Nonlinear Modeling

Kylie Ariel Bemis

3/19/2019

Nonlinear Models

All models are wrong, but some are useful.

– George Box

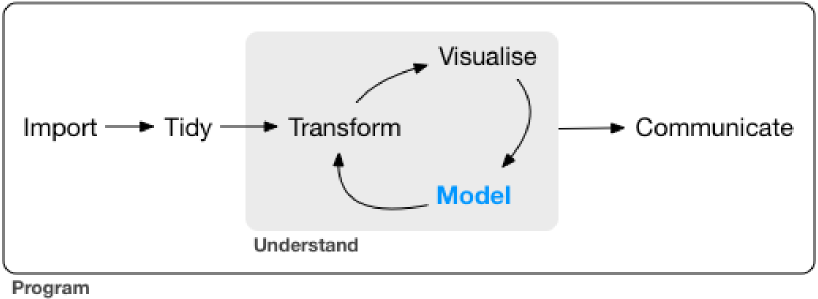


Figure 1: Wickham and Grolemund, *R for Data Science*

Overview of modeling goals

There are two general classes of analytic methods, designed for different purposes:

- Supervised learning
  - Prediction / classification
- Unsupervised learning
  - Clustering / class discovery

Statistical inference (including testing) can be a component of either supervised or unsupervised learning methods.

So far we have focused on linear regression, which is a type of supervised learning with a continuous response.

Supervised and unsupervised methods are discussed in greater detail in dedicated courses, but we will introduce them here.

Class comparison vs assignment vs discovery

Suppose we are interested in a categorical variable.

Then there are three general approaches we may wish to take:

- Class comparison
  - Use the categorical variable as an explanatory variable in linear regression
  - Use statistical tests to determine whether some continuous response depends on the level of the categorical variable
- Class assignment
  - Predict the level of a categorical response variable
  - Many supervised learning models are designed for classification
- Class discovery
  - The data is unlabeled, and you want to discover the class assignments

Let's focus on class assignment (classification) for now.

Suppose we want to predict delayed flights

```
library(nycflights13)
flights2 <- transmute(flights,
                      month = factor(month,
                                     levels=1:12),
                      dep_time = factor(dep_time %/% 100,
                                        levels=0:23),
                      arr_delay, dep_delay,
                      origin, dest, distance)
library(modelr)
set.seed(1)  # remember to set seeds for reproducibility!
flights3 <- resample(flights2, sample(nrow(flights), 20000))
flights4 <- as_tibble(flights3) %>%
  mutate(is_delayed = arr_delay > 0,
         status = factor(ifelse(is_delayed, "Delayed", "On_Time"),
                         levels=c("On_Time", "Delayed")))
```

```r
select(flights4, arr_delay:status, -origin, -dest)
```

```
## # A tibble: 20,000 x 5
##    arr_delay dep_delay distance is_delayed status
##        <dbl>     <dbl>    <dbl> <lgl>      <fct>
##  1        30         2     2565 TRUE       Delayed
##  2        -1        -5      319 FALSE      On_Time
##  3        14         9     1096 TRUE       Delayed
##  4         5         4     2446 TRUE       Delayed
##  5        -7        -5      733 FALSE      On_Time
##  6       -31         7     2475 FALSE      On_Time
##  7       -22        -9      479 FALSE      On_Time
##  8        -2        -5      628 FALSE      On_Time
##  9        70        47     1069 TRUE       Delayed
## 10        33        58      944 TRUE       Delayed
## # ... with 19,990 more rows
```

Logistic regression and GLMs

Logistic regression is similar to linear regression, but for a categorical response. It is a type of *generalized linear model* (GLM):

$$g(y) = \beta_0 + \beta_1 x_1 + ... + \beta_n x_n$$

A generalized linear model is parameterized similarly to a linear model, but the response is related to the linear predictor via a **link function** $g(y)$.

In addition, the response variable may follow a different probability distribution than the normal distribution.

Logistic regression

Logistic regression is defined by a **logit** link function that maps a binary response variable to continuous values:

$$logit(p) = log(\frac{p}{1-p})$$

The response variable is expected to follow either a Bernouli distribution (response is binary), or a Binomial distribution (response is # of "success" occurences out of some total # of binary occurences).

Like linear regression, logistic regression can be used either for prediction or for statistical testing. However, due to the link function, the interpretation of model coefficients is different than for linear regression.

We will focus on prediction.

Logistic regression for flight delays

The `glm()` function is used to fit generalized linear models.

The model is specified using a formula in the same was as `lm()`.

```
fit1 <- glm(status ~ month + dep_time + dep_delay + distance,
                  family=binomial(link="logit"), data=flights4)
```

We specify logistic regression by providing the model `family` as `binomial(link="logit")`.

Logistic regression for flight delays

```r
summary(fit1)
```

```
##
## Call:
## glm(formula = status ~ month + dep_time + dep_delay + distance,
##     family = binomial(link = "logit"), data = flights4)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -3.4502  -0.7186  -0.5373   0.4173   2.5308
##
## Coefficients:
##               Estimate Std. Error z value Pr(>|z|)
## (Intercept)  1.623e+00  1.066e+00   1.522 0.127888
## month2      -4.008e-02  9.105e-02  -0.440 0.659824
## month3      -2.946e-01  8.958e-02  -3.288 0.001008 **
## month4       5.878e-02  8.817e-02   0.667 0.504975
## month5      -5.659e-01  9.346e-02  -6.055  1.4e-09 ***
## month6      -2.200e-01  9.200e-02  -2.391 0.016799 *
## month7      -3.111e-01  9.068e-02  -3.430 0.000602 ***
## month8      -2.609e-01  8.760e-02  -2.978 0.002902 **
## month9      -8.414e-01  9.712e-02  -8.663  < 2e-16 ***
```

Predicted probabilities

Obtain predictions as a probabilty of status = “Delayed”.

```
flights4 %>%
  add_predictions(fit1, type="response") %>%
  select(arr_delay, is_delayed:pred)
```

```
## # A tibble: 20,000 x 4
##    arr_delay is_delayed status    pred
##        <dbl> <lgl>      <fct>    <dbl>
##  1        30 TRUE       Delayed 0.402
##  2        -1 FALSE      On_Time 0.243
##  3        14 TRUE       Delayed 0.612
##  4         5 TRUE       Delayed 0.411
##  5        -7 FALSE      On_Time 0.206
##  6       -31 FALSE      On_Time 0.420
##  7       -22 FALSE      On_Time 0.0724
##  8        -2 FALSE      On_Time 0.174
##  9        70 TRUE       Delayed 0.985
## 10        33 TRUE       Delayed 0.998
## # ... with 19,990 more rows
```

Prediction type

The `add_predictions()` function from the **modelr** package is actually just a wrapper for the `predict()` method that most modeling methods implement.

Many models may have predict values beyond just the desired response.

We can use the `type` argument in either `predict()` or `add_predictions()` to specify what kind of prediction is desired.

```
predict(fit1, flights4, type="response") %>% head()
```

```
##         1         2         3         4         5         6
## 0.4015101 0.2431121 0.6120191 0.4105554 0.2060036 0.4204553
```

For GLM's, we need `type = "response"` because we can obtain either the linear predictions (prior to transformation by the link function) or the response predictions (after transformation by the link function).

Obtaining predictions from models

Because `predict()` can be specialized for any type of model, the default help page (`?predict`) is not very useful for determining how to access different types of predictions.

Most modeling functions return an object of the same *class* as the name of the modeling function. The underlying `predict()` functions will be likewise named.

- `lm()` returns an object of type `lm`
  - `predict.lm()` is the underlying predict function
- `glm()` returns an object of type `glm`
  - `predict.glm()` is the underlying predict function

Therefore, use `?predict.glm` to see how to obtain the correct type of predictions for `glm` models.

Predicting a binary response

Many classifiers will output a numeric value (such as a probability) that must be converted into a binary value. Obtaining predictions typically requires choosing some kind of cutoff that will decide the class assignments:

```r
flights4 %>%
  add_predictions(fit1, type="response") %>%
  mutate(pred_status = ifelse(pred > 0.5,
                              "Delayed", "On_Time"),
         correct = status == pred_status ) %>%
  select(arr_delay, is_delayed:correct)
```

```
## # A tibble: 20,000 x 6
##    arr_delay is_delayed status    pred pred_status correct
##        <dbl> <lgl>      <fct>    <dbl> <chr>       <lgl>
##  1        30 TRUE       Delayed 0.402  On_Time     FALSE
##  2        -1 FALSE      On_Time 0.243  On_Time     TRUE
##  3        14 TRUE       Delayed 0.612  Delayed     TRUE
##  4         5 TRUE       Delayed 0.411  On_Time     FALSE
##  5        -7 FALSE      On_Time 0.206  On_Time     TRUE
##  6       -31 FALSE      On_Time 0.420  On_Time     TRUE
##  7       -22 FALSE      On_Time 0.0724 On_Time     TRUE
##  8        -2 FALSE      On_Time 0.174  On_Time     TRUE
##  9        70 TRUE       Delayed 0.985  Delayed     TRUE
```

Predicting a binary response (cont'd)

Note that it is important to know which class is being predicted (i.e., which is level is considered a "success").

When the binary response is a `factor`, the first level is coded as 0 ("failure") and the second level is coded as 1 ("success"). The predicted probabilities are the probabilities of "success":

```r
head(flights4$status)
```

```
## [1] Delayed On_Time Delayed Delayed On_Time On_Time
## Levels: On_Time Delayed
```

"Delayed" is the second level.

Therefore, we are predicting the probability that status = "Delayed".

Predicting a binary response (cont'd)

Varying the cutoff can change the accuracy:

```r
flights4 %>%
  add_predictions(fit1, type="response") %>%
  mutate(pred4 = ifelse(pred > 0.4, "Delayed", "On_Time"),
         pred5 = ifelse(pred > 0.5, "Delayed", "On_Time"),
         pred6 = ifelse(pred > 0.6, "Delayed", "On_Time")) %>%
  summarize(acc4 = mean(status == pred4, na.rm=TRUE),
            acc5 = mean(status == pred5, na.rm=TRUE),
            acc6 = mean(status == pred6, na.rm=TRUE))
```

```
## # A tibble: 1 x 3
##    acc4  acc5  acc6
##   <dbl> <dbl> <dbl>
## 1 0.792 0.797 0.795
```

Sensitivity vs specificity

Besides overall accuracy (% classified correctly), there are two types of accuracy that we can balance: **sensitivity** and **specificity**

- Sensitivity is the *true positive rate*
  - Proportion of correctly-identified positives among actual positives
  - If a flight will be delayed, how likely are we to classify it as delayed?
- Specificity is the *true negative rate*
  - Proportion of correctly-identified negatives among actual negatives
  - If a flight will *not* be delayed, how likely are we to classify it as *not* delayed?

These definitions depend on which class is considered the "positive" or "success" class.

It can be particularly important to pay attention to sensitivity and specificity when the response class sizes are unbalanced, as the overall accuracy can be misleading in such situations.

Sensitivity vs specificity for delayed flights

```
flights5 <- flights4 %>%
  add_predictions(fit1, type="response") %>%
  mutate(pred = ifelse(pred > 0.5,
                       "Predicted Delayed",
                       "Predicted On_Time"))
table(flights5$status, flights5$pred)[,2:1] # "confusion matrix"
```

```
##
##           Predicted On_Time Predicted Delayed
##   On_Time             10917               766
##   Delayed              3197              4606
```

```
4606 / (3197 + 4606) # sensitivity
```

```
## [1] 0.5902858
```

```
10917 / (10917 + 766) # specificity
```

```
## [1] 0.9344346
```

Calculating sensitivity and specificity

```
yobs <- flights4$is_delayed
ypred <- predict(fit1, flights4, type="response")

sens <- function(p) {
  mean((ypred > p)[yobs], na.rm=TRUE)
}
sens(0.5)
```

```
## [1] 0.5902858
```

```
spec <- function(p) {
  mean(!(ypred > p)[!yobs], na.rm=TRUE)
}
spec(0.5)
```

```
## [1] 0.9344346
```

Receiving operator characteristic (ROC) curve

Changing the cutoff probability for class assignment can affect the sensitivity and specificity.

It can be useful to calculate and plot the tradeoff between sensitivity and specificity for different cutoffs.

This is traditionally visualized as an ROC curve, which plots the *true positive rate* (sensitivity) against the *false positive rate* (1 - specificity):

```
roc <- tibble(p=seq(from=0, to=1, by=0.01))
roc <- roc %>%
  mutate(sensitivity = map_dbl(p, sens),
         specificity = map_dbl(p, spec))
```

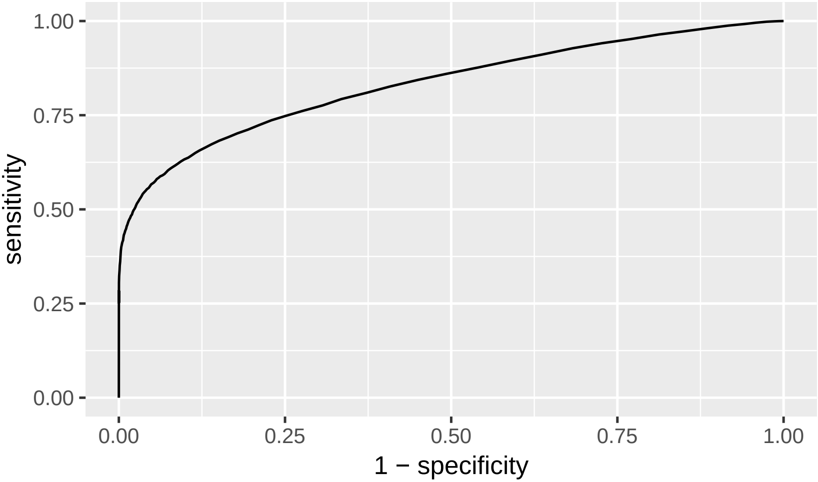
```r
roc
```

```
## # A tibble: 101 x 3
##        p sensitivity specificity
##    <dbl>       <dbl>       <dbl>
##  1  0          1         0
##  2  0.01       1         0
##  3  0.02       1         0
##  4  0.03       1         0.000257
##  5  0.04       1         0.00103
##  6  0.05       1.000     0.00282
##  7  0.06       1.000     0.00676
##  8  0.07       0.999     0.0135
##  9  0.08       0.998     0.0254
## 10  0.09       0.996     0.0417
## # ... with 91 more rows
```

Plotting the ROC

A strong classifier will have an area-under-the-curve (AUC) close to 1.

```
ggplot(roc, aes(x=1 - specificity, y=sensitivity)) + geom_line()
```



Logistic regression with many predictors

Suppose we have a classification problem with “many” predictors:

```r
N <- 1000
P <- 10
set.seed(2)
x <- matrix(rnorm(N * P), nrow=N, ncol=P)
colnames(x) <- paste0("x", 1:P)
y <- rbinom(N, 1, ifelse(x[,1] > 0, 0.6, 0.4))
data <- bind_cols(as_tibble(x), tibble(y=y))
```

Which predictors should we include in the model?

```r
fit2 <- glm(y ~ ., data=data, family=binomial(link="logit"))
summary(fit2)
```

```
##
## Call:
## glm(formula = y ~ ., family = binomial(link = "logit"), data = data)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -1.8103  -1.1607   0.8226   1.0993   1.6841
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept)  0.10449    0.06557   1.594    0.111
## x1           0.39305    0.06668   5.894 3.76e-09 ***
## x2           0.02853    0.06561   0.435    0.664
## x3          -0.07423    0.06395  -1.161    0.246
## x4           0.10222    0.06616   1.545    0.122
## x5           0.03855    0.06704   0.575    0.565
## x6           0.02463    0.06734   0.366    0.715
## x7           0.04161    0.06190   0.672    0.501
## x8          -0.03715    0.06700  -0.554    0.579
## x9           0.07379    0.06314   1.169    0.243
## x10         -0.09847    0.06713  -1.467    0.142
```

Sparse logistic regression

The `glmnet` package fits generalized linear models with an L1 and L2 penalty on the coefficients.

The L1 penalty (the "lasso") forces many of the coefficients to be 0, essentially removing them from the model.

```
library(glmnet)
fit3 <- glmnet(x, y, family="binomial")
```

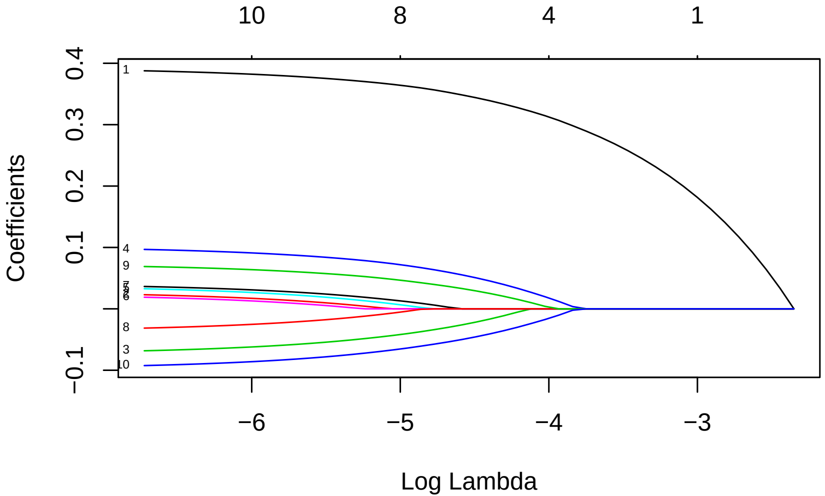
This is a type of "sparse" model.

A sparse model uses only a small subset of the input predictors (because most of the model parameters are forced to 0 by constraints).

Sparse coefficient estimates

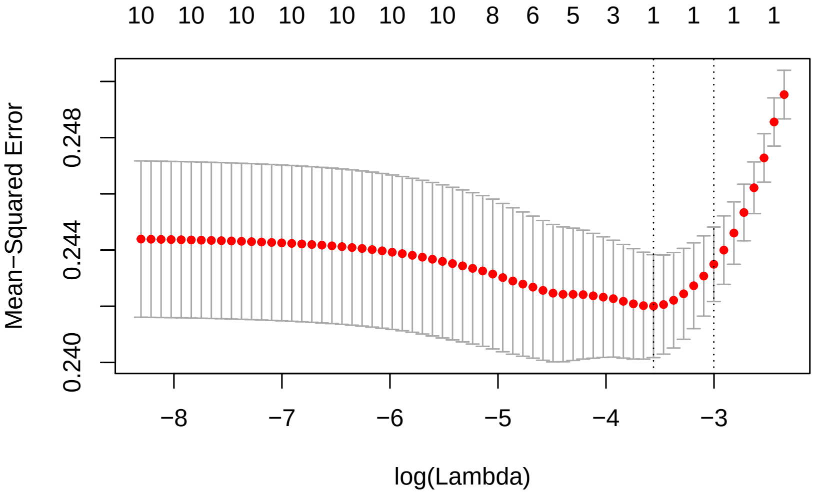
Larger values of the sparsity parameter λ force more coefficients to 0:

```
plot(fit3, xvar="lambda", label=TRUE)
```



Cross-validation is used to select lambda

```
fit4 <- cv.glmnet(x, y)
plot(fit4)
```



Sparse coefficients from selected model

```
coef(fit4, s="lambda.min")
```

```
## 11 x 1 sparse Matrix of class "dgCMatrix"
##                      1
## (Intercept) 0.52291302
## x1          0.06592032
## x2          .
## x3          .
## x4          .
## x5          .
## x6          .
## x7          .
## x8          .
## x9          .
## x10         .
```

Only `x1` has a non-zero coefficient estimate.

Using `glmnet` for sparse models

Because `glmnet` implements sparse GLM's, it can also be used for sparse linear regression, and many families of sparse generalized linear models.

- Good for automated selection of important predictor variables
- Can be useful for predictive accuracy by removing noisy predictors
- Statistical testing for sparse models is a subject of research…
- Model specification is different from `lm()` or `glm()`
  - No model formula, instead use a matrix of predictors
  - Categorical variables (factors) must be manually converted to indicator variables using `model.matrix()`
- Predictor variables must be on the same units scale for the sparsity constraint to make any sense
  - Predictors are automatically standardized
  - Coefficients are returned to original scale
- Interpretation of the regression coefficients can be difficult

Modeling packages

One of the advantages of R is almost every conceivable machine learning or statistical model is implemented in some package.

For example:

- `e1071` : support vector machines (SVMs)
- `rpart` : classification and regression trees
- `igraph` : network analysis
- `nnet` : neural networks and multinomial regression
- `randomForest` : random forests
- `kernlab` : kernel-based machine learning

etc.

One of the disadvantages of this is that many of these models are implemented with different function conventions and syntax.

Modeling packages (cont'd)

For example, here are a few different ways to obtain class probabilities from a classifier trained by different packages:

Function	Package	Code
`lda`	`MASS`	`predict(obj)`
`glm`	`stats`	`predict(obj, type = "response")`
`gbm`	`gbm`	`predict(obj, type = "response", n.trees)`
`mda`	`mda`	`predict(obj, type = "posterior")`
`rpart`	`rpart`	`predict(obj, type = "prob")`
`Weka`	`RWeka`	`predict(obj, type = "probability")`
`logitboost`	`LogitBoost`	`predict(obj, type = "raw", nIter)`
`pamr.train`	`pamr`	`pamr.predict(obj, type = "posterior")`

Figure 2: Modeling conventions

Supervised learning with `caret`

The `caret` package attempts to provide a consistent interface to 237 machine learning models from 30+ different R packages.

The basic strategy for training machine learning methods with `caret` is as follows:

```
Define sets of model parameter values to evaluate
for each parameter set do
    for each resampling iteration do
        Hold–out specific samples
        [Optional] Pre–process the data
        Fit the model on the remainder
        Predict the hold–out samples
    end
    Calculate the average performance across hold–out predictions
end
Determine the optimal parameter set
Fit the final model to all the training data using the optimal parameter set
```

Figure 3: https://topepo.github.io/caret

Supervised learning with `caret` (cont'd)

The `caret` package provides the following primary functions:

- `createDataPartition()` : partitions the data into train / test split, using stratified sampling to create balanced partitions
- `preProcess()` : pre-processes the data (centering, scaling, imputating missing data, etc.)
- `trainControl()` : controls various computational aspects of how the model is trained (type of cross-validation, etc.)
- `train()` : trains a model
- `predict()` : provides a consistent way of accessing predictions from any machine learning model supported by `caret`

Example: Sonar data

We will use the “Sonar” data from the `mlbench` package.

```
library(mlbench)
data(Sonar)
```

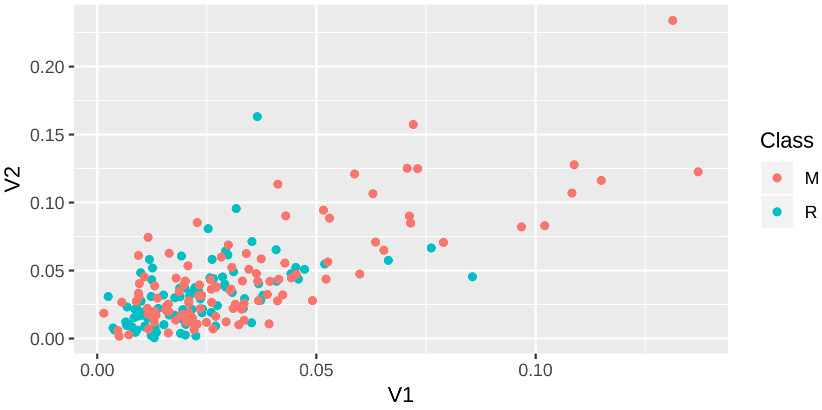
The dataset consists of 208 observations on 60 (continuous) explanatory variables and 1 (categorical) response variable.

The 60 explanatory variables represent energy from different frequencies of sonar signals.

The goal is to predict whether the sonar signals are being bounced off a metal cylinder (“M”) or a cylindrical rock (“R”).

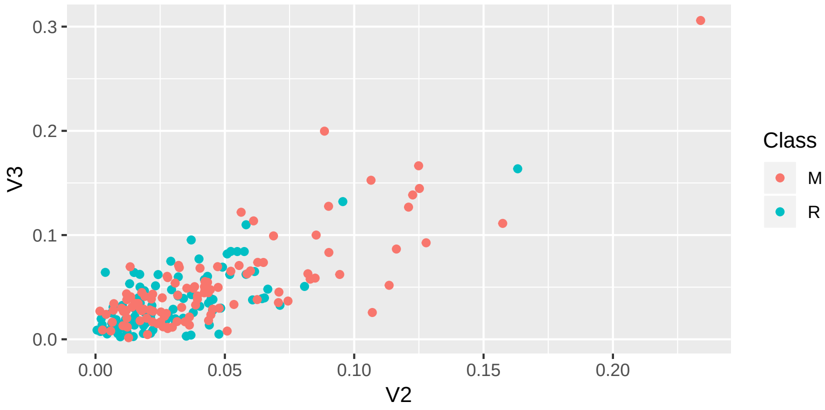
First two frequency channels

```r
ggplot(Sonar, aes(x=V1, y=V2, color=Class)) + geom_point()
```



Second two frequency channels

```
ggplot(Sonar, aes(x=V2, y=V3, color=Class)) + geom_point()
```



Partition the data

First, we use `createDataPartition()` to partition the data into training and testing sets.

```
library(caret)
set.seed(3) # reproducibility!
train_ids <- createDataPartition(Sonar$Class,
                                 p=0.75, list=FALSE)
Sonar_train <- Sonar[train_ids,]
Sonar_test <- Sonar[-train_ids,]
```

The training set will be used for cross-validation to select tuning parameters for the machine learning models.

Train logistic regression with caret

We use the `train()` function to train a logistic regression model.

```
glmFit <- train(Class ~ ., data=Sonar_train,
                method="glm", family=binomial(link="logit"),
                trControl=trainControl(method="none"))
```

```
## Warning: glm.fit: algorithm did not converge
```

```
## Warning: glm.fit: fitted probabilities numerically 0 or 1 occ
```

Most machine learning models have some kind of tuning parameters, but logistic regression does not.

Therefore, we set the training control to `method="none"`, because no cross-validation is required to select tuning parameters.

```r
glmFit
```

```
## Generalized Linear Model
##
## 157 samples
##  60 predictor
##   2 classes: 'M', 'R'
##
## No pre-processing
## Resampling: None
```

Confusion matrix for logistic regression

```
confusionMatrix(predict(glmFit, Sonar_test), Sonar_test$Class)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  M  R
##          M 20  5
##          R  7 19
##
##                Accuracy : 0.7647
##                  95% CI : (0.6251, 0.8721)
##     No Information Rate : 0.5294
##     P-Value [Acc > NIR] : 0.0004667
##
##                   Kappa : 0.53
##  Mcnemar's Test P-Value : 0.7728300
##
##             Sensitivity : 0.7407
##             Specificity : 0.7917
##          Pos Pred Value : 0.8000
##          Neg Pred Value : 0.7308
##              Prevalence : 0.5294
```

Train sparse logistic regression

We can use the same `train()` function to train a `glmnet` sparse logistic regression model.

We will use repeated 5-fold cross-validation to determine the sparsity parameter λ:

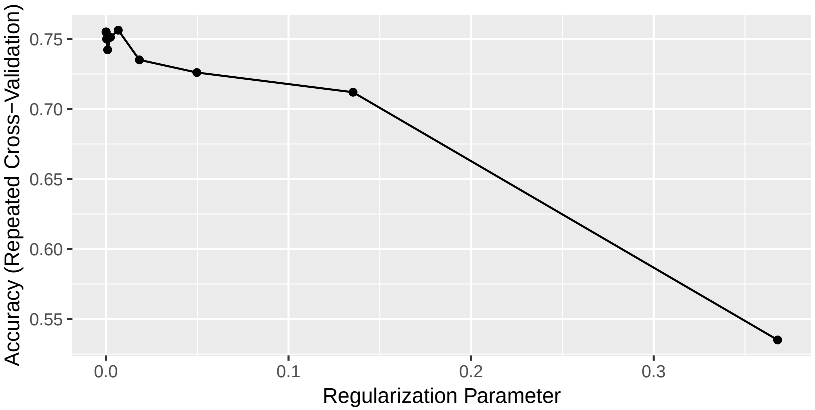
```
ctrl <- trainControl(method="repeatedcv", number=5, repeats=5)
grd <- expand.grid(alpha=1, lambda=exp(-10:-1))

set.seed(4) # training uses random samples for CV!
glmnetFit <- train(Class ~ ., data=Sonar_train,
                   method="glmnet", family="binomial",
                   trControl=ctrl, tuneGrid=grd)
```

```r
glmnetFit
```

```
## glmnet
##
## 157 samples
##  60 predictor
##   2 classes: 'M', 'R'
##
## No pre-processing
## Resampling: Cross-Validated (5 fold, repeated 5 times)
## Summary of sample sizes: 125, 126, 125, 126, 126, 125, ...
## Resampling results across tuning parameters:
##
##   lambda        Accuracy   Kappa
##   4.539993e-05  0.7550054  0.5062840
##   1.234098e-04  0.7549624  0.5064028
##   3.354626e-04  0.7498817  0.4960950
##   9.118820e-04  0.7422634  0.4813584
##   2.478752e-03  0.7511801  0.4996081
##   6.737947e-03  0.7562634  0.5105089
##   1.831564e-02  0.7350538  0.4671029
##   4.978707e-02  0.7260269  0.4481378
##   1.353353e-01  0.7119516  0.4138395
##   3.678794e-01  0.5350833  0.0000000
```

```r
ggplot(glmnetFit)
```



Check the coefficients from the best model

```r
coef(glmnetFit$finalModel, s=glmnetFit$bestTune$lambda)
```

```
## 61 x 1 sparse Matrix of class "dgCMatrix"
##                         1
## (Intercept)   7.0871054
## V1          -25.1603144
## V2            .
## V3           10.6081554
## V4          -11.4336999
## V5            .
## V6            1.2019394
## V7            3.8717424
## V8           10.0865614
## V9           -7.3967575
## V10           .
## V11          -2.1301003
## V12          -8.7172197
## V13           1.4771280
## V14           0.4331108
## V15           .
## V16           0.3511447
## V17           1.5755718
```

Confusion matrix for sparse logistic regression

```r
confusionMatrix(predict(glmnetFit, Sonar_test), Sonar_test$Class)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  M  R
##          M 20  5
##          R  7 19
##
##                Accuracy : 0.7647
##                  95% CI : (0.6251, 0.8721)
##     No Information Rate : 0.5294
##     P-Value [Acc > NIR] : 0.0004667
##
##                   Kappa : 0.53
##  Mcnemar's Test P-Value : 0.7728300
##
##             Sensitivity : 0.7407
##             Specificity : 0.7917
##          Pos Pred Value : 0.8000
##          Neg Pred Value : 0.7308
##              Prevalence : 0.5294
```

Train single-hidden-layer neural network

We can use the same `train()` function to train a single-hidden layer neural network using the `nnet` package.

We will use repeated 5-fold cross-validation to determine the optimal number of units in the hidden layer:

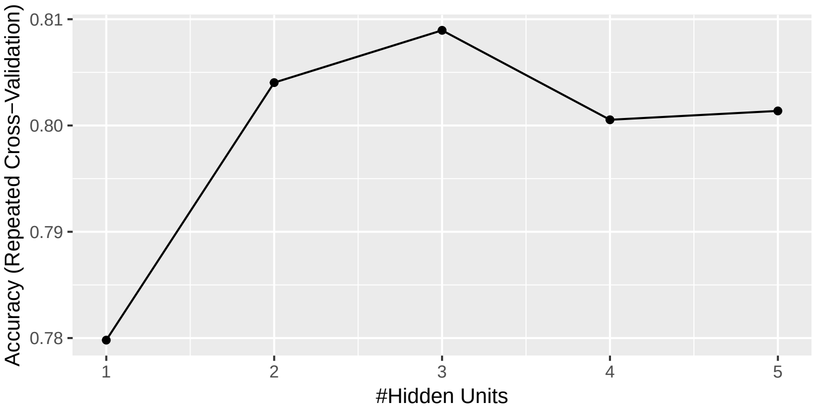
```
ctrl <- trainControl(method="repeatedcv", number=5, repeats=5)
grd <- expand.grid(size=1:5, decay=0)

set.seed(5) # training uses random samples for CV!
nnetFit <- train(Class ~ ., data=Sonar_train,
                 method="nnet", trControl=ctrl, tuneGrid=grd)
```

```
nnetFit
```

```
## Neural Network
##
## 157 samples
##  60 predictor
##   2 classes: 'M', 'R'
##
## No pre-processing
## Resampling: Cross-Validated (5 fold, repeated 5 times)
## Summary of sample sizes: 126, 127, 125, 125, 125, 125, ...
## Resampling results across tuning parameters:
##
##   size  Accuracy   Kappa
##   1     0.7798065  0.5516167
##   2     0.8040349  0.6032537
##   3     0.8089543  0.6121657
##   4     0.8005376  0.5965626
##   5     0.8013710  0.5983530
##
## Tuning parameter 'decay' was held constant at a value of 0
## Accuracy was used to select the optimal model using the largest valu
## The final values used for the model were size = 3 and decay = 0.
```

```r
ggplot(nnetFit)
```



Confusion matrix for neural network

```r
confusionMatrix(predict(nnetFit, Sonar_test), Sonar_test$Class)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  M  R
##          M 22  6
##          R  5 18
##
##                Accuracy : 0.7843
##                  95% CI : (0.6468, 0.8871)
##     No Information Rate : 0.5294
##     P-Value [Acc > NIR] : 0.0001502
##
##                   Kappa : 0.5661
##  Mcnemar's Test P-Value : 1.0000000
##
##             Sensitivity : 0.8148
##             Specificity : 0.7500
##          Pos Pred Value : 0.7857
##          Neg Pred Value : 0.7826
##              Prevalence : 0.5294
```

Other supervised learning packages: Keras

Another major machine learning package is the R interface to Keras.

The `keras` package maintained by R Studio is a high-level neural network API backed by TensorFlow, among other backends.

It provides a flexible interface for building and training many different types of neural networks.

The default installation is CPU-based, but the same models can also be trained on NVIDIA GPUs.

See https://keras.rstudio.com for more information.

Unsupervised learning

Suppose we want to discover classes within unlabeled data.

Consider the following dataset, giving measurements on different flowers.

```r
head(as_tibble(iris))
```

```
## # A tibble: 6 x 5
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##          <dbl>       <dbl>        <dbl>       <dbl> <fct>
## 1          5.1         3.5          1.4         0.2 setosa
## 2          4.9         3            1.4         0.2 setosa
## 3          4.7         3.2          1.3         0.2 setosa
## 4          4.6         3.1          1.5         0.2 setosa
## 5          5           3.6          1.4         0.2 setosa
## 6          5.4         3.9          1.7         0.4 setosa
```

```r
iris2 <- select(iris, Sepal.Length:Petal.Width)
```

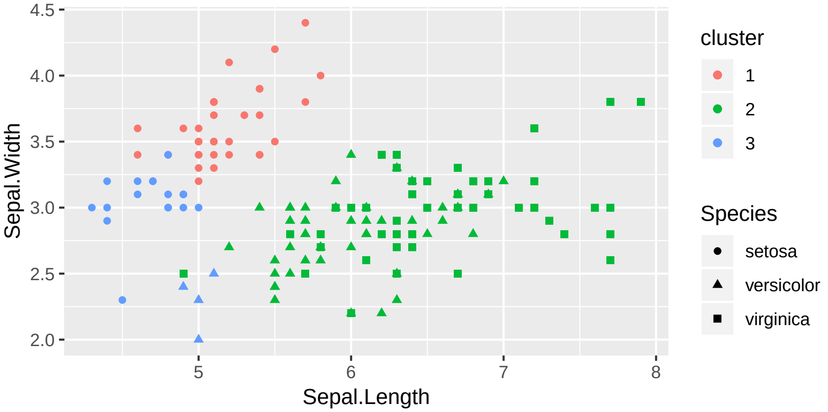
K-means clustering

```
fit_kmeans <- kmeans(iris2, centers=3)
fit_kmeans
```

```
## K-means clustering with 3 clusters of sizes 33, 96, 21
##
## Cluster means:
##   Sepal.Length Sepal.Width Petal.Length Petal.Width
## 1     5.175758    3.624242     1.472727   0.2727273
## 2     6.314583    2.895833     4.973958   1.7031250
## 3     4.738095    2.904762     1.790476   0.3523810
##
## Clustering vector:
##   [1] 1 3 3 3 1 1 1 1 3 3 1 1 3 3 1 1 1 1 1 1 1 1 1 3 3 1 1 1 3 3
##  [36] 1 1 1 3 1 1 3 3 1 1 3 1 3 1 1 2 2 2 2 2 2 2 3 2 2 3 2 2 2 2 2
##  [71] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 2 2 2 2 3 2 2
## [106] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
## [141] 2 2 2 2 2 2 2 2 2 2
##
## Within cluster sum of squares by cluster:
## [1]   6.432121 118.651875  17.669524
##  (between_SS / total_SS =  79.0 %)
##
```

K-means clustering (cont'd)

```
iris %>% mutate(cluster = as.factor(fit_kmeans$cluster)) %>%
  ggplot(aes(x=Sepal.Length, y=Sepal.Width,
             color=cluster, shape=Species)) +
  geom_point()
```



K-means clustering (cont'd)

```
iris %>%
  mutate(cluster = as.factor(fit_kmeans$cluster)) %>%
  ggplot(aes(x=Petal.Length, y=Petal.Width,
             color=cluster, shape=Species)) +
  geom_point()
```