The Theory of Information and Coding was developed to deal with the fundamental problem of communication, that of reproducing at one point, either exactly or approximately, a message sent from another point.

Information Theory:
Theoretical capabilities of these communication systems considering the communication rate and the probability of error of the codes.

Coding Theory:
This is concerned with the design of effective error-correction codes. When the code is designed to reduce the requirement of memory resources for storing data, it is a compressor code.

Part one: Information Theory
1. Introduction to Theory of Information
2. Discrete memoryless channels

Part two: Coding Theory
3. Linear codes
4. Cyclic codes

Part three: Compression
5. Data compression
6. Image compression

Part four:
7. Introduction to Data Mining

Computer classes based on the R system (syllabus)
Benefits of the R system:

- Computer science students know the art of programming
- They understand a theme better when they program it
- It interfaces with other languages (C)
- It is a free system
- It has a powerful programming language
- It contains extensive and powerful graphics abilities
- The R system is continuously being developed

Teaching methodology

The practicals include the design, use and programming with the R system. On one hand, the “InformationCoding Library” is still being developed, and on the other hand, the last chapter is available in R.

InformationCoding Library

This library is still in the construction stage and can be found in four main blocks:

1. Entropy functions
   These functions compute entropy (univariate, joint, conditional) and mutual information.

2. Simulation of communication channels
   This block lets the transmission of a message over a digital communication channel be simulated. Some numerical and graphical summaries are produced.

3. Run-length codes
   Coding and decoding of some classic algorithms used to compress data. Some of these codes are associated to the pioneer work of Claude Shannon.

4. Fixed-length codes
   These functions code and decode messages with some of the most important algorithms used in coding practice.

InformationCoding library:
Block 1: Entropy functions

Some of these functions are:

- entropyone(p): computes the entropy function given a probability p defining a two-result vector probability (p,1-p)
- entropytwo(x,y): computes the entropy function given two probabilities x,y which define a three-result vector probability (x,y,1-x-y)
- entropy(p): computes the entropy function from a probability vector p
- jointentropy(P): computes the joint entropy function from a probability matrix P
- condientropy(P,margin): computes the conditional entropy function from a probability matrix by conditioning on the rows (margin=1) or the columns (margin=2). It also obtains the conditional entropy for each of the rows or columns.
- mutinf(P): computes the mutual information given a probability matrix P.
Some of the previous functions can be utilized to graphically represent the entropy:

\[
\text{entropyone}\left(p\right) = p \log_2\left(\frac{1}{p}\right) + \left(1-p\right) \log_2\left(\frac{1}{1-p}\right)
\]

\[
\text{curve}(\text{entropyone}, 0, 1, 1000, \text{col} = \text{"blue"}, \text{lwd} = 2, \text{ylab} = \text{"H2"}, \text{main} = \text{"Entropy, H(X), n=2"}, \text{type} = \text{"l"})
\]

\[
\text{entropytwo}(x, y) = \begin{cases} 
-x \log_2(x) - y \log_2(y) - z \log_2(z), & z > 0 \\
0, & \text{otherwise}
\end{cases}
\]

\[
x = (0:100)/100 \\
y = (0:100)/100 \\
z = \text{outer}(x, y, \text{entropytwo})
\]

\[
\text{persp}(x, y, z, \theta = 15, \phi = 30, \text{expand} = 0.5, \text{col} = \text{"lightblue"}, \text{xlab} = \text{"p1"}, \text{ylab} = \text{"p2"}, \text{zlab} = \text{"H"}, \text{main} = \text{"H, n=3"})
\]

InformationCoding library:

Block 2: Simulation of Communication Channels

\[
\text{simulate.channel}(n, a, p, \text{prober}, \text{mis}): \text{this simulates the transmission of a message formed by } n \text{ symbols of the alphabet } a, \text{ with vector probability } p \text{ and a probability of error } \text{prober. mis=TRUE allows missing symbols (coded as -1) in the transmission.}
\]

\[
\text{simulate.bsc}(n, l, \text{prober}): \text{this simulates the transmission over a binary symmetric channel of a message formed by } n \text{ binary vectors of size } l, \text{ and a probability of error } \text{prober.}
\]

An example to illustrate the information produced by the function:

\[
\text{simulate.channel}(100, c(0,1,2), c(1/3,1/3,1/3), 0.15, \text{TRUE})
\]

IC 95\% probability of error= ( 0.1495564 , 0.1964436 )

Distribution of sent symbols (%):
0  1    2
34.1 33.5 32.4

Distribution of received symbols (%):
-1    0    1   2 5.3 31.8 32.3 30.6

Distribution of source symbols presenting errors:
0:  17.59\% 1:  17.31\% 2:  16.97\%  

InformationCoding library:

Block 3: Run-Length Codes

The Run-Length codes currently implemented are Shannon, Shannon-Fano, and Arithmetic code, while Huffman code is in developing phase.

For example:

\[
\text{Shannon.code}(m, a, p): \text{this obtains the Shannon codes for the message } m \text{ from an alphabet } a \text{ with associated probability vector } p.
\]

\[
\text{Shannon.decode}(\text{rec}, a, p): \text{decoding the Shannon code.}
\]

STEP 1: Sort the messages \(m_i\) by sorting their probabilities into decreasing order.

\[
\text{messsort}<-\text{sort}(p, \text{decreasing}=\text{TRUE}, \text{index.return}=\text{TRUE})
\]

\[
m<-m[\text{messsortSix}] \\
p<-\text{messsortSix}
\]

STEP 2: Compute \(\alpha_i; \alpha_1 = 0, \alpha_2 = P(m_1), \alpha_j = P(m_1) + P(m_2) + \ldots + P(m_j)\) (accumulated probabilities)

\[
\text{computealpha}<-\text{function}(p) \\
\{ \\
\text{\.long}<-\text{length}(p) \\
\text{c}(0, \text{cumsum}(p[.long])) \\
\}
\]

\[
\text{computealpha}(p)
\]

\[
\begin{array}{cccc}
m & p \\
0 & 0.30 & 0 & 0 \\
1 & 0.25 & 0 & 1 \\
2 & 0.20 & 1 & 0 \\
3 & 0.15 & 1 & 0 \\
4 & 0.10 & 1 & 0 \\
5 & 0.10 & 1 & 0 \\
6 & 0.05 & 1 & 1 \\
\end{array}
\]

\[
\text{computealpha}(p)
\]

\[
\begin{array}{cccc}
m & p \\
0 & 0.30 & 0 & 0 \\
1 & 0.25 & 0 & 1 \\
2 & 0.20 & 1 & 0 \\
3 & 0.15 & 1 & 0 \\
4 & 0.10 & 1 & 0 \\
5 & 0.10 & 1 & 0 \\
6 & 0.05 & 1 & 1 \\
\end{array}
\]

InformationCoding library:

Block 3: Run-Length Codes

The Run-Length codes currently implemented are Shannon, Shannon-Fano, and Arithmetic code, while Huffman code is in developing phase.

For example:

\[
\text{Shannon.code}(m, a, p): \text{this obtains the Shannon codes for the message } m \text{ from an alphabet } a \text{ with associated probability vector } p.
\]

\[
\text{Shannon.decode}(\text{rec}, a, p): \text{decoding the Shannon code.}
\]

STEP 1: Sort the messages \(m_i\) by sorting their probabilities into decreasing order.

\[
\text{messsort}<-\text{sort}(p, \text{decreasing}=\text{TRUE}, \text{index.return}=\text{TRUE})
\]

\[
m<-m[\text{messsortSix}] \\
p<-\text{messsortSix}
\]

STEP 2: Compute \(\alpha_i; \alpha_1 = 0, \alpha_2 = P(m_1), \alpha_j = P(m_1) + P(m_2) + \ldots + P(m_j)\) (accumulated probabilities)

\[
\text{computealpha}<-\text{function}(p) \\
\{ \\
\text{\.long}<-\text{length}(p) \\
\text{c}(0, \text{cumsum}(p[.long])) \\
\}
\]

\[
\text{computealpha}(p)
\]

\[
\begin{array}{cccc}
m & p \\
0 & 0.30 & 0 & 0 \\
1 & 0.25 & 0 & 1 \\
2 & 0.20 & 1 & 0 \\
3 & 0.15 & 1 & 0 \\
4 & 0.10 & 1 & 0 \\
5 & 0.10 & 1 & 0 \\
6 & 0.05 & 1 & 1 \\
\end{array}
\]

STEP 3: Determine \(n_i\): \(2^{n_i} \geq 1/p_i \geq 2^{n_i-1}\) (determining \(n_i\) such that is fulfilled)

\[
\text{seekN}<-\text{function}(p)
\{
\text{ceiling}(-\text{log2}(p))
\}
\]

STEP 4: The code of \(m_i\) is the binary expression of \(\alpha_i\) up to the \(n_i\)th binary digit

\[
\text{binarycode}<-\text{function}(a,n)
\{
\text{auxi}<-\text{c}()
\text{for(i in 1:n)}
\{
\text{auxi}[i]<-\text{floor}(a*2)
\text{a<-a*2-floor(a*2)}
\}
\}
\]

\[
\text{binarycode}(a,n)
\]

\[
\begin{array}{cccc}
m & p \\
0 & 0.30 & 0 & 0 \\
1 & 0.25 & 0 & 1 \\
2 & 0.20 & 1 & 0 \\
3 & 0.15 & 1 & 0 \\
4 & 0.10 & 1 & 0 \\
5 & 0.10 & 1 & 0 \\
6 & 0.05 & 1 & 1 \\
\end{array}
\]

STEP 3: Determine \(n_i\): \(2^{n_i} \geq 1/p_i \geq 2^{n_i-1}\) (determining \(n_i\) such that is fulfilled)

\[
\text{seekN}<-\text{function}(p)
\{
\text{ceiling}(-\text{log2}(p))
\}
\]

STEP 4: The code of \(m_i\) is the binary expression of \(\alpha_i\) up to the \(n_i\)th binary digit

\[
\text{binarycode}<-\text{function}(a,n)
\{
\text{auxi}<-\text{c}()
\text{for(i in 1:n)}
\{
\text{auxi}[i]<-\text{floor}(a*2)
\text{a<-a*2-floor(a*2)}
\}
\}
\]

\[
\text{binarycode}(a,n)
\]
The last block of this library includes the following fixed-length codes:

- **Repetition codes**
  - **Linear codes:**
    - Hamming
    - Golay
    - Reed-Muller

- **Cyclic codes:**
  - Polynomial codes
  - Reed-Solomon

---

### Repetition codes
The message $s$ is coded with a repetition code, where each symbol in $s$ is repeated $N$ times.

#### Reproduction
The received message $r$ is decoded by taking the majority vote of each $N$ consecutive bits.

#### Bit error probability
The function `proberr` computes the bit error probability of a repetition code in a symmetric binary channel with error probability $p$.

#### Implementation in R
```r
rep.code <- function(s, N) { rep(s, each = N) }
```

---

### Hamming Codes
- **Control matrix**:\( H = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 \end{bmatrix} \)
- **Code**: \( \{000, 001, 010, 011, 100, 101, 110, 111\} \)
- **Prober**: \( p_{\text{err}} = \frac{1}{2^r} \)
- **Implementation in R**
```r
controlmatrix.Ham(3)
```

---

### Plotting Bit Error Probability
The function `proberror` can be graphically represented by means of the `curve` function:

```r
curve(x^2, 0,1,100,col="black",lty=2,main="Bit probability error of the repetition code, N=2n+1",lwd=2,ylab="pE",xlab="p")
curve(proberror(x,3),0,1,200,col="red",lwd=2,add=TRUE)
curve(proberror(x,5),0,1,200,col="green",lwd=2,add=TRUE)
curve(proberror(x,11),0,1,200,col="blue",lwd=2,add=TRUE)
```
Information Coding library
Fixed-Length Codes, Linear Codes:
Golay Codes

\texttt{genmatrix.Golay(r)}: this builds the generator matrix of the Golay-24 code \((r=24)\) or the Golay-23 code \((r=23)\).
\texttt{code.Golay(m, r)}: this codes message \(m\) with the Golay code.
\texttt{decode.Golay(rec, r)}: this decodes message \(rec\) with the Golay code.

Reed Muller Codes

\texttt{genmatrix.RM(r)}: this builds the generator matrix of the Reed Muller code with length \(2^r\) and dimension \(r+1\).
\texttt{code.RM(m, r)}: this codes message \(m\) with the Golay code.
\texttt{decode.RM(rec, r)}: this decodes message \(rec\) with the Golay code.

Example of Reed Muller Code

These are the matrices of Reed Muller generator of order 3 and 4 respectively:

\begin{verbatim}
> genmatrix.RM(3)
   1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
   0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
   0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0
   0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0
> genmatrix.RM(4)
   1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
   0 1 0 1 1 1 0 1 0 1 1 1 0 1 0 1 1 1 0 1 0 1 1 1 0
   0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0
   0 0 0 0 1 1 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 0
   0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
\end{verbatim}

We can simulate the transmission of the coded message with the function \texttt{simulate.csb}:

\begin{verbatim}
> r<- simulate.csb(t,0.1)
   1 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0
   0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
   0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0
   0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0
   0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
\end{verbatim}

The decoding of the received message is obtained with the function \texttt{decode.RM}.

\begin{verbatim}
> decode.RM(r,3)
   0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0
   0 0 1 1 0 1 0 0 0 1 1 0 1 0 0 1 1 0 1 0 0 1 1 0 1
   0 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1
   0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
\end{verbatim}

Example of Reed Muller code

We suppose the following message formed by 10 vectors of size 4:

\begin{verbatim}
> m
  [1,]    0    0    0    1    1    1    1    1    1    1
  [2,]    0    1    0    1    1    0    1    0    1    0
  [3,]    0    0    0    0    0    0    0    0    0    0
  [4,]    0    0    1    1    0    0    1    1    0    0
  [5,]    0    1    1    0    0    1    1    0    0    0
  [6,]    1    0    0    1    1    0    0    1    1    1
  [7,]    0    0    0    0    0    0    0    0    1    1
  [8,]    0    0    1    1    0    0    1    1    0    0
  [9,]    0    0    0    0    1    1    1    1    0    0
 [10,]   0    0    0    0    0    0    0    0    0    1
\end{verbatim}

Given the generator matrix \(G\) then the coding of a vector \(s\) becomes \((s \times G) \mod 2\)

The resulting code would be:

\begin{verbatim}
> code.RM(m,3)
  [1,]    0    0    0    0    1    1    1    1    1    1
  [2,]    0    0    0    0    0    0    0    0    1    1
  [3,]    0    0    1    1    0    0    1    1    0    0
  [4,]    0    0    1    1    0    0    1    1    0    0
  [5,]    0    0    1    1    0    0    1    1    0    0
  [6,]    0    0    0    0    1    1    1    1    0    0
  [7,]    0    0    0    0    0    0    0    0    1    1
  [8,]    0    0    0    0    1    1    1    1    0    0
  [9,]    0    0    0    0    0    0    0    0    1    1
 [10,]   0    0    0    0    0    0    0    0    0    1
\end{verbatim}

We can simulate the transmission of the coded message with the function \texttt{simulate.csb}:

\begin{verbatim}
> r<- simulate.csb(t,0.1)
  [1,]    1    0    0    0    1    1    1    1    1    1
  [2,]    0    1    0    1    1    1    0    1    0    1
  [3,]    0    0    0    0    0    0    0    0    1    1
  [4,]    0    0    0    0    1    1    0    0    1    1
  [5,]    0    0    0    0    1    1    0    0    1    1
  [6,]    0    1    1    0    0    1    1    0    0    1
  [7,]    0    0    1    1    0    0    1    1    0    0
  [8,]    0    0    0    0    1    1    1    1    0    0
  [9,]    0    0    0    0    0    0    0    0    1    1
 [10,]   0    0    0    0    0    0    0    0    0    1
\end{verbatim}

The decoding of the received message is obtained with the function \texttt{decode.RM}:
These require the implementation of the elements of the Galois Field \( \text{GF}(2^n) \), including the sum and product functions.

The Reed Solomon codes work with blocks of \( n \) symbols: whereas in the previous codes \( K \) bits are coded as a codeword of size \( N \) bits, now \( K \) blocks of \( n \) bits are coded by \( N \) blocks of \( n \) bits.

**Reed Solomon Codes**

These require the implementation of the elements of the Galois Field \( \text{GF}(2^n) \), including the sum and product functions.

The Reed Solomon codes work with blocks of \( n \) symbols: whereas in the previous codes \( K \) bits are coded as a codeword of size \( N \) bits, now \( K \) blocks of \( n \) bits are coded by \( N \) blocks of \( n \) bits.

**Example of Reed Solomon Codes**

\( n=3, \ K=3, \ N=7 \)

We want to send this message, 3 elements of \( \text{GF}(2^3) \):

\[
110 \ 110 \ 111
\]

which becomes a coded message, 7 elements of \( \text{GF}(2^3) \):

\[
001 \ 000 \ 111 \ 000 \ 110 \ 110 \ 111
\]

**Polynomgen.RS(N, K, n)**: generator polynomial for a Reed Solomon code with length \( N \), dimension \( K \), over a Galois Field \( \text{GF}(2^n) \).

**code.RS(m, N, K, n)**: codes the message \( m \) with the Reed Solomon code.

**decode.RS(rec, N, K, n)**: decodes the message \( rec \) with the Reed Solomon code. It is based on the Berlekamp Massey algorithm.

**Data Mining**

The subject also includes an introduction to the main machine-learning models. The brief theoretical presentation is accompanied by some examples.

Neural Networks: multilayer perceptron with the \textit{nnet} library.

CART: Classification and regression trees with the \textit{rpart} library.

SVM: Support Vector Machines with the \textit{svm} function in the \textit{e1071} library.