STA3043S – GLMs, Machine Learning, and Bayesian Analysis

Lecture Notes on Machine Learning

Notes by Dr. Etienne A.D. Pienaar.

Department of Statistical Sciences

University of Cape Town

July 28, 2021

Preface

These course notes provide introductory material on what is loosely termed "Machine Learning". Though it is possible to distinguish machine learning in its pure form from the content covered here, at an introductory level the concepts and techniques covered here can be considered applications of machine learning. In any case, we need not concern ourselves with nomenclature, but rather we should focus on what is important: the overarching principles of statistical learning and how the machinery covered here is to be applied in the real world. As such, this section of the course will cover some theory, but the content will be mostly applied in nature.

These notes were compiled by Dr. Etienne A.D. Pienaar. The content borrows from various sources, but the relevant texts for the present content is that of James et al. (2013), Friedman et al. (2001), Abu-Mostafa et al. (2012), and Nielsen (2015).

1 Introduction

At present, you'll be familiar with the Generalised Linear Model framework and the suite of methods which arise out of imposing a simple linear structure on the relationship between predictors and responses within a suitably chosen family of distributions. Although the resulting class of models is extremely powerful and permits rigorous vetting of the assumptions on which it is built (residual analysis, leverage etc.) there are numerous situations in which an alternative statistical paradigm might yield more desirable results. To see this, consider the following:

1. Should we always be structured in our approach to the analysis? Can a statistician always **build** a mathematical model that replicates what is observed in the data in the first place?

2. Is it realistic to assume that relationships between predictors and responses are always linear, or that some translation of the predictors can be constructed which is linearly related to the response?

3. Sometimes we really only care about the predictive power of a model and we are not at all concerned with the structure of the relationship.

To reorientate ourselves, we take a step back and introduce the **Statistical Learning** paradigm which governs the content. First, we distinguish between two types of **learning**: **supervised** and **unsupervised**.

- **Supervised learning** concerns models and learning algorithms (an example of a model would be logistic regression, and the corresponding learning algorithm would be Iterative Weighted Least Squares (IWLS)) which are tasked with extracting a pattern from the data (a relationship between predictors and responses) with the aim of predicting some response. The task is called "supervised" because we have examples of the responses which we wish to predict. E.g., at each iteration of IWLS we check how *far* our predictions are from the observed responses and update the model parameters accordingly. That is, we **train** the model by supervising it's performance.

- **Unsupervised learning** concerns strategies for identifying structures in the data without the aid of a response variables. For example, do certain covariate patterns cluster in the predictor space?

Though you should be able to distinguish these two learning modes, the methods considered here will focus on supervised learning tasks. Within the domain of supervised learning tasks we can make further important distinctions. Note that, though we are altering our modelling paradigm, our universe of data has not changed and the type of supervised learning task we need to conduct is still dictated entirely by the data. That is, we are still presented with the same species of data: In one context we may be presented with binary or polytomous responses and in another context we may be presented with continuous responses. In the case of continuous responses, we refer to such a supervised learning task as a **regression problem**. In cases where responses are categorical in nature we refer to these as **classification problems**.

In what follows, we cover two model classes that pertain to the analysis of supervised classification and regression problems and develop the machinery at the hand of a number of practical examples.

2 Regression and Classification Trees

This section is based on Section 9.2 in Friedman et al. (2001).

2.1 Model Mechanics: Tree-Based Methods

Let's consider a simple classification problem: Suppose we have a set of binary responses $\{y_i : i = 1, 2, ...N\}$ and a vector of two covariates $\{\mathbf{x}_i^T = [x_{i1}, x_{i2}]^T : i = 1, 2, ...N\}$. Figure 1 plots a sample of such dataset with $N = 50$ observations. Clearly, higher values of the covariates x_{i1} and x_{i2} are associated with responses encoded as 1 (blue dots).

```
R> # Read in the data:
R> dat = read.table('Section 2.1 - Simulated Data.txt', h = T)
R> attach(dat)
R>
R> # Plot the responses in the predictor/feature space:
R> plot(Covariate_2~Covariate_1, pch = c(1,16)[Y+1], col = c(1,4)[Y+1], main = 'Dummy
    Classification Data')
R> legend('topright', legend = c('Response = 0', 'Response = 1'),pch = c(1,16),col = c(1,4),
    bty = 'n')
```

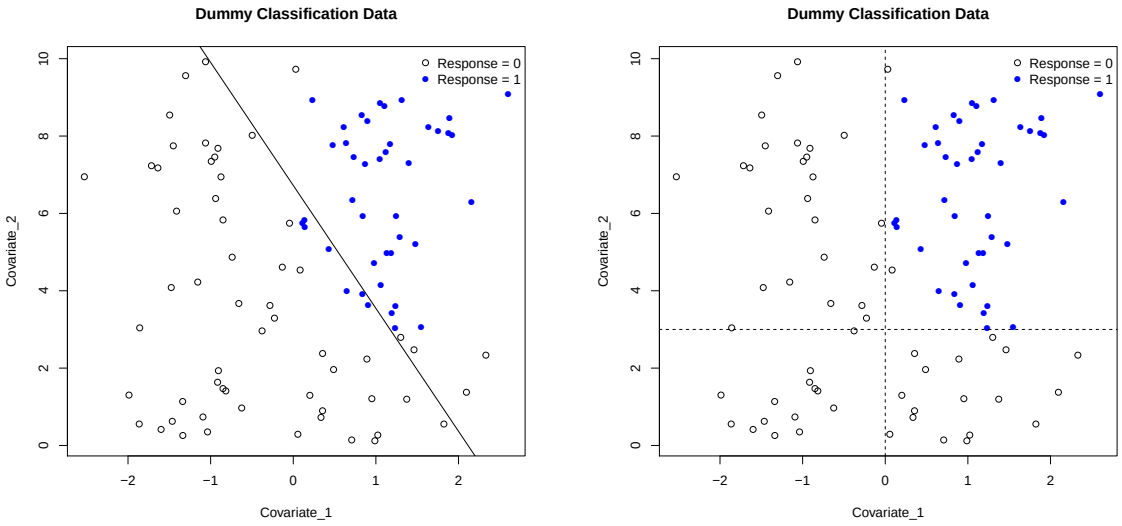


Figure 1: An artificial dataset for a simple binary classification problem. Left: Discriminating bound derived from a logistic regression bound. Right: Mechanical partitioning of the predictor space along its dimensions using manually chosen discriminants.

After fitting a logistic regression model to this data we can set up a discriminating boundary in the **predictor**/ **feature** space [1] by solving for all the points at which the prediction from the logistic regression model would be equal to 0.5. Since observations with $\hat{\pi}_i > 0.5$ would be classified as $\hat{y}_i = 1$ and 0 otherwise. Extracting the coefficients and solving for $\text{logit}(0.5) = \hat{\beta}_1 + \hat{\beta}_2 x_1 + \hat{\beta}_3 x_2 = 0$, we find the discriminant at $x_2 = -\hat{\beta}_1/\hat{\beta}_3 - \hat{\beta}_2/\hat{\beta}_3 x_1$.

```
R> # Fit a logistic regression model to the data:
R> res_1 = glm(Y~Covariate_1+Covariate_2, family = binomial(link = 'logit'))
R> summary(res_1)
```

[1] The predictor or feature space is simply the p-tuple in which the predictors *live*.

```
Call:
glm(formula = Y ~ Covariate_1 + Covariate_2, family = binomial(link = "logit"))

...

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)  -7.3487     2.4245  -3.031  0.00244
Covariate_1   2.6873     0.8837   3.041  0.00236
Covariate_2   1.2017     0.4236   2.837  0.00455

...
```

```
R> # Add a discriminant to the plot
R> beta_hat = coefficients(res_1)
R> abline(a = -beta_hat[1]/beta_hat[3], b = -beta_hat[2]/beta_hat[3])
```

Indeed, this is all that the logistic regression model is doing: it simply **partitions** the feature space. So, instead of partitioning the feature space at the hand of a model, why not do so more directly in a more mechanical fashion? [2] The so-called **tree-based methods** function by partitioning the **predictor**/**feature** space along each of its respective dimensions in such a way as to associate the regions which are created by these partitions with particular responses. Suppose, for example, that we place a discriminating bound along the first covariate axis at 0, and along the second covariate axis at 3. The region to the right of the first discriminating bound and to the top of the second now contains all of the blue responses with the remainder of the observations falling outside of this region. We have now *perfectly*[3] partitioned the feature space in terms of classifying the observations. We can thus write an algorithm for classifying the data as a set of logical rules encapsulated in a nested fashion as:

```
                      -- if(Covariate_1 < 0.01) ----------------------------> Predict 0
Data: Y|Covariates --|
                      -- if(Covariate_1 >= 0.01) -- if(Covariate_2 < 2.9)  --> Predict 0
                                                  |
                                                  -- if(Covariate_2 >= 2.9) --> Predict 1
```

The origin of the 'tree' metaphor should now be obvious. Now, obviously we chose the discriminants by visualising the data and finding points which separate the data. In practice, we need a more rigorous approach to defining these discriminants. For these purposes, most software packages use something called **recursive binary splitting**. This can, for example, be implemented in `R` using the `tree` package:

```
R> # Call the 'tree' library:
R> library(tree)
R>
R> # Type-cast Y to a factor so tree() knows to perform classification:
R> res_2 = tree(factor(Y) ~ Covariate_1 + Covariate_2)
R> summary(res_2)
```

```
Classification tree:
tree(formula = factor(Y) ~ Covariate_1 + Covariate_2)
Number of terminal nodes:  3
Residual mean deviance:  0 = 0 / 97
Misclassification error rate: 0 = 0 / 100
```

The `tree()` function will then return an object which contains all the information you need about the **classification tree** that it fitted to the data. Calling `summary(res2)` we see that the procedure finds 3 so-called **terminal nodes**. Referring to our little logic table, one can expect this to correspond to the final outcomes of our 'logic-tree'. Indeed, calling `res2`, we see that the partitioning algorithm found discriminants which are broadly similar to our crudely chosen bounds:

```
R> res_2
```

[2] As you will see, even the most advanced statistical machinery exhibit fundamentally very simple mechanics such as partitioning the feature space.

[3] At least with respect to the observations we have seen...

```
node), split, n, deviance, yval, (yprob)
      * denotes terminal node

1) root 100 136.10 0 ( 0.5800 0.4200 )
  2) Covariate_1 < 0.0969607 42   0.00 0 ( 1.0000 0.0000 ) *
  3) Covariate_1 > 0.0969607 58  68.32 1 ( 0.2759 0.7241 )
    6) Covariate_2 < 2.91191 16   0.00 0 ( 1.0000 0.0000 ) *
    7) Covariate_2 > 2.91191 42   0.00 1 ( 0.0000 1.0000 ) *
```

Here `split` indicates which predictor a partition is made on and at what point along its dimension. `n` indicates how many of the observations for which the condition defining the split is satisfied (after accounting for conditions further up in the tree). `yval` indicates the predicted value associated with the region defined by the condition defining the split. The output can be recovered in more legible fashion by noting that `res_2` is in fact a list, and the first element in that list outlines the tree in tabular form:

```
R> res_2[[1]]
```

```
          var   n      dev yval splits.cutleft splits.cutright   yprob.0   yprob.1
1 Covariate_1 100 136.0584    0     <0.0969607      >0.0969607 0.5800000 0.4200000
2      <leaf>  42   0.0000    0                                1.0000000 0.0000000
3 Covariate_2  58  68.3243    1       <2.91191        >2.91191 0.2758621 0.7241379
6      <leaf>  16   0.0000    0                                1.0000000 0.0000000
7      <leaf>  42   0.0000    1                                0.0000000 1.0000000
```

To see how this relates to the aforementioned logic tree, we can plot the model object:

```
R> # Plot the fitted tree and add text to the nodes:
R> plot(res_2)
R> text(res_2)
```

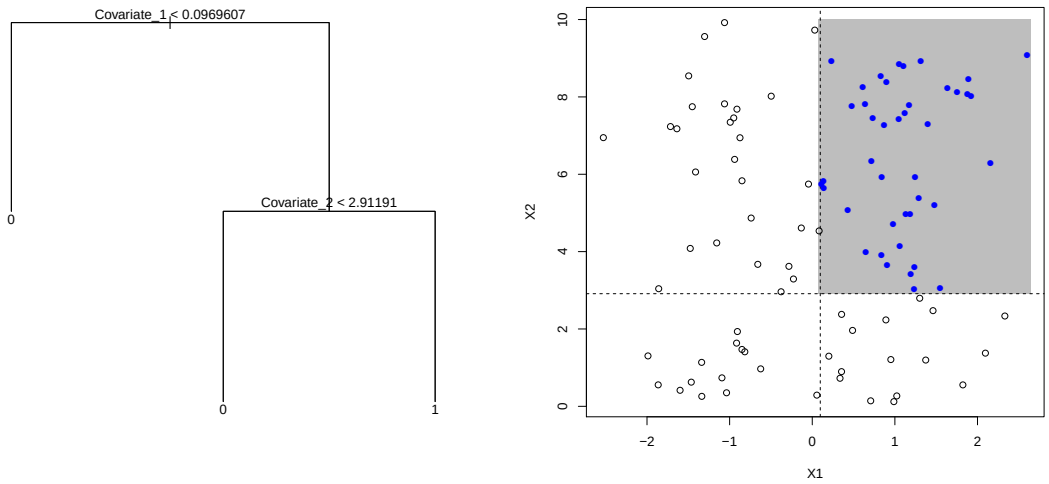


Figure 2: An artificial dataset for a simple binary classification problem. Left: Plot of the classification tree. Right: Predicted output for the observed regions (grey indicates $\hat{y} = 1$).

The model starts at the so-called **root-node**, then proceeds to split along one of the predictors, forming two **branches**. Subsequently, the model may split along each of these branches to form two new branches and so on and so forth. At some stage the learning algorithm stops splitting, resulting in so-called **terminal nodes** or **leaves** giving predictions for the response variables associated with the region defined by the sequence of splits tracing back to the root-node.

Although the model is communicated as a logic tree, the mechanics of the model dictate that it is, in principle just partitioning the feature space into subregions. In what follows, we consider two real-world examples and demonstrate how to interpret the resulting model.

Flu Shots Data: Taken from Neter et al. (1996).
The following data concerns a study conducted by a clinic on the uptake of flu shots prior to an expected flu epidemic. A number of clients were randomly selected and asked whether they had received a flu shot or not and additional information was collected on the age, gender, and health awareness of clients. The data consists of 159 observations on the following variables:

Variable	Description
`Shot`	`"Yes"` if a flu shot was received, `"No"` if not.
`Age`	Age in years.
`Gender`	Gender of client indicated as `"Male"` or `"Female"`.
`Awareness`	Health awareness index, higher values indicating greater awareness.

```
R> dat = read.table('Section 2.1 - Flu Shots Data.txt', h = T)
R> attach(dat)
R>
R> # Tabulate the data:
R> head(dat,10)
```

```
   Shot Age Awareness Gender
1    No  59        52 Female
2    No  61        55   Male
3   Yes  82        51 Female
4    No  51        70 Female
5    No  53        70 Female
6    No  62        49   Male
7    No  51        69   Male
8    No  70        54   Male
9    No  71        65   Male
10   No  55        58   Male
```

Now, let's fit a classification tree to the dataset. Note that for the present dataset, we have two covariates as well as a categorical predictor, `Gender`. Fortunately, categorical predictors do not present any real complication since the splits can be made according to the class labels for such predictors. In this case, `Male` and `Female`.

```
R> # Fit a classification tree to the data:
R> library(tree)
R> res1 = tree(Shot~., data = dat)
R> plot(res1)
R> text(res1)
R>
R> res1[1]
```

```
         var   n        dev yval splits.cutleft splits.cutright   yprob.No  yprob.Yes
1        Age 159 134.940762   No          <67.5           >67.5 0.84905660 0.15094340
2  Awareness 116  58.221515   No          <50.5           >50.5 0.93103448 0.06896552
4     <leaf>  15  19.095425   No                                0.66666667 0.33333333
5        Age 101  27.009045   No          <58.5           >58.5 0.97029703 0.02970297
10    <leaf>  60   0.000000   No                                1.00000000 0.00000000
11 Awareness  41  21.464688   No          <60.5           >60.5 0.92682927 0.07317073
22    <leaf>  25   0.000000   No                                1.00000000 0.00000000
23    <leaf>  16  15.442482   No                                0.81250000 0.18750000
3  Awareness  43  56.765180   No            <48             >48 0.62790698 0.37209302
6     <leaf>   9   9.534712  Yes                                0.22222222 0.77777778
7  Awareness  34  39.298682   No          <64.5           >64.5 0.73529412 0.26470588
14 Awareness  29  35.923825   No          <55.5           >55.5 0.68965517 0.31034483
```

```
28 Awareness  16  15.442482   No          <51.5           >51.5 0.81250000 0.18750000
56    <leaf>   8  10.585012   No                                0.62500000 0.37500000
57    <leaf>   8   0.000000   No                                1.00000000 0.00000000
29 Awareness  13  17.944828   No            <59             >59 0.53846154 0.46153846
58    <leaf>   8  10.585012  Yes                                0.37500000 0.62500000
59    <leaf>   5   5.004024   No                                0.80000000 0.20000000
15    <leaf>   5   0.000000   No                                1.00000000 0.00000000
```

```
R> ind = as.numeric(Shot)
R> plot(Awareness~Age, pch = c(1,16)[ind], col = c(1,4)[ind])
R>
R>
R> # Set-up an MxM lattice in the predictor space and plot the predictions for each coordinate
    :
R> M     = 100
R> X1    = rep(seq(min(Age), max(Age), length = M),M)
R> X2    = rep(seq(min(Awareness), max(Awareness), length = M),each = M)
R> Lat   = data.frame(Age = X1,Awareness = X2, Gender = rep('Male',M^2))
R> pred1 = predict(res1, newdata = Lat,type = 'class')
R>
R> plot(X2~X1, col = c('white','grey')[pred1], pch = 15,xlab ='Age', ylab ='Awareness')
R> points(Awareness~Age, pch = c(4,16)[ind], col = c(1,4)[ind])
```

As it turns out, the fitted tree produces numerous splits along the covariates `Age` and `Awareness`, resulting in what appears to be a complicated logical tree—see Figure 3. Furthermore, we note that the tree does not make use of the `Gender` variable in constructing its predictions (seen at the terminal nodes). To get a feel for what's going on, we repeat the analysis as before and construct a lattice in the `Age`×`Awareness` space and visualise the predictions made by the fitted tree. As you can see, despite the apparent complexity of the model, the tree simply identifies two regions associates with the 'Yes' response. From the plot, we conclude that these regions correspond to individuals with high `Age` and low `Awareness` scores. Furthermore, since `Gender` is not used in constructing the discriminants, we conclude that it is unimportant for purposes of predicting flu shot uptake.

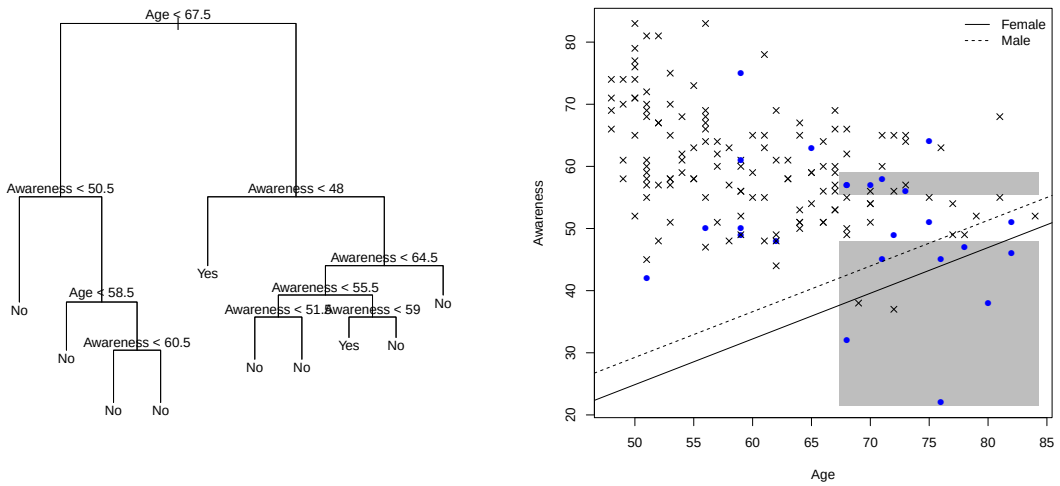


Figure 3: Left: Decision tree fitted to the Flu Shots data. Right: Regions for which the prediction from the decision tree are `'yes'` indicated in grey with the observations superimposed. Discriminants from a main-effects logistic regression partition the feature space diagonally.

Compare these results with that of logistic regression. Indeed, though the model mechanics differ significantly, the analysis yields similar results:

```
R> # Fit a logistic regression model to the data:
R> res_2 = glm(Shot~., data = dat, family = binomial(link = 'logit'))
R> summary(res_2)

Call:
glm(formula = Shot ~ ., family = binomial(link = "logit"), data = dat)

...

Coefficients:
             Estimate Std. Error z value Pr(>|z|)
(Intercept) -1.17716    2.98242  -0.395  0.69307
Age          0.07279    0.03038   2.396  0.01658 *
Awareness   -0.09899    0.03348  -2.957  0.00311 **
GenderMale   0.43397    0.52179   0.832  0.40558
```

As a final point, we consider the mechanism for constructing predictions. Recall that in the case of logistic regression, the fitted values are in fact probabilities, not binary outcomes like the original observations. In order to make a prediction, we posit a threshold probability above which we classify and observation as being "Yes". In the case of a classification tree, we see that the prediction for leaf node 6 (Age $>$ 67.5, Awareness $<$ 48, corresponding to the bottom right grey block in Figure 3) is indicated as "Yes". However, we are also given an estimate of the probability that the observation takes on the predicted value. This value is calculated by calculating the relative frequency of such an observation in that region. E.g., in the aforementioned region, we observe $7/9 = 0.7777778$ responses with label "Yes", and $2/9 = 0.2222222$ responses with label "No" (count the dots and crosses in this region).

California Data: Taken from Kaggle.
This data set appears in a paper on spatial statistics: Pace, R. Kelley, and Ronald Barry. "Sparse spatial autoregressions." Statistics & Probability Letters 33.3 (1997): 291-297. The aim of the study was to predict the value of real estate using geographic and demographic variables collected from census data. The authors' description of the data follow: "We collected information on the variables in (8)[4] using all the block groups in California from the 1990 Census. In this sample a block group on average includes 1425.5 individuals living in a geographically compact area. Naturally, the geographical area included varies inversely with the population density. We computed distances among the centroids of each block group as measured in latitude and longitude. We excluded all the block groups reporting zero entries for the independent and dependent variables. The final data contained 20,640 observations on 9 characteristics." So each observation summarises information on a block group with:

Variable	Description
median_house_value	Median house value in block group – the response variable.
longitude	Angular distance north or south of the earth's equator.
latitude	Angular distance east or west of the meridian.
housing_median_age	Median age of individuals in block group.
total_rooms	Total number of rooms observed in block group.
total_bedrooms	Total number of bedrooms observed in block group.
population	Population size in block group.
households	Number of households in block group.
median_income	Median income in block group.
ocean_proximity	No disambiguation given. Presumably a distance measure relative to coastline.

For now, we consider only the latitude[5] and longitude[6] predictors which give the geographical position of the observations only. Figure 4(a) superimposes the observations on a map of California with points

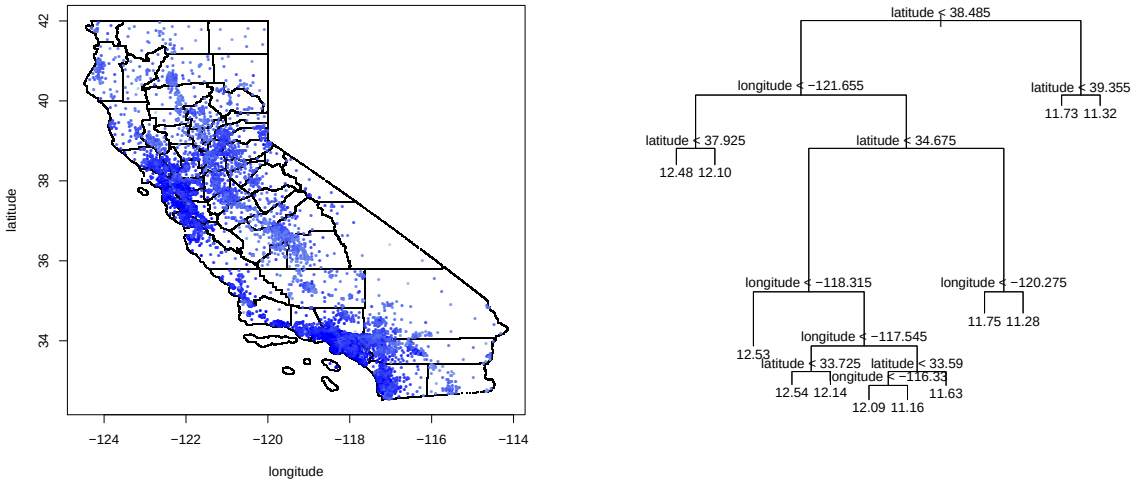
[4] (8) here refers to a model equation being discussed in the paper.
[5] Latitude: the angular distance of a place north or south of the earth's equator, or of a celestial object north or south of the celestial equator, usually expressed in degrees and minutes.
[6] Longitude: the angular distance of a place east or west of the meridian at Greenwich, England, or west of the standard meridian of a celestial object, usually expressed in degrees and minutes

colour-scaled from light to dark blue according to the value of the log of the median house price, i.e., `log()` — the response which we will attempt to predict.

As before, we fit a tree using the `tree` function, noting that the model is now a **'regression' tree** owing to the continuous nature of the response variable and arrive at the somewhat complicated tree model illustrated in Figure 4(b).

```
R> dat = read.csv('Section 2.1 - California Housing Data.csv',h = T)
R> attach(dat)
R> res1 = tree(log(median_house_value)~latitude+longitude)
R>
R> # Figure 3 (b):
R> plot(res1)
R> text(res1)
R> res1[[1]]
```

```
          var     n        dev     yval splits.cutleft splits.cutright
1    latitude 20640 6685.26264 12.08488        <38.485         >38.485
2   longitude 18579 5750.77416 12.13931      <-121.655       >-121.655
4    latitude  4638  960.79248 12.39194        <37.925         >37.925
8      <leaf>  3575  661.87263 12.48013
9      <leaf>  1063  177.59431 12.09533
5    latitude 13941 4395.51964 12.05527        <34.675         >34.675
10  longitude 11097 2688.67998 12.19497      <-118.315       >-118.315
20     <leaf>  2713  464.62721 12.52902
21  longitude  8384 1823.33017 12.08687      <-117.545       >-117.545
42   latitude  5545  941.96344 12.19446        <33.725         >33.725
84     <leaf>   687  108.12678 12.54467
85     <leaf>  4858  737.66416 12.14494
43   latitude  2839  691.79796 11.87672         <33.59          >33.59
86  longitude  1760  427.68901 12.02546       <-116.33        >-116.33
172    <leaf>  1630  310.55092 12.09440
173    <leaf>   130   12.24993 11.16103
87     <leaf>  1079  161.65670 11.63410
11  longitude  2844  645.27308 11.51018      <-120.275       >-120.275
22     <leaf>  1384  275.83117 11.75148
23     <leaf>  1460  212.47731 11.28145
3    latitude  2061  383.26413 11.59422        <39.355         >39.355
6      <leaf>  1387  240.39580 11.72928
7      <leaf>   674   65.51082 11.31630
```

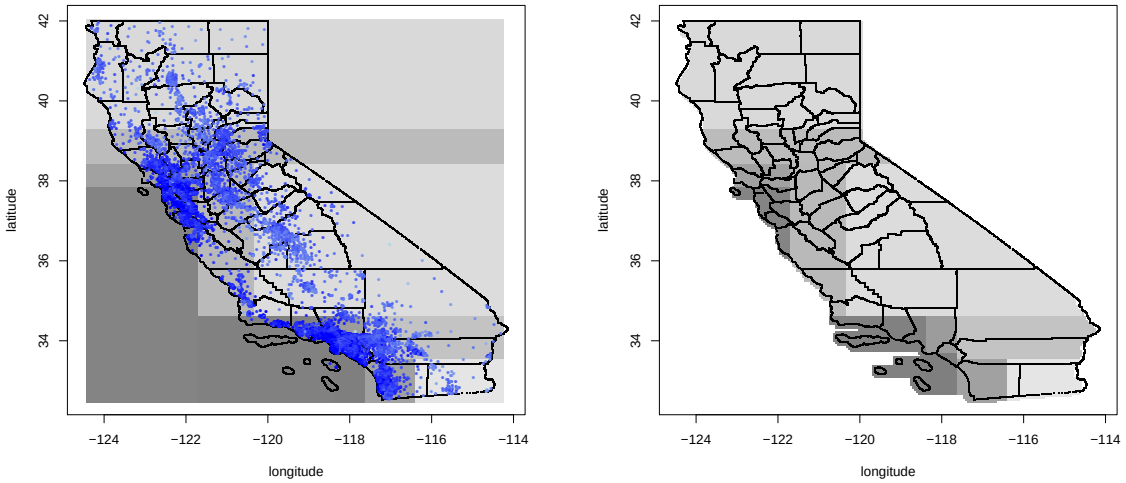


(a) Colour-scaled observations (increasing from light blue to dark blue w.r.t. `log(median_house_value)` superimposed on a map of California.

(b) A regression tree fitted using only `latitude` and `longitude` as predictors.

Figure 4: Using the geographical position of the observations to build a regression tree for predicting the median value of homes in the state of California.

Despite the complexity of the tree, the model can easily be used to formulate a prediction for block-groups on the map by simply noting its geographic location and which **cut-points** apply to it. Figure 5(a) superimposes colour-scaled predictions (increasing from light to dark grey) for the regions corresponding to the terminal nodes of the model. A naive interpretation of the figure would suggest that the model predicts rather expensive homes can be found somewhere off of the Pacific coast... Note, however, that these models are still subject to the pitfalls of extrapolation. As such, the Statistician would rather interpret the model within the bounds of what has been observed. I.e., in the mind's eye of the Statistician, he/she would see Figure 5(b). Again, the mechanism is clear: the model partitions the map into square regions (not to be confused with the 'block groups' from which the data were collected) with predictions based on the observations in those regions. Perhaps unsurprisingly, the model associates higher response values with regions which are both more densely populated and closer to the coast. Indeed, the analysis can be further improved by including the demographic predictors in the dataset – We'll revisit this dataset in the tutorials.



(a) Colour-scaled predictions under the tree model using a square region encapsulating the entire state of California.

(b) Colour-scaled predictions trimmed to fit the map to avoid extrapolation into unobservable regions in the feature space, i.e., regions that do not fall inside the relevant geographic region.

Figure 5: Predicted responses over a square region encapsulating the observations.

What the present example demonstrates is two-fold: First, tree-based models offer an advantage over linear models in that they can easily handle highly non-linear data (note the shape of the data in the feature space). Second, despite the aforementioned non-linearity, they remain easy to interpret. In the section that follows, we discuss how the models are fitted and strategies for tuning such models to ensure accurate representation of the data.

2.2 Model Mechanics: Learning Algorithms for Tree-Based Methods

Now that you are familiar with the principle of partitioning the feature space you might wonder how we decide on what constitutes an appropriate split position and which dimensions to split along in the first place? To demonstrate how this is achieved, we consider a simple regression problem. As in the case of simple linear regression, we may use residual sum of squares as a criteria for how good the predictions from the present configuration is. Assuming that the present configuration results in a set of J regions $R_1, R_2, \ldots, R_J$ in the feature space (corresponding to the leaf-nodes), with corresponding predictions $\hat{y}_{R_1}, \hat{y}_{R_2}, \ldots \hat{y}_{R_J}$, we evaluate:

$$\text{RSS} = \sum_{j=1}^{J} \sum_{i:\mathbf{x}_i \in R_j} (y_i - \hat{y}_{R_j})^2.$$

The procedure, or **learning algorithm** (Abu-Mostafa et al., 2012) for fitting a tree can then be described as follows:

1. Initial step: Start with a single region, the feature space. For each of the dimensions of the feature space, find the split position which leads to the **greatest reduction in RSS**, provided it exceeds some minimal **threshold** for reduction. Compare the maximal reduction in RSS across dimensions and retain the greatest reduction amongst those.

2. Iteration: Split the data according to the regions defined by the split chosen at the previous step. Repeat the procedure in the aforementioned step for each of these regions, retaining splits for each

(provided they occur).

3. Stop: If no split can be found that exceeds the reduction threshold, stop and return the resulting tree.

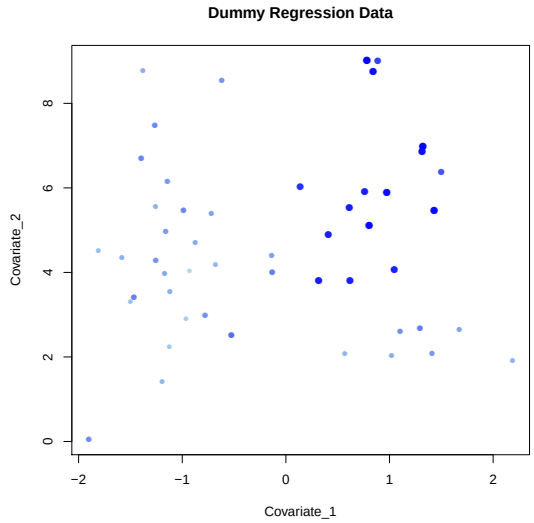
To see how the algorithm functions in practice, we consider a modified version of the simulated dataset of Section 2.1. For ease of exposition, the data were perturbed in such a way as to turn the responses into real-valued observations (i.e., a continuous measurement scale), thus rendering a regression problem which would result in similar partitions to the original data. Figure 6 (a) gives the colour-scaled data for the modified dataset. We've also sized the points according to their response magnitude. Stepping through the recursive partitioning procedure for this dataset, we plot the reduction in RSS for splits along `Covariate_1` and `Covariate_2` in Figure 6 (b). That is, for the unpartitioned data, we calculate the RSS: `sum((mean(Y)-Y)^2)` $= 116.0985$ and then subtract the combined RSS for the regions constructed by a split along each dimension to get the reduction in RSS. For example, splitting `Covariate_1` at 1, the revised RSS evaluates to `sum((mean(Y[Covariate_1<0.0036])-Y[Covariate_1<0.0036])^2)` $= 9.464326$ for the first region and `sum((mean(Y[Covariate_1>=0.0036])-Y[Covariate_1>=0.0036])^2)` $= 49.78052$ for the second region for a reduction of 56.85365. — the level of the first peak in Figure 6 (b). Note that the aforementioned RSS values correspond to the `dev` (for **deviance**) entries in the output from the `tree()` model object in:

```
R> res = tree(Y~Covariate_1+Covariate_2)
R> res[[1]]
```

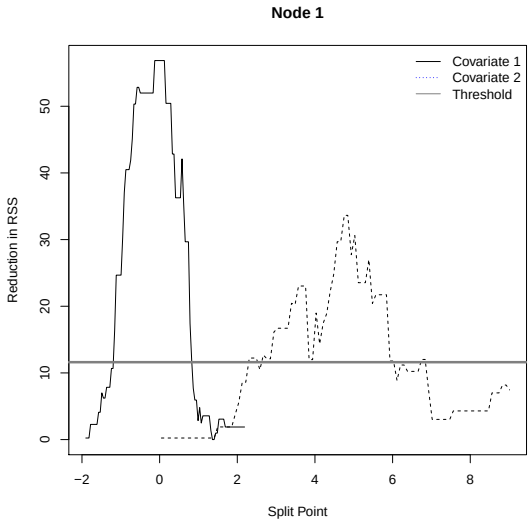
```
          var  n        dev        yval splits.cutleft splits.cutright
1 Covariate_1 50 116.098482  0.90897083    <0.00363747     >0.00363747
2      <leaf> 27   9.464326 -0.07521288
3 Covariate_2 23  49.780516  2.06431692       <3.23917        >3.23917
6      <leaf>  7   1.299800 -0.03908865
7      <leaf> 16   3.961047  2.98455686
```

Indeed, for the present example, we find that at the root node (Node 1), it is optimal to split `Covariate_1` at 0.00363747 — the point of maximal reduction in RSS that exceeds the minimum required reduction threshold, here defined as $0.1\times$RSS for the unpartitioned data.

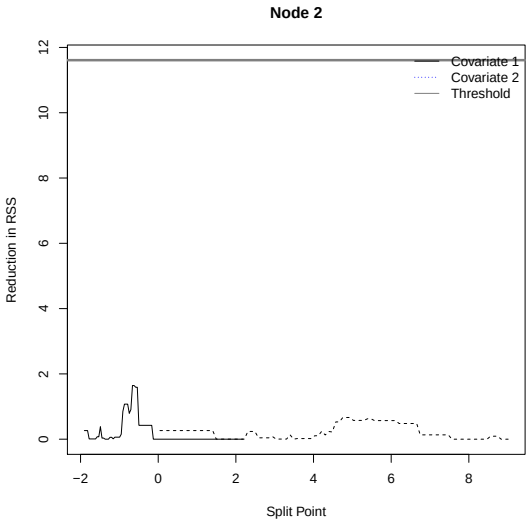
Following the aforementioned split, we repeat the procedure for the new regions R_1 and R_2, i.e., the feature space to the left and right of the first split point in turn. The corresponding reduction curves are given in figures 6 (c) and 6 (d), respectively. For R_1, we find that no split produces a reduction above the required threshold, and R_1 becomes a **terminal node** or **leaf** (i.e., there are no nodes 4 and 5 in the next level of the binary tree). For R_2, we find another split at 3.23917 along `Covariate_2` corresponding to the second peak in Figure 6 (d), defining new regions R_2 (since the previous R_2 has been split) and R_3. It the next iteration, no split along either of the new nodes are found to exceed the reduction threshold and the algorithm is terminated.



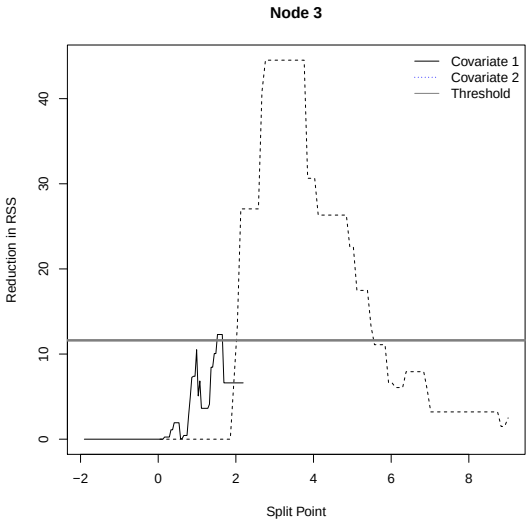
(a) Scatterplot of the responses in the feature space. Larger dots correspond to higher values if the response. Clearly, larger values are to be found in the top right of the feature space.



(b) For the first iteration, we evaluate potential split points in each of the dimensions. A split along covariate 1 would produce the largest reduction in RSS.



(c) On the first branch of the previous split we evaluate potential split points in each of the dimensions. No split point produces a sufficient reduction in RSS.



(d) On the second branch of the previous split we evaluate potential split points in each of the dimensions. A split along covariate 2 would produce the largest reduction in RSS.

Figure 6: A demonstration of the recursive binary splitting algorithm as applied to a simulated regression problem in two variables.

For classification problems, the algorithm operates in a similar fashion, however, the objective function needs to be modified since RSS does not constitute a suitable error metric for binary data — more on this in honours. For classification trees, we use criteria which contrast the observed proportion of observations from a particular class to those predicted by the model. These are also referred to as **impurity measures**; The premise being that if the predicted probabilities of belonging to a particular

class in a region does not clearly distinguish the observations, then the node will be *impure* and incur a large impurity score (since that region will contain numerous observations with different class labels). Recall, that for a binary classification tree, the prediction for region R_j is

$$\hat{\pi}_j = \frac{1}{N_j} \sum_{i:\mathbf{x}_i \in R_j} \mathbb{I}(y_i = 1),$$

where N_j is the number of elements in region R_j. We may then, for example, use the **misclassification error** as a criteria

$$1 - \max(\hat{\pi}_j, 1 - \hat{\pi}_j)$$

for measuring impurity, noting that whenever observations are correctly classified (0 or 1), then this quantity will be small, and when observations are misclassified, this quantity will be large. To see this, we plot this measure as a function of π_j in Figure 7. Other measures of impurity include the so-called **Gini-index**

$$2\hat{\pi}_j(1 - \hat{\pi}_j),$$

and **Shannon-Entropy**

$$-(\hat{\pi}_j \log_2(\hat{\pi}_j) + (1 - \hat{\pi}_j) \log_2(1 - \hat{\pi}_j)).$$

Plotting these measures as functions of $\hat{\pi}_j$ in Figure 7 reveals the broadly similar characteristics of the measures.

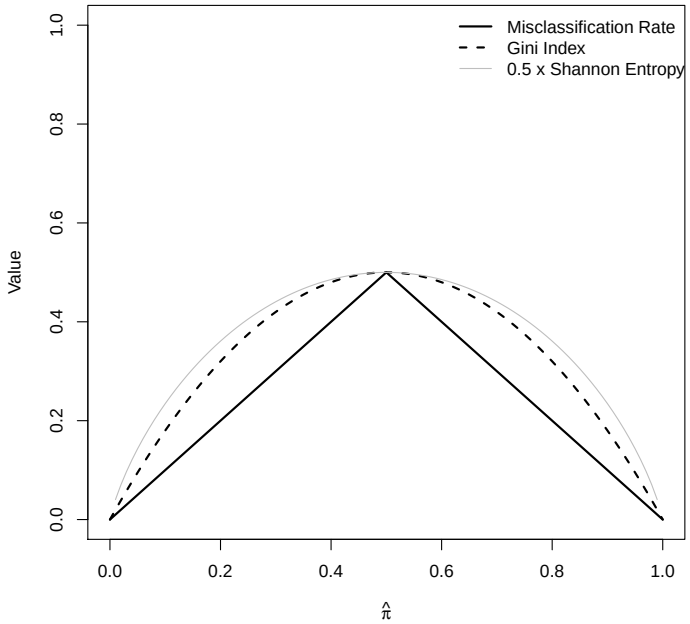


Figure 7: Various "impurity measures". The misclassification rate is the least sensitive to variations in $\hat{\pi}$, and Shannon Entropy the most sensitive. How this relates to 'purity': We wish to minimise these quantities in each node and thus want node probabilities on either end of the curve (pure). The more sensitive the measure, the greater the error incurred for being in the center of the curve (impure).

2.3 Managing Complexity: Cost-Complexity Pruning

One of the downsides to tree-based methods is that they are prone to **overfitting**. This occurs when a model replicates features seen in a particular data set rather than the features of the underlying

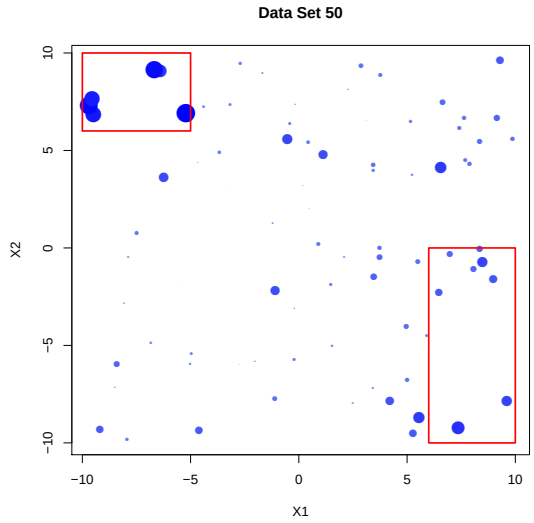
pattern in the data. Indeed, this is true of most machine learning algorithms and the premise of all pure mathematical analysis of machine learning algorithms focus on quantifying the probability that a particular algorithm can extract the true underlying pattern from a data set (more on this in Masters) rather than the applications themselves. Though this is not an altogether trivial phenomenon, it is easy to intuit and visualise. Consider another synthetic dataset:

Since this is a synthetic example, we can simulate multiple realisations of the data set and fit a regression

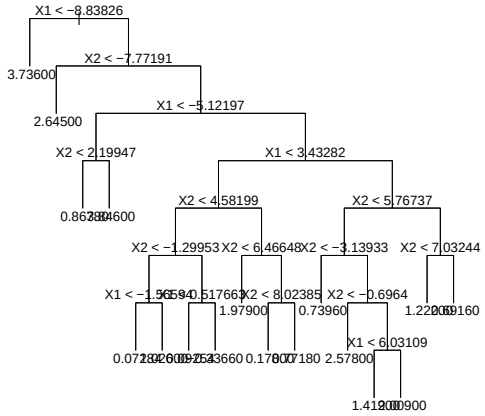
Variable	Description
`Y`	Numerical Response.
`X1`	Covariate 1 (x-axis).
`X2`	Covariate 2 (y-axis).

tree to each realisation. Furthermore, for each dataset, we generate new observations and measure the performance of the model on data which it has not seen. This procedure is visualised in Figure 8: For purposes of the exposition, we show the realisations and superimpose the true partitions in the data. We also plot the tree fitted to said data, and the resulting predicted partitions (or regions under the model). Finally, we measure the performance of the model in the seen data, or **in-sample-error** and compare that to the performance on unseen data, or**out-of-sample-error**. We observe a number of things: First, the model fit varies quite a lot over each data set, and we observe more partitions as predicted under the tree model than the true number of partitions. Second, we observe that on average there is a large discrepancy between the in-sample performance and out-of sample performance of the model.

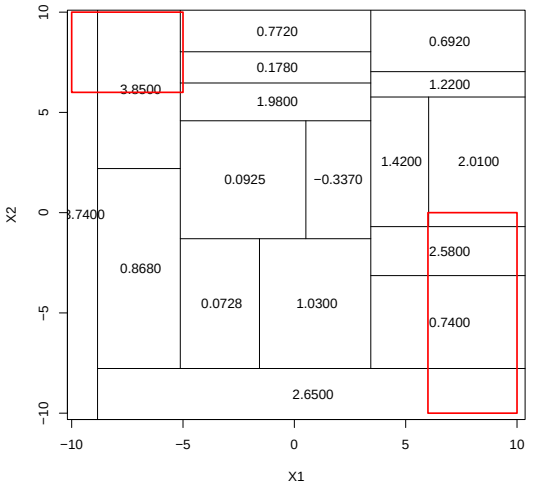
Now, suppose instead we limited the number of terminal nodes to say 4 when fitting the tree at each iteration of the data. Then, repeating the experiment using the exact same generated data-sets, we observe in Figure 9 that the number of predicted regions on average map closer to the true partitions and both the difference between the in-sample and out-of-sample performance is and the mean out-of-sample error is diminished. What this experiment demonstrates is the so-called **bias-variance-trade-off**, a key principle in the domain of statistical learning theory. The principle can be explained as follows: Sufficiently complex models can in theory replicate the data used to train said models almost exactly. This is not good for purposes of *learning* any underlying pattern, and such models exhibit "high variance" and will not perform well on unseen data. Though we can simplify models to remedy this, we also run the risk of simplifying the model too much (e.g., if we had a single node we'd simply predict the response as the mean of all of the responses, and we have modelled no association structure), resulting in a "biased" model, i.e., one which does not replicate salient features in the underlying pattern, also producing poor predictions. The trade-off principle is that we need to find a balance between the complexity of our model and its ability to extract the true underlying pattern. We thus need to manage the complexity of our machine learning models appropriately lest the models be of little use in practice!



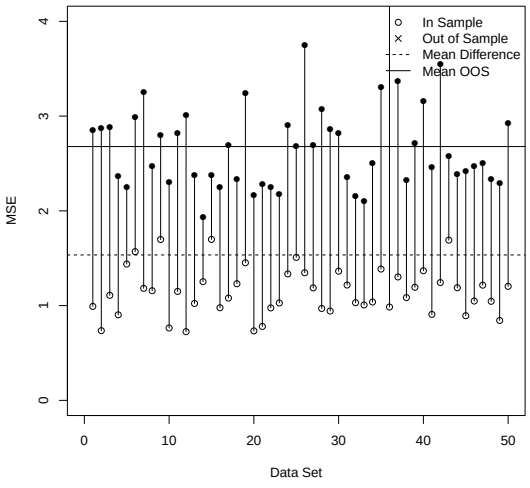
(a) Simulated data points ($N = 100$) drawn from three distributions, each corresponding to a region in the feature space defined by the true partitions (indicated by the red border lines).



(b) A plot of the regression tree fitted to the simulated data.

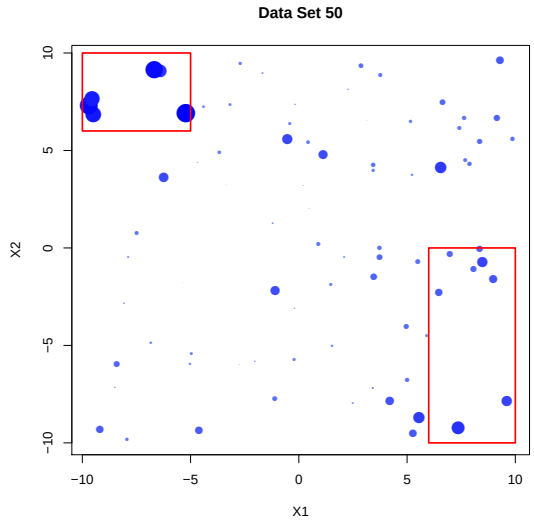


(c) Partitions/regions predicted by the tree model. The true partitions are indicated by the red border lines.

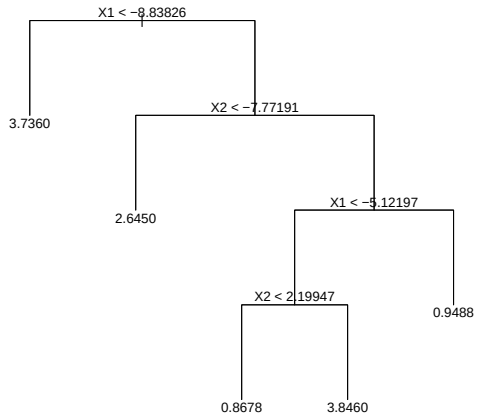


(d) In-sample vs. out-of-sample error for each iteration of the experiment. Running mean of the difference and out-of-sample error over iterations indicated.

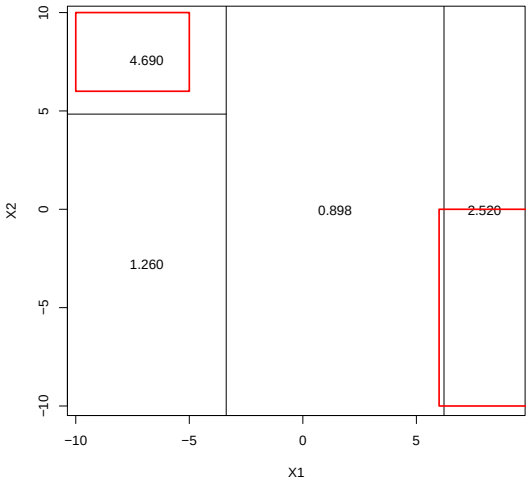
Figure 8: We simulate 50 realisations of a synthetic dataset and fit a tree model at each iteration. Subsequently, we compare the predicted pattern to the true pattern and evaluate the model's performance on unseen data. For the unconstrained tree model, we see quite a bit of variation in the fitted model structure and systematic deviation between the in-sample and out-of-sample performance of the model.



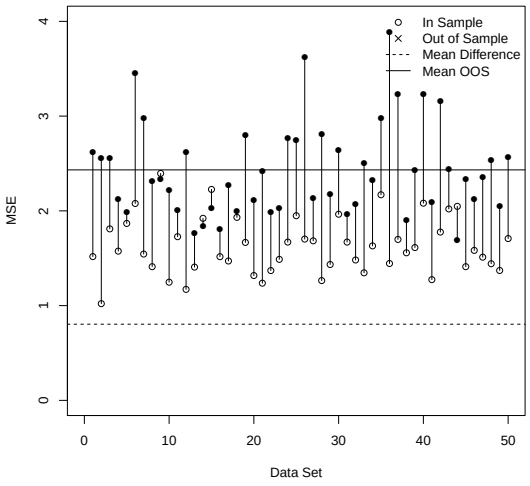
(a) Simulated data points ($N = 100$) drawn from three distributions, each corresponding to a region in the feature space defined by the true partitions (indicated by the red border lines).



(b) A plot of the regression tree fitted to the simulated data. Here the trees are constrained to have at most four terminal nodes.



(c) Partitions/regions predicted by the tree model. The true partitions are indicated by the red border lines.



(d) In-sample vs. out-of-sample error for each iteration of the experiment. Running mean of the difference and out-of-sample error over iterations indicated.

Figure 9: We repeat the simulation experiment, but this time apply an appropriate complexity control by fitting tress of at most 5 terminal nodes.

Although the bias-variance trade-off applies to all machine learning algorithms, we demonstrate how to manage the complexity of tree-based models using a procedure called cost-complexity pruning at the hand of a dataset.

Car Seats data:
The following data are taken from ISLR, the companion R-library to James et al. (2013). The purpose of the analysis is to predict Sales volumes using various demographic and salient variables. For purposes

Variable	Description
Sales	Unit sales (in thousands) at each location.
CompPrice	Price charged by competitor at each location.
Income	Community income level (in thousands of dollars).
Advertising	Local advertising budget for company at each location (in thousands of dollars).
Population	Population size in region (in thousands).
Price	Price company charges for car seats at each site.
ShelveLoc	A factor with levels Bad, Good and Medium indicating the quality of the shelving location for the car seats at each site.
Age	Average age of the local population.
Education	Education level at each location.
Urban	Population size in region (in thousands).
US	A factor with levels No and Yes to indicate whether the store is in the US or not.

of conducting the analysis, we'll make use of the rpart library. As you can deduce from the name, (recursive partitioning), this another excellent R package for performing tree-based analysis. We also use the rpart.plot library for creating improved tree plots.

```
R> library(rpart)
R> library(rpart.plot)
R> library(ISLR)
R> data(Carseats)
R> head(Carseats, 5)
```

```
  Sales CompPrice Income Advertising Population Price ShelveLoc Age Education Urban  US
1  9.50       138     73          11        276   120       Bad  42        17   Yes Yes
2 11.22       111     48          16        260    83      Good  65        10   Yes Yes
3 10.06       113     35          10        269    80    Medium  59        12   Yes Yes
4  7.40       117    100           4        466    97    Medium  55        14   Yes Yes
5  4.15       141     64           3        340   128       Bad  38        13   Yes  No
```

```
R> res1 = rpart(Sales~., data = Carseats)
R> res1[[1]]
```

```
           var   n  wt        dev      yval  complexity ncompete nsurrogate
1    ShelveLoc 400 400 3182.27470  7.496325 0.250510386        4          0
2        Price 315 315 1859.55959  6.762984 0.105072558        4          3
4    ShelveLoc 207 207  956.57240  6.018792 0.045671259        4          1
8   Population  61  61  240.81973  4.722459 0.012147082        4          5
16      <leaf>  25  25   88.22930  3.767200 0.006800514        0          0
17      <leaf>  36  36  113.93508  5.385833 0.007773687        0          0

       .-"-.
     _/_-.-_\_
    / __} {__ \
   / //  "  \\ \ So many branches! And I've learned nothing about the data!
  / / \'---'/ \ \
  \ \_/`"""`\_/ /
   \           /

24      <leaf>  12  12   36.64722  7.152500 0.010000000        0          0
25      Income  36  36  108.32347  9.272500 0.010506135        4          0
50      <leaf>   9   9    9.82780  7.603333 0.010000000        0          0
51      <leaf>  27  27   65.06227  9.828889 0.005324568        0          0
13      <leaf>   9   9   15.27049 11.921111 0.010000000        0          0
7       <leaf>  28  28   85.57727 12.187857 0.006650589        0          0
```

The resulting model has 18 **terminal nodes**—see Figure 11 (a). Although this model may perform well on the present dataset (the single realisation of observations at hand), it is likely that the partitions

are more representative of the what has been observed in **this** realisation of the data, rather than the pattern/mechanism in the **true data generating process**[7]. That is, we have likely **overfitted** the data, and need to trim the model down to a more **parsimonious** model which hopefully replicates the underlying pattern in the data, rather than the nuances particular to this set of observations.

One way to construct a simpler tree is to simply limit the number of terminal nodes. This can be achieved by increasing the threshold by which the particular objective function needs to drop before a split is affected as in Section 2.2. However, it turns out that this is not optimal as this might exclude partitions which are useful for prediction further down the branches of the tree (Friedman et al., 2001). So instead, we need to modify the objective function(s) used in the recursive partitioning algorithm in order to **penalise** the model for producing predictions which too closely match this set of observations. For tree models, the modified objective follows:

$$\text{Penalised Obj.} \;=\; \text{Objective} \;+\; \text{Penalty}.$$

Note, that we still want to minimise the objective, which naturally gets smaller as model complexity increases. However, by adding a **penalty** term which increases as the model complexity increases, the overall objective (LHS) may be larger for complex models than those which balance complexity (flexibility) and predictive performance on unseen observations. Mathematically, we have for a classification example:

$$\text{Penalised Obj.}(\alpha, J) = -\sum_{j=1}^{J}(\hat{\pi}_j \log_2(\hat{\pi}_j) + (1-\hat{\pi}_j)\log_2(1-\hat{\pi}_j)) + \alpha|J|$$

where the first term evaluates the entropy and the second term penalises the objective by scaling the number of terminal nodes ($|J|$) in the tree by a constant α. This constant is a so-called **hyper parameter**, which we may use to manage the **complexity** of our tree model. Noting that for $\alpha = 0$, the algorithm would produce the same complex tree as under the unmodified objective, we may start there and systematically increase α to produce simpler trees. In practice, we actually start with some maximal tree, with say $T_{\alpha=0}$ terminal nodes, and roll back splits from the terminal nodes to find a **nested sub-tree** which satisfies the penalised objective. This process is called **pruning**. Perhaps unsurprisingly, there exist intervals of values for α which correspond exactly to each element in the sequence of sub-trees. This revised strategy for fitting is called **cost-complexity pruning** (Friedman et al., 2001). We use this strategy to produce a sequence of nested trees, with $T_{\alpha=c_1}, T_{\alpha=c_2}, T_{\alpha=c_3}, \ldots, T_{\alpha=0}$[8] partitions for which we can use as potential candidate models for a **balanced** (in terms of generalisation) model. Subsequently, we evaluate an appropriate measure of fit for the sequence of trees and use the trajectory of the measure of fit to judge a point at which additional partitions produce only minor improvements in fit.

For the present example we can visualise this sequence using the `plotcp()` function, which produces Figure 10:

```
plotcp(res1)
```

[7] The term data generating process is here used to represent the hypothetical mechanism which produces observations. Our goal is to extract the rule(s)/pattern which govern this mechanism.

[8] The sequence of values $c_1, c_2, \ldots$ being those values of α at which the number of terminal nodes in the fitted tree would change.

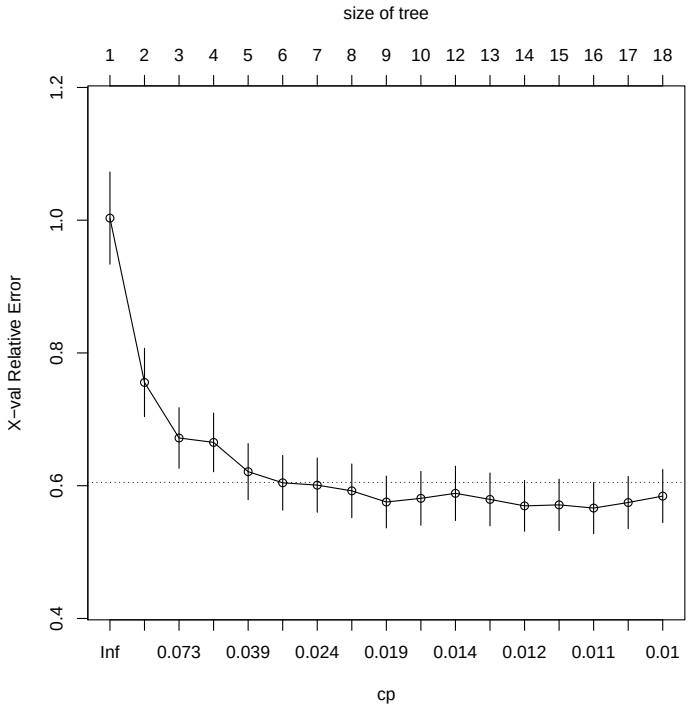


Figure 10: Evolution of the model error as the cost-complexity parameter is decreased from left to right. Large values of the cost complexity parameter correspond to small trees, whilst smaller values correspond to larger, more complex trees. The error for the a tree with 5 terminal nodes is almost within one standard deviation of the minimum of the curve (the dotted line sits one standard deviation above the minimum– see the 10th element), suggesting that there is little to be gained from additional partitions (note that the curve is almost flat from here for decreasing 'cp'.

The figure plots the deviance as a function of the number of terminal nodes and the sequence of corresponding $c_1, c_2, \ldots$. The figure suggests that beyond $T_{\alpha=0.039} = 5$ only minor improvement in fit is observed, and we prune the full tree down to one with 5 terminal nodes, giving our final model—Figure 11 compares the unconstrained and pruned trees.

```
R> res2 = prune.rpart(res1, cp = 0.039)
R> prp(res2)
R> res2[[1]]
```

```
        var   n  wt        dev      yval  complexity ncompete nsurrogate
1 ShelveLoc 400 400 3182.27470  7.496325 0.250510386        4          0
2     Price 315 315 1859.55959  6.762984 0.105072558        4          3
4 ShelveLoc 207 207  956.57240  6.018792 0.045671259        4          1
8    <leaf>  61  61  240.81973  4.722459 0.012147082        0          0
9    <leaf> 146 146  570.41418  6.560411 0.022163275        0          0
5    <leaf> 108 108  568.61745  8.189352 0.033592366        0          0
3     Price  85  85  525.52224 10.214000 0.051120592        4          4
6    <leaf>  57  57  277.26520  9.244386 0.024062792        0          0
7    <leaf>  28  28   85.57727 12.187857 0.006650589        0          0
```

From the pruned tree, we observe that despite all of the information, the variables which were most useful for predicting sales volumes were the price and location of the shelf of the product. Furthermore, 'good' shelf locations were associated with larger volumes whilst products with higher prices appear to sell in larger volumes.

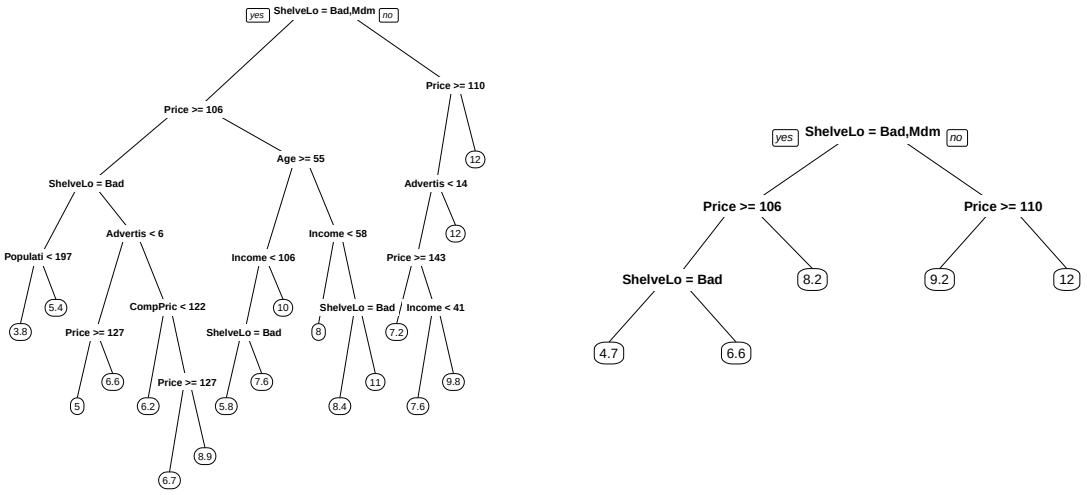


Figure 11: The unconstrained tree model has 18 terminal nodes (left), resulting in a complex decision tree. Subsequently, we prune the tree and monitor the change in relative error. Beyond a tree of size 5 or 6, improvements in performance are largely superficial. Fixing the cost-complexity parameter to 0.039, we fit a tree with 55 terminal nodes, resulting in a much more parsimonious model (right).

Validation Sets

Recall that we typically have a single realisation of the data at hand. As such, we do not have the luxury of directly measuring how our model varies with the data. We also don't have any idea of what the true pattern might look like so we typically cannot guess what an appropriate set of constraints would be to put on the model complexity. As such, our best strategy for assessing the predictive power of our model is to split the data that we have into parts so that we may mask some of the observations from our model and test what it has *learned* on the unseen observations. This is called the **validation set** approach (Abu-Mostafa et al., 2012). For these purposes, we split the data into a **training set**, on which we train the model, a **validation set**, on which we validate the performance of the model and use to modify/calibrate our model to more accurately predict the responses out of sample, and finally the **test set**, a set of observations that we are only allowed to touch **once**[9] to report the true performance of the model. If we conducted the validation analysis correctly, then the model should perform well on this test set and we will be confident that we have extracted the underlying pattern in the data.

Although implementing the validation split in this way is a simple technique for approximating the out of sample performance of a model, it does have two obvious downsides: First, we are wasting data, since we are not training on a significant proportion of the data. Second, since validation error is a function of the data, it will exhibit variation and it is important to have at least some measure of the amount by which it varies. For these purposes, we use so-called **K-fold cross-validation**. This is a process by which the data is split in to K 'folds' where the $K-1$ of the folds are used as training data and the remaining fold used to calculate the validation error. These folds are then cycled in sequence until all folds have been used to train and validate—see Figure 12. In this way, we can ensure both that all of the data are utilised and that we have a small sample (of size K) of validation error estimates, which we can use to estimate the mean validation error and its variance. Figure 12 compares the various validation

[9] DO NOT FOOL YOURSELF and look at the test set more than once! Every time you look at out of sample observations you contaminate the model and, as a consequence, you will overestimate its performance. High prediction performance will look good in your report but you will become a source of contention when your model fails to perform at implementation...

split configurations: a standard validation split of 70/30, 5-fold, and 10-fold cross-validation.

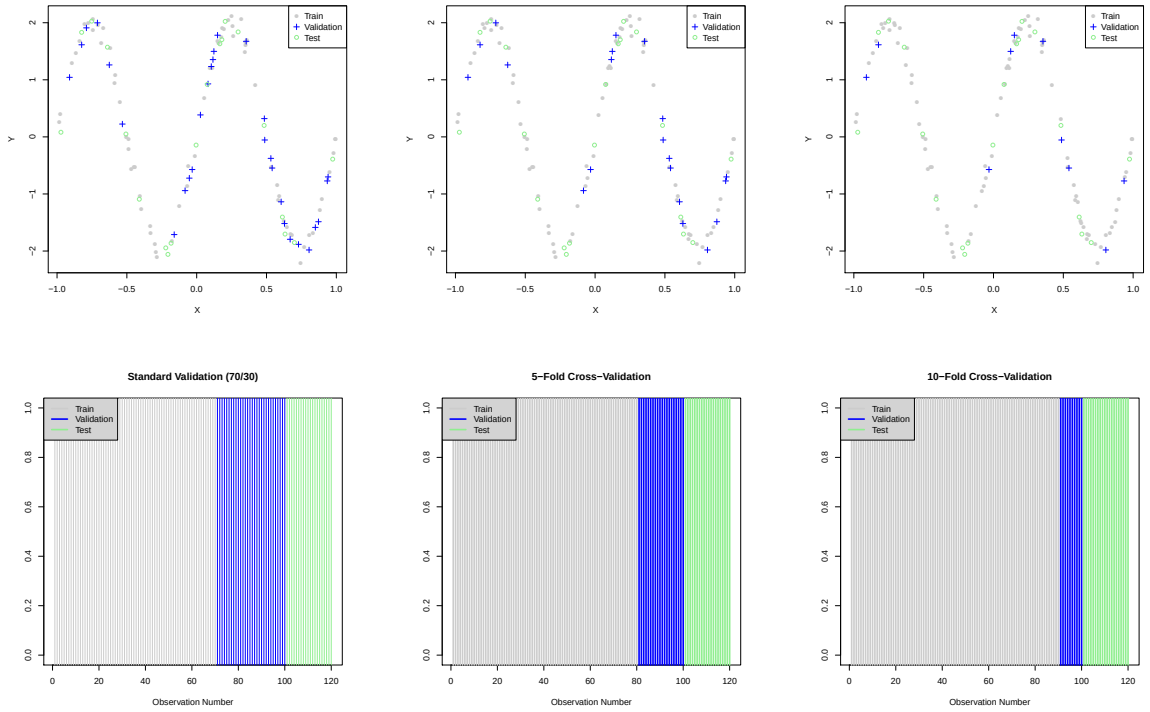


Figure 12:

Revisiting the Car Seats data from Section 2.3, we use the `cv.tree()` function to perform 10-fold cross-validation for purposes of determining the appropriate tree size. This is achieved by applying `cv.tree()` to the unconstrained tree. We can then store the resulting output in an object (`cv`) and produce a validation plot (Figure 13):

```
R> # Prune the tree using cross-validation:
R> res1  = tree(Sales~., data = Carseats)
R> cv    = cv.tree(res1,FUN = prune.tree, K = 10)
R> plot(cv)
R> res2  = prune.tree(res1, best = 5)
R> plot(res2); text(res2)
R> res2[[1]]
```

```
         var   n        dev      yval splits.cutleft splits.cutright
1  ShelveLoc 400 3182.27470  7.496325            :ac              :b
2      Price 315 1859.55959  6.762984         <105.5          >105.5
4     <leaf> 108  568.61745  8.189352
5  ShelveLoc 207  956.57240  6.018792             :a              :c
10    <leaf>  61  240.81973  4.722459
11    <leaf> 146  570.41418  6.560411
3      Price  85  525.52224 10.214000         <109.5          >109.5
6     <leaf>  28   85.57727 12.187857
7     <leaf>  57  277.26520  9.244386
```

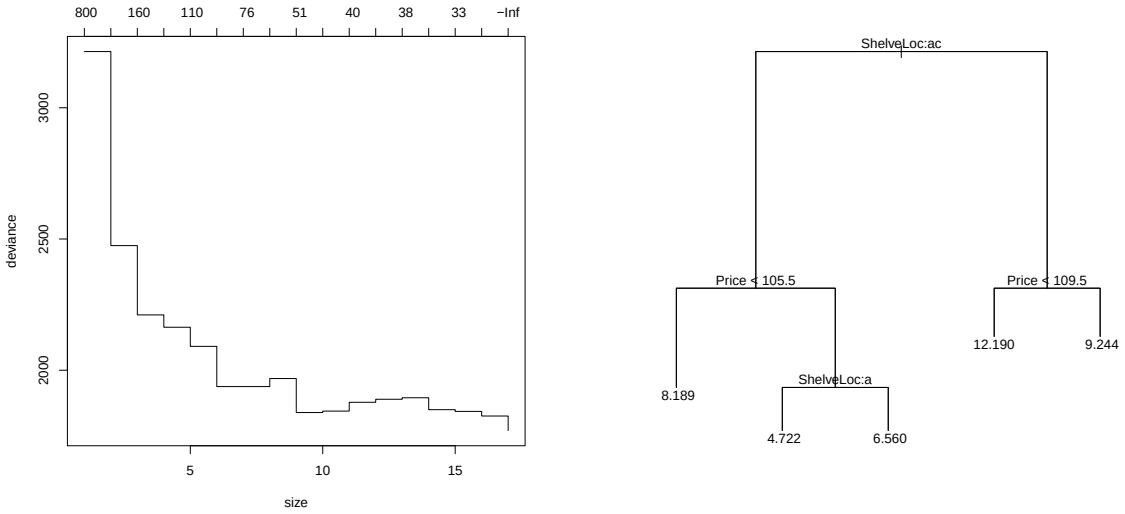


Figure 13: Validation plot produced using the `tree` library (left) and the corresponding pruned tree (right, using 5 terminal nodes).

The validation plot produced using the `tree` library looks very similar to Figure 10 (note the elbow at 5 terminal nodes). Indeed, the figure produced by the `rpart` library uses cross-validation to generate the error bars around the point sequence in Figure 10, and so essentially communicate the same information. The points are simply the mean of the validation errors over $K = 10$ folds.

3 Neural Networks

For this section, we follow the notation of the suggested literature, Nielsen (2015). The textbook is freely available online and provides an excellent graphical introduction to the basics of neural networks.

3.1 Model Machinery: Structure and Notation

Neural networks are a class of non-linear models typically used in **supervised learning** tasks (although they can easily be adapted to unsupervised learning tasks). The term 'neural' stems from early research on the topic where the function of biological neurons was used as a metaphor for explaining what the models *do*. As always, one should take care to not confuse a good metaphor for rigorous mathematics. The term 'network' describes the graphical representation of such models, where neural networks are illustrated by way of a directed graph, with nodes and edges representing **activations/activation functions** and **weights/parameters** of the model, respectively. To see this, first consider a standard multiple linear regression problem: We have a set of responses $\{y_i : i = 1, 2, ...N\}$ and predictors $\{\mathbf{x}_i : i = 1, 2, ...N\}$. We then model the relationship between the responses and predictors as:

$$y_i = \mathbf{x}_i^T \boldsymbol{\beta} + \epsilon_i; \quad \epsilon_i \sim \mathrm{N}(0, \sigma^2).$$

This mathematical relationship can be illustrated using a directed graph as in Figure 14.

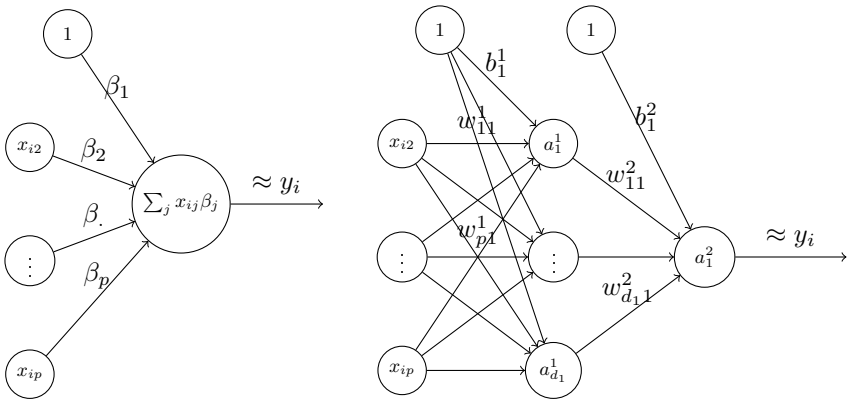


Figure 14: Directed graph of a normal linear model (left) vs. that of a standard feed-forward neural network, specifically, a (d_1)-network (right). Weights shown for the first columns of $\mathbf{W}_1$ and $\mathbf{W}_2$.

Nodes on the left of the graph depict the predictors. The node on the right contains the response. The predictor/**input** nodes are then connected with the response/**output** nodes by edges. The mathematical relationship is then captured by assigning parameters to each edge and then letting the cell of the node represent some mathematical translation. In this case, the translation is: take the input cells, multiply them by the parameters on their connecting edges, and aggregate these to perform a prediction for the response. I.e., $x_{i0}\beta_0 + x_{i1}\beta_1 + x_{i2}\beta_2 + x_{i3}\beta_3 ... = \mathbf{x}_i^T\boldsymbol{\beta}$. This extends to logistic regression as well, only, we have $\sigma(\mathbf{x}_i^T\boldsymbol{\beta})$ where $\sigma(x) = e^x/(1+e^x)$ from the logit transform, and the cell now represents the calculation $e^{\mathbf{x}_i^T\boldsymbol{\beta}}/(1+e^{\mathbf{x}_i^T\boldsymbol{\beta}})$ with the graph representing the function:

$$\pi_i = \sigma(\mathbf{x}_i^T\boldsymbol{\beta}); \quad y_i \ \sim \ \text{Be}(\pi_i).$$

Noting that the input cells do not transform the predictors in any way, why not replace them with some mathematical translation as well? In fact, why not use the same or a similar translation repeatedly, but with potentially different parameters? Evaluation of the model then becomes **recursive**, adding **layers** of transformations of the inputs to the network. Figure 14, for example, depicts the addition of one so-called **hidden** layer (since the nodes in this layer do not contain any direct observations, their *state* cannot be *known*). This hidden layer now becomes the inputs to the final layer in standard fashion. Neural networks arise as a generalisation of this idea: by simple repeated iteration of translations of the input (the number of repetitions giving the **depth** of the network) and specification of the calculation performed in each cell, we create a **non-linear** function of the input/predictor space which maps to the output.

In general, a standard feed-forward neural network can be parametrised as follows: Let a_j^l denote the j-th node on the l-th layer of a standard feed-forward neural network, then the network structure can be written as an updating equation:

$$a_j^l = \sigma_l\left(\sum_{k=1}^{d_{l-1}} a_k^{l-1} w_{kj}^l + b_j^l\right), \quad l = 1, 2, \ldots, L; j = 1, 2, \ldots, d_l,$$

($L-1$ hidden layers) where:

- $\sigma_l(.)$ denotes an activation function on layer l (e.g., sigmoid or hyperbolic tangent),
- d_{l-1} denotes the number of nodes in layer $l-1$ (the number of nodes in layer l is d_l),
- w_{kj}^l denotes the kj-th weight parameter linking the k-th node in layer $l-1$ and j-th node in layer l (although we transfer information from layer $l-1$ to layer l, we use the convention of indexing the weights using the superscript l—corresponding to the forward layer),

- b_j^l denotes the j-th bias in layer l,
- and the equation is evaluated subject to the initial conditions $a_j^{(0)} = x_{ij}$ for all j at the i-th training example.[10]

As complicated as this looks, the model equation can be captured in a simple **vector form** as:

$$\mathbf{a}^l = \sigma_l(\mathbf{W}_l^T \mathbf{a}^{l-1} + \mathbf{b}^l); \quad l = 1, 2, ...,$$

where $\mathbf{a}^l = [a_1^l, a_2^l, ...]^T$ denote vectors of activations and the parameters of the model are encapsulated in the **weight matrices** $\mathbf{W}_l = (w_{kj}^l)$, and **bias vectors** $\mathbf{b}^l = [b_1^l, b_2^l, ...]^T$, and the initial value(s) for the equation is $\mathbf{a}^0 = \mathbf{x}_i$. So, for example, in the case of a (3)-network with two inputs on a single response, we have

- dimensions $d_0 = 2$, $d_1 = 3$, $d_2 = 1$.
- Inputs $\mathbf{x}_i = [x_{i1}, x_{i2}]^T$.
- Evaluation of layer 1:
$$\mathbf{a}^1 = \sigma_1(\mathbf{W}_1^T \mathbf{x}_i + \mathbf{b}^1) \quad \text{(a } 3 \times 1 \text{ matrix)}.$$
- Evaluation of layer 2 (the output layer):
$$\mathbf{a}^2 = \sigma_2(\mathbf{W}_2^T \mathbf{a}^1 + \mathbf{b}^2) \quad \text{(a } 1 \times 1 \text{ matrix)}.$$
- $\mathbf{W}_1$ is a (2×3)-matrix, $\mathbf{W}_2$ is a (3×1)-matrix, $\mathbf{b}^1$ is a (3×1)-matrix, and $\mathbf{b}^2$ is a (1×1)-matrix.
- $\mathbf{a}^2$ supplies the prediction for response y_i (hence $\approx y_i$).
- Note that this evaluation is for a single observation! We need to apply the above steps for each observation in our data set.

As you can see, evaluation of a network is actually rather straight-forward (pun not intended) so don't be intimidated by the mystique of model class: This equation can be read simply as a potentially highly non-linear function in the predictors, almost like a flexible species of polynomial—if we add more nodes and layers, the degree of complexity increases and vice versa, like the order of a polynomial.

Before we continue with more detail, consider the following simulated data: For purposes of modelling

Variable	Description
`Y`	Real-valued response variable containing noise (not Normal).
`X`	Covariate.

this data, we make use of the `neuralnet` library in R. Since we have a single covariate and response, we know the dimensions of the input and output layers: one node in each. It is left to the user to specify the remaining dimensions of the model. We fit a (7)-network to the data as follows: Note that the nomenclature for the model specification specifies the number of nodes in each if hidden layers. Here, we have a single hidden layer with seven nodes:

```
R> set.seed(2019)
R> N = 200
R> X = runif(N,-1,1)
R> Y = sin(3*pi*X)+rgamma(N,1,2)
R>
R> Simulation_1 = data.frame(Y=Y,X= X)
R>
R> library(neuralnet)
R>
R> mod1 = neuralnet(Y~X, hidden = c(7), data = Simulation_1)
R> plot(mod1)
```

[10] The updating equation thus needs to be evaluated for each training example.

Alternatively, we could fit a $(7, 2)$-network using:

```
R> mod2 = neuralnet(Y~X, hidden = c(7,2), data = Simulation_1)
R> plot(mod2)
```

Figure 15 gives the directed graphs for the two models respectively. These are produced by calling the `plot()` function on a `neuralnet` model object. The graphs are annotated by giving the numerical values for the *optimal* weights calculated during the model fit procedure as well as some basic diagnostics for the model fit procedure—we'll revisit these diagnostics shortly.

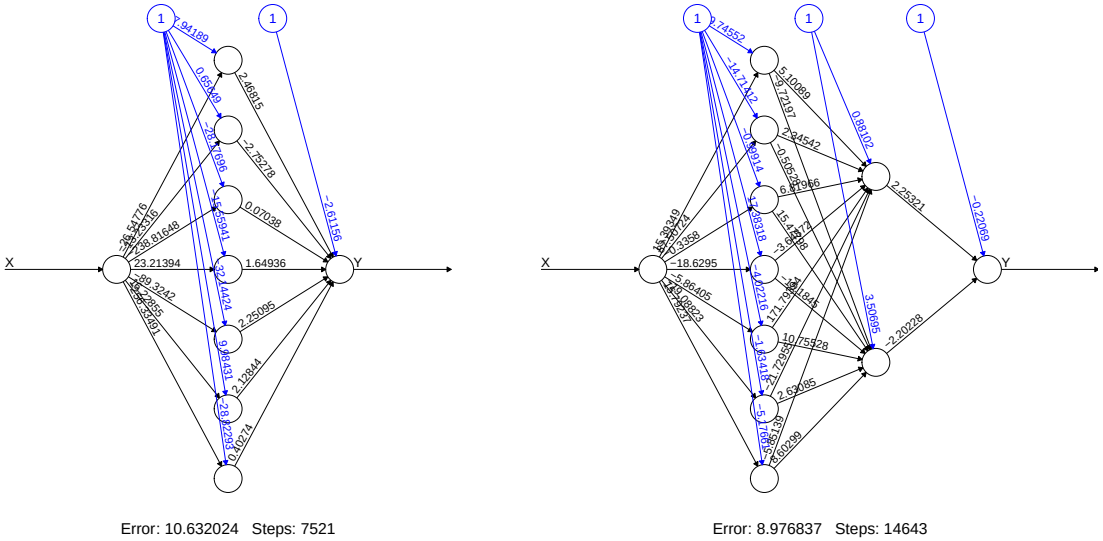


Figure 15: (7)-network vs a $(7, 2)$-network fitted to a simulated regression problem.

Although the aforementioned plots are useful for communicating the architecture of the model, and it is sometimes useful to analyse the magnitudes of the weight parameters, they do not give us any indication of **goodness of fit**. For these purposes, we can evaluate the predicted responses over the input space (we just evaluate our non-linear function as a function of x here). In R, we create a regularly spaced set of x-values, calculate the predicted values under the fitted model(s), and super-impose this on the data. The resulting plots are given in Figure 16. Indeed, both the (7)-network and $(7, 2)$-networks produce non-linear **response functions**. But which one is best? Or is either one even satisfactory? Compare the fitted models in Figure 16 to the true **data generating process (DGP)**, given by:

$$y_i = \sin(3\pi x_i) + \epsilon_i; \quad \epsilon_i \sim \text{Gamma}(1,2), \; x_i \sim \text{U}(-1, 1).$$

A naive interpretation would be that since both are quite close to the true underlying pattern, they must be accurate representations of the DGP. A more accurate analysis would be that either model has sufficient degrees of freedom to approximate the underlying process, but it is most likely that we have **overfit** the data. That is, our models are **a priori** capable of producing more complex functions than the underlying pattern and we need to account for this explicitly while fitting models to the data in order to produce a valid predictive model.

In order to remedy this, we need to understand two things: We need to know how the models are fitted in the first place, and then we need to introduce a mechanism for managing model complexity, as we did for tree-based methods.

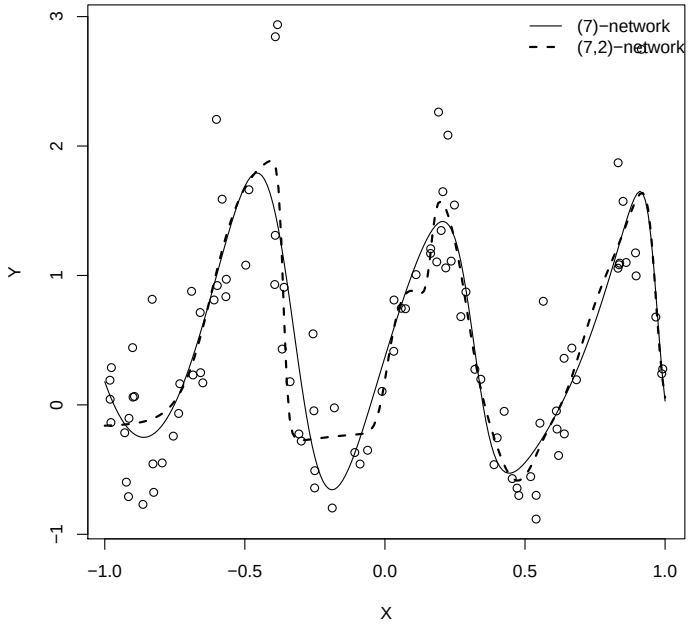


Figure 16: Fitted patterns for (7)-network vs a $(7,2)$-networks vs. the observed responses.

3.2 Back-Propagation and Gradient Descent

Neural networks are fitted using a standard optimisation technique called **gradient descent**. A heuristic explanation of gradient descent follows: Suppose we are on a surface in $\mathbb{R}^2$ defined by the function $g(x, y)$. We wish to find the coordinate $\mathbf{x}^\star = [x^\star, y^\star]^T$ at which this surface is a minimum. Supposing that we start at some point $\mathbf{x}_0 = [x_0, y_0]^T$, we can decide which direction to move by evaluating the gradient of the surface where we are now and find which direction is *downward* from our present position and take a step in that direction. In mathematical terms, we can encode this as:

$$\mathbf{x}_{new} = \mathbf{x}_0 - \lambda \nabla g(\mathbf{x}_0),$$

where $\nabla g(\mathbf{x}_0)$ evaluates the partial derivatives of $g(.)$ (the gradients), and we take a step that is proportional to the size of the gradient (if it's steep we step further in that direction), but we scale the magnitude using λ which is referred to in the present context as the **learning rate**. Note that this equation looks quite similar to Newton's method and even the method of scoring. Indeed, the corresponding iterative equation follows:

$$\mathbf{x}_{i+1} = \mathbf{x}_i - \lambda \nabla g(\mathbf{x}_i),$$

for $i = 1, 2, \ldots$ some maximum number of steps are taken or a predefined **tolerance** on the gradients (or step size), $|\nabla g(\mathbf{x}_i)|$, is reached.

How this relates to the present class of models is by way of the objective functions used to fit neural networks. Recall that neural networks can be viewed simply as non-linear functions. Let's say, $f_{NN}(\mathbf{x}, \boldsymbol{\theta})$, denotes a neural network where the vector $\boldsymbol{\theta}$ encapsulates all of the parameters of the model. I.e., $\boldsymbol{\theta}$ contains all the elements in the set $\{w^l_{kj}, b^l_j \ \forall \ k, j, l\}$. Then we may, in the case of a regression problem

define the **mean square error** objective function:

$$g(\boldsymbol{\theta}, \mathbf{X}, \mathbf{y}) = \frac{1}{N}\sum_{i=1}^{N} g_i(\boldsymbol{\theta}, \mathbf{x}_i, y_i) = \frac{1}{N}\sum_{i=1}^{N}(f_{NN}(\mathbf{x}_i, \boldsymbol{\theta}) - y_i)^2,$$

which we would seek to minimize. This can then be achieved by application of gradient descent: Starting with some initial set of parameters, $\boldsymbol{\theta}_0$, we evaluate:

$$\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i - \lambda \nabla g(\boldsymbol{\theta}_i)$$

for $i = 1, 2, ...$ until the gradients of the objective function are sufficiently small (the objective surface is flat), at which point you find $\hat{\boldsymbol{\theta}}$. In the previous section, where we fitted neural networks to a simulated data set, we noted that some diagnostics are printed when one plots the model objects. Figures 15 (a) and (b) each report the number of iterations or steps taken until convergence was achieved along with an error metric 'MSE', which corresponds to the objective function evaluated at the final iteration. For example, for the (7)-network, we have $\hat{\boldsymbol{\theta}} = \boldsymbol{\theta}_{7521}$ with an MSE of 10.632. Previously, we used the default parameters for λ and the tolerance on the gradient. These can be defined manually, for example:

```
R> mod1 = neuralnet(Y~X, hidden = c(7)  ,learningrate = 0.1,threshold = 0.01, data =
    Simulation_1)
R> mod2 = neuralnet(Y~X, hidden = c(7,2),learningrate = 0.1,threshold = 0.01, data =
    Simulation_1)
```

Back-propagation, hailed as a revolutionary idea in the world of machine learning, concerns a single aspect of the model fit procedure: calculation of the gradients of such objective functions for neural networks. Although calculating the first derivatives of elementary functions by application of the chain rule is hardly revolutionary, the strategy is lauded for its elegance. This follows since back-propagation provides simple iterative relations for calculating the gradients of an objective function w.r.t. weights and biases in the network. For example, it can be shown that (you'll cover this in honours):

$$\mathbf{d}_l = \frac{\partial g_i}{\partial \mathbf{b}^l} = \mathbf{W}_{l+1}\mathbf{d}_{l+1} \odot \sigma_l'(\mathbf{W}_l^T \mathbf{a}^{l-1} + \mathbf{b}^l)$$
$$\frac{\partial g_i}{\partial \mathbf{W}^l} = \mathbf{a}^{l-1}\mathbf{d}_l^T$$

for $l = L-1, L-2, ...$ where $\sigma_l'(.)$ denotes the derivative of the activation function in layer l and the **terminal condition** for the equations are given by:

$$\mathbf{d}_L = \frac{\partial g_i}{\partial \mathbf{a}^L} \odot \sigma_L'(\mathbf{W}_L^T \mathbf{a}^{L-1} + \mathbf{b}^L),$$

noting that the prediction from the network, $f_{NN}(.)$, in fact corresponds to the activation in the final layer, $\mathbf{a}_L$. Note that these equations closely resemble the evaluation of the forward equations which define the neural network. Note also that the index runs backwards from the final layer, hence 'back'-propagation.

Depending on the software being used, various derivatives (alterations of the above) of back-propagation and gradient descent may be used. See for example the `algorithm` option in `neuralnet`, where the specification `'backprop'` refers to the above, and a number of alternative algorithms are given.

Architecture

Other aspects to consider w.r.t. neural network architectures are the various specifications of **activation functions** and objective functions. Numerous **activation functions** exist with the most commonly used

variants for neural networks being **logistic**, **hyperbolic tangent**, and **rectified linear units (ReLU)**. These in-turn are defined by the expressions:

$$\sigma(z) = \frac{e^z}{1+e^z} = \frac{1}{1+e^{-z}} \quad \in \quad [0,1],$$

for the logistic activation.

$$\sigma(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} = \frac{e^{2z}-1}{e^{2z}+1} \quad \in \quad [-1,1],$$

for the so-called *tan-h* function, and finally

$$\sigma(z) = \max(0, z) \quad \in \quad \mathbb{R}^+,$$

for the rectified linear units. Although any combination of these can be assigned to the hidden units, of the network (though we typically define a uniform specification across hidden units), care needs to be taken to assign an appropriate activation function to the final/output layer. For **regression problems** we typically assign the **identity functioon** ($\sigma_L(x) = x$) to the output layer, whilst for **classification problems** we typically use a specification such as the logistic function, which map to the interval $[0, 1]$., so that the predictions from the network correspond to probabilities[11].

Depending on the context, we also need to modify the objective function. In regression problems, we typically use mean square error for the objective function as before. For **classification problems**, a more appropriate choice[12] of objective would be the so-called **cross-entropy error**[13] function:

$$-\frac{1}{N}\sum_{i=1}^{N}(y_i \log(f_{NN}(\mathbf{x}_i, \boldsymbol{\theta})) + (1-y_i)\log(1 - f_{NN}(\mathbf{x}_i, \boldsymbol{\theta}))$$

where $y_i \in 0, 1$ in binary classification problems and

$$-\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{d_L} y_{ij} \log(f_{NN}(\mathbf{x}_i, \boldsymbol{\theta}))$$

where $y_{ij} = 1$ if the response in in class j and 0 otherwise for multi-class problems. Note again that $f_{NN}(.)$ corresponds to a_j^L, the j-th output on the final layer, for the i-th training example—the need for the output layer to map to probabilities should now be obvious.

Pregnancy duration data:
The following data are taken from Neter et al. (1996). The study concerns the association between several risk factors and the duration of pregnancies. The dataset consists of 102 observations on the following variables:

Variable	Description
`Cat`	Category for the response variable, 1,2, or 3.
`D1`	Dummy variable for the response: duration < 36 weeks (preterm).
`D2`	Dummy variable for the response: duration $36 - 37$ weeks.
`D3`	Dummy variable for the response: duration > 37 weeks (full term).
`Nutrition`	Covariate for nutritional status. Higher values indicate better nutritional status.
`Age21.30`	Dummy variable for the age group $21 - 30$ years of age.
`Age30`	Dummy variable for the age group > 30 years of age.
`Alc`	Dummy variable for Alcohol use, $1 \Rightarrow$ history of alcohol use.
`Smoke`	Dummy variable for smoking status: $1 \Rightarrow$ smoker.

[11] In multi-valued classification problems we typically assign a special kind of activation function which produces a discrete distribution over the outputs.
[12] We detail the technical reasons for this departure in honours and masters.
[13] Not to be confused with the Shannon-entropy of earlier

Note that the response is categorical, and can take on one of three values. The response was thus encoded as:

$$y_{i1} = \begin{cases} 1 & \text{if duration} < 36 \text{ weeks}, \\ 0 & \text{otherwise}, \end{cases}$$

$$y_{i2} = \begin{cases} 1 & \text{if duration } 36 \text{ - } 37 \text{ weeks}, \\ 0 & \text{otherwise}, \end{cases}$$

and

$$y_{i3} = \begin{cases} 1 & \text{if duration} > 37 \text{ weeks}, \\ 0 & \text{otherwise}, \end{cases}$$

for pre, mid, and full term pregnancy.

Fortunately, software packages make it quite easy to accommodate such specifications. Using the `neuralnet` package, for example, we simply specify the model equation as `D1+D2+D3~Nutrition+Age21.30+Age30+Alc+Smoke`. Furthermore, we can also specify the appropriate objective function by passing an argument to the `err.fct = "ce"` parameter. For purposes of the analysis, we fit a (5)-network with logistic activations to the data (note: the model will have 3 output nodes, but we specify the model in terms of the hidden layers), using the standard back-propagation algorithm, with a learning rate of $\lambda = 0.01$, and a threshold on the stepsize of 0.01. We'll also initialise the parameter set $\boldsymbol{\theta}_0$ by drawing 48 random $\mathrm{U}(-2, 2)$ deviates (there are 48 parameters in total in this model, weights and biases included):

```
R> rm(list = ls())
R> library(neuralnet)
R> PREG = read.table('Section 3.2 - Pregnancy Duration Data.txt', h = T)
R> head(PREG,5)
```

```
  No Cat D1 D2 D3 Nutrition Age21.30 Age30 Alc Smoke
1  1   1  1  0  0       150        0     0   0     1
2  2   1  1  0  0       124        1     0   0     0
3  3   1  1  0  0       128        0     0   0     1
4  4   1  1  0  0       128        1     0   0     1
5  5   1  1  0  0       133        0     0   1     1
```

```
R> set.seed(2019)
R> PREG$Nutrition = scale(PREG$Nutrition)
R> mod1 = neuralnet(D1+D2+D3~Nutrition+Age21.30+Age30+Alc+Smoke, hidden = c(5), data = PREG,
+          learningrate = 0.01,threshold = 0.01, algorithm = 'backprop',
+          stepmax = 10^6, err.fct = "ce", linear.output = FALSE,lifesign + ='full',
+          startweights = runif(48,-1,1),act.fct = 'logistic')
R> plot(mod1)
```

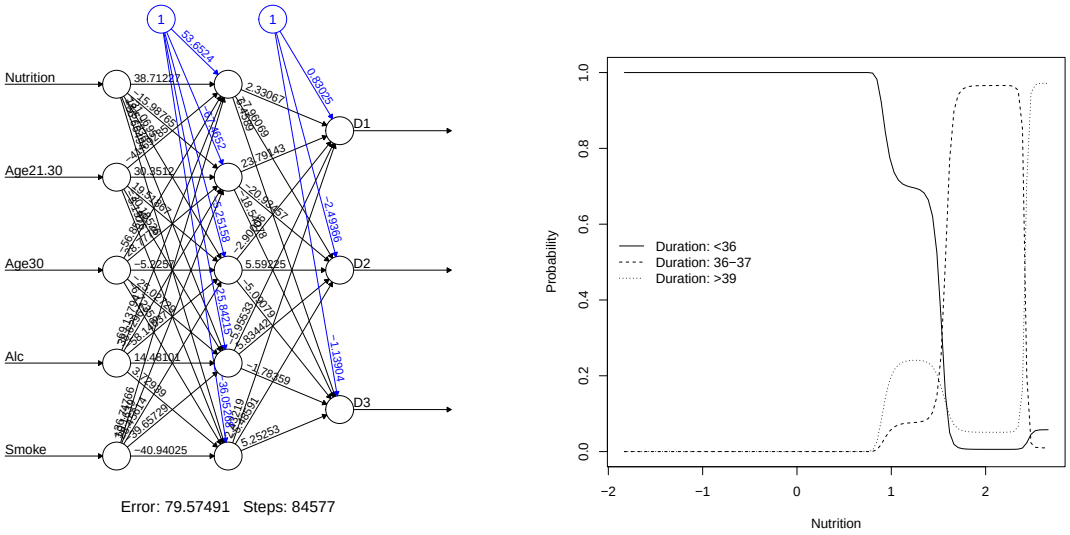


Figure 17: Left: (5)-network fitted to the pregnancy duration data. Note that the output layer has three nodes corresponding to the response encoding. Right: Drawing a response (to nutrition) curve for an individual aged 21-30 with a history of alcohol use and smoking, we see that the risk of preterm labour is increased for poor nutrition.

As mentioned before, by virtue of the fact that the model is non-linear, it is difficult to interpret the fitted model as is. However, we can use the model to predict responses for various specifications of the inputs. We may, for example, evaluate the effect of nutrition for an individual in the age group 21-30 who both smokes and uses alcohol, by fixing the relevant explanatory variables and drawing a response curve (how the probabilities of being in a particular response category change w.r.t. the input) over the observed range of the nutrition covariate. This can be achieved by feeding the network a regularly spaced sequence of nutrition values, with the other factors fixed at the appropriate value and plotting the predicted probabilities under the model. In R:

```
R> # Draw the response curve under the present model:
R> #  -- Create a sequence of values for which to predict:
R> dummy = seq(min(PREG$Nutrition),max(PREG$Nutrition), length = 100)
R> XNEW = data.frame(Nutrition = dummy, Age21.30 = 1,Age30 = 0,Alc = 1, Smoke = 1)
R>
R> # -- Plot the predicted probabilities:
R> pred = predict(mod1,XNEW)
R> plot(pred[,1]~dummy, xlab ='Nutrition', ylab = 'Probability', ylim = c(0,1), type = 'l')
R> lines(pred[,2]~dummy, lty =2)
R> lines(pred[,3]~dummy, lty =3)
```

Using the response curve as a means to interpret the model, we observe that the likelihood of pre-term pregnancy is significantly increased for individuals with poor nutrition. Furthermore, mid or full term pregnancies are only expected in individuals from this cohort (age and smoking status) for nutrition levels above 1.9 (scaled).

3.3 Managing Complexity: Regularisation

Although neural networks provide a very general platform for analysing a wide range of data, they are not without flaw. Indeed, the analysis up to this point is at best only half of the story... To see this, re-run the analysis of the pregnancy duration data using a different seed-value, say `set.seed(2020)`—see Figure 20. Although the resulting response function leads to similar analysis of the relationship between nutrition and pregnancy duration, the predicted response function differs from what was observed before.

Indeed, this is actually not a surprising result: When working with over-parametrised models, we know *a priori* that there are multiple solutions to the objective optimisation problem. As a consequence, when the parameters are initialised using different starting points, the trajectory of the gradient decent process may converge to local minima on the objective surface corresponding to different models which have similar predictive performance. This is particularly prevalent in neural networks. Another, more broadly applicable issue is that of **model variance** due to complexity. Reiterating from Section 2.3, in general, models that are capable of producing complex patterns are quite sensitive to the data used to estimate its parameters. This follows since, a model which is sufficiently complex to replicate the observed responses exactly (when predictions equate to the observations), the resulting model would vary wildly of we were able to repeatedly draw samples from the underlying process. Figure 18 demonstrates the phenomena by fitting a (5)-network to samples drawn repeatedly from a data generating process:

$$y_i = 2 + 2x_i + \sin(1.5\pi x_i) + \epsilon_i; \quad \epsilon_i \sim \mathrm{N}(0,1), \; x_i \sim \mathrm{U}(-1,1). \tag{1}$$

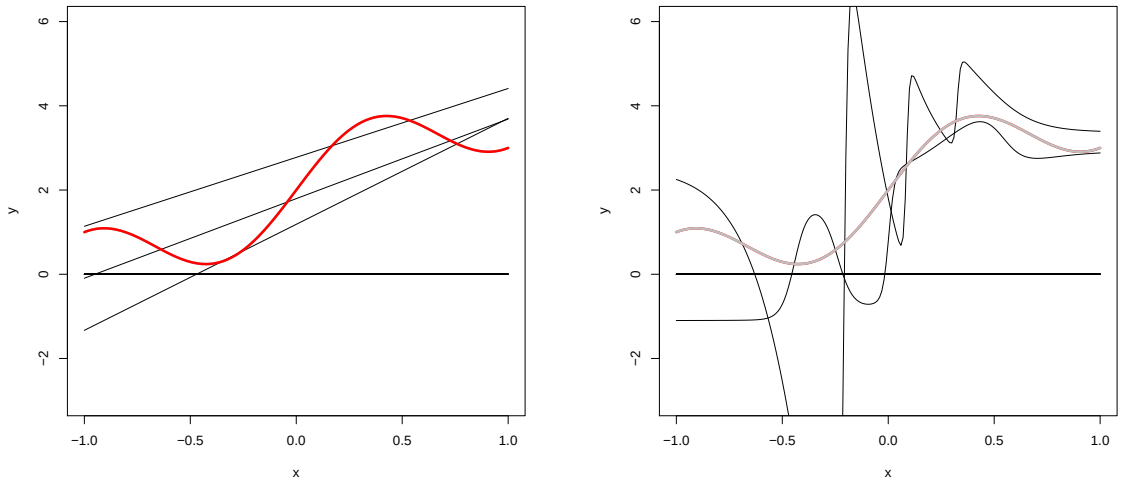


Figure 18: Linear model (left) and (5)-network fitted to repeated draws of $N = 20$ observations from the DGP in (1). Although the linear model cannot replicate the true patter, the fitted model varies significantly less over samples and will, in fact, on average be closer to the true pattern (red) than the neural network which can replicate the true pattern exactly.

Comparing this to a simpler model, say a simple linear regression, we see significantly more variation in the more complex model. Now, although this variation is obviously something we would want to avoid, it is also clear that the linear regression would not ever be able to produce the true underlying pattern. The answer is thus to find a model which is somewhere in between these two extremes. Now, a naive approach in the context of neural networks is to enumerate the number of hidden nodes to create a sequence ranging from a linear model to a the aforementioned network—like varying the order of a polynomial. Though there are contexts in which this is appropriate, for various technical reasons this is not advised in the context of machine learning — more on this in honours. Instead, what we do is we typically start with an over-specified model and then constrain the model systematically to limit its complexity. As in the case of tree-based methods, this is achieved by penalising fit, or as it is known in the modern literature, **regularisation**. For these purposes, we add a penalty term to the objective function. For example, in the case of a regression problem:

$$\frac{1}{N}\sum_{i=1}^{N}\left(f_{NN}(\mathbf{x}_i, \boldsymbol{\theta}) - y_i\right)^2 + \frac{\nu}{N}\sum_{j,k,l}(w_{jk}^l)^2,$$

where the additional term is referred to as "**L2-regularisation**". By increasing the ν parameter, we increase the penalty incurred and vice versa. Setting $\nu = 0$ results in an unconstrained model, with high variance. Setting ν to a large value overconstrains the model, resulting in a simpler model, i.e., one with increased bias, but lower variance. The procedure for fitting a neural network thus becomes: 1) Start with an overspecified model, 2) find a value for ν that balances the complexity of the model. To find such a value, we fit the model for increasing values of ν and monitor the error on the validation set. Figure 19 demonstrates the effect of increasing the amount of regularisation on a single draw of $N = 50$ observations from the DGP in (1). The point at which the validation error reaches a minimum gives the appropriate value of ν. Note that other forms of regularisation exist. Indeed, the form of the penalty function has implications for the underlying optimisation problem—more on this in honours. However, for most applications the simple forms L2 and L1[14] are sufficient.

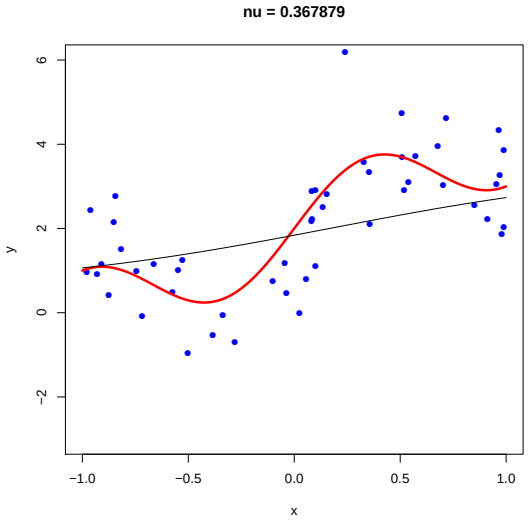


Figure 19: (5)-network fitted to a single draw of $N = 50$ observations from the DGP in (1) for increasing values of ν under L2-regularisation.

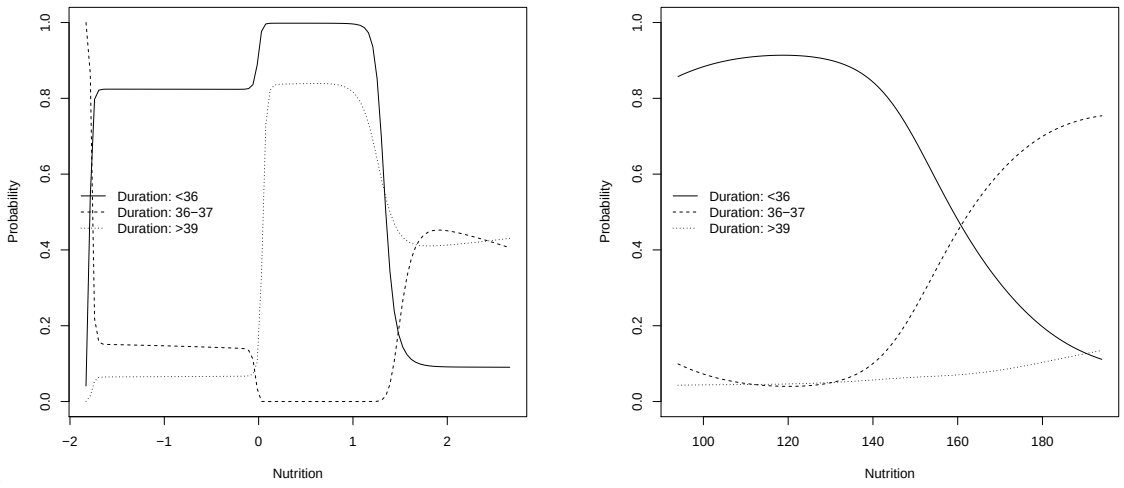
Interestingly, the `neuralnet` package does not provide any interface for applying such regularisation and is thus only suited to simple problems with a sufficient amount of data. For purposes of fitting a balanced model, we turn to more modern (albeit more clunky) software. For example, the `h2o` package provides routines for fitting quite advanced neural network architectures and allows explicitly for the inclusion of regularisation terms. Revisiting the pregnancy data example, we fit a (5)-network using the `h2o` package:

```
R> PREG = read.table('Section 3.2 - Pregnancy Duration Data.txt', h = T)
R> head(PREG,5)
R>
R> PREG$Cat = factor(PREG$Cat)
R>
R>
R> library(h2o)
R> h2o.init() # Initialize h2o
R>
R> # Create a data frame and cast it as an h2o object:
R> frm = PREG[,-c(1,3:5)]
R> train = as.h2o(frm)
R>
R> # Fit a (5)-network to the data
```

[14] For L1-regularisation, we use $\sum_{j,k,l} |w_{jk}^{(l)}|$.

```
R> model <- h2o.deeplearning(
+  x = 2:7,
+  y = 1,
+  training_frame = train,
+  distribution = "multinomial",
+  hidden = c(5),
+  activation='Tanh',
+  l2 = 0.01,
+  epochs = 250000,
+  loss = 'CrossEntropy',
+  stopping_tolerance = 0.01,
+  mini_batch_size = 102,
+  rate = 0.05)
R>
R> # Plot yhe response function:
R> dummy = seq(min(PREG$Nutrition),max(PREG$Nutrition), length = 100)
R> XNEW = data.frame(Nutrition = dummy, Age21.30 = 1,Age30 = 0,Alc = 1, Smoke = 1)
R>
R> pred = h2o.predict(model, newdata = as.h2o(XNEW)) # Predict for this particular set of
    predictors.
R> pred = as.data.frame(pred)
R> plot(pred$p1~dummy, xlab ='Nutrition', ylab = 'Probability', ylim = c(0,1), type = 'l')
R> lines(pred$p2~dummy, lty =2)
R> lines(pred$p3~dummy, lty =3)
```

Here the `as.h2o(frm)` command casts the data frame into an object class which can be recognised by the `h2o` back-end (which is passed to the `training_frame` argument) and `h2o.deeplearning()` is the function which fits the neural network by way of back-propagation and gradient descent. The variabels `x` and `y` give the column indices of the input and output variables respectively. That is, the response is on column 1 (`y = 1`) and the predictors on columns 2-7 of the training frame (`x = 2:7`). The remainder of the variables are somewhat self-explanatory, but for purposes of regularisation, the relevant parameter is that of `l2 = 0.01`, which sets $\nu = 0.01$, thus penalising the cross-entropy error objective, effectively constraining the parameters of the model to produce a simpler model. As a consequence, the neural network produces a smoothed response curvem (Figure 20). Indeed, this produces more stable response curve for the present problem from which we can more reliably make inference about the likelihood of pre-term durations as a function of nutrition.



5

Figure 20: Left: Model fitted using `neuralnet()` but with `set.seed(2020)`. Right:, (5)-network fitted using L2-regularisation ($\nu = 0.01$), producing a smoothed response curve.

Houising data (revisited)
Although neural networks do not ecxhibit the ease of interpretation that tree-based methods do, neural networks do offer significant advantages in terms of replicating local features in the data. Indeed, the although we pay the price for the ability to replicate complex patterns, this can often lead to more realistic representations of the data under the fitted model. This can, as one example, be easily demonstrated by the housing data example from Section 2.1. Observant readers should have spotted a rather fundemental downside to the tree-based approach of partitioning along the marginal dimensions of the feature space: *A priori* there is no reason for the data to be orientated in such a way that it is optimal to partition along those dimensions. That is, why should the optimal split-points lie along the axes of latiude and longitude if the data is clearly orientated along an axis more parrellel to the coastline? Indeed, if this is the case, surely the regression tree will artificially create too many partitions in an attempt to capture the underlying pattern. Fortunately, neural networks are orientation-agnostic in the feature space, and can capture both global and local features in the data with little modification of the data—provided of course that we have enough data.

Fitting a $(50, 10)$-network to the housing data, using only latitude and longitudes as inputs with $\nu = 0.0001$, we may map a predicted response surface and superimpose this on a map of the housing data—see the R code below. Figure 21 shows a colour-scaled contour map of the surface with the original data points overlayed, demonstrating various local peaks and troughs in the predicted median house prices. Furthermore, the overall shape of the contours match closely the diagonal orientation of the data.

```
R> dat = read.csv('Section 2.1 - California Housing Data.csv',h = T)
R> dat$median_house_value = log(dat$median_house_value)
R>
R> library(h2o)
R> h2o.init()
R>
R> frm = dat
R> train = as.h2o(frm)
R>
R> model <- h2o.deeplearning(
R>  x = 1:2,
R>  y = 9,
R>  training_frame = train,
R>  hidden = c(50,10),
R>  activation='Tanh',
R>  l2 = 0.0001,
R>  epochs = 2000,
R>  loss = 'Automatic')
```

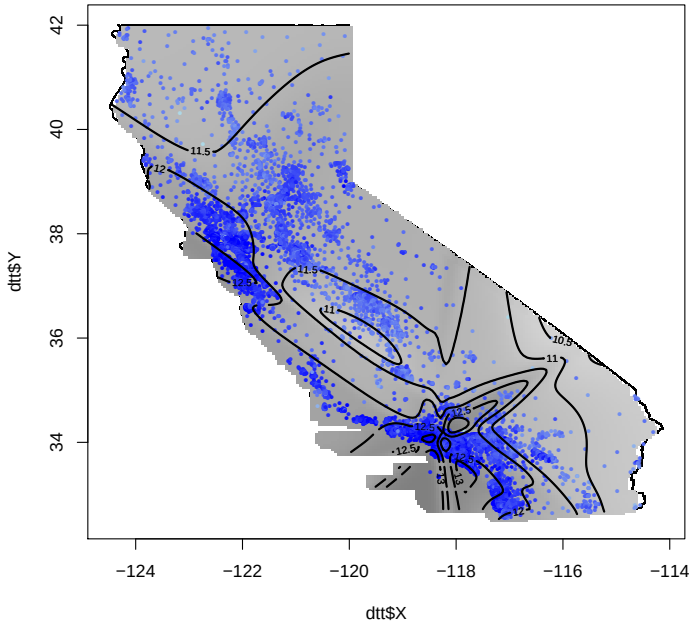


Figure 21: Predicted values for log-meadian house price calculated by fitting a $(50, 10)$-network to the California Housing Data set. Compared to a regression tree, the neural network is able to capture local features more accurately and is implicitly orientation-agnostic in the feature space.

4 Closing Remarks

So how do the model classes covered here constitute an introduction to 'Machine Learning'? Indeed, as you may have come to appreciate by now, the departure point for Machine Learning in its purest form is not with neural networks or tree based methods at all, but rather with the (mathematical) ability of a given algorithm to learn. That said, these model classes are appropriate for the present context in that they exhibit traits that are common to all machine learning applications. First, the models are algorithmic in nature, and thus less structured than traditional statistical model classes (GLMs, GAMs, etc,). The benefits of having less structured models are that such models lend quite naturally to domains where the analysis requires one to extract complex non-linear patterns in the data, for example, recognising objects in high resolution media (image or video), pattern recognition in speech and natural language processing, and complex decision rules in artificial intelligence. Unfortunately, what we gain in performance, we typically lose in interpretability, not least because of the "black-box" nature of the algorithms but also the dependence on large amounts of high resolution data and the need for the underlying pattern to *dominate* the noise/variation in the data — key assumptions in the practical application of machine learning (Abu-Mostafa et al., 2012).

At this stage it is perhaps best to orientate oneself and consider what one knows about machine learning rather than fuss about what it is (or is not). For these purposes, we highlight the algorithms covered here in the broader context of machine learning algorithms and knowledge which needs to be attained in order to qualify oneself as a certified practitioner in the field of machine learning. Though not an exhaustive list, Table 1 outlines relevant machine learning concepts and where they are covered in the syllabus of various courses offered by the Department of Statistical Sciences at the University of Cape Town.

Course	Level	Scope
STA304X - GLMs, ML, Bayesian Analysis.	UG	• Fit a decision/regression tree. • Fit a simple artificial neural network (ANN).
STA40XX - Analytics.	H	• Tree-based methods. • Random Forests. • Boosted Trees. • High-Performance Computing. • Code ANNs from scratch. • Validation Analysis for ANNs. • Implement Backpropagation. • Feature Engineering. • K-means/medoids • Hierarchical clustering • Model-based clustering • Self-organising Maps
STA5068 - Machine Learning	M	• The Machine Learning Paradigm. • Generalisation Error and Hoeffding Inequality • VC-Dimension and Calculus of Complexity. • Model Complexity and Learning Feasibility • Probability of Overfitting • Perceptron Learning: *MLP, ANNs, Backpropagation, and Convolutional NNs.* • Linear Dimension Reduction. • Dimensional Reduction: *Autoencoders, Non-linear Dimension Reduction, Wavelet Transforms.* • The Support Vector Machine (SVM): *Kernel Methods, VC-Dimensions of SVMs and generalisation error.* • Managing Algorithm Failure: *design, code, build and modifying ML algorithms to manage model risk and algorithm failure through testing.* • Introduction to Adversarial Learning. • Introduction to Similarity based methods.
STA5086 - Advanced Portfolio Theory	M	• Introduction to Stochastic Dynamic Programming. • Mean-Variance Portfolio Control. • Estimation Uncertainty for Financial Decision Making. • State and Feature Extraction for Risk Management. • Bayesian Portfolio Selection and Control. • Machine Learning Portfolio Selection.
STA5091 - Data-Analysis for High Frequency Trading	M	• EDA and Phenomenology for Trade Decision Making. • Technology Stack Design and Data Workflow Design. • Introduction to Dynamic Stochastic Control. • Hawkes Processes for State Extraction. • Introduction to Random Cluster Models. • State Extraction for Learning and Control. • Introduction to Reinforcement Learning for Trading. • Machine Learning Trading Strategies.
STA5080 - Advanced Regression	M	• Singular Value Decompositions • Mathematics of the Bias-Variance Trade Off • Mathematics of Cross Validation + Code • Regression splines • Recap of Bayesian methods: MCMC, Gibbs Sampling + Code • MARS (Multivariate Adaptive Regression Splines) • Shrinkage methods: Ridge regression, Effective Degrees of Freedom, Bayesian Interpretation of Shrinkage Methods. • LASSO Regression: Generalisations of the LASSO using different penalty structures, Estimation under LASSO. • Principal Components Regression • Partial Least Squares • Kernel Regression Methods

Table 1: Relevant topics in 'Machine Learning' offered in the Department of Statistics. Topics covered in the present course highlighted in green.

5 Selected R Code Examples

Fit a Pruned Tree to the Car Seats Data

```
rm(list  =  ls())

# Load the relevant libraries:
library(rpart)
library(rpart.plot)
# The data is from:
library(ISLR)

# Tabulate the data:
data(Carseats)
head(Carseats, 5)

# Fit an unconstrained regression tree:
res1 = rpart(Sales~., data = Carseats)
res1[[1]]

# Use the prp() function to plot the tree:
prp(res1)

# Conduct complexity analysis:
plotcp(res1)

# Prune the tree and fit the revised model
res2 = prune.rpart(res1, cp = 0.039)

# Plot the revised model and interpret the results:
prp(res2)
res2[[1]]


# Using the tree library with CV:
library(ISLR)
# Prune the tree using cross-validation:
res1  = tree(Sales~., data = Carseats)
cv    = cv.tree(res1,FUN = prune.tree, K = 10)
plot(cv)
res2  = prune.tree(res1, best = 5)
plot(res2)
res2[[1]]
plot(res2)
text(res2)
```

Fit a Neural Network to the Pregnancy Data

```
rm(list = ls())
PREG = read.table('Section 3.2 - Pregnancy Duration Data.txt', h = T)
head(PREG,5)

PREG$Cat = factor(PREG$Cat)


library(h2o)
h2o.init() # Initialize h2o

# Create a data frame and cast it as an h2o object:
frm = PREG[,-c(1,3:5)]
train = as.h2o(frm)

# Fit a (5)-network to the data
```

```
model <- h2o.deeplearning(
 x = 2:7,
 y = 1,
 training_frame = train,
 distribution = "multinomial",
 hidden = c(5),
 activation='Tanh',
 l2 = 0.01,
 epochs = 250000,
 loss = 'CrossEntropy',
 stopping_tolerance = 0.01,
 mini_batch_size = 102,
 rate = 0.05)


# Plot yhe response function:
dummy = seq(min(PREG$Nutrition),max(PREG$Nutrition), length = 100)
XNEW = data.frame(Nutrition = dummy, Age21.30 = 1,Age30 = 0,Alc = 1, Smoke = 1)


pred = h2o.predict(model, newdata = as.h2o(XNEW)) # Predict for this particular set of
    predictors.
pred = as.data.frame(pred)
 plot(pred$p1~dummy, xlab ='Nutrition', ylab = 'Probability', ylim = c(0,1), type = 'l')
 lines(pred$p2~dummy, lty =2)
 lines(pred$p3~dummy, lty =3)
 legend('left', lty = 1:3, legend = paste0('Duration: ',c('<36','36-37','>39')), bty = 'n')
savePlot(paste0('../Figures/Examples8A.pdf'), type = 'pdf')

h2o.shutdown()
```

References

Yaser S Abu-Mostafa, Malik Magdon-Ismail, and Hsuan-Tien Lin. *Learning From Data*, volume 4. AMLBook New York, NY, USA:, 2012.

Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The elements of statistical learning*, volume 1. Springer series in statistics New York, NY, USA:, 2001.

Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An introduction to statistical learning*, volume 112. Springer, 2013.

John Neter, Michael H Kutner, Christopher J Nachtsheim, and William Wasserman. *Applied linear statistical models*, volume 4. Irwin Chicago, 1996.

Michael A Nielsen. *Neural networks and deep learning*. Determination Press, 2015.