994]

Bagging (**B**ootstrap **Agg**regat**ing**) is a systematic method of training and combining models using a *bootstrap sample* of the training data. The technique is very simple to implement and can significantly reduce the generalization error and stability of a model.

14.2.1 Bootstrap Samples

Suppose once again we have a data set $\Im = \{(\mathbf{x}_n, y_n), n = 1,...,N\}$ drawn from a distribution $P_{(\mathbf{X},Y)}$. A bootstrap sample of $\Im$ is a sample of size $\|\Im\| = N$ drawn uniformly from $\Im$ *with replacement.* We shall represent a bootstrap sample of $(\mathbf{x}, y)$ by $\Im^* = \{(\mathbf{x}_1, y_1)^*, \ldots, (\mathbf{x}_N, y_N)^*\}$.



14.2.2 Bootstrap Estimates

Let $\theta(P_{(\mathbf{X},Y)})$ be some quantity of interest to us (for example, mean, variance, etc), and let θ be estimated from the sample data. To use the estimate $\hat{\theta}$ effectively we need some measure of how good it is. Specifically, we would like to know the *standard error* of the estimate, and the *bias* of the estimate.

- Mean Squared Error $\mathrm{E}(\hat{\theta}-\mathrm{E}(\hat{\theta}))^2$ = (Standard Error)^2
- Bias $\mathrm{E}(\hat{\theta})-\theta$

Of course, to compute the bias and standard error exactly we would need to know the distribution $P_{(\mathbf{X},Y)}$. However, we may use bootstrap samples to estimate these quantities. The intuition is simple – if the estimator $\hat{\theta}$ has some bias $E(\hat{\theta})-\theta$, then we might reasonably expect an associated bias to occur in a *bootstrap replication* of $\hat{\theta}$. That is, for a bootstrap sample $\mathfrak{I}^*$ of $\mathfrak{I}$, we can expect some correspondence between the quantity $E^*(\hat{\theta}^*)-\hat{\theta}$ and the bias of $\hat{\theta}$. Here E^* denotes expectation with respect to all possible bootstrap samples from the original sample $\mathfrak{I}$.

To compute bias and standard error estimates using the bootstrap, we begin with B independent bootstrap samples $\mathfrak{I}_1^*,\mathfrak{I}_2^*,\ldots,\mathfrak{I}_B^*$, each consisting of N data values drawn with replacement from $\mathfrak{I}$. For each bootstrap sample, we compute the associated bootstrap replication $\hat{\theta}_b^* = \hat{\theta}(\mathfrak{I}_b^*)$, $t=1,2,\ldots,B$. Then the bootstrap estimate of the bias is given by

$$\widehat{\mathrm{bias}}_{Boot} = \hat{\theta}_{Boot} - \hat{\theta}, \text{ where } \hat{\theta}_{Boot} = \sum_{b=1}^{B} \frac{\hat{\theta}(\mathfrak{I}_b^*)}{B}.$$

The bootstrap estimate of the standard error is given by sample standard deviation of the B bootstrap replications

$$\widehat{\mathrm{se}}_{Boot} = \left\{ \sum_{b=1}^{B} \frac{(\hat{\theta}(\mathfrak{I}_b^*) - \hat{\theta}_{Boot})^2}{(B-1)} \right\}^{\frac{1}{2}}.$$

14.2.3 Bagging

In many cases we are not able to generate multiple independent training sets for combined models, however we *can* generate multiple bootstrap samples from our given sample $\mathfrak{I}$. A separate model may be fitted to each bootstrap sample, and the resulting models combined as detailed in the previous section. This process is known as *bagging*. Despite the simplicity of the approach, bagging can often lead to remarkable increases in model accuracy. Breiman ([B:1994]) reports reduction in misclassification rates (for classification problems) ranging from 20% to 47%, and improvements of a similar magnitude for regression problems.

Following the analysis in the previous section it is clear that bagging is only effective when the variance between models fitted to different bootstrap samples is large. Furthermore, whereas combining models fitted on independent datasets is guaranteed to not degrade on average with the addition of more datasets, the same cannot be said of bagging. In particular, if we attempt to fit a stable model using bagging, we cannot guarantee that the bagged predictor

$$f_{Bag}(\mathbf{X}) = \frac{1}{B}\sum_{b=1}^{B} f(\mathbf{X}, \Im_b^*)$$

will be as accurate for data drawn from $P_{(\mathbf{X},Y)}$ as the original predictor $f(\mathbf{X}, \Im)$. Therefore, as with combining models fitted on multiple independent datasets, we use base predictors f that have high variance over different training sets.

Bagging is an easy procedure to implement, but can offer substantial improvements in predictor accuracy. Furthermore, it does not make strong assumptions on the error distributions, and makes good use of existing data. On the other hand, apart from possible degradation of prediction if applied to a stable model, bagging also reduces the interpretability of the individual models (as do most combined model methods). Clearly extra computation time is also required, which may be an issue with larger datasets.

Breiman, [B:1994], gives a "rule of thumb" for the number of bootstrap replications necessary for an effective bagging model. In the case of classification datasets, 50 replications are recommended, and 25 for regression datasets. In the case of models that are computationally expensive to fit, such as neural networks, it may be desirable to use less bootstrap replications.

14.2.4 Example

There is no standard function for bagging in R, however it is simple to write one. The following code compares a bagged rpart predictor for the glass dataset to a single rpart predictor. The central loop is repeated 100 times to give a more accurate estimate of the average performance of the two methods. At each iteration of the loop, 20 observations from the dataset (about 10% of the data) are set aside as a test set, and the remaining observations are used for training. Following Breiman's recommendation, 50 bootstrap replications are used to fit the bagged predictor in this classification example.

```
library(MASS)
library(rpart)
data(fgl)

nobs <- dim(fgl)[1]

# control vector for original RPART tree, before pruning
ctrl <- rpart.control(cp=0,minsplit=2,minbucket=1)

# formula
predi <- charmatch("type",names(fgl))
fmla <- as.formula(paste("type~",paste(names(fgl)[-predi], collapse="+")))

# levels
levs <- levels(fgl[,"type"])

# bagging models
err0 <- NULL
berr <- NULL
```

```
bagit <- 100
for ( bit in 1:bagit)
{
# split into training and test (use Breiman's test)
tstidx <- sample(1:nobs,20)
trnidx <- (1:nobs)[-tstidx]

# initial model
tree0 <- rpart(fmla, fgl[trnidx,],control=ctrl)
cptab <- tree0$cptable
minidx <- order(cptab[,"xerror"])[1]
tree0 <- prune.rpart(tree0,cp=cptab[minidx,"CP"])
err0 <- c(err0,
 sum(predict(tree0,fgl[tstidx,],type="class")!=fgl[tstidx,"type"])/20)

# now do the 50 bootstrap models
predmat <- matrix(0,ncol=length(levs),nrow=20)

for (bootit in 1:50)
{
boottrn <- sample(trnidx,length(trnidx),replace=T)
tree<- rpart(fmla, fgl[boottrn,],control=ctrl)
cptab <- tree$cptable
minidx <- order(cptab[,"xerror"])[1]
tree<- prune.rpart(tree,cp=cptab[minidx,"CP"])
pred<-predict(tree,fgl[tstidx,],type="vector")
for ( i in 1:length(pred) ) {

# add 1 to the class column for each positive prediction
      predmat[i,pred[i]] <- predmat[i,pred[i]]+1

} # end of loop – on i
} # end of loop – 50 bootstrap models

bootpred <- NULL
# Now determine the "type" with the most votes for each case in the
# test set
for (tsti in 1:20) {
      # select the column with the maximum count for each observation
       bootpred <-c(bootpred, rev(order(predmat[tsti,]))[1])

} # end of loop – determining "type" for each test of 20 test cases

berr <- c(berr,sum(levs[bootpred]!=fgl[tstidx,"type"])/20)

} # end of loop – 100 different choices of test and learning sets

# mean error of single method and bagging method
mean(err0)   # single model method
mean(berr)   # bagging method
```

This function gave an average bagging misclassification rate of 0.256, compared to a misclassification rate of 0.313 using the standard model. This corresponds to a 18% reduction in the misclassification rate.

14.3 Boosting

[S:2002]

Boosting is a learning technique based on an intuitive idea of reinforcement learning. That is, you do the best you can on a set of problems, and then focus on those you got wrong, and so on. For example, suppose we have a classification problem and we fit some model that does only slightly better than random guessing. Then we can re-weight the data set, adding more weight to those examples on which the first model misclassified. This process continues until we have a sequence of classifiers. These can then be combined in some way to a classifier that is expected to perform much better than any individuals in the sequence.

More formally, suppose we have classification data $\{(\mathbf{x}_1, y_1),\ldots,(\mathbf{x}_n, y_n)\}$ where $\mathbf{x}_i$ is a vector of input features, and $y_i \in \{-1,1\}$ is the class of the observation. Let $\{D_k(1),\ldots,D_k(n)\}$ be the weights given to each observation at iteration k. The weights $D_1(i)$ are initialized to $1/n$. For each iteration we train a new classifier $h_k : \mathrm{X} \mapsto \Re$, and the classification is given by $\mathrm{sign}(h_k(\mathbf{x}))$. Then in a typical boosting algorithm iteration k proceeds as follows:

1. Train a new classifier h_k using the training data weighted by D_k.
2. Choose a "classifier weight" α_k. A suggested value is $\alpha_k = \frac{1}{2}\ln\left(\frac{1-\varepsilon_k}{\varepsilon_k}\right)$ where ε_k is the estimated misclassification rate of the classifier on the weighted input distribution.
3. Update the weights according to $D_{k+1}(i) = \frac{D_k(i)\exp\left(-\alpha_k y_i h_k(\mathbf{x}_i)\right)}{Z_k}$ where $Z_k = \sum_{i=1}^{m} D_k(i)\exp\left(-\alpha_k y_i h_k(\mathbf{x}_i)\right)$ is a normalization factor.

After K iterations the classifier is given by:

$$H(\mathbf{x}) = \mathrm{sign}\left(\sum_{k=1}^{K}\alpha_k h_k(\mathbf{x})\right).$$

Note that this algorithm doesn't depend on the form of the incremental classifiers h_k, so can be implemented as a complement to any existing classification algorithm. Furthermore, the method can be easily extended to regression problems with appropriate modifications to the weights update and the combined classifier expression.

Boosting techniques claim quite impressive generalization ability, as well as efficient, simple training. Some of the literature does suggest that choosing overly complex incremental classifiers may lead to an overfitted model, thus the user is encouraged to use very simple classifiers.

14.3.1 Example

To implement straight AdaBoost, we require a two sample classification problem. In this case we shall use the two largest classes in the forensic glass dataset.

```
# need a two sample problem here - use largest two glass types
tp <- fgl[,"type"]
fgl2 <- fgl[tp==levs[1] | tp==levs[2],]
nobs2 <- dim(fgl2)[1]
levs2 <- levels(fgl2[,"type"])
#fgl2[,"type"] <- 2*(as.numeric(fgl2[,"type"])-1.5)

err0 <- NULL
berr <- NULL

booit <- 100
for ( bit in 1:booit)
```

```
{
# split into training and test (use Breiman's test)
tstidx <- sample(1:nobs2,15)
trnidx <- (1:nobs2)[-tstidx]

# initial model
tree0 <- rpart(fmla, fgl2[trnidx,],control=ctrl)
cptab <- tree0$cptable
minidx <- order(cptab[,"xerror"])[1]
tree0 <- prune.rpart(tree0,cp=cptab[minidx,"CP"])
err0 <- c(err0,
 sum(predict(tree0,fgl2[tstidx,],type="class")!=fgl2[tstidx,"type"]))

# now do the boosting models

T <- 10

# boosting prediction
bpred <- rep(0,length(tstidx))

# initial weights
wts <- rep(1/length(trnidx),length(trnidx))

for (t in 1:T)
{
# use the default tree model
tree<- rpart(fmla, fgl2[trnidx,],weights=wts)
ptrn <- predict(tree,fgl2[trnidx,],type="class")
epsilon <- sum((ptrn!=fgl2[trnidx,"type"])*wts)/sum(wts)
alpha <- 0.5 * log((1-epsilon)/epsilon)

# predict from this tree
pred <- as.character(predict(tree,fgl2[tstidx,],type="class"))
pred[pred==levs2[1]] <- -1
pred[pred==levs2[2]] <- 1

bpred <- bpred + (alpha*as.numeric(pred))

# update the weights
wts <- wts * exp(alpha * ((ptrn!=fgl2[trnidx,"type"]) -
(ptrn==fgl2[trnidx,"type"])))
wts <- wts / sum(wts)
}

bpred <- sign(bpred)
berr <- c(berr,sum((bpred==1) & (fgl[tstidx,"type"]==levs2[1])) +
sum((bpred==-1) & (fgl[tstidx,"type"]==levs2[2])))

}

# mean error of single model method and boosting method
mean(err0)   # single model method
mean(berr)   # boosting method
```

This gave an average misclassification rate of 0.135 for the boosting model, compared with 0.221 for the method using only a single model.

15 Other Topics

15.1 The Bootstrap and Jackknife Estimates of Error and Bias [ET:1993]

Suppose we have some data set $\mathbf{x} = \{x_1, \ldots, x_n\}$ drawn from a distribution F. Let $\theta = t(F)$ be some summary statistic of interest to us (for example, mean, variance, etc). θ is estimated from the sample data by $\hat{\theta} = s(\mathbf{x})$, where s is some function estimating the required quantity. Note that $s(\mathbf{x})$ may be the plug-in estimate $t(\hat{F})$, where $\hat{F}$ is the empirical distribution generated from our sample data, but is not necessarily so. To use the estimate $\hat{\theta}$ effectively we need some measure of how good it is. Specifically, we would like to know the *standard error* or *mean squared error* of the estimate, and the *bias* of the estimate.

- Mean Squared Error $\mathrm{E}(\hat{\theta} - \mathrm{E}(\hat{\theta}))^2$
- Bias $\mathrm{E}(\hat{\theta}) - \theta$

Two popular methods for estimating these quantities are *the jackknife* and *the bootstrap*.

15.1.1 Jackknife Estimates

The jackknife technique makes use of the i^{th} *jackknife samples*, $i = 1, \ldots, n$ of the data set $\mathbf{x}$, which are the samples with i^{th} observation removed:

$$\mathbf{x}_{(i)} = (x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n).$$

Then $\hat{\theta}_{(i)} = s(\mathbf{x}_{(i)})$ is the i^{th} *jackknife replication* of $\hat{\theta}$.

The jackknife estimate of bias is defined by:

$$\widehat{\mathrm{bias}}_{\mathrm{jack}} = (n-1)(\hat{\theta}_{(\cdot)} - \hat{\theta}) \text{ where } \hat{\theta}_{(\cdot)} = \sum_{i=1}^{n} \frac{\hat{\theta}_{(i)}}{n}.$$

The jackknife estimate of the standard error is given by:

$$\widehat{\mathrm{se}}_{\mathrm{jack}} = \left[\frac{n-1}{n} \sum (\hat{\theta}_{(i)} - \hat{\theta}_{(\cdot)})^2 \right]^{\frac{1}{2}}.$$

15.1.2 Bootstrap Estimates

Bootstrap methods depend on the notion of a *bootstrap sample*. A bootstrap sample of $\mathbf{x}$ is a sample of size $\|\mathbf{x}\| = n$ drawn uniformly from $\mathbf{x}$ *with replacement.* We shall represent a bootstrap sample of $\mathbf{x}$ by $\mathbf{x}^* = (x_1^*, \ldots, x_n^*)$. Corresponding to a bootstrap sample is a *bootstrap replication* of $\hat{\theta}$, $\hat{\theta}^* = s(\mathbf{x}^*)$.

To compute bias and standard error estimates using the bootstrap, we begin with B independent bootstrap samples $\mathbf{x}^{*1}, \mathbf{x}^{*2}, \ldots, \mathbf{x}^{*B}$, each consisting of n data values drawn with replacement from $\mathbf{x}$. For each bootstrap sample, we compute the associated bootstrap replication $\hat{\theta}^*(b) = s(\mathbf{x}^{*b})$, $b = 1,2,\ldots,B$. Then the bootstrap estimate of the bias is given by

$$\hat{\text{bias}}_B = \hat{\theta}^*(\cdot) - t(\hat{F}), \text{ where } \hat{\theta}^*(\cdot) = \sum_{b=1}^{B} \frac{\hat{\theta}^*(b)}{B} = \sum_{b=1}^{B} \frac{s(\mathbf{x}^{*b})}{B}.$$

The bootstrap estimate of the standard error is given by sample standard deviation of the B bootstrap replications

$$\hat{\text{se}}_B = \left\{ \sum_{b=1}^{B} \frac{(\hat{\theta}^*(b) - \hat{\theta}^*(\cdot))^2}{(B-1)} \right\}^{\frac{1}{2}}.$$

15.1.3 Application to Predictive Models

Let us return now to predictive models and consider what application the bootstrap and jackknife techniques may have there. Recall that for a predictive model we have data $\{(\mathbf{x}_1, \mathbf{y}_1), (\mathbf{x}_2, \mathbf{y}_2), \ldots, (\mathbf{x}_n, \mathbf{y}_n)\}$ drawn from the joint distribution of random variables (X, Y). For a regression problem, we model the observations of Y (say) as responses to the observations of predictor X. Using this we fit a model $f(X)$ which estimates the random variable $E(Y \mid X)$.

Suppose there is some value $\mathbf{x}_0$ of X which is of interest. Then the point estimate of the expectation $E(Y \mid X = \mathbf{x}_0)$ given by $f(\mathbf{x}_0)$ is a summary statistic from the conditional distribution $Y \mid X$. We may estimate the bias and standard error of this quantity using Jackknife or Bootstrap.

15.2 Support Vector Machines

[KM:2002]

Consider two-class data (classes -1, +1), with feature vectors $\mathbf{x} \in \mathrm{X} \subseteq \Re^n$. Then we may write a linear classifier as a decision function $f(\mathbf{x}) : \mathrm{X} \mapsto \Re$, given by

$$f(\mathbf{x}) = < \mathbf{w}, \mathbf{x} > + b,$$

where $\mathbf{w} \in \Re^n$ is a weight vector, and an observation is classified by $h(\mathbf{x}) = sign(f(\mathbf{x}))$.

A simple learning rule for this model is to update the weights at iteration *k* according to

$$\text{if } y_i(< \mathbf{w}_k, \mathbf{x}_i > + b) \le 0, \text{ then } \mathbf{w}_{k+1} \leftarrow \mathbf{w}_k + \eta y_i \mathbf{x}_i$$

where $\eta \ge 0$ is a user defined learning parameter.

This is in fact the *perceptron* learning algorithm. Note that at each iteration, the weight vector may be represented as a linear combination of the terms $y_i \mathbf{x}_i$. Therefore, once learning is concluded,

we have a weight vector of the form $\mathbf{w} = \sum \alpha_i y_i \mathbf{x}_i$, where α_i is the appropriate multiple of η. Note that the weights only include observations $\mathbf{x}_i$ that are "informative" in the sense that they are highlighted by a classification error at some stage of the training.

Since we know the form of the weights, we may rewrite the decision function as

$$f(\mathbf{x}) = \sum \alpha_i y_i < \mathbf{x}_i, \mathbf{x} > + b$$

and the update rule as

if $y_i(\sum \alpha_j y_j < \mathbf{x}_j, \mathbf{x}_i > + b) \leq 0$, then $\alpha_{i+1} \leftarrow \alpha_i + \eta$.

This is known as the *dual representation* of the problem. Note that the only occurrence of elements from the feature space X, both in the decision function and the update rule, occur inside the dot product brackets.

Of course, the perceptron is only a linear function and as such is restricted to classifying linearly separable classes. However, as with other linear methods, we may wish to enrich the feature space with non-linear functions of the features, and feature interactions. In particular, suppose

$$\phi(\mathbf{x}) : \mathrm{X} \mapsto \mathrm{H}$$

is some nonlinear operator mapping to a higher dimensional space H. Then by fitting a linear classifier to $\phi(\mathbf{x})$, we may be able to capture some of the non-linear structure of the dataset, and obtain a more powerful classifier. However, there are two major problems with this approach:

1. Higher dimensional feature vectors can introduce a much higher computational load.
2. Mapping a dataset in a low dimensional space to a higher dimension leads to a sparser coverage of the higher dimensional space (the *curse of dimensionality*). This, in turn, leads to models that don't tend to generalize well.

The use of *kernels* is a resolution to these problems that still allows access to the added flexibility of non-linear functions. In kernel methods we seek a decision function of the form

$$f(\mathbf{x}) = \sum \alpha_i y_i < \phi(\mathbf{x}_i), \phi(\mathbf{x}) > + b,$$

and (if using the perceptron algorithm), update rule

if $y_i(\sum \alpha_j y_j < \phi(\mathbf{x}_j), \phi(\mathbf{x}_i) > + b) \leq 0$, then $\alpha_{i+1} \leftarrow \alpha_i + \eta$.

Once again we note that these expressions only involve the inner product of the higher dimensional feature vectors. Thus we needn't know what ϕ is at all, only the value of $k(\mathbf{x}, \mathbf{z}) = < \phi(\mathbf{x}), \phi(\mathbf{z}) >$ for all $\mathbf{x}, \mathbf{z} \in \mathrm{X}$.

The function k is an example of a *kernel function*. Examples of simple kernel functions are

1. Polynomial kernels: $k(\mathbf{x}, \mathbf{z}) = < \mathbf{x}, \mathbf{z} >^d$.
2. Gaussian kernels: $k(\mathbf{x}, \mathbf{z}) = \exp(-\|\mathbf{x} - \mathbf{z}\|^2 / 2\sigma)$.

Of course, there are a large number of possible kernel functions, the suitability of each depending on the problem at hand. In broad terms, a function k may be admitted as a kernel function if

$$\forall \mathbf{x}, \mathbf{z} \in \mathrm{X} \qquad k(\mathbf{x}, \mathbf{z}) = k(\mathbf{z}, \mathbf{x}) \qquad \textit{(symmetry)}$$

and $\forall \mathbf{x}_1, \ldots, \mathbf{x}_m \in \mathrm{X}, \quad \forall a_1, \ldots, a_m \in \Re,$

$$\sum_{i,j} a_i a_j k(\mathbf{x}_i, \mathbf{x}_j) \geq 0 \qquad \textit{(positive definiteness)}$$

A *support vector machine* is simply a linear learning machine that

1. has a dual representation (i.e. the only occurrences of feature vectors is in a dot product)
2. operates in a kernel induced feature space.

Support vector machines have quickly become very popular tools for both classification and regression. The model stems from a well understood foundation in statistical learning theory, and the models tend to have very attractive training and generalization characteristics. The weights in the linear part of the classifier are defined by the set of *support vectors*, which are those observations misclassified at some point in the training. There are much more sophisticated training methods available, but they are all based on the simple ideas mentioned above.

15.3 Genetic Algorithms
[GC:2000]

Genetic algorithms are based on the evolutionary concept that through random mutations, and selective breeding, any population will adapt itself to better suit environmental pressures. They are broadly applicable stochastic search & optimization techniques that don't tend to suffer from the local minima problems of many other methods of optimization. In general genetic algorithms make few assumptions on the structure of the data set, and are simple to program and execute. On the other hand, they tend to converge very slowly and require a quite high computational overload.

A genetic algorithm for solving a particular problem (of any general format suitable for optimization) has 5 basic components:

1. A genetic representation of solutions to the problem.
2. A way to create an initial population of solutions.
3. An evaluation function rating solutions in terms of their fitness.
4. Genetic operators (mutation & crossover) that alter the genetic composition of children during reproduction.
5. Values for the parameters of the genetic algorithm.

Genetic algorithms maintain a population $P(t)$ of individuals for each generation t. A generation corresponds to an iteration of the genetic algorithm. Furthermore, each individual $\mathbf{x}_i^{(t)}$ has an associated *fitness* $f(\mathbf{x}_i^{(t)})$. The individuals each supply a solution to the problem, and the fitness measure evaluates the accuracy of that solution.

At each generation some individuals produce offspring by means of *mutation*, which describes changes within a single solution, and *crossover*, which is the creation of a new individual from parts of two parents. The offspring from generation t are referred to by $C(t)$. At the conclusion of each generation, the fitness measures of all individuals in both $P(t)$ and $C(t)$ are used to select a new population, $P(t+1)$.

```
begin
t ← 0
initialize P(0)
evaluate P(0)
while (stopping condition not satisfied)
{
      recombine P(t) via mutations and crossovers to yield C(t)
      evaluate C(t)
      select P(t+1) from P(t) & C(t)
      t ← t+1
}
end
```

We have discussed the fitness evaluation process many times under the name of scoring functions. As before, the fitness function should be chosen to accurately reflect the fitting characteristics we desire. The stopping criteria will also be similar to "stop-training" criteria we have already encountered. For example, we may stop when the "best" individual has a training or validation error lower than some threshold. Alternatively we may stop when the improvement in training error is below some minimal increment. Most algorithms will also have a bound on the number of generations allowed before stopping.

The size of each generation is an issue in genetic algorithms. For example, we may want to allow the population to grow, changing the selection criteria for mutations and crossovers concurrently to reflect some intuition about population dynamics. However, larger populations can become a computational problem, in particular if the process of selecting pairs for crossovers requires evaluating some function on all possible pairs. This can be addressed by (for example), only allowing crossovers between individuals with fitness over a certain threshold. Alternatively we may rank individuals according to their fitness and allow crossovers between individuals whose rank differs by at most some number. Mutations don’t tend to pose the same combinatorial problems, but we may still chose to select for mutations based on some fitness measure.

15.3.1 Genetic Encodings

In order to apply a genetic algorithm to a particular problem, we need to find some encoding of the solution space that will be amenable to the genetic operators. In general, this encoding will take the form of a data structure with numerical entries. For example, suppose a linear model is proposed, then we may describe each possible solution as a vector of the model parameters. In general the encoding will take the form of a vector, matrix, tree or some other data structure. Ideally an encoding should fulfill the following requirements:

1. Non-redundant. Every possible individual in the encoding should correspond to a unique solution to the problem, and every solution should be represented by exactly one of the possible individuals in the encoding space.
2. Legitimate. Essentially we don't want to be able to arrive at any individuals in the encoding space that correspond to "illegal" solution, from legitimate operation on a legitimate population. The easiest way to do this is to have an encoding in which any permutation of the encoding units on any individual will lead to another legitimate individual in the encoding space. For example, if we have an encoding in which

some of the elements are integral and some are continuous, then a permutation may lead to an illegitimate individual.
3. Independent of Parent Context. Suppose an encoding involves some element which is an index of some parameter set held by the parent and not known by any children. This encoding is dependent on the parent context, and transmitting some or all of it to a child is meaningless.
4. Small variations in the encoding should imply small variations in the solution.

15.3.2 Genetic Operators

The genetic operators of mutation and crossover will clearly be specific to the encoding chosen. However it is possible to get a general understanding of the sorts of operators used. Mutations take place within an individual's encoding. These can be in the form of additions, deletions, switches and changes. An addition operation takes a null (or 0) entry and assigns a random (valid) non-zero number. A deletion nullifies an existing non-zero entry. A switch transposes two entries, and a change is any other random change to an entry. In more complex structure, such as trees, we could define a mutation which destroys a branch of the tree below a selected node, or transposes sections of the tree, or simply copies sections of the tree to other nodes.

As noted before, the issue of choosing two individuals for crossover reproduction can be non-trivial in large populations. However once we have selected two individuals, the operation can be quite simple. We need to create a child from the parent encodings. In the case of fixed length encodings we can simply make a random selection for each element (or logical block of elements, if that makes more sense) from the two parent candidates. A more dynamic data structure, such as a tree, will require a problem specific algorithm.

15.3.3 Hybrid Genetic Algorithms

Genetic algorithms often take many iterations to fit the data set, and depending on the size of the parameter space may simply never arrive at a reasonable solution. However they do provide an effective way to search through a large model space, if the actual fitting of the parameters can be handled some other way. For example, suppose we have a large number of possible predictors for a logistic regression model. Rather than fit all the predictors, we can define a genetic algorithm where individuals correspond to a binary vector with 1's determining the predictors to be included in the corresponding model. The models for each individual are fitted using standard techniques. In this way we can leverage the well-developed theory of logistic regression, and use the genetic algorithm approach to heuristically search the space of predictors for a good, or optimal set. This is an example of a *hybrid genetic algorithm*. The advantage of hybrid algorithms is that each individual corresponds to a fitted model, thus saving a lot of time searching through sub-optimal parameter values.