Computational Statistics - Summary

Lorenz Walthert

Contents

1 Introduction **5**

2 Multiple Linear Regression **7**

3 Nonparametric Density Estimation **9**

4 Nonparametric Regression **11**

5 Cross Validation **13**
5.1 Motivation and Core Idea . . . 13
5.2 Loss Function . . . 14
5.3 Implementation . . . 15
5.4 K-fold Cross-Validation . . . 15
5.5 Properties of the different schemes . . . 16
5.6 Shortcuts for (some) linear fitting operators . . . 16
5.7 Examples . . . 17

6 Bootstrap **23**
6.1 Motivation . . . 23
6.2 The Bootstrap Distribution . . . 25
6.3 Bootstrap Consistency . . . 25
6.4 Boostrap Confidence Intervals . . . 27
6.5 Boostrap Estimator of the Generalization Error . . . 28
6.6 Out-of-Boostrap sample for estimating the GE . . . 28
6.7 Double Boostrap Confidence Intervals . . . 28
6.8 Three Versions of Boostrap . . . 29
6.9 Conclusion . . . 30

7 Classification **31**
7.1 Indirect Classification - The Bayes Classifier . . . 31
7.2 Direct Classification - The Discriminant View . . . 31
7.3 Indirect Classification - The View of Logistic Regression . . . 32
7.4 Discriminant Analysis or Logistic Regression? . . . 33
7.5 Multiclass case (J > 2) . . . 33

8 Flexible regression and classification methods **35**
8.1 Additive Models . . . 35
8.2 MARS . . . 37
8.3 Example . . . 38
8.4 Neural Networks . . . 39
8.5 Projection Pursuit Regression . . . 42

9 Bagging and Boosting **45**

10 Introduction **47**

Chapter 1

Introduction

This is a summary of the class computational statistics at ETH Zurich.

Chapter 2

Multiple Linear Regression

Chapter 3

Nonparametric Density Estimation

Chapter 4

Nonparametric Regression

Chapter 5

Cross Validation

5.1 Motivation and Core Idea

Cross-validation is a tool for estimating the performance of an algorithm on new data points, the so-called the generalization error. An estimate of the generalization allows us to do two important things:

- Tuning the parameters of a statistical technique.
- Comparing statistical techniques with regard to their accuracy.

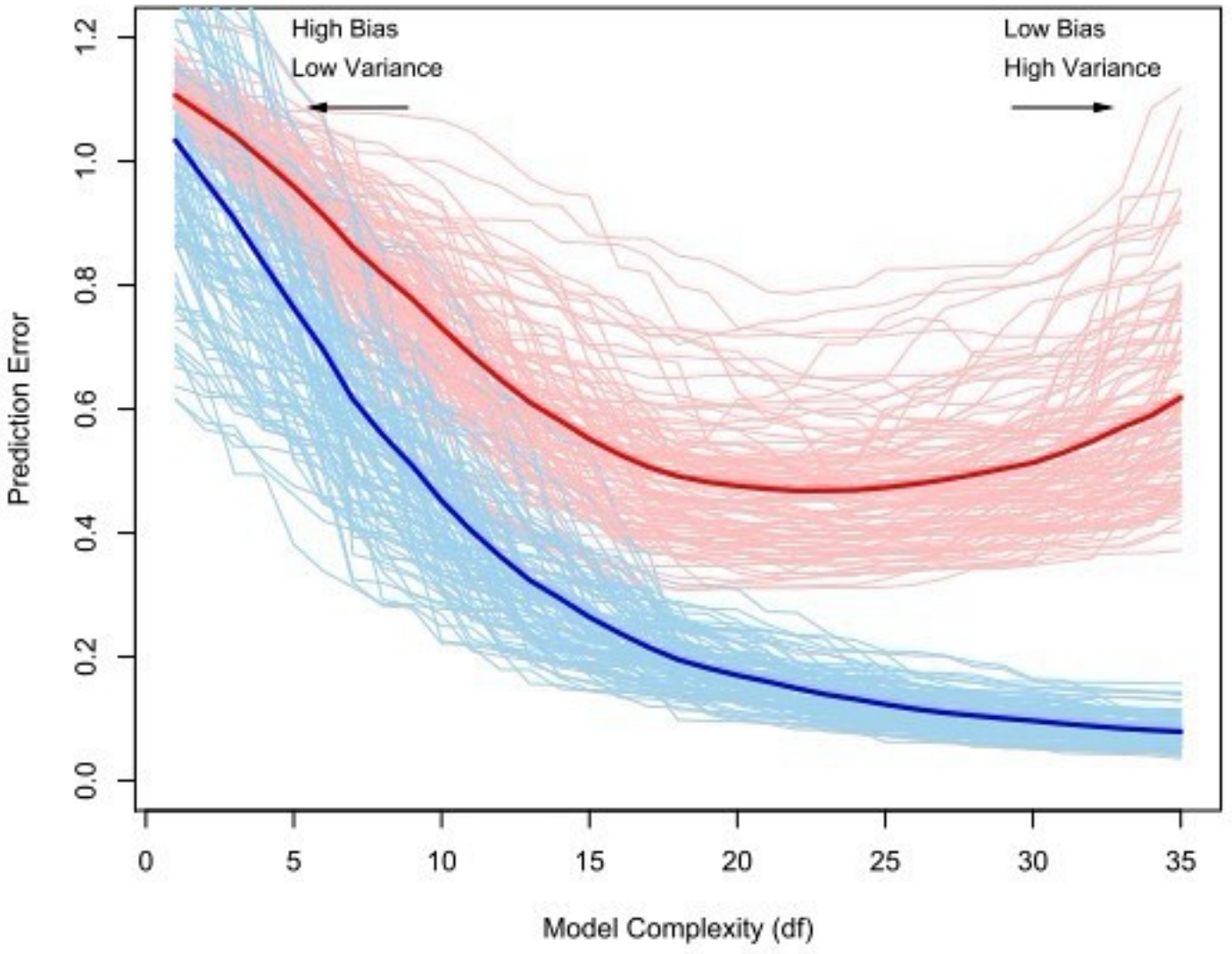
If we use the **training data** to evaluate the performance of an algorithm, this estimate will be over-optimistic because an estimator is usually obtained by minimizing some sort of error in the training data. Therefore, we use a separate data pool, called the **test data** to evaluate the performance out of sample. Consider the regression function estimate $\hat{m}$ based on a sample $(X_1, ..., X_n.)$. By increasing the number of parameters in the model and by allowing for interactions between them, we can make the regression model fitting arbitrarily well to the data. However, such an extremely complex model will not perform as well with new data, that is, will not generalize well to other situations, since we essentially modeled also a lot of noise. We use the following notation:

$$l^{-1} \sum_{i=1}^{l} \rho(Y_{new,i}, \hat{g}(X_{new,i}))$$

Where ρ is a loss function to be evaluated on the new data points $(Y_{new,1}, ..., Y_{new,l})$ and the prediction made for $(X_{new,1}, ..., X_{new,l})$ with the function $\hat{g}$, which was estimated from the training data $(X_1, ..., X_n)$. When l gets large, this approximates the **test error**

$$\mathbb{E}_{(X_{new}, Y_{new})}[\rho(Y_{new}, \hat{m}(X_{new})]$$

which is still a function of the training data (since it is conditional on the training data). Note that the *test error* is not the same as the **generalization error**. The latter is an expectation over both the training and the test data. The typical relationship between the test error and the training error is depicted in the figure below.



5.2 Loss Function

Depending on the application, one can imagine different loss-functions. For example the squared deviance from the *true* value is often used, i.e.

$$n^{-1}\sum_{i=1}^{n}\rho(Y_i,\hat{m}(X_i)) = n^{-1}\sum_{i=1}^{n}(Y_i-\hat{m}(X_i))^2$$

Hence, larger deviance is penalized over-proportionally. For classification, one often uses the zero-one error, i.e.

$$n^{-1}\sum_{i=1}^{n}1_{\hat{m}(X_i)=Y_i}$$

. However, it might also be appropriate to use asymmetric loss functions if false negatives are worse than false positives (i.e. for cancer tests).

5.3 Implementation

There are different ways to do cross validation while adhering to the principles introduced above.

5.3.1 Leave-one-out

- Use all but one data point to construct a model and predict on the remaining data point.
- Do that n times until all n points were used for prediction once.
- Compute the test error as an average over all n errors measured, i.e

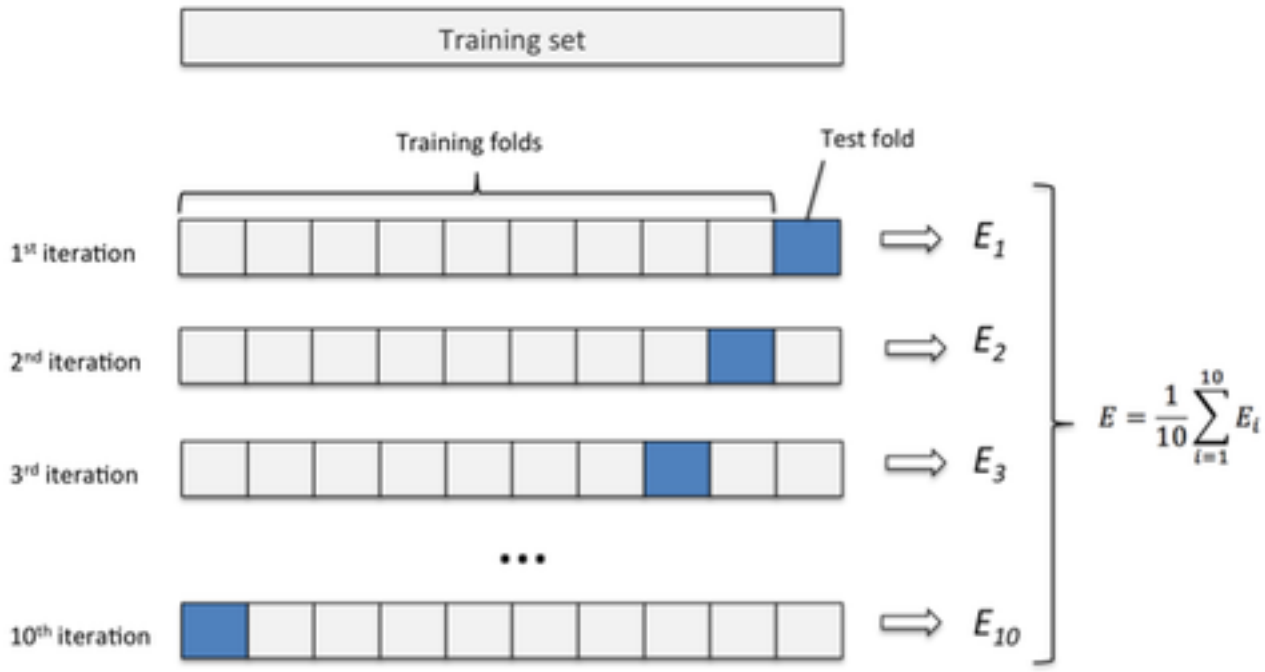
$$n^{-1} \sum_{i=1}^{n} \rho(Y_i, \hat{m}_{n-1}^{-i}(X_i))$$

And use that as an approximation of the *generalization error*.

5.4 K-fold Cross-Validation

This method is best explained with a picture.

```
knitr::include_graphics("figures/k_fold_cv.png")
```



Here, one splits the data set into k equally sized fold. Then, we use all $k-1$ folds to build a model and the remaining fold to evaluate the model. Then, we average the k estimates of the generalization error. Or in mathematical notation:

$$K^{-1} \sum_{k=1}^{K} |B_k|^{-1} \sum_{i \in B_k} \rho(Y_i, \hat{m}_{n-|B_k|}^{-B_k}(X_i))$$

Note that leave-one out cv is the same as k-fold cross validation with $= n$.

5.4.1 Random Division into test and training data set

The problem of K-fold cross-validation is that it depends on **one realization** of the split into k folds. Instead, we can generalize leave-one-out to leave-d-out. That means, we remove d observations from our initial data, apply our estimation procedure and evaluate on the d observations.

$$\hat{\theta}_{n-k}^{-C_k} \quad \text{for all possible subsets } C_k, \quad k = 1, ..., \binom{n}{d}$$

The generalization error can be estimated with

$$\binom{n}{d}^{-1} \sum_{k=1}^{\binom{n}{d}} d^{-1} \sum_{i \in C_k} \rho(Y_i, \hat{m}_{n-d}^{-C_k}(X_i))$$

For $d > 3$, the computational burden becomes immense. For that reason, instead of considering all $\binom{n}{d}$ sets, we can uniformly draw B sets $(C_1^*, ...C_B^*)$ from $C_1, ..., C_{\binom{n}{d}}$ *without replacement*. For $B = \binom{n}{d}$, we obviously get the full leave-d-out solution. The **computational cost** for computing such an approximation to the leave-d-out is linear in B (since evaluating is almost for free). For leave-one-out, the cost is linear in n in the same way. Hence, the stochastic approximation for leave-d-out can be even smaller than for leave-one-out if $B < n$.

5.5 Properties of the different schemes

- **leave-one-out** is an asymptotically **unbiased** estimator for the generalization error and the true prediction. However, we use a sample size $n - 1$ instead of n, which causes a slight bias (meaning we have less data as we do in a real world scenario, which most likely makes the CV score a tiny little bit worse than it should be). Because the training sets are very similar to each other the leave-one-out scheme has a **large variance**.
- **leave-d-out** has a **higher bias** than leave-one-out because the sample size is even smaller than $n - 1$ (for $d > 1$). However, since we aggregate over more ($\binom{n}{d}$ instead of n) cv scores, which can be shown to decrease the variance of the final cv estimator.
- **k-fold** cv has a **higher bias** than both leave one out.

5.6 Shortcuts for (some) linear fitting operators

Leave-one-out cv score for some linear fitting procedures such as least squares or smoothing spline can be computed via a shortcut when our loss function is $\rho(y, \hat{y}) = |y - \hat{y}|^2$. In particular, we can compute the estimator for such a linear fitting procedure **once**, compute the linear fitting operator S, which satisfies $\mathbf{Y} = \mathbf{SY}$ and plug it in this formula:

$$n^{-1} \sum_{i=1}^{n} \left(\frac{Y_i - \hat{m}(X_i)}{1 - S_{ii}} \right)^2$$

Computing $\mathbf{S}$ requires $O(n)$ operations (see exercises).

Historically, it has been computationally easier to compute the trace of $\mathbf{S}$ so there is also a quantity called generalized cross validation (which is a misleading terminology), which coincides with the formula above in certain cases.

5.7 Examples

5.7.1 Practical CV in R

Key concepts to do CV are

- Do not split the data, split the indices of the data and work with them if ever possible and subset the data. `sample()` is your friend.
- use `purrr::map()` and friends to "loop" over data.
- Always work with lists, never work with data frames of indices. The reason is that data frames have structural constrains (all columns must have same number of elements) that are not natural in some situations. For example, out-of-bootstrap cv *does* have the same number of observations in the training set, but not in the test set.
- In conjunction with `sample()`, you can use `purrr::rerun` or `replicate` to create lists of indices.
- use helper function to solve "the small problems in the big problem".

Let's first declare our functions.

```
library("purrr")
data(ozone, package = "gss")

#' Estimate the generalization error of a ridge regression
#' @param test Test indices.
#' @param train Train indices.
#' @param .data The data.
#' @param lamda The lamda parameter for the ridge regression.
ge_ridge <- function(test, train, .data, lambda) {
  fit <- MASS::lm.ridge(upo3~.,
                  lambda = lambda,
                  data = .data[train,])
  pred <- as.matrix(cbind(1, .data[test, -1])) %*% coef(fit)
  mean((pred - .data[test,]$upo3)^2)
}


##  ....................................................................
##  functions to return list with indices                          ####


get_boostrap_mat <- function(B, n) {
  rerun(B, sample(n, n, replace = TRUE))

}

get_all_mat <- function(B, n) {
  rerun(B, 1:n)
}

get_complement <- function(mat, n){
  map(mat, ~setdiff(1:n, .x))
}

get_k_fold <- function(k, n) {
  step <- trunc(n/k)
  current <- list()
```

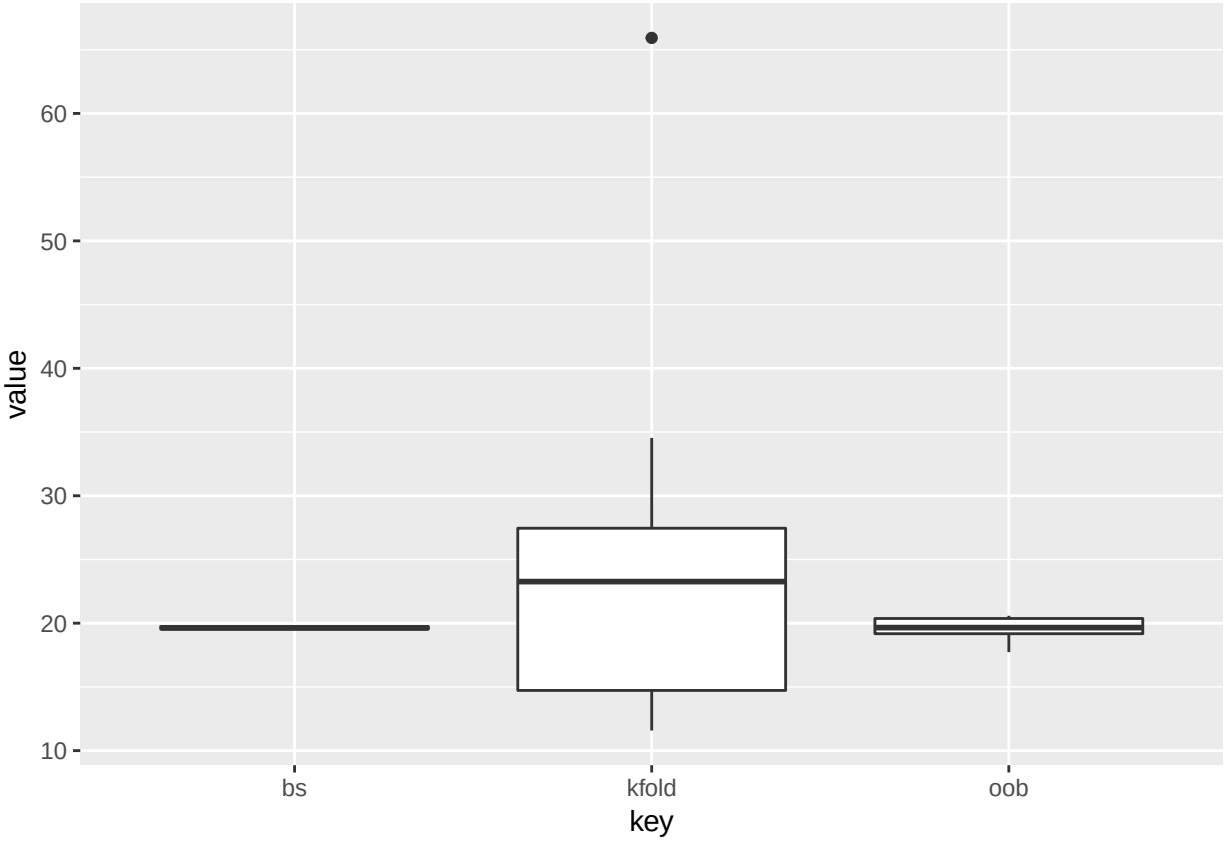
```
  for (i in (0:(k-1) * step + 1))  {
    current <- append(current, list(
      (i:(i+step-1))
    )
    )
  }
  current
}
```

Now, let us apply the functions for three cv schemes to estimate the generalization error.

```
##  ........................................................................
##  boostrap                                                          ####
# use boostrap sample to train, use all to test
n <- nrow(ozone)
train <- get_boostrap_mat(10, n)
test <- get_all_mat(10, n)
bs <- map2_dbl(test, train, ge_ridge, .data = ozone, lambda = 5)



##  ........................................................................
##  10-fold                                                           ####
test <- get_k_fold(10, n)
train <- map(test, ~setdiff(1:n, .x))
kfold <- map2_dbl(test, train, ge_ridge, .data = ozone, lambda = 5)

##  ........................................................................
##  out-of-boostrap                                                   ####
train <- get_boostrap_mat(10, n)
test <- map(test, ~setdiff(1:n, .x))
oob <- map2_dbl(test, train, ge_ridge, .data = ozone, lambda = 5)
```

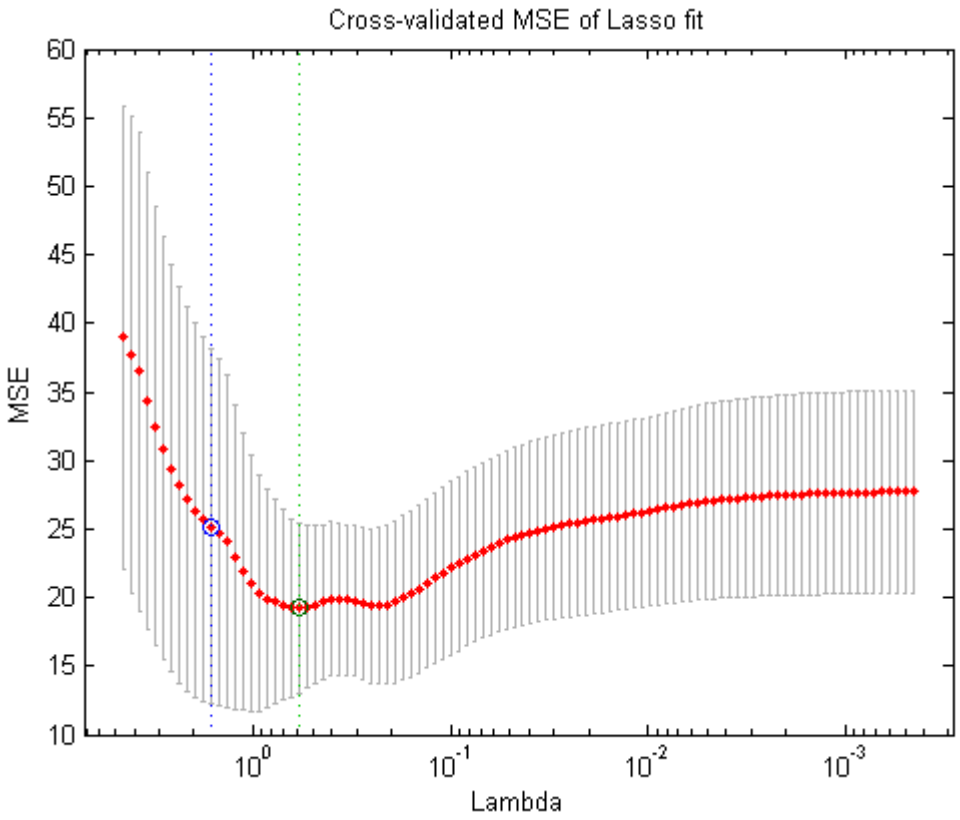
The results are as follows:

```
out <- cbind(bs, kfold, oob) %>%
  as_data_frame() %>%
  gather(key, value)

ggplot(out, aes(y = value, x = key)) +
  geom_boxplot()
```



5.7.2 Parameter Tuning

We want to use the scheme k-fold cross validation for parameter tuning with a lasso. We first calculate the test set error for *one* value of lamda (as we did above). Then, change the value of lamda and recompute the model and then test set error, so that the test set error becomes a function of lamda, as depicted below.

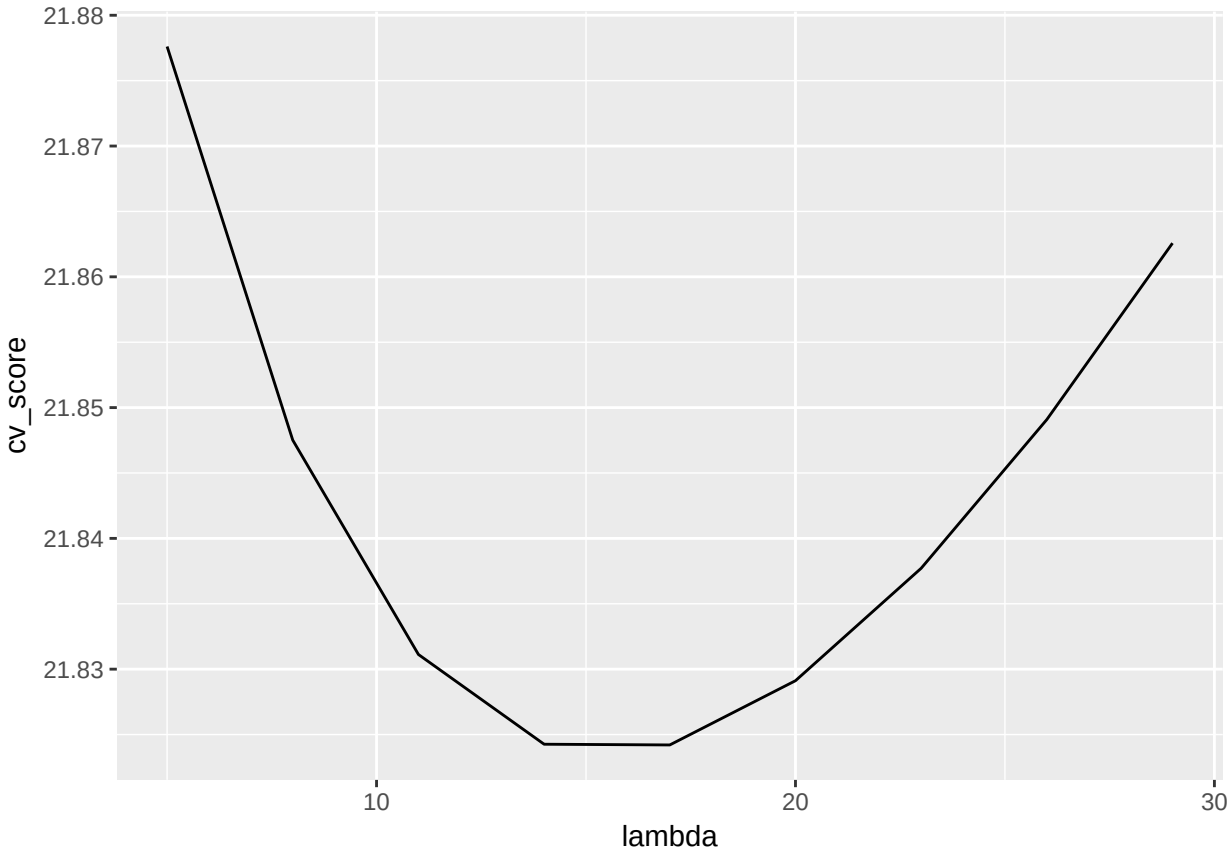


Then pick an optimal lamda, e.g. the one with the lowest test error (a bit arbitrary) or one according to some other rule (e.g. pick the least complex model that is within one standard error of the best model).

```
#' Given lambda, compute the test set error with k folds
find_lambda_kfold_one <- function(lambda, k, n, .data, ...) {
  x_test <- get_k_fold(k, n)
  x_train  <- get_complement(x_test, n)
  map2_dbl(x_test, x_train, ge_ridge, lambda = lambda, .data = .data, ...) %>%
    mean()
}

#' Given a sequence of lambdas, return the corresponding test set errors
find_lambda_kfold <- function(seq, k, .data) {
  cv <- map_dbl(seq, find_lambda_kfold_one,
                k = k, n = nrow(.data), .data = .data)
  results <- data_frame(lambda = seq, cv_score = cv)
  results
}
```

We are almost done. Let us now compute the test set error that we use as an approximation of the generalization error and plot it against different values of lamda.

```
find_lambda_kfold(seq = seq(5, 30, by = 3), 100, ozone) %>%
  ggplot(aes(x = lambda, y = cv_score)) +
  geom_line()
```



That looks reasonable. We could improve on that by also showing the distribution of the test set error at various lambadas. This could by done by altering `find_lambda_kfold_one()` to not return the mean, but also the upper and lower 95% confidence interval.

Chapter 6

Bootstrap

- Bootstrap can be summarized as "simulating from an estimated model"
- It is used for inference (confidence intervals / hypothesis testing)
- It can also be used for estimating the predictive power of a model (similarly to cross validation) via out-of-bootstrap generalization error

6.1 Motivation

Consider i.i.d. data.

$$Z_1, ..Z_n \sim \ P \ \ with \ \ Z_i = (X_i, Y_i)$$

And assume a statistical procedure

$$\hat{\theta} = g(Z_1, ..., Z_n)$$

$g(\cdot)$ can be a point estimator for a regression coefficient, a non-parametric curve estimator or a generalization error estimator based on one new observation, e.g.

$$\hat{\theta}_{n+1} = g(Z_1, ..., Z_n, Z_{new}) = (Y_{new} - m_{Z_1,...,Z_n}(X_{new})^2$$

To make inference, we want to know the distribution of $\hat{\theta}$. For some cases, we can derive the distribution analytically if we know the distribution P. The central limit theorem states that the sum of random variables approximates a normal distribution with $n \to \infty$. Therefore, we know that the an estimator for the mean of the random variables follows the normal distribution.

$$\hat{\theta}_n = n^{-1} \sum x_i \sim N(\mu_x, \sigma_x^2/n) \quad n \to \infty$$

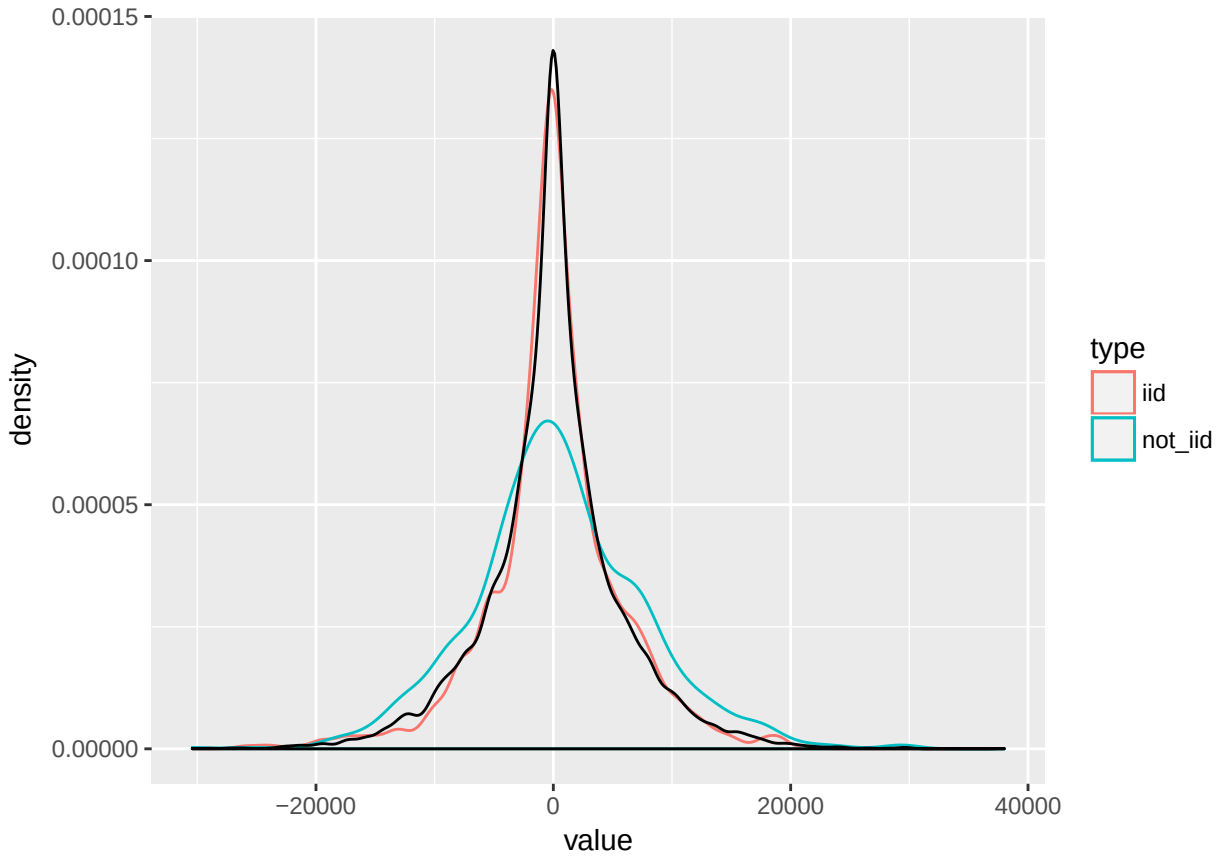
for *any* P. However, if $\hat{\theta}$ does not involve the sum of random variables, and the CLT does not apply, it's not as straightforward to obtain the distribution of $\hat{\theta}$. Also, if P is not the normal distribution, but some other distribution, we can't find the distribution of $\hat{\theta}$ easily. The script mentions the median estimator as an example for which the variance already depends on the density of P. Hence, deriving properties of estimators analytically, even the asymptotic ones only, is a pain. Therefore, if we knew P, we could simply simulate many times and get the distribution of $\hat{\theta}$ this way. That is, draw many $({X_i}^*, {Y_i}^*)$ from that distribution and compute $\hat{\theta}$ for each draw.

The problem is that we don't know P. But we have a data sample that was generated from P. Hence, we can instead take the **empirical** distribution $\hat{P}$that places probability mass of $1/n$ on each observation, draw a sample from this distribution (which is simply drawing uniformly from our sample with replacement) and compute our estimate of interest from this sample.

$$\hat{\theta}^* = g({Z_1}^*, ..., {Z_{new}}^*)$$

We can do that many times to get an approximate distribution for $\hat{\theta}$. A crucial assumption is that $\hat{P}$ reassembles P. If our data is not i.i.d, this may not be the case and hence bootstrapping might be misleading. Below, we can see that i.i.d. sampling (red line) reassembles the true distribution (black line) quite well, whereas biased sampling (blue line) obviously does not. We produce a sample that places higher probability mass on the large (absolute) values.

```
library("tidyverse")
pop <- data_frame(pop = rnorm(10000) * 1:10000)
iid <- sample(pop$pop, 1000) # sample iid

# sample non-iid: sample is biased towards high absolute values
ind <- rbinom(10000, size = 1, prob = seq(0, 1, length.out = 10000))
not_iid <- pop$pop[as.logical(ind)] # get sample
not_iid <- sample(not_iid, 1000) # reduce sample size to 1000

out <- data_frame(iid = iid, not_iid = not_iid) %>%
  gather(type, value, iid, not_iid)
ggplot(out, aes(x = value, color = type)) +
  geom_density() +
  geom_density(aes(x = pop, color = NULL), data = pop)
```



We can summarize the bootstrap procedure as follows.

- draw a bootstrap sample ${Z_1}^*, ..., {Z_n}^*$
- compute your estimator $\hat{\theta}^*$ based on that sample.
- repeat the first two steps B times to get bootstrap estimators $\hat{\theta}_1^*, ..., \hat{\theta}_B^*$ and therefore an estimate of the distribution of $\hat{\theta}$.

Use the B estimated bootstrap estimators as approximations for the bootstrap expectation, quantiles and so on. $\mathbb{E}[\hat{\theta}_n^*] \approx B^{-1} \sum_{j=1}^{n} \hat{\theta}_n^{*j}$

6.2 The Bootstrap Distribution

With P^*, we denote the bootstrap distribution, which is the conditional probability distribution introduced by sampling i.i.d. from the empirical distribution $\hat{P}$. Hence, P^* of $\hat{\theta}^*$ is the distribution that arises from sampling i.i.d. from $\hat{P}$ and applying the transformation $g(\cdot)$ to the data. Conditioning on the data allows us to treat $\hat{P}$ as fixed.

6.3 Bootstrap Consistency

The bootstrap is is called consistent if

$$\mathbb{P}[a_n(\hat{\theta} - \theta) \leq x] - \mathbb{P}[a_n(\hat{\theta}^* - \hat{\theta}) \leq x] \to 0 \ \ (n \to \infty)$$

Consistency of the bootstrap typically holds if the limiting distribution is normal and the samples $Z_1, .., Z_n$ are i.i.d. Consistency of the bootstrap implies consistent variance and bias estimation:

$$\frac{Var^*(\hat{\theta}^*)}{Var(\hat{\theta})} \to 1$$

$$\frac{\mathbb{E}^*(\hat{\theta}^*) - \hat{\theta}}{\mathbb{E}(\hat{\theta}) - \theta} \to 1$$

You can think of θ as the real parameter and $\hat{\theta}$ as the estimate based on a sample. Similarly, in the bootstrap world, $\hat{\theta}$ is the *real* parameter, and $\hat{\theta}_i^*$ as an estimator of the *real* parameter $\hat{\theta}$. The bootstrap world is an analogue of the real world. So in our bootstrap simulation, we know the *true* parameter $\hat{\theta}$. From our simulation, we get many $\hat{\theta}_i^*$ and can find the bootstrap expectation $\mathbb{E}[\hat{\theta}_n^*] \approx B^{-1} \sum_{j=1}^{n} \hat{\theta}_n^{*j}$. The idea is now to generalize from the *boostrap* world to the *real* world, i.e. by saying that the relationship between $\hat{\theta}^*$ and $\hat{\theta}$ is similar to the one between $\hat{\theta}$ and θ.

A simple trick to remember all of this is:

- if there is no hat, add one
- if there is a hat, add a star.

6.4 Boostrap Confidence Intervals

The **pivot** confidence interval argues
approximately the same as the beha

$$
\begin{aligned}
0.95 &\approx \mathbf{P}[\hat{\theta}^*_{(.025)} \leq \hat{\theta} \\
&= \mathbf{P}[\hat{\theta}^*_{(0.025)} - \\
&= \mathbf{P}[\hat{\theta} - \hat{\theta}^*_{(0.025} \\
&\approx \mathbf{P}[\hat{\theta} - \hat{\theta}^*_{(0.025} \\
&= \mathbf{P}[2\hat{\theta} - \hat{\theta}^*_{0.025}
\end{aligned}
$$

So

$$L = 2\hat{\theta} - \hat{\theta}^*_{(0.975)}$$

Note that there confidence intervals are not simply taking the quantiles of the bootstrap distribution. The trick is really to make use of the analogy between the *real* world and the *boostrap* world. So when we see our bootstrap expectation $\mathbb{E}[\hat{\theta}_n^*]$ is way higher than $\hat{\theta}$, then we also should believe that our $\hat{\theta}$ is higher than θ. The above procedure accounts for that.

6.5 Boostrap Estimator of the Generalization Error

We can also use the bootstrap to estimate the generalization error.

$$\mathbb{E}[\rho(Y_{new}, m^*(X_{new}))]$$

- We draw a sample $({Z_1}^*, ..., {Z_n}^*, Z_{new})$ from $\hat{P}$
- We compute the bootstrapped estimator $m(\cdot)^*$ based on the sample
- We estimate $\mathbb{E}[\rho(Y_{new}, m^*(X_{new})^*)]$, which is with respect to both training and test data.

We can rewrite the generalization error as follows:

$$\mathbb{E}[\rho(Y_{new}, m^*({X_{new}}^*))] = \mathbb{E}_{train}[E_{test}[\rho({Y_{new}}^*, m^*({X_{new}}^*))|train]]$$

Conditioning on the training data in the inner expectation, $m(\cdot)$ is non-random / fixed. The only random component is ${Y_{new}}^*$. Since we draw from the empirical distribution and place a probability mass of $1/n$ on every data point. we can calculate the inner (discrete) expectation easily via $\mathbb{E}(X) = \sum_{j=1}^{n} p_j * x_j = n^{-1} \sum_{j=1}^{n} x_j$. The expectation becomes

$$\mathbb{E}_{train}[n^{-1} \sum \rho(Y_i, m^*(X_i))] = n^{-1} \sum \mathbb{E}[\rho(Y_i, m^*(X_i))]$$

We can see that there is no need to draw Z_{new} from the data. The final algorithm looks as follows:

- Draw $({Z_1}^*, ..., {Z_n}^*)$
- compute bootstrap estimator $\hat{\theta}^*$
- Evaluate this estimator on all data points and average over them, i.e $err^* = n^{-1} \sum \rho(Y_i, m^*(X_i))$
- Repeat steps above B times and average all error estimates to get the bootstrap GE estimate, i.e. $GE^* = B^{-1} \sum {err_i}^*$

6.6 Out-of-Boostrap sample for estimating the GE

One can criticize the GE estimate above because some samples are used in the **test as well as in the training set**. This leads to **over-optimistic estimations** and can be avoided by using the out-of-bootstrap approach. With this technique, we first generate a bootstrap sample to compute our estimator and then use the remaining observations not used in the bootstrap sample to evaluate the estimator. We do that B times and the size of the test set may vary. You can see this as some kind of cross-validation with about 30% of the data used as the test set. The difference is that some observations were used multiple times in the training data, yielding a **training set always of size n** (instead of - for example $n * 0.9$ for 10-fold-CV).

6.7 Double Boostrap Confidence Intervals

Confidence intervals are almost never exact, meaning that

$$\mathbb{P}[\theta \in I^{**}(1-\alpha)] = 1 - \alpha + \Delta$$

Where $I^{**}(1-\alpha)$ is a α-confidence interval. However, by changing the *nominal* coverage of the confidence interval, it is possible to make the actual coverage equal to an arbitrary value, i.e

$$\mathbb{P}[\theta \in I^{**}(1-\alpha')] = 1-\alpha$$

The problem is that α' is unknown. But another level of bootstrap can be used to **estimate** α, denoted by $\hat{\alpha}$, which typically achieves

$$\mathbb{P}[\theta \in I^{**}(1-\hat{\alpha}')] = 1-\alpha+\Delta'$$

with $\Delta' < \Delta$

To implement a double bootstrap confidence interval, proceed as follows:

1. Draw a bootstrap sample $(Z_1{}^*, ..., Z_n{}^*)$.
   a. From this sample, draw B second-level bootstrap samples and compute the estimator of interest and *one* confidence interval $I^{**}(1-\alpha)$ based on B second-level bootstrap samples.
   b. evaluate whether $\hat{\theta}$ lays within the bootstrap confidence interval from a. $cover^*(1-\alpha) = 1_{[\hat{\theta} \in I^{**}(1-\alpha)]}$
2. Repeat the above M times to get $cover^{*1}, ..., cover^{*M}$ and hence approximate $\mathbb{P}[\theta \in I^{**}(1-\alpha)]$ with

$$p^*(\alpha) = M^{-1} \sum_{m=1}^{M} cover^{*m}$$

3. Vary α in all of the steps above to find α' so that $p^*(\alpha') = 1-\alpha$

Question here (see Google docs)

6.8 Three Versions of Boostrap

- So far, we discussed the **fully non-parametric** bootstrap, which is simulating from the empirical distribution.
- On the other extreme of the scale, there is the **parametric bootstrap**.
- The middle way is the model-based bootstrap

6.8.1 Non-parametric Regression

We draw a bootstrap sample $(Z_1{}^*, ..., Z_n{}^*) \sim \hat{P}$, i.e. we sample from the *empirical distribution* data with replacement.

6.8.2 Parametric Boostrap

Here, we assume the data are realizations from a *known* distribution P, which is is determined up to some unknown parameter (vector) θ. That means we sample $(Z_1, ..., Z_n) \sim P_{\hat{\theta}}$. For example, take the following regression model $y = X\beta + \epsilon$ where we know the errors are Gaussian.

1. We can estimate our regression model, obtain residuals and compute the mean (which is zero) and the standard deviation of them.
2. To generate a bootstrap sample, we simulate residuals ϵ^* from $N(0, \hat{\mu})$ and
3. Add them to our observed data, i.e. we obtain $(Y_1{}^*, ..., Y_n{}^*)$ from $X\beta + \epsilon^*$. Hence, the final bootstrap sample we use is $(x_1, Y_1{}^*), ..., (x_1, Y_n{}^*))$ where the x_i are just the observed data.
4. We can compute bootstrap estimates $\hat{\beta}^*$, the bootstrap expectation $\mathbb{E}^*[\beta*]$ as well as confidence intervals for the regression coefficients or generalization errors just as shown in detail above. The difference is only *how* the bootstrap sample is obtained.

Similarly, for time series data, we may assume an AR(p) model.

1. Initializing ${X_0}^*, ..., {X_{-p+1}}^*$ with 0.
2. Generate random noise ${\epsilon_1}^*, ..., {\epsilon_{n+m}}^*$ according to $P_{\hat{\theta}}$.
3. Construct our time series ${X_t}^* = \sum_{j=1}^{p} \hat{\theta} {X_{t-j}}^* + {\epsilon_t}^*$ $(X_1, ..., X_{n+m})$
4. Throw away the first M observations that were used as fade-in.
5. Proceed with the B bootstrap samples $({X_1}^*, ..., {X_n}^*)$ as outlined above for the non-parametric bootstrap to obtain coefficient estimates for θ, confidence intervals or estimating the generalization error of the model.

6.8.3 Model-Based Bootstrap

The middle way is the model based bootstrap. As with the parametric bootstrap, we assume to know the model, e.g. $y = m(x) + \epsilon$ (where $m(\cdot)$ might be a non-parametric curve estimator), but we do not make an assumption about the error distribution. Instead of generating ϵ^* from the known distribution with unknown parameters ($\epsilon^* \sim P_{\hat{\theta}}$, as in the parametric bootstrap)), we draw them with replacement from the *empirical* distribution. To sum it up, these are the steps necessary:

1. Estimate $\hat{m(\cdot)}$ from all data.
2. Simulate ${\epsilon_1}^*, ..., {\epsilon_n}^*$ by drawing from $\hat{P}$ with replacement.
3. Obtain $({Y_1}^*, ..., {Y_n}^*)$ from $\hat{m}(x) + \epsilon^*$. As for the parametric bootstrap, the final bootstrap sample we use is $(x_1, {Y_1}^*), ..., (x_n, {Y_n}^*))$ where the x_i are just the observed data.
4. Again, you can use the bootstrap samples as for the other two methods.

6.9 Conclusion

Which version of the bootstrap should I use? The answer is classical. If the parametric model fits the data very well, there is no need to estimate the distribution explicitly. Also, if there is very little data, it might be very difficult to estimate P. On the other hand, the non-parametric bootstrap is less sensitive to model-misspecification and can deal with arbitrary distributions (? is that true?).

Chapter 7

Classification

7.1 Indirect Classification - The Bayes Classifier

In classification, the goal is to assign observations to a group. Similar to regression, where we have $m(x) = E[Y|X = x]$, we want to assign class probabilities to the observations

$$\pi_j(x) = P[Y = j|X = x] \quad (j = 0, 1, ..., J-1)$$

Def: A classifier maps A multidimensional input vector to a class label. Or mathematically: $C : \mathbb{R}^p \rightarrow \{0, ..., J-1\}$ The quality of a classifier is measured via the zero-one test-error.

$$\mathbb{P}[C(X_{new}) \neq Y_{new}]$$

The optimal classifier with respect to the zero-one Error is the Bayes Classifier. It classifies an observation to the group for which the predicted probability was the highest.

$$C_{bayes}(x) = \arg \max_{0<j<J-1} \pi_j(x)$$

Hence, the Bayes Classifier is a point-wise classifier. For the Bayes Classifier, the zero-one test error is known as the *Bayes Risk*.

$$\mathbb{P}[C_{Bayes}(X_{new}) \neq Y_{new}]$$

In practice, $\pi_j(\cdot)$ is unknown (just as the MSE in regression is unknown) and hence, the the Bayes Classifier and Risk is unknown too. However, we can estimate $\pi_j(\cdot)$ from the data and plug it in the Bayes Classifier.

$$\hat{C}(X) = \arg \max_{0<j<J-1} \hat{\pi}_j(x)$$

This is an indirect estimator, since we first estimate the class probabilities $\pi_j(\cdot)$ for each observation x and then assign the class to it for which the probability was the highest. Question how is that more indirect than Discriminant analysis? Don't we use the Bayes classifier in the end?

7.2 Direct Classification - The Discriminant View

7.2.1 LDA

One example for a direct classification is discriminant analysis. Using Bayes Theorem

$$\mathbb{P}[Y = j|X] = \frac{\mathbb{P}[X = x|y = j]}{\mathbb{P}[X = x]} * \mathbb{P}[Y = j]$$

And assuming

$$(X|Y) \sim N_p(\mu_j, \Sigma); \ \sum_{k=0}^{J-1} p_k = 1$$

We can write

$$\mathbb{P}[Y = j|X = x] = \frac{f_{x|Y=j} * p_j}{\sum_{k=0}^{J-1} f_{x|Y=k} * p_k}$$

Note that there is no distributional assumption on Y so far. You can estimate

$$\mu_j = \sum_{i=1}^{n} x_i * 1_{Y_i=j} / 1_{Y_i=j}$$

and

$$\Sigma = \frac{1}{n-j} \sum_{j=0}^{J-1} \sum_{i=1}^{n} (x_i - \mu_j)(x_i - \mu_j)' \ 1_{Y_i=j}$$

Note that the means of the response of the groups are different, but the covariance structure is the same for all of them. We now also need to estimate p_j. A straight-forward way is

$$\hat{p}_j = n^{-1} \sum_{i=1}^{n} 1_{[Y_i=j]} = \frac{n_j}{n}$$

From here, you can easily compute the classifier (as done in the exercise) by maximizing the log-likelihood. Then, you can derive the decision boundary by using $\delta_j - \delta_k = 0$. In a two dimensional predictor space with two classes, the decision boundary is a line. Every combination of the two predictors on one side of the line will result in a prediction of class one, everything on the other side of the line of class two. Note that both the decision function (and hence the decision boundary) are linear in x.

7.2.2 QDA

Quadratic discriminant analysis loosens the assumption of shared covariance matrices, namely each group has their own covariance matrix. This leads to quadratic decisions functions δ and hence to non-linear decision boundaries. QDA is more flexible but for high p, the problem of over-fitting can occur, since the number of variables to estimate is $J * p(p+1)$ variable for the covariance matrix only (instead of $p * (p+1)$ for LDA.

7.3 Indirect Classification - The View of Logistic Regression

There are also ways to come up with indirect assignment of the class label, namely via the Bayes classifier $\hat{C}(X) = \arg\max_{0<j<J-1} \hat{\pi}_j(x)$. The logistic model for $\pi_j(x) = \mathbb{P}(Y = j|X = x)$ is $\log(\frac{\pi_j(x)}{1-\pi_j(x)}) = g(\cdot)$ or $\pi_j(x) = \frac{e^{g(\cdot)}}{1+e^{g(\cdot)}}$ equivalently. That model maps the real value $g(\cdot)$ can take to the interval $(0, 1)$, which gives a natural interpretation of the response as a probability. Note that we want the transformation to be monotone so the mapping is invertible. The response variable $Y_1, ..., Y_n$ are distributed according to a Bernoulli distribution, i.e.. $Y_1, ..., Y_n \sim \text{Bernulli}(\pi(x_i))$. The logistic model belongs to the class of generalized linear models. These models have three characteristics:

- A link function (in our case the logit function $\log(\frac{\pi_j(x)}{1-\pi_j(x)}) = g(x)$).
- A response distribution (i.e. Bernoulli)

- The concrete form of $g(\cdot)$ (in logistic regression most often just a linear model $g(x) = \sum_{j=1}^{p} \beta_j x_j$).

This allows us to write down the likelihood of the logistic model

$$\begin{aligned} L(\beta, (x_1, Y_1), ..., (x_n, Y_n)) &= \prod_{i=1}^{n} \mathbb{P}(Y_i = y_i) \\ &= \prod_{i=1}^{n} \pi(x_i)^{Y_i - 1}(1 - \pi(x_i))^{Y_i - 1} \\ \log(L(\cdot) &= \sum_{i=1}^{n} Y_i \pi(x_i) + (1 - Y_i)(1 - \pi(x_i)) \end{aligned} \tag{7.1}$$

There is no closed-form solution for the above problem, hence we need to rely on gradient descent to find the maximum likelihood solution. You can fit a logistic regression model in R as follows:

```
# fit the model
fit <- glm(response~predictors, data = my_data, family = "binomial")
# predict
prediction <- predict(fit, newdata = my_data, type = "response") > 0.5
# evaluate in-sample
mean(prediction == my_data$response)
```

7.4 Discriminant Analysis or Logistic Regression?

- The logistic regression assumes the log-odds to be *linear* in the predictors, i.e. $\log\left(\frac{\pi}{1-\pi}\right) = \sum_{i=1}^{p} \beta_i x_i$.
- The discriminant analysis assumes $X|Y \sim N(\mu_j, \Sigma$, which leads to linear model in the decision variables. Hence, the methods are **quite similar**.
- It is quite natural to use factors with logistic regression, while for discriminant analysis, it is not very natural (?even not reasonable?).
- Empirically, the two methods yield similar results even under a violation of the normality assumption.

TODO multinomial likelihood (see footnote.)

7.5 Multiclass case (J > 2)

With logistic regression, you can not model multiclass cases directly, but indirectly using one of the following methods:

- Encode a J-case problem as J binary problems (one class against the rest), that is

$$Y_i^{(j)} = \begin{cases} 1 & \text{if} \quad Y_i = j \\ 0 & \text{else.} \end{cases} \tag{7.2}$$

For each case you want to label, you will obtain $J - 1$ probability estimates, i.e. $\pi_j(x) = \frac{\exp\left(\sum \beta_j^{(j)} x_j\right)}{1+\exp\left(\sum \beta_j^{(j)} x_j\right)}$. They don't sum up to one necessarily, but you can normalize to obtain normed probabilities. Then, use the Bayes classifier to choose the class label ($\arg\max_{0<j<J-1} \pi_j$)

- Similarly, you can also model *all against the reference*, $\log(\frac{\pi_1}{\pi_0})$. This might be helpful when we want to compare different group memberships with a reference group. For example in clinical trials, we want to compare how many times more likely someone belongs to group *ill with disease A* than *healthy* (the reference).
- You can also look at pair-wise comparisons. Choose two groups you want to compare, drop all observations that don't belong to one of the two groups and estimate the model. There are $\binom{J}{2}$ possible models with all models potentially having different number of observations.
- In the special case of **ordered groups**, the correct model is often proportional odds model that models

$$\text{logit}(\mathbb{P}(Y < k|X) = \alpha_k + g(\cdot))$$

with $\alpha_1 < \alpha_2 < ... < \alpha_{J-1}$. The log odds are proportional, which becomes obvious if we take e to the power of the above equation. Check this webpage for more information. Note that proportionality in the log odds does *not* mean proportionality in the probabilities, since they are only linked through a non-linear mapping (the logistic transformation).

Chapter 8

Flexible regression and classification methods

The curse of dimensionality makes it very hard to estimate fully nonparametric regression function $\hat{m} = \mathbb{E}[Y|X = x]$ or classification function $\hat{\pi}_j = \mathbb{P}[Y = j|X = x]$. Hence, by making some (reasonable) structural assumptions, we can improve our models significantly. Generally, we consider the mapping $\mathbb{R}^p \to \mathbb{P}$ via the function $g(\cdot)$ for both the regression and the classification problem.

8.1 Additive Models

8.1.1 Structure

One assumption we can make is to assume a particular functional form of $g(\cdot)$, namely an *additive*. That is

$$g_{add} = \mu + \sum_{j=1}^{p} g(x_j)$$

$E[g(x_j)] = 0$ is required to make the model identifiable only. Note that we have not placed any assumption on $g(x_j)$ yet, that is, $g(x_j)$ can be fully non-parametric, but each dimension is mapped separately. In other words every $g(x_j)$ $j = 1, ..., p$ models one input dimension and mapping of input to output is obtained by summing the transformed inputs up. This eliminates the possibility of interaction effects.

8.1.2 Fitting Procedure

Additive models can be estimated with a technique called back-fitting. However, the model can be estimated with any nonparametric method for one-dimensional smoothing. Here is the receipt:

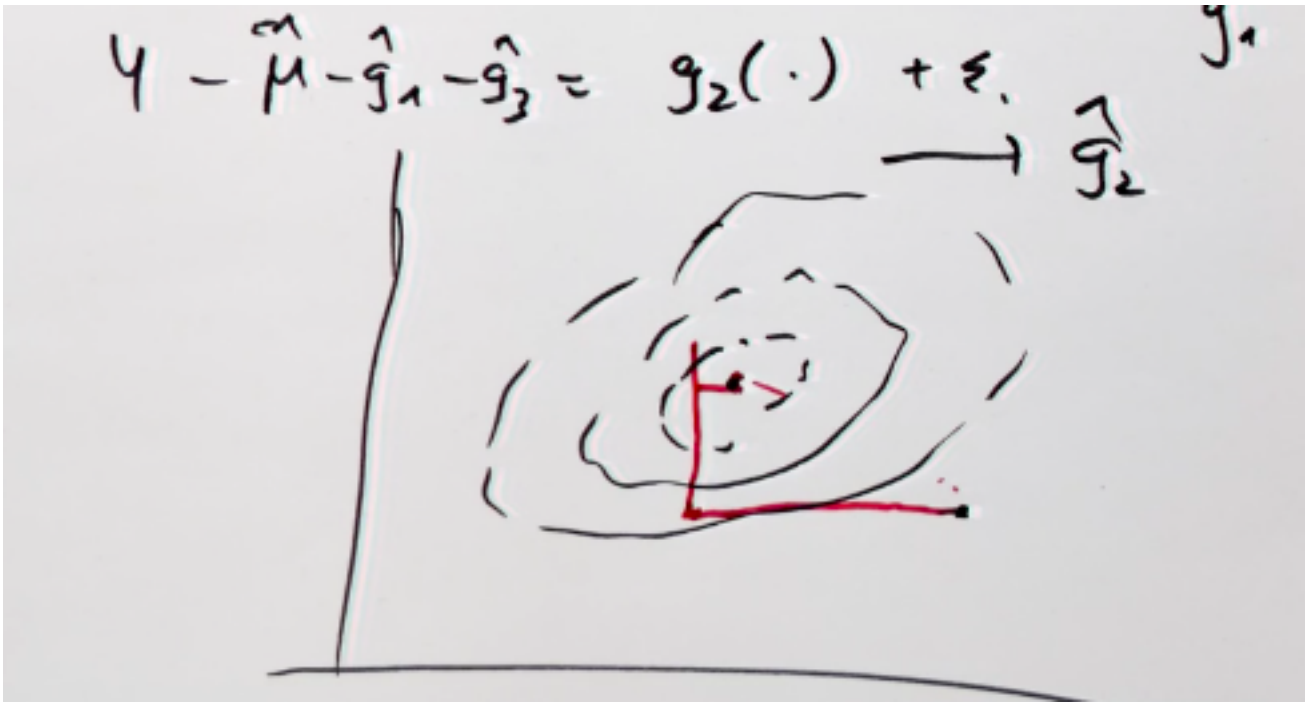
- since we assume an additive model, we need to initialize all p components of it with zero, that is setting $g_j(\cdot) = 0$ $j = 1, ..., p$ plus setting $\mu = n^{-1} \sum Y_i$. * Then we fit one-dimensional smoother repeatedly, that is solving the one-dimensional smoothing problem $Y - \mu - \sum_{j \neq k} \hat{g}_j = \hat{g}_j(x_j)$, or put differently $\hat{g}_j = S_j(Y - \mu 1 - \sum_{j \neq k} g_j)$. This has to be done repeatedly for $j = 1, ..., p, 1, ..., p$etc..
- Stop iterating when functions don't change much anymore, that is, when the following quantity is less than a certain tolerance.

$$\frac{|\hat{g}_{i,new} - \hat{g}_{i,old}|}{|\hat{g}_{i,old}|}$$

- Normalize the functions by subtracting the mean from them:

$$\tilde{g}_j = \hat{g}_j - n^{-1} \sum_{i=1}^{n} \hat{g}_j(x_{ij})$$

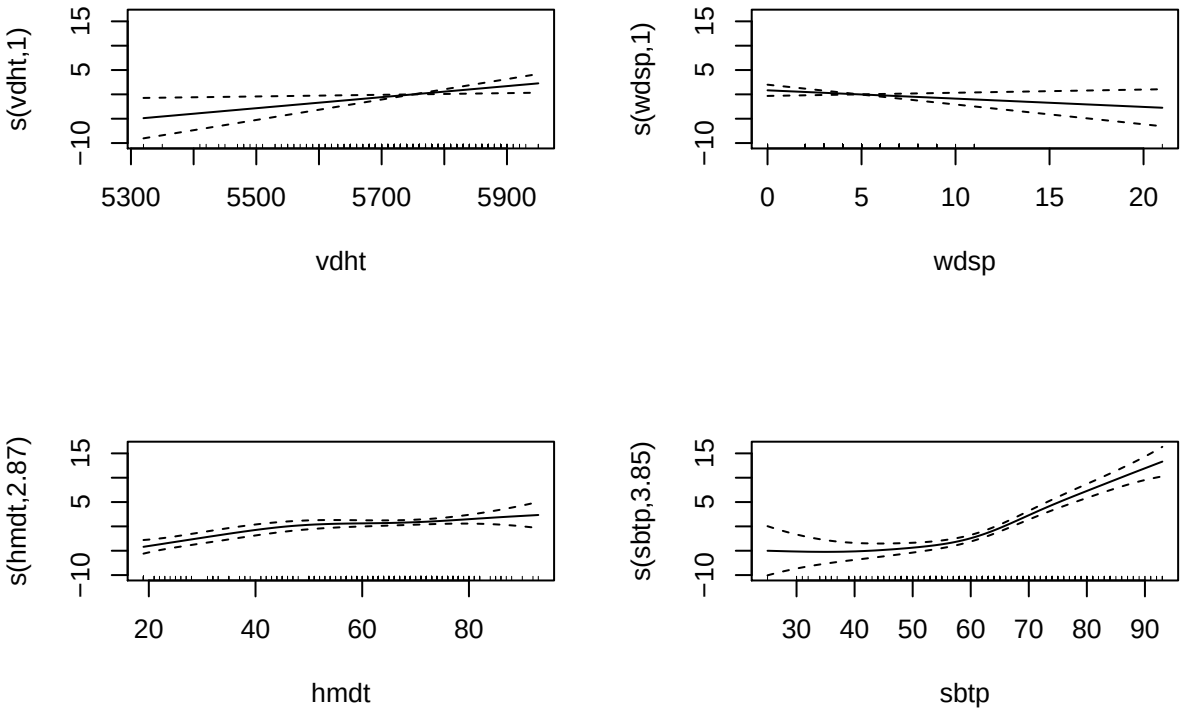
Back-fitting is a **coordinate-wise** optimization method that optimizes one coordinate at the time (one $g(\cdot)$, but can be more than one parameter), which may be slower in convergence than a general gradient descent that optimizes all directions simultaneously but also more robust.



8.1.3 Additive Models in R

You can use the package **mgcv**

```
data("ozone", package = "gss")
fit <- mgcv::gam(upo3 ~ s(vdht) + s(wdsp) + s(hmdt) + s(sbtp),
           data = ozone)

plot(fit, pages = 1)
```



You can see that vdht enters the model almost linearly. That is, with an increase of one unit of vdht, the predicted 03 value increases linearly. sbtp is different. Depending on the value of sbtp, the increase in the predicted value is different. Low sbtp values hardly have an impact on the response, higher values do.

8.2 MARS

MARS stands for multivariate adaptive regression splines and allows pice-wise linear curve fitting. In contrast to GAMs, they allow for interactions between the variables. MARS is very similar to regression trees, but it has a continuous response. It uses reflected pairs of basis functions

$$(x_j - d)_+ = \begin{cases} x_j - d & \text{if} \quad x_j - d > 0 \\ 0 & \text{else.} \end{cases} \tag{8.1}$$

and it counterpart $(d - x_j)_+$. The index j refers to the j-th predictor variable, d is a knot at one of the $x_j s$. The pool of basis functions to consider is called $\mathbf{B}$ and contains all variables with all potential knots, that is

$$\mathcal{B} = \{(x_j - d)_+ \quad (d - x_j)_+ \quad j = \{1, ..., p\} \quad d = \{x_{1j}, ..., x_{nj}\}\}$$

The model then is

$$g(x) = \mu + \sum_{m=1}^{M} \beta_m h_m(x) = \sum_{m=0}^{M} \beta_m h_m(x)$$

The model uses forward selection and backward deletion and optimizes some (generalized) cross-validation criterion. Here is the receipt:

1. initialize $\mathcal{M} = \{h_0(\cdot) = 1\}$ and estimate β_0 as the data average of the response $n^{-1} \sum_{i=1}^{n} Y_i$. 2. for $r = 1, 2, ...$ search for a new pair of function $(h_{2r-1} \;\; h_{2r})$ which are of the form

$$h_{2r-1} = (x_j - d)_+ \times h_l$$

$$h_{2r} = (d - x_j)_+ \times h_l$$

that reduce the residual sum of squares the most with some h_l from $\mathcal{M}$ that some basis functions from $\mathcal{B}$ does *not* contain
x_j. The model becomes

$$\hat{g}(x) = \sum_{m=0}^{2r} \hat{\beta}_m h_m(x)$$

which can be estimated by least squares. Update the set $\mathcal{M}$ to be $\mathcal{M} = \mathcal{M}_{old} \cup \{h_{2r-1}, h_{2r}\}$ 3. Iterate the above step until the set $\mathcal{M}$ becomes *large enough*. 4. Do backward deletion (*pruning*) by removing the *one* function from a reflected pair that increases the residual sum of squares the least. 5. Stop the pruning when some GCV score reaches its minimum.

8.2.1 Details for Dummies

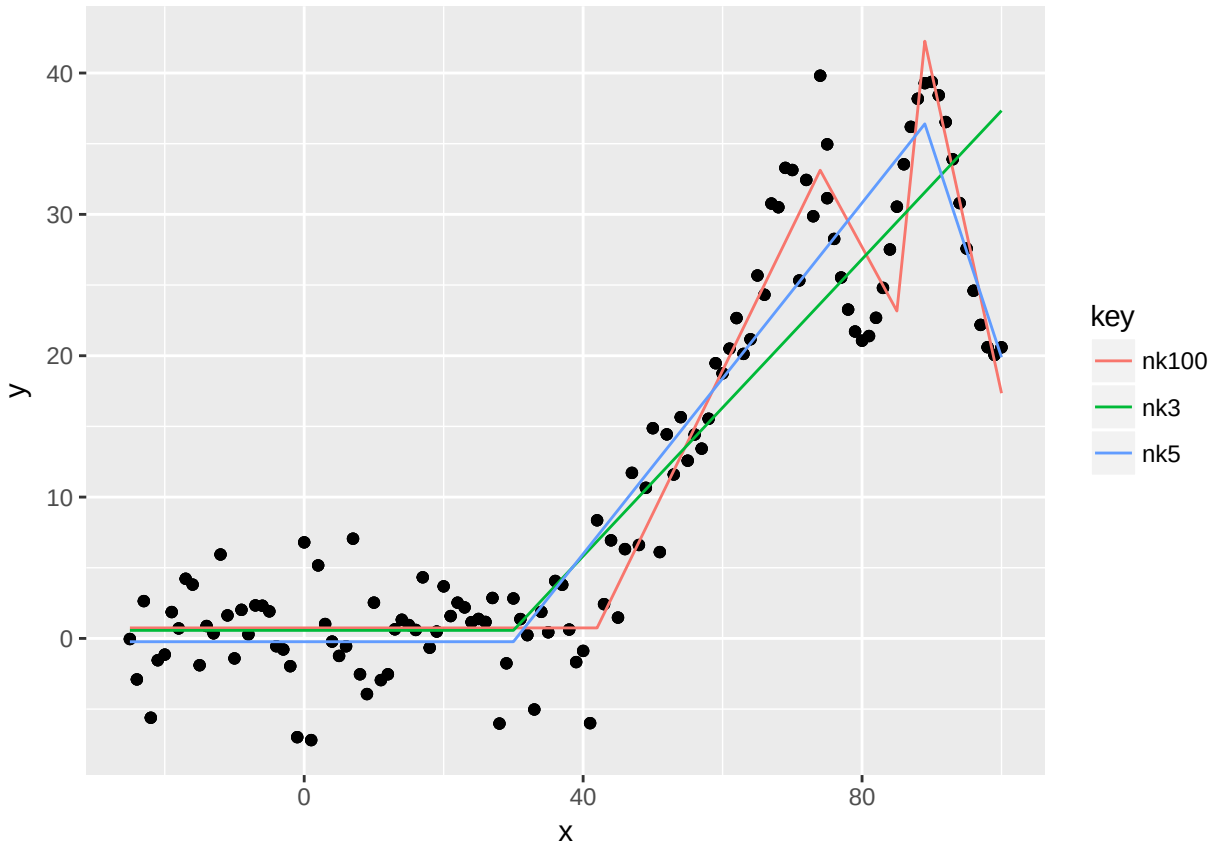
Note that by using reflective pairs $\{(x_j - d)_+ \quad (d - x_j)_+ \}$, we construct a piece-wise linear function with one knot at d, since we sum the functions (which both a have a zero part that does not overlap) up. This hinge function (or better the two parts of individually) β will be multiplied with a respective β, so the slope is adaptive. Also, since each of the functions have its own d, the kink of the function must not occur at $y = 0$.

From the receipt above, we can see that d-way interactions are only possible if a $d-1$-way interaction with a subset of the d-way interaction is already in the model. For interpretability and other reasons, restricting the number of interactions to three or two is beneficial. Restricting the degree of interaction to one (which is actually no interaction) will yield an additive model.

8.3 Example

Let us consider the simple case of a one dimensional predictor space. We add noise to data that is piece-wise linear up to $x = 100$ and then follows a sine-wave. We then fit three mars models with different number of maximal knots.

```
library("earth")
sim <- data_frame( x = -25:75,
                   y = pmax(0, x - 40) + rnorm(100, 0, 3)) %>%
  bind_rows(data_frame( x = 75:100, y = -1*sqrt(x)* sin(x/3) + 30))

fitted <- data_frame(
  nk3  = earth(y~x, data = sim, nk = 3),
  nk5  = earth(y~x, data = sim, nk = 5),
  nk100 = earth(y~x, data = sim, nk = 100)
)


sim2 <- fitted %>%
  map_df(~predict(.x)) %>%
  bind_cols(sim) %>% gather(key, value, -x, -y)


ggplot(sim2) +
  geom_point(aes(x = x, y = y)) +
  geom_line(aes(x = x, y = value, color = key))
```



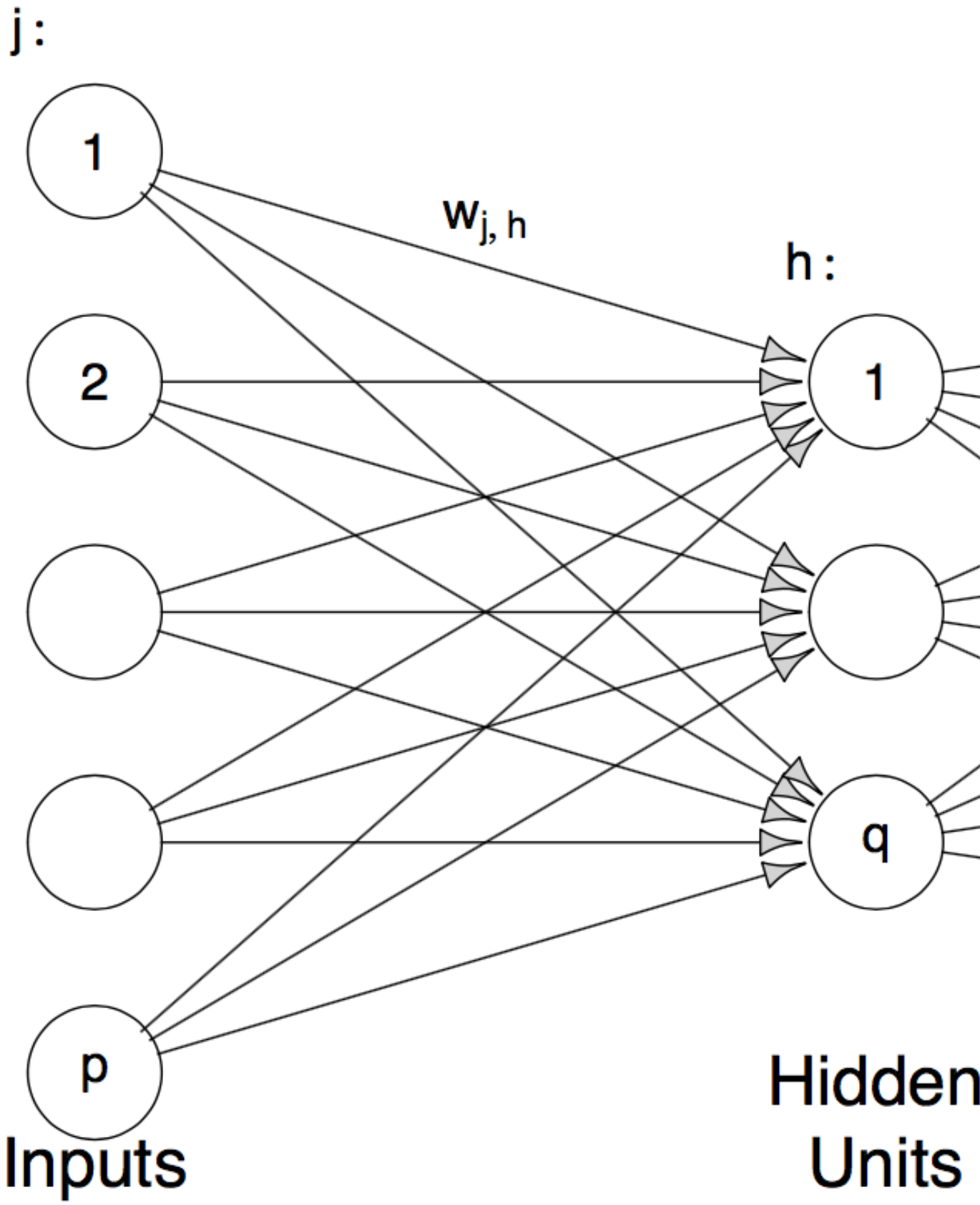

```
summary(fitted$nk3)
```

```
## Call: earth(formula=y~x, data=sim, nk=3)
##
##             coefficients
## (Intercept)    0.5687189
## h(x-30)        0.5252673
##
## Selected 2 of 3 terms, and 1 of 1 predictors
## Termination condition: Reached nk 3
## Importance: x
## Number of terms at each degree of interaction: 1 1 (additive model)
## GCV 31.36253    RSS 3797.088    GRSq 0.8270925    RSq 0.8325381
```

The example shows what we expected. The green line with a maximum of three knots uses just one knot around 30, and only the right part of the reflected pair is used. It cannot distinguish between the linear segment between 40 and 75 and the sine-wave afterwards. By allowing more knots, we can see that the red line fits the data quite well. Note that the default of `degree` is just 1, so we don't have interaction terms in the model. This does not matter for a one-dimensional example anyways.

8.4 Neural Networks

Neural networks are high-dimensional non-linear regression models. The way it works is best illustrated with a picture.


j :
1
2
p
Inputs
$w_{j,h}$
h :
1
q
Hidden
Units

This is a neural network with one hidden layer, p input layers and q output layers. Mathematically speaking, the model is:

$$g_k(x) = f_0\Big(\alpha_k + \sum_{h=1}^{q} w_{ij}\phi(\tilde{\alpha}_h + \sum_{j=1}^{p} w_{jh}x_j)\Big)$$

Where $\phi(x)$ is the sigmoid function $\frac{exp(x)}{1+exp(x)}$, f_0 is the sigmoid function for classification and the identity for regression. In the case of regression $k = 1$ is used, for classification we use $g_0, ..., g_{J-1}$ and then use the Bayes classifier $\hat{\mathcal{C}}(\S) = \arg\max_{0<j<J-1} g_j(x)$ (is that correct?), which is called the softmax in the neural network literature.

8.4.1 Fitting Neural Networks (in R) The `nnet` function from the package with the

same name basically uses gradient descent to maximize the likelihood. It is important to first scale the data so the gradient descent does not get stuck in flat regions of the sigmoid function.

```
set.seed(22)
data("ozone", package = "gss")
unloadNamespace("MASS")
scaled <- ozone %>%
  select(-upo3) %>%
  scale() %>%
  as_data_frame() %>%
  mutate(log_upo3 = log(ozone$upo3))

fit <- nnet::nnet( log_upo3 ~., data = scaled,
                   size = 3, # how many nodes in the *one* hidden layer.
                   decay = 4e-4, # regularization. Multiply weights by 1 - decay after
                   # gradient step.
                   linout = TRUE, # linear output units (refers to f0?).
                   skip = TRUE, # add skip-layer connections between output and input.
                   maxit = 500 )
```

The weights between the nodes are:

```
summary(fit)
```

```
## a 9-3-1 network with 43 weights
## options were - skip-layer connections  linear output units  decay=4e-04
##  b->h1 i1->h1 i2->h1 i3->h1 i4->h1 i5->h1 i6->h1 i7->h1 i8->h1 i9->h1
##  -2.28   1.27  -0.34  -2.57   1.46   0.03   0.10  -1.02  -0.39  -0.33
##  b->h2 i1->h2 i2->h2 i3->h2 i4->h2 i5->h2 i6->h2 i7->h2 i8->h2 i9->h2
## -12.43   5.09   2.04   8.19  -7.66  -7.01   2.40  -0.31   3.59  -1.19
##  b->h3 i1->h3 i2->h3 i3->h3 i4->h3 i5->h3 i6->h3 i7->h3 i8->h3 i9->h3
## -19.77  -6.64   1.49  -4.53  -3.95   2.28   6.05   5.19  10.05  -0.20
##   b->o  h1->o  h2->o  h3->o  i1->o  i2->o  i3->o  i4->o  i5->o  i6->o
##   2.50  -1.81   0.68   0.71   0.11  -0.09  -0.49   0.72   0.01  -0.03
##  i7->o  i8->o  i9->o
##   0.04  -0.29  -0.15
```

The in-sample MSE for the regression case is

```
mean(residuals(fit)^2)
```

```
## [1] 0.1036706
```

8.5 Projection Pursuit Regression

Projection pursuit regression is similar to both neural nets and additive models. It can be seen as an additive model whereas the predictors were first projected into the optimal direction. The model takes the form

$$g_{PPR} = \mu + \sum_{k=1}^{q} f_k(\sum_{j=1}^{p} \alpha_{jk} x_j)$$

With $\sum_{j=1}^{p} \alpha_j = 1$ and $E[f_k(\sum_{j=1}^{p} \alpha_{jk} x_j)] = 0$ for all k.

$\alpha_k x_j$ is the projection of the j-th column in the design matrix onto α_k. The functions f_k only vary along one direction and are hence called ridge functions. The model requires much smaller q than a neural net requires hidden units, at the expense of estimating the ridge functions (which is not necessary for neural nets, as it is already fully specified by the sigmoid function).

8.5.1 Proejction Pursuit Example

In the following, we illustrate how optimal projections of the initial predictor space can allow us to use an additive functional form to deal with interaction terms. Let us consider the following data-generating model

$$Y = X_1 \times X_2 + \epsilon \text{ with } \epsilon \sim N(0,1) \text{ and } X_1, X_2 \sim \text{Unif}(-1,1)$$

Where $X \in \mathbb{R}^2$, i.e. a two-dimensional predictor space with the predictors X_1 and X_2. Using elementary calculus, this can be rewritten as

$$Y = \frac{1}{4}(X_1 + X_2)^2 - \frac{1}{4}(X_1 - X_2)^2$$

Hence, we rewrote a multiplicative model as an additive model. As we are using arbitrary *smooth* functions f_k, we can easily fit the quadratic terms in the equation above, so the problem to solve becomes

$$Y = \mu + f_1(X_1 + X_2) - f_2(X_1 - X_2)$$

Therefore, the remaining question is how can we choose the two vectors α_1 and α_1 such that the result of the projection is $X_1 + X_2$ and $X_1 - X_2$. With the restriction $|\alpha| = 1$, it turns out we can preceded as follows: We project the first predictor onto $(\alpha_{11}, \alpha_{12}) = (0.7, 0.7)$ and the second predictor onto $(\alpha_{11}, \alpha_{12}) = (0.7, -0.7)$. This yields $0.7(X_1 + X_2) - 0.7(X_1 - X_2)$.

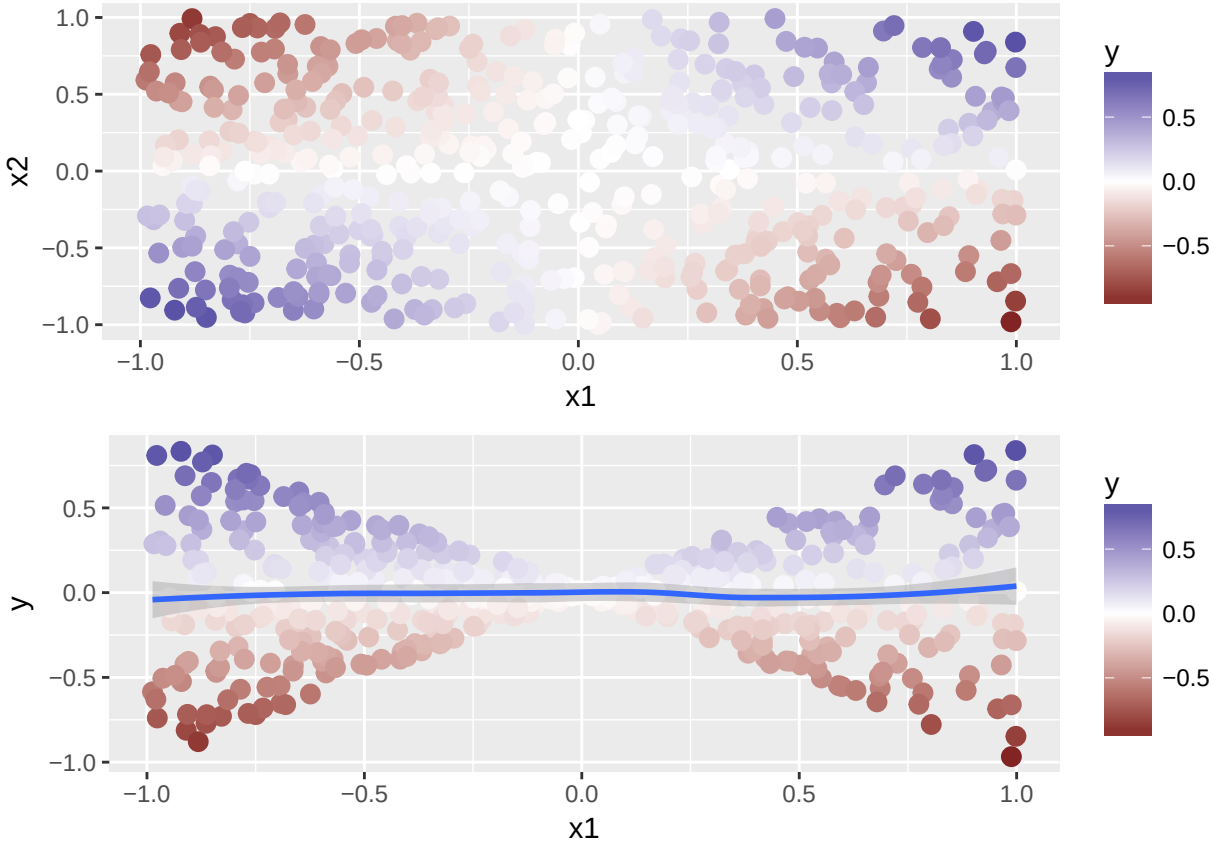
Let's implement that with R

```
data <- data_frame(
  x1 = runif(500, -1, 1),
  x2 = runif(500, -1, 1),
  y = x1*x2 + rnorm(500, 0, 0.005)
)
all <- ggplot(data, aes(x = x1, y = x2)) +
  geom_point(aes(color = y), size = 3) +
  scale_color_gradient2()
```

We can see the obvious pattern, but we can also see that an additive model would not do well on that.

```
x1y <- ggplot(data, aes(x = x1, y = y)) +
  geom_point(aes(color = y), size = 3) +
  geom_smooth() +
  scale_color_gradient2()
```

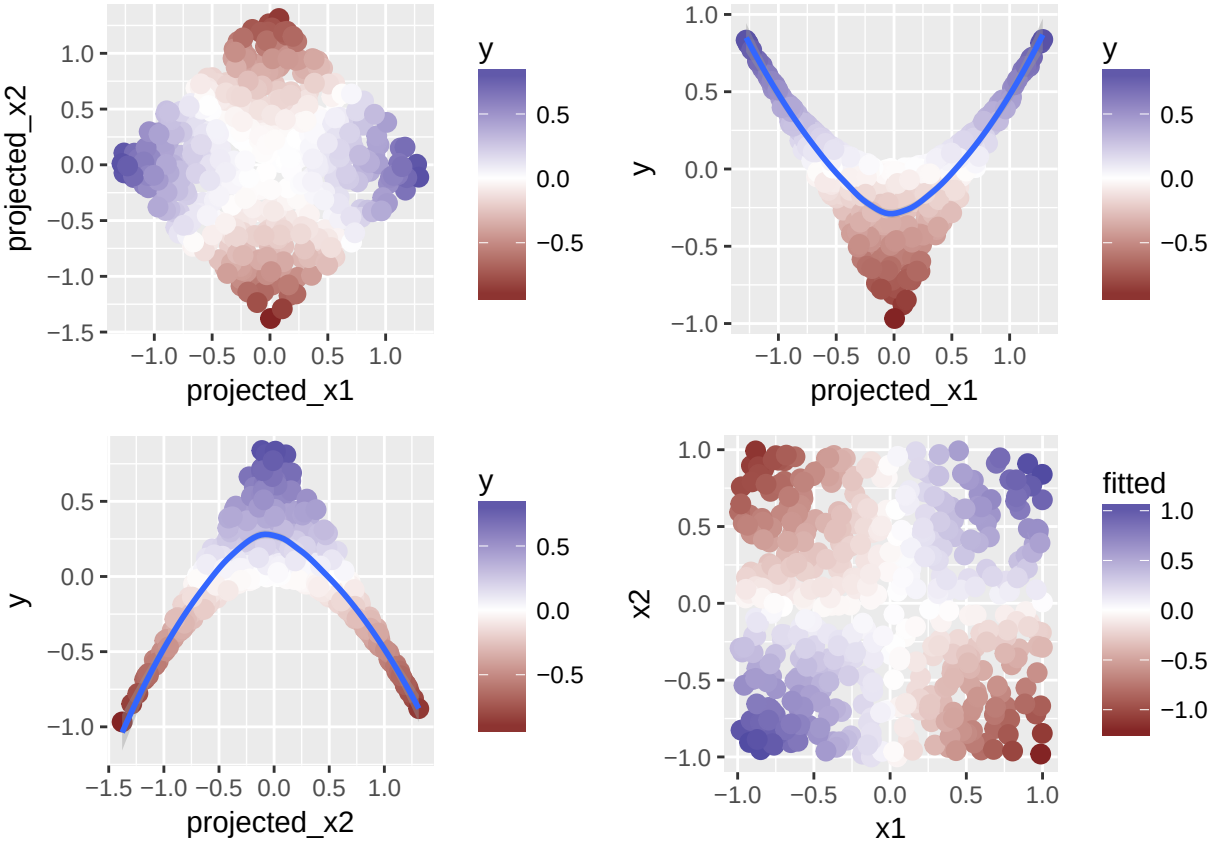
```
grid.arrange(all, x1y)
```



How about using the aforementioned projection?

```
data <- data %>%
  mutate(
    projected_x1 = 0.7*(x1 + x2),
    projected_x2 = -0.7*(x1 - x2)
  )

projected_all <- ggplot(data, aes(x = projected_x1, y = projected_x2)) +
  geom_point(aes(color = y), size = 3) +
  scale_color_gradient2()

projected_x1 <- ggplot(data, aes(x = projected_x1, y = y)) +
  geom_point(aes(color = y), size = 3) +
  geom_smooth() +
  scale_color_gradient2()

projected_x2 <- ggplot(data, aes(x = projected_x2, y = y)) +
  geom_point(aes(color = y), size = 3) +
  geom_smooth() +
  scale_color_gradient2()

fitted_x1 <- mgcv::gam(y~s(projected_x1), data = data)
fitted_x2 <- mgcv::gam(y~s(projected_x2), data = data)
```

```
data <- data %>%
  mutate(fitted = predict(fitted_x1) + predict(fitted_x2))

fitted <- ggplot(data, aes(x = x1, y = x2)) +
  geom_point(aes(color = fitted), size = 3) +
  scale_color_gradient2()
grid.arrange(projected_all, projected_x1, projected_x2, fitted, nrow = 2)
```



The bottom right picture shows the predictions with the projection pursuit approach, which resembles the original data pretty well. Again, the idea is to use an additive model to account for the interactions properly by first projecting the predictors optimally.

Chapter 9

Bagging and Boosting

Chapter 10

Introduction

You can label chapter and section titles using `{#label}` after them, e.g., we can reference Chapter 10. If you do not manually label them, there will be automatic labels anyway, e.g., Chapter **??**.

Figures and tables with captions will be placed in `figure` and `table` environments, respectively.

```
par(mar = c(4, 4, .1, .1))
plot(pressure, type = 'b', pch = 19)
```

Reference a figure by its code chunk label with the `fig:` prefix, e.g., see Figure 10.1. Similarly, you can reference tables generated from `knitr::kable()`, e.g., see Table 10.1.

```
knitr::kable(
  head(iris, 20), caption = 'Here is a nice table!',
  booktabs = TRUE
)
```

You can write citations, too. For example, we are using the **bookdown** package [@R-bookdown] in this sample book, which was built on top of R Markdown and **knitr** [@xie2015].

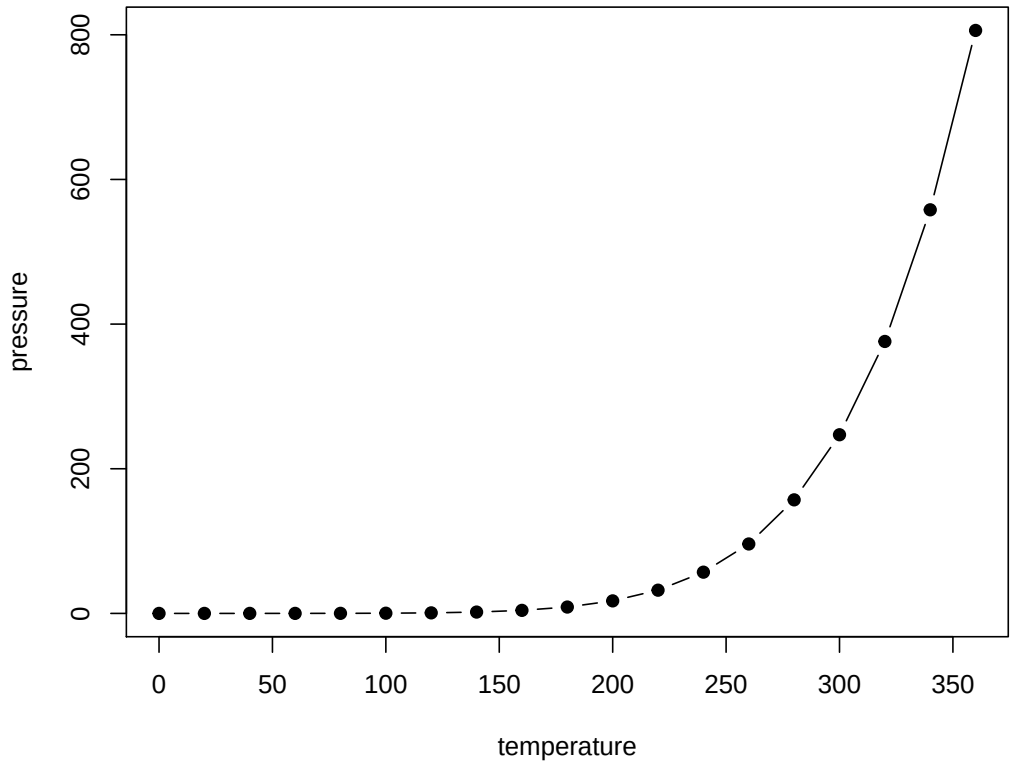


Figure 10.1: Here is a nice figure!

Table 10.1: Here is a nice table!

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.1	3.5	1.4	0.2	setosa
4.9	3.0	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa
5.0	3.6	1.4	0.2	setosa
5.4	3.9	1.7	0.4	setosa
4.6	3.4	1.4	0.3	setosa
5.0	3.4	1.5	0.2	setosa
4.4	2.9	1.4	0.2	setosa
4.9	3.1	1.5	0.1	setosa
5.4	3.7	1.5	0.2	setosa
4.8	3.4	1.6	0.2	setosa
4.8	3.0	1.4	0.1	setosa
4.3	3.0	1.1	0.1	setosa
5.8	4.0	1.2	0.2	setosa
5.7	4.4	1.5	0.4	setosa
5.4	3.9	1.3	0.4	setosa
5.1	3.5	1.4	0.3	setosa
5.7	3.8	1.7	0.3	setosa
5.1	3.8	1.5	0.3	setosa