Initiation to IA

Silvia Bottini
Silvia.Bottini@univ-cotedazur.fr

What is Machine Learning?

In 1959, **Arthur Samuel**, a computer scientist who pioneered the study of artificial intelligence, described machine learning as **“the study that gives computers the ability to learn without being explicitly programmed.”**

Alan Turing’s seminal paper (**Turing,** 1950) introduced a benchmark standard for demonstrating machine intelligence, such that **a machine has to be intelligent and responsive in a manner that cannot be differentiated from that of a human being.**

A more technical definition given by **Tom M. Mitchell**’s (1997) : “**A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience**

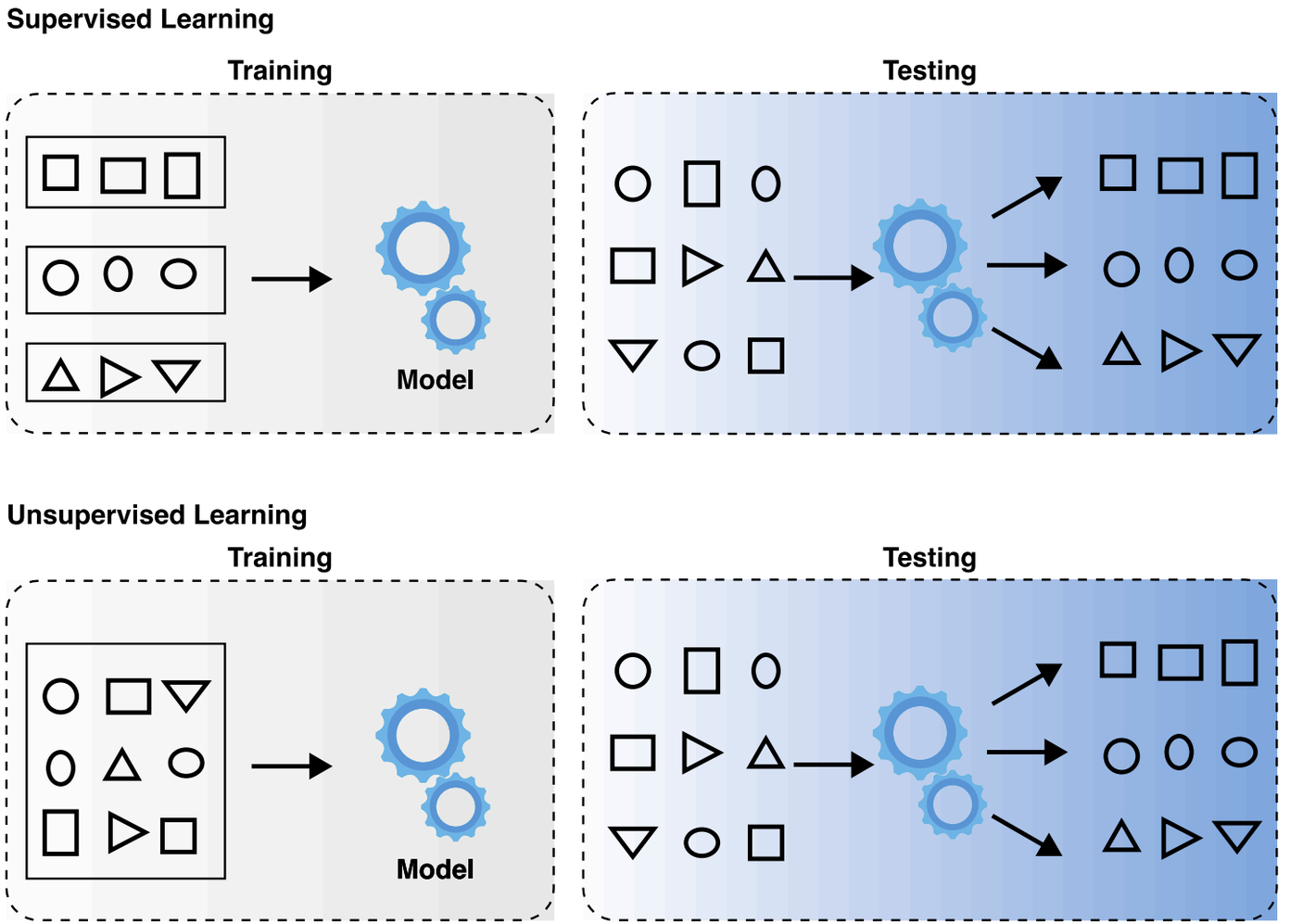
Machine learning

Machine learning is becoming widespread among data scientist and is deployed in hundreds of products you use daily. One of the first ML application was **spam filter**.

Following are other application of Machine Learning

- Identification of unwanted spam messages in email
- Segmentation of customer behavior for targeted advertising
- Reduction of fraudulent credit card transactions
- Optimization of energy use in home and office building
- Facial recognition

Supervised vs unsupervised learning



Supervised learning

In supervised learning the machine experiences the examples along with the labels or targets for each example. The labels in the data help the algorithm to correlate the features.
Two of the most common supervised machine learning tasks are **classification** and **regression**.

In **regression** problems the machine must predict the value of a continuous response variable. Examples of regression problems include predicting the sales for a new product, or the salary for a job based on its description.

In **classification** problems the machine must learn to predict discrete values. That is, the machine must predict the most probable category, class, or label for new examples. Applications of classification include predicting whether a stock's price will rise or fall, or deciding if a news article belongs to the politics or leisure section.

Supervised learning

In **supervised learning**, the training data you feed to the algorithm includes a label.

Here is the list of some fundamental supervised learning algorithms.

- Linear regression
- Logistic regression
- Nearest Neighbors
- Support Vector Machine (SVM)
- Decision trees and Random Forest
- Neural Networks

Unsupervised learning

When we have unclassified and unlabeled data, the system attempts to uncover patterns from the data . There is no label or target given for the examples. One common task is to group similar examples together called clustering.

In **unsupervised learning**, the training data is unlabeled. The system tries to learn without a reference. Below is a list of unsupervised learning algorithms.

- K-mean
- Hierarchical Cluster Analysis
- Expectation Maximization
- Visualization and dimensionality reduction
- Principal Component Analysis
- Kernel PCA
- Locally-Linear Embedding

Reinforcement learning

Reinforcement learning refers to goal-oriented algorithms, which learn how to attain a complex objective (goal) or maximize along a particular dimension over many steps. This method allows machines and software agents to automatically determine the ideal behaviour within a specific context in order to maximize its performance. Simple reward feedback is required for the agent to learn which action is best; this is known as the reinforcement signal. For example, maximize the points won in a game over many moves.

Today’s program

Machine learning

- **Supervised learning**
  - **Simple Linear regression**
  - **Multiple Linear regression**
  - **Factor regression**
  - **Logistic regression**
  - **Decision Trees**
  - **Random Forests**
  - **SVM**
- **Unsupervised learning**
  - **Dimensionality reduction**
  - **Clustering**

Today’s program

Machine learning

- **Supervised learning**
  - **Simple Linear regression**
  - **Multiple Linear regression**
  - **Factor regression**
  - **Logistic regression**
  - **Decision Trees**
  - **Random Forests**
  - **SVM**
- **Unsupervised learning**
  - **Dimensionality reduction**
  - **Clustering**

Simple Linear regression

Linear regression answers a simple question: Can you measure an exact relationship between one target variables and a set of predictors?

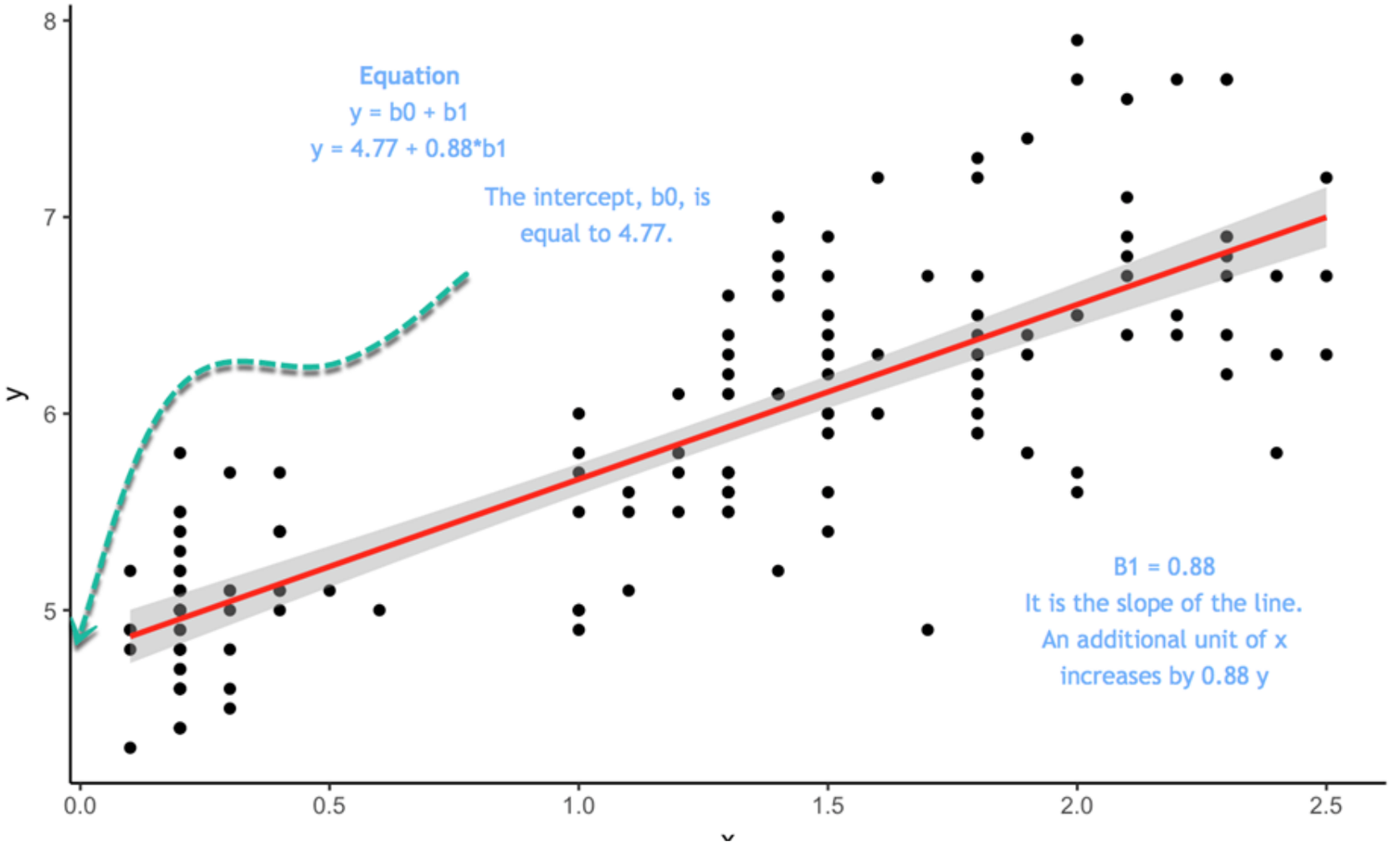
The simplest of probabilistic models is the straight line model:

$$y = \beta_0 + \beta_1 x + \varepsilon$$

where

- y = Dependent variable
- x = Independent variable
- ε = random error component
- β_0 = intercept
- β_1 = Coefficient of x

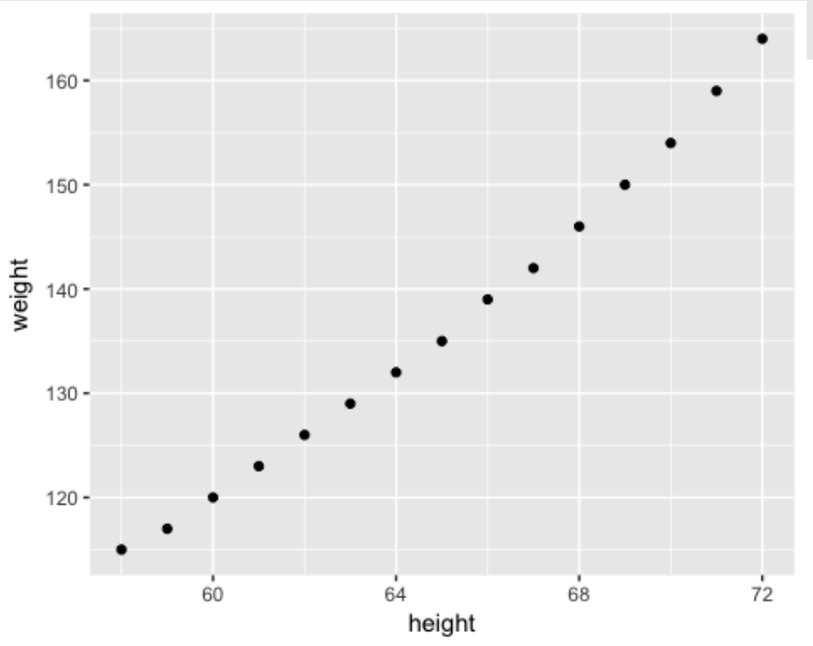
Simple Linear regression



Scatter plot

We will use a very simple dataset to explain the concept of simple linear regression. We will import the Average Heights and weights for American Women. The dataset contains 15 observations. You want to measure whether Heights are positively correlated with weights.

```
library(ggplot2)
path <- ‘./datasets/women.csv’
df <-read.csv(path)
ggplot(df,aes(x=height, y = weight))+ geom_point()
```



The scatterplot suggests a general tendency for y to increase as x increases.
Let’s now find out by how much increases for each additional .

Least square estimate

You want to estimate: $y = \beta_0 + \beta_1 x + \varepsilon$

The goal of the OLS regression is to minimize the following equation:

$$\Sigma(y_i - \hat{y}_i)^2 = \Sigma e_i^2$$

where

y_i is the actual value and $\hat{y}_i$ is the predicted value.

The solution for β_0 is $\beta_0 = \overline{y} - \beta_1 \overline{x}$

Note that $\overline{x}$ means the average value of x

The solution for β is $\beta = \frac{Cov(x,y)}{Var(x)}$

The beta coefficient implies that for each additional height, the weight increases by 3.45.

```
beta <- cov(df$height, df$weight) / var (df$height) beta
##[1] 3.45
```

Today’s program

Machine learning

- **Supervised learning**
  - **Simple Linear regression**
  - **Multiple Linear regression**
  - **Factor regression**
  - **Logistic regression**
  - **Decision Trees**
  - **Random Forests**
  - **SVM**
- **Unsupervised learning**
  - **Dimensionality reduction**
  - **Clustering**

Multiple Linear regression

More practical applications of regression analysis employ models that are more complex than the simple straight-line model. The probabilistic model that includes more than one independent variable is called **multiple regression models**. The general form of this model is:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_k x_k + \varepsilon$$

In matrix notation, you can rewrite the model:

- $Y = \beta X + \varepsilon$

The dependent variable y is now a function of k independent variables. The value of the coefficient β_i determines the contribution of the independent variable x_i and β_0.

We briefly introduce the assumption we made about the random error ε of the OLS:

- Mean equal to 0
- Variance equal to σ^2
- Normal distribution
- Random errors are independent (in a probabilistic sense)

Multiple Linear regression

You need to solve for β , the vector of regression coefficients that minimise the sum of the squared errors between the predicted and actual y values.

The closed-form solution is:

$$\beta = (X^T X)^{-1} X^T Y$$

with:

- indicates the **transpose** of the matrix X
- $(X^T X)^{-1}$ indicates the **invertible matrix**

Mtcars dataset

The data was extracted from the 1974 *Motor Trend* US magazine, and comprises fuel consumption and 10 aspects of automobile design and performance for 32 automobiles (1973–74 models).

A data frame with 32 observations on 11 (numeric) variables.

[, 1]	mpg	Miles/(US) gallon
[, 2]	cyl	Number of cylinders
[, 3]	disp	Displacement (cu.in.)
[, 4]	hp	Gross horsepower
[, 5]	drat	Rear axle ratio
[, 6]	wt	Weight (1000 lbs)
[, 7]	qsec	1/4 mile time
[, 8]	vs	Engine (0 = V-shaped, 1 = straight)
[, 9]	am	Transmission (0 = automatic, 1 = manual)
[,10]	gear	Number of forward gears
[,11]	carb	Number of carburetors

Multiple Linear regression

We use the mtcars dataset. Our goal is to predict the mile per gallon over a set of features.

```
library(dplyr)
df <- mtcars % > %
select(-c(am, vs, cyl, gear,
carb))
glimpse(df)
```

This is like a transposed version of print: columns run down the page, and data runs across. This makes it possible to see every column in a data frame.

```
## Observations: 32
## Variables: 6
## $ mpg <dbl> 21.0, 21.0, 22.8, 21.4, 18.7, 18.1, 14.3, 24.4, 22.8,
19....
## $ disp <dbl> 160.0, 160.0, 108.0, 258.0, 360.0, 225.0, 360.0,
146.7, 1...
## $ hp <dbl> 110, 110, 93, 110, 175, 105, 245, 62, 95, 123, 123,
180, ...
## $ drat <dbl> 3.90, 3.90, 3.85, 3.08, 3.15, 2.76, 3.21, 3.69, 3.92,
3.9...
```

lm() function

You can use the lm() function to compute the parameters. The basic syntax of this function is:

```
lm(formula, data, subset)
Arguments:
-formula: The equation you want to estimate
-data: The dataset used
-subset: Estimate the model on a subset of the
dataset
```

Your objective is to estimate the mile per gallon based on a set of variables. The equation to estimate is:

$$mpg = \beta_0 + \beta_1 disp_1 + \beta_2 hp_2 + \beta_3 drat_3 + \beta_4 wt_4 + \varepsilon$$

```
model <- mpg~.disp + hp + drat + wt
fit <- lm(model, df)
fit
```

Store the model to estimate

Estimate the model with the data fra

output

```
## ## Call:
## lm(formula = model, data = df)
##
## Coefficients:
## (Intercept) disp hp drat wt
## 16.53357 0.00872 -0.02060 2.01577 -4.38546
## qsec
## 0.64015
```

You can access more details such as the significance of the coefficients, the degree of freedom and the shape of the residuals with the summary() function.

```
summary(fit)
```

output

```
## return the p-value and coefficient
##
## Call:
## lm(formula = model, data = df)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -3.5404 -1.6701 -0.4264  1.1320  5.4996
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 16.53357   10.96423   1.508  0.14362
## disp         0.00872    0.01119   0.779  0.44281
## hp          -0.02060    0.01528  -1.348  0.18936
## drat         2.01578    1.30946   1.539  0.13579
## wt          -4.38546    1.24343  -3.527  0.00158 **
## qsec         0.64015    0.45934   1.394  0.17523
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.558 on 26 degrees of freedom
## Multiple R-squared:  0.8489, Adjusted R-squared:  0.8199
## F-statistic: 29.22 on 5 and 26 DF, p-value: 6.892e-10
```

- The table proves that there is a strong negative relationship between wt and mileage and positive relationship with drat.

- Only the variable wt has a statistical impact on mpg. Remember, to test a hypothesis in statistic, we use:
  - H0: No statistical impact
  - H3: The predictor has a meaningful impact on y
  - If the p value is lower than 0.05, it indicates the variable is statistically significant

- Adjusted R-squared: Variance explained by the model. In your model, the model explained 82 percent of the variance of y. R squared is always between 0 and 1. The

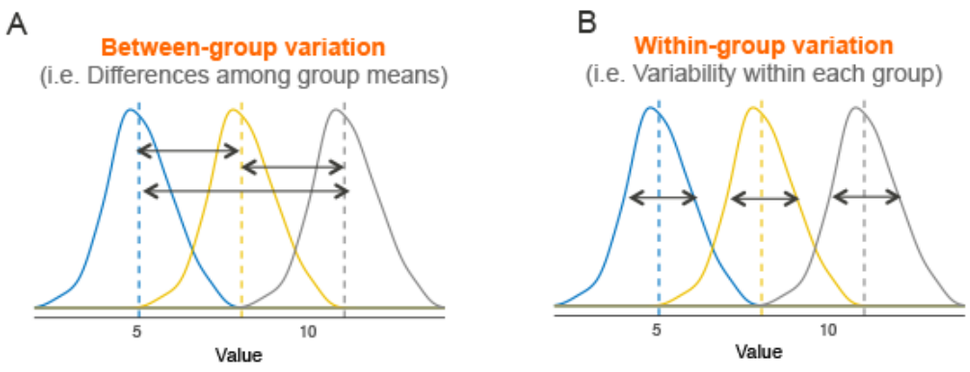
ANOVA

You can run the ANOVA test to estimate the effect of each feature on the variances with the anova() function.

```
anova(fit)
```

```
## Analysis of Variance Table
##
## Response: mpg
##           Df Sum Sq Mean Sq  F value   Pr(>F)
## disp       1 808.89  808.89 123.6185 2.23e-11 ***
## hp         1  33.67   33.67   5.1449 0.031854 *
## drat       1  30.15   30.15   4.6073 0.041340 *
## wt         1  70.51   70.51  10.7754 0.002933 **
## qsec       1  12.71   12.71   1.9422 0.175233
## Residuals 26 170.13    6.54
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The idea behind the ANOVA test is very simple: if the average variation between groups is large enough compared to the average variation within groups, then you could conclude that at least one group mean is not equal to the others.



ANOVA tells us if there are differences among group means, but not what the differences are. To find out which groups are statistically different from one another, you can perform a Tukey's Honestly Significant Difference (Tukey's HSD) post-hoc test for pairwise comparisons

Model performances

A more conventional way to estimate the model performance is to display the residual against different measures.

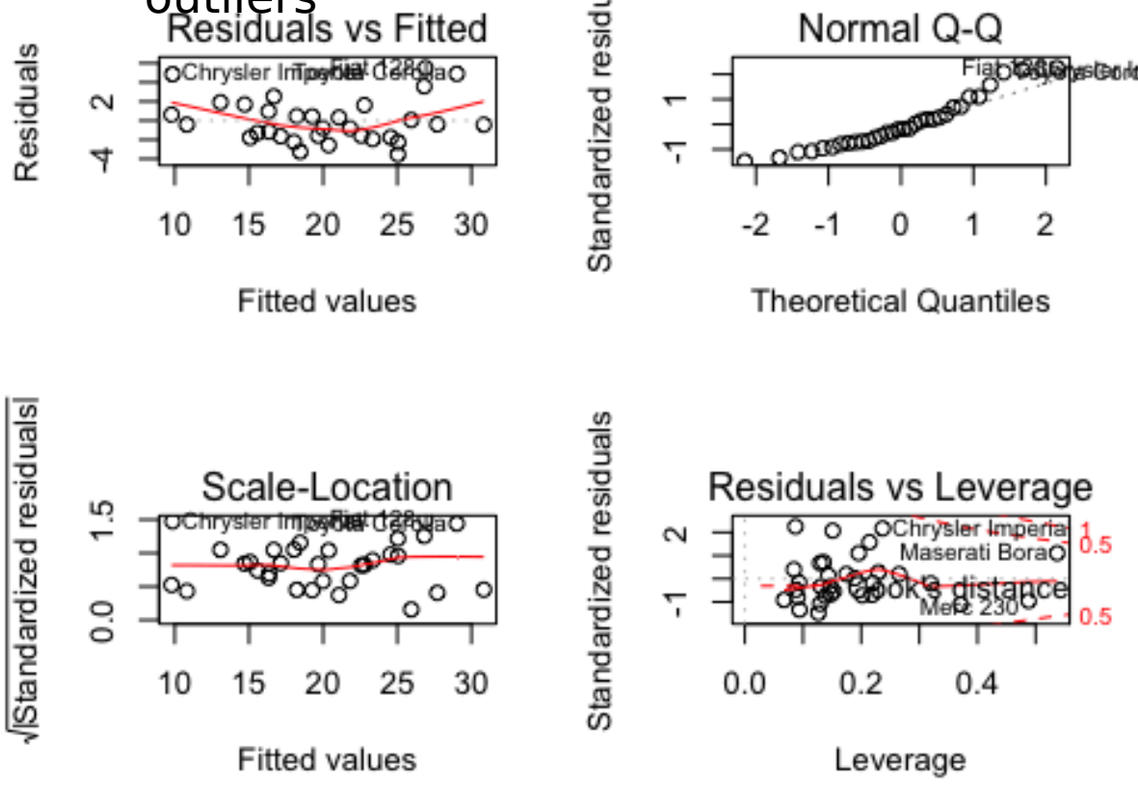
You can use the plot() function to show four graphs:

- Residuals vs Fitted values
- Normal Q-Q plot: Theoretical Quartile vs Standardized residuals
- Scale-Location: Fitted values vs Square roots of the standardised residuals
- Residuals vs Leverage: Leverage vs Standardized

```
par(mfrow=(2,2))
plot(fit)
```

Check for outliers

heteroscedasticity



Today’s program

Machine learning

- **Supervised learning**
  - **Simple Linear regression**
  - **Multiple Linear regression**
  - **Factor regression**
  - **Logistic regression**
  - **Decision Trees**
  - **Random Forests**
  - **SVM**
- **Unsupervised learning**
  - **Dimensionality reduction**
  - **Clustering**

Factor regression

In the last model estimation, you regress mpg on continuous variables only. It is straightforward to add factor variables to the model. You add the variable am to your model. It is important to be sure the variable is a factor level and not continuous.

```
df <- mtcars % > %
mutate(cyl = factor(cyl),
       vs = factor(vs),
       am = factor(am),
       gear = factor(gear),
       carb = factor(carb))
summary(lm(model, df))
```

R uses the first factor level as a base group. You need to compare the coefficients of the other group against the base group.

```
## Call:
## lm(formula = model, data = df)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -3.5087 -1.3584 -0.0948  0.7745  4.6251
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 23.87913   20.06582   1.190   0.2525
## cyl6        -2.64870    3.04089  -0.871   0.3975
## cyl8        -0.33616    7.15954  -0.047   0.9632
## disp         0.03555    0.03190   1.114   0.2827
## hp          -0.07051    0.03943  -1.788   0.0939 .
## drat         1.18283    2.48348   0.476   0.6407
## wt          -4.52978    2.53875  -1.784   0.0946 .
## qsec         0.36784    0.93540   0.393   0.6997
## vs1          1.93085    2.87126   0.672   0.5115
## am1          1.21212    3.21355   0.377   0.7113
## gear4        1.11435    3.79952   0.293   0.7733
## gear5        2.52840    3.73636   0.677   0.5089
## carb2       -0.97935    2.31797  -0.423   0.6787
## carb3        2.99964    4.29355   0.699   0.4955
## carb4        1.09142    4.44962   0.245   0.8096
## carb6        4.47757    6.38406   0.701   0.4938
## carb8        7.25041    8.36057   0.867   0.3995
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' '
##
## Residual standard error: 2.833 on 15 degrees of freedom
## Multiple R-squared:  0.8931, Adjusted R-squared:  0.779
## F-statistic:  7.83 on 16 and 15 DF,  p-value: 0.000124
```

Summary

Library	Objective	Function	Arguments
base	Compute a linear regression	lm()	formula, data
base	Summarize model	summarize()	fit
base	Exctract coefficients	lm()$coefficient	
base	Exctract residuals	lm()$residuals	
base	Exctract fitted value	lm()$fitted.values	

Today’s program

Machine learning

- **Supervised learning**
  - **Simple Linear regression**
  - **Multiple Linear regression**
  - **Factor regression**
  - **Logistic regression**
  - **Decision Trees**
  - **Random Forests**
  - **SVM**
- **Unsupervised learning**
  - **Dimensionality reduction**
  - **Clustering**

Logistic regression

Logistic regression is used to predict a class, i.e., a probability. Logistic regression can predict a binary outcome accurately.

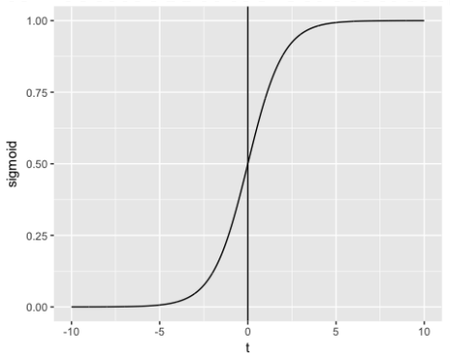
Imagine you want to predict whether a loan is denied/accepted based on many attributes. The logistic regression is of the form 0/1. y = 0 if a loan is rejected, y = 1 if accepted.

A logistic regression model differs from linear regression model in two ways.

•First of all, the logistic regression accepts only dichotomous (binary) input as a dependent variable (i.e., a vector of 0 and 1).

•Secondly, the outcome is measured by the following probabilistic link function called **sigmo** ped.:

$$\sigma(t) = \frac{1}{1 + exp(-t)}$$



Adult dataset

The objective is to predict whether the annual income in dollar of an individual will exceed 50.000. The dataset contains 46,033 observations and ten features:

```
Library(dplyr)
path <- ‘./datasets/adult.csv’
data_adult <-read.csv(path)
 glimpse(data_adult)
```

Output:

```
Observations: 48,842
Variables: 10
$ x              <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,...
$ age            <int> 25, 38, 28, 44, 18, 34, 29, 63, 24, 55, 65, 36, 26...
$ workclass      <fctr> Private, Private, Local-gov, Private, ?, Private,...
$ education      <fctr> 11th, HS-grad, Assoc-acdm, Some-college, Some-col...
$ educational.num <int> 7, 9, 12, 10, 10, 6, 9, 15, 10, 4, 9, 13, 9, 9, 9,...
$ marital.status  <fctr> Never-married, Married-civ-spouse, Married-civ-sp...
$ race           <fctr> Black, White, White, Black, White, White, Black, ...
$ gender         <fctr> Male, Male, Male, Male, Female, Male, Male, Male,...
$ hours.per.week  <int> 40, 50, 40, 40, 30, 30, 40, 32, 40, 10, 40, 40, 39...
$ income         <fctr> <=50K, <=50K, >50K, >50K, <=50K, <=50K, <=50K, >5...
```

Step 1) Check continuous variables

```
continuous <-select_if(data_adult, is.numeric)
summary(continuous)
## X age educational.num hours.per.week
## Min. : 1 Min. :17.00 Min. : 1.00 Min. : 1.00
## 1st Qu.:11509 1st Qu.:28.00 1st Qu.: 9.00 1st Qu.:40.00
## Median :23017 Median :37.00 Median :10.00 Median :40.00
## Mean :23017 Mean :38.56 Mean :10.13 Mean :40.95
## 3rd Qu.:34525 3rd Qu.:47.00 3rd Qu.:13.00 3rd Qu.:45.00
## Max. :46033 Max. :90.00 Max. :16.00 Max. :99.00
```

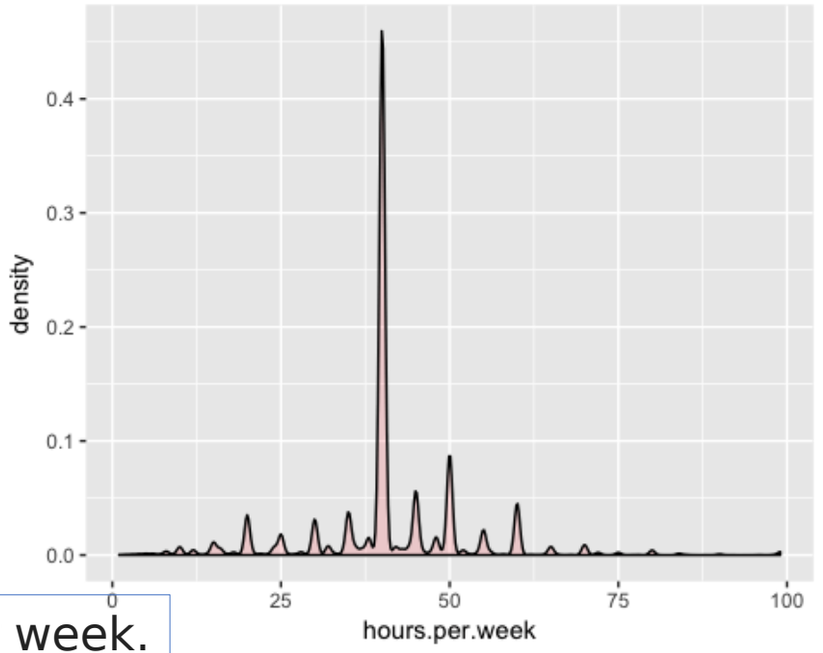
the data have totally different scales and hours.per.weeks has large outliers (.i.e. look at the last quartile and maximum value)

You can deal with it following two steps:

- 1: Plot the distribution of hours.per.week
- 2: Standardize the continuous variables

Plot the distribution of hours.per.week

```
# Histogram with kernel density curve
library(ggplot2)
ggplot(continuous, aes(x = hours.per.week)) +
geom_density(alpha = .2, fill = "#FF6666")
```



The variable has lots of outliers and not well-defined distribution. You can partially tackle this problem by deleting the top 0.01 percent of the

```
top_one_percent <-
quantile(data_adult$hours.per.week, .99)
top_one_percent
## 99%
## 80
```

98 percent of the population works under 80 hours per week.

You can drop the observations above this threshold. You use the filter from the

```
data_adult_drop <-data_adult %>%
filter(hours.per.week<top_one_percent)
dim(data_adult_drop)
## [1] 45537 10
```

Standardize the continuous variables

You can standardize each column to improve the performance because your data do not have the same scale. You can use the function mutate_if from the dplyr library.

```
data_adult_rescale <- data_adult_drop % > %
mutate_if(is.numeric, funs(as.numeric(scale(.))))
head(data_adult_rescale)
```

```
##           X         age        workclass    education educational.num
## 1 -1.732680 -1.02325949          Private         11th     -1.22106443
## 2 -1.732605 -0.03969284          Private      HS-grad     -0.43998868
## 3 -1.732530 -0.79628257        Local-gov   Assoc-acdm      0.73162494
## 4 -1.732455  0.41426100          Private Some-college     -0.04945081
## 5 -1.732379 -0.34232873          Private         10th     -1.61160231
## 6 -1.732304  1.85178149 Self-emp-not-inc  Prof-school      1.90323857
##       marital.status  race gender hours.per.week income
## 1      Never-married Black   Male    -0.03995944  <=50K
## 2 Married-civ-spouse White   Male     0.86863037  <=50K
## 3 Married-civ-spouse White   Male    -0.03995944   >50K
## 4 Married-civ-spouse Black   Male    -0.03995944   >50K
## 5      Never-married White   Male    -0.94854924  <=50K
## 6 Married-civ-spouse White   Male    -0.76683128   >50K
```

Step 2) Check factor variables

This step has two objectives:

•Check the level in each categorical column

•Define new levels

We will divide this step into three parts:

•Select the categorical columns

•Store the bar chart of each column in a list

•Print the graphs

Step 2) Check factor variables

```
# Select categorical column
factor <- data.frame(select_if(data_adult_rescale, is.factor))
ncol(factor)
## [1] 6
```

The dataset contains 6 categorical variable

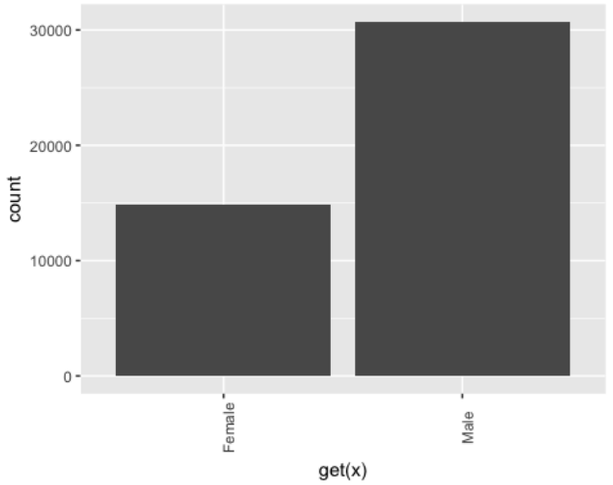
You want to plot a bar chart for each column in the data frame factor. It is more convenient to automatize the process, especially in situation there are lots of columns.

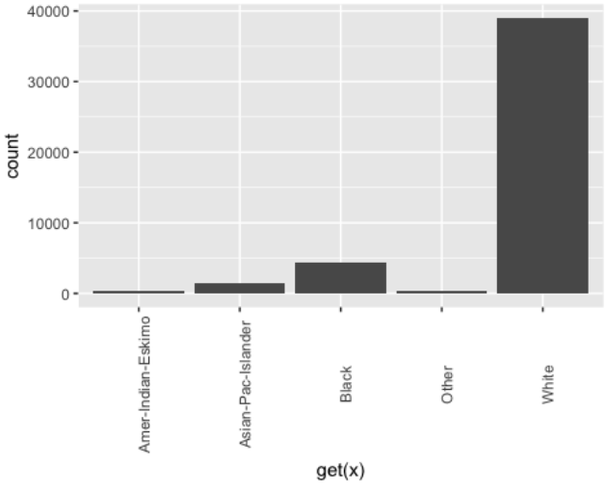
```
library(ggplot2)
# Create graph for each column
graph <- lapply(names(factor),
      function(x)
            ggplot(factor, aes(get(x))) + geom_bar() +
      theme(axis.text.x = element_text(angle = 90)))
```

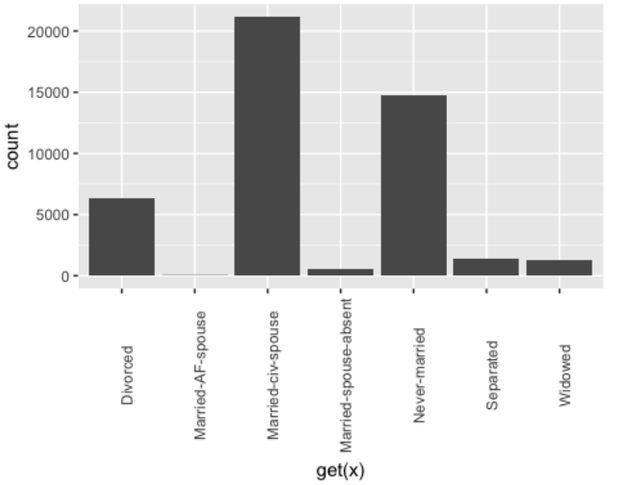
to pass a function in all the columns of the dataset. You store the output in a list

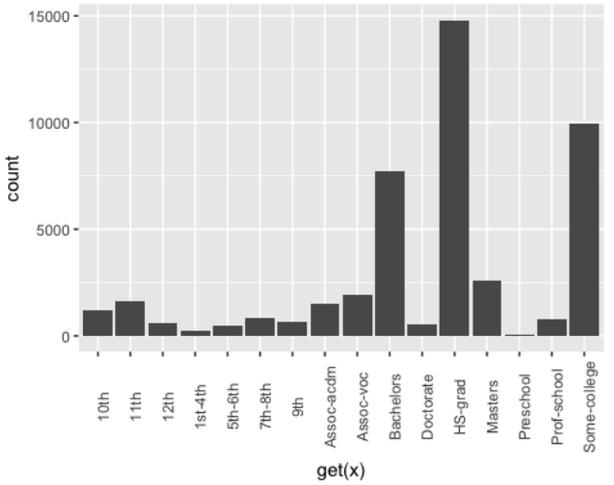
The function will be processed for each x. Here x is the columns

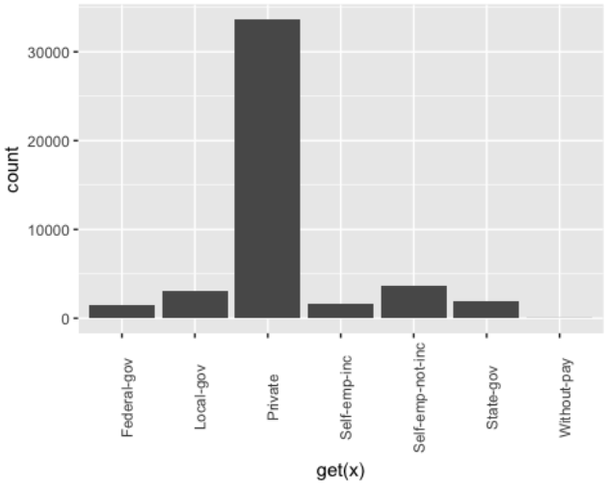
Step 2) Check factor variables

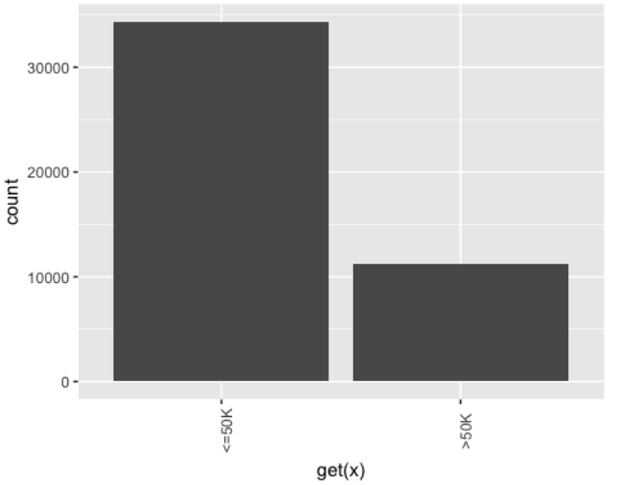




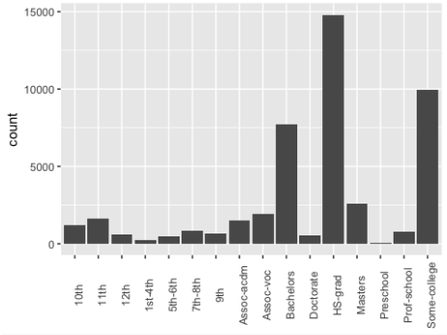








Step 3) Feature engineering



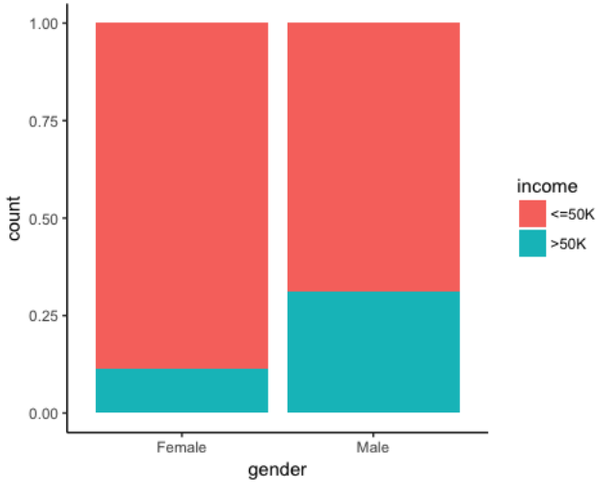
From the graph , you can see that the variable education has 16 levels. This is substantial, and some levels have a relatively low number of observations. If you want to improve the amount of information you can get from this variable, you can recast it into higher level. Namely, you create larger groups with similar level of education. For instance, low level of education will be converted in dropout. Higher levels of education will be changed to master.

```
recast_data <- data_adult_rescale % > %
      select(-X) % > %
      mutate(education = factor(ifelse(education == "Preschool" | education == "10th" | education == "11th" |
education == "12th" | education == "1st-4th" | education == "5th-6th" | education == "7th-8th" | education
== "9th", "dropout",
      ifelse(education == "HS-grad", "HighGrad", ifelse(education == "Some-college" | education == "Assoc-
acdm" | education == "Assoc-voc", "Community",
             ifelse(education == "Bachelors", "Bachelors", ifelse(edu                                    of-
school", "Master", "PhD")))))))
```

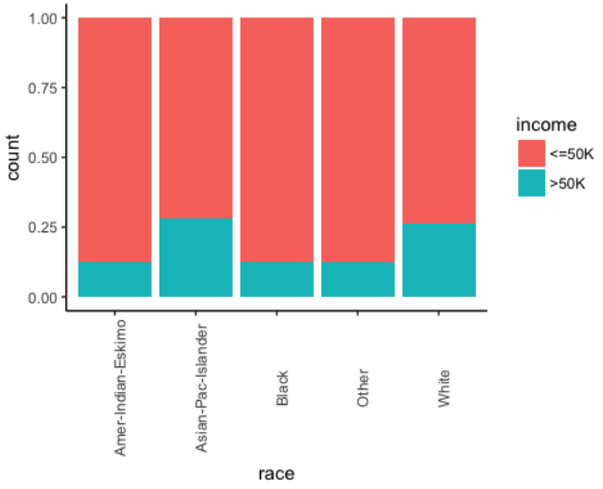
```
recast_data % > %
group_by(education) % > %
summarize(average_educ_year =
mean(educational.num), count = n()) % > %
arrange(average_educ_year)
```

```
## # A tibble: 6 x 3
## education average_educ_year count
## <fctr> <dbl> <int>
 ## 1 dropout -1.76147258 5712
## 2 HighGrad -0.43998868 14803
## 3 Community 0.09561361 13407
## 4 Bachelors 1.12216282 7720
## 5 Master 1.60337381 3338
## 6 PhD 2.29377644 557
```

Step 4) Summary Statistic

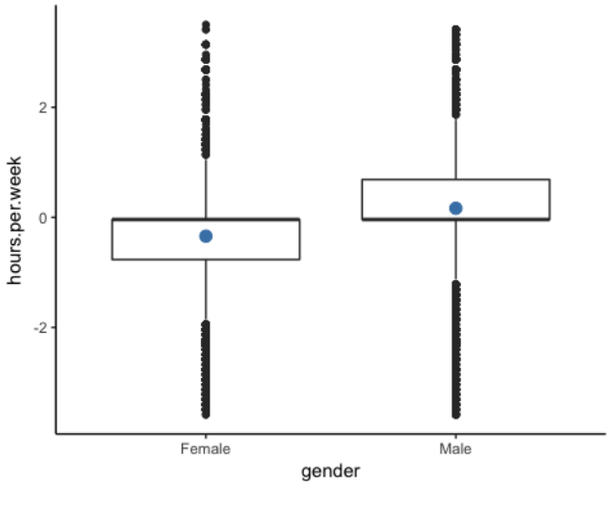



```
# Plot gender income
ggplot(recast_data, aes(x =
gender, fill = income)) +
geom_bar(position = "fill") +
theme_classic()
```

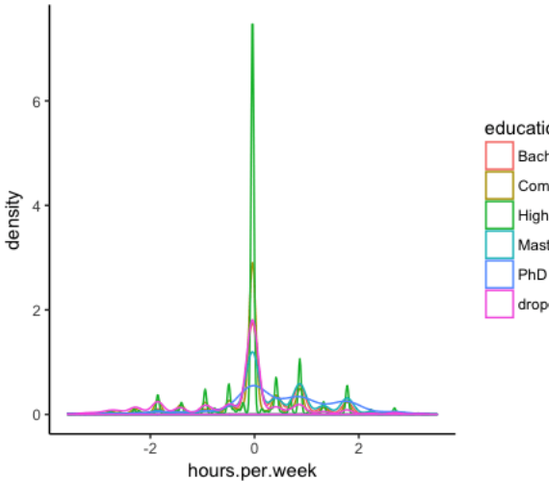



```
# Plot origin income
ggplot(recast_data,
aes(x = race, fill =
income)) +
geom_bar(position =
"fill") + theme_classic
+ theme(axis.text.x =
element_text(angle =
90))
```

```
# box plot gender working
time
ggplot(recast_data, aes(x
= gender, y =
hours.per.week)) +
geom_boxplot() +
stat_summary(fun.y =
mean, geom = "point",
size = 3, color =
"steelblue") +
theme_classic()
```




```
# Plot distribution
working time by
education
ggplot(recast_data,
aes(x =
hours.per.week)) +
geom_density(aes(col
or = education), alpha
= 0.5) +
theme_classic()
```



The box plot confirms that the distribution of working time fits different groups. In the box plot, both genders do not have homogeneous

Step 5) Train/test set

Any supervised machine learning task require to split the data between a train set and a test set.

```
set.seed(1234)
create_train_test <- function(data, size = 0.8, train
= TRUE) { n_row = nrow(data) total_row = size *
n_row train_sample <- 1: total_row
if (train == TRUE) {
return (data[train_sample, ])
} else {
return (data[-train_sample, ])
}
}
data_train <- create_train_test(recast_data, 0.8,
train = TRUE) data_test <-
create_train_test(recast_data, 0.8, train = FALSE)

dim(data_train)
## [1] 36429 9
dim(data_test)
## [1] 9108 9
```

Step 6) Build the model

glm(formula, data=data, family=linkfunction())
Argument:

- formula: Equation used to fit the model
- data: dataset used
- Family:
  - binomial: (link = "logit")
  - gaussian: (link = "identity")
  - Gamma: (link = "inverse")
  - inverse.gaussian: (link = "1/mu^2")
  - poisson: (link = "log")
  - quasi: (link = "identity", variance = "constant")
  - quasibinomial: (link = "logit")
  - quasipoisson: (link = "log")

```
formula <- income~.
logit <- glm(formula, data = data_train, family = 'binomial’)
summary(logit)
```

In statistics, the **generalized linear model** (**GLM**) is a flexible generalization of ordinary linear regression that allows for response variables that have error distribution models other than a normal distribution. The GLM generalizes linear regression by allowing the linear model to be related to the response variable via a *link function* and by allowing the magnitude of the variance of each measurement to be a function of its predicted value.
Additional reading:
https://www.r-bloggers.com/generalized-linear-models-understanding-the-link-function/

Create the model to fit

Fit a logistic model

Print the summary of the model

Output

The performance of a logistic regression is evaluated with specific key metrics.

•AIC (Akaike Information Criteria): This is the equivalent of **R2** in logistic regression. It measures the fit when a penalty is applied to the number of parameters. Smaller **AIC** values indicate the model is closer to the truth.

•Null deviance: Fits the model only with the intercept. The degree of freedom is n-1. We can interpret it as a Chi-square value (fitted value different from the actual value hypothesis testing).

•Residual Deviance: Model with all the variables. It is also interpreted as a Chi-square hypothesis testing.

•Number of Fisher Scoring iterations: Number of iterations before converging

```
##
## Call:
## glm(formula = formula, family = "binomial", data = data_train)
## ## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -2.6456  -0.5858  -0.2609  -0.0651   3.1982
##
## Coefficients:
##                            Estimate Std. Error z value Pr(>|z|)
## (Intercept)                 0.07882    0.21726   0.363  0.71675
## age                         0.41119    0.01857  22.146  < 2e-16 ***
## workclassLocal-gov         -0.64018    0.09396  -6.813 9.54e-12 ***
## workclassPrivate           -0.53542    0.07886  -6.789 1.13e-11 ***
## workclassSelf-emp-inc      -0.07733    0.10350  -0.747  0.45499
## workclassSelf-emp-not-inc -1.09052    0.09140 -11.931  < 2e-16 ***
## workclassState-gov         -0.80562    0.10617  -7.588 3.25e-14 ***
## workclassWithout-pay       -1.09765    0.86787  -1.265  0.20596
## educationCommunity         -0.44436    0.08267  -5.375 7.66e-08 ***
## educationHighGrad          -0.67613    0.11827  -5.717 1.08e-08 ***
## educationMaster             0.35651    0.06780   5.258 1.46e-07 ***
## educationPhD                0.46995    0.15772   2.980  0.00289 **
## educationdropout           -1.04974    0.21280  -4.933 8.10e-07 ***
## educational.num             0.56908    0.07063   8.057 7.84e-16 ***
## marital.statusNot_married -2.50346    0.05113 -48.966  < 2e-16 ***
## marital.statusSeparated   -2.16177    0.05425 -39.846  < 2e-16 ***
## marital.statusWidow       -2.22707    0.12522 -17.785  < 2e-16 ***
## raceAsian-Pac-Islander      0.08359    0.20344   0.411  0.68117
## raceBlack                   0.07188    0.19330   0.372  0.71001
## raceOther                   0.01370    0.27695   0.049  0.96054
## raceWhite                   0.34830    0.18441   1.889  0.05894 .
## genderMale                  0.08596    0.04289   2.004  0.04506 *
## hours.per.week              0.41942    0.01748  23.998  < 2e-16 ***
## ---## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' '
## ## (Dispersion parameter for binomial family taken to be 1)
## ##     Null deviance: 40601  on 36428  degrees of freedom
## Residual deviance: 27041  on 36406  degrees of freedom
## AIC: 27087
##
## Number of Fisher Scoring iterations: 6
```

Output

The output of the glm() function is stored in a list. The code below shows all the items available in the logit variable we constructed to evaluate the logistic regression.
The list is very long, print only the first three elements

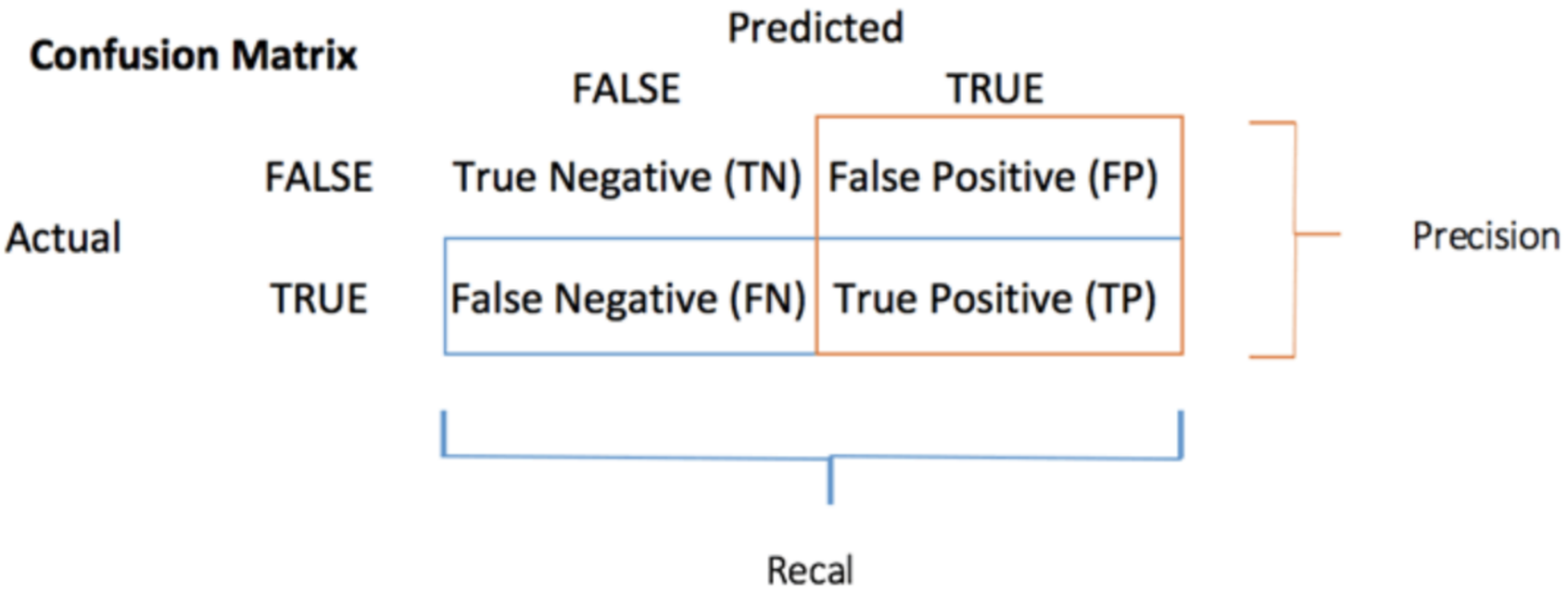
```
lapply(logit, class)[1:3]
## $coefficients
## [1] "numeric"
##
## $residuals
## [1] "numeric"
##
## $fitted.values
## [1] "numeric"
```

Each value can be extracted with the $ sign follow by the name of the metrics. For instance, you stored the model as logit. To extract the AIC criteria, you use:

```
logit$aic
## [1] 27086.65
```

Step 7) Assess the performance of the model

The **confusion matrix** is a better choice to evaluate the classification performance compared with the different metrics you saw before. The general idea is to count the number of times True instances are classified are False.



Step 7) Assess the performance of the model

To compute the confusion matrix, you first need to have a set of predictions so that they can be compared to the actual targets.

```
predict <- predict(logit, data_test, type =
'response')
# confusion matrix
table_mat <- table(data_test$income, predict >
0.5)

table_mat
##
## FALSE TRUE
## <=50K 6310 495
## >50K 1074 1229

accuracy_Test <- sum(diag(table_mat)) /
sum(table_mat)
## [1] 0.8277339
```

Compute the prediction on the test set.

Compute the confusion matrix.

The model appears to suffer from one problem, it overestimates the number of false negatives. This is called the **accuracy test paradox**. We stated that the accuracy is the ratio of correct predictions to the total number of cases. We can have relatively high accuracy but a useless model. It happens when there is a dominant class. If you look back at the confusion matrix, you can see most of the cases are classified as true negative.

Step 7) Assess the performance of the model

In such situation, it is preferable to have a more concise metric. We can look at:

•Precision=TP/(TP+FP)

•Recall=TP/(TP+FN)

Precision looks at the accuracy of the positive prediction. **Recall** is the ratio of positive instances that are correctly detected by the classifier;

```
precision <-
function(matrix) {
# True positive
tp <- matrix[2, 2]
 # false positive
fp <- matrix[1, 2]
return (tp / (tp + fp)) }

prec <- precision(table_mat)
## [1] 0.712877
```

You can create the F_1 score based on the precision and recall. The F_1 is a harmonic mean of these two metrics, meaning it gives more weight to the lower values.

```
f1 <- 2 * ((prec * rec) / (prec + rec)) f1
## [1] 0.6103799
```

```
recall <- function(matrix) {
# true positive
tp <- matrix[2, 2]
# false positive
fn <- matrix[2, 1]
return (tp / (tp + fn)) }

rec <- recall(table_mat)
## [2] 0.5336518
```

Precision vs Recall tradeoff

It is impossible to have both a high precision and high recall.
If we increase the precision, the correct individual will be better predicted, but we would miss lots of them (lower recall). In some situation, we prefer higher precision than recall. There is a concave relationship between precision and recall.

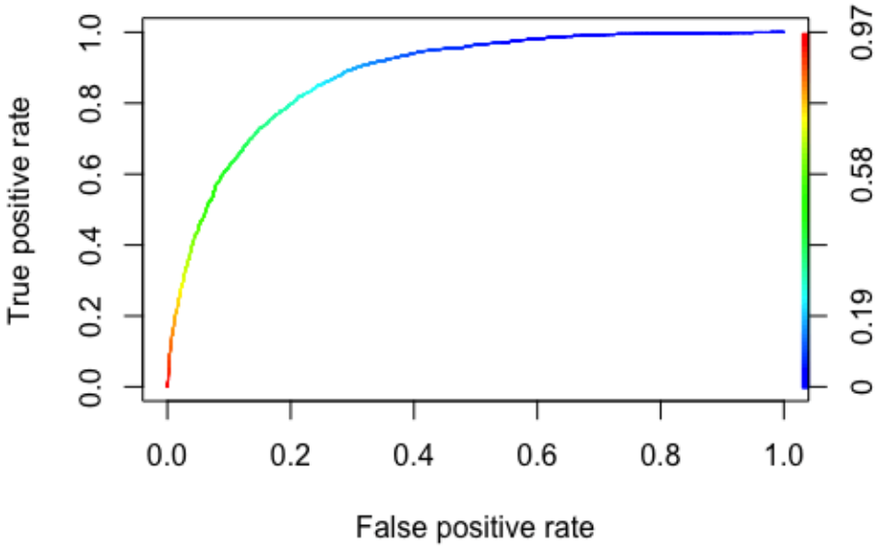
Imagine, you need to predict if a patient has a disease. **You want to be as precise as possible.**

If you need to detect potential fraudulent people in the street through facial recognition, it would be better to catch many people labeled as fraudulent even though the **precision is low**. The police will be able to release the non-fraudulent individual.

The ROC curve

The **Receiver Operating Characteristic** curve is another common tool used with binary classification.
It is very similar to the precision/recall curve, but instead of plotting precision versus recall, the ROC curve shows the true positive rate (i.e., recall) against the false positive rate.
The false positive rate is the ratio of negative instances that are incorrectly classified as positive.
It is equal to one minus the true negative rate.
The true negative rate is also called **specificity**.
Hence the ROC curve plots **sensitivity** (recall) versus 1-s

```
library(ROCR)
ROCRpred <- prediction(predict, data_test$income)
ROCRperf <- performance(ROCRpred, 'tpr', 'fpr')
plot(ROCRperf, colorize = TRUE, text.adj = c(-0.2, 1.7))
```



Summary

Package	Objective	function	argument
-	Create train/test dataset	create_train_set()	data, size, train
glm	Train a Generalized Linear Model	glm()	formula, data, family*
glm	Summarize the model	summary()	fitted model
base	Make prediction	predict()	fitted model, dataset, type = 'response'
base	Create a confusion matrix	table()	y, predict()
base	Create accuracy score	sum(diag(table())/sum(table()	
ROCR	Create ROC : Step 1 Create prediction	prediction()	predict(), y
ROCR	Create ROC : Step 2 Create performance	performance()	prediction(), 'tpr', 'fpr'
ROCR	Create ROC : Step 3 Plot graph	plot()	performance()

Today’s program

Machine learning

- **Supervised learning**
  - **Simple Linear regression**
  - **Multiple Linear regression**
  - **Factor regression**
  - **Logistic regression**
  - **Decision Trees**
  - **Random Forests**
  - **SVM**
- **Unsupervised learning**
  - **Dimensionality reduction**
  - **Clustering**

Decision Tree

- Decision trees are versatile Machine Learning algorithm that can perform both classification and regression tasks.
- They are very powerful algorithms, capable of fitting complex datasets.
- Besides, decision trees are fundamental components of random forests, which are among the most potent Machine Learning algorithms available today.

Step 1) Import the data

We are going to use the titanic dataset. The purpose of this dataset is to predict which people are more likely to survive after the collision with the iceberg. The dataset contains 13 variables and 1309 observations. The dataset is ordered by the variable X

```
set.seed(678)
path <- './datasets/titanic_data.csv'
titanic <-read.csv(path)
head(titanic)
```

```
##   X pclass survived                                             name    sex
## 1 1      1        1                    Allen, Miss. Elisabeth Walton female
## 2 2      1        1                   Allison, Master. Hudson Trevor   male
## 3 3      1        0                     Allison, Miss. Helen Loraine female
## 4 4      1        0             Allison, Mr. Hudson Joshua Creighton   male
## 5 5      1        0 Allison, Mrs. Hudson J C (Bessie Waldo Daniels) female
## 6 6      1        1                               Anderson, Mr. Harry   male
##       age sibsp parch ticket     fare   cabin embarked
## 1 29.0000     0     0  24160 211.3375      B5        S
## 2  0.9167     1     2 113781 151.5500 C22 C26        S
## 3  2.0000     1     2 113781 151.5500 C22 C26        S
## 4 30.0000     1     2 113781 151.5500 C22 C26        S
## 5 25.0000     1     2 113781 151.5500 C22 C26        S
## 6 48.0000     0     0  19952  26.5500     E12        S
##                         home.dest
## 1                     St Louis, MO
## 2 Montreal, PQ / Chesterville, ON
## 3 Montreal, PQ / Chesterville, ON
## 4 Montreal, PQ / Chesterville, ON
## 5 Montreal, PQ / Chesterville, ON
## 6                     New York, NY
```

You can notice the data is not shuffled. This is a big issue! When you will split your data between a train set and test set, you will select **only** the passenger from class 1 and 2 (No passenger from class 3 are in the top 80 percent of the observations), which means the algorithm will never see the features of

Step 1) Import the data

```
shuffle_index <- sample(1:nrow(titanic))
 head(shuffle_index)

## [1] 288 874 1078 633 887 992

titanic <- titanic[shuffle_index, ]
head(titanic)
```

Generate a random list of index from 1 to 1309 (i.e. the maximum number of rows)

You will use this index to shuffle the titanic dataset.

```
##         X pclass survived
## 288   288      1        0
## 874   874      3        0
## 1078 1078      3        1
## 633   633      3        0
## 887   887      3        1
## 992   992      3        1
##                                                                 name    sex age
## 288                                        Sutton, Mr. Frederick   male  61
## 874                         Humblen, Mr. Adolf Mathias Nicolai Olsen   male  42
## 1078                                    O'Driscoll, Miss. Bridget female  NA
## 633  Andersson, Mrs. Anders Johan (Alfrida Konstantia Brogren) female  39
## 887                                          Jermyn, Miss. Annie female  NA
## 992                                             Mamee, Mr. Hanna   male  NA
##      sibsp parch ticket    fare cabin embarked           home.dest## 288      0     0
36963 32.3208   D50        S     Haddenfield, NJ
## 874      0     0 348121  7.6500 F G63        S
## 1078     0     0  14311  7.7500             Q
## 633      1     5 347082 31.2750             S Sweden Winnipeg, MN
## 887      0     0  14313  7.7500             Q
## 992      0     0   2677  7.2292             C
```

Step 2) Clean the dataset

The structure of the data shows some variables have NA's. Data clean up to be done as follows

•Drop variables home.dest,cabin, name, X and ticket
•Create factor variables for pclass and survived
•Drop the NA

```
library(dplyr)
# Drop variables clean
titanic <- titanic % > %
select(-c(home.dest, cabin, name, X, ticket)) % >
%
#Convert to factor level
mutate(pclass = factor(pclass, levels = c(1, 2, 3),
labels = c('Upper', 'Middle', 'Lower')),
survived = factor(survived, levels = c(0, 1), labels
= c('No', 'Yes'))) % > %
na.omit()
glimpse(clean_titanic)
```

Drop unnecessary variables

Add label to the variable pclass. 1 becomes Upper, 2 becomes MIddle and 3 becomes lower

Add label to the variable survived. 1 Becomes No and 2 becomes Yes

Remove the NA observations

Step 2) Clean the dataset

```
## Observations: 1,045
## Variables: 8
## $ pclass   <fctr> Upper, Lower, Lower, Upper, Middle, Upper, Middle, U...
## $ survived <fctr> No, No, No, Yes, No, Yes, Yes, No, No, No, No, No, Y...
## $ sex      <fctr> male, male, female, female, male, male, female, male...
## $ age      <dbl> 61.0, 42.0, 39.0, 49.0, 29.0, 37.0, 20.0, 54.0, 2.0, ...
## $ sibsp    <int> 0, 0, 1, 0, 0, 1, 0, 0, 4, 0, 0, 1, 1, 0, 0, 0, 1, 1,...
## $ parch    <int> 0, 0, 5, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 2, 0, 4, 0,...
## $ fare     <dbl> 32.3208, 7.6500, 31.2750, 25.9292, 10.5000, 52.5542, ...
## $ embarked <fctr> S, S, S, S, S, S, S, S, S, C, S, S, S, Q, C, S, S, C...
```

Step 3) Create train/test set

Before you train your model, you need to perform two steps:

•Install rpart.plot from the console

```
install.packages("rpart.plot")
```

•Create a train and test set: You train the model on the train set and test the prediction on the test set (i.e. unseen data)

The common practice is to split the data 80/20, 80 percent of the data serves to train the model, and 20 percent to make predictions. You need to create two separate data frames. You don't want to touch the test set until you finish building your model. You can create a function name create_train_test() that takes three arguments.

Step 3) Create train/test set

```
create_train_test(df, size = 0.8, train = TRUE)
arguments:
-df: Dataset used to train the model.
-size: Size of the split. By default, 0.8. Numerical value
-train: If set to `TRUE`, the function creates the train set, otherwise the test set. Def
ault value sets to `TRUE`. Boolean value.You need to add a Boolean parameter because R do
es not allow to return two data frames simultaneously.
```

```
create_train_test & lt;- function(data, size = 0.8, train = TRUE) {
    n_row = nrow(data)
    total_row = size * n_row
    train_sample & lt; - 1: total_row
    if (train == TRUE) {
        return (data[train_sample, ])
    } else {
        return (data[-train_sample, ])
    }
}
```

Add the arguments in the functio

Count number of rows in the data

Return the nth row to construct the train

Select the first row to the nth rows

If condition sets to true, return the train set, else the test set.

Step 3) Create train/test set

```
data_train <- create_train_test(clean_titanic, 0.8,
train = TRUE) data_test <-
create_train_test(clean_titanic, 0.8, train = FALSE)
dim(data_train)
## [1] 836 8
dim(data_test)
## [1] 209 8

prop.table(table(data_train$survived))
## No Yes
## 0.5944976 0.4055024

prop.table(table(data_test$survived))
## No Yes
## 0.5789474 0.4210526
```

The train dataset has 1046 rows while the test dataset has 262 rows.

to verify if the randomization process is correct.

In both dataset, the amount of survivors is the same, about 40 percent.

Step 4) Build the model

You are ready to build the model. The syntax for Rpart() function is:

```
rpart(formula, data=, method='')
arguments:
- formula: The function to predict
- data: Specifies the data frame- method:
- "class" for a classification tree
- "anova" for a regression tree
```

You use the class method because you predict a class.

```
library(rpart)
library(rpart.plot)
fit <- rpart(survived~., data = data_train, method = 'class')
rpart.plot(fit, extra = 106
```

Function to fit the model

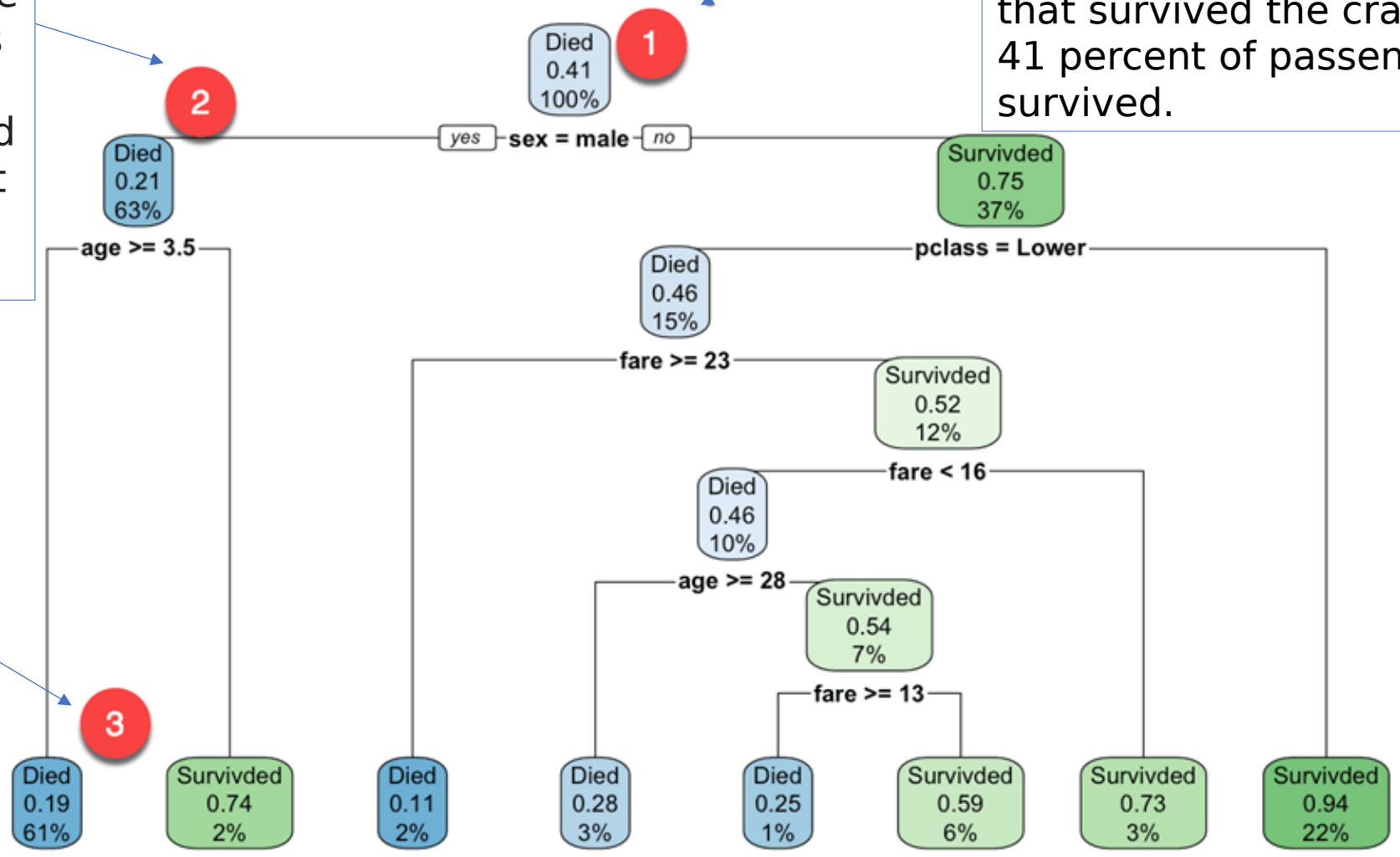
Plot the tree.

Step 4) Build the model

This node asks whether the gender of the passenger is male. If yes, then you go down to the root's left child node (depth 2). 63 percent are males with a survival probability of 21 percent.

At the top, it is the overall probability of survival. It shows the proportion of passenger that survived the crash. 41 percent of passenger survived.

In the second node, you ask if the male passenger is above 3.5 years old. If yes, then the chance of survival is 19 percent.



Step 5) Make a prediction

You can predict your test dataset. To make a prediction, you can use the predict() function. The basic syntax of predict for decision trees is:

```
predict(fitted_model, df, type = 'class')
arguments:
- fitted_model: This is the object stored after model estimation.
- df: Data frame used to make the prediction
- type: Type of prediction
    - 'class': for classification
    - 'prob': to compute the probability of each class
    - 'vector': Predict the mean response at the node level
```

You want to predict which passengers are more likely to survive after the collision from the test set. It means, you will know among those 209 passengers, which one will survive or not.

```
predict_unseen <-predict(fit, data_test, type = 'class')
```

Predict the class (0/1) of the test se

Step 5) Make a prediction

esting the passenger who didn't make it and those who did.

```
table_mat <- table(data_test$survived,
predict_unseen) table_mat
```

Create a table to count how many passengers are classified as survivors and passed away compare to the correct classification

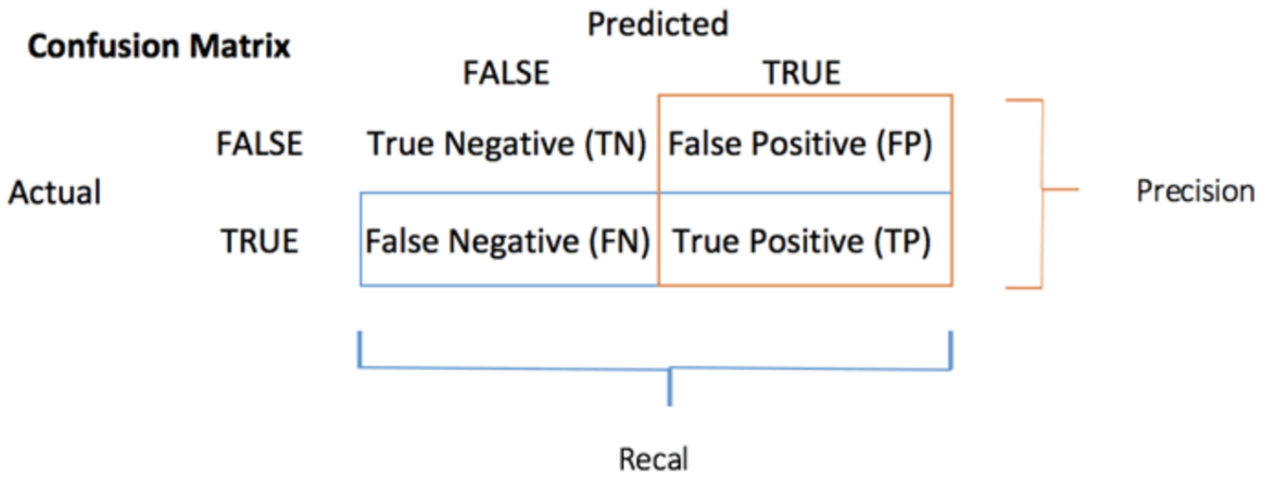
```
## predict_unseen
##       No Yes
## No 106 15
 ## Yes 30 58
```

The model correctly predicted 106 dead passengers but classified 15 survivors as dead. By analogy, the model misclassified 30 passengers as survivors while they turned out to be dead.

Step 6) Measure performance

You can compute an accuracy measure for classification task with the **confusion matrix**:

The **confusion matrix** is a better choice to evaluate the classification performance. The general idea is to count the number of times True instances are classified are False.



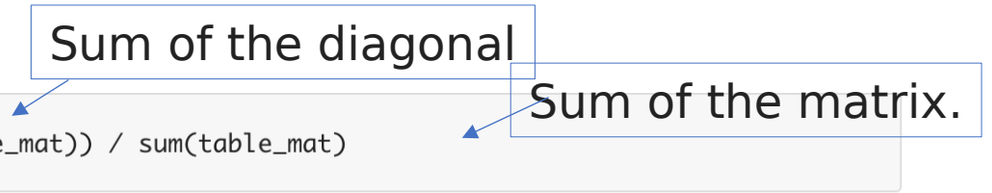
Each row in a confusion matrix represents an actual target, while each column represents a predicted target. The first row of this matrix considers dead passengers (the False class): 106 were correctly classified as dead (**True negative**), while the remaining one was wrongly classified as a survivor (**False positive**). The second row considers the survivors, the positive class were 58 (**True positive**), while the **True negative** was 30.

Step 6) Measure performance

You can compute the **accuracy test** from the confusion matrix:

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

It is the proportion of true positive and true negative over the sum of the matrix. With R, you can code as follow:




```
accuracy_Test <- sum(diag(table_mat)) / sum(table_mat)
```

You can print the accuracy of the test set:

```
print(paste('Accuracy for test', accuracy_Test))
```

Output:

```
## [1] "Accuracy for test 0.784688995215311"
```

Step 7) Tune the hyper-parameters

Decision tree has various parameters that control aspects of the fit. In rpart library, you can control the parameters using the rpart.control() function. In the following code, you introduce the parameters you will tune.

```
rpart.control(minsplit = 20, minbucket = round(minsplit/3), maxdepth = 30)
Arguments:
-minsplit: Set the minimum number of observations in the node before the algorithm perfor
m a split
-minbucket:  Set the minimum number of observations in the final note i.e. the leaf
-maxdepth: Set the maximum depth of any node of the final tree. The root node is treated
a depth 0
```

We will proceed as follow:

- Construct function to return accuracy
- Tune the maximum depth
- Tune the minimum number of sample a node must have before it can split
- Tune the minimum number of sample a leaf node must have

Step 7) Tune the hyper-parameters

You can write a function to display the accuracy. You simply wrap the code you used b

```
accuracy_tune <- function(fit) {
    predict_unseen <- predict(fit, data_test, type = 'class')
    table_mat <- table(data_test$survived, predict_unseen)
    accuracy_Test <- sum(diag(table_mat)) / sum(table_mat)
    accuracy_Test
}
```

You can try to tune the parameters and see if you can improve the model over the default value.

As a reminder, you need to get an accuracy higher than 0.78

```
control <- rpart.control(minsplit = 4,
    minbucket = round(5 / 3),
    maxdepth = 3,
    cp = 0)
tune_fit <- rpart(survived~., data = data_train, method = 'class', control = control)
accuracy_tune(tune_fit)
```

```
## [1] 0.7990431
```

minsplit = 4
minbucket= round(5/3)
maxdepth = 3
cp=0

Summary

Library	Objective	function	class	parameters
rpart	Train classification trees	rpart()	class	formula, df, method
rpart	Train regression tree	rpart()	anova	formula, df, method
rpart	Plot the trees	rpart.plot()		fitted model
base	predict	predict()	class	fitted model, type
base	predict	predict()	prob	fitted model, type
base	predict	predict()	vector	fitted model, type
rpart	Control parameters	rpart.control()		minsplit	Set the minimum number of observations in the node before the algorithm perform a split
				minbucket	Set the minimum number of observations in the final note i.e. the leaf
				maxdepth	Set the maximum depth of any node of the final tree. The root node is treated a depth 0
rpart	Train model with control parameter	rpart()		formula, df, method, control	

Today’s program

Machine learning

- **Supervised learning**
  - **Simple Linear regression**
  - **Multiple Linear regression**
  - **Factor regression**
  - **Logistic regression**
  - **Decision Trees**
  - **Random Forests**
  - **SVM**

- **Unsupervised learning**
  - **Dimensionality reduction**
  - **Clustering**

Random Forest

- Random forests are based on a simple idea: 'the wisdom of the crowd'. Aggregate of the results of multiple predictors gives a better prediction than the best individual predictor. A group of predictors is called an **ensemble**. Thus, this technique is called **Ensemble Learning**.

- We learned how to use **Decision trees** to make a binary prediction. To improve our technique, we can train a group of **Decision Tree classifiers**, each on a different random subset of the train set.

- To make a prediction, we just obtain the predictions of all individuals trees, then predict the class that gets the most votes. This technique is called **Random Forest**.

Step 1) Import the data

We will use the same train and test dataset created earlier

```
library(dplyr)
data_train <- read.csv(    " . /datasets/train.csv")
glimpse(data_train)
 data_test <- read.csv(" . /datasets/test.csv  ")
glimpse(data_test)
```

Step 2) Train the model

One way to evaluate the performance of a model is to train it on a number of different smaller datasets and evaluate them over the other smaller testing set. This is called the **F-fold cross-validation** feature. R has a function to randomly split number of datasets of almost the same size. For example, if k=9, the model is evaluated over the nine folder and tested on the remaining test set. This process is repeated until all the subsets have been evaluated. This technique is widely used for model selection, especially when the model has parameters to tune.

Now that we have a way to evaluate our model, we need to figure out how to choose the parameters that generalized best the data.

Random forest chooses a random subset of features and builds many Decision Trees. The model averages out all the predictions of the Decisions trees.

Random forest has some parameters that can be changed to improve the generalization of the prediction. You will use the function RandomForest() to train the model.

Step 2) Train the model

RandomForest(formula, ntree=n, mtry=FALSE, maxnodes = NULL)
Arguments:

- Formula: Formula of the fitted model
- ntree: number of trees in the forest
- mtry: Number of candidates draw to feed the algorithm. By default, it is the square of the number of columns.
- maxnodes: Set the maximum amount of terminal nodes in the forest
- mportance=TRUE: Whether independent variables importance in the random forest be assessed

Tuning a model is very tedious work. There are lot of combination possible between the parameters. You don't necessarily have the time to try all of them. A good alternative is to let the machine find the best combination for you.

There are two methods available:

•Random Search
•Grid Search

Grid Search definition

The grid search method is simple, the model will be evaluated over all the combination you pass in the function, using cross-validation.
For instance, you want to try the model with 10, 20, 30 number of trees and each tree will be tested over a number of mtry equals to 1, 2, 3, 4, 5. Then the machine will test 15 different models:

```
   .mtry ntrees
1      1     10
2      2     10
3      3     10
4      4     10
5      5     10
6      1     20
7      2     20
8      3     20
9      4     20
10     5     20
11     1     30
12     2     30
13     3     30
14     4     30
15     5     30
```

The algorithm will evaluate:

```
RandomForest(formula, ntree=10, mtry=1)
RandomForest(formula, ntree=10, mtry=2)
RandomForest(formula, ntree=10, mtry=3)
RandomForest(formula, ntree=20, mtry=2)
...
```

Each time, the random forest experiments with a cross-validation. One shortcoming of the grid search is the number of experimentations. It can become very easily explosive when the number of combination is high. To overcome this issue, you can use the random search

Random Search definition

The big difference between random search and grid search is, random search will not evaluate all the combination of hyperparameter in the searching space. Instead, it will randomly choose combination at every iteration. The advantage is it lower the computational cost.

You will proceed as follow to construct and evaluate the model:
•Evaluate the model with the default setting
•Find the best number of mtry
•Find the best number of maxnodes
•Find the best number of ntrees
•Evaluate the model on the test dataset

Before you begin with the parameters exploration, you need to install two libraries.

caret: R machine learning library
e1071: R machine learning library

```
library(randomForest)
library(caret)
library(e1071)
```

Default setting

K-fold cross validation is controlled by the trainControl() function

```
# Define the control
trControl <- trainControl(method = "cv", number = 10,
search = "grid")
```

The method used to resample the dataset.

Number of folders to create

Use the search grid method.

Let's try the build the model with the default values.

```
set.seed(1234)

 # Run the model
rf_default <- train(survived~., data =
data_train, method = "rf", metric =
"Accuracy", trControl = trControl)

# Print the results
print(rf_default)
```

Output

```
## Random Forest
##
## 836 samples
##   7 predictor
##   2 classes: 'No', 'Yes'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 753, 752, 753, 752, 752, 752, ...
## Resampling results across tuning parameters:
##
##   mtry  Accuracy   Kappa
##    2    0.7919248  0.5536486
##    6    0.7811245  0.5391611
##   10    0.7572002  0.4939620
##
## Accuracy was used to select the optimal model using  the largest value.
## The final value used for the model was mtry = 2.
```

The algorithm uses 500 trees and tested three different values of mtry: 2, 6, 10.

The final value used for the model was mtry = 2 with an accuracy of 0.78.

Let's try to get a higher score.

Step 2) Search best mtry

You can test the model with values of mtry from 1 to 10

```
set.seed(1234)
tuneGrid <- expand.grid(.mtry = c(1: 10))
rf_mtry <- train(survived~.,
    data = data_train,
    method = "rf",
    metric = "Accuracy",
    tuneGrid = tuneGrid,
    trControl = trControl,
    importance = TRUE,
    nodesize = 14,
    ntree = 300)
print(rf_mtry)
```

```
## Random Forest
##
## 836 samples
##   7 predictor
##   2 classes: 'No', 'Yes'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 753, 752, 753, 752, 752, 752, ...
## Resampling results across tuning parameters:
##
##   mtry  Accuracy   Kappa
##    1    0.7572576  0.4647368
##    2    0.7979346  0.5662364
##    3    0.8075158  0.5884815
##    4    0.8110729  0.5970664
##    5    0.8074727  0.5900030
##    6    0.8099111  0.5949342
##    7    0.8050918  0.5866415
##    8    0.8050918  0.5855399
##    9    0.8050631  0.5855035
##   10    0.7978916  0.5707336
##
## Accuracy was used to select the optimal model using  the largest valu
## The final value used for the model was mtry = 4.
```

Step 3) Search the best maxnodes

```
store_maxnode <- list()
tuneGrid <- expand.grid(.mtry = best_mtry)
for (maxnodes in c(5: 15)) {
    set.seed(1234)
    rf_maxnode <- train(survived~.,
        data = data_train,
        method = "rf",
        metric = "Accuracy",
        tuneGrid = tuneGrid,
        trControl = trControl,
        importance = TRUE,
        nodesize = 14,
        maxnodes = maxnodes,
        ntree = 300)
    current_iteration <- toString(maxnodes)
    store_maxnode[[current_iteration]] <- rf_maxnode
}
results_mtry <- resamples(store_maxnode)
summary(results_mtry)
```

The results of the model will be stored in this list

Use the best value of mtry

Compute the model with values of maxnodes starting from 15 to 25.

For each iteration, maxnodes is equal to the current value of maxnodes. i.e 15, 16, 17, ...

Store as a string variable the value of maxnode.

Save the result of the model in the list.

Arrange the results of the model

Print the summary of all the combination.

Step 3) Search the best maxnodes

```
## Call:
## summary.resamples(object = results_mtry)
##
## Models: 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15
## Number of resamples: 10
##
## Accuracy
##         Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
## 5  0.6785714 0.7529762 0.7903758 0.7799771 0.8168388 0.8433735    0
## 6  0.6904762 0.7648810 0.7784710 0.7811962 0.8125000 0.8313253    0
## 7  0.6904762 0.7619048 0.7738095 0.7788009 0.8102410 0.8333333    0
## 8  0.6904762 0.7627295 0.7844234 0.7847820 0.8184524 0.8433735    0
## 9  0.7261905 0.7747418 0.8083764 0.7955250 0.8258749 0.8333333    0
## 10 0.6904762 0.7837780 0.7904475 0.7895869 0.8214286 0.8433735    0
## 11 0.7023810 0.7791523 0.8024240 0.7943775 0.8184524 0.8433735    0
## 12 0.7380952 0.7910929 0.8144005 0.8051205 0.8288511 0.8452381    0
## 13 0.7142857 0.8005952 0.8192771 0.8075158 0.8403614 0.8452381    0
## 14 0.7380952 0.7941050 0.8203528 0.8098967 0.8403614 0.8452381    0
## 15 0.7142857 0.8000215 0.8203528 0.8075301 0.8378873 0.8554217    0
##
## Kappa
##         Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
## 5  0.3297872 0.4640436 0.5459706 0.5270773 0.6068751 0.6717371    0
## 6  0.3576471 0.4981484 0.5248805 0.5366310 0.6031287 0.6480921    0
## 7  0.3576471 0.4927448 0.5192771 0.5297159 0.5996437 0.6508314    0
## 8  0.3576471 0.4848320 0.5408159 0.5427127 0.6200253 0.6717371    0
## 9  0.4236277 0.5074421 0.5859472 0.5601687 0.6228626 0.6480921    0
## 10 0.3576471 0.5255698 0.5527057 0.5497490 0.6204819 0.6717371    0
## 11 0.3794326 0.5235007 0.5783191 0.5600467 0.6126720 0.6717371    0
## 12 0.4460432 0.5480930 0.5999072 0.5808134 0.6296780 0.6717371    0
## 13 0.4014252 0.5725752 0.6087279 0.5875305 0.6576219 0.6678832    0
## 14 0.4460432 0.5585005 0.6117973 0.5911995 0.6590982 0.6717371    0
## 15 0.4014252 0.5689401 0.6117973 0.5867010 0.6507194 0.6955990    0
```

The last value of maxnode has the highest accuracy. You can try with higher values to see if you can get a higher score.

Reapply the previous function with maxnodes in c(20:30)

The highest accuracy score is obtained with a value of maxnode equals to 22.

Step 4) Search the best ntrees

Now that you have the best value of mtry and maxnode, you can tune the number of trees. The metho
is exactly the same as maxnode.

```
store_maxtrees <- list()
for (ntree in c(250, 300, 350, 400, 450, 500, 550, 600, 800, 1000, 2000)) {
    set.seed(5678)
    rf_maxtrees <- train(survived~.,
        data = data_train,
        method = "rf",
        metric = "Accuracy",
        tuneGrid = tuneGrid,
        trControl = trControl,
        importance = TRUE,
        nodesize = 14,
        maxnodes = 24,
        ntree = ntree)
    key <- toString(ntree)
    store_maxtrees[[key]] <- rf_maxtrees
}
results_tree <- resamples(store_maxtrees)
summary(results_tree)
```

```
## Call:
## summary.resamples(object = results_tree)
##
## Models: 250, 300, 350, 400, 450, 500, 550, 600, 800, 1000, 2000
## Number of resamples: 10
##
## Accuracy
##           Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
## 250  0.7380952 0.7976190 0.8083764 0.8087010 0.8292683 0.8674699    0
## 300  0.7500000 0.7886905 0.8024240 0.8027199 0.8203397 0.8452381    0
## 350  0.7500000 0.7886905 0.8024240 0.8027056 0.8277623 0.8452381    0
## 400  0.7500000 0.7886905 0.8083764 0.8051009 0.8292683 0.8452381    0
## 450  0.7500000 0.7886905 0.8024240 0.8039104 0.8292683 0.8452381    0
## 500  0.7619048 0.7886905 0.8024240 0.8062914 0.8292683 0.8571429    0
## 550  0.7619048 0.7886905 0.8083764 0.8099062 0.8323171 0.8571429    0
## 600  0.7619048 0.7886905 0.8083764 0.8099205 0.8323171 0.8674699    0
## 800  0.7619048 0.7976190 0.8083764 0.8110820 0.8292683 0.8674699    0
## 1000 0.7619048 0.7976190 0.8121510 0.8086723 0.8303571 0.8452381    0
## 2000 0.7619048 0.7886905 0.8121510 0.8086723 0.8333333 0.8452381    0
##
## Kappa
##           Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
## 250  0.4061697 0.5667400 0.5836013 0.5856103 0.6335363 0.7196807    0
## 300  0.4302326 0.5449376 0.5780349 0.5723307 0.6130767 0.6710843    0
## 350  0.4302326 0.5449376 0.5780349 0.5723185 0.6291592 0.6710843    0
## 400  0.4302326 0.5482030 0.5836013 0.5774782 0.6335363 0.6710843    0
## 450  0.4302326 0.5449376 0.5780349 0.5750587 0.6335363 0.6710843    0
## 500  0.4601542 0.5449376 0.5780349 0.5804340 0.6335363 0.6949153    0
## 550  0.4601542 0.5482030 0.5857118 0.5884507 0.6396872 0.6949153    0
## 600  0.4601542 0.5482030 0.5857118 0.5884374 0.6396872 0.7196807    0
## 800  0.4601542 0.5667400 0.5836013 0.5910088 0.6335363 0.7196807    0
## 1000 0.4601542 0.5667400 0.5961590 0.5857446 0.6343666 0.6678832    0
## 2000 0.4601542 0.5482030 0.5961590 0.5862151 0.6440678 0.6656337    0
```

Train the model

You have your final model. You can train the random forest with the following parameters:
•ntree =800: 800 trees will be trained
•mtry=4: 4 features is chosen for each iteration
•maxnodes = 24: Maximum 24 nodes in the terminal nodes (leaves)

```
fit_rf <- train(survived~.,
    data_train,
    method = "rf",
    metric = "Accuracy",
    tuneGrid = tuneGrid,
    trControl = trControl,
    importance = TRUE,
    nodesize = 14,
    ntree = 800,
    maxnodes = 24)
```

Step 5) Evaluate the model

The library caret has a function to make prediction.

```
predict(model, newdata= df)
argument
- `model`: Define the model evaluated before.
- `newdata`: Define the dataset to make prediction
```

```
prediction <-predict(fit_rf, data_test)
```

You can use the prediction to compute the confusion matrix and see the accuracy score

```
confusionMatrix(prediction, data_test$survived)
```

Output

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  No Yes
##        No  110  32
##        Yes  11  56
##
##               Accuracy : 0.7943
##                 95% CI : (0.733, 0.8469)
##    No Information Rate : 0.5789
##    P-Value [Acc > NIR] : 3.959e-11
##
##                  Kappa : 0.5638
##  Mcnemar's Test P-Value : 0.002289
##
##             Sensitivity : 0.9091
##             Specificity : 0.6364
##          Pos Pred Value : 0.7746
##          Neg Pred Value : 0.8358
##              Prevalence : 0.5789
##          Detection Rate : 0.5263
##    Detection Prevalence : 0.6794
##       Balanced Accuracy : 0.7727
##
##        'Positive' Class : No
##
```

You have an accuracy of 0.7943 percent, which is higher than the default value

Step 6) Visualize Result

Lastly, you can look at the feature importance with the function varImp().

```
varImpPlot(fit_rf)
```

Output:

```
varImp(fit_rf)
## rf variable importance
##
##               Importance
## sexmale          100.000
## age               28.014
## pclassMiddle      27.016
## fare              21.557
## pclassUpper       16.324
## sibsp             11.246
## parch              5.522
## embarkedC          4.908
## embarkedQ          1.420
## embarkedS          0.000
```

It seems that the most important features are the sex and age. That is not surprising because the important features are likely to appear closer to the root of the tree, while less important features will often appear closed to the leaves.

Summary

Library	Objective	function	parameter
randomForest	Create a Random forest	RandomForest()	formula, ntree=n, mtry=FALSE, maxnodes = NULL
caret	Create K folder cross validation	trainControl()	method = "cv", number = n, search ="grid"
caret	Train a Random Forest	train()	formula, df, method = "rf", metric= "Accuracy", trControl = trainControl(), tuneGrid = NULL
caret	Predict out of sample	predict	model, newdata= df
caret	Confusion Matrix and Statistics	confusionMatrix()	model, y test
caret	variable importance	cvarImp()	model

Today’s program

Machine learning

- **Supervised learning**
  - **Simple Linear regression**
  - **Multiple Linear regression**
  - **Factor regression**
  - **Logistic regression**
  - **Decision Trees**
  - **Random Forests**
  - **SVM**

- **Unsupervised learning**
  - **Dimensionality reduction**
  - **Clustering**

Support Vector Machines

In machine learning, **support vector machines** are supervised learning models with associated learning algorithms that analyze data used for classification and regression analysis. However, they are mostly used in classification problems.

The **advantages** of support vector machines are:

•Effective in high dimensional spaces.

•Still effective in cases where number of dimensions is greater than the number of samples.

•Uses a subset of training points in the decision function (called support vectors), so it is also memory efficient.

•Versatile: different Kernel functions can be specified for the decision function. Common kernels are provided, but it is also possible to specify custom kernels.

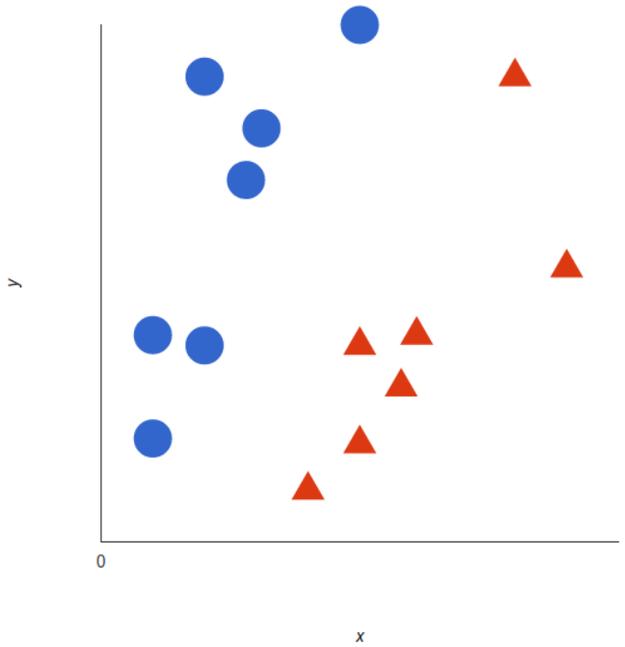
The **disadvantages** of support vector machines include:

•If the number of features is much greater than the number of samples, avoid over-fitting in choosing Kernel functions and regularization term is crucial.

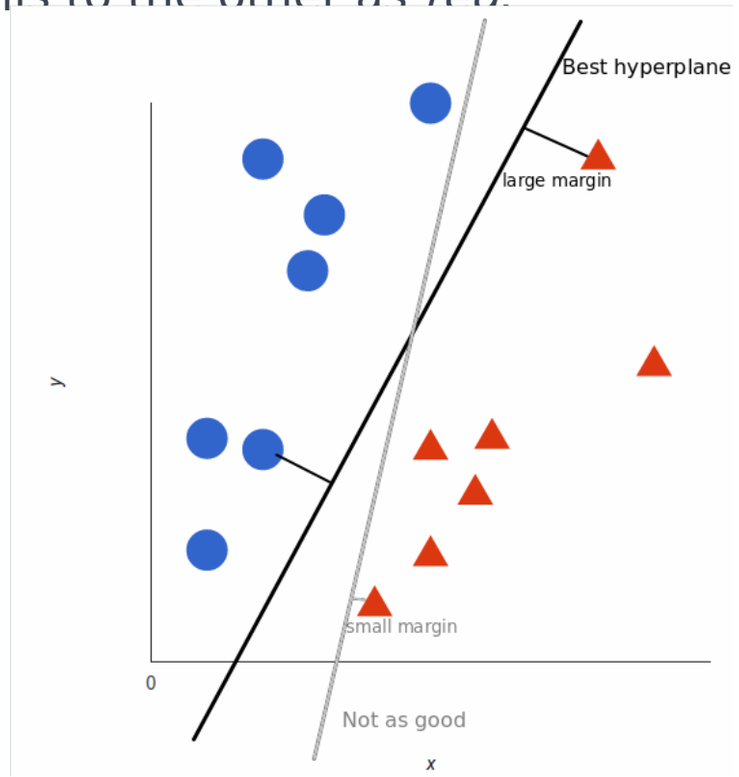
•SVMs do not directly provide probability estimates, these are calculated using an expensive five-fold cross-validation.

SVM - **Linear Data**

Let's imagine we have two tags: *red* and *blue*, and our data has two features: x and y. We want a classifier that, given a pair of (x,y) coordinates, outputs if it's either *red* or *blue*. We plot our already [illegible] a plane:

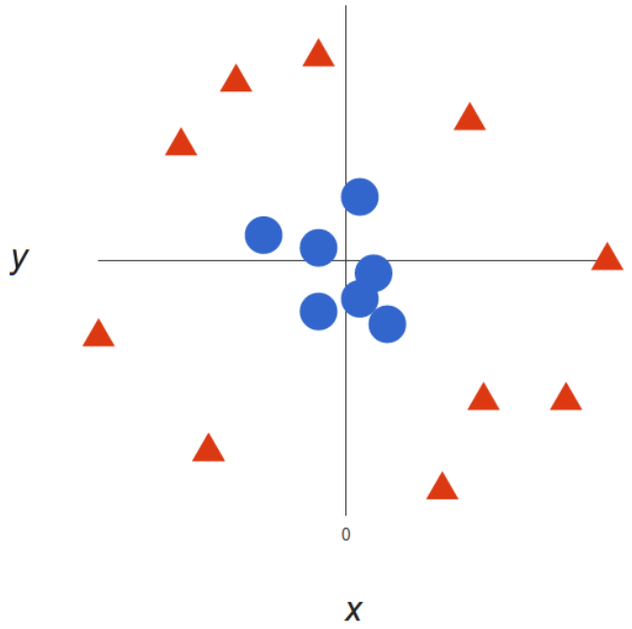


A support vector machine takes these data points and outputs the hyperplane (which in two dimensions it's simply a line) that best separates the tags. This line is the **decision boundary**: anything that falls to one side of it we will classify as *blue*, and anything that falls to the other as *red*.



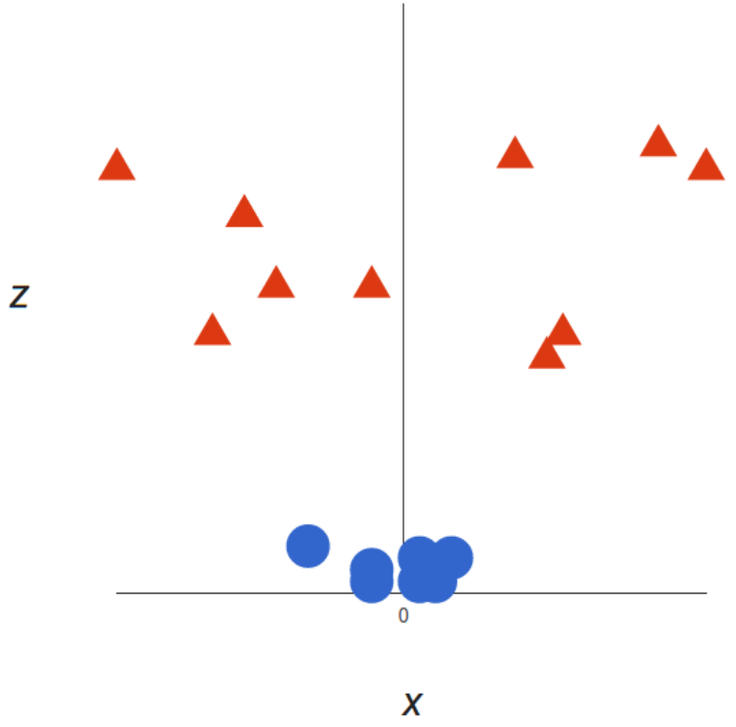
But, what exactly is the best hyperplane? For SVM, it's the one that maximizes the margins from both tags. In other words: the hyperplane (remember it's a line in this case) whose distance to the nearest element of each

SVM - **Non-Linear Data**



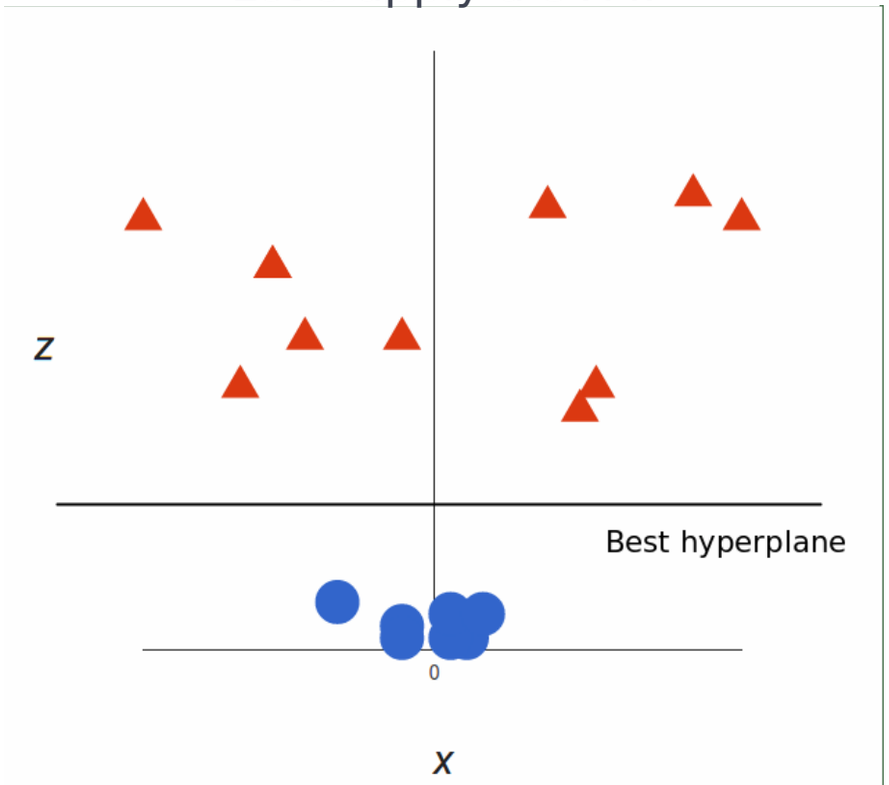
It’s pretty clear that there’s not a linear decision boundary (a single straight line that separates both tags). However, the vectors are very clearly segregated, and it looks as though it should be easy to separate

We will add a third dimension. Up until now, we had two dimensions: xx and yy. We create a new z dimension, and we rule that it be calculated a certain way that is convenient for us: $z=x^2+y^2z=x^2+y^2$ (you’ll notice that’s the equation for a circle).
This will give us a three-dimensional space. Taking a slice of th



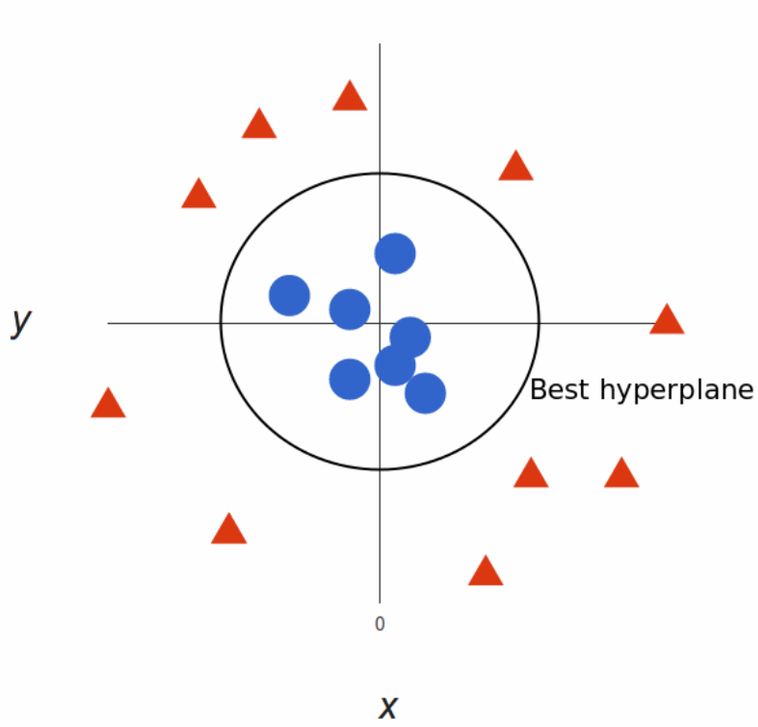
SVM - **Non-Linear Data**

Let’s apply a SVM:



Note that since we are in three dimensions now, the hyperplane is a plane parallel to the x axis at a certain z (let’s say z=1).

What’s left is mapping it back to two dimension



Kernel

In the above example, we found a way to classify nonlinear data by cleverly mapping our space to a higher dimension. However, it turns out that calculating this transformation can get pretty **computationally expensive**: **there can be a lot of new dimensions**, each one of them possibly involving a complicated calculation. Doing this for every vector in the dataset can be a lot of work, so it'd be great if we could find a cheaper solution.
Here's a trick: SVM doesn't need the actual vectors to work its magic, it actually can get by only with the dot products between them. This means that we can sidestep the expensive calculations of the ne

- Imagine the new $z = x^2 + y^2$ ant:
- Figure out what the dot product in that space looks like:

$$a \cdot b = x_a \cdot x_b + y_a \cdot y_b + z_a \cdot z_b = x_a \cdot x_b + y_a \cdot y_b + (x_a^2 + y_a^2) \cdot (x_b^2 + y_b^2)$$

- Tell SVM to do its thing, but using the new dot product — we call this a *kernel function.*

Common types of kernels used to separate non-linear data are **polynomial kernels, radial basis kernels, and linear kernels** (which are the same as support vector classifiers). Simply, these kernels transform our data to pass a linear hyperplane and thus classify our data.

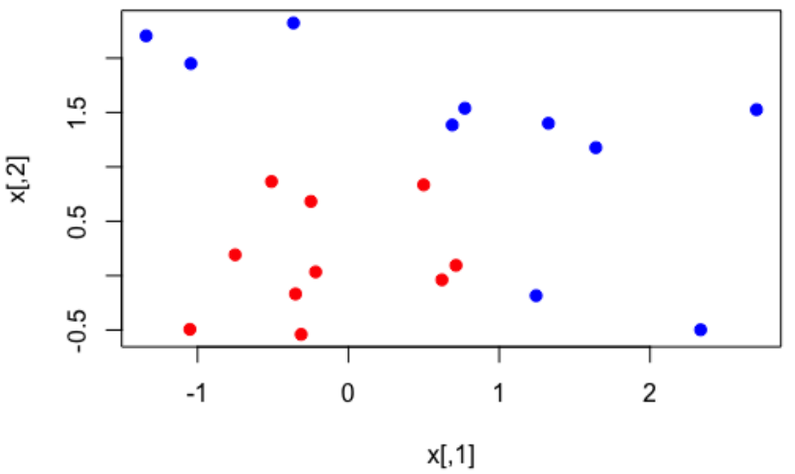
Support Vector Machines in R

Linear SVM Classifier

```
#generate some data in 2 dimensions, and make them a little
separated
set.seed(10111)
x = matrix(rnorm(40), 20, 2)
y = rep(c(-1, 1), c(10, 10))
x[y == 1,] = x[y == 1,] + 1
plot(x, col = y + 3, pch
```

make a matrix *x*, normally distributed with 20 observations in 2 classes on 2 variables

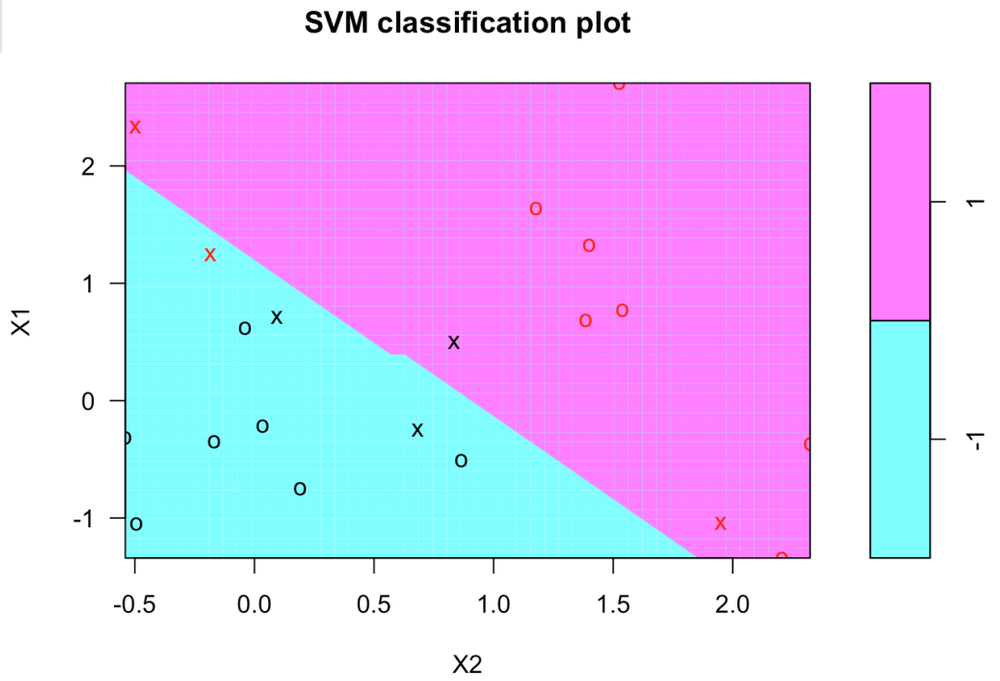
make a *y* variable, which is going to be either -1 or 1, with 10 in each class



Support Vector Machines in R

Linear SVM Classifier

```
library(e1071)
dat = data.frame(x, y = as.factor(y))
svmfit = svm(y ~ ., data = dat, kernel = "linear", cost = 10,
scale = FALSE) print(svmfit)
plot(svmfit, dat)
```



As you can see in the plot, the points in the boxes are close to the decision boundary and are instrumental in determining that boundary.

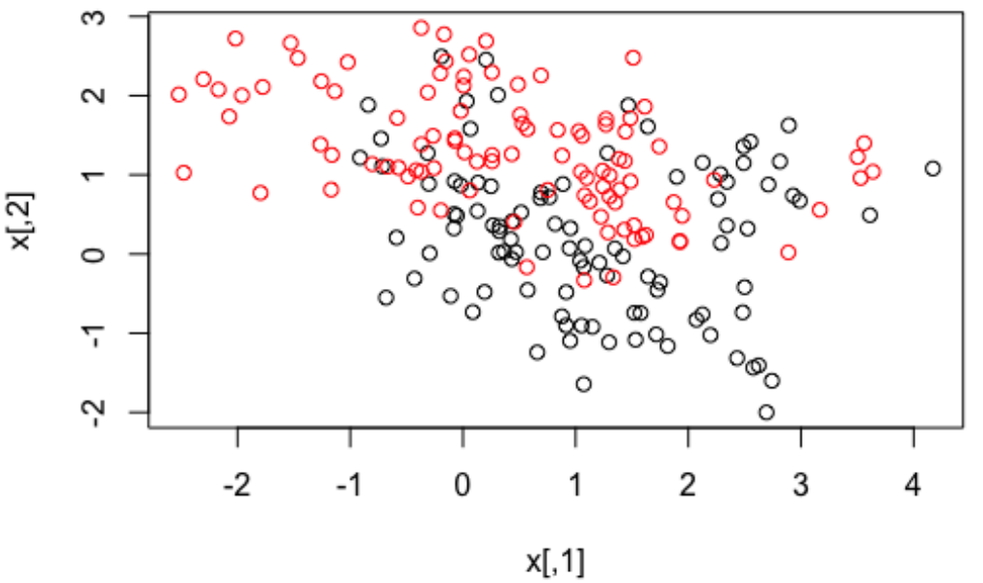
Support Vector Machines in R

Non-Linear SVM Classifier

Let's use the example from the textbook **Elements of Statistical Learning**, which has a canonical example in 2 dimensions where the decision boundary is non-linear.
The training data are x and y.

```
load(file = "ESL.mixture.rda")
names(ESL.mixture)
rm(x, y)
attach(ESL.mixture)
plot(x, col = y + 1)
```

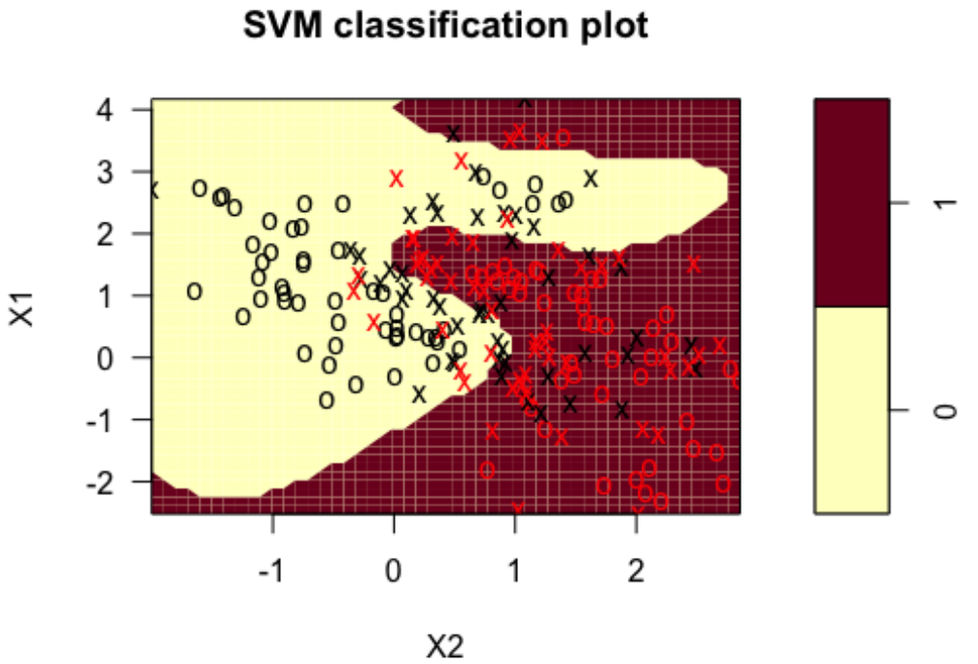
we've already created and x and y for the previous example



Support Vector Machines in R

Non-Linear SVM Classifier

```
dat = data.frame(y = factor(y), x)
fit = svm(factor(y) ~ ., data = dat, scale = FALSE, kernel = "radial", cost = 5)
plot(fit, dat)
```



Today’s program

Machine learning

- **Supervised learning**
  - **Simple Linear regression**
  - **Multiple Linear regression**
  - **Factor regression**
  - **Logistic regression**
  - **Decision Trees**
  - **Random Forests**
  - **SVM**
- **Unsupervised learning**
  - **Dimensionality reduction**
  - **Clustering**

What is Dimensionality Reduction?

Many Machine Learning problems involve thousands of features, having such a large number of features bring along many problems, the most important ones are:

- *Makes the training extremely slow*

- *Makes it difficult to find a good solution*

This is known as the ***curse of dimensionality*** and the Dimensionality Reduction is the process of reducing the number of features to the most relevant ones in simple terms.

What is Dimensionality Reduction?

Most Dimensionality Reduction applications are used for:

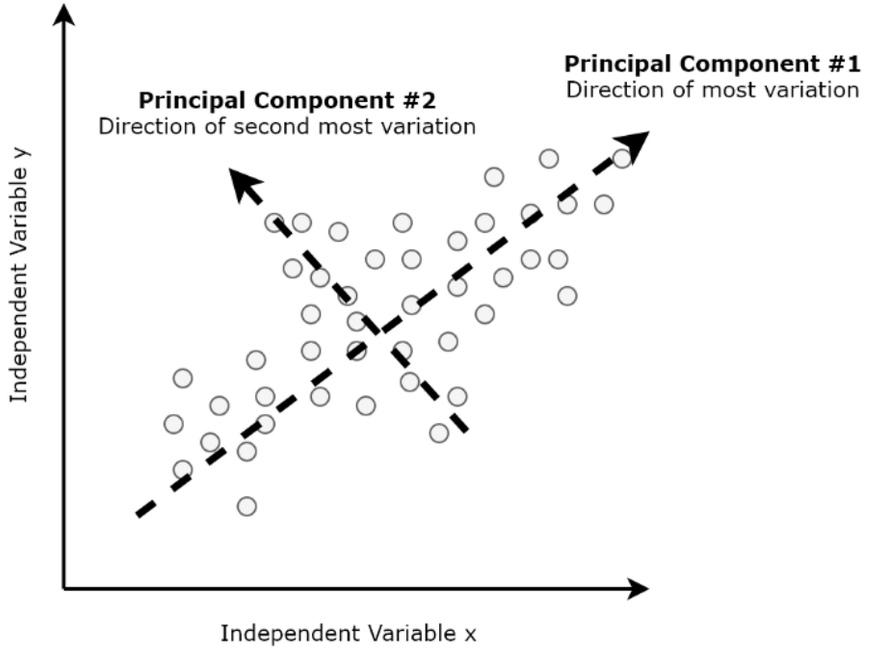
- **Data Compression**
- **Noise Reduction**
- **Data Classification**
- **Data Visualization**

One of the most important aspects of Dimensionality reduction, it is ***Data Visualization.*** Having to drop the dimensionality down to two or three, make it possible to visualize the data on a 2d or 3d plot, meaning important insights can be gained by analysing these patterns in terms of clusters and much more.

PCA (Principal Component Analysis)

One of the most known dimensionality reduction “unsupervised” algorithm is PCA(Principal Component Analysis).

> This works by identifying the hyperplane which lies closest to the data and then projects the data on that hyperplane while retaining most of the variation in the data set.



Principal Components
The axis that explains the maximum amount of variance in the training set is called the ***Principal Components***.
The axis orthogonal to this axis is called the **second principal component**. As we go for higher dimensions, PCA would find a third component orthogonal to the other two components and so on, for visualization purposes we always stick to 2 or maximum 3 principal components.

t-SNE (T-distributed stochastic neighbour embedding)

t-SNE was created in 2008 by **(**Laurens van der Maaten and Geoffrey Hinton) for dimensionality reduction that is particularly well suited for the visualization of high-dimensional datasets.

> **(t-SNE)** takes a high dimensional data set and reduces it to a low dimensional graph that retains a lot of the original information. It does so by giving each data point a location in a two or three-dimensional map. This technique finds clusters in data thereby making sure that an embedding preserves the meaning in the data. t-SNE reduces dimensionality while trying to keep similar instances close and dissimilar instances apart.

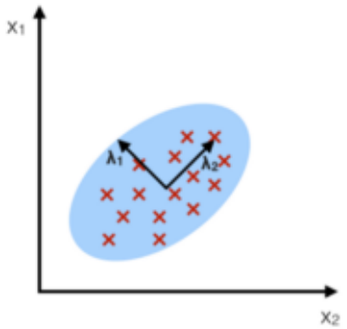
t-SNE is the dimensionality reduction which maps data in a higher dimensional space to that of a lower dimensional space just like PCA but uses a similarity measure like Euclidean distance to learn about discrepancies between pairs of data. While doing this, the local structures of data are preserved which is not the case of PCA.

LDA (Linear Discriminant Analysis)

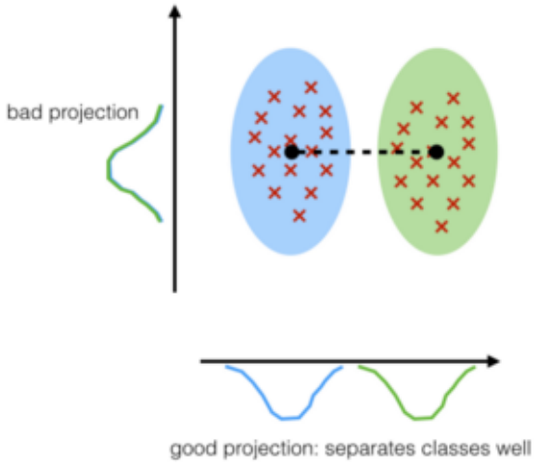
Linear Discriminant Analysis (LDA) is most commonly used as a dimensionality reduction technique in the pre-processing step for **pattern-classification.**

> The goal is to project a dataset onto a lower-dimensional space with good class-separability in order to avoid overfitting and also reduce computational costs.

PCA:
component axes that maximize the variance



LDA:
maximizing the component axes for class-separation



The general approach is very similar to PCA, rather than finding the component axes that maximize the variance of our data, ***we are additionally interested in the axes that maximize the separation between multiple classes*** (LDA).

LDA is “supervised” and computes the directions (“linear discriminants”)

UMAP (Uniform Manifold Approximation and Projection)

Uniform Manifold Approximation and Projection created in 2018 by **(**Leland McInnes, John Healy, James Melville) is a general-purpose manifold learning and dimension reduction algorithm.

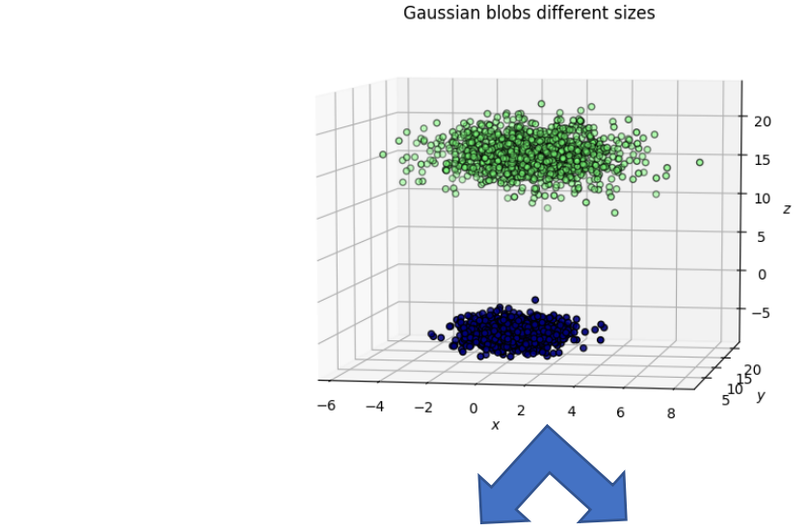
> UMAP is a ***nonlinear*** dimensionality reduction method, it is very effective for visualizing clusters or **groups of data points and their relative proximities**.

The significant difference with TSNE is **scalability**, it can be applied directly to sparse matrices thereby eliminating the need to applying any Dimensionality reduction such a*s* PCA or Truncated SVD(Singular Value Decomposition) as a prior pre-processing step*.*

Put simply, it is similar to t-SNE but with probably higher processing speed, therefore, faster and probably better visualization.

Table of Difference between PCA and t-SNE

	PCA	t-SNE
1.	It is a linear Dimensionality reduction technique.	It is a non-linear Dimensionality reduction technique.
2.	It tries to preserve the global structure of the data.	It tries to preserve the local structure(cluster) of data.
3.	It does not work well as compared to t-SNE.	It is one of the best dimensionality reduction technique.
4.	It does not involve Hyperparameters.	It involves Hyperparameters such as perplexity, learning rate and number of steps.
5.	It gets highly affected by outliers.	It can handle outliers.
6.	PCA is a deterministic algorithm.	It is a non-deterministic or randomised algorithm.
7.	It works by rotating the vectors for preserving variance.	It works by minimising the distance between the point in a guassian.
8.	We can find decide on how much variance to preserve using eigen values.	We cannot preserve variance instead we can preserve distance using hyperparameters.



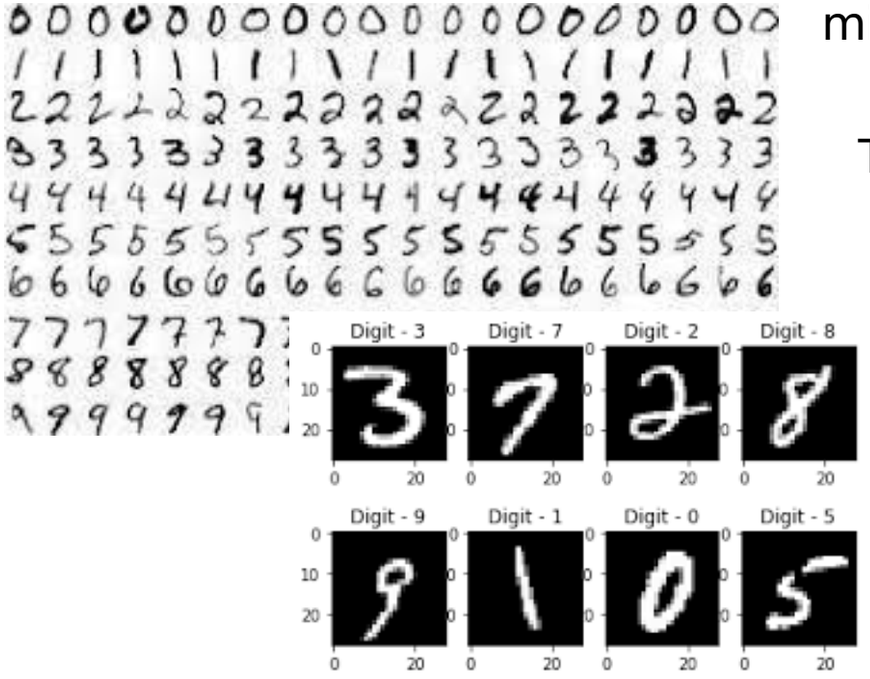
The MNIST dataset

THE MNIST DATABASE

of handwritten digits

Yann LeCun, Courant Institute, NYU
Corinna Cortes, Google Labs, New York
Christopher J.C. Burges, Microsoft Research, Redmond

The MNIST database of handwritten digits, available from http://yann.lecun.com/exdb/mnist/, has a training set of 60,000 examples, and a test set of 10,000 examples. It is a subset of a larger set available from NIST. The digits have been size-normalized and centered in a fixed-size image. It is a good database for people who want to try learning techniques and pattern recognition methods on real-world data while spending minimal efforts on preprocessing and formatting.



Transformation in csv

Image (handwritten numbers from 0 to 9)

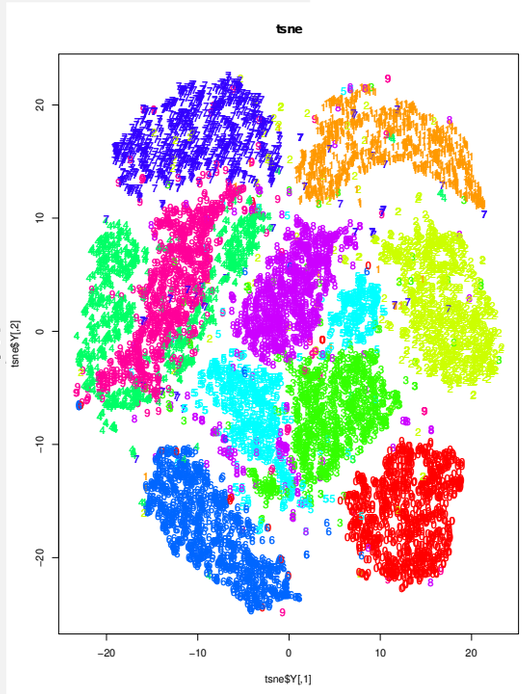
Pixel number

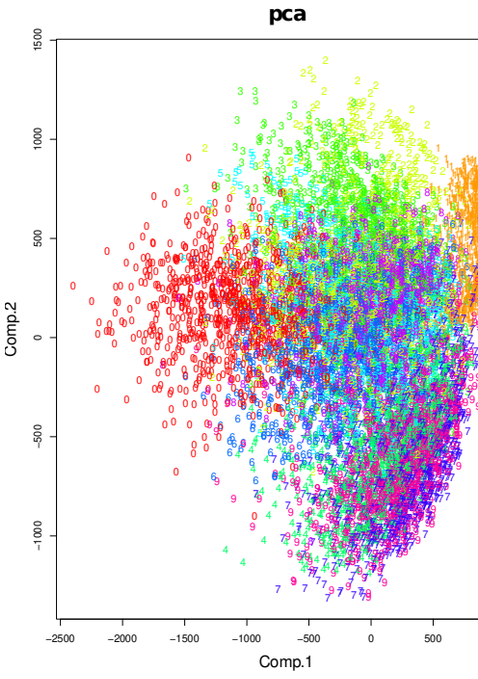
Color code associated to each pixel of each image

Let’s practice in R

Transformation of the original dataset in csv

```
train<-
read.csv("/Users/silviabottini/work/courses/2021/BIM_2021/IA/datasets/MNIST.csv")
## Curating the database for analysis with both t-SNE and PCA
Labels<-train$label
train$label<-as.factor(train$label)
## for plotting
colors = rainbow(length(unique(train$label)))
names(colors) = unique(train$label)

#t-SNE
library(Rtsne)
## Executing the algorithm on curated data & plotting
tsne <- Rtsne(train[,-1], dims = 2, perplexity=30, verbose=TRUE, 500)
plot(tsne$Y, t='n', main="tsne")
text(tsne$Y, labels=train$label, col=colors[train$label])

#PCA
## Executing the algorithm on curated data & plotting
pca = princomp(train[,-1])$scores[,1:2]
```





Today’s program

Machine learning

- **Supervised learning**
  - **Simple Linear regression**
  - **Multiple Linear regression**
  - **Factor regression**
  - **Logistic regression**
  - **Decision Trees**
  - **Random Forests**
  - **SVM**
- **Unsupervised learning**
  - **Dimensionality reduction**
  - **Clustering**

Types of Clustering Methods

Clustering methods are used to identify groups of similar objects in a multivariate data sets collected from fields such as marketing, bio-medical and geo-spatial. They are different **types of clustering** methods, including:

- **Partitioning methods**
- **Hierarchical clustering**
- Fuzzy clustering
- Density-based clustering
- Model-based clustering

```
library("cluster")
library("factoextra")
library("magrittr")
```

Data preparation

```
# Load and prepare the data
data("USArrests")
my_data <- USArrests %>%
na.omit() %>% # Remove missing values (NA)
scale() # Scale variables

# View the firt 3 rows
head(my_data, n = 3)
```

- Demo data set: the built-in R dataset named USArrest
- Remove missing data
- Scale variables to make them comparable

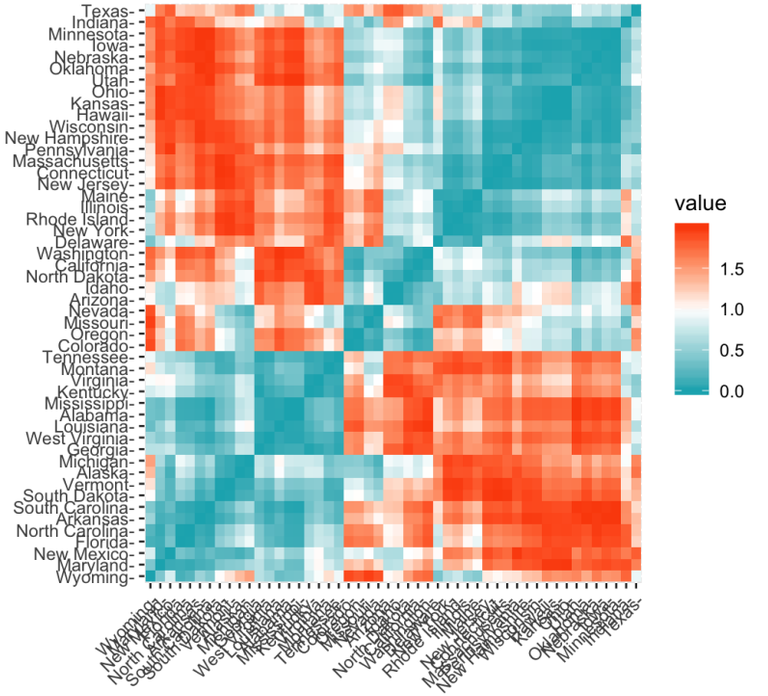
```
##      Murder Assault UrbanPop Rape
## Alabama 1.2426 0.783 -0.521 -0.00342
## Alaska 0.5079 1.107 -1.212 2.48420
## Arizona 0.0716 1.479 0.999 1.04288
```

Distance measures

The classification of objects, into clusters, requires some methods for measuring the distance or the (dis)similarity between the objects.
It’s simple to compute and visualize distance matrix using the functions get_dist() and fviz_dist() [factoextra R package]:

- get_dist(): for computing a distance matrix between the rows of a data matrix. Compared to the standard dist() function, it supports correlation-based distance measures including “pearson”, “kendall” and “spearman” methods.
- fviz_dist(): for visualizing a distance matrix

```
res.dist <- get_dist(USArrests, stand = TRUE, method = "pearson")
fviz_dist(res.dist, gradient = list(low = "#00AFBB", mid = "white", high = "#FC4E07"))
```



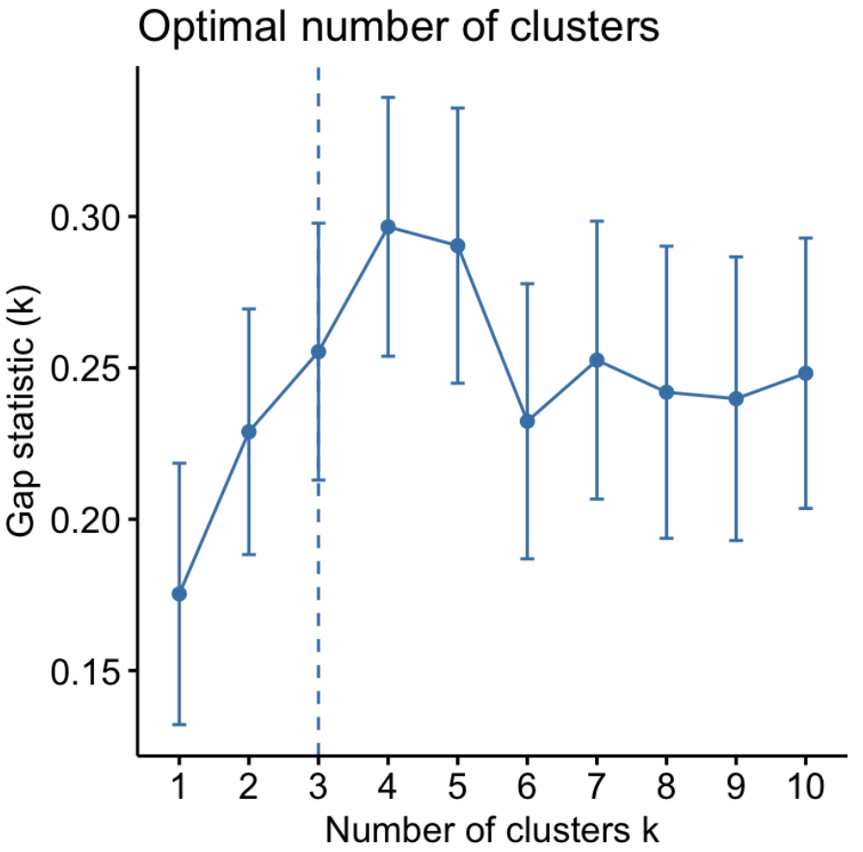
Partitioning clustering

Partitioning algorithms are clustering techniques that subdivide the data sets into a set of k groups, where k is the number of groups pre-specified by the analyst.
There are different types of partitioning clustering methods. The most popular is the K-means clustering (MacQueen 1967), in which, each cluster is represented by the center or means of the data points belonging to the cluster. The K-means method is sensitive to outliers.
An alternative to k-means clustering is the K-medoids clustering or PAM (Partitioning Around Medoids, Kaufman & Rousseeuw, 1990), which is less sensitive to outliers compared to k-means.

Step 1: Determining the optimal number of clusters:

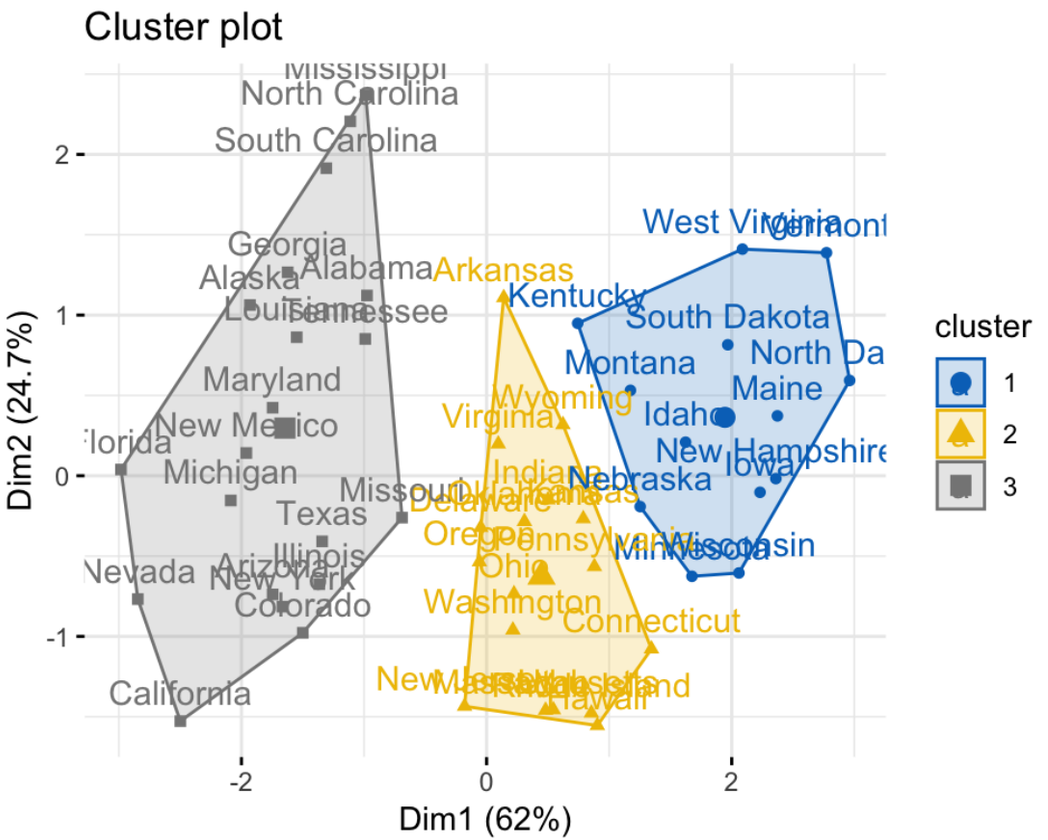
```
library("factoextra")
fviz_nbclust(my_data, kmeans, method = "gap_stat")
```

Suggested number of cluster: 3



Step 2: Compute and visualize k-means clustering

```
set.seed(123)
km.res <- kmeans(my_data, 3, nstart = 25)
# Visualize
library("factoextra")
fviz_cluster(km.res, data = my_data,
ellipse.type = "convex", palette = "jco",
ggtheme = theme_minimal())
```



Step 2 (alternative): Compute and visualize k-medoids/pam clustering

```
# Compute PAM
library("cluster")
pam.res <- pam(my_data, 3)
# Visualize
fviz_cluster(pam.res)
```

Hierarchical clustering

Hierarchical clustering is an alternative approach to partitioning clustering for identifying groups in the dataset.

It does not require to pre-specify the number of clusters to be generated.

The result of hierarchical clustering is a tree-based representation of the objects, which is also known as dendrogram.

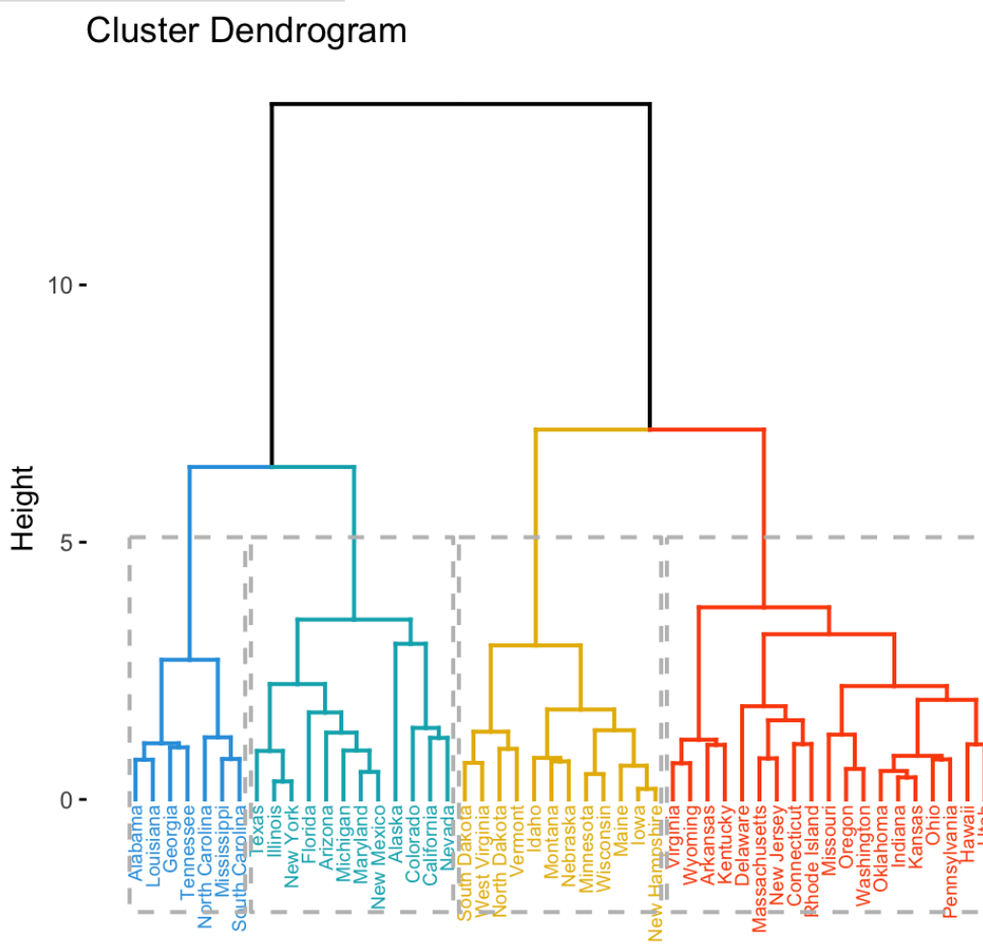
Observations can be subdivided into groups by cutting the dendrogram at a desired similarity level.

Step 1: compute hierarchical clustering

```
# Compute hierarchical clustering

res.hc <- USArrests %>% scale() %>% # Scale the data

dist(method = "euclidean") %>% # Compute dissimilarity matrix

hclust(method = "ward.D2") # Compute hierachical clustering
```

Step 2: cut and visualize

```
# Visualize using factoextra
# Cut in 4 groups and color by groups

fviz_dend(res.hc, k = 4, # Cut in four groups
      cex = 0.5, # label size
      k_colors = c("#2E9FDF", "#00AFBB", "#E7B800", "#F
      color_labels_by_k = TRUE, # color labels by groups
      rect = TRUE # Add rectangle around groups )
```



Clustering validation and evaluation

Clustering validation and evaluation strategies, consist of measuring the goodness of clustering results.

Before applying any clustering algorithm to a data set, the first thing to do is to assess the *clustering tendency*. That is, whether the data contains any inherent grouping structure.

If yes, then how many clusters are there.

Next, you can perform hierarchical clustering or partitioning clustering (with a pre-specified number of clusters).

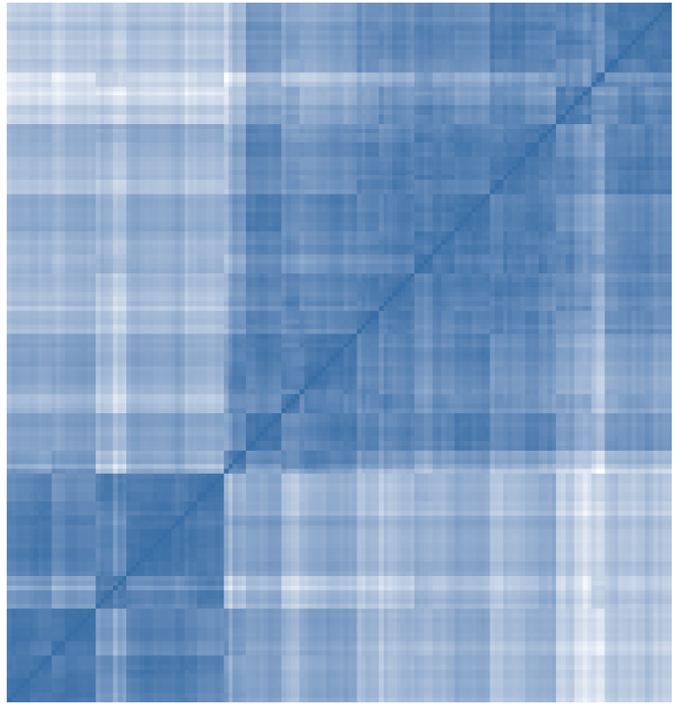
Finally, you can use a number of measures, to evaluate the goodness of the clustering results.

Assessing clustering tendency

To assess the clustering tendency, the Hopkins' statistic and a visual approach can be used. This can be performed using the function get_clust_tendency() [factoextra package], which creates an ordered dissimilarity image (ODI).
Hopkins statistic: If the value of Hopkins statistic is close to 1 (far above 0.5), then we can conclude that the dataset is significantly clusterable.
Visual approach: The visual approach detects the clustering tendency by counting the number of square shaped dark (or colored) blocks along the diagonal in the d

```
gradient.color <- list(low = "steelblue", high =
"white")
iris[, -5] %>% # Remove column 5 (Species
 scale() %>% # Scale variables
 get_clust_tendency(n = 50, gradient = gradient.color)
```

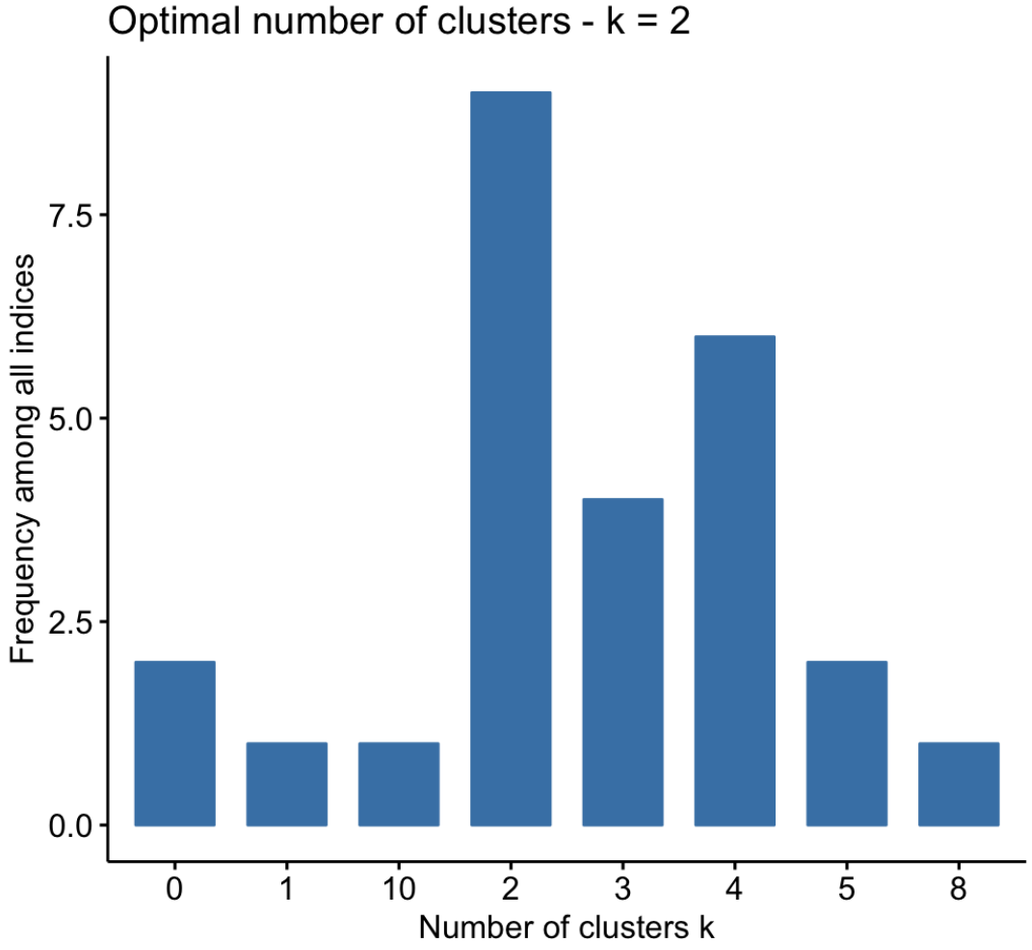
```
## $hopkins_stat
## [1] 0.8
##
## $plot
```



Determining the optimal number of clusters

In the R code below, we'll use the NbClust R package, which provides 30 indices for determining the best number of clusters.

```
set.seed(123)
# Compute
library("NbClust")
res.nbclust <- USArrests %>%
scale() %>%
NbClust(distance = "euclidean", min.nc =
2, max.nc = 10, method = "complete", index
="all")
# Visualize
library(factoextra)
fviz_nbclust(res.nbclust, ggtheme =
theme_minimal())
```



Clustering validation statistics

A variety of measures has been proposed in the literature for evaluating clustering results. The term clustering validation is used to design the procedure of evaluating the results of a clustering algorithm.

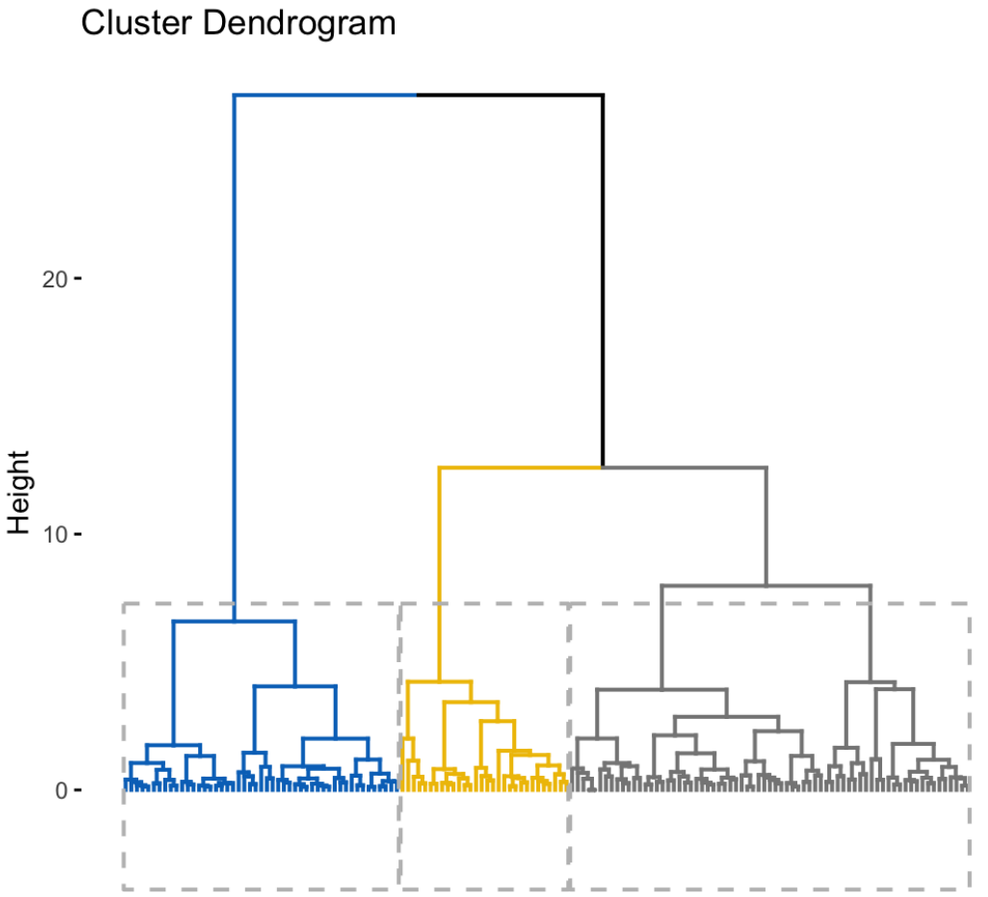
The ***silhouette plot*** is one of the many measures for inspecting and validating clustering results.

The silhouette (Si) measures how similar an object ii is to the the other objects in its own cluster versus those in the neighbor cluster. Si values range from 1 to - 1:

- A value of Si close to 1 indicates that the object is well clustered. In the other words, the object ii is similar to the other objects in its group.
- A value of Si close to -1 indicates that the object is poorly clustered, and that assignment to some other cluster would probably improve the overall results.

Step 1: Compute and visualize hierarchical clustering

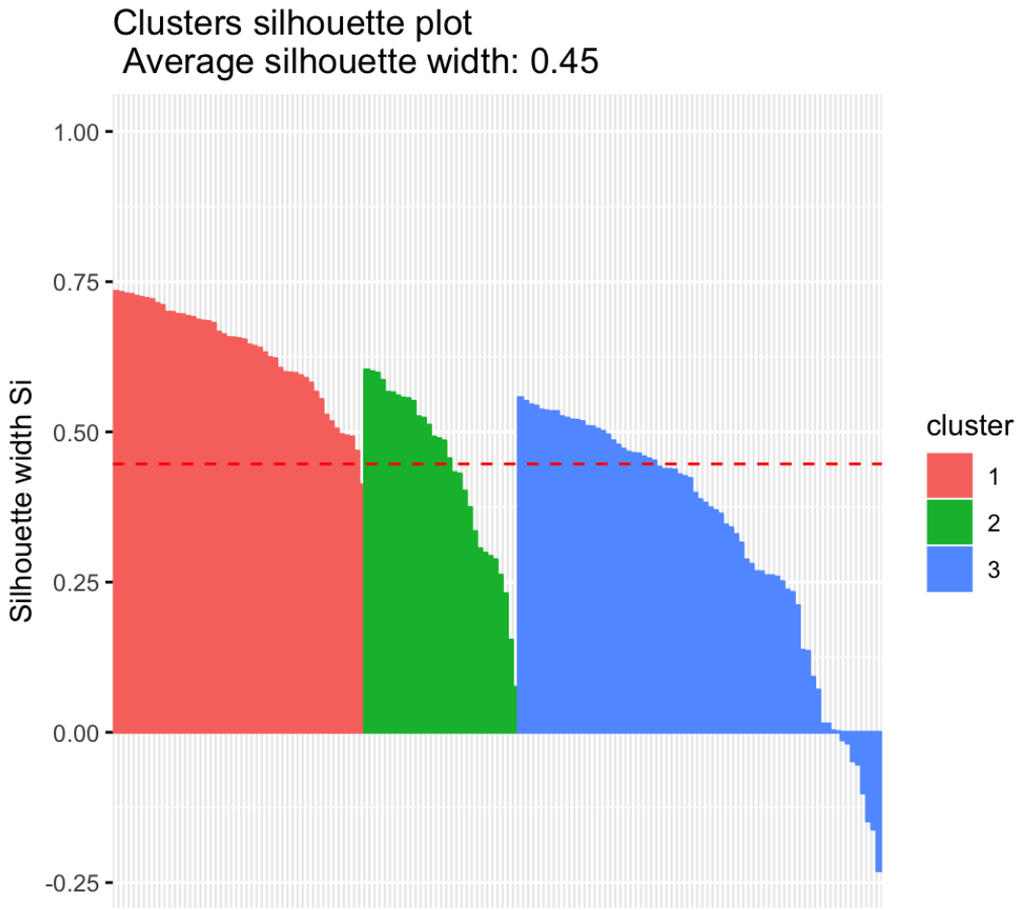
```
set.seed(123)
# Enhanced hierarchical
clustering, cut in 3 groups
res.hc <- iris[, -5] %>% scale()
%>% eclust("hclust", k = 3, graph
= FALSE)
# Visualize with factoextra
fviz_dend(res.hc, palette =
"jco", rect = TRUE, show_labels =
FALSE)
```



Step 2: Inspect the silhouette plot

```
fviz_silhouette(res.hc)
```

```
## cluster size ave.sil.width
## 1 1 49 0.63
## 2 2 30 0.44
## 3 3 71 0.32
```



Step 3: Which samples have negative silhouette? To what cluster are they closer?

```
# Silhouette width of observations
sil <- res.hc$silinfo$widths[, 1:3]
# Objects with negative silhouette
neg_sil_index <- which(sil[, 'sil_width'] < 0)
sil[neg_sil_index, , drop = FALSE]
```

```
## cluster neighbor sil_width
## 84 3 2 -0.0127
## 122 3 2 -0.0179
## 62 3 2 -0.0476
## 135 3 2 -0.0530
## 73 3 2 -0.1009
## 74 3 2 -0.1476
## 114 3 2 -0.1611
## 72 3 2 -0.2304
```