

Data Scientist y lenguaje R

Guía de autoformación
para el uso de Big Data



Archivos complementarios
para descarga



Henri LAUDE

Introducción

1. Data scientist, una disciplina de moda

2. Las data sciences

3. El Big Data

4. La dinámica de este libro

- 4.1 Nuestros objetivos
- 4.2 La estructura del libro
 - 4.2.1 Los dos recorridos complementarios
 - 4.2.2 Recursos complementarios

5. Pequeño bestiario de las data sciences

- 5.1 Los fundamentos
 - 5.1.1 Aprendizaje y clasificación
 - 5.1.2 Pequeño vocabulario gráfico del machine learning
 - 5.1.3 Regresión
 - 5.1.4 Regresión lineal generalizada
 - 5.1.5 Árboles de decisión, poda, tala
 - 5.1.6 Clustering, k-means
 - 5.1.7 k-NN
 - 5.1.8 Modelos paramétricos
 - 5.1.9 Lazy algorithm (algoritmo perezoso)
 - 5.1.10 Overfitting: sobredeterminación, sobreaprendizaje
 - 5.1.11 Validación cruzada, regularización, bagging
 - 5.1.12 Optimización, descenso por gradiente
 - 5.1.13 Algoritmo voraz (greedy algorithm)
 - 5.1.14 Programación lineal, simplex, punto interior
 - 5.1.15 Estimación mediante el método de Monte-Carlo
 - 5.1.16 Entropía, independencia e información mutua
 - 5.1.17 Discretización
- 5.2 Métodos «conjunto»

- 5.2.1 Random forest
- 5.2.2 AdaBoost (adaptative boosting)
- 5.3 Leyes de probabilidad y de distribución
 - 5.3.1 Generalidades
 - 5.3.2 Pequeño bestiario de leyes de probabilidad
- 5.4 Los grafos
 - 5.4.1 Vocabulario básico
 - 5.4.2 Conversión de una tabla de observaciones en un grafo, semejanza

6. Informática profesional y datasciences

- 6.1 La tecnología
- 6.2 Business Intelligence versus Big Data
 - 6.2.1 Diferencias en términos de arquitectura
 - 6.2.2 Diferencias en términos de uso
 - 6.2.3 En resumen

7. Notación

- 7.1 Notación de los parámetros
- 7.2 Otras notaciones
 - 7.2.1 Funciones y aplicaciones ... $f(x)$, $d(x,y)$...
 - 7.2.2 Algunas posibles confusiones

8. Ahora, ¡es su turno!

Primeros pasos con R

1. Instalación de los componentes

- 1.1 Instalación y ejecución de R
- 1.2 Instalación y ejecución de RStudio
- 1.3 Instalación de nuevos paquetes
- 1.4 Instalación de paquetes: complementos

2. Toma de contacto con R

- 2.1 R, una calculadora eficaz
- 2.2 R, un lenguaje vectorial
- 2.3 Funciones que trabajan sobre vectores
 - 2.3.1 Un primer análisis rápido de los datos
 - 2.3.2 Algunas estadísticas sencillas sobre los vectores
 - 2.3.3 Ordenar un vector
 - 2.3.4 Diversas funciones con suma, producto, min y max
- 2.4 Tipos de datos simples
 - 2.4.1 Los booleanos
 - 2.4.2 Conjuntos
 - 2.4.3 Listas
 - 2.4.4 Factores
 - 2.4.5 Tablas
- 2.5 Las funciones
 - 2.5.1 Creación y uso de una función simple
 - 2.5.2 Creación de un operador a partir de una función de dos variables
 - 2.5.3 Uso de las funciones y alcance de las variables
 - 2.5.4 Aplicación de las funciones sobre las matrices: apply
 - 2.5.5 Las funciones: completamente útiles
- 2.6 Estructuras de control
 - 2.6.1 Instrucciones comunes con otros lenguajes
 - 2.6.2 Recorrer una matriz mediante bucles for
- 2.7 Las cadenas de caracteres
- 2.8 El formato de los números
- 2.9 Fechas y tiempos
- 2.10 Medir la duración de un algoritmo
- 2.11 Los números complejos
 - 2.11.1 Manipulación básica de los números complejos
 - 2.11.2 Visualización de números complejos
- 2.12 Programación orientada a objetos
 - 2.12.1 Clases y objetos, breve descripción
 - 2.12.2 Constructores
 - 2.12.3 Herencia
 - 2.12.4 Objetos mutables
 - 2.12.5 Gestión de la pila: implementación Orientada a Objetos con RC

3. Manipulación de los datos

- 3.1 Lectura de los datos: fundamentos
- 3.2 Manipulación de las columnas de un data.frame
- 3.3 Cálculos simples sobre un data.frame
 - 3.3.1 Cálculos sobre las columnas y las filas
 - 3.3.2 Manipulación de las filas
 - 3.3.3 Aplicación: comparación de elementos de clases y Khi-2
 - 3.3.4 Creación de columnas calculadas
 - 3.3.5 Ordenar un data.frame mediante order()
- 3.4 Análisis visual de los datos
 - 3.4.1 Visualización simple de los datos
 - 3.4.2 Visualización de variables numéricas 2 a 2 con mención de las clases
 - 3.4.3 Correlación entre variables numéricas
 - 3.4.4 Separación por clase, ggplot2, qplot
 - 3.4.5 Visualización 3D, relación entre tres variables numéricas
 - 3.4.6 Gráficos por pares
 - 3.4.7 Diagramas de caja y eliminación de outliers
 - 3.4.8 Creación de un modelo por árbol de decisión

Dominar los fundamentos

1. Ponerse en armonía con los datos

- 1.1 Algunas nociones fundacionales
 - 1.1.1 Fenómeno aleatorio
 - 1.1.2 Probabilidad, variable aleatoria y distribución
 - 1.1.3 Un poco de matemáticas: notaciones y definiciones útiles
 - 1.1.4 Momentos de una variable aleatoria discreta X
 - 1.1.5 Primeras consideraciones sobre los errores y estimaciones
- 1.2 Familiarizarse con los datos
 - 1.2.1 R Commander
 - 1.2.2 Rattle

2. Matrices y vectores

- 2.1 Convenciones, notaciones, usos básicos
- 2.2 Matrices, vectores: una introducción a la noción de aprendizaje supervisado
- 2.3 Ir más lejos en la manipulación de matrices con R
 - 2.3.1 Operaciones básicas
 - 2.3.2 Algunos trucos útiles sobre las matrices de R
 - 2.3.3 Normas de vectores y normas de matrices
 - 2.3.4 Matrices y vectores: diversas sintaxis útiles

3. Estimaciones

- 3.1 Planteamiento del problema de estimación
 - 3.1.1 Formulación general del problema
 - 3.1.2 Aplicación y reformulación del problema de estimación
- 3.2 Indicadores de desviación utilizados en machine learning
 - 3.2.1 MSE, RMSE, SSE, SST
 - 3.2.2 MAE, ME
 - 3.2.3 NRMSE/NRMSD, CV_MRSE
 - 3.2.4 SDR
 - 3.2.5 Accuracy, R2

4. Puesta en práctica: aprendizaje supervisado

- 4.1 Preparación
- 4.2 Comprobar las hipótesis, p_value
 - 4.2.1 Análisis gráfico interactivo con iplots
 - 4.2.2 Test de Breush-Pagan y zoom sobre p_value
- 4.3 Creación de un modelo (regresión lineal múltiple)
- 4.4 Establecer una predicción
- 4.5 Estudio de los resultados y representación gráfica
- 4.6 Indicadores habituales - cálculos
- 4.7 Estudio del modelo lineal generado
- 4.8 Conclusión sobre el modelo lineal
- 4.9 Uso de un modelo «Random Forest»

Técnicas y algoritmos imprescindibles

1. Construir la caja de herramientas

2. Representación gráfica de los datos

2.1 Un gráfico «simple»

2.2 Histogramas avanzados

 2.2.1 Distribución multiclase

 2.2.2 Mezcla de varias distribuciones por clase

 2.2.3 Visualización de la densidad de una distribución

 2.2.4 Otra mezcla por clase

 2.2.5 Una variable, pero un histograma para cada clase

 2.2.6 Gráfico con una densidad por clase

2.3 Diagrama de pares y de facetas

 2.3.1 Diagrama por pares, versión simple

 2.3.2 Clases en configuración XOR

 2.3.3 Diagrama por pares con «factores»

 2.3.4 Facetas y escala logarítmica

3. Machine learning: prácticas corrientes

3.1 Recorrido teórico acelerado

 3.1.1 Linealidad

 3.1.2 Errores in y out, noción de dimensión VC

 3.1.3 Hiperplanos, separabilidad con márgenes

 3.1.4 Kernel Trick, núcleos, transformaciones, feature space

 3.1.5 Problemas de la regresión: introducción a la regularización

3.2 Práctica por práctica

 3.2.1 Cross validation: k-fold CV

 3.2.2 Naive Bayes

 3.2.3 C4.5 y C5.0

 3.2.4 Support Vector Machines (SVM)

 3.2.5 Clusterización, k-means

4. ¿ Dónde nos encontramos en nuestro aprendizaje ?

4.1 Sus conocimientos operacionales

4.2 Las posibles lagunas que es preciso cubrir ahora

Marco metodológico del data scientist

1. El problema metodológico a nivel del proyecto

- 1.1 La expresión de una necesidad
- 1.2 La gestión del proyecto

2. El ciclo interno de data sciences

- 2.1 Revisión detallada del problema planteado
- 2.2 Trabajos previos sobre los datos
 - 2.2.1 Exigencias sobre los datos
 - 2.2.2 Recogida, limpieza y comprensión de los datos
- 2.3 El ciclo de modelado
 - 2.3.1 Feature engineering
 - 2.3.2 Modelado y evaluación
 - 2.3.3 Escoger el mejor modelo
 - 2.3.4 Test, interpretación y confrontación con negocio
- 2.4 Preparación de la industrialización y despliegue
- 2.5 Preparación de las siguientes iteraciones
 - 2.5.1 Elementos que es preciso tener en cuenta
 - 2.5.2 Documentación gestionada por los data scientists

3. Complementos metodológicos

- 3.1 Clasificar sus objetivos
- 3.2 Trucos y argucias

Procesamiento del lenguaje natural

1. Definición del problema

2. Análisis semántico latente y SVD

- 2.1 Aspectos teóricos
 - 2.1.1 SVD: generalidades
 - 2.1.2 Una justificación de la descomposición SVD

- 2.1.3 SVD en el contexto LSA
- 2.1.4 Interpretación
- 2.1.5 Alternativa no lineal, Isomap (MDS, geodésico, variedad, manifold)
- 2.2 Puesta en práctica
 - 2.2.1 Inicialización
 - 2.2.2 En el núcleo de LSA
 - 2.2.3 Resultados
 - 2.2.4 Manipulaciones, interpretaciones recreativas y no fundadas

Grafos y redes

1. Introducción

2. Primeros pasos

- 2.1 Algunas nociones y notaciones complementarias básicas
- 2.2 Manipulaciones simples de grafos con R
- 2.3 Estructura de los grafos

3. Grafos y redes (sociales)

- 3.1 Análisis de las redes sociales: conceptos básicos
- 3.2 Puesta en práctica
- 3.3 Detección de comunidades

Otros problemas, otras soluciones

1. Series temporales

- 1.1 Introducción
- 1.2 Modelo estacionario
 - 1.2.1 Proceso estacionario: los fundamentos
 - 1.2.2 Proceso autorregresivo AR: ir más lejos
 - 1.2.3 Consideraciones (muy) útiles
- 1.3 Procesos no estacionarios

- 1.3.1 El modelo ARIMA
- 1.3.2 Procesos estacionales: SARIMA
- 1.3.3 Modelos ARCH y GARCH
- 1.3.4 Convolución y filtros lineales
- 1.4 Puesta en práctica
 - 1.4.1 Los fundamentos de la manipulación de las series temporales en R
 - 1.4.2 Estudio de las series temporales
 - 1.4.3 Predicciones sobre ARIMA (AR MA SARIMA)
- 1.5 Minibestiario ARIMA

2. Sistemas difusos

3. Enjambre (swarm)

- 3.1 Swarm y optimización: el algoritmo PSO
 - 3.1.1 Presentación de PSO
 - 3.1.2 Descripción de PSO
- 3.2 Puesta en práctica de PSO

Feature Engineering

1. Feature Engineering, los fundamentos

- 1.1 Definición del problema
- 1.2 Sobre qué hay que estar muy atento
 - 1.2.1 La calidad de la distribución
 - 1.2.2 La naturaleza de las features
- 1.3 Dominar la dimensionalidad
- 1.4 Una solución práctica: el PCA
- 1.5 Un ejemplo simple del uso del PCA
- 1.6 Los valores desconocidos y las features mal condicionadas
- 1.7 Creación de nuevas features
- 1.8 A modo de conclusión

2. PCA clásico, elementos matemáticos

3. Reducción de los datos (data reduction)

4. Reducción de la dimensionalidad y entropía

4.1 Descripción teórica del problema

4.2 Implementación en R y discusión

Complementos útiles

1. GAM: generalización de LM/GLM

2. Manipulación de imágenes

2.1 Creación, visualización, lectura y escritura de imágenes

2.2 Transformaciones de imágenes

2.2.1 Ejemplos de manipulación del color y de las intensidades

2.2.2 Ejemplos de manipulación de la geometría de la imagen

2.2.3 Aplicación de filtros sobre las imágenes

3. Cómo crear una muestra: LHS (hipercubo latino)

4. Trabajar sobre datos espaciales

4.1 Variograma

4.1.1 Campo y variable regionalizada

4.1.2 Determinación del variograma

4.2 Krigeage (kriging)

4.2.1 La teoría, brevemente

4.2.2 Implementación en R

5. Buenas prácticas útiles

5.1 Trazar una curva ROC

5.2 Una red neuronal (primeros pasos hacia el deeplearning)

6. Gradient Boosting y Generalized Boosted Regression

- 6.1 Los grandes principios
- 6.2 Los parámetros y los usos (paquete GBM)
 - 6.2.1 Covarianza
 - 6.2.2 Loss
 - 6.2.3 Optimización del algoritmo
- 6.3 Puesta en práctica

Anexos

1. Acerca de la utilidad de estos anexos

2. Fórmulas

3. Estrategias según la naturaleza de los datos

- 3.1 Recuentos
- 3.2 Proporciones
- 3.3 Variable de respuesta binaria
- 3.4 Datos que inducen un modelo mixto (mixed effect)
- 3.5 Datos espaciales
- 3.6 Grafos
- 3.7 Análisis de supervivencia (survival analysis)

4. Filtros (sobre imágenes)

5. Distancias

6. Trucos y pequeños consejos

- 6.1 Acerca de los tests
- 6.2 Gestión de las variables
- 6.3 Análisis y manipulación de resultados
 - 6.3.1 Residuos

6.3.2 Manipulación de los modelos

7. Paquetes y temas para estudiar

- 7.1 Creación de gráficos JavaScript con R
- 7.2 Crear uniones como en SQL
- 7.3 Reglas de asociación
- 7.4 Exportar un modelo
- 7.5 Tensores
- 7.6 SVM para la detección de novedades (novelty detection)

8. Vocabulario y «tricks of the trade»

- 8.1 Complementos sobre las bases del machine learning
- 8.2 Complementos sobre los aspectos bayesianos
- 8.3 Vocabulario (en inglés) de los modelos gaussianos

9. Algoritmos para estudiar

10. Algunas formulaciones de álgebra lineal

Conclusión

índice

Información

Data Scientist y lenguaje R

Guía de autoformación para el uso de Big Data

Todos los expertos se ponen de acuerdo en afirmar que el 90 % de los usos del Big Data provienen del uso de las data sciences. El objetivo de este libro es proponer una **formación completa y operacional** en las data sciences que permita producir **soluciones mediante el uso del lenguaje R**.

De este modo, el autor plantea un recorrido didáctico y profesional que, sin más requisito previo que un nivel de enseñanza secundaria en matemáticas y una gran curiosidad, permita al lector:

- integrarse en un equipo de data scientists,
- abordar artículos de investigación con un alto nivel en matemáticas,
- llegado el caso, desarrollar en lenguaje R, incluso nuevos algoritmos y producir bonitos gráficos,
- o simplemente gestionar un equipo de proyecto en el que trabajen data scientists, siendo capaces de dialogar con ellos de manera eficaz.

El libro no se limita a los algoritmos del "machine learning", sino que aborda diversos asuntos importantes como el **procesamiento del lenguaje natural**, las **series temporales**, la **lógica difusa**, la **manipulación de imágenes**.

La dinámica del libro ayuda al lector paso a paso en su **descubrimiento de las data sciences** y en el desarrollo de sus competencias teóricas y prácticas. El profesional descubrirá a su vez **muchas buenas prácticas** que puede adquirir y el gestor podrá surfear el libro tras haber leído con atención el **bestiario de las data sciences** de la introducción, que sin inexactitud o excesiva banalización presenta el tema ahorrando en aspectos matemáticos o en formalismos disusivos.

Los programas en R descritos en el libro están accesibles **para su descarga** en esta página y pueden ejecutarse paso a paso.

Los capítulos del libro:

Introducción – Primeros pasos con R – Dominar los fundamentos – Técnicas y algoritmos imprescindibles – Marco metodológico del data scientist – Procesamiento del lenguaje natural – Grafos y redes – Otros problemas, otras soluciones – Feature Engineering – Complementos útiles – Anexos – Conclusión

Henri LAUDE

Henri LAUDE es un reconocido profesional de las Computer Sciences. Ha afrontado numerosos proyectos de I+D relativos a las data sciences, vinculados a la inteligencia económica, a la IA, a los riesgos, a la detección de fraude y a la ciberseguridad. Presidente de la APIEC (Asociación para la Promoción de la Inteligencia Económica), Chief Data Scientist y fundador del Laboratorio de Data Sciences y Big Data BlueDsX del grupo BlueSoft, es también cofundador de la startup Advanced Research Partners, donde dirige el desarrollo de algoritmos muy novedosos implementados en R y en Python sobre plataformas Hadoop y Spark.

Es un apasionado de las data sciences y pretende transformar a todos los lectores de su libro en data geeks o, al menos, dotarlos de la suficiente cultura general en este tema para que la utilicen en alcanzar sus objetivos.

Introducción

Data scientist, una disciplina de moda

El grupo de LinkedIn llamado «Data Scientists» comprende a más de 20 000 profesionales.

La mañana en la que el autor escribe estas líneas, una consulta desde su cuenta de LinkedIn le muestra que 49 000 personas de su red directa reivindican el título de «data scientist».

Esa misma mañana, el sitio Indeed anuncia 1 300 ofertas de empleo de data scientist en New York, 4 000 ofertas relacionadas con el Big Data, frente a 1 000 ofertas de trader.

Como apasionados de las data sciences, sabemos que estas cifras no significan nada si no se ponen en perspectiva. Por lo tanto, estas cifras no son ajenas al entusiasmo actual de los periodistas por el Big Data y las data sciences.

Desde hace más de un cuarto de siglo, son numerosos los expertos que manipulan de manera cotidiana los conceptos y los algoritmos que están en el núcleo de las prácticas actuales de las nuevas data sciences.

Estas prácticas a menudo nacen de la inteligencia artificial y de la experiencia operacional de los «quant» que trabajan después como traders, de los «actuarios» que operan en el mundo de los seguros, de los biólogos o, simplemente, de todos los investigadores que deben emitir y cualificar estadísticamente hipótesis antes de difundirlas entre la comunidad científica.

El Big Data ha cambiado el dato popularizando los usos de los data scientists. Durante mucho tiempo, el problema que se planteaba residía a menudo en la poca cantidad de muestras a nuestra disposición. ¡En la actualidad, dudamos en ocasiones de nuestra capacidad para procesar de manera útil la cantidad masiva de datos que podrían proporcionarse para el análisis!

Las data sciences

Las ciencias de datos (del inglés *data sciences*) comportan muchos aspectos y afectan a campos de estudio muy variados.

Encontramos técnicas que permiten clasificar y ordenar objetos de nuestro alrededor, predecir eventos, reconocer patrones o secuencias de información, identificar reglas subyacentes a los datos, analizar series temporales, interpretar textos, detectar valores anormales o señales débiles y analizar grafos industriales o redes sociales.

El desarrollo de las data sciences ha encontrado un nuevo auge en las prácticas del Big Data, aunque sería incorrecto reducirlas a este contexto.

En efecto, la experiencia utilizada proviene de disciplinas relativamente antiguas e interconectadas, como son la inteligencia artificial, la búsqueda operacional, la estadística y la lingüística.

Evidentemente, estas disciplinas utilizan unas matemáticas y una algoritmia potentes.

En muchas obras de calidad, las data sciences parecen reducirse a las cautivadoras técnicas del «data-mining» y del «machine learning», es decir, a la capacidad de establecer una clasificación o una predicción mediante un ordenador. Estos aspectos se cubrirán sobradamente en los siguientes capítulos.

Sin embargo, otras tecnologías y otras prácticas importantes encontrarán su lugar en este libro, como por ejemplo los algoritmos por enjambre de partículas, que, de manera análoga a los procesos de la naturaleza, aportan resultados operacionales brillantes (por ejemplo, para encontrar un recurso escondido en una red no exhaustivamente conocida).

En la actualidad, las data sciences encuentran sus aplicaciones en dominios científicos y económicos muy variados, tales como la astronomía, la ecología, las finanzas e incluso la acción comercial.

Las data sciences también permiten responder a problemáticas en otros campos de búsqueda dentro de las ciencias humanas (estudio de las redes sociales) o de la seguridad informática.

El Big Data

El Big Data aporta muchas innovaciones mediante el uso de las data sciences vinculadas a su capacidad innata para procesar volúmenes y naturalezas de datos que parecían inaccesibles a nuestras calculadoras.

Este conjunto de tecnologías no se caracteriza únicamente por el volumen de los datos, sino también por su naturaleza, en ocasiones imprecisa, incompleta, que adolecía de errores variados y de fenómenos efímeros.

Los datos tratados pueden tomar formas muy diversas, comprendiendo un desorden de bases de datos relacionales o no, archivos estructurados o no, datos textuales en diversos idiomas, datos multimedia...

Los datos incluyen diversos vínculos temporales, lógicos o de causalidad. Hablamos de series temporales (por ejemplo: cursos de bolsa), de grafos (por ejemplo: relaciones en las redes sociales), de secuencias (por ejemplo: los genes), de reglas (por ejemplo: «si llueve y es sábado entonces voy al cine»)...

El Big Data representa el conjunto de técnicas (informáticas) que nos van a permitir gestionar, explotar y realizar distintos usos eficaces de estos datos voluminosos y terriblemente complejos.

De este modo, las data sciences permiten explotar, manipular estos datos, comprender, prever y establecer usos que habrían sido imposibles sin ellas, debido a la complejidad y al volumen que hay que procesar.

Funcionalmente hablando, el Big Data no aporta una ruptura real respecto a las técnicas antiguas, salvo por los usos que ha hecho posibles mediante las data sciences. Sin embargo, la manera de implementar estas técnicas contrastadas se ha visto marcada profundamente por los volúmenes que es preciso procesar, la velocidad de los cambios de estado de la información y variedad. Es lo que los consultores suelen denominar las «3V».

En términos de implementación, técnicas como los sistemas expertos, los métodos estadísticos descriptivos e inferenciales, el «data-mining», las tecnologías decisionales de «Business Intelligence» y las arquitecturas que convergen en datawarehouses se enfrentan a nuevos retos.

La comprensión de estos problemas no puede realizarse de forma nativa. Debido a esta complejidad, paradójicamente, los usuarios son cada vez más propensos a exigir interfaces sencillas e intuitivas. Ambas constantes llevan a un aumento exponencial de la inteligencia que hay que introducir entre los datos y los usuarios finales. A través de esta exigencia, las tecnologías del Big Data y del mundo digital se ven obligadas a converger hacia soluciones integradas, mientras que sus paradigmas de base divergen.

La dinámica de este libro

1. Nuestros objetivos

Una de las ambiciones de este libro, en el caso de una lectura exhaustiva realizando cada ejemplo, es transmitir al lector las competencias de un data scientist «junior», para hacer que sea totalmente operacional.

Este data scientist, formado a partir de este libro y de sus búsquedas complementarias eventuales sobre la Web, será también capaz de implementar modelos de predicción eficaces mediante el uso del lenguaje R y de sus paquetes. Utilizará una gama limitada de técnicas, pero lo hará con conocimiento y tendrá una idea clara de lo que debe abordar para continuar con su recorrido iniciático. Dispondrá del vocabulario y de los conceptos básicos para poder dialogar con sus colegas y para abordar la lectura de artículos de investigación que no dejarán de aumentar a lo largo de su práctica futura. Podrá equipararse a sus colegas compitiendo en el sitio Kaggle.com y resultar útil en un equipo.

Cada capítulo de este libro es independiente. No es indispensable haber asimilado todo el contenido de los capítulos previos para poder sacar provecho. Para facilitar una lectura no lineal de este libro, no hemos dudado en introducir ligeras redundancias.

Si no quiere aprender R, no se preocupe; conténtese con leer el código y realice una fuerte criba en el capítulo Primeros pasos con R. Su viaje no será 100 % operacional, pero podrá formarse una opinión y reflexiones muy sólidas acerca del universo de las data sciences.

El libro no tiene vocación de resultar académico, ni tampoco es una demostración estructurada que va de lo general a lo particular o de conceptos básicos a conceptos muy avanzados. Evidentemente, el libro posee una dinámica propia, pero no es escolar o demostrativa de ninguna manera.

El libro se ha concebido como un objeto didáctico donde la intuición del lector y la práctica de la interpretación de los datos y de los gráficos le ayudarán a descubrir paso a paso el problema que se le plantea. Nuestra pretensión es que jamás deje de lado la acción, pero también evitar encantarse en demostraciones intermedias que están, por otro lado, perfectamente documentadas en la literatura disponible.

Si es manager, jefe de proyecto, aprendiz o curioso pero no implicado en un ciclo de autoformación en las data sciences, aborde los capítulos como mejor le plazca para adquirir los suficientes elementos de la cultura general sobre el asunto para alcanzar sus objetivos.

La lectura del libro no requiere poseer conocimientos matemáticos de un nivel superior al de la formación secundaria. En contraposición, introduciremos distintos objetos de matemáticas superior para permitir una mejor comprensión de los textos de investigación.

Muchas fuentes de información acerca de estos asuntos están escritos en inglés, y también se produce mucha mezcla de términos en inglés y en castellano. En el libro abundan términos anglosajones, pues nuestro objetivo es facilitar la lectura de numerosas publicaciones y artículos disponibles en inglés.

2. La estructura del libro

Este manual se organiza en capítulos de longitud variable, pues tienen objetivos muy diferentes.

Podemos escoger el dejarnos guiar abordando el libro de cuatro maneras diferentes:

- De manera lineal, de la A a la Z, para aprovechar la didáctica del libro.
- Ir picoteando entre capítulos, para su placer o para buscar información precisa relacionada con un problema práctico.

- Un recorrido «profesionalización rápida» para los «debutantes».
- Un recorrido «imaginación e innovación» para los data scientists que estén buscando nuevas ideas o información complementaria a sus prácticas habituales.

a. Los dos recorridos complementarios

Capítulos 1, 2, 3, 9 y anexos: profesionalización rápida

El primer capítulo (el presente) no tiene como único objetivo presentar el libro. A continuación aparece un compendio de conceptos, representaciones y ciertos elementos de vocabulario que resultan útiles en la profesión. Este capítulo no busca explotar su capacidad de análisis o de comprensión más fina, sino interpelar a su percepción y su intuición.

El segundo capítulo (Primeros pasos con R) tiene como objetivo permitir el aprendizaje de los distintos aspectos del lenguaje R, introduciendo aspectos relacionados con las probabilidades, con las estadísticas descriptivas y el análisis de datos en general mediante R. Si nunca ha programado, este capítulo podrá también servirle como introducción al arte de la programación. Propone también una iniciación a la programación orientada a objetos (en R, evidentemente). Al final de este capítulo tendrá la ocasión de crear un primer modelo predictivo y evaluar su pertinencia, este modelo estará basado en la técnica de los árboles de decisión. Este capítulo gira en torno a la práctica del lenguaje, pero también permite descubrir algunas técnicas del data scientist a lo largo del camino.

El tercer capítulo (Dominar los fundamentos) es el núcleo de este libro. Introduce conceptos matemáticos básicos y el sentido de muchas notaciones que encontraremos en la literatura. Plantea el problema de la predicción. Para terminar, propone una puesta en práctica de los conceptos, mediante la comparación visual de un modelo realizado mediante dos algoritmos distintos (regresión lineal multivariante y Random Forest). Para formarse de manera rápida y operacional en machine learning mediante R, en caso de que no se disponga de mucho tiempo, hay que abordar con prioridad los capítulos 1, 2, 3 y 9 y eventualmente el cuarto (Técnicas y algoritmos imprescindibles).

El noveno y penúltimo capítulo (Feature Engineering) completa esta profesionalización. Aborda uno de los principales retos de esta disciplina: el «feature engineering», es decir, el arte de extraer y transformar datos para optimizar la eficacia de sus clasificaciones o de sus predicciones.

Capítulos 1, 4, 5, 6, 7, 8 y 10: «de la imaginación, siempre de la imaginación»

Estos capítulos proporcionan las claves para imaginar soluciones y prácticas mejor adaptadas a numerosos problemas de las data sciences.

Le recomendamos, de nuevo, comenzar en primer lugar por nuestro compendio de conceptos sobre data sciences del presente capítulo.

El cuarto capítulo (Técnicas y algoritmos imprescindibles) aporta información sobre diversas prácticas de representación de los datos, sobre algoritmos y técnicas potentes que utilizan de manera cotidiana los profesionales. Su objetivo es extender su campo de visión sobre asuntos bien conocidos en las data sciences y generarle deseos de explorar las técnicas de machine learning. La segunda parte de este capítulo presenta una formación acelerada sobre las bases teóricas del machine learning. Evidentemente, la formulación es un poco más matemática, lo cual puede resultar en cierto modo reconfortante a aquellos lectores que posean una formación científica.

El quinto capítulo (Marco metodológico del data scientist) formaliza el conjunto de técnicas del machine learning en un proceso constructivo y convergente. Constituye el capítulo metodológico.

El sexto capítulo (Procesamiento del lenguaje natural) abre una ventana a las técnicas de procesamiento del lenguaje natural para que cada uno pueda abordar el gran potencial de los datos disponibles de este tipo.

El séptimo capítulo (Grafos y redes) se centra en los grafos y las redes sociales, como complemento al sexto capítulo, para abordar el mundo de la Web 3.0.

El octavo capítulo (Otros problemas, otras soluciones) abre el campo de las posibilidades y ofrece pistas de estudio que sobrepasan el campo clásico del machine learning «industrial». Este capítulo será una etapa en la creación de su propia esfera de innovación personal. Se realizará un zoom sobre las series temporales y los sistemas de lógica difusa (fuzzy).

El décimo y último capítulo (Complementos útiles) propone elementos complementarios, en ocasiones periféricos al mundo cotidiano de los data scientists, pero que hemos considerado interesantes para su éxito.

b. Recursos complementarios

Este libro trata de ser «autocontenido». Se ha construido de manera que no tenga que referirse a otras fuentes para comprender su contenido.

Sin embargo, a pesar de toda la atención que se haya podido prestar durante la redacción de este libro, cada uno construye su comprensión sobre bases diferentes y, en ocasiones, se verá conducido a consultar otras fuentes para perfeccionar su conocimiento en los asuntos abordados.

Preste atención, pues el uso asiduo de la Wikipedia puede generar cierta confusión sobre asuntos precisos e incitar a un proceso de «surfeo» permanente [y, en ciertos casos, a una procrastinación estudiantil grave ;-)].

A diferencia de otros libros, le proponemos desde esta misma introducción una selección reducida de fuentes complementarias, cualificadas y coherentes, para que pueda encontrar rápidamente la información complementaria y prosiga con su lectura.

Correspondiente a machine learning y R:

- <http://www.jmlr.org/>: aquí hay... todo lo importante, con una búsqueda «full text» y artículos de calidad, en inglés. ¡Debe utilizarse sin moderación!
- <http://finzi.psych.upenn.edu/nmz.html>: búsqueda sobre toda la documentación de paquetes R situados en tres sitios de referencia, incluida la base de datos CRAN, <http://cran.r-project.org/>.

Correspondiente a las matemáticas:

- <https://es.khanacademy.org/math/>: cubre todos los niveles escolares tras la educación secundaria. Su contenido es justo, preciso, coherente y de calidad. No encontrará nada relacionado con matemáticas de ingeniería.
- <http://mathworld.wolfram.com/>: matemáticas de buen nivel, pragmático, articulado sobre conceptos que se utilizan habitualmente.
- https://wwwENCYCLOPEDIAofmath.org/index.php/Main_Page: enciclopedia Springer, en inglés, un «must» con una búsqueda por palabras clave eficaz.

Correspondiente a problemas cotidianos de informática:

- <http://www.computerhope.com/>: para no quedarse bloqueado debido a un problema trivial en Windows o Linux.

Correspondiente... al resto!

- <https://stackexchange.com/sites>: utilice la búsqueda full text desde la raíz de este conjunto de sitios, o plantee preguntas en el sitio temático correspondiente. Es el sitio «definitivo» de preguntas/respuestas.

Pequeño bestiario de las data sciences

Las descripciones e imágenes siguientes tienen como único objetivo familiarizarse con el vocabulario y las representaciones que utilizan a menudo los data scientists. No busque cómo construir una estructura lógica compleja a partir de esta información; conténtese con enriquecer el número de asociaciones de ideas entre sus descubrimientos y sus conocimientos previos.

1. Los fundamentos

a. Aprendizaje y clasificación

Una máquina «aprende» cuando su configuración evoluciona en función de las circunstancias.

El aprendizaje supervisado es la aplicación más directa de este fenómeno. Tras haber recopilado información, se divide cada observación en dos partes, una llamada explicativa y otra explicada. Se escoge, a continuación, una mecánica de cálculo que parece permitir deducir las variables explicadas a partir de las variables explicativas. La afinación de este modelo consiste en encontrar la configuración que resulte más eficaz. Este proceso de aprendizaje se llama «supervisado», puesto que está bajo el control de las variables explicadas del conjunto de datos de entrenamiento.

Cuando se recogen nuevas observaciones que no posean valores conocidos para las variables explicativas, basta con aplicar el modelo con sus parámetros para obtener lo que se denomina a su vez una predicción o, simplemente, una estimación.

Las demás formas de aprendizaje no tienen como objetivo predecir cualquier cosa, sino que proporcionan estimaciones de patrones (es decir, esquemas identificables y repetitivos) que no aparecen a simple vista.

En inglés hablamos de «machine learning». Las variables explicativas se denominan de varias maneras: «features», «atributos» o «covariables». Las variables explicativas, por su parte, se denominan «response variables».

Cuando intervenimos en un proceso de aprendizaje introduciendo una información que no estaba disponible al principio, esto se denomina refuerzo del aprendizaje.

Cuando la variable explicada, y por lo tanto predicha, es una clase (0/1, un color, un género...), hablamos simplemente de clasificación.

No debe confundirse la técnica de aprendizaje llamada «clasificación», que identifica el aprendizaje supervisado, con el descubrimiento de clases antes desconocidas («clusterisation» en inglés) que no está supervisada por una definición de clases a priori.

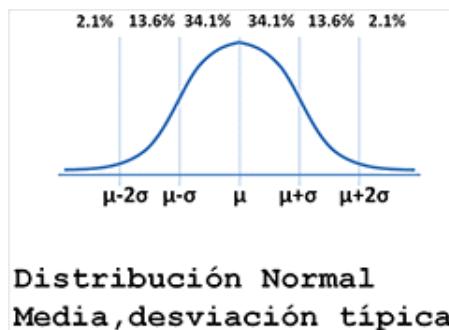
b. Pequeño vocabulario gráfico del machine learning

Los data scientists comparten ciertas representaciones mentales. Ciertos esquemas visuales son comunes a una gran mayoría de profesionales, pues se trata de representaciones que les han acompañado durante su formación.

Estas representaciones visuales y el vocabulario asociado representan en ocasiones todo lo que le queda a una persona que ha estudiado las data sciences sin ponerlas en práctica, o a un alumno distraído que no ha perfeccionado sus clases. Como con el aprendizaje de los fundamentos de un lenguaje, mediante un método global, antes incluso de comprender, cuestionar o tratar de retener cualquier cosa, hay que familiarizarse gradualmente con los conceptos que se van a asimilar. Le proponemos la siguiente experiencia: observe atentamente y trate de impregnarse de los 25 elementos visuales del capítulo, evitando cometer interpretaciones erróneas acerca de su significado.

Si algunas fórmulas matemáticas se le escapan, no importa; se abordarán más adelante en el libro. Le mantendrán a flote a lo largo de la lectura de este libro, y también en sus demás lecturas.

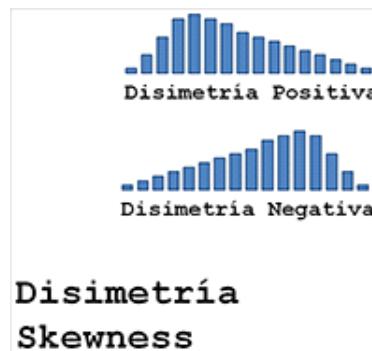
Johann Carl Friedrich Gauss (1777-1855) ha impuesto su nombre en lo que conocemos como distribución normal o curva de Gauss. Cuando un fenómeno sigue una distribución gaussiana, si conocemos su media y su desviación típica, podemos calcular fácilmente la probabilidad de que un individuo de la población posea un valor determinado.



Las poblaciones comprenden, en ocasiones, mezclas propias de diversas clases que siguen distribuciones normales de manera individual. Existen algoritmos, como el algoritmo EM (*Expectation-Maximization*), que permiten extraer las clases originales. Este algoritmo se expresa en su forma actual desde 1977 gracias a Arthur Dempster, Nan Laird y Donald Rubin.

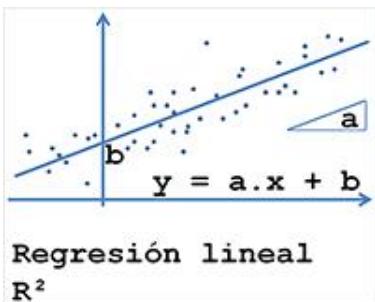


No todas las distribuciones son simétricas, incluso cuando no son mezclas gaussianas. Existe una manera de caracterizar la disimetría mediante un indicador llamado *skewness* por los ingleses.

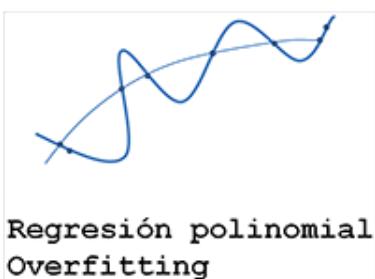


La determinación de la pertenencia a una clase se concreta por el hecho de etiquetar sus muestras con el nombre de la clase: clase_1, clase_2... Esto produce una visión muestreada del mundo (llamada discreta). Para predecir no una clase, sino un valor continuo, hay que utilizar una mecánica llamada de regresión. La versión antigua de este

método se denomina regresión lineal (de Ruder Josip Boškovic, mediados del siglo XVIII), a la que se asocia a menudo una noción de coeficiente de correlación, o su cuadrado, el coeficiente de determinación, para concretar el nivel de calidad de la aproximación realizada.

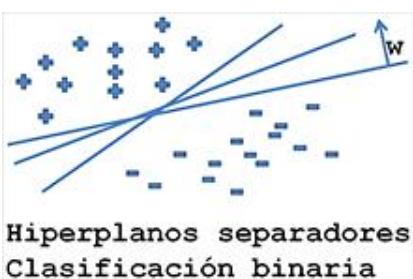


¡Resulta difícil aproximar una nube de puntos que no está alineada sobre una línea recta mediante una regresión lineal! El repertorio de los data scientists incluye numerosas herramientas que permiten resolver este problema. La regresión polinomial es una de ellas. Observando sus efectos se descubre el gran miedo de la profesión, el «sobreaprendizaje», u *overfitting* en inglés. Su efecto es desastroso; en lugar de encontrar una aproximación correcta, el hecho de querer aproximarnos lo máximo posible a todos los puntos de una nube hace que el modelo resulte inadecuado y poco generalizable para la predicción de un nuevo punto. El peligro del overfitting no es inherente a la regresión polinomial, sino que salta a la vista cuando se utiliza mal esta herramienta.

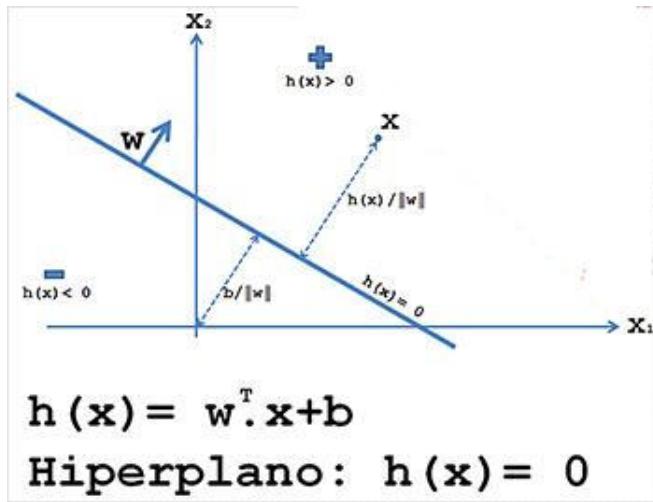


El hecho de querer hacer pasar la curva por los 2 x 2 puntos extremos de la nube de puntos ha deformado completamente la aproximación.

El problema de la regresión es menos didáctico que el problema de la clasificación. Uno y otro están originalmente vinculados mediante la noción de recta, o su generalización en el espacio de más dimensiones en lo que denominamos hiperplanos. En efecto, el problema de clasificación más sencillo de representar toma la forma de dos clases separadas por una línea recta o un hiperplano. El objetivo de nuestros algoritmos es, por lo tanto, encontrar el hiperplano que mejor separe ambas clases. Resulta cómodo caracterizar un hiperplano mediante un vector perpendicular a él, como se representa en el siguiente esquema.



Para los matemáticos, el siguiente esquema no supondrá problema alguno. Para los demás, cobrará sentido poco a poco conforme se avance en este libro y deberán volver a él en ocasiones para comprender sus aspectos más matemáticos.



Globalmente, este esquema representa un hiperplano de 2 dimensiones, es decir, una línea recta. Se define la ecuación general de los hiperplanos, así como la forma de utilizarla para separar la clase positiva de la clase negativa. Por motivos de comodidad y en caso de duda, se indica la distancia del hiperplano al origen, así como la distancia de un punto cualquiera con el hiperplano. Ambas distancias se definen en función del módulo (la norma) del vector perpendicular al hiperplano que contiene los parámetros de este.

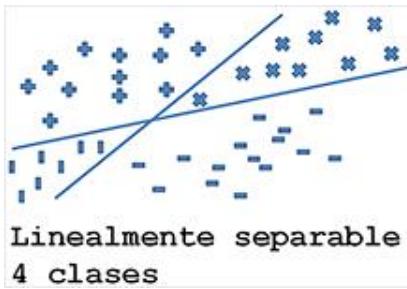
Para determinar estos hiperplanos, hay que utilizar diversos algoritmos. Entre ellos, muchos tienen un punto en común: en uno u otro momento hay que encontrar el mínimo (o máximo) de una función que sirve para controlar el algoritmo, por ejemplo una función que mida la tasa de error. Uno de los métodos empleados con mayor frecuencia es el método de «descenso por gradiente», que posee numerosas versiones y muchas aplicaciones.



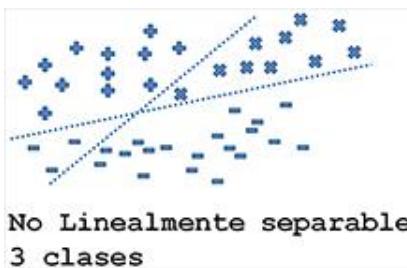
Estos métodos son sensibles a las condiciones iniciales, es decir, a la manera en la que se los inicializa. También son sensibles a la topología del problema planteado, es decir, a las características de las funciones que se desea optimizar. La noción utilizada más habitualmente para determinar si una función va a ser fácil de optimizar o no es la noción de convexidad. Para determinar si una forma cerrada es convexa, hay que imaginar hiperplanos tangentes en cada punto de su frontera donde es posible definir un hiperplano tangente. Si alguno de los distintos hiperplanos no interseca con el interior de la forma, entonces es convexa.



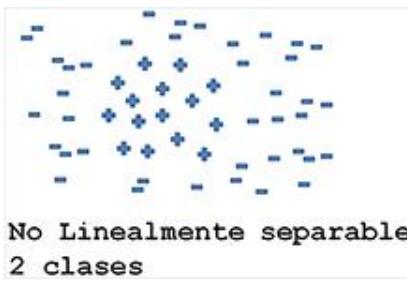
Las configuraciones en las que hay que separar varias clases pueden abordarse creando varios hiperplanos. En función de las topologías, será posible o no separar las clases de manera lineal.



En ciertos casos, aparentemente sencillos, no existe una manera trivial de separar el espacio mediante hiperplanos completos.

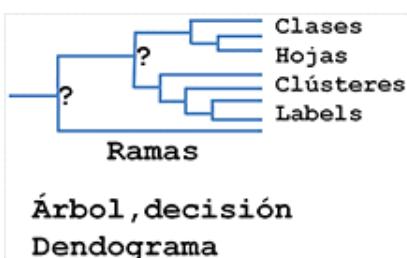


Las clases, aunque estén bien formadas, no siempre son separables por estructuras lineales; para convencerse de ello, basta con visualizar el siguiente esquema:

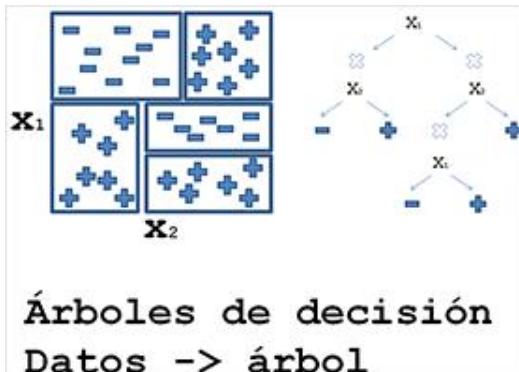


En este caso, pero también en casos linealmente separables, la clasificación sigue siendo posible introduciendo otros algoritmos. Una primera manera de proceder es apoyarse en algoritmos que generan árboles de decisión sobre separaciones parciales del espacio.

La representación mediante dendogramas, como el siguiente, se utiliza a menudo para describir clases «descubiertas por la máquina pero desconocidas previamente», llamadas clústeres. De este modo, es posible utilizarla para describir el árbol que permite separar observaciones acerca de las clases (labels) cuya existencia se conocía previamente (como ocurre en el caso que hemos abordado en los esquemas previos).



El siguiente esquema tiene como objetivo poner de relieve cómo un conjunto no separable de dos clases puede representarse fácilmente mediante un árbol de decisión.



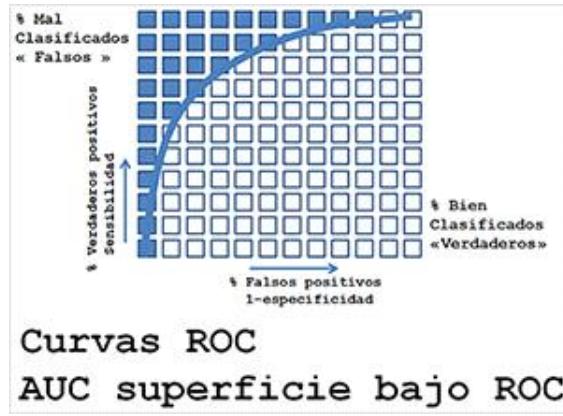
Es posible utilizar un algoritmo de clasificación en casos en los que ciertos puntos se ven ahogados dentro de otras clases, o en casos en los que las fronteras de clasificación resulten algo difusas. Además, los algoritmos y sus parámetros no siempre están optimizados para los datos sobre los que se va a trabajar. De hecho, una gran parte del trabajo de un data scientist consiste en mejorar su gama de algoritmos y parámetros para adaptarse a estos casos prácticos. La noción de overfitting se ha presentado más arriba: queriendo hacerlo demasiado bien, es posible perder el foco y sobredefinir el modelo. El concepto de matriz de confusión es uno de los más simples de implementar para percibir la tasa de error de clasificación. Basta con contar cómo se reparten las clases encontradas respecto a las clases reales.

		Realidad	
		-	+
Predicción	-	-	-
	Verd.	-	Falsos
	+	Falsos	Verd.

Matriz de confusión
Errores de clasificación

La observación de esta matriz pone de relieve nociones sencillas e interesantes, como el número de «verdaderos negativos», de «falsos negativos», de «falsos positivos» y de «verdaderos positivos». La relación entre el número de todos los «verdaderos» dividido entre el número total de observaciones refleja la tasa de error total, se denomina *accuracy* en inglés.

A partir de esta información, existe una técnica de cálculo llamada ROC (*Receiver Operating Characteristic*), que se ha desarrollado en el marco de la interpretación de las señales de radar durante la Segunda Guerra Mundial para evitar un segundo «Pearl Harbor». Esquemáticamente, la curva ROC se construye haciendo variar una cantidad y anotando en un gráfico las parejas de valores correspondientes. Los dos miembros de la pareja son la tasa de falsos positivos (que se corresponde con la resta a 1 de lo que llamamos especificidad, reportada sobre el eje de las x), y la tasa de verdaderos positivos (llamada sensibilidad y reportada sobre el eje de las y). Cuanto mayor sea el valor situado debajo de esta curva, mejor será la calidad de la predicción. Resulta muy práctica para comparar el rendimiento de dos clasificaciones binarias (es decir, dos algoritmos asociados con sus respectivas parametrizaciones). La superficie debajo de la curva se denomina AUC (*Area Under the Curve*).



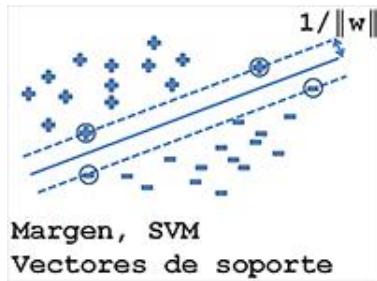
El lector atento puede que se plantea la siguiente reflexión: «¿Puedo realmente identificar estas cantidades de falsos y verdaderos cuando trabajo sobre datos antiguos que ya conozco?». Efectivamente, la noción de aprendizaje supone que se acepta la idea de que la experiencia adquirida aprovecha para realizar una decisión futura. Como el data scientist solo dispone de información pasada, la divide en dos o tres conjuntos de datos: los datos de entrenamiento (*training dataset*), los datos de validación (*validation dataset*) y los datos de prueba (*test dataset*). Los datos de entrenamiento permiten realizar un entrenamiento del modelo (icomo con los ejercicios finales de un curso!), los datos de validación permiten afinar el modelo y sus parámetros (icomo con los exámenes!) y por último los datos de prueba permiten asegurar un buen funcionamiento para todos de la eficacia del conjunto (icomo con la Selectividad!) antes de poner el modelo en producción (como en la vida...).

Así como aprenderse de memoria las respuestas a los ejercicios tiene un efecto nefasto en el rendimiento durante los exámenes, existe un punto de equilibrio entre un buen aprendizaje y un aprendizaje sobreajustado. Esto es lo que se muestra en el siguiente esquema:

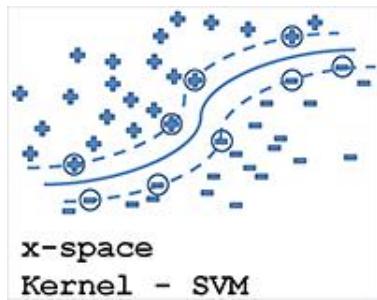


Un buen modelo (con su parametrización) no solo debe obtener un buen rendimiento en términos de tasa de error, sino que también debe ser resistente a pequeñas perturbaciones. Agregar o eliminar algunas observaciones no debería cambiar sus conclusiones. Esto significa que, por ejemplo, en el caso de la búsqueda de hiperplanos de separación, hay que encontrar la forma de que la determinación de estos no sea azarosa. Agregando un margen (*margin*) alrededor de los hiperplanos, el número de hiperplanos que separan ambas clases poco espaciadas disminuye rápidamente. Creando una tolerancia a partir del hecho de que algunas observaciones pueden estar mal clasificadas en torno o en el corredor así formado, aumenta la resistencia del conjunto frente a las perturbaciones (*soft margin*). Encontramos este procedimiento en un algoritmo muy potente llamado SVM (*Support Vector Machine*), cuyos conceptos se desarrollaron en los años 60 y 70 por investigadores y eminentes en machine learning (Vladimir Vapnik, Chervonenkis, A. Lerner Richard Duda y Peter Hart) y que ha sido objeto de publicaciones completas y de un título en los años 90.

El algoritmo selecciona un número de observaciones limitado, que se denomina «vectores de soporte» y que definen las fronteras entre el hiperplano embutido en sus márgenes.



Mediante un truco llamado *kernel trick* que permite aplicar de manera eficaz una transformación sobre las variables explicativas, llegamos a hacer separables linealmente nubes de clases que no lo eran; el nuevo espacio se denomina *z-space* o *feature space*. A continuación es posible reposicionar los vectores de soporte en su espacio original.



Todos estos cálculos se realizan utilizando distintos tipos de distancias o de semejanzas. La distancia más utilizada es la distancia euclídea, es decir, la noción de distancia tal y como la practicamos en el mundo real. Otra distancia célebre es la distancia de Manhattan, en alusión al desplazamiento en una gran ciudad con calles perpendiculares. La elección de una distancia adaptada a un problema influye bastante en la eficacia del algoritmo.

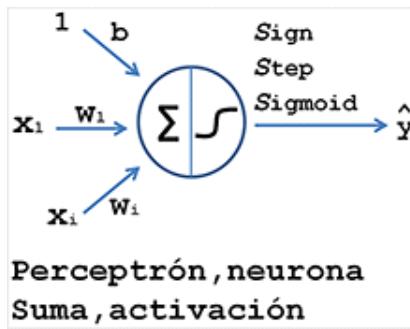


La clasificación también puede realizarse mediante otro tipo de algoritmo muy conocido por el gran público: las redes neuronales, que proceden de una analogía (simplista) con las redes neuronales de un cerebro biológico y que se basan en la modelización de una neurona mediante un algoritmo llamado perceptrón. El perceptrón lo creó en 1957 Frank Rosenblatt. Tras el entusiasmo por este algoritmo, la mala interpretación por la comunidad de los trabajos de Lee Minsky y Seymour Papert ha limitado la atención dedicada a estas tecnologías (e incluso ha afectado a los créditos para la inteligencia artificial).

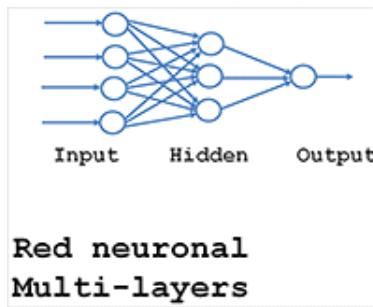
En la actualidad, estos algoritmos se utilizan a menudo y ha revivido el interés por un perceptrón más o menos aislado, pues su rendimiento es compatible con los volúmenes de Big Data.

La función que permite decidir la correcta clasificación de una observación en el seno de un perceptrón puede

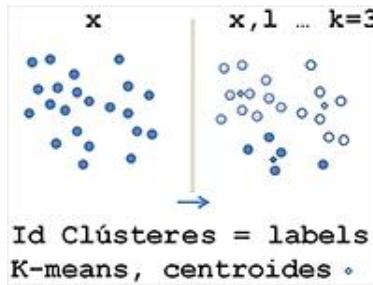
tomar diversas formas, entre ellas una forma en S muy característica (sigmoide).



Los perceptrones se articulan en capas cuyos resultados se inyectan en las siguientes. En ciertas arquitecturas, se reintroducen los resultados como entrada a los perceptrones situados previamente (*back propagation*). Cuanto más compleja sea la red, más difícil resultará su entrenamiento.

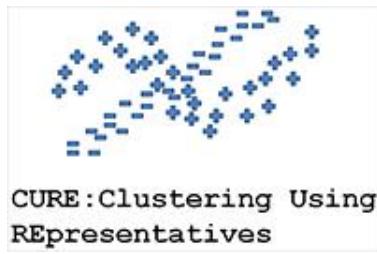


Otro algoritmo histórico y que se utiliza en la actualidad a menudo por los profesionales tiene como objeto la clusterización (búsqueda de clases latentes). Este algoritmo llamado k-means, cuyos fundamentos se establecieron en los años 60, determina mediante sucesivas iteraciones los k clústeres pedidos por los usuarios. La idea consiste en determinar clústeres cuyos puntos estén lo más cerca posible de los centroides de los clústeres.



Ciertas configuraciones son bastante más complejas que otras, existen numerosos algoritmos para otros tantos usos.

Por ejemplo, el siguiente algoritmo permite abordar el caso en que las nubes posean formas anidadas y se intersecten localmente.



No podemos concluir este rápido tour por las data sciences sin evocar el problema de las dimensiones. Cuantas más features haya, más difícil será disponer de un número suficiente de observaciones para aprender de todos los casos reales posibles y más vacías e incompletas serán las estructuras de tratamiento matricial que utilice el algoritmo eventualmente (*sparse matrix*).

Una de las técnicas que pueden utilizarse resulta del hecho de transformar ciertas matrices grandes mediante un algoritmo que produce matrices más pequeñas en las que podrán suprimirse ciertas columnas sin una gran pérdida de información.

La formalización de la siguiente ilustración es un poco más matemática. Pero no se preocupe, esta formalización se detallará en otro capítulo (cuando abordemos el tratamiento del lenguaje natural).

$$X = U \Sigma V^T$$

Descomposición de una matriz $X = U\Sigma V^T$

Descomposición en Valores singulares

Hemos terminado de explorar las principales representaciones gráficas del universo mental de los data scientists. Evidentemente, esta selección de 25 imágenes para resumir de forma gráfica este universo gráfico es totalmente empírica e incompleta, aunque esperamos que estas representaciones mentales le ayuden a sostener eficazmente sus objetivos.

c. Regresión

Cuando la variable explicada es continua, hablamos de regresión.

Regresión lineal

La regresión más conocida se denomina regresión lineal (*Linear regression Model*: LM en inglés). Se buscan los parámetros de una combinación lineal de variables explicativas cuyo resultado sea la variable explicada.

En una regresión lineal, los parámetros son coeficientes que multiplican a los valores de las variables explicativas y a los que se agrega una constante: la expresión del modelo resulta de la suma de todas estas cantidades.

La regresión lineal más simple que existe es la regresión sobre una variable explicativa única, llamémosla x, para predecir una variable explicada y. Buscamos los parámetros a y b tales que para el mayor número posible de observaciones tengamos:

$$y \sim ax + b$$

El signo \sim se utiliza a menudo para formular la búsqueda de una aproximación del miembro de la izquierda bajo la forma paramétrica de la derecha.

En el caso general, que incluye varias variables explicativas y se denomina «regresión lineal multivariante», el problema reviste un aspecto menos atractivo. Consideremos tres variables explicativas x_1, x_2, x_3 ; en el caso de un problema multivariante buscaríamos los parámetros a, b, c, d tales que para el mayor número posible de observaciones tengamos:

$$y \sim a x_1 + b x_2 + c x_3 + d$$

Regresión polinomial

En la regresión llamada polinomial, se aplican los parámetros sobre diversas potencias de una variable explicativa ($1, x, x^2, x^3 \dots$).

El grado de esta regresión se corresponde con la potencia más alta que se utiliza; por ejemplo, para una regresión polinomial de grado 3 no se utilizan potencias superiores a tres. Esto puede escribirse así:

$$y \sim a x + b x^2 + c x^3 + d$$

Imagine la regresión polinomial como una regresión lineal sobre nuevas variables que son, de hecho, las respectivas potencias de la variable con la que está trabajando: se han fabricado nuevas features de manera sencilla a partir de x .

Para realizar esta regresión polinomial, es preferible utilizar técnicas especializadas, pues las nuevas variables obtenidas aplicando una potencia a nuestra variable son muy dependientes las unas de las otras y sería una pena no considerar este hecho en la constitución de la variable explicada.

-  La regresión lineal monovariable podría verse como una regresión polinomial de grado 1 y la media como una regresión polinomial de grado 0 (pues $x^0 = 1$).

Generalización del concepto de regresión polinomial

El siguiente ejemplo detalla una regresión polinomial generalizada:

- Consideremos dos variables explicativas (dos features): x_1 y x_2 .
- Consideremos la variable explicada (response variable): y .
- Creemos otro panel de variables a partir de las variables explicativas:

$$x_1, x_2, x_1^2, x_2^2 \text{ y } x_1 \cdot x_2$$

Tratemos de encontrar y como una combinación lineal de estas nuevas variables, es decir, encontrar los parámetros a, b, c, d, e, f tales que:

$$y \sim a x_1 + b x_2 + c x_1^2 + d x_2^2 + e x_1 \cdot x_2 + f$$

He aquí que hemos definido una regresión polinomial «generalizada».

 Nota importante: si bien se utilizan exactamente las mismas variables con las mismas potencias, las dos expresiones siguientes no representan la misma regresión:

Primera formulación:

$$y \sim a x_1 + b x_2 + c$$

Segunda formulación:

$$y \sim a (x_1 + x_2) + c$$

En la primera se buscan tres parámetros, en la segunda se buscan dos, y previamente se ha creado una única feature realizando la suma de nuestras dos variables explicativas.

d. Regresión lineal generalizada

La técnica de regresión lineal generalizada posee muchas aplicaciones, los algoritmos correspondientes son muy rápidos y eficaces. Por lo tanto, es una herramienta en la que debemos pensar antes de embarcarnos en el uso de técnicas más pesadas.

En ocasiones, las técnicas de regresión simples fracasan, entonces es posible intentar aplicar una transformación a la variable explicada (*response variable*), y calcular a continuación una regresión lineal para predecir esta nueva cantidad antes de aplicar a esta predicción la función inversa de la transformación anterior. Es el principio de regresión lineal generalizada (*Generalized Linear Model*: GLM en inglés).

 Para predecir la pertenencia a clases en las que existe una relación de orden (por ejemplo: frío, templado, caliente), podemos tratar de utilizar una regresión considerando los niveles de las clases como ocurrencias de una variable continua: 0,1,2. Esto funciona... en ocasiones.

e. Árboles de decisión, poda, tala

Para encontrar un algoritmo que determine la pertenencia de una nueva observación a una clase determinada (clasificación), resulta natural inclinarse hacia los árboles de decisión.

Suponga que hoy fuera festivo e imagine tres clases de familias, que consideraremos disjuntas para el ejemplo y que cubran todos los casos de nuestro problema:

- «C»: las familias que irán hoy a distraerse a un espacio cerrado (cine, teatro...).
- «A»: las familias que irán hoy a distraerse a un espacio abierto (bosque, parque de atracciones...).
- «F»: las familias que se quedarán en familia (en su casa, en casa de algunos amigos...).

Estas tres clases son clases de nuestra variable explicada (*response variable*), que llamaremos actividad.

Las variables explicativas a nuestra disposición son:

- El hecho de que el tiempo sea bueno o no.
- El hecho de que todos en la familia estén en forma o no.
- El hecho de que se disponga del suficiente dinero para llegar a fin de mes o no.

Llamemos a estas tres variables explicativas de la siguiente manera: Tiempo, Forma, Dinero.

La tabla de observaciones tendría el siguiente aspecto:

Tiempo Forma Dinero Actividad

nok	nok	ok	C
ok	ok	ok	A
ok	nok	ok	F
nok	ok	ok	A
ok	ok	nok	A
nok	ok	ok	C
nok	ok	ok	C
ok	nok	nok	F
nok	ok	ok	C
ok	ok	nok	A
nok	ok	ok	C
nok	nok	ok	C
nok	ok	nok	F
ok	ok	ok	A
nok	nok	nok	F

.../...

El trabajo de un algoritmo de generación de árboles de decisión consiste en deducir un árbol de reglas que se aplicarán en un orden determinado y que refleje lo mejor posible los datos (sabiendo que ciertas líneas de observación, minoritarias, no son compatibles con el árbol de decisión obtenido, como ocurre aquí con la cuarta línea, por ejemplo).

He aquí dicho árbol de decisión.

```
Si el tiempo es bueno: actividad <- "A" o "F"
  Si todos en la familia están en forma: actividad <- "A"
    Si no: actividad <- "F"
  Si no: actividad <- "C" o "F"
    Si se dispone del suficiente dinero para llegar a fin de mes:
      actividad <- "C"
    Si no: actividad <- "F"
```

En ocasiones, la profundidad del árbol de decisión es tal que los efectivos que corresponden a las ramas más largas del árbol (número de observaciones correspondiente) son despreciables. Esto representa un caso de overfitting claro, el árbol tiene en cuenta decisiones demasiado específicas. En este caso es necesario podar la rama en cuestión mediante la eliminación de ciertas reglas partiendo de la base.

En inglés, podar se dice: *to prune*.

f. Clustering, k-means

El método de clustering más conocido se denomina k-means. Se trata de crear grupos de observación agrupados de manera que dichos grupos estén bien discriminados.

El número de grupos se fija previamente (es igual a k).

Una observación pertenece a un grupo si está más próxima al centro de este grupo que al centro de los demás grupos.

Los centros de cada grupo están bien determinados si se ha conseguido minimizar la suma de todas las distancias entre las observaciones y los centros de sus respectivos grupos.

g. k-NN

El algoritmo de los k vecinos más próximos (*k- Nearest Neighbors*) es un algoritmo de clasificación. En su versión más sencilla, la clase de un objeto viene determinada por la mayoría de las clases de sus vecinos más próximos.

La distancia utilizada es, a menudo, la distancia euclíadiana, es decir, la generalización de la noción de distancia con la que vivimos en nuestro espacio cotidiano a una cantidad cualquiera de dimensiones. Las dimensiones son, simplemente, las variables explicativas.

Es posible utilizar otras distancias o medidas de similitudes que se comportan más o menos como las distancias euclidianas, pero mejor adaptadas al problema concreto (por ejemplo, en procesamiento del lenguaje natural: una medida de similitud semántica entre dos textos).

h. Modelos paramétricos

El número de parámetros del modelo k-NN crece con el número de observaciones sobre las que se entrena el modelo; este tipo de modelo se denomina no paramétrico, dado que los parámetros que hay que encontrar no está fijado a priori.

El problema no puede resumirse en una optimización de parámetros.

De manera inversa, la regresión lineal es, por su parte, un modelo paramétrico, pues el número de parámetros del modelo está fijado a priori y el propio objetivo del algoritmo es encontrar los valores más adecuados de los parámetros. Los modelos paramétricos ahorran tiempo de cálculo, en el sentido de que son menos sensibles al aumento del número de observaciones.

i. Lazy algorithm (algoritmo perezoso)

No es habitual traducir este término en español. Sin embargo, ¿en qué sentido estos algoritmos no son muy trabajadores?

La característica de este tipo de algoritmo es realizar solamente el principio del análisis, lo que permite preparar la respuesta a todas las preguntas que se le puedan someter (consultas, peticiones de predicción o de clasificación).

En cambio, este tipo de modelo difiere una parte del procesamiento hasta el momento en que se le invoca para responder a esta nueva pregunta.

Aclaremos esto mediante una comparación entre LM y k-NN.

Si, a través de una regresión lineal LM, se obtiene una fórmula matemática sencilla que permite calcular la respuesta en función de las variables explicativas, se deduce entonces que el algoritmo de construcción del modelo no es un *lazy algorithm*. De hecho, ha sido bastante «trabajador», digamos sistemático, para producir una solución general a su problema.

De manera inversa, en el caso de k-NN, para determinar si una nueva observación pertenece a una clase concreta, el modelo realiza una serie de cálculos complementarios. Este algoritmo no ha sido demasiado «valiente», pues queda cierto trabajo nada desdeñable por realizar en el momento de la consulta.

Observe que, por su naturaleza, ciertos problemas solo pueden tratarse de manera eficaz mediante *lazy algorithms*, puesto que no poseen una solución paramétrica y calculable en plazos de tiempo razonables.

j. Overfitting: sobredeterminación, sobreaprendizaje

En inglés, el hecho de ajustar un modelo a las observaciones del conjunto de entrenamiento (*training set*) se dice *to fit*.

Si este ajuste es demasiado fuerte, el modelo ya no refleja una ley general al fenómeno estudiado, sino solamente las características del conjunto de entrenamiento. Se ha capturado el ruido de los datos de entrenamiento en el modelo. Este pierde sus capacidades de predicción cuando se aplique sobre nuevas variables explicativas. Este fenómeno se conoce como *overfitting*.

En términos operacionales, se podría decir fácilmente que «lo mejor es enemigo de lo bueno» y que la búsqueda del modelo ideal produce un sobreaprendizaje.

k. Validación cruzada, regularización, bagging

Cross validation

Existen varias técnicas para limitar el overfitting. La validación cruzada (*cross validation* o CV) se basa en la idea de separar en varios conjuntos las observaciones que están disponibles para construir el modelo: conjunto de entrenamiento, conjunto de validación, conjunto de prueba.

El modelo se afina basándose en el conjunto de entrenamiento; se comprueba a continuación que se ha obtenido un modelo óptimo que parece no «overfittado» (terrible neologismo) sobre el modelo de validación.

Tras varias iteraciones «entrenamiento + validación», se dispone de un modelo que puede probarse «definitivamente» sobre el conjunto de prueba.

Si todo va bien, sin tener que volver a iterar el modelo de prueba, la probabilidad de overfitting se minimiza. Es lo que se denomina validación cruzada o *cross validation*. Unas veces se utiliza un único conjunto de prueba y de validación y otras veces se utilizan otras etapas de validación...

K-fold cross validation

Cuando el riesgo de overfitting es grande, es habitual dividir el conjunto de entrenamiento (que incluye al conjunto de validación) en una partición formada aleatoriamente para entrenar el modelo sobre esta partición antes de consolidar el conjunto. El conjunto se divide en k partes, de ahí el nombre de *k-fold cross validation*.

Bootstrap aggregating

El *bootstrap aggregating*, o *bagging*, es un método más sofisticado.

El conjunto de entrenamiento se divide en una partición no disjunta de conjuntos de entrenamiento: el muestreo se realiza de manera que las observaciones están duplicadas entre un conjunto de entrenamiento y otro. Sirve para conservar al menos una tasa del 60 % de información no duplicada.

Cada entrenamiento a partir de un conjunto determinado produce un modelo que se aplica sobre los demás datos. El conjunto de resultados se agrega mediante *voting* en el caso de una clasificación, o promedio en el caso de una regresión.

Regularización

La regularización se basa en distintos principios. La idea es bastante sencilla. Para comprenderla, hay que destacar que la búsqueda de los mejores parámetros de un modelo es de hecho la búsqueda de los parámetros que minimizan una función que se ha creado específicamente para controlar la eficacia de esta optimización (una función *ad hoc*).

En el caso del machine learning, esta función *ad hoc* se parece a la «medición del error medio» cometido en la aplicación de este modelo sobre los datos de entrenamiento.

Más adelante llamaremos a esta función «Riesgo»; algunos hablan de «función de coste». Si en lugar de minimizar esta función se minimiza la suma de la función y de una pequeña cantidad adicional, un «margen de error» (*margin*), se disminuye la precisión del ajuste del modelo a los datos.

La introducción de este margen nos permite imaginar que se disminuye el riesgo de sobreajuste, es decir, el riesgo de overfitting. Este método se denomina *regularization* en inglés.

En función del aspecto del término de regularización seleccionado (es decir, el pequeño margen), oirás hablar a menudo de *Tikhonov regularization* (o *Ridge regularization*) o bien de *LASSO regularization* o incluso de *elastic net*, que combina las dos anteriores. Existen otros métodos de regularización; no dude en diseñar el suyo para problemas particulares.

I. Optimización, descenso por gradiente

Como habrá comprendido, la búsqueda de un buen modelo equivale a la búsqueda del modelo menos malo. Del mismo modo, si consideramos un modelo parametrizado (es decir, un modelo paramétrico, o un modelo no paramétrico pero que posea algunos parámetros generales, como la k de k-NN), nuestro trabajo consistirá en encontrar el juego de parámetros menos malo.

Muchos problemas de la vida cotidiana se reducen a problemas de optimización: la optimización del presupuesto familiar se parametriza mediante porcentajes de reparto de los gastos sobre diversos universos (alimentación, alojamiento, vacaciones, impuestos...), la búsqueda del camino más corto mediante un GPS es un problema no paramétrico pero que también es, evidentemente, un problema de optimización.

Optimizar consiste en encontrar los lugares de un mínimo o un máximo.

Para comprender el descenso por gradiente, es habitual describir el proceso que permitiría realizar la búsqueda de la cima de una colina (*hill climbing*) o del fondo de un valle. El ejemplo que hemos seleccionado para ilustrar el descenso por gradiente está más próximo a la realidad psicológica de una optimización «a ciegas», tal y como la va a vivir.

El siguiente ejemplo es próximo a la realidad de la posición del data scientist de cara a un problema de optimización: la caza de los huevos de Pascua en el jardín de los abuelos burlones.

Un huevo de Pascua está oculto, el niño va en alguna dirección, se detiene y pregunta a los abuelos que, en función de la cercanía del huevo le dirán: «frío», «templado», «caliente», «ite quemás!». El niño desea encontrar el huevo lo más rápidamente posible. ¿Cuál sería la estrategia de este niño?

Cuando se le dice «frío», cambia de dirección. Cuando sigue estando frío, en el siguiente punto de control, cambia de nuevo de dirección utilizando la información pasada para orientarse cuanto antes en la dirección en la que piensa que aumentarán lo más rápido posible sus posibilidades de ganar.

Si, por el contrario, se calienta, acelerará en esta nueva dirección y se detendrá mucho más lejos. Allí, puede tener una sorpresa dividida: «ite quemás!», o bien llegar a un nuevo punto frío. En este último caso, se dará la vuelta v

rehará la mitad o un tercio del camino anterior para encontrar de nuevo el lugar más caliente sobre esta línea... y así sucesivamente... es una de las versiones del descenso por gradiente.

Con este algoritmo, el niño va a encontrar la zona del huevo bastante rápidamente; cuando llegue al lugar «quemando» cambiará de estrategia para converger rápidamente y abandonará el descenso por gradiente por un método más sistemático (como observar debajo de todas las flores que le rodeen).

Pero ¿qué ocurre si hay huevos de tamaños diferentes? El descenso por gradiente puede fallar, su éxito depende del lugar del que se parte... y el niño no está seguro de encontrar el huevo más grande. El niño puede haber convergido a un óptimo local sin saber si este óptimo es un óptimo absoluto o no (desafortunadamente, se queda bloqueado en un óptimo local si no sabe que hay otros huevos).

Cuando existen obstáculos que sortear, o si el huevo está en el borde de la curva interior de un terreno con forma de alubia, el descenso por gradiente puede fallar en función del lugar donde empieza la búsqueda.

La paradoja es la siguiente: el descenso por gradiente nos permite buscar un óptimo en un entorno desconocido, pero más vale conocer una serie de características acerca de este entorno para aplicarlo o adaptarse a él.

Por este motivo las matemáticas nos invitan a utilizar este método en ciertos contextos y nosotros lo desaconsejamos en otros.

El descenso por gradiente está bien adaptado al contexto de la minimización del riesgo de una regresión lineal en la medida en que se ha comprobado previamente que el terreno es convexo (a diferencia de las alubias), isótropo (la dificultad de desplazarse es la misma en todas las direcciones, si no existen obstáculos) o solo existe un único óptimo (un solo huevo).

m. Algoritmo voraz (*greedy algorithm*)

Volvamos a nuestro problema de optimización, en el caso de tener varios óptimos. Nuestro pequeño decide correr sistemáticamente en la dirección más caliente y verificamos que no siempre encuentra el huevo más interesante.

Es una característica de los algoritmos voraces (*greedy algorithm*), cuyo heurístico es muy eficaz y los hace converger rápidamente hacia una solución sin ninguna garantía de que sea la mejor. En ocasiones, estos algoritmos caen en impases y quedan bloqueados.

Un algoritmo voraz realiza la mejor elección (aparente) en cada etapa; la sucesión de estas mejores elecciones produce a menudo una buena solución, lo que no tiene nada de cierto al final, en particular cuando se trata de sortear un obstáculo.

El hecho de que un algoritmo sea voraz no es un defecto en sí, sino una característica del algoritmo utilizado. Permanece a menudo sobre buenas opciones, pero debe asumir las consecuencias y los límites en sus conclusiones y en los intentos de generalización de su modelo.

Por motivos de rendimiento, en el caso del machine learning, los algoritmos voraces representan en ocasiones las únicas opciones posibles de cara a resolver un problema complejo y que manipula un número muy grande de observaciones.

n. Programación lineal, simplex, punto interior

Este término se corresponde con una técnica muy útil, en particular para ciertos problemas de optimización bajo restricciones que encontramos en contextos macro y microeconómico.

Preste atención a las confusiones, la programación lineal no trata ni de programación, en el sentido informático del término, ni de regresión lineal.

 Hemos visto ante un método de optimización de una función cualquiera que tenía algunos defectos pero una gran universalidad. El problema que planteamos aquí es más complejo en ciertos aspectos de convergencia, pues se busca optimizar una función basándose en un juego de restricciones aplicadas sobre las variables. El problema planteado aquí es el de la optimización de una función lineal. Por el contrario, el descenso por gradiente está mejor adaptado a funciones más complejas que posean sus propios óptimos y muchas zonas «convexas» donde se esté seguro de «no perderse», salvo en el caso de quedar atrapado en un óptimo local.

En el caso de la programación lineal, los términos (parámetros) de una función lineal, obtenida por ejemplo mediante las técnicas que hemos visto más arriba, ya están disponibles.

El objetivo es encontrar una solución satisfactoria a un juego de restricciones lineales que minimice una función de coste que es, a su vez, una función lineal de las variables.

Consideremos una función lineal $f()$ de un vector cualquiera x . Nuestro objetivo consiste en encontrar un óptimo (mínimo o máximo) de esta función bajo una serie de restricciones, siendo estas restricciones también lineales.

Por ejemplo: $x_2 + 7 \cdot x_8 \leq 5$ es una restricción lineal.

Típicamente, esta función podría representar el margen bruto de una fábrica en función de diversas variables: información variada sobre los costes de producción de diversos productos por día, sobre los recursos materiales útiles para producir cada producto y sobre los precios de venta en función de las cantidades de estos productos...

La cuestión sería maximizar este margen encontrando la mejor composición de distintos componentes del vector x que represente la cantidad de cada producto entregado por día. ¿Qué cantidad de cada producto hay que producir para optimizar el margen respetando las restricciones?

La naturaleza de las restricciones podría ser no almacenar más que una cantidad determinada de materias primas, que ciertas máquinas no puedan producir más de una cantidad determinada, que una máquina no pueda utilizarse simultáneamente por dos personas y que al menos una de las dos posea una formación...

Cada una de estas restricciones debe expresarse como una restricción lineal del tipo $g_j(x) \leq b_j$, siendo las funciones g_j , por su parte, funciones lineales de x .

Por otro lado, se impone sistemáticamente que todos los valores de coordenadas de x sean positivos o nulos (esto siempre es posible, no hay más que cambiar los signos de ciertos coeficientes en las funciones lineales):

$$x_j \geq 0.$$

El algoritmo histórico para resolver este problema se denomina «algoritmo del simplex»; se utiliza en la actualidad a menudo un algoritmo más general llamado «método de punto interior». Le invitamos a consultar los «pattern search optimization». Estos últimos algoritmos pueden ayudarle a abordar la optimización de funciones no lineales.

 De manera global, lo que diferencia a estos tres métodos es el camino efectuado por nuestro «buscador de huevos de Pascua». En el caso del «simplex», el algoritmo se permite realizar un recorrido exterior por zonas que no respetan la restricción. En el caso del «método del punto interior», el algoritmo utiliza precozmente las restricciones para converger más rápido, aunque por este motivo es mucho más complejo. Por último, en el caso del «pattern search», el algoritmo realiza un recorrido algo más «sistématico», pues no soporta características fuertes de derivabilidad de la función de coste; su filosofía es muy diferente a la del descenso por gradiente. La lectura de la documentación de los paquetes R «optim simplex» e «intpoint» y de las referencias citadas en estos paquetes puede aportarle cierta información útil acerca de estos asuntos tan delicados.

o. Estimación mediante el método de Monte-Carlo

El método de Monte-Carlo permite obtener diversa información relacionada con problemas difíciles de definir

completamente y resolverlos por métodos analíticos.

Consideremos el siguiente problema: encontrar el volumen de un recipiente cúbico de cinco caras sólidas de 10 cm de lado interior. Un método analítico nos daría directamente la fórmula que permite calcular un contenido de un litro. Un método experimental podría consistir en llenar el cubo de agua, recoger esta agua y pesarla a continuación para deducir su volumen, también un litro, pero con cierta incertitud.

El método de Monte-Carlo es un método estadístico que simula un método experimental. Para encontrar el volumen de un cubo virtual sin utilizar el método analítico, el método de Monte-Carlo sería el siguiente:

- Imaginar un espacio de referencia que englobe el cubo y cuyo volumen sea conocido.
- Crear aleatoriamente un número importante pero conocido de partículas (puntos) repartidos uniformemente en este espacio tanto en el exterior como en el interior del recipiente.
- Contar las partículas que están en el interior del cubo (comprobando las coordenadas de cada punto).
- Aproximar el volumen del cubo realizando el ratio del número de puntos interiores/exteriores acompañado (multiplicado por) del volumen conocido del espacio de referencia.

Este método funcionará con todos los recipientes de cualquier forma, siempre y cuando sepa decir si la partícula (el punto) está o no dentro del recipiente, sin tener que determinar analíticamente la fórmula que describe al volumen de cada recipiente.

Descubramos una aplicación práctica de este método en el contexto de las data sciences.

Imagine que ha creado un modelo de clasificación. A partir de los valores de diez variables, el modelo predice la clase de la instancia observada. ¿Cómo obtener la probabilidad de que una observación pertenezca a una clase, sin disponer de ninguna información acerca de los valores de estas variables?

- En primer lugar, recopile lo que sepa acerca de las distribuciones teóricas de cada variable y genere observaciones factuales que cubran el ámbito de los posibles valores de cada variable, aunque teniendo en cuenta su distribución teórica.
- Por ejemplo, en el caso de que una variable sea la temperatura corporal, sabiendo que las temperaturas corporales de los seres humanos vivientes son, raramente, inferiores a 35°, se utiliza este conocimiento para generar menos casos a 35° que a 37°.
- Para cada observación factual, aplique el modelo de clasificación.
- Tan solo queda contar los efectivos de cada clase y deducir la probabilidad de que una observación pertenezca a una clase determinada como el ratio de este efectivo respecto al efectivo total.

Para saber cuál es la probabilidad de pertenecer a una clase determinada sabiendo que la temperatura corporal es de 37.5° y el tamaño comprendido entre 1.76 m y 1.85 m, basta con utilizar el mismo método, pero acotando únicamente estos dos valores a los valores conocidos previamente.

p. Entropía, independencia e información mutua

Presentación de los conceptos

La entropía es una medida del esfuerzo que debe realizar un observador para reducir la incertitud que percibe cuando «escucha» una fuente de información. Por ejemplo, si se monitoriza sucesivamente el valor de una variable y esta variable no cambia nunca, la incertitud respecto a esta variable parece nula y la entropía de esta fuente de información es nula, pues no es necesario realizar ningún esfuerzo para predecir la siguiente ocurrencia.

De manera inversa, cuando se confirma durante la observación de una variable que pueden existir varios estados,

y que todos los estados aparecen con la misma frecuencia, se declaran estos estados como equiprobables. En este caso, la incertitud producida por esta fuente parece importante. La entropía de esta fuente única es máxima y depende solamente del número de estados.

Si la probabilidad de uno o varios estados de la fuente es superior o inferior a los demás, la variable considerada es más discriminante y, por lo tanto, genera menos esfuerzo para ser utilizada, pues se «sabe algo más» acerca de ella. Su entropía es más fiable que en el caso equiprobable.

- Para profundizar todavía más: consideremos una variable discreta X con un número finito de estados posibles x_1 , $x_2 \dots x_c$ cuyas probabilidades respectivas sean P_1 , $P_2 \dots P_c$. Su entropía se definiría según la expresión
- $$H(X) = \sum_{k=1}^c -p_k \log_b(p_k).$$

\log_b significa logaritmo en base b. A menudo se utiliza b = 2; la entropía se mide, en este caso, en bits (en ocasiones llamados shannon en homenaje a su creador, Mr. Shannon). En base «e» este \log_e es el logaritmo neperiano, a menudo descrito como I_n , la unidad de la entropía es, en este caso, el «nat». En base 10 hablamos de «hartley».

El cálculo de la información mutua permite formalizar de manera robusta la dependencia de dos variables: cuando el conocimiento del valor de una primera variable no disminuye la entropía observada de una segunda variable, es posible afirmar que estas variables son independientes y que su información mutua es nula.

Dicho de otro modo, la información mutua de dos variables independientes es nula, puesto que no se disminuye la entropía de la segunda variable conociendo la primera (el esfuerzo que debe realizarse para predecir la segunda variable no se ha visto disminuido por el hecho de conocer la primera).

Una formulación algo más precisa

Existe una formulación muy simple que permite reflejar de manera global estas últimas reflexiones. Denotaremos:

- $H(X)$ la entropía de una fuente X.
- $H(X/Y)$ la entropía de esta misma fuente si ya se conoce Y (se dice a menudo «X sabiendo Y»).
- $I(X,Y)$ la información mutua.
- y $H(X,Y)$ la entropía conjunta de las variables X e Y, es decir $X \times Y$.

Esta última noción merece cierta atención: imagine una variable X con dos posibles estados $\{a,b\}$ y una variable Y con dos posibles estados $\{a',b'\}$.

$X \times Y$ sería una variable con los estados $\{(a,a'),(a,b'),(b,a'),(b,b')\}$.

Con estas notaciones tenemos que:

$$I(X,Y) = H(X) - H(X/Y)$$

$$= H(Y) - H(Y/X)$$

$$= H(X) + H(Y) - H(X,Y)$$

Se confirma que *ssi* $I(X,Y) = 0$, esto quiere decir que X e Y son independientes, se tiene:

- $H(X) = H(X/Y) \rightarrow$ conocer Y no aporta ningún conocimiento nuevo acerca de X.

- $H(Y) = H(Y/X)$ --> conocer X no aporta ningún conocimiento nuevo acerca de Y.
- $H(X)+H(Y) = H(X,Y)$ --> si se conoce la distribución $X \times Y$, es posible deducir las distribuciones de X y de Y, y viceversa.

 Recuerde: «ssi» es la abreviatura de «si y solamente si», «iff» en inglés, que quiere decir que las expresiones de la derecha y de la izquierda de este «ssi» son estrictamente equivalentes, es decir, que es posible deducir una de la otra, y viceversa.

Para evitar perderse en su lectura

La entropía, en el sentido de Shannon que hemos abordado aquí, no debe confundirse con la entropía tal y como se considera en las ciencias físicas.

q. Discretización

Las variables numéricas pueden ser reales y continuas o bien enteras y discretas. Representan clases; cuando el progreso del valor de estas clases tiene sentido, las clases son ordinales.

La técnica que consiste en identificar intervalos de una variable numérica continua para transformarla en valores discretos se denomina discretización. Como se parte de una variable continua y, por lo tanto, ordenada, las clases en cuestión son naturalmente ordinales.

Esta técnica resulta útil, aunque peligrosa, y debe realizarse con precaución.

En particular, es conveniente comprobar la sensibilidad de los modelos de predicción o de clasificación que se aplicarán sobre estas clases frente a las variaciones de los parámetros de la discretización implementada. Los intervalos transformados en valores discretos no son sistemáticamente del mismo tamaño.

Podemos crear, por ejemplo, intervalos que engloben rangos equivalentes, lo cual difiere de una división en «pasos» constantes. En efecto, en el caso de una distribución de la variable continua en forma de campana, el rango del primer intervalo fabricado para el primer «paso» es infinitamente más pequeño que el rango de un intervalo de un «paso» cercano a la media.

Encontramos también técnicas de discretización basadas en el hecho de supervisar el proceso de discretización mediante la noción de entropía de los distintos intervalos discretizados (por ejemplo, definiendo pasos con la misma entropía). Esta técnica resiste naturalmente la discretización de distribuciones multimodales, es decir, con varios máximos.

La discretización permite utilizar algoritmos que no están adaptados a las variables continuas (por ejemplo, ciertos algoritmos de clasificación o basados en redes neuronales). También puede contribuir a mejorar el rendimiento o a la gestión del overfitting.

2. Métodos «conjunto»

Cuando se dispone de varios algoritmos, o de varias parametrizaciones que devuelven ciertos resultados, resulta tentador utilizarlos «en conjunto» de una manera inteligente. Esta es una manera de mejorar sus capacidades de *feature engineering*, de clustering, de clasificación, de predicción y, en general, de resolver cualquier problema para el que no resulta eficaz ninguna solución analítica.

El término utilizado para este tipo de ejercicio es, simplemente: «conjunto». Preste atención, esto no tiene nada que ver con la noción de conjunto en matemáticas.

a. Random forest

Los árboles de decisión producidos sobre grandes conjuntos de datos de entrenamiento pueden ser sensibles al hecho de que algunas observaciones sean incoherentes entre sí.

En las data sciences, hay que intentar evitar la creación de algoritmos sensibles a las pequeñas variaciones de los datos de entrenamiento. Existen dos razones principales para ello: el miedo al overfitting y el miedo a que esto sea síntoma de alguna anomalía más importante en el proceso de construcción del modelo.

La idea del *random forest* consiste en articular numerosos árboles de decisión. El algoritmo es eficaz tanto en clasificación como en regresión.

La creación del conjunto se realiza combinando varios conceptos: el bagging, que hemos visto más arriba, junto a una técnica de selección aleatoria de las variables explicativas (*features*). Se articulan árboles de decisión producto de conjuntos de entrenamiento parciales con dos condiciones: no incluyen todas las observaciones disponibles (*data/tree bagging*) ni tampoco incluyen todas las features disponibles (*feature bagging*).

El autor de este algoritmo es uno de los mejores especialistas en técnicas de bagging (Leo Breiman).

Como ocurre en otros algoritmos, la creación del conjunto se realiza mediante un mecanismo de ponderación («*pesos-weight*» aplicados sobre distintos árboles).

Este algoritmo es bastante robusto y posee una cierta universalidad, en particular a través de uno de sus productos derivados: proporciona una clasificación de la importancia de las variables en la toma de decisiones e introduce una noción de «pureza de los nodos» que permiten reflexionar acerca de las features con las que se trabaja.

b. AdaBoost (adaptive boosting)

Un algoritmo de tipo *adaptativo* es un algoritmo que realiza importantes adaptaciones en su comportamiento en función de las circunstancias «intrínsecas» de su funcionamiento. Aquí, «intrínsecas» significa que las elecciones importantes en la adaptación no se realizan necesariamente al comienzo del algoritmo, sino a lo largo de su desarrollo.

Un *boosting algorithm* es un algoritmo de tipo «conjunto» basado en el hecho de que pueden combinarse varios algoritmos débiles para crear un algoritmo fuerte (si fuera necesario, más fuerte que combinaciones de algoritmos que ya incluían algún algoritmo fuerte: ¿sorprendente?). De hecho, cuando se habla de boosting, se piensa a menudo en el conjunto de algoritmos de clasificación (*learners*), que denominamos *weak learners* cuando son débiles y *strong learners* cuando son fuertes.

Los términos *weak* y *strong* se refieren simplemente a la eficacia de estos algoritmos en términos de minimización de la función de riesgo que representa los errores cometidos en los conjuntos de prueba.

Detrás del término *boosting*, se extiende una noción de rapidez o de eficiencia. De este modo, para poder calificarse como *boosting*, el algoritmo de combinación de los *weak learner* debe ser rápido. Esta eficiencia se obtiene a menudo mediante sus capacidades adaptativas y la recombinación inteligente de los distintos elementos utilizados por el algoritmo.

Para comprender el espíritu de este algoritmo, volvamos a la noción de «claseificador» o *learner*. Consideraremos una clasificación basada en dos clases; el «claseificador» es, simplemente, una función que devuelve +1 cuando el objeto que se desea clasificar está en una clase y -1 para la otra clase.

El algoritmo de búsqueda de los coeficientes de ponderación (*weights*) de los distintos clasificadores y el clasificador resultante es el signo de la suma de los clasificadores ponderada por sus coeficientes de ponderación

(de hecho, es el signo multiplicado por 1). Esta parte del algoritmo no es específica de AdaBoost, sino que se corresponde con una técnica denominada *Multiplicative Weights Update*.

El boosting en AdaBoost se caracteriza por la manera de seleccionar los subconjuntos de entrenamiento, y a continuación calcular el error cometido sobre estos mismos subconjuntos mediante los distintos learners. Se trabaja así para determinar los coeficientes de ponderación y obtener un criterio de parada del algoritmo, extremadamente práctico y rápido.

Este criterio de parada precoz, basado en el error de clasificación, poda toda una rama de cálculos potenciales.

Este tipo de técnica de parada precoz se denomina *early termination*. Su objetivo es disminuir el overfitting, aplicando el viejo principio: «lo mejor es enemigo de lo bueno».

Esta descripción rápida de AdaBoost debería hacerle reflexionar acerca de la construcción de sus propios algoritmos. Un algoritmo puede ser revolucionario (los autores de AdaBoost, Yoav Freund y Robert Schapire han obtenido el premio Gödel Prize en 2003) y ser, sin embargo, una combinación de técnicas conocidas.

3. Leyes de probabilidad y de distribución

a. Generalidades

Ley de probabilidad

Es la lista de posibles valores de una variable con sus respectivas probabilidades.

Sirve para representar estos valores mediante un gráfico que muestra las variables en abscisas y la probabilidad (o su densidad) en ordenadas; el ejemplo más conocido tiene la forma de una campana.

Distribución

Es una ley empírica que se obtiene recopilando datos. Piense en la ley de probabilidad como una ley «idealizada» y la distribución como la ley confirmada. No se preocupe demasiado; por abuso del lenguaje o bien por accidente se intercambian a menudo ambos términos.

Ley de probabilidad discreta

El uso de una ley de probabilidad discreta permite obtener directamente la probabilidad correspondiente a cada valor de la variable aleatoria.

Ley de probabilidad continua

En este caso, hablamos de densidad de probabilidad. El uso de leyes continuas permite calcular la probabilidad de que los valores de una variable aleatoria estén situados en un intervalo identificado. Basta con medir la superficie debajo de la curva que se corresponde con el intervalo en cuestión.

Función de distribución

Puede definirse como la función que acumula los valores de una distribución de probabilidad sobre un intervalo de valores posibles de la variable desde $-\infty$ hasta el valor considerado de la variable. Estos valores se extienden de 0 a 1.

Dado que las probabilidades están comprendidas entre 0 y 1, esta función es también la superficie debajo de la curva que representa la ley correspondiente, superficie entre el valor $-\infty$ y el valor considerado de la variable.

Dos leyes de probabilidad son iguales si sus funciones de distribución son iguales, y viceversa.

Cuando se establece la diferencia entre dos valores de una función de distribución correspondiente a dos valores de la variable aleatoria, se obtiene la superficie debajo del intervalo correspondiente a su ley de probabilidad y, por lo tanto, la probabilidad de que los valores de una variable aleatoria estén situados en el intervalo correspondiente.

► Formalmente, si X es una variable aleatoria real, la función de distribución F_X asociada a un número real t posee un valor comprendido entre 0 y 1 incluido tal que $F_X(t) = \mathbb{P}(X \leq t)$.

Y se obtiene $\mathbb{P}(a < X < b) = F_X(b^-) - F_X(a^+)$ si $a < b$.

Si se dispone de una función de densidad de probabilidad f_X para la variable aleatoria X , entonces $F_X(t) = \int_{-\infty}^t f_X(t) dt$.

Ley marginal

► Observación previa: trabaje este párrafo con la mente descansada, este concepto no resulta nada evidente la primera vez que se aborda.

Suponga que dispone de la función de reparto conjunto de dos variables aleatorias y que intenta determinar la función de reparto considerando únicamente una de las variables; basta con hacer tender la variable que desea neutralizar a infinito (es decir, hacia valores tales que su función de reparto sea igual a 1). Esta función se denomina función de reparto marginal.

Preste atención: hasta el momento, hablamos de función de reparto y no de ley.

La pendiente en cualquier punto, vista la función de reparto de la variable restante (la derivada), es la ley marginal.

La función de reparto de una ley continua es la integral de una ley, de modo que es comprensible que la derivada de una función de reparto sea una ley.

También podemos obtenerla realizando un sumatorio de la ley de probabilidad conjunta sobre la otra variable.

► Ayudémonos de una formulación matemática para comprenderlo mejor y tomemos el caso de una ley absolutamente continua y de dos variables aleatorias X e Y , que denominaremos ley de probabilidad conjunta $f_{x,y}$.

La ley marginal de X se expresaría mediante su densidad de probabilidad f_x , con: $f_X(x) = \int_{\mathbb{R}} f_{x,y}(x, y) dy$.

Se trata de una proyección de $f_{x,y}$ sobre X , mediante una integral que debe recordarle el vínculo con la función de reparto.

Esta expresión es generalizable a X (e Y), variables aleatorias de varias dimensiones.

Preste atención: salvo en casos particulares (independencia), conocer el conjunto de leyes marginales no permite calcular la ley de una variable aleatoria conjunta.

b. Pequeño bestiario de leyes de probabilidad

Existen muchas leyes de probabilidad, adaptadas a contextos muy variados, aunque debemos conocer al menos las siguientes leyes.

Ley uniforme (continua o discreta)

En el caso discreto, esta ley expresa simplemente que todos los distintos eventos unitarios posibles son equiprobables, como la probabilidad de obtener una de las seis caras cuando se tira un dado no trucado.

En el caso continuo, la ley uniforme posee un valor constante sobre un intervalo y posee un valor nulo fuera de él.

Cuando se quiere simular un fenómeno del que no se sabe nada en términos de distribución a priori, será esta distribución la que habrá que construir empíricamente y, a continuación, inyectar en ciertas variables explicativas.

Distribución normal

Caracterizada por un valor medio y por una desviación típica, es la ley continua en forma de campana que todo el mundo ha visto, puede que la conozca como curva de Gauss. La distribución normal centrada reducida es una versión de la distribución normal en la que la media es 0 y la desviación típica vale 1. Se la denomina también distribución normal estándar.

Esta ley es particularmente importante, pues si se suma un gran número de variables aleatorias independientes y con una distribución cualquiera, se converge a una distribución normal (teorema del límite central, algo simplificado...).

Ley binomial

Imagine una prueba de tipo «cara o cruz», y considere que su cara es un éxito. Este tipo de variable se denomina variable de Bernoulli.

La ley binomial responde a la pregunta que permite saber cuál es la probabilidad de obtener un cierto número de éxitos k , tras haber lanzado sucesivamente n veces una moneda al aire. Si ambos eventos, cara o cruz, no tienen la misma probabilidad, esta ley también permite responder a la pregunta planteada.

Es, evidentemente, una ley discreta.

Ejemplo de uso: una máquina fabrica objetos que tienen una probabilidad $p = 1/1000$ de no cumplir con una determinada norma. ¿Cuál sería la probabilidad de obtener un rechazo tras realizar 30 pruebas de conformidad?

Ley geométrica

Es la pendiente de la anterior, aunque se concentra en el primer éxito.

Le indica qué probabilidad de éxito esperar tras un cierto número de intentos de tipo «cara o cruz».

Ley de Poisson

Cuando los eventos se producen aleatoriamente en el tiempo o a lo largo de un eje cualquiera que podríamos asemejar a una forma de temporalidad como una noción de sucesión (como los pasos de una persona paseando

por la playa), podemos plantear legítimamente la cuestión de una ley de Poisson.

Este tipo de proceso se caracteriza por el hecho de que la probabilidad de un evento es proporcional al intervalo de tiempo sobre el que tiene lugar, hasta tender a 0 si el intervalo de tiempo es muy pequeño.

En intervalos de tiempo separados, la probabilidad de un evento no se ve afectada sobre el segundo intervalo de tiempo por el hecho de que haya tenido lugar o no en el primer intervalo. Se dice que «el proceso no posee memoria».

- Como esta ley modela de hecho la probabilidad de que se produzca un número de eventos escaso en una sucesión infinita de eventos y su cálculo es muy sencillo, se utiliza para aproximar la ley binomial y ahorrar tiempo de cálculo.

Una ley de Poisson se caracteriza por un número medio de eventos esperados, a partir del cual se calculará la ley.

A menudo la preocupación no es calcular una probabilidad, sabiendo que el fenómeno sigue una ley de Poisson, sino más bien validar o invalidar que sigue o no una ley de Poisson, es decir, asegurar que los eventos se suceden de forma aleatoria (la información acerca de un periodo pasado no resulta útil para predecir un valor en un periodo futuro).

Ejemplo de uso: suponga que dispone de una escala de peligrosidad de ataques ciberneticos sobre sus sistemas de información que va de 1 a 10. Usted ha constatado dos ataques de peligrosidad 5 de media por mes en los últimos años, ¿cuál sería la probabilidad de ser atacado más de 24 veces el próximo año con ataques de una peligrosidad superior a 5? (La respuesta le permite, por ejemplo, anticipar el número de expertos en seguridad informática y Forensic que debería contratar).

4. Los grafos

Cuando se abordan los grafos, nos vemos asaltados en primer lugar por un problema de vocabulario: hace falta algo de valor para adentrarse, ipero qué satisfacciones se obtienen! En efecto, este campo de estudio es casi infinito y posee un gran número de aplicaciones.

Para defenderse en el universo de los grafos, tal y como podría abordarlos funcionalmente en un contexto de data sciences, considere el siguiente ejemplo. Está inspirado en un prototipo interactivo que presentamos en el Salón Big Data Paris 2014, concebido por el autor y otro apasionado de las data sciences, el señor Frédéric Fourré.

Investigadores intercambian opiniones en un blog. Podemos representar cada investigador como uno de los nodos de un grafo. Cada vez que encuentra a dos investigadores implicados en una misma conversación (*thread*), trace una línea entre los investigadores. Si no conserva los duplicados y si desea eliminar los bucles (un investigador hablando consigo mismo), habrá trazado lo que se denomina un grafo simple (no orientado).

A partir de este grafo, es posible responder a numerosas cuestiones:

- ¿Es posible encontrar grupos de investigadores muy relacionados globalmente?
- ¿Es posible identificar aquellos investigadores centrales a estos grupos (influencia...)?
- ¿Ciertos investigadores representan, ellos mismos, vínculos entre grupos y en ocasiones son el único vínculo existente (investigadores que posean alguna doble competencia, innovadores, transmisores de ideas...)?
- ¿Ciertos grupos de investigadores están aislados?

Tras obtener la respuesta a estas preguntas mediante el uso de algoritmos específicos de la teoría de grados, aparece inmediatamente una primera estrategia: etiquetar a los investigadores en clases que se corresponden con

las respuestas a estas preguntas. Cada pregunta habrá generado, en este caso, una o varias features que podremos abordar «convencionalmente» mediante técnicas de machine learning.

De este modo, es posible responder a otras preguntas, como por ejemplo:

- ¿La formación de base de los investigadores está relacionada con la posición de «trasmisor de ideas»?
- ¿Los grupos de investigadores poseen alguna característica común (formación, departamento, número de años de experiencia, posición jerárquica, tema de investigación)?
- ¿Los «influenciadores» son directores de departamento?

Conservando la orientación del diálogo, es decir «quién responde a quién», es posible generar un grafo orientado (con flechas) e imaginar a través de qué investigador podría transitar una idea entre dos investigadores alejados entre sí.

Conservando cada arista, es decir, no eliminando las dobles aristas del grafo cuando existen varios intercambios entre dos investigadores, aparece lo que se denomina un hipergrafo. El procesamiento de los hipergrafos resulta mucho más complejo que el procesamiento de los grafos.

Si se anotan las aristas del grafo con etiquetas o valores, se accede a procesamientos todavía más sofisticados (y en ocasiones costosos en términos de uso de potencia de máquina). En este caso es posible concentrar un hipergrafo sobre un grafo «evaluado»: en nuestro ejemplo anterior habríamos podido sumar el número de arcos entre dos investigadores, o sumar los arcos en un sentido con el valor +1, y los arcos en el otro sentido con el valor -1, para obtener aristas o arcos únicos entre investigadores.

a. Vocabulario básico

Un grafo $G(V,E)$ es una estructura que incluye cimas, en ocasiones llamadas nodos (*vertices* o *nodes*), y aristas (*edges*). Un grafo en el que los nodos no están ordenados (es decir, sin flechas) es un grafo no orientado (*undirected graph*), de manera inversa un grafo con nodos ordenados es un grafo orientado (*directed graph*).

En los grafos orientados, los vínculos con flechas se denominan arcos y en el caso de una arista es una pareja de arcos de ida y vuelta entre dos nodos.

Una arista o un arco de un nodo hacia sí mismo se denomina bucle (*loop*).

Un grafo no orientado sin bucle se denomina grafo simple.

El grado de un nodo de un grafo simple es el número de aristas incidentes al nodo. Para los grafos orientados, también es posible calcular el número de arcos entrantes y salientes.

Si se clasifican los distintos grados de los nodos de grafo, del más pequeño al más grande, se obtiene una serie de números, es decir, una variable discreta, a la que es posible asociar la frecuencia de cada uno de estos grados (por ejemplo: «hay un 5 % de los nodos que poseen un grado igual a 4 en este grafo»). Esta distribución empírica resulta muy útil en el análisis de grafos.

Ilustremos los grados: imagine un grafo con 1000 nodos. Si la frecuencia, y por lo tanto la probabilidad, de tener un grado igual a 900 es del 70 %, puede imaginar que muchos nodos estarán conectados entre sí. Imagine ahora otro grafo de 1000 nodos con una probabilidad del 95 % de tener un grado entre 1 y 3; este grafo es muy poco denso, y será interesante comprobar si este grafo presenta formas, patrones conocidos o si no posee una estructura casi arbórea.

Recuerde la facilidad con la que se ha vinculado aquí una característica de la estructura de un grafo con nuestras

herramientas habituales que trabajan con probabilidades.

Representación de grafos en tablas (matrices)

La matriz que representa el grafo más fácil de manipular se denomina matriz de adyacencia.

Para los grafos no orientados, la matriz de adyacencia del grafo es la tabla que se obtiene situando un 1 cada vez que existe una arista entre dos nodos y un 0 en caso contrario. Esta tabla es simétrica respecto a su diagonal descendente.

Para los grafos orientados, esta tabla se obtiene incluyendo un 1 para cada arco, seleccionando una conversión de sentido entre las columnas y las filas de la tabla, y en general esta tabla no es simétrica.

Por último, si tenemos pesos sobre un grafo evaluado, se informa esta matriz con los pesos correspondientes.

-  No debe confundirse la matriz de adyacencia con la matriz de incidencia, que expresa la relación entre los nodos y las aristas (o arcos), mientras que la matriz de adyacencia expresa la relación entre nodos a través de las aristas (o arcos).

En ocasiones se utiliza otra matriz, matriz nodo a nodo de los grados.

También se utiliza en ocasiones la matriz de Laplace nodo a nodo, pero incluyendo los grados en la diagonal y -1 si los nodos están vinculados (0 en caso contrario).

Estas distintas matrices están vinculadas entre sí mediante diversas fórmulas.

b. Conversión de una tabla de observaciones en un grafo, semejanza

Cuando se dispone de una tabla de observaciones, cada fila de variable explicativa puede considerarse como un punto en un espacio.

En este espacio es posible definir una distancia (a menudo la distancia euclídea) y construir una función de esta distancia que tiende a 1 cuando la distancia es nula y a 0 cuando la distancia es grande. En este contexto, a este tipo de función se la denomina función de semejanza. El valor 1 significa «completamente semejante» y 0 significa «nada semejante».

Si se dota de un umbral, imaginemos 0,9, podríamos decir que si la semejanza entre ambos puntos es superior al umbral de 0,9, entonces ambos puntos se consideran lo suficientemente similares como para estar vinculados (o no, en caso contrario). Esta mecánica nos permite obtener un grafo de semejanza sobre el que podríamos aplicar los algoritmos propuestos en la teoría de grafos.

Llegados a este punto, la primera idea que emerge es evidentemente utilizar un algoritmo de «clusterización» del grafo para hacer emerger una nueva feature de pertenencia de las observaciones a un clúster u otro.

Informática profesional y datasciences

1. La tecnología

Las plataformas de aplicaciones Big Data se han desarrollado a través de diversas ofertas open source o propietarias.

Conviene tener en mente que es posible implementar soluciones de data sciences en las grandes «cloud» (Amazon, Azure, BlueMix...), sabiendo que los contextos que necesitan el despliegue de muchos nodos (nodos Hadoop, típicamente) no se prestan fácilmente a implementaciones en la cloud.

Concentrémonos en las herramientas open source que pueden desplegarse en el seno de una organización (*on-premises*).

El ecosistema Big Data más conocido se denomina Hadoop. La fundación Apache alberga las suites de aplicaciones correspondientes, así como muchas otras suites de aplicaciones: <https://www.apache.org/>.

Apache

He aquí ciertos componentes de la fundación Apache que resultan particularmente importantes y útiles, entre una lista demasiado larga como para ser estudiada aquí con detalle:

- Hadoop es un sistema distribuido, que comprende un sistema de archivos distribuido y redundante llamado HDFS (el factor de redundancia de los datos es, al menos, igual a 3) y la implementación de un algoritmo de paralelización extremadamente eficaz llamado Mapreduce.
- HDFS permite acceder a los archivos de texto o .csv (formato Excel) en condiciones de rendimiento extraordinarias, sea cual sea su tamaño. Por otro lado, el uso de esta tecnología solo tiene sentido para archivos relativamente voluminosos. HDFS también puede albergar los archivos de diversas bases de datos.
- HBase es una base de datos orientada a columnas, muy eficaz en tiempo real en el marco del Big Data, que permite gestionar tablas con millones de filas y millones de columnas.
- Hive es un almacén de datos (*datawarehouse*) compatible con un almacenamiento distribuido, lo que no ocurría con los estándares de otras tecnologías de datawarehouse. Hive incluye herramientas de carga y de transformación de datos de tipo ETL (*data Extract, Transform, Load*).
- Lucene es un sistema de búsqueda de tópicos en textos con un rendimiento muy elevado. El sistema permite realizar consultas full-text y cross-plataforma.
- Giraph es un gestor de grafos, basado en principios iterativos y una alta escalabilidad (capacidad para seguir el aumento de los volúmenes). Facebook utiliza Giraph.

Respecto al machine learning propiamente dicho, conviene seguir la evolución de las siguientes aplicaciones:

- Spark, que permite realizar a la vez machine learning, análisis de grafos y stream processing (permitiendo la interacción sobre grandes cantidades de datos).
- Mahout, que integra algoritmos de machine learning, y por lo tanto accesos a una tecnología open source diferente de Apache llamada H2O. Esta tecnología también es directamente accesible mediante paquetes R.

Por último, Apache proporciona componentes vitales para gestionar y controlar todo el conjunto, entre ellos:

- Oozie, para gestionar la planificación de los procesos (gestor de workflow).
- Zookeeper, para gestionar la coordinación de los componentes del sistema.

Las distribuciones de Hadoop

Estos componentes, junto a otros, se empaquetan a menudo en diversas ofertas dentro de distribuciones en las que cada empresa agrega un cierto valor añadido. El autor ha utilizado personalmente las distribuciones de las aplicaciones Apache provistas por Hortonworks e IBM. Puede dar fe de la calidad de dichas soluciones y está certificado en los entornos Big Data de IBM, ya sean open source o propietarios.

Habida cuenta de la intensidad de la competencia, es probable que las demás distribuciones posean a su vez una excelente integración y características propias muy interesantes: no dude jamás en implementar un prototipo y medir objetivamente la capacidad de las distintas soluciones a partir de sus objetivos.

Cuando escoja una solución, no descuide jamás el hecho de que toda implementación informática posee anomalías, por lo que resulta vital asegurar varios aspectos: transparencia del fabricante respecto a los bugs (incluidos los problemas de seguridad), la existencia de correctivos, una comunidad activa para obtener información y una buena cantidad de recursos disponibles en el mercado.

Paradójicamente, si no posee los recursos financieros para adquirir recursos humanos de alto nivel en caso de dificultad, utilice soluciones menos integradas, con un despliegue menos automatizado, pero cuyos detalles y pormenores domine.

Evidentemente, asegúrese en este caso de que sus equipos gestionan su configuración de la aplicación de manera profesional utilizando de forma adecuada gestores de configuración como SVN (Apache Subversion), donde cada componente, y particularmente los scripts de instalación y de actualización, se gestionarán correctamente (versiones y árboles, versiones de desarrollo, pruebas, integración y producción sobre cada entorno, así como topología de su red). Sea intransigente en lo que respecta a la existencia de una documentación vigente, ligera, actualizada e indexada sobre todo el conjunto. Esté también atento a la trazabilidad y a la gestión de las exigencias técnicas y funcionales que han afectado a la implementación de cada script, herramienta o sección de código.

Estas distribuciones pueden descargarse en su versión community edition (gratuita) y puede instalarlas en máquinas locales para realizar un primer uso. Si tiene dudas sobre su capacidad de instalación, puede descargar máquinas virtuales que pueden utilizarse con VMware u Oracle VirtualBox. Y a la inversa, si usted es curioso, puede instalar manualmente las últimas versiones de los componentes Apache descargándolos de apache.org.

Utilizar R

Al margen de estas distribuciones, también puede utilizar R en contextos estándares Windows o Linux sin ningún tipo de dificultad. Incluso en estos entornos simples, existen medios de implementar ciertas funciones de paralelismo y gestionar el rendimiento.

En el siguiente capítulo, encontrará una pequeña comparativa (bench mark) de rendimiento que compara dos formas de aplicar una función sobre una única columna de una tabla en R. Esta prueba dura solo unos pocos minutos en un PC moderno con tan solo 16 MB de memoria, y sin embargo el último bucle manipula 10 millones de filas.

2. Business Intelligence versus Big Data

Muchas personas se preguntan cuál es la diferencia entre Business Intelligence (BI) y Big Data.

Business Intelligence y Big Data se refieren, en la actualidad, a tecnologías, arquitecturas y usos que poseen ciertos aspectos comunes, pero también muchas diferencias que podríamos resumir de la siguiente manera.

a. Diferencias en términos de arquitectura

En las arquitecturas típicas de BI, los datos están centralizados tras haber sido tratados y estructurados. Las máquinas utilizadas están dotadas de una gran cantidad de memoria, de procesadores avanzados y de discos de gran capacidad con un rendimiento extremadamente bueno.

En las arquitecturas características de Big Data, se divide y transporta las consultas hasta diversas máquinas (*data node*). Estas máquinas presentan numerosas redundancias en términos de datos (típicamente un factor igual a 3 o 4). La posibilidad de que falle alguna máquina forma parte del modelo, si bien las máquinas correspondientes no están sometidas a situaciones tan severas como las máquinas que se utilizan de manera clásica en las arquitecturas BI.

En las arquitecturas Big Data, es habitual utilizar en paralelo numerosas máquinas con costes unitarios razonables (en inglés se habla de *commodities*), lo que puede generar ahorros importantes en las inversiones y los costes de mantenimiento. La capacidad de extender una arquitectura Big Data resulta nativa (agregando nuevas máquinas), a diferencia de BI, cuya «escalabilidad» depende de la capacidad de ampliar la potencia de una máquina central.

Estas arquitecturas pueden dimensionarse para soportar volúmenes del orden del petabyte, mientras que BI se limita clásicamente a terabytes. Sin embargo, cabe destacar que la estructuración rígida y potente de las arquitecturas BI permite obtener de manera natural excelentes rendimientos en términos de reporting o de navegación en los datos. Destacaremos, sin embargo, que las arquitecturas Big Data evolucionan parcialmente hacia arquitecturas de flujo (*stream*) para proporcionar mejores rendimientos interactivos en casos en los que los volúmenes lo permiten.

El Big Data, por su parte, permite gestionar datos de naturaleza variada y no estructurada. Eventualmente, estos datos pueden ser incompletos y variar en el tiempo, en términos de valor pero también en términos de estructura.

De manera inversa, el BI trabaja sobre datos estructurados y esquemas relativamente estables (en particular en lo relativo a los ejes que representan las principales dimensiones de análisis contempladas). Accidentalmente, el Big Data, a diferencia del BI, resulta ineficiente en contextos que requieren una representación OLAP nativa (cubos de datos similares a grandes tablas dinámicas).

b. Diferencias en términos de uso

La tecnología Big Data se distribuye, a menudo, en open source, lo que resulta menos habitual en el universo de BI. Las arquitecturas son muy avanzadas y, por lo tanto, las técnicas de agilidad y de prototipado están ampliamente difundidas en este entorno.

El BI proporciona a los usuarios informes, o bien recorridos interactivos a través de los datos (drill down...) típicamente implementados en los cubos OLAP o bien herramientas de consulta sofisticadas. La interpretación y la decisión se dejan de parte del usuario. Algunos sistemas proporcionan data-mining, pero sobre volúmenes restringidos bien estructurados. El uso de este data-mining está, a menudo, limitado a técnicas de clusterización cuyo objetivo es clasificar poblaciones en grupos «homogéneos» sobre los que se podrá aplicar una interpretación y elaborar decisiones estudiando las diferencias entre los grupos.

El Big Data integra de manera natural el data-mining tal y como se ha descrito más arriba, pero dotado a su vez de capacidades predictivas de gran potencia.

En este sentido, hay quien estima que el Big Data es una evolución del BI. De manera objetiva y pragmática, este punto de vista resulta erróneo, puesto que los casos de uso básicos, la arquitectura o los tipos de datos no se superponen.

Por supuesto, haciendo evolucionar la definición de uno y otro de estos términos, siempre podremos decir que uno engloba al otro. Pero este punto de vista resulta peligroso, pues oculta las reflexiones previas acerca de la

implementación o de la evolución de estos sistemas.

En el actual estado del arte, nos parece más prudente determinar e implementar arquitecturas que asuman su diversidad y ordenar precisamente los casos de uso adaptados a las fortalezas y debilidades de ambas tecnologías. Estos dos universos colaboran de manera eficaz desde el momento en que no se las deja pervertirse mutuamente.

c. En resumen

La colaboración más natural entre el Big Data y el BI consiste en alimentar los datawarehouses (almacenes de datos) y los cubos OLAP con elementos predictivos propios del procesamiento de las data sciences realizados sobre los datos gestionados por una arquitectura Big Data.

A menudo resulta dramático limitar los procesamientos de las data sciences a datos ordenados, estructurados y «esterilizados» dentro de los datawarehouses; sin embargo se trata de un error de arquitectura muy habitual.

Las data sciences se practican con una eficacia infinitamente mayor en entornos de Big Data, puesto que incitan a una gestión rigurosa e iterativa de modelos de predicción potentes. Sin embargo, muchos casos de uso de las data sciences no requieren ni una arquitectura Big Data ni una arquitectura BI cuando los volúmenes de datos que se desea manipular resultan pequeños.

Notación

Las distintas notaciones entre los autores, sus hábitos o los usos propios en la descripción de un problema determinado pueden desconcertarle y hacer que su comprensión sea más lenta, de modo que vamos a abordar el tema de la notación matemática en varias ocasiones dentro de este libro.

Para poder evolucionar en las data sciences, no necesita obligatoriamente un buen nivel en matemáticas (esto dependerá de sus tareas y de su ambición), aunque necesitará de cualquier manera leer e interpretar textos que incluirán expresiones matemáticas en ocasiones algo abstractas. Si no posee una formación superior en matemáticas, considere esto como el aprendizaje de un idioma, que le permitirá acceder al pensamiento de los autores que lea.

Vamos a centrarnos en primer lugar en las distintas maneras de describir los parámetros y la expresión de una regresión lineal. La tarea es algo ardua, pero una vez superada podrá acceder a numerosas lecturas sin tener que preocuparse por la notación.

1. Notación de los parámetros

En lo que respecta a los parámetros, las notaciones difieren según las circunstancias, los hábitos y los autores, de modo que no se deje impresionar, ien particular por las letras griegas!

Más arriba en este libro, para describir un ejemplo de regresión lineal multivariante, hemos formulado la siguiente expresión, donde las distintas variables explicativas están expresadas en x con índices y los parámetros a estimar por la sucesión de las primeras letras de nuestro alfabeto:

$$y \sim a_1 x_1 + b x_2 + c x_3 + d$$

Nuestras letras $a, b, c, d\dots$ se convertirán, en ocasiones, en:

- $a_1, a_2, a_3, a_4\dots$ o bien (a_j) en una notación más compacta. En ocasiones, se prefiere empezar con el índice 0 para diferenciar al parámetro que no está multiplicando a una variable, lo que produce para la regresión lineal simple: $y \sim a_1 x + a_0$. Usaremos esta notación en ciertos lugares dentro de este libro.
- $w_0, w_1, w_2, w_3, w_4\dots$ o bien (w_j) en una notación más compacta, para expresar la noción de pesos (weight) y, por lo tanto, de ponderación de las variables, pero con w_0 no ponderando a nada o bien a una variable «ficticia» (*dummy*) que vale siempre 1. Esto produce para la regresión lineal simple: $y \sim w_1 x + w_0$.
- $w_1, w_2, w_3, b\dots$ para expresar que el último (o el primero) de los parámetros no se multiplica por ninguna variable. Esto produce: $y \sim w_1 x + b$ para la regresión lineal simple.
- $\beta_0, \beta_1, \beta_2, \beta_3, \beta_4$ o bien (β_j) en notación más compacta: una notación utilizada por los autores anglosajones en estadística/regresión lineal. Esto produce: $y \sim \beta_1 x + \beta_0$ para la regresión lineal simple y «evidentemente» esto para la regresión lineal multivariante escrita en notación relativamente compacta: $y \sim \sum_{j=1}^p \beta_j x_j + \beta_0$.

Usaremos esta notación en ciertos lugares de este libro.

- $\theta_0, \theta_1, \theta_2, \theta_3, \theta_4$ o bien (θ_i) en notación más compacta: muy habitual, similar a la anterior, pero sin insistir en la noción de regresión lineal, lo que permite designar cualquier tipo de parámetro...
- o, incluso más elegante para nosotros, para la regresión lineal multivariante: $y \sim w^T \cdot x + b$ para expresar que se puede comprender como el producto escalar (noción que veremos más adelante) de dos vectores w y x , al que se suma b . Evidentemente, en lugar de las w , todas las demás letras utilizadas más arriba y todas las que quiera utilizar están disponibles, incluidas en esta última expresión. Usaremos esta

notación en ciertas secciones de este libro.

- Más rara, pero extremadamente compacta, la notación de Einstein, que elimina el signo de sumatorio aplicando una convención de índices superiores e inferiores que estipula que, si encuentra dos veces el mismo índice en un producto, uno arriba y otro debajo, esto significa que debe realizar un sumatorio, es decir: $y \sim \sum_{\mu=0}^p a_\mu x_\mu$ se convierte en algo del tipo: $y \sim a^\mu x_\mu$

Aquí los índices superiores no son potencias, y la convención utilizada es que los índices expresados en letras griegas empiezan en 0, mientras que los índices expresados en letras latinas empiezan en 1.

2. Otras notaciones

a. Funciones y aplicaciones ... $f(x)$, $d(x,y)$...

Las funciones se denotan, a menudo, mediante las letras f , g , h , aunque ciertos autores adoptan convenciones particulares para tipos de funciones particulares:

- Φ para las funciones no lineales, en particular cuando se corresponden con alguna transformación astuta de las variables explicativas o explicadas.
- d para las distancias.
- p para las probabilidades.

b. Algunas posibles confusiones

Cuando el contexto se comprende bien, es improbable confundirse, pero en cualquier caso...

El signo Σ representa el signo de sumatorio, aunque muchos autores lo utilizan también para designar a otros objetos, en particular matrices y, a menudo, la matriz de varianza-covarianza (que utilizaremos en el capítulo Feature Engineering).

El signo ' (prima) se utiliza de forma muy variada:

- Para designar a otro objeto del mismo tipo: x' , x'' , x''' en lugar de x_1 , x_2 , x_3 ...
- Para designar la derivada (a menudo la derivada en función del tiempo, es decir, el límite de la tasa de variación de x en función de la distancia considerada cuando esta duración tiende a 0... es decir, la velocidad instantánea!).
- La traspuesta de una matriz (consulte el capítulo Dominar los fundamentos), nosotros daremos preferencia a la notación X^t o tX en lugar de X' para expresar la traspuesta, aunque reconocemos que es muy rápido de escribir!

Φ o bien \varPhi designa a menudo la ley de probabilidad estándar normal; en negrita o en mayúscula, Φ representa la función de distribución correspondiente. La distribución normal se designa, a menudo, por $N(\mu, \sigma^2)$, lo que significa: distribución normal de media μ y desviación típica σ .

\emptyset en lugar de \varnothing puede, en ocasiones, designar el conjunto vacío o el evento «imposible».

β_1 y β_2 designan a menudo los coeficientes de Pearson, donde el primero expresa la asimetría (*skewness*) de una distribución y el segundo su curtosis o apuntamiento (*kurtosis*).

En una prueba, α es el error de tipo I (type I), que es la probabilidad de rechazar la hipótesis nula H_0 cuando esta es verdadera.

En una prueba, β es el error de tipo II (type II), que es la probabilidad de aceptar la hipótesis nula H_0 cuando esta es falsa.

Producto escalar de dos vectores

El producto escalar de dos vectores (*inner product*) puede tener distintas notaciones en función de la naturaleza del espacio vectorial considerado y de los hábitos de los autores:

- $u \cdot v$: es el caso más común.
- $(u|v)$: alusión a los espacios vectoriales complejos.
- $\Phi(u,v)$: ídem.
- $\langle u,v \rangle$: ídem o alusión a los espacios vectoriales de funciones.
- $k(u,v)$ o bien $k\langle u,v \rangle$: otros productos escalares o generalizaciones...

Recordemos que, en el caso más común, el producto escalar es la suma de los productos miembro a miembro de las coordenadas de dos vectores: $u \cdot v = \sum u_j v_j = u^T \cdot v$ (de hecho, esta última expresión es una matriz con un único valor, que es un escalar y que se asemeja al escalar en cuestión).

Este producto es nulo si los vectores son perpendiculares, positivo si el ángulo entre los dos vectores está comprendido entre -90° y $+90^\circ$... y negativo en caso contrario.

Producto diádico de dos vectores (two vectors outer product)

No hay que confundirlo con el producto vectorial, que se aplica solamente en el caso de dos vectores de tres dimensiones. Como tratar solamente tres dimensiones es algo lo suficientemente extraño en data sciences, no encontrará con demasiada frecuencia el producto vectorial en data sciences.

Cuando se dispone de una base de un espacio vectorial, es posible expresar los vectores fácilmente como columnas, y se dispone de matrices para expresar las funciones lineales (que estudiaremos en el capítulo Dominar los fundamentos). Este es el caso más habitual en data sciences, y parte de este producto como versión de otro producto llamado producto de Kronecker. La buena noticia es que resulta muy fácil de construir:

$$u \otimes v \leftarrow u \cdot v^T$$

Visualmente, es «la expresión inversa» de la que le da un producto escalar, aunque el resultado es una matriz.

En lugar de escribir un signo igual, es habitual escribir un signo «flecha» aquí, pues la matriz obtenida no es sino una representación de este producto diádico. En la práctica esto no tendrá ninguna importancia.

Ahora, ¡es su turno!

Queda dicho... puede acceder al código de este libro desde la página Información.

Reparte y dedica su tiempo a:

- El desarrollo de algoritmos geniales para su startup de data sciences, llamada Advanced Research Partners.
- La presidencia y dirección de los trabajos de investigación del french chapter de la asociación mundial de especialistas en inteligencia económica y competitiva: SCIP.org, cuya representación en Francia realiza APIEC.org.
- La animación y moderación de una pequeña comunidad de data scientists, Data-scientiX.org.
- Y la animación del laboratorio de investigación y de prototipado en data sciences, Big Data y Resiliencia del grupo informático Blue Soft, el laboratorio BlueDsX.

Si por algún motivo está interesado en alguna de estas comunidades, no dude en iniciar el diálogo!

Primeros pasos con R

Instalación de los componentes

La instalación y la invocación de los componentes básicos son fáciles. Veamos cómo proceder.

1. Instalación y ejecución de R

En primer lugar hay que instalar R en función de su plataforma de trabajo.

R está disponible en la siguiente URL: <https://cran.r-project.org/>

Si trabaja con Unix, Linux o alguno de sus OS derivados, sería más práctico utilizar su administrador de paquetes habitual en lugar de instalarlo desde el sitio CRAN (Ubuntu [Synaptic], dpkg [Debian], pkg [Mac OS X], RPM [Red Hat]...).

Para Windows, cuando se le pide seleccionar entre la versión de 64 bits (la más común en la actualidad) o de 32 bits, intente trabajar con 64 bits si su máquina lo permite. Desde mi punto de vista, conviene evitar instalar las dos simultáneamente pues esto produce, en ocasiones, pequeños problemas técnicos, en particular para los paquetes R que invocan Java como tarea de fondo.

Una vez realizada la instalación, para ejecutar la interfaz gráfica de R, basta con hacer clic sobre su ícono.

Para comprobar su instalación, es conveniente crear un nuevo script de R. Para ello, presione simultáneamente las teclas [Ctrl] **N**, lo que abrirá una segunda ventana.

En esta ventana, introduzca un cálculo sencillo, como por ejemplo **1+1**. A continuación presione [Ctrl] **R** en la línea del cálculo que desea realizar (o sobre una selección de líneas). El resultado se muestra a continuación en la primera ventana, que se denomina **console**. Para guardar este script, presione [Ctrl] **S**, recorra su sistema de archivos, seleccione un lugar donde guardar el archivo y llámelo **miscript.R** prestando atención a darle la extensión **.R** en mayúsculas. Para abrir un script existente, basta con presionar [Ctrl] **O** y buscar el script en su equipo.

Todas estas manipulaciones pueden realizarse utilizando los menús y los iconos a su disposición en la interfaz gráfica de R.

 Existen interfaces gráficas más agradables para manipular R; por nuestra parte, nos encanta utilizar RStudio.

2. Instalación y ejecución de RStudio

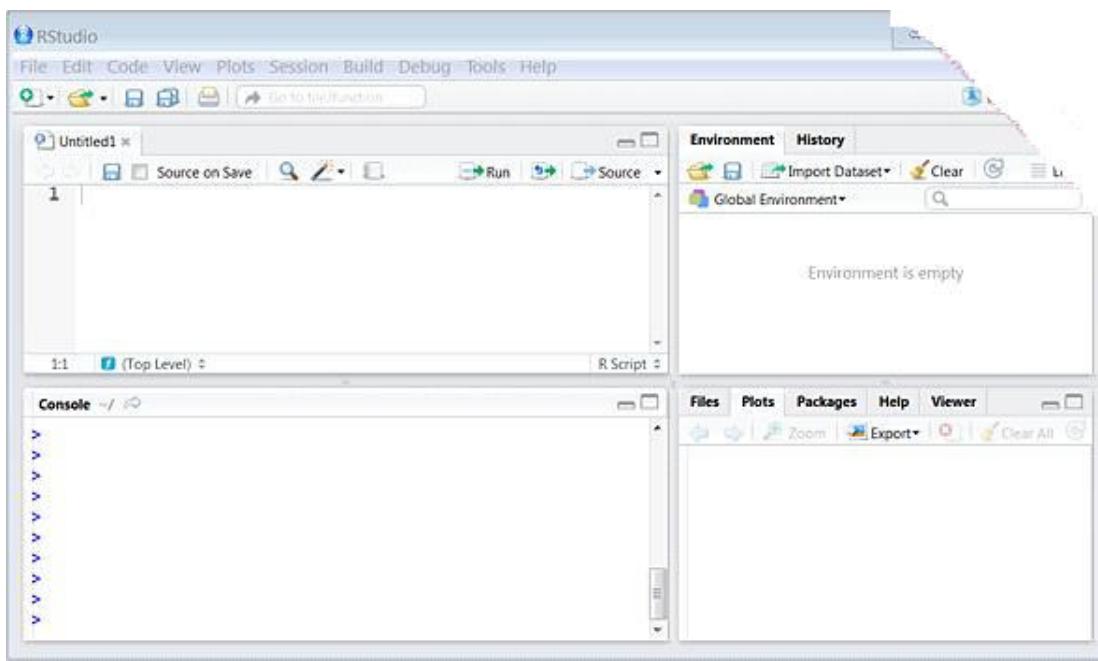
Encontrará RStudio en la siguiente URL: <https://www.rstudio.com/>

Instálela tras haber instalado R.

Existen varias versiones; seleccione sin duda alguna instalar la versión Desktop - open source - community edition, que es gratuita y totalmente operacional.

Una vez realizada la instalación, conviene comprobar RStudio creando un nuevo script. Para hacerlo, vamos a presionar [Ctrl][Shift] **N** o utilizar el menú.

Ahora puede navegar en cuatro zonas de una interfaz gráfica muy agradable.



Interfaz gráfica de RStudio

En la zona superior izquierda, puede acceder a los distintos scripts mediante un juego de pestañas.

En la zona superior derecha, puede acceder a todas las estructuras de datos que va a manipular y a la lista de comandos que se han ejecutado anteriormente. Puede volver a invocar estos comandos o transferirlos a la zona de script o a la zona de consola para reutilizarlos o modificarlos.

En la parte inferior izquierda, la consola le permite o bien ver el resultado de la ejecución de los scripts, o bien ver la ejecución de las líneas de R que haya introducido.

En la parte inferior derecha, puede visualizar los gráficos que haya producido y obtener ayuda acerca de los paquetes o las funciones de R que tenga instaladas en su entorno.

- R incluye muchas funciones nativas, pero comprobará rápidamente el interés de invocar paquetes complementarios.
En interés de este biotopo, R incluye paquetes para numerosos casos de uso en data science. Estos paquetes se encuentran, a menudo, embebidos, así como su documentación básica, en el sitio CRAN donde se encuentra R.

3. Instalación de nuevos paquetes

En el caso de uso más habitual, los paquetes se instalan de la manera descrita más arriba. Algunos paquetes específicos no están presentes en CRAN y poseen instalaciones específicas documentadas por sus autores.

Instalemos un paquete dedicado al cálculo matricial (paquete complementario al cálculo matricial ya integrado en R). En RStudio, en la zona de script, copie las líneas de código propuestas más adelante, selecciónelas con el ratón y ejecútelas presionando [Ctrl][Intro].

En este script, la primera línea se corresponde con la instalación del paquete **matrixcalc**; observe las comillas. Lo que se escribe detrás del símbolo # no se tiene en cuenta en R, se trata de comentarios que le ayudarán a recordar el sentido de su código. La segunda línea invoca su paquete.

```
install.packages("matrixcalc") # para manipular matrices
```

```
library(matrixcalc)          # se lanza el paquete
```

En la consola (debajo a la izquierda) aparece la respuesta a sus acciones:

```
> install.packages("matrixcalc") # para manipular matrices
trying URL 'http://cran.rstudio.com/bin/windows/contrib/3.1/
matrixcalc_1.0-3.zip'
Content type 'application/zip' length 163518 bytes (159 Kb)
opened URL
downloaded 159 Kb

package 'matrixcalc' successfully unpacked and MD5 sums checked
```

The downloaded binary packages are in

```
C:\Users\boss1\AppData\Local\Temp\Rtmp84gGPp\downloaded_packages
> library(matrixcalc)          # se lanza el paquete
Warning message:
el paquete 'matrixcalc' se ha copiado con la versión R 3.1.3

>
```

Encontramos en primer lugar la llamada a la primera línea ejecutada, a continuación el lugar a partir del que R busca y descarga la última versión del paquete. R le indica que ha descomprimido el archivo comprimido del paquete. A continuación, R le indica que el paquete se ha invocado correctamente.

➤ Encuentra también un recordatorio de la versión de R con la que se ha compilado el paquete; a menudo esta no se corresponde con su versión actual de R, aunque esto no tiene ninguna consecuencia. En caso de duda (rara vez), puede querer reinstalarlo todo para fabricar un nuevo entorno. (Le aconsejamos hacer esto de media una vez al año, cuando se liberan versiones principales, aunque comprobando siempre que sus paquetes preferidos se han compilado correctamente con la nueva versión de R. Para ello, instale R temporalmente en una máquina virtual y compruebe su código). Para mejorar todas las instalaciones de los paquetes, puede resultar conveniente escribirlos en un único script y asignarle un nombre fácil de recordar, como por ejemplo **instalacion.R**. Este script podrá ejecutarse cuando deba instalar una nueva versión de R o durante la instalación de una nueva máquina.

Cuando termine el procesamiento, el cursor de la consola, es decir el pequeño símbolo **>**, está disponible de nuevo para continuar trabajando. Durante la ejecución de sus comandos, aparece un pequeño ícono rojo que representa un **stop** en la parte superior derecha de la ventana de la consola, que desaparece una vez se devuelve el cursor.

Para conocer el número de la versión de R que hay instalada en su entorno, basta con ejecutar:

```
version
```

```
platform      x86_64-w64-mingw32
arch         x86_64
os           mingw32
system       x86_64, mingw32
status
major        3
minor        1.2
year         2014
month        10
day          31
```

```
svn rev          66913
language        R
version.string R version 3.1.2 (2014-10-31)
nickname       Pumpkin Helmet
```

Esta instrucción nos devuelve mucha información. Es posible comprobar que se trabaja en 64 bits, que nuestra versión es un poco vieja. Pero confirmamos que es una versión 3.1.2 y, por lo tanto, no muy diferente a la versión 3.1.3 con la que se ha compilado nuestro paquete (y que producía un warning). Esto confirma que sin duda no resultará útil tener en cuenta el warning (advertencia).

La versión posee un nombre; realizando la siguiente búsqueda en un motor de búsqueda en Internet, podrá encontrar, llegado el caso, información acerca de ella:

"Pumpkin Helmet" R

Hemos visto cómo instalar un paquete, pero vamos a confirmar cómo este método puede mejorarse.

4. Instalación de paquetes: complementos

La manera de instalar paquetes que hemos visto antes puede mejorarse. En efecto, ciertos paquetes necesitan la instalación previa de otros paquetes y, en ocasiones, resulta duro que falle una instalación porque los paquetes no se han instalado en orden (sobre todo cuando no siempre se conoce este orden). Además, en un script R destinado a trabajar sobre diversas máquinas, resulta útil lanzar la instalación de los paquetes que faltan (y únicamente si no se encuentran instalados).

La sintaxis que debe utilizarse es la siguiente, y puede introducirse al comienzo de un script que necesite un paquete en particular. Imagine que desea instalar el paquete llamado **un_paquete**:

```
## código ficticio    "NOT RUN" ##

if(require("un_paquete")==FALSE)
  install.packages("un_paquete",
                   dependencies=c("Depends", "Suggests"))
library(un_paquete)
```

Tras los comentarios, la primera línea ejecutable se divide en tres líneas. Se comprueba si el paquete ya se encuentra instalado y si no es el caso se lanza la instalación del paquete y de todos los paquetes dependientes o sugeridos como vinculados y potencialmente útiles en la documentación del paquete.



iDe momento, no se centre en la sintaxis de este código, cuya interpretación requiere poseer algún conocimiento sobre el lenguaje R!

Para terminar, le propongo instalar tres paquetes técnicos que no necesitaremos forzosamente, pero que le servirán de cualquier modo algún día.

```
## paquetes de sistema a instalar ##  
  
install.packages("rJava",  
    dependencies=c("Depends", "Suggests"))
```

```
library('rJava')

install.packages("downloader",
                 dependencies=c("Depends", "Suggests"))
library(downloader)

install.packages("devtools",
                 dependencies=c("Depends", "Suggests"))
```

Puede diferir estas instalaciones, pero lo que ya se ha hecho ya no se tiene que hacer...

El paquete **rJava** permite a ciertas herramientas proporcionarle diversas funciones suplementarias escritas en Java. El paquete **downloader** permite, entre otros, descargar ciertos paquetes que no se instalan desde CRAN. El paquete **devtools** permite, entre otros, realizar ciertas instalaciones que no se realizan desde CRAN.

 Si la instalación del paquete **rJava** plantea algún problema, le recomiendo reinstalar Java (vaya al sitio web de Oracle o directamente a: <https://www.java.com/es/download/> y, a continuación, descargue el JRE). En el sitio web de Oracle puede leer: «También se hace referencia al software de Java para su computadora (o Java Runtime Environment) como Java Runtime, Runtime Environment, Runtime, JRE, máquina virtual de Java, máquina virtual, Java VM, JVM, VM, plugin de Java, complemento de Java o descarga de Java».

Tendrá que comprobar que existe coherencia entre las instalaciones de Java y de R (típicamente todo en 64 bits o todo en 32 bits, y sin mezclas entre 64 y 32 bits).

Ahora estamos preparados para trabajar con R.

Toma de contacto con R

Según sus necesidades y sus competencias, el usuario puede utilizar R a niveles de abstracción y de potencia muy variables. R puede servir, por tanto, como calculadora sofisticada, universo de iniciación a la estadística, para las data sciences y la programación o como potente herramienta para los investigadores o los data scientists experimentados, incluso en entornos Big Data.

Las siguientes secciones le permiten realizar un uso eficaz de R en el marco de las data sciences, no como desarrollador de paquetes R, sino como data scientist que prepara prototipos de calidad. Nuestra elección de sintaxis y de estilo de programación es, sobre todo, didáctico, con el ánimo de obtener una buena legibilidad e ilustrar las diversas ideas principales en nuestras prácticas. Nuestro objetivo no es realizar un recorrido sistemático de R ni describirle R como podría aprenderlo un informático o incluso un estadístico.

La correcta asimilación de este capítulo no es banal, pues debería abrirle el camino hacia una verdadera **eficacia operacional acerca del uso de R como data scientist**. La eficacia operacional de la que hablamos aquí no significa un dominio de R, un uso ortodoxo de la filosofía de este lenguaje o la garantía de obtener siempre el mejor rendimiento. En cambio, imagine más bien que esta toma de contacto le va a permitir **expresar todas sus ideas, comprender el código de sus colegas** y los ejemplos disponibles en la literatura científica y en la ayuda de los paquetes R a su disposición.

1. R, una calculadora eficaz

Los comandos descritos más adelante pueden ejecutarse directamente en la consola de RStudio, pero le recomendamos introducir este código en un script de R en RStudio y, a continuación, ejecutarlo línea a línea para impregnarse bien de su sintaxis.

Tras cada línea, observe atentamente la evolución de las variables en la ventana «Environment» (en la parte superior derecha de RStudio, por ejemplo). Para progresar en el dominio del lenguaje, es imprescindible plantearse pequeños retos personales y, al menos, modificar el código que se le presenta para alcanzar lo que realmente espera de dicho código.

Algunos cálculos sencillos:

```
10*(1+1+1.5) # = 35      # cálculos  
  
10**2          # = 100      # cuadrado  
10^2           # ídem  
  
100**(1/2)     # = 10      # potencia  
100^(1/2)      # ídem
```

Observe el * para la multiplicación, el punto como separador decimal, las dos formas de expresar una potencia y el signo # para definir un comentario no interpretado por la máquina.

Se dispone de numerosas funciones y de la constante pi:

```
sqrt(100)      # = 10      # raíz  
pi            # = 3.1416    # pi  
cos(pi)       # = -1      # cos  
sin(pi/2)     # = 1       # sin  
exp(1)         # = 2.71882  # exponencial
```

```
log(1)      # = 0          # logaritmo neperiano
```

Se dispone de funciones de redondeo:

```
round(2.566)    # redondeo a un entero
round(pi,0)      # ídem
round(pi,2)      # redondeo a dos cifras decimales tras la coma
```

Que devuelven, respectivamente:

```
[1] 3
[1] 3
[1] 3.14
```

Delante de cada línea de resultado, se encuentra un número de línea [1], esta es una herramienta práctica cuando el resultado se reparte en varias líneas. No tenga en cuenta de momento este número de línea.

Para asignar un valor a una variable, a los desarrolladores R no les gusta utilizar el signo **=** como se hace en otros lenguajes, si bien esto funciona. Se prefiere utilizar **<-**. Por ejemplo, para asignar el valor 100 a la variable **a**:

```
a <- 100
```

Visualicemos el contenido **a**, bien invocando **a**, o bien utilizando la función **print()**:

```
a
print(a)
```

En ambos casos se obtiene:

```
[1] 100
```

Evidentemente, ahora puede realizar todas las operaciones a su disposición sobre estas variables numéricas y almacenarlas como mejor le convenga.

A continuación vamos a utilizar variables de un tipo particular, los vectores.

2. R, un lenguaje vectorial

Una gran parte de las operaciones que podemos realizar sobre una variable clásica pueden realizarse sobre una combinación de varias variables que se denomina **vector** en R. Por otro lado, las variables unitarias que hemos manipulado más arriba son también **vectores**, pero son simplemente de dimensión 1.

Las variables de tipo **vector** se llaman así porque ciertos aspectos de su manipulación las hacen parecerse a los vectores en matemáticas. Cuando estos **vectores** están formados por números, es posible multiplicarlos por un valor real, sumarlos o restarlos entre sí. También es posible aplicarles distintas funciones. Veamos esto.

Para crear un **vector**, la forma más común es utilizar la función **c()**, que tiene como objetivo combinar varios

elementos de la misma naturaleza para construir un **vector**.

```
v <- c(10,20,30) # un vector  
v  
length(v) # longitud del vector
```

```
[1] 10 20 30  
[1] 3
```

La longitud de este **vector** es igual a 3.

Mostremos que la variable **a** es también un **vector**.

```
length(a)  
is.vector(a)
```

```
[1] 1  
[1] TRUE
```

El interés de esta estructura va a ponerse de manifiesto, observe lo fácil que es trabajar sobre un **vector**, por ejemplo:

```
2*v+1 # 2*x+1 sobre cada componente x de v  
v**2 # cuadrado de cada componente  
log(v) # log de cada componente
```

Que devuelven, respectivamente:

```
[1] 21 41 61  
[1] 100 400 900  
[1] 2.302585 2.995732 3.401197
```

La sintaxis habría sido la misma para un **vector** que hubiera contenido millones de elementos, y en este caso con muy buenas condiciones de rendimiento (velocidad de ejecución y memoria utilizada).

A continuación vamos a realizar operaciones sobre dos **vectores**.

```
w <- c(1,2,3) # otro vector  
v-w # resta miembro a miembro  
v*w # multiplicación miembro a miembro  
v/w # división miembro a miembro  
v%*%w # producto escalar
```

```
[1] 9 18 27  
[1] 10 40 90
```

```
[1] 10 10 10  
[1] 140      [aquí se obtiene un número, es decir, un escalar,  
de longitud 1]
```

Observe que el producto normal `*` es, aquí, el producto miembro a miembro y no tiene nada que ver con el producto clásico de vectores en matemáticas. Por el contrario, puede realizar un producto escalar de **vectores** (como productos miembro a miembro) utilizando el operador `%*%`. La noción de producto escalar se explicará más adelante en este libro.

Existen funciones que recopilan todos los miembros de un **vector** (suma, media...).

```
sum(v)      # = 60  suma  
mean(v)     # = 20  media  
min(v)      # = 10  mínimo  
max(v)      # = 30  máximo  
sd(v)       # = 10  desviación típica  
median(v)   # = 20  mediana
```

También es posible acceder a cada miembro de un **vector**.

```
u <- c(1,2,3,4,5,6,7,8) # otro vector  
u[2]                  # segunda componente
```

```
[1] 2
```

Es posible extraer varios miembros en una única operación.

```
u[3:5]          # nuevo vector  
               # resultado de las componentes 3 a 5
```

```
[1] 3 4 5
```

Es posible interactuar sobre un miembro.

```
u[8] <- 80          # asignación de una componente  
u
```

```
[1] 1 2 3 4 5 6 7 80
```

También es posible asignar un mismo valor a varios miembros.

```
u[1:5] <- 1          # asignación de cinco componentes  
u
```

```
[1] 1 1 1 1 1 6 7 80
```

Existen funciones muy potentes de manipulación de vectores.

3. Funciones que trabajan sobre vectores

Estas funciones están optimizadas y pueden utilizarse sobre **vectores** de gran tamaño: evite escribir bucles innutiles cuando disponga de funciones nativas en R o paquetes especializados que le ofrecen la garantía de un buen rendimiento sobre estructuras de datos de grandes dimensiones.

Para probar estas funciones, declaremos tres **vectores**.

```
v <- c(10,20,30,30,60,50)      # un vector
w <- c(20,10,31,31,61,51)      # otro vector
u <- c(5 ,5 ,5 ,32,62,49)      # otro vector
```

a. Un primer análisis rápido de los datos

Vamos a responder a una primera preocupación: ¿cuál es el contenido de este **vector**?

```
str(v)                      # echar un vistazo a los datos
```

```
num [1:6] 10 20 30 30 60 50
```

Se trata de un **vector** numérico de dimensión 6, cuyos miembros podemos ver.

Otra preocupación sería saber si todos los datos están informados.

En R, un dato que falta se indica mediante **NA** (*Not Available*). Se cuenta el número de **NA** de la siguiente manera:

```
sum(is.na(v))              # num de valores faltantes
```

```
[1] 0
```

Este código aplica en primer lugar la función **is.na()** al vector, lo que devuelve un **vector** de valores **FALSE** o **TRUE**, y a continuación se realiza la suma de los miembros del **vector** resultante aplicando la función **sum()**, sabiendo que R va a considerar en esta suma que los valores **FALSE** valen 0 y que los valores **TRUE** valen 1.

Para probar esto vamos a fabricar un vector que contenga tres **NA**.

```
v_ <- c(NA,v,NA,NA)      # un vector con tres valores faltantes
v_
sum(is.na(v_))          # num de valores faltantes
```

```
[1] NA 10 20 30 30 60 50 NA NA  
[1] 3
```

También es posible plantear la cuestión del rango de valores del **vector**.

Calculemos los valores extremos de un **vector**; el resultado es un **vector** de dimensión 2.

```
range(v) # min y max del vector
```

```
[1] 10 60
```

Desafortunadamente, resulta muy frustrante aplicar la misma función a un **vector** que posee valores faltantes.

```
range(v_) # min y max del vector ¡FALLA!
```

```
[1] NA NA
```

Muchas funciones de R disponen de una opción que permite esquivar este problema; se trata de la opción **na.rm=TRUE**.

```
range(v_ , na.rm = TRUE) # sin tener en cuenta los NA
```

```
[1] 10 60
```

Como se esperaba, hemos encontrado los dos valores extremos.

 Preste atención: muchas de las anomalías (bugs) de los programas provienen de un mal tratamiento de los NA, o de un tratamiento implícito de los NA que no se ha documentado.

b. Algunas estadísticas sencillas sobre los vectores

Una forma cómoda de comprender el aspecto de la distribución de los valores de un **vector** consiste en ordenar sus valores en cuatro paquetes equivalentes (25 %) para poder observar solamente el 25 % de los datos, el 50 % o el 75 % de los datos situados entre «tal y tal valor». Las fronteras interiores de nuestros cuatro paquetes se llaman cuartiles (las fronteras exteriores son, evidentemente, los valores extremos del vector). Veamos esto sobre nuestro vector **v**.

```
quantile(v) # cuantiles de v
```

```
0% 25% 50% 75% 100%  
10.0 22.5 30.0 45.0 60.0
```

A continuación puede afirmar (por ejemplo) que el 25 % de estos datos poseen valores comprendidos entre 30 y 45, o que el 50 % de los datos poseen valores situados entre 22.5 y 45.

 Observe que el valor medio (cuartil 50 %) no se corresponde con la media (salvo en casos particulares).

Puede determinar sus propias fronteras (es decir, sus propios cuantiles). Por ejemplo, «al estilo Pareto», es decir, buscando el 80-20, podríamos imaginar dos fronteras, una en el 10 % y la otra en el 90 %.

```
quantile(v, probs =c(0,0.1,0.9,1)) # 80/20
```

```
0% 10% 90% 100%
10   15   55   60
```

Aquí podríamos afirmar que el 80 % de los valores del **vector** están comprendidos entre 15 y 55 (puesto que el 10 % son inferiores a 15, y el 10 %, superiores a 55).

Habrá observado que el valor que se pasa en el parámetro **probs** es en sí mismo un vector que es la combinación ordenada de cuatro valores (dos valores extremos **0** y **1** para expresar el 0 % y el 100 % y dos valores intermedios, **0.1** para el 10 % y **0.9** para el 90 %). Este es un hábito sintáctico muy común y práctico en R, que consiste en pasar **vectores** como parámetros de una función, lo que evita tener que gestionar una larga lista de parámetros intermedios de número variable!

Podemos obtener un resumen rápido (cuartiles y media) mediante la función genérica **summary()**, que puede aplicarse sobre muchos objetos.

```
summary(v)                      # resumen
```

```
Min. 1st Qu. Median     Mean 3rd Qu.      Max.
10.00    22.50   30.00    33.33    45.00    60.00
```

Tras observar las estadísticas básicas sobre un **vector**, se revela rápidamente que resultaría útil echar un vistazo a la dependencia entre **vectores**. Identifiquemos el nivel de correlación lineal entre dos **vectores**.

```
cor(v,w)                      # coeficiente de correlación entre vectores
```

```
[1] 0.9433573 [son más o menos proporcionales]
```

A continuación veremos cómo dominar una de las manipulaciones más corrientes sobre un **vector**, la ordenación.

c. Ordenar un vector

Ordenar un **vector** consiste en transformarlo en función de una relación de orden entre sus miembros. Se dispone de funciones especializadas y eficaces (pero, preste atención, la ordenación es una función costosa).

```
sort(v) # vector ordenado en orden creciente
```

```
[1] 10 20 30 30 50 60
```

Habríamos podido ordenarlo según un orden decreciente. Observe la manera de definir una opción en una función. Si no se define nada, se utiliza la opción por defecto (aquí **decreasing = FALSE**). Una función como **sort()** dispone de diversas opciones que puede consultar en la ayuda (**help**), en la zona inferior derecha de RStudio.

```
sort(v, decreasing = TRUE) # vector ordenado en orden  
# decreciente
```

```
[1] 60 50 30 30 20 10
```

En lugar de fabricar un nuevo **vector** ordenado, en ocasiones resulta interesante obtener un **vector** que define cada situación del **vector** de origen (es decir, un «puntero») y que define el orden del elemento considerado respecto a los demás. Esto resulta algo más sutil, pero evita tener que duplicar grandes estructuras en memoria o perder el **vector** original. Piense en ello cuando quiera realizar **ordenaciones sobre grandes volúmenes**.

```
order(w) # devuelve punteros ordenados
```

```
[1] 2 1 3 4 6 5 [ comparar con w: [1] 20 10 31 31 61 51 ]  
[ el elemento más pequeño es el segundo de w ]  
[ el siguiente es el primero de w ... ]  
[ el más grande es el quinto de w ]
```

Observe que una ordenación normal decide el orden en el que se devuelven los valores que son iguales «sin consultarle». Por ejemplo, en el caso del vector **w** abordado aquí, encontramos el valor **31** en los rangos **3** y **4**, como resultado de la aplicación de la función **order()**. En lugar de **2-1-3-4...** habríamos podido querer obtener **2-1-4-3...**

¿Por qué resulta molesto?

Imagine que ha organizado una competición deportiva, y dos puntuaciones iguales a **31** habrían merecido el mismo resultado y obtener el mismo rango (ex aequo).

Para resolver este problema, es necesario utilizar la función **rank()**.

```
rank(w, ties.method="min") # vector de rangos  
# min se utiliza a menudo para  
# las competiciones deportivas
```

```
[1] 2 1 3 3 6 5 [ a comparar con el resultado de order() : ]  
[1] 2 1 3 4 6 5 [ que diferenciaba de manera ficticia dos miembros]
```

En términos generales, hay que dar una gran importancia a todo lo que puede generar interpretaciones abusivas o procesamientos erróneos.

La elección del método de ranking no es banal. Si se quiere poner de relieve el hecho de que los primeros valores son más «significativos» o «importantes», se tenderá a utilizar principalmente la opción **max** (por otro lado es la práctica más habitual), por ejemplo.

```
rank(w, ties.method="max")      # vector de rangos  
                                # max se utiliza a menudo en  
                                # las data sciences
```

```
[1] 2 1 4 4 6 5
```

Se confirma aquí que los ex aequo están definidos con un rango igual a **4**. Esto los «aleja» un poco más del «rendimiento» de los dos primeros.

Veremos otras funciones que permiten ahorrar mucho tiempo evitando realizar grandes bucles inútiles.

d. Diversas funciones con suma, producto, min y max

Vamos a utilizar los siguientes **vectores** para comprender cuatro nuevas funciones.

```
> v  
[1] 10 20 30 30 60 50  
> w  
[1] 20 10 31 31 61 51  
> u  
[1] 5 5 5 32 62 49
```

Sobre varios **vectores**, tres en este caso, vamos a calcular el máximo o mínimo miembro a miembro.

```
pmax(v,w,u)          # valores máximos miembro a miembro
```

```
[1] 20 20 31 32 62 51
```

```
pmin(v,w,u)          # valores mínimos miembro a miembro
```

```
[1] 5 5 5 30 60 49
```

Es posible realizar sumas o productos **sucesivos** de los términos de un **vector**.

```
cumsum(v)            # sumas acumuladas  
cumprod(v)          # productos sucesivos
```

```
[1]      10      30      60      90     150     200
[1] 1.00e+01 2.00e+02 6.00e+03 1.80e+05 1.08e+07 5.40e+08
```

De manera más sutil, pero basándose en el mismo principio, intente aplicar sucesivamente max o min (en lugar de aplicar sucesivamente la suma o la multiplicación).

```
cummax(w)          # máximo entre miembros
                   # considerados y miembros precedentes
cummin(w)          # ídem con min
```

```
[1] 20 20 31 31 61 61
[1] 20 10 10 10 10 10
```

R permite manipular otros tipos de estructuras de datos y de variables, que son útiles habitualmente.

4. Tipos de datos simples

El dato más simple que existe es el dato booleano: verdadero-falso.

R, como ocurre en otros lenguajes, le permite almacenar valores booleanos y utilizarlos en numerosos cálculos.

a. Los booleanos

En primer lugar vamos a obtener valores booleanos mediante comparaciones simples de variables.

```
# booleanos - comparaciones          #
a <- 1
b <- 2

(a == 1)           # TRUE
(a == b)           # FALSE
(a <= b)          # TRUE
```

```
[1] TRUE
[1] FALSE
[1] TRUE
```

Como ocurre con los demás tipos simples, los booleanos (en inglés: *logical*) pueden combinarse en un **vector**.

Tras crear dos **vectores** de booleanos, vamos a combinarlos.

```
A <- c(TRUE,TRUE,FALSE,FALSE)
B <- c(TRUE,FALSE,TRUE,FALSE)
A & B                      # tabla de verdad "Y"
```

```
[1] TRUE FALSE FALSE FALSE
```

Se corresponde con el resultado esperado para la tabla de verdad "Y", además:

A B	# tabla de verdad "O"
-------	-----------------------

```
[1] TRUE TRUE TRUE FALSE
```

! A	# no-A
xor(A,B)	# tabla de verdad O exclusivo
!A B	# tabla de implicación A==>B

```
[1] FALSE FALSE TRUE TRUE
[1] FALSE TRUE TRUE FALSE
[1] TRUE FALSE TRUE TRUE
```

El resultado de una comprobación puede almacenarse en una variable.

c <- (a > b)	# almacenar el resultado de una comprobación
c	

```
[1] FALSE
```

A continuación vamos a utilizar una sintaxis muy eficaz que nos va a permitir actuar sobre el conjunto de un **vector, que sin duda debemos recordar.**

v <- c(10,20,30,30,60,50)	# un vector
t <- (v > 30)	# vector resultante de la comprobación
t	# miembro a miembro

```
[1] FALSE FALSE FALSE FALSE TRUE TRUE
```

Hemos obtenido un vector que puede utilizarse como **filtro** para realizar alguna otra operación.

w <- v[(v>30)]	# solo guardamos los miembros
	# con expresión TRUE
w	

```
[1] 60 50
```

Este método nos permitirá manipular grandes vectores de datos con una gran eficacia en términos de rendimiento y en términos de legibilidad del código, todo ello **sin utilizar ningún bucle explícito.**

Utilizando la instrucción **which**, podemos obtener los índices correspondientes a una selección:

```
which(v == 30)      # encuentra los índices para los cuales  
                    # el miembro es igual a 30
```

```
[1] 3 4
```

La selección puede realizarse mediante una función vectorial básica.

```
which(v == max(v))  # encuentra los índices para los cuales  
                    # el miembro posee un valor máximo  
  
which(v == min(v))  # idem pero buscando el mínimo
```

```
[1] 5  
[1] 1
```

Resulta muy sencillo transformar un **vector** de valores booleanos en un **vector** de 0 y 1.

```
s <- 1*t          # transformación en un vector de 1,0  
s
```

```
[1] 0 0 0 0 1 1
```

Por último, disponemos de dos funciones para identificar si todos los valores de un **vector** son verdaderos o no, o si al menos uno es verdadero.

```
v <- c(10,20,70,30,60,50)  # un vector  
all(v > 5)                 # ¿los valores son superiores a 5?  
any(v < 5)                 # ¿los valores son inferiores a 5?
```

```
[1] TRUE  
[1] FALSE
```

R ofrece a su vez la posibilidad de manipular conjuntos, lo que completa el arsenal de capacidades de procesamiento de elementos lógicos.

b. Conjuntos

Se trata de **vectores** que soportan la noción de conjunto en R. Como un conjunto no está ordenado, los operadores no tienen en cuenta los índices de los distintos elementos de los conjuntos seleccionados. Disponemos de la unión y la intersección.

En un conjunto, cada elemento es, por naturaleza, único. Si desea estar seguro de que sus **vectores** se comportan como conjuntos, no dude en aplicar la función **unique()**.

```
H <- unique(c("e", "g", "g", "h", "h", "h"))
H
```

```
[1] "e" "g" "h"
```

La sintaxis es trivial.

```
# conjuntos                                #
P <- c("e", "f", "g", "h")
Q <- c("g", "h", "i", "j")
union(P, Q)
```

```
[1] "e" "f" "g" "h" "i" "j"
```

```
intersect(P, Q)
```

```
[1] "g" "h"
```

Se dispone también de la diferencia (que no es simétrica).

```
setdiff(P, Q)
setdiff(Q, P)
```

```
[1] "e" "f"
[1] "i" "j"
```

Es posible obtener una diferencia simétrica de la siguiente manera.

```
union(setdiff(P, Q), setdiff(Q, P))
```

```
[1] "e" "f" "i" "j"
```

Se dispone también de un operador de pertenencia vectorizado extremadamente potente, que indica para cada uno de los elementos del primer conjunto si pertenece al segundo conjunto.

```
H
P
```

```
H %in% P  
P %in% H
```

```
[1] "e"   "g"   "h"  
[1] "e"   "f"   "g"   "h"  
  
[1] TRUE  TRUE  TRUE  
[1] TRUE  FALSE TRUE  TRUE
```

Si quiere obtener los índices correspondientes, utilice **which()**:

```
which(P %in% H)
```

```
[1] 1 3 4
```

Los conjuntos no representan la más genérica de las estructuras utilizadas en los lenguajes informáticos. En muchos lenguajes, se dispone de la noción de lista y la noción de diccionario, que permiten soportar diversos algoritmos y apuntar a variables de cualquier naturaleza. En R, ambas nociones se abordan bajo una noción común de lista.

c. Listas

Una primera manera de utilizar las listas puede ser utilizarlas como diccionarios «clave-valor» (como en el lenguaje Python, por ejemplo).

```
# listas #  
  
l <- list (nombre = "Pedro", # lista clave-valor  
            edad = 25,  
            vect = v)  
l$nombre # valores  
l$edad  
l$vect
```

```
> l$nombre  
[1] "Pedro"  
> l$edad  
[1] 25  
> l$vect  
[1] 10 20 70 30 60 50
```

Observe cómo se accede al valor, indicando el nombre de la lista, y a continuación la clave precedida del símbolo \$.

Los valores de una lista pueden incluir otra lista.

De este modo, es posible acceder a los valores de una lista mediante su índice, y el uso de una clave no resulta obligatorio en las listas.

```
l[[1]]                      # por índice  
l[[2]]  
l[[3]]
```

Esto devuelve el mismo resultado.

iObserve el doble corchete!

La lista no transforma la naturaleza de su contenido, siempre es posible acceder directamente.

```
is.vector(l[[3]])          # es, efectivamente, un vector  
l[[3]][2]                  # he aquí el segundo elemento
```

```
[1] TRUE  
[1] 20
```

En data sciences a menudo hay que utilizar datos que representan la pertenencia a una clase, o hay que tratar de determinar clases (problema de clusterización, que es una forma de aprendizaje no supervisado). Disponemos en R de dos tipos dedicados a ello (un tipo no ordenado y un tipo ordenado). Podríamos pensar que el hecho de enumerar las clases debería ser suficiente, pero en realidad los algoritmos escritos en R tienen en cuenta la información que se les proporciona habiendo declarado explícitamente una «feature» como compuesta de una variable de clase (se llama «factor» para diferenciar esta noción de la noción de clase objeto, que veremos más adelante).

d. Factores

Clases no ordenadas

Tomemos como ejemplo los colores, sin una semántica adicional particular (longitud de onda, asociación de un sentimiento a un color...), estas clases no necesitan incluir una noción de orden que podría perturbar la interpretación de nuestros datos. Sin embargo, se deduce fácilmente que resulta interesante enumerar estas clases, en lugar de nombrarlas, pues en efecto almacenar el nombre de una clase es más costoso que almacenar un número. El número asociado a una clase se denomina **factor** y la clase no ordenada se llama **level**.

Vamos a crear un **vector** que incluya valores de clase y ver qué es.

```
col <- factor(c("azul", "verde",  
                 "rojo", "azul", "verde")) # factor = sin orden  
col
```

```
[1] azul verde rojo azul verde
```

```
Levels: azul rojo verde
```

Hemos obtenido el **vector** compuesto de cinco miembros que utilizan los tres nombres de colores; observe que no incluyen comillas, pues no se trata de cadenas de caracteres, sino de nombres de **level**. Debajo obtiene la lista de los **levels** utilizados en el **vector**.

Observemos ahora los **factor** asociados.

```
str(col)      # muestra los levels y su índice
```

```
Factor w/ 3 levels "azul","rojo",...: 1 3 2 1 3
```

Se comprueba que el **vector** está codificado de manera eficaz (utilizando solamente tres valores enteros: 1, 2 y 3). Debe **dar preferencia a esta codificación siempre que sea posible**.

El acceso a la información no plantea ningún problema específico.

```
col[2]      # verde
```

```
[1] verde  
Levels: azul rojo verde
```

R tiene además la inteligencia de recordarle los distintos niveles posibles de esta información, de modo que sabe que aquí se tiene el verde, pero entre solamente tres colores disponibles.

Si se desea conocer todos los niveles de **col**, basta con utilizar **levels()**.

```
levels(col)      # los levels
```

```
[1] "azul"  "rojo"  "verde"
```

Para comprobar el nivel de un miembro concreto, se escribe el nombre del **level** entre comillas, pues en caso contrario R actuará como si quisiera invocar a un objeto llamado **verde**, que contendría eventualmente el valor del factor que desea comprobar.

```
t <- (col[2] == "verde") # comprueba si verde  
t
```

```
[1] TRUE
```

Para actuar globalmente sobre **vectores**, necesitamos a menudo índices de miembros que se corresponden con una selección concreta. Por ejemplo, aquí los índices de los miembros de **col** que tienen el nivel **verde**, o a la inversa, los índices que no tengan el nivel **verde**.

```
which(col == "verde")    # índice si verde  
which(col != "verde")    # índice si no verde
```

```
[1] 2 5
```

En muchos casos, como con la clasificación, el uso de diversos algoritmos no adaptados a los valores continuos, o simplemente por motivos de dificultad de interpretación o de rendimiento, resulta útil dividir las series de números en intervalos que se convertirán en clases sobre las que se posará su ojo experto. Evidentemente, la manera de operar **esta división puede influir notablemente en los resultados obtenidos** y tendrá que probar diversas divisiones. La práctica que vamos a abordar ahora le será de gran utilidad.

División de una serie en factores

La primera idea podría ser dividir su serie según intervalos escalonados mediante un paso constante; por ejemplo, para valores comprendidos entre 0 y 1, podríamos plantear los siguientes cuatro niveles: [0;0.25],]0.25;0.5],]0.5;0.75] y]0.75;1].

Escoger un nivel de amplitud equivalente puede resultar seductor por su aparente simplicidad o si las fronteras de estos niveles se utilizan habitualmente en el ámbito de estudio. Sin embargo, eso plantea un problema de representatividad; en efecto, nada prueba que el conjunto de los elementos de cada clase sea globalmente igual. Imagine que una clase contuviera el 80 % de los efectivos; habría perdido bastante información en cuanto a la feature considerada, el poder de separabilidad de esta feature resultará demasiado débil.

De modo que si se impone un número de clases concreto (4, 10, 100...) parece necesario componerlo con **efectivos más o menos iguales**, en particular si hay pocas clases. Observe que si sus clases son demasiado finas y no están basadas en los efectivos existentes, puede fabricar **accidentalmente las clases que estarán vacías**, lo que añade una complejidad innecesaria a su problema!

Cuando no disponga de información específica y la distribución de su población no parezca netamente multimodal, es decir, cuando los efectivos no posean varios picos netos en función de su valor (por ejemplo: la famosa distribución normal), la idea más sencilla sería apoyarse en cuantiles más o menos numerosos en función de lo fino que se quiera realizar el análisis (sabiendo que si es demasiado fino pueden generarse diversos problemas). Aquí, nuestro ejemplo se basará en cuartiles, aunque hemos visto más arriba que la función `quantile()` nos permite trabajar con separaciones más finas y sobre clases de efectivos modulables a su voluntad.

En ocasiones, cuando la distribución sea multimodal, o cuando posea un buen conocimiento del significado de los datos en el contexto de su estudio, podrá realizar una división «a medida». Por ejemplo, en el caso de que un valor represente un umbral conocido, es evidente que puede utilizar este umbral (por ejemplo: ingresos inferiores o no respecto a un umbral que desencadena ventajas sociales, edad legal de jubilación, porcentaje tolerable de un producto nocivo en un alimento...).

Utilizando diversas nociones vistas más arriba, el código propuesto aquí posee ciertas sutilidades y algunas novedades. Hemos escogido no eludir las dificultades y proponerle estudiar un código que le será útil en otras circunstancias. Si encuentra mucha dificultad en comprender las siguientes líneas de código, vuelva a ellas más adelante tras haber obtenido una mayor experiencia a lo largo de este libro.

Nuestra primera tarea consiste en crear un **vector** de valores más o menos continuos para disponer de algo para dividir en intervalos.

La primera instrucción `set.seed(0)` inicializa el generador de números aleatorios de R, basándose en un valor cualquiera (la semilla = seed). Aquí hemos escogido el 0.

Esta práctica le resultará muy útil, pues `set.seed()` inicializa siempre el generador de números aleatorios de una manera concreta en función de la semilla que le proporcione, si bien sus colegas o usted mismo partirá de la misma aleatoriedad con cada uso de su código y podrá comparar sus resultados con detalle de una vez a la siguiente.

La siguiente instrucción genera un **vector** de 100 miembros cuyos valores están repartidos aleatoriamente, pero bajo la forma de una distribución normal (desviación típica igual a 1, con un pico efectivo alrededor del 0 que se corresponde con la media de esta distribución).

```
# división de una serie en factores          #
set.seed(0)           # inicialización del número aleatorio
v <- c(rnorm(100))    # creación del vector de prueba
summary(v)            # resumen
sd(v, na.rm = TRUE)   # desviación típica
```

```
num [1:100] 1.263 -0.326 1.33 1.272 0.415 ...
Min.     1st Qu.    Median      Mean      3rd Qu.      Max.
-2.22400 -0.56940 -0.03296  0.02267  0.62540  2.44100
[1] 0.8826502
```

Vemos que obtenemos 100 valores numéricos, comprendidos entre **-2.2** y **2.5**, con una media en torno a **0 (0.023)** y una desviación típica no muy alejada de **1 (0.88)**. El número de valores pedidos, al ser demasiado débil, hace que la aleatoriedad no se «adhiera» en detalle con la petición explícita realizada mediante **rnorm (100)**... No nos preocuparemos, pues se trata de un nivel de aleatoriedad compatible con los objetivos y su uso hará más robustas sus conclusiones.

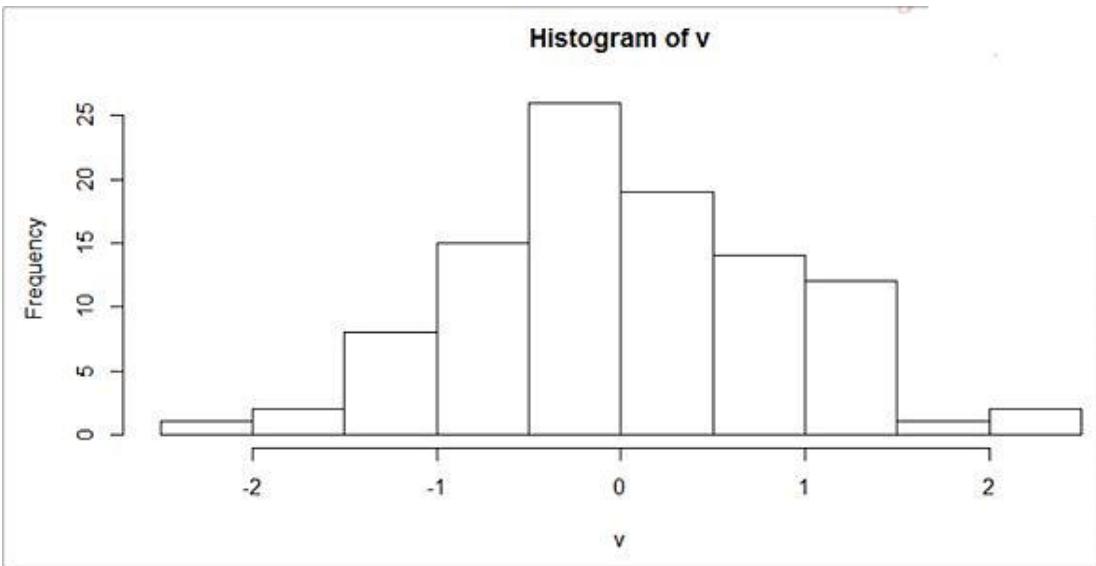
Podemos representar visualmente esta distribución, de hecho debemos hacerlo, pues resulta una práctica básica que sirve para visualizar los datos, sin dejarse influir no obstante demasiado por su aspecto visual: esto evitará muchos desengaños.

 Tras cada etapa de su trabajo, imagine lo que podría visualizar y a qué podría parecerse dicha visualización. Se sorprenderá por su pensamiento a priori constatando la diferencia entre lo que había imaginado y la realidad de sus datos. Esta práctica es muy reveladora.

Para obtener un histograma sencillo de su distribución, que le permita obtener la frecuencia de los valores de **v** en función de sus valores, basta con una pequeña instrucción.

```
hist(v)          # distribución obtenida
```

El resultado aparece en una ventana (en la parte inferior derecha) de RStudio. Haciendo clic en el zoom, y luego haciendo más grande la ventana, podrá obtener una mejor visión de este.



Distribución de los valores del vector v

A nivel global, esto es lo que se esperaba; ni intente en ningún caso interpretar con detalle este histograma, así como el hecho de que las columnas cercanas al 0 sean muy diferentes. Comprobará más adelante que existe una gran sensibilidad del aspecto visual de los histogramas respecto a los parámetros que se les aplica (es decir, la anchura de las bandas).

Si de manera muy aproximada suma todos los valores debajo de la curva formada por la cima de las 10 bandas del histograma, de izquierda a derecha ($1 + 2 + 8 + 15 + 25 + 19 + 14 + 13 + 1 + 2$), obtendrá 100. La superficie debajo de la curva de una distribución en frecuencia que cubre el espectro de todos los valores de un vector es, de hecho, y debido a su construcción, siempre igual a 1 (es decir el 100 %).

La función **cut()** nos permite realizar nuestra división y su opción **breaks** nos permite declarar los valores frontera, aunque siempre vamos a tropezar con una pequeña trampa que tendremos que corregir.

```
# división en cuartiles
w <- cut(v, breaks = quantile(v), labels = FALSE)
w
```

```
[1] 4 2 4 .../... NA 1 3 3 1 2 1 3 1 3 .../...
[algunos valores entre los valores ]
```

Algo no funciona bien, aparece un valor **NA** en **w**.

Por suerte, hemos reparado en el **NA**. ¡Pero no habríamos tenido la misma suerte si el vector tuviera un millón de líneas!

Un mejor método para comprender rápidamente nuestros datos consiste en utilizar la función **table()**, que devuelve el número de elementos por clase (si faltan es porque existe uno o varios **NA**).

```
table(w) # reparto de efectivos
sum(table(w)) == length(w) # no iguales
```

```
1 2 3 4  
24 25 25 25      [ efectivos de 24 en la primera clase]  
  
[1] FALSE        [ efectivos totales no iguales a la longitud de w]
```

Veamos esto.

```
which(is.na(w))      # ¡uy!, un valor desconocido
```

```
[1] 59
```

El 59º valor es un **NA**, ¿a qué se corresponde?

```
w[59]  
v[59]          # ¡es uno de los bordes!
```

```
[1] NA  
[1] -2.2239
```

Existe, de hecho, una función particular dedicada a buscar la **primera** posición de un **predicado lógico** (es decir, algo que devuelva verdadero o falso); esta función posee a su vez un parámetro para devolver la última posición.

```
Position(is.na,w)  # alternativa que devuelve la primera posición
```

```
[1] 59
```

El valor de **v[59]** es, simplemente, el valor mínimo de los miembros del vector que «misteriosamente» no se ha tenido en cuenta porque el primer intervalo está «abierto» a la derecha.

► Comprobará que el comportamiento de las funciones escritas por sus colegas no se corresponden siempre a lo que supone. Conviene, por tanto, observar y estar atento sistemáticamente a los resultados intermedios de sus cálculos en relación con lo que usted espera.

¿Cómo corregir este defecto? Una idea sencilla sería extender ligeramente el primer intervalo hacia la «izquierda», un poco por debajo del valor mínimo. Para resistir a casos en los que el problema ocurra por la «derecha», haremos lo mismo con el valor máximo. De modo que vamos a definir nuevos bordes.

```
bordes <- c(quantile(v)[1]- 0.0001*abs(min(v)),  
           quantile(v)[2],  
           quantile(v)[3],  
           quantile(v)[4],  
           quantile(v)[5]+ 0.0001*abs(max(v))  
           )  
bordes
```

```
0%          25%          50%          75%          100%
-2.22412266 -0.56941871 -0.03296148  0.62535107  2.44160877
```

Comprobamos que los valores extremos de los cuartiles 0 % y 100 % se han ampliado ligeramente. Veamos si nuestro problema se ha resuelto.

```
w <- cut(v, breaks = bordes, labels =c("Q1","Q2","Q3","Q4"))
w
which(is.na(w))
w[59]
```

```
[1] Q4 Q2 Q4 Q4 Q3 Q1 Q1 Q2 Q3 Q4 .../...
Levels: Q1 Q2 Q3 Q4

integer(0)           [sin valor NA]

[1] Q1               [ el 59º valor está dentro de Q1]
Levels: Q1 Q2 Q3 Q4           [ entre los 4 levels ]
```

Hemos repartido nuestros valores sobre cuatro **levels** denominados **Q1, Q2, Q3 y Q4**.

Comprobemos si estos niveles se corresponden con lo que esperamos.

Vamos a construir la sintaxis de verificación paso a paso.

```
which(w == "Q1")      # Índices de Q1
```

```
[1]  6   7  12 13 18 20 28 29 34 44 45 46 47 54 59 60 63 65 67 73
79 80 81 92 94
```

Esta instrucción nos devuelve todos los índices correspondientes a **Q1**. Lo que aplicado a **v** nos permite extraer todos los valores de **v** que están «en» **Q1**.

```
v[which(w == "Q1")]      # valor original de Q1
```

```
[1] -1.5399500 -0.9285670 -0.7990092 -1.1476570 -0.8919211.../...
```

Utilizando esta técnica, basta ahora con comprobar si los bordes extremos de los valores clasificados en **Q1, Q2, Q3 y Q4** se corresponden con los cuartiles esperados (utilizando la función **range()**).

```
quantile(v)           # para recordar
range(v[which(w == "Q1")]) # Q1
range(v[which(w == "Q2")])
range(v[which(w == "Q3")])
range(v[which(w == "Q4")])
```

```

0%          25%         50%         75%        100%
-2.22390027 -0.56941871 -0.03296148  0.62535107  2.44136463

[1] -2.2239003   -0.6490101
[1] -0.54288826 -0.05487747
[1] -0.01104548  0.61824329
[1]  0.6466744   2.4413646

```

Los intervalos de nuestros valores clasificados en sus respectivos cuartiles son, efectivamente, compatibles con la definición de cuartiles.

Hasta aquí hemos considerado niveles no ordenados, veamos ahora la sintaxis que permite gestionar niveles ordenados.

Clases ordenadas

La sintaxis difiere ligeramente, pero lo que hemos aprendido hasta el momento sigue siendo útil.

```

orden <- factor(c("pequeño", "mediano", "grande"),
                 ordered=TRUE,                      # creación de orden
                 )
orden

```

```

[1] pequeño mediano grande
Levels: grande < mediano < pequeño

```

Aquí no pasa lo que habríamos esperado, pues R ha utilizado el orden alfabético y no el orden de los parámetros. Una solución sencilla consiste en prefijarlos con caracteres que nos permitan ordenar correctamente.

```

orden <- factor(c("0_pequeño", "1_mediano", "2_grande"),
                 ordered=TRUE,                      # creación de orden
                 )
orden

```

```

[1] 0_pequeño 1_mediano 2_grande
Levels: 0_pequeño < 1_mediano < 2_grande

```

Comprobemos que esto es operacional.

```

orden[1] < orden[2]                      # TRUE

```

```

[1] TRUE

```

A continuación podemos utilizar los tipos simples que hemos estudiado para construir estructuras más complejas y, sobre todo, tablas.

e. Tablas

Como en una hoja de cálculo, vamos a trabajar con tablas compuestas por los tipos más simples estudiados más arriba (numéricos, factores...).

La tabla más común que vamos a manipular será una tabla de observaciones con las observaciones en las filas y las variables observadas en las columnas. Estudiaremos con detalle las tablas compuestas de dos ejes en los siguientes capítulos bajo la denominación «matrices». De momento nos concentraremos en el uso operacional de las tablas y comprobaremos que disponemos de una gran cantidad de posibilidades de manipulación para nuestras tablas, parecidas a lo que se denomina «tablas dinámicas» o «tablas cruzadas» en el mundo de las hojas de cálculo, o cubos OLAP en el mundo del Business Intelligence.

Para nuestras pruebas, vamos a construir una hoja de cálculo ficticia de observaciones de cuatro columnas. Las tres primeras columnas representan atributos de la observación y la cuarta posee un atributo particular que percibiremos como dependiente de los otros tres. Aquí, para ayudarle a trabajar con las tablas, vamos a construir los términos de la cuarta columna de una forma mecánica y absurda ($100 * \text{columna1} + 10 * \text{columna2} + \text{columna3}$).

En el mundo real, las tres primeras columnas podrían, por ejemplo, representar el género, otra el hecho de haber cursado estudios superiores, la tercera la categoría socioprofesional y la cuarta el número de páginas web consultadas en un sitio...

Creación - matrix()

Creemos la tabla. Primero fabricamos un **vector** no estructurado y, a continuación, construiremos una tabla de tipo **matrix** con cuatro columnas y que posea un número de filas suficiente.

```
o <- c( 1,1,1,111,    # un vector de observaciones
       1,1,2,112,
       1,1,3,113,
       1,2,1,121,
       1,2,2,122,
       1,2,3,123,
       2,1,1,211,
       2,1,2,212,
       2,1,3,213,
       2,2,1,221,
       2,2,2,222,
       2,2,3,223
     )
```

Este **vector** no es más que una lista de valores y no tiene en cuenta la estructura visual filas por columnas que queremos darle ahora.

```
m <- matrix(byrow = TRUE,          # la matriz correspondiente
             nrow   = length(o)/4,
             o)
m
```

```
[,1] [,2] [,3] [,4]
[1,] 1 1 1 111
[2,] 1 1 2 112
[3,] 1 1 3 113
[4,] 1 2 1 121
[5,] 1 2 2 122
[6,] 1 2 3 123
[7,] 2 1 1 211
[8,] 2 1 2 212
[9,] 2 1 3 213
[10,] 2 2 1 221
[11,] 2 2 2 222
[12,] 2 2 3 223
```

Ahora disponemos de una estructura de datos que refleja nuestras observaciones. Como en una tabla, podemos acceder a las celdas de esta tabla.

```
m[2,4] # celda fila 2, columna 4
```

```
[1] 112
```

Podemos acceder a una fila de la siguiente manera (sin indicar la columna).

```
m[2,] # fila 2
```

```
[1] 1 1 2 112
```

Y podemos acceder a los valores de una columna sin indicar la fila.

```
m[,4] # valores traspuestos de la columna 4
```

```
[1] 111 112 113 121 122 123 211 212 213 221 222 223
```

Desafortunadamente, R no conserva la geometría «columna»; esto ocupa menos espacio en la pantalla, pero resulta algo molesto. Si necesita imponer la geometría columna, es preciso utilizar de nuevo la instrucción **matrix()**, que incluye una opción por defecto para transformar las filas de un «vector fila» en columnas.

```
matrix(m[,4]) # columna 4
```

```
[,1]
[1,] 111
[2,] 112
[3,] 113
[4,] .... [y así con el resto ...]
```

Tabla xtabs, tabla cruzada dinámica, cube...

A continuación vamos a fabricar nuestra tabla cruzada. Observará una nueva sintaxis en la siguiente expresión.

En R, se denomina "**formula**" a este tipo de sintaxis:

V4 ~ V1 + V2 + V3, que significa **V4 en función de V1 y V2 y V3**.

Por defecto, las columnas de una matriz se referencian como **V1, V2...**

```
cubo <- xtabs( V4 ~ V1 + V2 + V3, data = m)
```

Esta instrucción permite transformar **m** en una tabla cruzada donde **V4** se expresa en función de los otros tres ejes. Observe que es más bien un paralelepípedo que un cubo. Esta sintaxis nos permite fabricar estructuras con el número de ejes que deseemos (que llamaremos, sin duda, «hipercubos»).

Exploraremos nuestro cubo.

```
cubo
```

```
, , V3 = 1      [ para V3 = 1 podemos ver la siguiente copa:      ]  
          V2  
V1   1   2      [ una subtabla de V4 en función de V1 y V2 ]  
 1 111 121      [ sabiendo que V3 = 1                      ]  
 2 211 221  
  
, , V3 = 2      [ ídem sabiendo que  V3 = 2                  ]  
          V2  
V1   1   2  
 1 112 122  
 2 212 222  
  
, , V3 = 3      [ ídem con V3 = 3                  ]  
          V2  
V1   1   2  
 1 113 123  
 2 213 223
```

Observando atentamente el resultado, comprobará que posee tres «franjas» de un cubo que observa en la dirección **V3**.

Esto va a resultar más explícito manipulando el «cubo» (es decir, una tabla cruzada). En primer lugar, comprobemos sus dimensiones según sus tres ejes.

```
dim(cubo)
```

```
[1] 2 2 3
```

Tenemos, efectivamente, una estructura capaz de arbitrar los 12 valores de **V4** ($2 * 2 * 3$).

Para sostener nuestra interpretación de las siguientes manipulaciones, vamos ahora a imaginar que el eje **V1** representa las filas del cubo, el eje **V2** las columnas y el eje **V3** una profundidad.

Accedamos a **una** celda del cubo.

```
cubo[1,2,3] # fila 1, columna 2, profundidad 3
```

```
[1] 123
```

Accedamos a una «franja» del cubo fijando un valor para la profundidad. Como los ejes **V1** (filas) y los ejes **V2** (columnas) son de dimensión 2, se espera obtener cuatro valores de **V4**.

```
cubo[,2] # franja en profundidad = 2
```

```
V2  
V1 1 2  
1 112 122  
2 212 222
```

Ahora vamos a «girar» el cubo y construir franjas fijando los valores sobre otro eje, intentemos con **V2** (columnas). Como los demás ejes tienen dimensión 2 y 3, se espera obtener seis valores de **V4**.

```
cubo[,2,] # franja en columna = 2
```

```
V3  
V1 1 2 3  
1 121 122 123  
2 221 222 223
```

Ahora podemos «observar» una «franja» del cubo en el eje **V2**.

Ahora que hemos seleccionado una «arista», es decir, la intersección de dos «franjas», por ejemplo la intersección de la franja tal que **V1** (fila) sea igual a 1 y la franja tal que **V2** (columna) sea igual a 1. La intersección se encuentra a lo largo del eje de profundidad y se espera obtener, de este modo, tres valores de **V4**.

```
cubo[1,1,] # intersección fila=1,columna=1
```

```
1 2 3  
111 112 113
```

Observe que las franjas son **matrix** completas.

```
is.matrix(cubo[1,,]) # coherencia matrix array  
dim(cubo[1,,]) # respeta las dimensiones
```

```
[1] TRUE  
[1] 2 3
```

Ahora vamos a realizar diversos cálculos sobre esta tabla.

Cálculos sobre el cubo

Realicemos un cálculo sencillo, la suma de toda la tabla.

```
sum(cubo[, ,])      # suma todos los valores
```

```
[1] 2004
```

Ahora vamos a abordar por primera vez una **sintaxis fundamental** de R, las funciones de tipo **apply()**.

Existen varias que descubriremos a lo largo de este libro. El uso de estas funciones permite aplicar una función sobre el conjunto de los componentes de una estructura. Permiten, a su vez, aplicar una función de agregación sobre diversas dimensiones de la estructura, por ejemplo efectuar la suma o calcular el máximo o la media de una columna o de una fila.

También podemos aplicar nuestras propias funciones. Aquí, vamos a utilizar **apply()** para agregar franjas del cubo (mediante la función **sum**, aunque habríamos podido escoger **max** o **min**...). Este tipo de cálculo es exactamente el que acostumbramos a hacer cuando utilizamos tablas cruzadas.

```
apply(cubo, 1, sum)  # suma el segundo y tercer eje
```

```
1     2  
702 1302
```

Para comprender bien lo que hemos obtenido, observe los dos cálculos siguientes:

```
sum(cubo[1,,])      # suma de toda la fila = 1  
sum(cubo[2,,])      # suma de toda la fila = 2
```

```
[1] 702  
[1] 1302
```

Se han «comprimido» los ejes **V2** (columna) y **V3** (profundidad) y se obtiene una fila de suma de valores de **V4** para cada valor de **V1** (fila).

Intentémoslo sobre algún otro eje.

```
apply(cubo, 3, sum)  # suma el primer y segundo eje
```

```
1 2 3  
664 668 672
```

Ahora, vamos a realizar el cálculo a lo largo de un solo eje, lo que nos devolverá una consolidación de las «franjas», un poco como si observáramos todas las sumas de los elementos sobre una cara del cubo. Con esta operación encontrará cálculos que le permitirán verificar los resultados de **apply**.

```
apply(cubo,c(1,2),sum) # suma sobre la tercera dimensión  
sum(cubo[1,1,])  
sum(cubo[1,2,])  
sum(cubo[2,1,])  
sum(cubo[2,2,])
```

```
V2  
V1 1 2  
1 336 366  
2 636 666  
  
[1] 336  
[1] 366  
[1] 636  
[1] 666
```

Dispone ahora de una solución que le permite realizar cálculos sobre tablas cruzadas (hipercubos) con tantos ejes como le sea útil (estas técnicas resultan muy eficaces sobre grandes cubos, en el sentido de que los problemas de rendimiento no aparecen más pronto que si se trabajase con matrices sencillas del mismo tamaño). Recuerde que puede utilizar este método con otras funciones, como en el siguiente ejemplo con **max()**.

```
apply(cubo,c(1,2), max) # max sobre la tercera dimensión
```

```
V2  
V1 1 2  
1 113 123  
2 213 223
```

Matrices simples para cálculos de lógica o de probabilidad

Aplicación a la lógica

En lógica, se utilizan tablas de verdad que permiten saber cómo reacciona un conector lógico en función de los valores que va a manipular. En el mundo real del data scientist, ciertos valores no son conocidos. Gracias a una función muy útil para crear matrices, vamos a visualizar la tabla de verdad extendida a los valores desconocidos del «Y» lógico.

```
x <- c(NA, FALSE, TRUE) # nuestro caso de uso  
x
```

```
[1] NA FALSE TRUE
```

```
names(x) <- c("Desconocido", "Falso", "Verdadero") # agrega los nombres  
# de las columnas  
x
```

```
Desconocido    Falso    Verdadero  
NA      FALSE     TRUE
```

```
outer(x, x, "&") # tabla de verdad
```

```
Desconocido    Falso    Verdadero  
Desconocido NA FALSE     NA  
Falso        FALSE FALSE FALSE  
Verdadero    NA FALSE     TRUE
```

Resulta interesante observar este resultado, pues la tabla no es simétrica: cuando uno de los dos valores es desconocido en un «Y», no se conoce su unión con un VERDADERO o un FALSO, y de manera inversa se sabe que su unión con un FALSO devolverá de cualquier manera un FALSO. Reparar en esto le permite evitar diversos desengaños cuando utilice datos reales que pueden incluir valores desconocidos.

► Cuando tenga que fabricar una malla de datos para una representación en 3D o aplicar un operador a todos los cruces para todos los casos entre los valores de dos vectores, piense en utilizar **outer()**.

Aplicación a la probabilidad

La probabilidad está en el corazón de nuestra práctica, de modo que es perfectamente útil realizar algunas simulaciones para medir los valores de una probabilidad. Se contará el número de casos favorables dividido por el número de casos posibles para conocer la probabilidad buscada.

He aquí un microproblema que puede resolverse de forma numérica: se echan al aire cuatro monedas no trucadas. ¿Cuál es la probabilidad de obtener una tirada con dos caras y dos cruces?

Vamos a simular las tiradas gracias a la función **expand()**; tomaremos como convención 1 para cara, 0 para cruz.

```
tabla_v <- expand.grid(0:1,0:1,0:1,0:1)  
tabla_v
```

	Var1	Var2	Var3	Var4
1	0	0	0	0
2	1	0	0	0
3	0	1	0	0
4	1	1	0	0
5	0	0	1	0
6	1	0	1	0
7	0	1	1	0
8	1	1	1	0
9	0	0	0	1
10	1	0	0	1
11	0	1	0	1

```
12    1    1    0    1  
13    0    0    1    1  
14    1    0    1    1  
15    0    1    1    1  
16    1    1    1    1
```

Vemos que existen 16 casos posibles (16 filas, es decir **row** en la tabla).

```
nrow(tabla_v)
```

```
[1] 16
```

Identifiquemos el número de caras por tirada.

```
num<- apply(tabla_v,1,sum)  
num
```

```
[1] 0 1 1 2 1 2 2 3 1 2 2 3 2 3 3 4
```

Veamos los casos favorables (dos caras).

```
(num ==2)
```

```
[1] FALSE FALSE FALSE TRUE FALSE TRUE TRUE FALSE FALSE TRUE  
TRUE FALSE TRUE FALSE FALSE FALSE
```

Contémoslos, sabiendo que **sum()** cuenta los valores **TRUE** como 1 y **FALSE** como 0.

```
sum(num==2)
```

```
[1] 6
```

De modo que la probabilidad buscada es:

```
sum(num==2)/nrow(tabla_v)
```

```
[1] 0.375
```

¿Se lo había imaginado?

De hecho, era más rápido fabricar una pequeña función para calcular todas las probabilidades con un único paso y aplicarla sobre una tabla que identifique los cinco casos posibles (0 caras, 1 cara, 2 caras...).

```
f      <- function(x){sum(num==x) / nrow(tabla_v) }
res    <- sapply(0:4,f)
names(res) <- as.character(0:4)
res
```

```
0      1      2      3      4
0.0625 0.2500 0.3750 0.2500 0.0625
```

Y como somos desconfiados, comprobaremos que la suma de estas probabilidades vale 1.

```
sum(res)
```

```
[1] 1
```

Complementos relativos a las tablas

La sintaxis Arrays

La forma más genérica de crear una tabla cuando no se basa en una matriz preexistente como en nuestro ejemplo anterior es la declaración de un **array**.

```
array(1:8, c(2,4)) # crear una tabla con valores de 1 a 8
```

```
[ ,1] [ ,2] [ ,3] [ ,4]
[1,]     1     3     5     7
[2,]     2     4     6     8
```

Nociones propias de OLAP

En el vocabulario propio de OLAP, las operaciones que hemos realizado más arriba se denominan a menudo:

- Slicing: cuando se selecciona una franja de información.
- Scoping: cuando se extrae un bloque de información cualquiera.
- Drill_up: cuando se realiza una síntesis de información perpendicular respecto a un eje.
- Agregation: cuando se realiza una síntesis tras un scoping.
- Rotate: típicamente cuando se seleccionan dos ejes de entre los demás.

Relación con la noción de tensores

Para aquellos que posean un pasado matemático, observe que las tablas le permiten codificar ciertos **tensores**.

En el reconocimiento de imágenes, se utilizan a menudo tensores en 4D para codificarlas (capas de píxeles de color con intensidades diferentes por capa).

Hemos visto cómo aplicar funciones sobre nuestras tablas; veamos ahora cómo crear nuestras propias funciones, que podremos aplicar sobre distintos tipos de datos y estructuras de datos.

5. Las funciones

R es un lenguaje «funcional», de modo que la noción de función está en el núcleo del lenguaje.

a. Creación y uso de una función simple

Vamos a crear una función simple que aumente un valor concreto en un 10 %.

```
f <- function(x){x*1.1}          # función simple
```

Observe la sintaxis: se asigna el comportamiento de la función a **f**, los argumentos de la función se escriben entre paréntesis, el cuerpo de la función entre llaves y la última expresión es la que devuelve la función.

Probemos la función **f()**.

```
f(100)
```

```
[1] 110
```

También es posible dar nombre a los argumentos de una función.

```
f(x = 10)          # podemos establecer los nombres  
                   # de las variables
```

```
[1] 11
```

La declaración de la función puede incluir valores argumentos con valores por defecto.

```
f <- function(x = 1){x*1.1}      # f con x por defecto  
f()                           # f(1)
```

```
[1] 1.1
```

La función **sapply()** permite aplicar fácilmente su función sobre un **vector** (sabiendo que, en este caso sencillo, **f(v)** habría funcionado perfectamente).

```
v <- c(1,2,3,4,5)
```

```
sapply(v,f)          # aplicar f sobre miembros de v
```

```
[1] 1.1 2.2 3.3 4.4 5.5
```

La función **sapply()** le permite aplicar una función cuya codificación no es lo bastante genérica para tratar toda una estructura de una sola vez; en este caso, **sapply()** le permite aplicar la función unitaria sobre cada miembro.

Naturalmente, es posible crear funciones de varias variables.

```
g <- function(x,y){ a <- 10^x      # función de dos variables
                     b <- 10^y
                     (b-a)/(b+a)}
g(1,1)
g(1,1.5)
g(1.5,1.5)
```

```
[1] 0
[1] 0.5194939
[1] 0
```

b. Creación de un operador a partir de una función de dos variables

Vamos a crear un operador a partir de la función **g** de la sección anterior.

Aquí, entendemos por operador la definición de una operación al mismo nivel que la multiplicación, la suma...

```
`%g_op%`     <- g          # un operador
1 %g_op% 1        # ídem g(1,1)
1 %g_op% 1.5       # ídem g(1,1.5)
```

```
[1] 0
[1] 0.5194939
```

Los operadores están delimitados por símbolos %. Nuestro operador **%g_op%** puede utilizarse en un contexto vectorizado.

```
v <- c(1, 1 ,1 ,1.5, 1.5, 1.5)
w <- c(1, 1.5,1 ,1.5, 1,    1.5)
v %g_op% w          # operador entre vectores
```

```
[1] 0.0000000 0.5194939 0.0000000 0.0000000 -0.5194939 0.0000000
```

c. Uso de las funciones y alcance de las variables

Vamos a ver con más detalle lo que se transforma o no con una función.

Veamos en primer lugar una función minimalista, sin argumentos de entrada y sin ningún cálculo.

```
f <- function(){  
  100  
}  
f()                      # devuelve el último valor
```

```
[1] 100
```

Creemos una función con un argumento que posea un valor por defecto y veamos el comportamiento de las variables.

```
f <- function(x = 100){  
  y <- x + 1  
  y  
}  
f()                      # utiliza el valor por defecto  
f(200)                   # suma 1
```

```
[1] 101  
[1] 201
```

```
y <- 300      #  
f(y)         # suma 1  
y            # queda intacto
```

```
[1] 301  
[1] 300
```

Este comportamiento es muy sano y tranquilizador; la variable **y**, global al programa, no se ha confundido con la variable **y** propia de la función. Salvo si se utiliza una sintaxis particular, la asignación de un valor a una variable dentro de una función tiene un **alcance restringido a la función**.

Existe una sintaxis que permite crear y asignar un valor a una variable dentro de una función y hacerla global (es decir, con un alcance exterior a la función; esto debería utilizarse solamente en casos muy concretos, como por ejemplo en una función cuyo único objetivo sea la inicialización de diversas variables del programa).

```
f <- function(x = 100){ # ;no devuelve ningún valor!  
  x <<- x + 1          # observe <<-  
}  
f()                      # ¿no hace nada?  
f(200)                   # ¿no hace nada?
```

De hecho, hemos creado la variable global **x** y le hemos asignado un valor cuyo alcance es general a todo el programa y no solamente a la función **f()**.

```
x # ahora existe una variable global x  
# y vale 301
```

```
[1] 301
```

El siguiente código le muestra lo que ocurre si la variable ya existe.

```
f <- function(x = 100){  
  y <- x - 1      # observe <-  
  z <- x + 1  
  z          # z es local  
}  
  
y <- 300  
z <- 400  
f(z)      # suma 1  
y          # vale 399: ¡atención!  
z          # z queda intacta
```

```
[1] 401  f(z)  
[1] 399  y  
[1] 400  z
```

En una función, esté siempre atento a asignar un valor a todas las variables utilizadas en los cálculos realizados por su función. Es así como podrá exportarla a otro ámbito y evitará efectos indeseados.

Las dos maneras habituales de asignar un valor a una variable **x** son las siguientes:

- **x** posee un valor por defecto y es un argumento de la función. Por ejemplo: **f(x=1){...}**
- y/o se encuentra **x** a la izquierda de una asignación en su código. Por ejemplo: **x <- 1+1**

Como conclusión parcial, quedese con la idea de que, para cambiar el valor de una variable **z** mediante una función **f()**, de manera eficaz y sin efectos indeseados, salvo si lo hace con alguna intención concreta, es eficaz utilizar la siguiente sintaxis:

```
z <- f(z)      # para aplicar sobre z  
z              # z vale 401
```

Esto no restringe el número de valores que puede devolver **f()**. No olvide que puede aplicar la función sobre una estructura rica (**array**, **vector**, **list**...) y que existe la posibilidad de devolver otra estructura rica y, por lo tanto, un gran número de valores aplicando una función.

```
z <- c(10,100,1000)  
z <- f(z)      # para aplicar sobre z  
z
```

```
[1] 11 101 1001
```

Esto funciona sobre una tabla.

```
cubo <- f(cubo)      # en nuestro cubo  
cubo
```

```
, , V3 = 1  
  
V2  
V1   1   2  
 1 112 122  
 2 212 222  
  
, , V3 = 2  
  
V2  
V1   1   2  
 1 113 123  .../... y así sucesivamente ...
```

► De hecho, para realizar una operación matemática sencilla en R sobre una estructura básica como una tabla o un vector, no siempre resulta útil definir una función. ¡Una sintaxis del tipo **cubo <- 2 * cubo + 1** funciona a la perfección!

Desafortunadamente, no todas las operaciones son operaciones aritméticas sencillas. En los casos más difíciles, hay que utilizar funciones como **apply()**. Para demostrarlo, vamos a realizar a continuación un intento fallido de aplicar directamente una función sobre nuestro cubo original, y a continuación un intento eficaz con **apply()**. Tome el tiempo necesario para comprender la pequeña función que comprueba si un número entero es impar que proponemos al principio del código; será un buen ejercicio.

En primer lugar, reinicializamos el cubo y creamos la función que comprueba si un valor es impar o no.

```
cubo <- xtabs( V4 ~ V1 + V2 + V3, data = m)  
isImpar <- function(n){as.logical(n%%2)}
```

Intentemos aplicarla directamente sobre el cubo.

```
isImpar(cubo)      # no se obtiene el resultado esperado!!!
```

```
[1] TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE  TRUE  TRUE  
TRUE  TRUE
```

Esto no funciona del todo; hemos perdido completamente la noción de tabla, debido al hecho de que el contenido de nuestra función no es únicamente aritmético.

Probemos con la función **apply()** sobre todos los valores, es decir, sin dar preferencia a ningún eje, de ahí el valor de la configuración **c(1,2,3)**.

```
cubo_imp <- apply(cubo,c(1,2,3),isImpar)
cubo_imp
```

```
, , V3 = 1      [ para V3 = 1, por construcción del valor      ]
                  [ de V4 termina con un 1, de modo que es impar      ]
V2
V1      1      2
1 TRUE TRUE
2 TRUE TRUE

, , V3 = 2      [ a la inversa, para V3 = 2 es par      ]

V2
V1      1      2
1 FALSE FALSE
2 FALSE FALSE      [ .../... y así sucesivamente ...      ]
```

He aquí cómo hemos conseguido aplicar nuestra función sobre el cubo.

d. Aplicación de las funciones sobre las matrices: **apply**

Para aplicar funciones sobre las tablas de dos dimensiones (matrices), vamos a dar preferencia al uso de funciones de tipo **apply()**.

Para realizar nuestras pruebas, fabriquemos una matriz **m** con los 12 valores enteros distribuidos en tres filas.

```
m <- matrix( 1:12, nrow = 3)
m
```

```
[,1] [,2] [,3] [,4]
[1,]    1     4     7    10
[2,]    2     5     8    11
[3,]    3     6     9    12
```

Apliquémosle una función exponencial sobre cada término.

```
# aplicación de una función sobre cada término
h <- apply(m,c(1,2),exp) # aplicación exponencial por todo
print(h)                  # c(1,2) significa sobre fila y columna
```

```
[,1]      [,2]      [,3]      [,4]
[1,] 2.718282 54.59815 1096.633 22026.47
[2,] 7.389056 148.41316 2980.958 59874.14
[3,] 20.085537 403.42879 8103.084 162754.79
```

Creemos un vector columna con las sumas de cada fila; observe el hecho de que aquí se reutiliza la instrucción **matrix()**.

```
# suma de cada fila
```

```
h <- matrix(apply(m,1,sum)) # vector columna, suma por fila  
print(h)                      # "1" significa "por fila"
```

```
[ ,1]  
[1,] 22  
[2,] 26  
[3,] 30
```

Además, creamos un vector fila (traspuesto de una columna) que será la suma de cada columna; observe el uso de **aperm()** para realizar la trasposición.

```

# suma de cada columna
h <- aperm(matrix(apply(m,2,sum)), # vector fila
            c(2, 1))                  # se trasponen las dimensiones 2 y 1
                                    # con la suma por columna
print(h)                      # "2" significa "por fila"

```

```
[,1] [,2] [,3] [,4]
[1,]    6   15   24   33
```

Podemos aplicar una función anónima, aquí una comprobación mediante la **muy útil función `ifelse()`**.

```
# aplicación de una función lambda sobre cada término
# por ejemplo, transforma un 1 en los lugares donde el valor sea >= 3,
0 en caso contrario
h <- apply(m, c(1,2), function(x) {ifelse(x >= 3, 1, 0)})
print(h)
```

Le dejamos que verifique el resultado.

R es un lenguaje que implementa numerosas funciones; a continuación vamos a abordar algunos de los aspectos menos triviales de las funciones de R.

e. Las funciones: completamente útiles

Si tiene experiencia en otros lenguajes, puede que le sorprenda el hecho de que R implemente mecanismos eficaces que le permiten manipular sus propias funciones, como en los lenguajes modernos. Puede que tenga el deseo de trabajar sobre su propia implementación de «closures» en R.

Acceso explícito a una variable global

Hemos visto cómo resultaba peligroso acceder a una variable global en una función. Si debe hacerlo, conviene hacerlo de una manera muy explícita, que le incite a gestionarla. Esto se realiza mediante la función **get()**, cuyo objetivo es buscar una variable precisa, si es necesario en un entorno concreto.

```
# acceso explícito a una variable global  
rm(x) # destrucción de x
```

```
f <- function() {get( "x" ) # acceso explícito  
                 x+1  
               }  
  
f() # error
```

Error in get("x") : object 'x' not found

Cuando **x** no existe, su programa falla, con el correspondiente error.

Si **x** existe en el entorno considerado, todo funciona correctamente.

```
x <- 10  
f() # funciona
```

[1] 11

Funciones anónimas

Como en otros lenguajes, es posible utilizar funciones creadas al vuelo, sin nombrarlas, con una sintaxis muy compacta. Observe los parámetros utilizados en la definición de la función.

```
(function(x) 2*x+1)(100)
```

[1] 201

Es posible que estas funciones anónimas sean funciones de varias variables, y resulta sencillo fabricar una estructura más compleja como valor de retorno (aquí un **vector**).

```
(function(x,y) {c(x+y,x*y,x-y,x/y)})(1,2)
```

[1] 3.0 2.0 -1.0 0.5

Es posible invocar directamente estas funciones dentro de funciones como **apply()**. Ya hemos visto más arriba un ejemplo similar sobre una matriz. Aquí determinaremos si los miembros de un vector son impares o no.

```
v <- c(10,21,30)  
sapply(v,(function(x){as.logical(x%%2)}))
```

[1] FALSE TRUE FALSE

Sin embargo, un **uso que necesariamente debe estudiar y recordar** es el caso en que necesita aplicar una función de varias variables mediante una función como **apply()**.

```
f <- function(x,y){x+y}
z <- 100
sapply(v,function(x) f(x,z))) # sapply función de dos variables
```

```
[1] 110 121 130
```

En efecto, la siguiente sintaxis, que habría podido parecerle natural, va a fallar.

```
sapply(v,f(x,z)) # ¡pues no es posible!
```

```
Error in match.fun(FUN) :
  'f(x, z)' no es una función, una cadena de caracteres o
  un símbolo
```

De hecho, el parámetro **FUN**, a saber, la «*FUNCTION*» que desea aplicar, debe ser una función de una sola variable y efectivamente se trata de una función anónima de una sola variable que habíamos fabricado en el ejemplo anterior. Esto, fijando una de las dos variables de **f(,)** y creando una función anónima de una variable.

► ¿Pero por qué no habríamos podido construir una función no anónima que habría podido utilizarse a continuación en el **sapply()**?

Esto es posible, pero presenta el siguiente inconveniente: imagine que ha realizado este **sapply()** en un **bucle con un valor de z que cambia**, habría tenido que recrear la función anónima una gran cantidad de veces y destruirla las mismas veces (utilizando **rm(mi_funcion)**). O incluso habría tenido que encontrar una solución más larga (por ejemplo, creando otra función para encapsular estas operaciones).

Otro buen motivo para utilizar una función anónima es el hecho de referenciar una estructura de datos sin paso de parámetro, pero también sin efectos indeseados. Hoy un becario del laboratorio plantea la siguiente pregunta: *Tengo un vector x: c(10,25,4,10,9,9). Me gustaría calcular las repeticiones dentro de x y obtener un vector de la misma longitud y = c(2,1,1,2,2,2). ¿Cómo podría hacer esto con R?*

He aquí una posible respuesta con una función anónima.

```
x <- c(10,25,4,10,9,9)
y <- sapply(x,function(z){sum((x==z))})
y
```

```
[1] 2 1 1 2 2 2
```

La función anónima se aplica mediante un verdadero argumento (**z**), pero invoca discretamente a otro parámetro (**x**) que es, a su vez, sobre el que se aplica la función; se gana cierto número de bucles. Además, esto tiene sentido en términos funcionales de lectura del código:

- Detectar los elementos iguales, mediante (**x == z**).
- Contarlos, mediante **sum()**.
- Aplicar sobre cada elemento de **x**, mediante **sapply()**.

Acceso al núcleo de las funciones

Diversas funciones de introspección permiten visualizar la intimidad de las funciones (salvo en ciertas funciones particulares).

```
f <- function(x=0) {x^2}
body(f)
```

```
{
  x^2
}
```

La función **body()** nos permite ver el cuerpo de una función. Pero también transformarlo (esto sin duda no es una buena práctica, salvo en el marco de la realización provisional de transformaciones de funciones **para llevar a cabo ciertas pruebas sin transformar el resto del código**).

```
f(2)
body(f) <- quote(x^3)
body(f)
f(2)
```

```
[1] 4
x^3
[1] 8
```

Es bastante sorprendente.

Esto permite comprender una función contenida en un paquete, incluso aunque no esté documentada. Puede utilizar la instrucción **args()** para conocer los argumentos de la función y consultar el cuerpo de la función con **body()**. Veamos lo que se obtiene con **sapply()**.

```
# estudiar una función
args(sapply) # argumentos
body(sapply) # cuerpo
```

```
function (X, FUN, ..., simplify = TRUE, USE.NAMES = TRUE)
NULL

{
  FUN <- match.fun(FUN)
  answer <- lapply(X = X, FUN = FUN, ...)
  if (USE.NAMES && is.character(X) && is.null(names(answer)))
    names(answer) <- X
```

```

if (!identical(simplify, FALSE) && length(answer))
  simplify2array(answer, higher = (simplify == "array"))
else answer
}

```

Es posible aprender bastante estudiando el código de nuestros colegas; aquí podemos destacar que la función **sapply()** es una implementación adaptada de **lapply()**.

Funciones que devuelven funciones

A continuación vamos a fabricar una función polimorfa. Observe los dos puntos siguientes: se asigna una función en una lista que contiene tres funciones; la función **f** devuelve un valor de la lista y, por lo tanto, una función.

```

f <- function(n){
  g <- function(z) {summary(z)}
  h <- function(z) {sd(z)}
  k <- function(z) {range(z)}
  l <- list(g,h,k)
  l[[((n%%3)+1)]]
}

```

```

v <- c(1,2,3,4)
f(1)(v)      # resumen
f(2)(v)      # desviación típica
f(3)(v)      # min y max

```

```

Min. 1st Qu. Median Mean 3rd Qu. Max. [ summary ]
1.00    1.75    2.50   2.50   3.25   4.00

[1] 1.290994 [ desviación típica ]
[1] 1 4        [ min y max ]

```

Observe la sintaxis **f(1)(v)**: se aplicará la función determinada por el valor **1**.

Pasar funciones como argumento

Es posible pasar funciones como argumento a otras funciones.

Como pequeño ejercicio de estilo para apelar a nuestra creatividad, vamos a crear una función «dual» de un **vector v**, que vamos a llamar **v_**, y cuyo resultado será la aplicación de una función cualquiera sobre **v**.

```

v_ <- function(f) {f(v)} #dualizar
v_(function(x) {2*x})
v_(mean)

```

```

[1] 2 4 6 8
[1] 2.5

```

La función **v_** recibe como argumento otras funciones, anónimas o no, aplicándolas sobre **v**.

Este último ejemplo no le ha convencido, necesariamente, de la utilidad de pasar una función como argumento a otra función, pero el siguiente sin duda va a despertar su interés.

```
segundo_grado <- function(f,a,b,c,x){a*f(x)^2+b*f(x)+c}
segundo_grado(cos,2,0,-1,v)           # 2 cos(x)**2 - 1
```

```
[1] -0.4161468 -0.6536436  0.9601703 -0.1455000
```

Hemos compuesto un polinomio de segundo grado de la función seno en una única instrucción. Esto puede aplicarse si se desea sobre otras funciones con otros coeficientes.

La base de la programación clásica son los bucles y las estructuras de control. La programación vectorial nos permite liberarnos en gran medida de los bucles en R, la mayoría de las veces de una manera óptima. Por este motivo hemos aplazado su descripción en esta toma de contacto con R para que su primer reflejo sea siempre utilizar los mecanismos vectoriales de R. Pero, evidentemente, los necesitaremos en muchos casos. Veámoslo.

6. Estructuras de control

a. Instrucciones comunes con otros lenguajes

La sintaxis de una prueba es muy simple. Si ya ha programado en el pasado, no se sentirá desorientado; en caso contrario, estudie esto muy atentamente.

```
x <- 20                                # variable comprobada
if (x < 10) {print("x < 10")}   else
  {print ("x >= 10")}
```

```
[1] "x >= 10"
```

La condición está **entre paréntesis** y los dos bloques que se han de ejecutar están **entre llaves**. El segundo bloque con el **else** que precede es opcional. El **else** sigue directamente a la llave que cierra el primer bloque ejecutable.

La sintaxis de un bucle **for** es también clásica, observe la forma de recorrer un **vector** con la sintaxis **in**.

```
for (x in c(1,10,20)) {print(2*x)}    # bucle for sobre
                                         # una secuencia
```

```
[1] 2
[1] 20
[1] 40
```

Podemos fabricar las secuencias que queramos con la instrucción **seq()**.

```
s <- seq(from= 1,to= 20,by = 4)      # fabrica una secuencia  
s
```

```
[1] 1 5 9 13 17
```

```
for (x in s) {print (x)}           # bucle sobre una secuencia
```

```
[1] 1  
[1] 5  
[1] 9  
[1] 13  
[1] 17
```

También disponemos de la instrucción **while** (*hacer mientras que*). El siguiente ejemplo vacía una lista elemento a elemento. Este ejemplo le servirá sin duda en el futuro, estúdielo atentamente.

```
l <- list("tres","dos","uno")      # lista  
while (length(l) > 0) {           # mientras que hacer:  
  print(l[[length(l)]])  
  l[[length(l)]] <- NULL          # vacía una posición  
}                                  # fin del bucle  
length(l)
```

```
[1] "uno"  
[1] "dos"  
[1] "tres"  
  
[1] 0
```

La instrucción **repeat** (*hacer hasta que*) ejecuta al menos una vez el código que se encuentra dentro de las llaves. Por último, el bucle comprueba la condición para volver a repetirse o no. Fabriquemos una lista utilizando esta instrucción.

```
l <- list()  
repeat {                           # los 999 primeros cuadrados  
  i <- max(length(l),0)  
  l[[length(l)+1]] <- i^2  
  if (length(l) >= 1000) {break}    # repeat until break  
}  
head(l)                            # los 6 primeros  
tail(l)                            # los 6 últimos
```

```
[[1]]  
[1] 0  
  
[[2]]  
[1] 1
```

```
.../...
```

```
[[5]]  
[1] 996004
```

```
[[6]]  
[1] 998001
```

En realidad, resulta más eficaz trabajar de la siguiente manera, sin bucle:

```
l <- as.list(seq(from = 1, to = 999)^2)  
tail(l)
```

```
.../...  
[[5]]  
[1] 996004
```

```
[[6]]  
[1] 998001
```

b. Recorrer una matriz mediante bucles for

Ya que abordamos los bucles, veamos cómo podríamos recorrer una matriz utilizando un bucle. Un apasionado de R lo evitaría, pero no ser capaz le pondría en aprietos.

El siguiente ejemplo busca los índices de una matriz **m** que se corresponde con un **valor** concreto, por defecto **TRUE**. Esta función permite seleccionar todas las «celdas» de la matriz que se corresponden con una condición y devuelve la lista. Si no está entrenado en programación clásica, un profundo estudio de las siguientes líneas de código le resultará muy útil, encontrará en ellas aspectos básicos de la programación tal y como se practica desde hace más de medio siglo...

```
check_matriz <- function(mat,valor = TRUE){  
  k <- 0  
  x <- vector()  
  y <- vector()  
  for (i in 1:nrow(mat)){  
    for (j in 1:ncol(mat)){  
      if (mat[i,j] == valor){  
        k <- k+1  
        x[k] <- i  
        y[k] <- j  
      }  
    }  
  }  
  xy <- matrix(rbind(x,y),nrow = 2)  
}
```

Habrá observado:

- El **valor** por defecto a **TRUE**.
- La inicialización de **k** a **0**, **k** es el contador de celdas que «concuerdan».

- La inicialización de los vectores **x** e **y**, indispensable para que todo funcione bien en la primera iteración; en caso contrario la asignación de **x** o de **y** no funcionaría.
- El uso de funciones que devuelven el número de filas y de columnas de mat.
- El incremento de **k** con la instrucción **k<-k+1**, típica de la programación clásica.
- La anidación de bucles y el posicionamiento de las llaves (es posible realizar otro sangrado o alinearlos de manera distinta, practicaremos un sangrado propio de la práctica habitual de Python que no todo el mundo comparte, aunque lo importante es disponer de unas reglas propias y dar preferencia a la legibilidad).
- El uso de **rbind()** para asociar las dos filas.
- ¡La ausencia de comentarios! Esto no es bueno, aunque como los comentarios están aquí, en el propio texto del libro, no queríamos duplicarlos.

Utilicemos una matriz con los 12 primeros valores enteros distribuidos en 4 columnas para realizar nuestras pruebas.

```
m <- matrix(1:12,nrow = 3)
m
```

```
[,1] [,2] [,3] [,4]
[1,]    1     4     7    10
[2,]    2     5     8    11
[3,]    3     6     9    12
```

Para comprender bien la prueba de la función, observe en primer lugar lo que produce la siguiente instrucción:

```
(m > 5)
```

```
[,1] [,2] [,3] [,4]
[1,] FALSE FALSE TRUE TRUE
[2,] FALSE FALSE TRUE TRUE
[3,] FALSE  TRUE TRUE TRUE
```

Hemos obtenido una matriz para la que los valores a verdadero se corresponden con las celdas de **m** estrictamente superiores a 5. Utilicemos esta sintaxis en nuestra función.

```
c <- check_matriz((m>5))
c
```

```
[,1] [,2] [,3] [,4] [,5] [,6] [,7]
[1,]    1     1     2     2     3     3     3
[2,]    3     4     3     4     2     3     4
```

La matriz obtenida devuelve siete celdas de **m** cuyo valor es estrictamente mayor que 5. Por ejemplo, sabe que **m[1,3]** y **m[3,4]** son, respectivamente, la primera y la última celda que satisfacen la condición. Imagine que la matriz **m** tuviera 10 000 filas; comprende ahora el interés de dicha función, que le permite extraer la información deseada.

Un bonito y compacto bucle **for** le permite encontrar los valores de origen en la matriz **m**.

- Observe que la instrucción **cat()** no aporta nada a la demostración, no es más que la instrucción que le permite mostrar sus resultados por fila utilizando el carácter de escape **\n**.

```
for ( k in 1:ncol(c)) {  
  cat(k,"° valor > 5 : ",m[c[1,k],c[2,k]],"\n")  
}
```

```
1 ° valor > 5 : 7  
2 ° valor > 5 : 10  
3 ° valor > 5 : 8  
4 ° valor > 5 : 11  
5 ° valor > 5 : 6  
6 ° valor > 5 : 9  
7 ° valor > 5 : 12
```

Pero repasemos lo que hemos visto acerca de las funciones. La siguiente sintaxis permite a su vez acceder a los valores de **m**, aquí para **k=3**.

```
(function(k)(m[c[1,k],c[2,k]]))(3)
```

```
[1] 8
```

Apliquemos esta sintaxis de función anónima sobre el conjunto de índices de **c**, para obtener nuestros valores de **m**.

```
sapply(1:ncol(c),(function(k)(m[c[1,k],c[2,k]])))
```

```
[1] 7 10 8 11 6 9 12
```

Esto funciona bien. Conviene recordar el método: para extraer distintos valores de una estructura de datos, en ocasiones resulta práctico aplicar una función anónima con este fin, que evite un bucle **for**.

Los bucles y las estructuras de control son herramientas muy rústicas y sólidas, aunque el código producido resulta, en ocasiones, más difícil de depurar. Los usuarios de R las evitan en beneficio de los métodos vectoriales y del uso de funciones. Pero, al final, lo que debe recordar es, sin duda, que la legibilidad de su código debe primar (para permitir realizar una relectura ágil de su código no solo por parte de otros, sino también por usted mismo pasado un tiempo).

Hasta ahora hemos manipulado números o valores lógicos organizados en diversas estructuras de datos. Las siguientes funciones ilustran la manera de tratar cadenas de caracteres.

7. Las cadenas de caracteres

Es posible juntar cadenas de caracteres entre sí, con o sin un carácter de separación.

```
paste("es","un","bonito")          # pega con espacio  
paste0("g","a","t","o")           # pega sin espacio
```

```
[1] "es un bonito"  
[1] "gato"
```

Las cadenas de caracteres pueden almacenarse en estructuras de datos que conocemos, en este caso una lista.

```
r <- list("un","pequeño","gato")  # una lista de palabras  
str(r)  
paste(r,collapse = "-")          # pega los elementos de la lista con un guion
```

```
[1] "un-pequeño-gato"
```

```
r <- c("p","e","r","r","o")  
str(r)  
paste(r,collapse = "")           # pega los elementos del vector sin espacio
```

```
[1] "perro"
```

Es posible extraer una cadena a partir de otras cadenas.

```
substr("un pequeño gato",4,10)    # extracción de una subcadena
```

La división de cadenas también resulta sencilla, se obtiene una lista de **vectores** que contienen las cadenas de caracteres. La división de cadenas de caracteres resulta muy útil, entre otros para la extracción y conteo de información que se convertirá en sus futuras features.

```
strsplit("un pequeño gato"," ")    # divide la cadena según  
                                  # los espacios
```

```
[[1]]  
[1] "un"      "pequeño" "gato"
```

```
strsplit("un pequeño gato"," ")[[1]][2]
```

```
[1] "pequeño"
```

```
strsplit("un pequeño gato","t") # divide la cadena según las "e"
```

```
[[1]]  
[1] "un p" "qu"      "ño gato"
```

```
t <- c("un pequeño gato","un pequeño perro")  
strsplit(t, " ")
```

```
[[1]]  
[1] "un"      "pequeño" "gato"
```

```
[[2]]  
[1] "un"      "pequeño" "perro"
```

Para limpiar las cadenas de caracteres antes de extraer la información correspondiente, a menudo resulta útil sustituir una cadena de caracteres por otra.

```
sub("pequeño", "gran", "un pequeño gato") # sustitución de cadena
```

```
[1] "un gran gato"
```

La siguiente instrucción resulta muy útil, pues le permite realizar una sustitución con la **garantía de no cambiar la longitud de la cadena de caracteres**.

```
chartr("H.", "Sr...", "H. Pérez") #sustitución del mismo número de caracteres  
chartr("H.", " ", "H. Pérez")
```

```
[1] "Sr Pérez"  
[1] " Pérez"
```

Podemos identificar la presencia de una palabra en una cadena para obtener los índices de los elementos de un **vector** que incluyen esta palabra. Recuperaremos nuestro vector **vector t** de dos cadenas utilizado más arriba.

```
grep("pequeño", t)  
grep("gato" , t)
```

```
[1] 1 2  [ "pequeño" aparece en t[1] y t[2]      ]  
[1] 1     [ "gato"   solo aparece en t[1]       ]
```

Cuando analice una columna de texto sobre una serie de observaciones, **obtiene una nueva feature** cualificando las observaciones a partir de si contienen o no la expresión buscada.

Para poder reconocer que las palabras son las mismas, incluso aunque las mayúsculas y minúsculas sean diferentes (como, por ejemplo, en el inicio de una frase), resulta práctico en ocasiones transformar una cadena en minúsculas o mayúsculas.

```
tolower("UN qran DÍA")      # poner en minúsculas
```

```
toupper("UN gran DÍA")      # poner en mayúsculas
```

```
[1] "un gran día"  
[1] "UN GRAN DÍA"
```

La implementación de estas instrucciones no es trivial. Soportan los caracteres acentuados, no deforman los caracteres de escape como `\n` para el salto de línea, o como `\"` para poder introducir unas comillas dentro de una cadena de caracteres que está definida entre comillas.

```
toupper("& $ € éçèêùµ+} \172 \n \" ")
```

```
[1] "& $ € ÉÇÈÊÙµ+} Z \n \" "
```

Observe, por ejemplo, que la cadena `\172` se corresponde con la `z` minúscula; codificada en octal, se ha interpretado correctamente y se ha convertido en una `Z` mayúscula.

Si quiere convencerse de que `\172` vale `z`, intente escribir `print("\172")` o eche un vistazo al sitio web:
<http://www.asciitable.com/>

Dispone a su vez de toda la potencia de las **expresiones regulares**, que no detallaremos aquí, pero que debe dominar y que permiten realizar muchas conversiones muy complejas entre textos.

Un ejemplo sencillo sería la sustitución de caracteres variados:

```
gsub("[a/e/o]","_","un pequeño gato") # sustitución mediante una expresión  
                                # regular
```

```
[1] "un p_quñ_ g_t_"
```

He aquí otro ejemplo: la idea sería eliminar las URL de los textos. Cada fase de la función está detallada, su estudio le permitirá comprender bien diversos mecanismos que hemos visto más arriba. Observe atentamente la evolución del contenido de las variables en cada fase.

```
# eliminación de las URL          #  
  
f_sup_url <- function(s){        # eliminación de las URL  
  t <- unlist(strsplit(s," "))    # división en palabras  
  l <- length(t)                # l es el número de palabras  
                                # i = índice de palabras que son URL  
  i <- grep("(http://|https://|www://|ftp://)",t) # regex  
  
  if (length(i)>0){            # si existe una URL  
    j <- setdiff(1:l,i)         # todos los índices salvo los i  
    s <- t[(j)]                 # conserva lo que no está en una URL  
    s <- paste(s,collapse = " ") # se recomponen las cadenas  
  }  
}
```

```

s
}

#-----#
# eliminación de las URL - prueba          #

t1 <- "bla bla http://t.co/Orl2YyU7xv ftp://x.us fin"
t2 <- "bla bla https://www.lugar.es http://dominio/eee.com?x=27"
t3 <- "https://t.co/vlcAvs2"
t4 <- "no hay URL"

print(f_sup_url(t1))
print(f_sup_url(t2))
print(f_sup_url(t3))
print(f_sup_url(t4))

```

```

[1] "bla bla fin"
[1] "bla bla"
[1] ""
[1] "no hay URL"

```

8. El formato de los números

Para editar números, en muchas tablas por ejemplo, conviene dominar el formato de los números. He aquí las principales funciones que permiten dar formato a los números.

```

w <- v*pi                         # ;para tener comas decimales!
w
w1 <- format(w)                   # transforma en caracteres
w1

```

```

[1] 3.141593   6.283185   9.424778   12.566371
[1] "3.141593" "6.283185" "9.424778" "12.566371"

```

```

w2 <- format(w, digits = 2)        # ídem con dos dígitos (.x)
w2

```

```

[1] " 3.1" " 6.3" " 9.4" "12.6"

```

```

as.numeric(w1)-as.numeric(w2)      # no son los mismos valores

```

```

[1] 0.041593 -0.016815  0.024778 -0.033629

```

```

format(w, nsmall = 10)             # 10 cifras tras la coma
format(w, digits = 1)              # round (no es trunc)

```

```
[1] " 3.1415926536"    " 6.2831853072"    " 9.4247779608"
```

```
"12.5663706144"
```

```
[1] " 3"    " 6"    " 9"    "13"
```

Los datos pueden representar a su vez tiempos y fechas. Su formato es bastante rico, y existen numerosas opciones para manipularlos.

9. Fechas y tiempos

El primer problema al que nos enfrentamos consiste en recuperar los valores manipulables de fechas con formato que se encuentran en los archivos de datos.

```
d <- as.Date("01/01-2016", "%d/%m-%y") # decodifica una fecha  
d  
str(d)                                # es efectivamente una fecha
```

```
[1] "2020-01-01"  
Date[1:1], format: "2020-01-01"
```

Se ha obtenido un dato en formato fecha. Sobre estas fechas es posible agregar o sustraer duraciones o calcular desfases.

```
d+366                                # suma 366 días
```

```
[1] "2021-01-01"
```

Intente ejecutar los siguientes comandos:

```
Sys.Date()                      # fecha en curso de nuestra máquina  
  
format(Sys.Date(), "%a %d %b")  # formato de la fecha  
  
? strptime                      # veamos las opciones  
  
Sys.time()                       # hora de sistema de la máquina
```

10. Medir la duración de un algoritmo

Ahora que sabemos manipular fechas, podemos responder a una pregunta que se planea a menudo: ¿es cierto que los bucles consumen en general más tiempo que hacer uso de las capacidades vectoriales de R?

Construiremos una función que podremos aplicar en muchas ocasiones. Inicialicemos los **vectores** que vamos a manipular.

```

f <- function(x){x^3+x^2+1/(x+10)} # una función

x_    <- vector()                      # inic. eje de las x
big   <- vector()                      # inic. gran vector a tratar
y1_   <- vector()                      # inic. eje y para la curva 1
y2_   <- vector()                      # inic. eje y para la curva 2

```

Vamos a realizar diez veces dos pruebas, sobre números de filas cada vez más grandes, hasta alcanzar los 10 millones de filas. Cada prueba consistirá en aplicar la función **f** una vez mediante la función vectorial **sapply()**, y una vez mediante un bucle **for**.

Antes de cada prueba reiniciaremos el generador de números aleatorios, reiniciaremos el gran **vector** llamado **big**, anotaremos el tiempo de inicio, aplicaremos una gran cantidad de veces la función **f** sobre el vector **big** y almacenaremos el tiempo empleado para aplicar la función.

```

# Bando de pruebas comparativo de sapply y el bucle for          #

f <- function(x){x^3+x^2+1/(x+10)} # una función

x_    <- vector()                      # inic. eje de las x
big   <- vector()                      # inic. gran vector a tratar
y1_   <- vector()                      # inic. eje y para la curva 1
y2_   <- vector()                      # inic. eje y para la curva 2

for (j in 1:10) {
  n <- 50000 * (j^2 * round(log(j+1))+1) # distancia creciente en n
  x_[j] <- n                                # esta iteración sobre j
                                              # se hace sobre n filas

  set.seed(1789)                            # inic. generador aleatorio
  big   <- runif(n)                         # big: n valores aleatorios
  t_    <- Sys.time()                        # t al inicio

  big   <- sapply(big,f)                   # aplicación de f sobre big

  d_    <- Sys.time() - t_                  # t fin
  y1_[j] <- d_                             # recuerda d_final para la iteración j

  set.seed(1789)
  big   <- runif(n)
  t_    <- Sys.time()

  for (i in 1:n ) {big[i] <- f(big[i])}

  d_    <- Sys.time() - t_
  y2_[j] <- d_
}

#-----#
# almacena el resultado

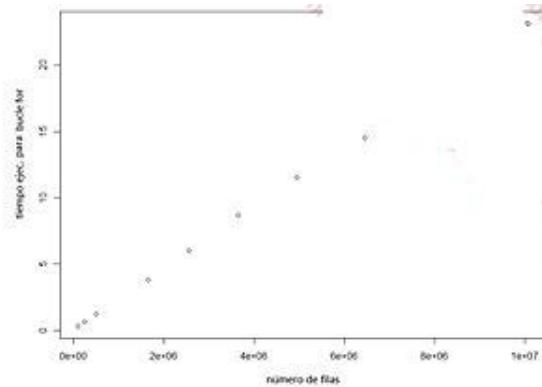
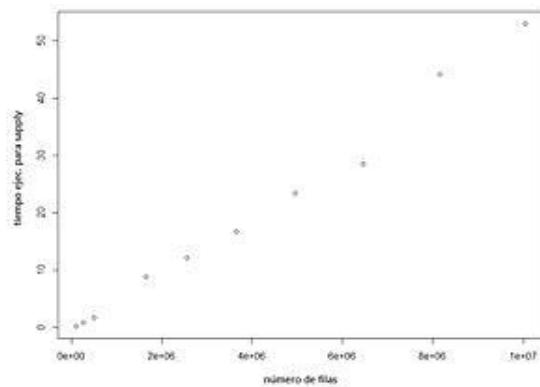
bench_sapply <- cbind(x_,y1_)      # j filas n, delta tiempo

```

```
| bench_for      <-  cbind(x_,y2_)      # j filas n, delta tiempo
```

Este programa va a tardar varios minutos en ejecutarse, tratamos de realizar un verdadero «banco de pruebas» y en cada una de las 10 iteraciones se aumenta drásticamente el número de filas de la prueba.

Intentemos construir un gráfico con el resultado.



Difícil comparación entre los dos gráficos, apply y for

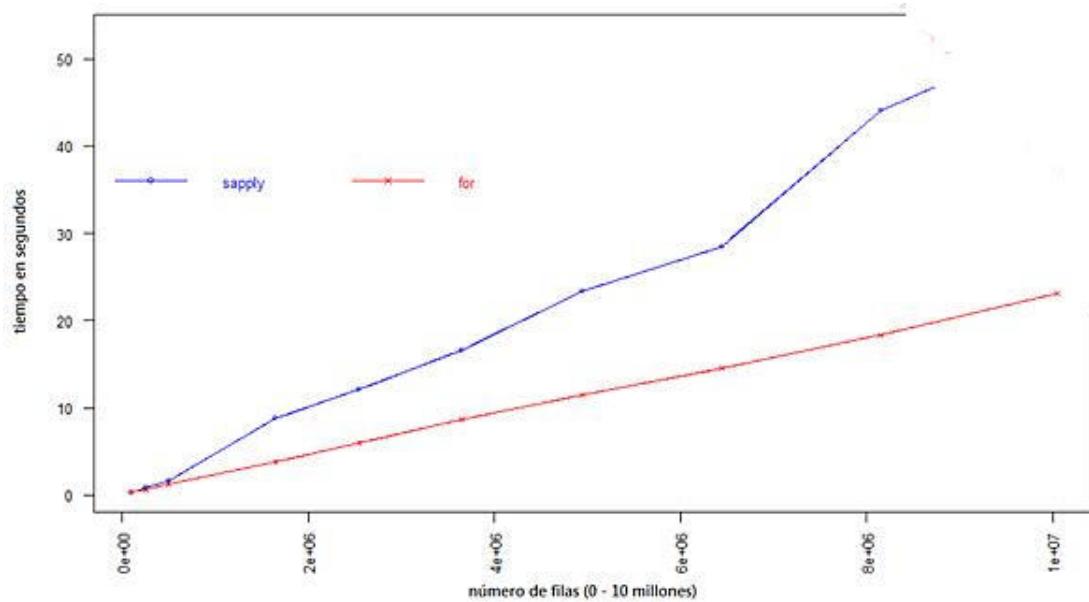
El resultado no es ni demasiado bonito ni muy legible, mejoraremos el gráfico.

```

lines(bench_sapply, type ="l",lwd=1,col = "blue")
lines(bench_sapply, type ="b",lwd=1,col = "blue"
      ,pch = 1,cex =0.5)
      # curva for, puntos
lines(bench_for, type ="l",lwd=1,col = "red")
lines(bench_for, type ="b",lwd=1,col = "red"
      ,pch = 4,cex =0.5)

legend("topleft",           # una leyenda arriba a la izquierda
       legend=c("sapply","for"),# leyenda
       lty=1,                  # trazo continuo
       lwd=1,                  # trazo poco grueso
       pch=c(1,4),             # tipo de símbolo
       col=c("blue","red"),
       ncol=2,                 # leyenda sobre dos columnas
       bty="n",                # sin marco
       cex=0.8,                # escribir más pequeño
       text.col=c("blue","red"),
       inset=0.01)            # posición de la leyenda

```



Comparación entre la duración de sapply y bucle for, de 0 a 10 millones de filas

Sorprendentemente, el tiempo que consume la función vectorial **sapply()** es más largo, para 10 millones de filas tenemos un factor de 2.

Observando el código, vemos que **sapply()** carga el **vector big** sobre sí mismo (**big <- sapply(big,f)**). Vamos a intentar mejorar su rendimiento agregando un array **big2** que se inicializa justo después de **big**, y cargando **big2** en lugar de cargar **big** sobre sí mismo. Las nuevas líneas que debe incluir en su código son las siguientes (agregue la primera al comienzo y sustituya las otras dos):

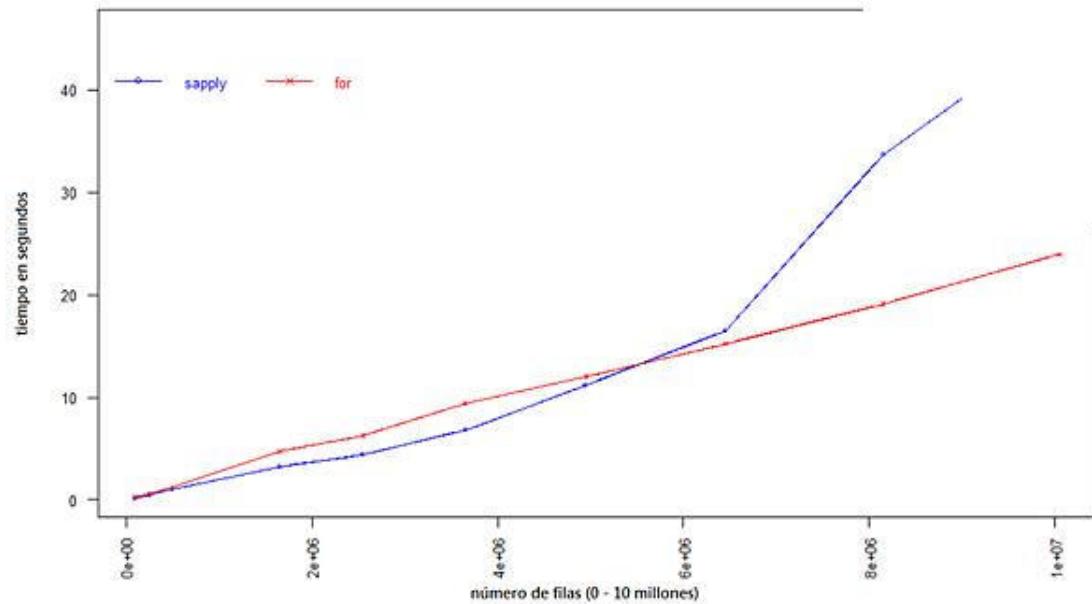
```
big2 <- vector()           # inic. gran vector a tratar
```

.../...

```
big2 <- sapply(big,f) # aplicación de f sobre big
```

.../...

```
for (i in 1:n) {big2[i] <- f(big[i])}
```



Duración de `sapply` y bucle `for`, de 0 a 10 millones de filas, sobre `big2`

Con esta forma de trabajar, `sapply()` resulta un poco más rápido que el bucle `for` para tratar volúmenes de menos de 5,5 millones de filas, y después su rendimiento se degrada y se vuelve hasta dos veces más costoso que el bucle `for` para un procesamiento de 10 millones de filas. Podemos imaginar que `sapply()` se vuelve más lento sobre la máquina utilizada cuando el uso de memoria es importante.

Lo que debe extraerse de esta experiencia, realizada sobre un PC ordinario, es que todo lo relativo al rendimiento no debe tomarse a la ligera.

El consejo de utilizar las capacidades vectoriales de R sigue siendo un buen consejo, pero cuando valide su programa y si encuentra un problema de rendimiento, no dude:

- En probar otros modos de programación cuando resulte demasiado lento.
- En comprobar la manera en la que se utiliza la memoria.

Respecto a los gráficos, estudie con atención la sintaxis del gráfico con las dos líneas quebradas. Encontramos diversas técnicas: diseño de un marco vacío antes de dibujar las dos líneas, opciones de presentación tales como el tamaño de la letra, cómo agregar dos líneas, grosor de la línea, superposición de una línea y de símbolos, configuración de una leyenda...

Tras trabajar con el tiempo, vamos a abordar un tipo nombrado que se utiliza a menudo en matemáticas y en física, que representa vectores sobre un plano: los números complejos.

11. Los números complejos

a. Manipulación básica de los números complejos

El lenguaje R permite manipular fácilmente números complejos, que puede almacenar en las estructuras que escoja.

Creemos un número complejo, veamos su longitud, su parte real y su parte imaginaria.

```
z <- 4 + 3i    # un número complejo
Mod(z)         # longitud de z
Re(z)          # la parte real
Im(z)          # la parte imaginaria
```

```
[1] 5
[1] 4
[1] 3
```

Ahora, veamos el argumento (el ángulo respecto a 1) de un número complejo.

```
z1 <- 1 + 1i
Arg(z1)*180/pi # ángulo en grados
Arg(z1)        # ángulo en radianes (argumento)
```

```
[1] 45
[1] 0.7853982
```

Como un número complejo puede verse como un vector sobre el plano, vamos a crear un pequeño programa que nos permita visualizar un número complejo.

b. Visualización de números complejos

El código de esta función utiliza nociones que ha visto antes; puede resultar un buen ejercicio detenerse sobre él un poco después de haber estudiado el resultado.

```
# visualizar los números complejos                      #

                                         # una función de visualización
plot_complex <- function(v){
  mr <- max(abs(Re(v)))      # semilongitud del gráfico
  mi <- max(abs(Im(v)))      # semialtura del gráfico

  w1 <- c(-mr,-mi)           # calcula 2 puntos extremos
                               # del gráfico
  w2 <- c( mr, mi)
  w  <- rbind(w1,w2)          # se construye una tabla
```

```

# de dos filas

t <- paste(as.character(v),
           "/longitud",
           as.character(round(Mod(v),3)),
           "/angulo",
           as.character( round(Arg(v)*180/pi)),
           "o"
           )

plot(w,                      # dibuja marco invisible punteado
      type ="n",
      bty = "n",
      xlab="1 -> Re(z)",
      ylab="i -> Im(z)"
      )
abline(v=0, lty = 4)          # ejes centrales punteados
abline(h=0, lty = 4)

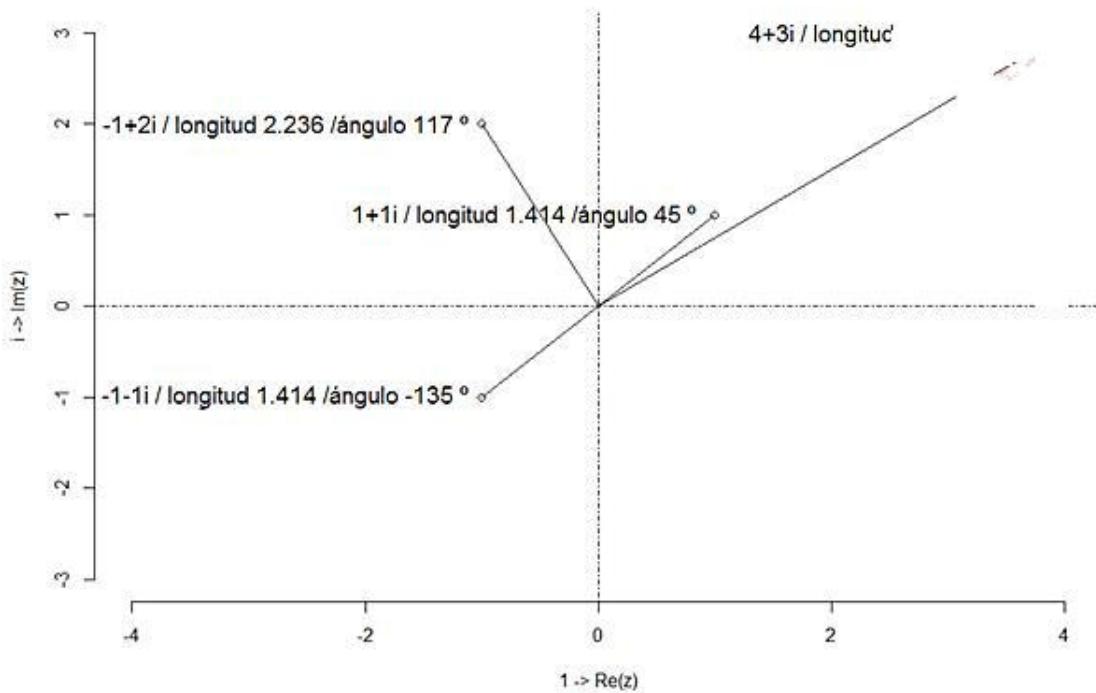
for (i in 1:length(v)) {     # linea+punto para cada complejo
  lines(c(0,v[i]),type ="l", asp = 1)
  lines(c(v[i]) ,type ="p", asp = 1)
  text(c(v[i]),labels =t[i], pos = 2,offset=0.5,cex=1.3)
}
}
```

Probemos nuestra función.

```

z2 <- -1 - 1i
z3 <- -1 + 2i

v <- c(z,z1,z2,z3)          # varios complejos
plot_complex(v)
```



Representación en el plano de: $4+3i$, $1+1i$, $-1+2i$, $-1-1i$

12. Programación orientada a objetos

En la actualidad, el desarrollo orientado a objetos se ha convertido en un estándar de la industria del desarrollo informático. Puede escribir sus propios programas en R sin dominar el desarrollo orientado a objetos. Y, a la inversa, si ya conoce la orientación a objetos o si desea acceder a una forma de pensamiento del desarrollo informático más estructurado, no dude en abordar este asunto.

Existen varios modelos de desarrollo orientado a objetos en R. El lenguaje R está construido, de hecho, como una ejecución de otro lenguaje llamado S que comprende, entre otros, mecanismos de orientación a objetos. Esto explica los nombres de los modelos orientados a objetos que hemos encontrado en alguna ocasión: S3, S4... Vamos a centrarnos en el modelo *Reference Class*: RC. Este modelo es muy **próximo al que se encuentra en otros lenguajes orientados a objetos, como Python**.

Este modelo es muy didáctico, no impone de entrada al usuario una comprensión de todos los conceptos de la orientación a objetos. Puede utilizarlo de manera sencilla y sin temor alguno, y a continuación evolucionar su pensamiento orientado a objetos mediante sucesivas iteraciones, sin peligro alguno.

Encontramos conceptos de orientación a objetos muy documentados, como las clases, los objetos/instancias, la instancia, la herencia, los métodos, la encapsulación, el «message passing» o la «mutabilidad».

 Los gurús de R escriben objetos utilizando el modelo «S3», que se corresponde con la manera en la que se han escrito los principales objetos básicos de R, cuyo comportamiento es muy previsible por parte de los usuarios de R. La implementación de los conceptos de orientación a objetos es parcial y no demasiado didáctica en términos de programación orientada a objetos.

Existen objetos «S4» criticados en ocasiones por grandes fabricantes como Google que no implementan el concepto de «mutabilidad», y poco el de encapsulación.

El modelo «RC», sobre el que nos vamos a centrar, se denomina en ocasiones «R5». Está codificado en R y utiliza el modelo «S4» aportándole la mayoría de los paradigmas orientados a objetos que han hecho que muchos desarrolladores indecisos despertaran su interés en «S4».

Existe un modelo «R6» muy prometedor, pero del que todavía no se sabe si sus creadores tendrán la capacidad de asegurar su viabilidad.

a. Clases y objetos, breve descripción

Clases

Como primera aproximación, consideremos una clase como un tipo de registro gestionado en memoria. En la siguiente definición de clase, vemos que este tipo de registro está definido en primer lugar por una lista de nombres de campos y por sus tipos respectivos.

```
## una clase simple sin método usuario ##

Persona <- setRefClass("Persona",
  fields = list (
    id      = "numeric",
    nombre   = "character",
    atributos = "vector",
    memoria  = "ANY",          # tipo todavía no conocido
    timer     = "ANY"           # ídem
  )
)
```

Esta definición describe dos campos particulares que todavía no han recibido el tipo, es una posibilidad muy interesante de los objetos RC.

El siguiente método permite obtener la lista de campos de una clase.

```
Persona$fields() # los campos
```

id	nombre	atributos	memoria	timer
"numeric"	"character"	"vector"	"ANY"	"ANY"

También es posible acceder a la lista de métodos de la clase, los métodos son funciones que están vinculadas a la clase y representan su comportamiento genérico, iveremos más adelante para qué sirve esto!

```
Persona$methods() # todos los métodos
```

[1]	"callSuper"	"copy"	"export"	"field"
	"getClass"	"getRefClass"	"import"	"initFields"
	"show"			
[10]	"trace"	"untrace"	"usingMethods"	

Si bien no hemos declarado métodos propios de la clase **Persona**, ya está dotada de ciertos métodos generales.

Objeto - instancia

Es momento de utilizar nuestra clase **Persona**. Para ello vamos a «instanciar» un objeto; dicho de otro modo, vamos a crear un «registro» en memoria que tendrá la estructura de la clase **Persona**.

```
## una instancia ##  
  
alguien <- Persona$new( # nueva instancia  
    id = 1, # primer campo  
    nombre = "Alguien" # otro campo  
)
```

Hemos creado el objeto **alguien**, que es una instancia de la clase **Persona**. Es muy fácil acceder a los valores de sus campos internos mediante el símbolo **\$**.

► Llegados a este punto, aquellos que trabajen en programación orientada a objetos con otros lenguajes podrían pensar que RC encapsula mal estos objetos, pero vamos a descubrir más adelante la noción de **accesor**, que aporta una encapsulación perfectamente correcta de los objetos RC.

```
alguien$id # simplemente
```

```
[1] 1
```

Nuestros slots libres pueden contener otras estructuras complejas, intentemos con una matriz.

```
m <- matrix(seq(1:9),3,3) # una matriz de 3 x 3  
alguien$memoria <- m # almacena los valores de la matriz  
alguien$memoria
```

```
[ ,1] [ ,2] [ ,3]  
[1,] 1 4 7  
[2,] 2 5 8  
[3,] 3 6 9
```

Encapsulación - accesores

El interés del siguiente código puede que no le parezca inmediato si no está familiarizado con los usos del desarrollo orientado a objetos. El hecho de acceder directamente a los valores de los campos de los objetos se considera como una práctica peligrosa si estos campos deben responder a restricciones de «calidad». Por ejemplo, se querría poder prohibir el hecho de introducir nombres vacíos o nombres cuya primera letra no estuviera escrita en mayúscula... Este asunto lo aborda la encapsulación de datos mediante procesamientos.

Para proteger los datos internos de sus objetos, los programadores experimentados utilizan métodos específicos para actualizarlos y extraerlos, llamados **accesores (accessor)**.

Hay preferencia por llamar a estos métodos de la siguiente manera: **getNombreDelcampo** para la extracción y **setNombreDelcampo** para la actualización del campo (en ocasiones los autores prefieren utilizar la palabra

`put`). Esto hace que para cada campo se declaren dos métodos; aquí, por ejemplo, para el campo nombre sería: `getNombre` y `setNombre`.

Si el objeto contiene muchos campos, el uso de este enfoque resulta algo tedioso. R proporciona una manera de trabajar que permite crear accesores triviales que puede sobrecargar más tarde para imponer sus propios controles. He aquí cómo proceder. Tenga en cuenta que no se le impide manipular los objetos, aunque esto supone una buena práctica en el marco de los objetos escritos en un estilo ortodoxo de programación robusta.

Para permitirle comprender el siguiente código, examinemos su primera instrucción clave.

```
# los campos de la clase Persona  
attributes(Persona$fields())
```

```
$names  
[1] "id"          "nombre"        "atributos"    "memoria"      "timer"
```

La función `attributes()`, muy general en R, nos permite extraer los atributos de cualquier objeto R (objeto, matriz...). Lo utilizamos aquí, pues sabemos que las clases RC contienen un método `fields()` que posee un atributo `names`. Por ello podemos construir una fachada de `vector` con los nombres de los campos como cadenas de caracteres.

 Cuando descubra un nuevo artefacto en R, intente utilizar la función `attributes` (`de_la_cosa`) para descubrir sus eventuales atributos, pues resulta muy ilustrativo.

Nuestra primera línea de código es la siguiente:

```
pf <- attributes(Persona$fields())$names  
pf
```

```
[1] "id"          "nombre"        "atributos"    "memoria"      "timer"
```

El método genérico `accessors()` nos va a permitir crear nuestros accesores `getX...` y `setX...` automáticamente.

```
Persona$accessors(pf)    # accessors getX... y setX...  
Persona$methods()         # get o set ¡y luego field con mayúsculas!
```

```
[1] "callSuper"      "copy"           "export"        "field"  
"getAtributos"     "getClass"       "getId"         "getMemoria"  
"getNombre"  
[10]"getRefClass"   "getTimer"        "import"        "initFields"  
"setAtributos"     "setId"          "setMemoria"    "setNombre"  
"setTimer"  
[19]"show"          "trace"          "untrace"       "usingMethods
```

Comparándolo con la misma lista de métodos mostrada más arriba en el párrafo, comprobará que se han creado automáticamente nuevos métodos, los accesores: `getAtributos()`, `getId()`, `getMemoria()`, `getNombre()`, `getTimer()`, `setAtributos()`, `setId()`, `setMemoria()`, `setNombre()`, `setTimer()`.

La manera de utilizarlos es trivial.

```
alguien$getNombre()                      # se obtiene el nombre
alguien$setMemoria(m+1)                   # actualización de la memoria
alguien$getMemoria()                     # leer la memoria
```

```
[1] "Alguien"
[,1] [,2] [,3]
[1,]    2    5    8
[2,]    3    6    9
[3,]    4    7   10
```

Zoom sobre la noción de atributos

Este tema no pertenece a la programación orientada a objetos, pero, como hemos descubierto esta noción, podemos abordarla ahora.

Función attributes()

Hemos visto más arriba la función **attributes()** de R. Vamos a aplicarla sobre diversos artefactos de R:

```
attributes(alguien)      # atributo en el sentido R
attributes(Persona)
m <- matrix(1:12, nrow = 4)
attributes(m)
attributes(m)$dim
```

```
[ atributos de alguien --> alguien es un objeto de la clase Persona ]
$.xData
<environment: 0x000000014592630>

$class
[1] "Persona"
attr(,"package")
[1] ".GlobalEnv"

[ atributos de la clase Persona; hay muchos, ¡los cortaremos! ]
$className
[1] "Persona"
attr(,"package")
[1] ".GlobalEnv"

$package
[1] ".GlobalEnv"

$generator
Reference class object of class "refGeneratorSlot"
Field "def":
Reference Class "Persona":

Class fields:

Name:      id      nombre atributos     memoria      timer
Class:  numeric character      vector        ANY        ANY
```

.../...

```
[ atributos de una matriz m: SUS DIMENSIONES]
$dim
[1] 4 3
```

Esta función **attributes()** devuelve los atributos que ha incluido el programador del artefacto, cada artefacto posee o no diversos atributos accesibles mediante el símbolo **\$** (isintaxis que es preciso recordar!).

```
attributes(mi_artefacto)$uno_de_sus_atributos
```

Método **atributos()** de RC [sin «e»]

En RC dispone de una noción de atributo que puede utilizar fácilmente sobre sus objetos. No se corresponde con la noción de atributo que acabamos de ver, sino que puede resultar útil para cualificar de un vistazo un objeto (pero no una clase).

```
# en el sentido RC
alguien$setAtributos(c("guapo", "grande")) # creación de atributos
alguien$getAtributos()                      # obtención de atributos
                                              # actualización de atributos
alguien$setAtributos(c(alguien$getAtributos(), "fuerte"))
alguien$getAtributos()
```

```
[1] "guapo"   "grande"
[1] "guapo"   "grande" "fuerte"
```

El método **setAtributos()** anula y reemplaza los atributos previos de un objeto RC. Si bien, cuando quiera actualizar los atributos RC, debe realizarlo como se ha mostrado en el código anterior, extrayendo los atributos anteriores y agregando a continuación los nuevos antes de recrear el conjunto de nuevos atributos.

b. Constructores

Para instanciar un nuevo objeto a partir de una clase RC, se invoca a un método particular, que llamamos constructor en programación orientada a objetos.

Instanciando el objeto **alguien** mediante el método **new()**:

```
alguien <- Persona$new()
```

De hecho, hemos invocado el constructor genérico de los objetos de RC.

Este constructor puede sobrecargarse, es decir, puede agregarle sus propias instrucciones, como por ejemplo imponer valores por defecto para algunos de sus campos o realizar procesamientos y controles antes de crear el objeto.

Como puede adivinar, esta sobrecarga se realiza a nivel de la clase. Extrañamente, el nombre íntimo del constructor en RC es **initialize()**.

Al principio, la forma de escribir el constructor puede parecerle algo singular, pues hace referencia a nociones que todavía no hemos abordado. Mientras no domine estas nociones, le bastará con situar su código donde le vamos a proponer y conservar tal cual la sección del código de encapsulación que le proporcionamos, sin reflexionar mucho en ello.

Vamos a agregar una nueva versión del constructor (que es un método más entre los otros) a la clase **Persona**.

```
Persona$methods( initialize = # siempre el nombre del constructor
  function ( ##
    ## su propio código, 1 de 2
    ## valores por defecto de sus parámetros
    ##
    nombre = "Desconocido", # argumento 1
    id = 0,                 # argumento 2
    ##
    ## fin de su código, 1 de 2
    ##
    ...){ # ... : param super clase
  callSuper(...) # llamada param super clase
  ##
  ## inicio de su propio código 2 de 2
  ## utilice <<- en lugar de <-
  ##
  nombre <<- nombre # recuperar argumento
  id     <<- id   # recuperar argumento
  timer  <<- Sys.time() # para el ejemplo
  ##
  ## fin de su propio código, 2 de 2
  ##
  .self      # ¡hecho!
}
)
```

Sin duda le interesarán los siguientes puntos explicativos:

- **methods()** es el método que permite crear un método.
- **initialize** = es la sintaxis que permite lanzar la definición del método **initialize()**.
- Este método es, de hecho, una función con sus parámetros entre paréntesis.
- El último de los parámetros es algo extraño, se escribe mediante tres puntos: "...".
- Estos "..." se corresponden con una sintaxis habitual en R, cuyo objetivo es indicar que se pasará a la función otros parámetros si hiciera falta, típicamente estos parámetros son aquellos que se utilizan en otras funciones invocadas dentro de la función. Los parámetros deberán tener los nombres esperados por las funciones invocadas dentro de la nueva función. Aquí, se trata de la función **callSuper()**, que recupera todos los parámetros que necesita mediante los tres puntos.
- **callSuper()** invoca el constructor genérico de la clase situada por encima de **Persona** en la jerarquía de clases.
- Se utiliza **<<-** para aumentar el alcance de la variable, es decir, para que pueda verse desde los demás métodos de la clase.
- **nombre <<- nombre** significa que utilizaremos el nombre introducido como argumento, sabiendo que, si no se proporciona ningún nombre al constructor durante la creación del objeto, se utilizará el nombre por defecto. Aquí se

ha seleccionado «**Desconocido**» como nombre por defecto.

- **.self** es la instrucción que designa la referencia a sí mismo, lo que permite que nuestra función devuelva el objeto que se está construyendo. En efecto, la última variable de una función es la que se devuelve por esta función (aquí, el constructor **initialize()**) y es, por tanto, **.self**, que se corresponde con el propio objeto que se ha de construir.

Tranquilícese, este código es un código que no resulta fácil de comprender para un debutante en programación orientada a objetos, pero que **puede contentarse con modificar** para fabricar sus propios constructores. Le bastará con transformar los dos bloques de código entre los **## ##** (sin olvidarse de cambiar el nombre de la clase sobre la que se está creando un constructor... ¡no todas las clases se llaman **Persona**!).

Una vez realizado el trabajo, el hecho de instanciar un nuevo objeto de la clase **Persona** invoca automáticamente el constructor redefinido.

```
alguien <- Persona$new()
alguien$getNombre()
```

```
[1] "Desconocido"
```

Como no le hemos proporcionado ningún **nom** ni **id** durante la creación del objeto **alguien** como instancia de la clase **Persona**, el constructor que hemos escrito más arriba ha asignado los valores por defecto, en este caso "Desconocido" para el nombre (y 0 para el **id**).

Si se crea un nuevo individuo, pero se informan los argumentos, el constructor los tendrá en cuenta. Evidentemente, en ambos casos el constructor ha informado automáticamente el campo **timer**, pues le hemos pedido que lo haga en la construcción (instanciación) de los objetos de la clase Person.

```
pablo <- Persona$new(nombre = "Pablo",
                      id = 2)
pablo$getNombre()
pablo$getId()
pablo$getTimer()
```

```
[1] "Pablo"
[1] 2
[1] "2017-10-04 15:54:18 CET"
```

c. Herencia

Una de las nociones más potentes de la programación orientada a objetos es la herencia. La herencia le permite gestionar todas las situaciones **para las que «cualquier cosa» es una especie de «otra cosa»**.

Podemos ver esto como la expresión de la relación «es un» o «is_a» en inglés: «un perro es un animal» (es un tipo de animal).

Tras constatar este hecho, todas las características generales de los animales son aplicables al perro, salvo las características que son propias de los perros. En términos de orientación a objetos, podríamos decir que la **clase perro hereda de la clase animal**, y que se va a **sobrecargar la clase perro** con las especificidades que le son propias.

En el siguiente ejemplo, crearemos una clase data scientist, utilizando el hecho de que los data scientists son también Personas según el mecanismo de herencia: «Data_scientist is_a Persona».

Vamos además a dotar a los data scientists de competencias eventuales en R y en estadísticas, y de la capacidad de presentarse diciendo hola, mediante un mecanismo de herencia de la clase **Persona**.

```
## herencia y nuevos campos
Data_scientist <- setRefClass("Data_scientist",

  contains = "Persona",

  fields = list (
    competencia_R      = "logical",
    competencia_estad = "logical"
  )
)
```

La clase **Data_scientist** hereda de la clase **Persona** mediante la mención **contains =**.

Se agregan dos campos correspondientes a las eventuales competencias de esta.

```
## sobrecarga de métodos, aquí el constructor          ##
Data_scientist$methods( initialize =
  function(competencia_R = FALSE,...){
    callSuper(...)
    competencia_R      <- competencia_R
    competencia_estad <- TRUE
    .self
  }
)
```

El constructor se ha **sobrecargado** agregando la inicialización de las correspondientes competencias. La **competencia_R** vendrá del valor que se proporcione al constructor (valor proporcionado o valor por defecto que es **FALSE** si no se indica nada), la **competencia_estad** vale **TRUE**.

Dotemos a nuestros data scientists de una capacidad de comunicación.

```
## un nuevo método para esta clase          ##
Data_scientist$methods( comunica =
  function( ){
    ch1 <- ""
    ch2 <- ""
    if (competencia_R){
      ch1 <- ", especialista en R"
    }
    if (competencia_estad){
      ch2 <- ", apasionado de la estadística"
    }
  }
)
```

```

        print (paste("Hola, soy",
                     .self$nombre,
                     ch1,
                     ch2
                   ))
      }
)

```

El data scientist puede, ahora, presentarse indicando sus eventuales competencias.

Para mantener un buen formato, creemos los accesores de ambas competencias.

```

## creación de nuevos accesores
## Data_scientist$accessors(c("competencia_R",
#                            "competencia_estad"
# ))    # accessors getX... y setX...

```

Conviene comprobar que todo funciona bien y que nuestra clase contiene los campos y los métodos que deseamos.

```

Data_scientist$fields()
Data_scientist$methods()

```

Es hora de intentar crear un data scientist desconocido.

```

alguien <- Data_scientist$new() # valores por defecto
alguien$getNombre()
alguien$getCompetencia_R()

```

```

[1] "Desconocido"
[1] FALSE

```

Y ahora probemos a inicializar un data scientist concreto.

```

pedro <- Data_scientist$new(nombre = "Pedro",
                           id = 100,
                           competencia_R = TRUE,
                           memoria = "Pi = 3.1416",
                           atributos = c("pequeño", "astuto")
                         )

```

En la construcción de este data scientist, evocamos argumentos propios de los data scientist (la eventual competencia en R), y también argumentos vinculados al hecho de que es una Persona y, por último, agregamos algunos atributos al vuelo.

Pedro está en posición de comunicarse, basta con invocar su método correspondiente; para ello decimos en ocasiones que **los métodos representan el comportamiento del objeto**.

```
pedro$comunica()           # pedro comunica ... ;-)
```

```
[1] "Hola, soy Pedro, especialista en R, apasionado de la estadística"
```

Pedro no es tan bueno en estadística, de modo que lo vamos a indicar utilizando el acceso correspondiente.

```
pedro$setCompetencia_estad(FALSE)
```

Pedro va ahora a tener en cuenta este hecho en su presentación.

```
pedro$comunica()           # pedro comunica ... :-()
```

```
[1] "Hola, soy Pedro, especialista en R"
```

Pero, de hecho, ¿quién es Pedro y qué contiene su memoria?

```
pedro$getId()  
pedro$getMemoria()  
pedro$getAtributos()
```

```
[1] 100  
[1] "Pi = 3.1416"  
[1] "pequeño" "astuto"
```

Es un pequeño astuto cuyo identificador es 100 y que ha memorizado el valor de Pi.

d. Objetos mutables

Hagamos una pequeña experiencia esquizofrénica: asignemos Pedro a Pablo.

```
pablo <- pedro  
pablo$comunica()
```

```
[1] "Hola, soy Pedro, especialista en R"
```

Pablo se comunica como Pedro, pero ¿es Pedro o simplemente una copia de Pedro?

Para responder, se impone una nueva experiencia, vamos a cambiar el contenido de la memoria de Pablo e interrogar a la memoria de Pedro. ¿Qué le parece?

```
pablo$setMemoria("Pi = oh no lo sé")
pedro$getMemoria()
```

```
[1] "Pi = oh no lo sé "
```

Efectivamente, Pablo y Pedro son un único y el mismo data scientist.

Los objetos RC son mutables: si asigna **en su totalidad** un objeto a otro o dentro de otro como argumento, y a continuación cambia el primero, el otro estará totalmente sincronizado. Ocurre al revés que el comportamiento habitual de las variables en R, aunque es una correcta implementación de los conceptos de orientación a objetos.

Si no desea este comportamiento, tendrá que utilizar el método **copy()**, método genérico de sus objetos RC.

```
tristan <- pedro$copy()
pedro$setMemoria("¿quién soy? ....")
tristan$getMemoria()
```

```
[1] "Pi = oh no lo sé "
```

Tristán es otro data scientist, un clon que evoluciona a su ritmo, sea cual sea la evolución de Pedro (incluso tras su destrucción de memoria mediante la instrucción **rm()**).

```
rm(pedro)
tristan$setNombre("Tristan clon de Pedro")
tristan$comunica()
pedro$comunica()
```

```
[1] "Hola, soy Tristan clon de Pedro, especialista en R"
Error: object 'pedro' not found
```

Pero ¿qué ha sido de Pablo tras la destrucción de Pedro, con el que estaba perfectamente sincronizado? ¿Qué le parece?

```
pablo$comunica()
```

```
[1] "Hola, soy Pedro, especialista en R"
```

La destrucción de Pedro no ha afectado a Pablo, la destrucción no es mutable en el modelo RC.

La programación orientada a objetos permite resolver de manera elegante y fiable numerosos problemas de programación, aunque no es, con diferencia, la más practicada entre los data scientists. Sin embargo, aquellos que la dominen pueden ir más lejos en la resolución de nuevos problemas, en particular cuando se salgan del campo del aprendizaje estadístico. Los equipos informáticos codifican en programación orientada a objetos; sería una pena privarse de este ámbito común de conocimiento y ser incapaz de utilizar, aunque solo sea en grandes líneas, la implementación operacional que obtendrá de sus algoritmos.

Existe una gran comunidad de práctica entre los especialistas de la inteligencia artificial y los data scientists. **La programación orientada a objetos puede ser una manera de abordar la implementación de la IA**, esta sería otra buena razón para aprenderla. Además, la programación orientada a objetos solicita su imaginación y le ayudará a tomar perspectiva sobre sus prácticas de escritura de programas.

Ahora vamos a pasar por encima de una estructura de datos muy útil **que podrá reutilizar en sus programas**, y que vamos a implementar en programación orientada a objetos: una «pila» o *stack* en inglés.

e. Gestión de la pila: implementación Orientada a Objetos con RC

Desde el inicio de la informática y, a menudo, en el mundo de la inteligencia artificial, pero también en el núcleo de sus máquinas, se encuentra la noción de pila (y, por consiguiente, la noción de cola).

Una pila (*stack*) representa un apilamiento de estructuras de datos cualesquiera. Para imaginarse su funcionamiento, piense simplemente en una pila de platos:

- La pila puede estar vacía.
- Se agregan los platos por la parte superior de la pila: en inglés *put*.
- Se quitan platos por la parte superior de la pila: en inglés *pop*.
- El último plato apilado será el primero desapilado (en inglés *Last In - First Out: LIFO*, último en entrar, primero en salir).

 Puede pensar en una cola como una pila sobre la que habría decidido arriesgarse para retirar los platos por la parte inferior: «pull».

Hemos escogido implementar el segundo algoritmo descrito en el siguiente artículo de Wikipedia bajo la mención de «linked list» (página accesible en 2015): [https://en.wikipedia.org/wiki/Stack_\(abstract_data_type\)](https://en.wikipedia.org/wiki/Stack_(abstract_data_type)).

No dude en consultarla.

Esta resiste a ciertos problemas de desbordamiento de memoria.

A este algoritmo le hemos agregado diversas funcionalidades para facilitar la explotación de las pilas, en particular el apilamiento/desapilamiento de una lista de objetos.

El ejercicio resulta interesante, pues sirve para constatar que la programación orientada a objetos de RC le permite implementar con bastante facilidad algoritmos que puede encontrar en diversas fuentes.

Con este código dispondrá de una sólida base para gestionar sus propias pilas en R e implementar usted mismo los numerosos algoritmos que utilizan esta noción.

La potencia de la implementación orientada a objetos le permitirá crear numerosas pilas, duplicarlas, fabricar pilas de pilas mutables...

En primer lugar es preciso crear dos clases para gestionar la memorización de las posiciones de nuestros «platos».

```
#-----#
## un contenedor para los datos y el puntero          ##
Frame <- setRefClass("Frame",
                      fields = list (
```

```

        data_ = "ANY", # data
        last_ = "ANY" # Frame ou NULL
    )
)

#-----#
## la pila ##

Stack <- setRefClass("Stack",
    fields = list (
        head_ = "ANY",      # Frame o NULL
        size_ = "numeric", # integer sería
                           # un poco limitante

        # propio de mi implementación
        # variables "globales" de la stack
        memo_ = "ANY",      # data o NULL
        head_new = "ANY"    # Frame o NULL
    )
)
#-----#

```

La sintaxis es la que hemos visto más arriba; la capacidad de definir campos sin precisar el tipo, mediante «**ANY**», resulta muy útil.

Para poder extender ciertas funciones de esta estructura, hemos agregado dos campos que no están en el algoritmo descrito en Wikipedia: **memo_** y **head_new**. No se preocupe respecto a la comprensión general del algoritmo.

La pila necesita una inicialización precisa, con una longitud (altura) de pila nula. Por lo tanto, hay que definir su constructor para disponer de una pila vacía tras la instanciación.

Se trabaja con enteros de tipo long (L) para que el tamaño de la pila no pueda crear complicaciones (generar un bug) rápidamente en su programa.

```

## método de inicialización de la pila ##

Stack$methods( initialize =
    function (...){ # ... : param super clase
        callSuper(...) # appel param super classe
        head_ <- NULL
        size_ <- 0L
        memo_ <- NULL
        .self       # ;hecho!
    }
)

```

El primer método que hay que crear es el que nos va a permitir agregar un «plato» sobre la pila (vacía, llegado el caso).

```

## apilar un objeto en la pila ##
```

```

Stack$methods( put =
    function (x){    # put data

        head_new      <<- Frame$new()
        head_new$data_ <<- x
        head_new$last_ <<- head_
        head_          <<- head_new
        head_new       <<- NULL
        size_          <<- size_ + 1L
    }
)

```

Tras invocar este método, hay un nuevo objeto sobre la pila y el tamaño de esta se ve incrementado.

Llegados a este punto, podríamos querer leer, sin desapilarlo, el objeto de la parte superior de la pila. El siguiente método no es indispensable; sin embargo, es fácil de codificar y resulta muy práctico. En inglés, se suele llamar **peek()**.

```

## leer el objeto de la parte superior de la pila (sin desapilarlo!)##

Stack$methods( peek =
    function (){
        return(.self$head_$data_)
    }
)

```

Con la misma idea, se dota un accesror para conocer el tamaño de la pila.

```

## obtener el tamaño de la pila ##

Stack$methods( size =
    function (){
        return(.self$size_)
    }
)

```

He aquí, por último, el método más interesante, el método **pop()**, que permite desapilar el objeto situado en lo alto de la pila.

```

## desapila y obtiene el objeto de la parte superior de la pila ##

Stack$methods( pop =
    function (){
        if (size_ == 0) {
            memo_ <<- NULL
            return(NULL)
        }

        memo_      <<- head_$data_

```

```

        head_      <<- head_$last_
        size_      <<- size_ - 1L
        return(memo_)
    }
)

```

Se puede utilizar el método **pop()** e invocarlo varias veces para desapilar toda una lista de objetos. Se obtiene la lista de objetos desapilados: todos los objetos, si no se ha pasado ningún argumento, o el número de objetos deseado si se ha precisado un número de objetos para desapilar.

```

## desapila una lista de objetos                      ##
## por defecto, todos los objetos, o el número deseado ##

Stack$methods( pop_list =
  function (n = -1){
    i <- .self$size_
    if (n >= 0){ m <- n} else {m <- i}
    l <- list()
    while ( (m > 0) && !is.null(.self$pop()) ){
      l[[m]] <- .self$memo_
      m <- m- 1
    }
    return(l)
  }
)

```

Por último, podemos apilar toda una lista.

```

## apila el contenido de una lista (inverso de desapilar)      ##

Stack$methods( put_list =
  function (l){ # put data
    n <- length(l)
    for (i in 1:n) {.self$put(l[[i]])}
  }
)

```

Deberíamos realizar algunas pruebas.

```

s <- Stack$new()          # s es una nueva pila
s$put("primero")         # en la parte inferior de la pila
s$peek()                  # está bien ahí
s$put("segundo")
s$put("tercero")
s$size()                  # pila de 3 elementos

s$pop()                   # se desapila del último al primero
s$pop()
s$pop()

```

```
s$peek()          # ya no hay nada,  
s$pop()           # sin problema si se desapila demasiado  
s$peek()          # es efectivamente una pila vacía
```

```
[1] "primero"  
[1] 3  
  
[1] "tercero"  
[1] "segundo"  
[1] "primero"  
  
NULL  
NULL  
NULL  
[1] 0
```

No dude en comprobar los demás métodos (que gestionan listas).

A continuación puede apilar estructuras de cualquier naturaleza, objetos RC, matrices, otras pilas: compruébelo a su vez.

Los objetos RC mutables siguen siendo mutables mediante este mecanismo.

Como ejercicio, intente fabricar un método **pull()** que extraiga un objeto a partir desde la parte inferior, lo cual producirá una cola (no optimizada, ipero compatible con la pila!).

Esta toma de contacto de R llega a su fin en lo que respecta a aquellos aspectos relacionados con la programación y la manipulación de estructuras de datos. A continuación nos centraremos en la manipulación de los datos de manera previa a su uso dentro del marco del machine learning.

Manipulación de los datos

1. Lectura de los datos: fundamentos

Para realizar sus manipulaciones básicas, los archivos que utiliza son, a menudo, archivos .csv (se trata de un formato de exportación habitual de muchas aplicaciones, entre ellas Excel).

Para cargar dichos archivos, hay que definir los separadores de los campos, el símbolo que codifica la coma decimal, si existen o no los títulos de las columnas, los caracteres que se corresponden con los datos que faltan y el tipo de codificación del archivo (habitualmente UTF8). Si no dispone de esta información, puede realizar sucesivos intentos y converger rápidamente a una lectura eficaz de su archivo. A menudo basta con abrir el archivo con un editor como Emacs o Notepad++ para saber qué se trae uno entre manos.

He aquí un código típico, que va a leer un archivo que encontrará en el sitio que acompaña a este libro.

```
## Leer los datos ##

data <- read.csv("datatest1.csv",
                 sep=";",
                 dec=",",
                 na.strings=c(".", "NA", "", "?", "#DIV/0!"),
                 strip.white=TRUE,
                 encoding="UTF-8")
```

Si hace doble clic en «data» dentro de la ventana superior derecha de RStudio, obtendrá una tabla; he aquí las primeras filas.

	v1	v2	v3	v4	v5	v6	v7	v8	v9	v10
1	0.54946252	1.586331941	2.10823617	#.05586156	-0.0179278	-1.257893248	-1.259113248	0.486620908	1.426345e+05	5.166489813
2	1.48579132	1.597558143	1.13792558	0.65437230	-0.89705506	-1.147172055	-0.287118468	0.095308470	1.272934e+01	9.5060609047
3	1.73990167	0.934108768	0.98130931	5.30271879	-1.17964568	-0.486741275	-0.000148375	-0.551518446	1.489988e+02	9.363577816
4	0.16007145	0.498876783	0.16257231	#.55807357	0.16057152	0.297406759	-0.310121299	-0.283699385	1.002504e+00	2.877580545
5	1.14592353	0.004120035	2.61612099	1.94668338	-0.39551664	0.386818000	-1.348478446	-1.289517129	4.303599e-01	7.035678000
6	0.97562693	1.007962456	2.685435158	2.20193524	-0.24141447	-0.365582917	-1.450810461	-1.321969645	3.890265e+02	24.826151913

Visualización de «data» en RStudio

La última columna se corresponde con una clase que vamos a transformar en **fatores**.

```
data$clase=as.factor(data$clase) # es conveniente transformar
                                # los valores discretos
                                # no ordenados
                                # en factores
```

La estructura de datos de **data** es de un tipo que no hemos estudiado: **data.frame**. Este tipo de estructura de datos está muy bien adaptado a las tareas más habituales del machine learning (para aquellos que conocen Python, podría compararse con las estructuras de datos de Panda).

La manera de acceder a las filas y a las columnas es igual que con las matrices, aunque se utiliza también a menudo la sintaxis **mi_data_frame\$mi_columna** para acceder a una columna. Las siguientes funciones se utilizan de manera trivial; pruébelas y observe los resultados.

```

str(data)      # ver un extracto de los datos y su tipo
dim(data)      # núm filas, núm columnas... donde:
nrow(data)     # núm filas
ncol(data)     # núm columnas
names(data)    # nombre de las columnas

```

Podemos observar los tipos de datos implicados en este **data.frame**.

```

typeof(data$v1)      # devuelve la respuesta: "double"
                     # es el formato de codificación interna,
                     # ¡no es su naturaleza en R!
is.double(data$v1)   # ¡confirmación!
is.numeric(data$v1)  # también verdadero, es un valor numérico
class(data$v1)       # naturaleza de la variable

anyNA(data)          # comprueba que los datos no contienen
                     # valores desconocidos

```

```
[1] FALSE
```

No faltan datos en nuestro **data.frame**.

Para crear un **data.frame** a partir de una estructura cualquiera de datos, basta con invocarla.

```
df <- data.frame(data)  # creación de un data.frame
```

Podemos acceder fácilmente a las filas.

```

df[1,]              # primera fila
d <- df[1,]          # almacenar la primera fila
is.data.frame(d)    # un data frame de una fila
df[10:20,]           # filas 10 a 20 incluidas

```

Podemos jugar con las columnas.

```

## las columnas                      ##
df[,1]                  # columna 1
c <- df[,1]               # la almacenamos
is.vector(c)             # es un vector que representa la columna

c <- data.frame(c) # transformación del vector en data.frame
is.data.frame(c) # ¡funciona!

c <- df[,1:5]   # si se toman cinco columnas a la vez
is.data.frame(c) # ¡es un data.frame de cinco columnas!

```

```

d <- df[4:6,3:5] # extracción de las columnas 3 a 5 y de las filas 4 a 6
dim(d)
is.data.frame(d) # es un data.frame de tres filas por tres columnas

df$v1           # columna 1 a partir de su nombre: v1
c <- df["v1"]    # es un data.frame
is.data.frame(c) # ;funciona!

```

Es posible obtener estadísticas básicas sobre el **data.frame df**.

```
summary(df)          # estadísticas básicas sobre df
```

Vamos a ver con más detalle la manera de extraer varias columnas de un **data.frame**.

2. Manipulación de las columnas de un data.frame

Podemos escoger las columnas a partir de su nombre.

```

v <- c("v1","v2","v3","clase") # las clases que queremos guardar
df_ <- df[,v]                 # data.frame de las cuatro columnas
                                # extraídas
names(df_)

```

```
[1] "v1"      "v2"      "v3"      "clase"
```

Disponemos de la función **subset()** para realizar numerosas manipulaciones.

```

## filtrar las columnas con subset                         ##
df_ <- subset(df_, select = -2)      # eliminar columna 2
df_ <- subset(df_, select = -c(1,2)) # eliminar columnas 1 y 2
df_ <- subset(df_, select = -v1)     # eliminar columna v1
df_ <- subset(df_, select = -c(v1)) # ídem eliminar columna v1
df_ <- subset(df_, select = -c(v1,v2))# eliminar columnas v1 y v2
df_ <- subset(df_, select = c(1,2)) # guardar columnas 1 y 2
df_ <- subset(df_, select = c(v1,v3))# guardar columnas v1 y v3
df_ <- subset(df_, select = v1:v3)   # guardar columnas v1 a v3

```

¿Cuáles son los distintos valores de la columna **clase**?

```
unique(df$clase)      # tabla de factores
```

```
[1] 0 1 2 3
```

Para obtener las proporciones de las clases, se utiliza **prop.table(table())**.

```
prop.table(table(df_$clase)) # proporción por factor
```

```
0      1      2      3  
0.25  0.25  0.25  0.25
```

3. Cálculos simples sobre un data.frame

a. Cálculos sobre las columnas y las filas

Cálculos integrados en R

Vamos a realizar estos cálculos sobre **data.frame** de cuatro columnas.

```
v <-c("v1","v2","v3","clase")  
df_ <- df[,v]
```

```
# funciones muy rápidas, no utilizarlas si existen NA  
# pues su implementación es diferente según los sistemas  
  
colSums(df_)           # suma por columna (4 valores)  
colMeans(df_)          # media por columna (4 valores)  
  
rowSums(df_)           # suma por fila (100 valores)  
rowMeans(df_)          # media por fila (100 valores)
```

```
v1          v2          v3          clase  
4.237215   12.993813  -3.099839  150.000000  
v1          v2          v3          clase  
0.04237215 0.12993813 -0.03099839 1.50000000
```

.../...

Agrupación por nivel de ruptura entre las filas

Tras haber manipulado las columnas enteras, podemos querer realizar cálculos (es decir, obtener subtotales, medias parciales...) por ruptura, aquí las clases. Es un poco parecido al «group by» del lenguaje SQL.

```
## aggregate ... group by ##  
  
if(require("stats")==FALSE) install.packages("stats")  
require("stats")           # estadísticas básicas
```

```

library(help = "stats")                      # la ayuda

a <- aggregate(.~ clase, data = df_, mean)   # data.frame
                                              # media por clase
                                              # por columna numérica

a

```

	clase	v1	v2	v3
1	0	1.0071463	1.1175014	1.0375074
2	1	-1.0497790	-0.8778992	-0.8517238
3	2	-0.9960628	1.2509849	-1.1667655
4	3	1.2081841	-0.9708346	0.8569884

La sintaxis de **aggregate()** es interesante, se definen las rupturas mediante una fórmula, y se indica la función que se aplicará sobre cada nivel de ruptura.

b. Manipulación de las filas

Combinación de filas y row.names

La primera manipulación que se presenta aquí tiene como objetivo crear un **data.frame** con filas extraídas de otro **data.frame**.

```

## combinación de filas
## 
a <- df_[1:5, ]      # extracción de las 5 primeras filas
b <- df_[96:100, ]    # extracción de las 5 últimas filas
c <- rbind(a,b)       # combinación de los dos grupos de filas
nrow(c)

```

[1] 10

Obtenemos efectivamente 10 filas en este pequeño **data.frame**.

Cuando visualiza **c**, obtiene una sorpresa: aparece la noción de número de fila. Estudie bien las siguientes explicaciones para no llevarse sorpresas desagradables en el eventual uso de **row.names**.

	row.names	v1	v2	v3
1	1	0.5494625	1.586331941	2.108236
2	2	1.4057952	1.597558143	1.13792558
3	3	1.7199037	0.934108768	0.98138931 0
4	4	-0.1600714	0.498876281	0.16257231 0
5	5	1.1459235	0.004180835	2.61611093 0
6	96	1.4234600	0.014393778	-0.06852483 3
7	97	1.6994855	-0.410463491	0.93231236 3
8	98	1.3187510	-0.491640404	1.60657818 3
9	99	0.8765281	-1.554613409	2.06370286 3
10	100	1.1922164	-1.181973952	0.84221728 3

data.frame obtenido mediante la extracción y combinación de filas

Si tiene la curiosidad de visualizar el **data.frame** original o el **data.frame** de las cinco primeras filas, ¡comprobará que no hacen mención a **row.names**!

A la inversa, el **data.frame** formado por estas últimas filas sí contiene esta noción. ¿Por qué?

R «sabe» que ha roto la secuencia original de las filas. «Por si acaso» conserva los números de fila originales para que pueda fijarse en ellos. Cuando mezcle dos **data.frame** según las filas y si al menos uno de los dos menciona los números de fila, se gestionan los números de fila para todas las filas.

Si lo observa con más detalle, verá que estos números no son de tipo numérico, sino cadenas de caracteres, pues se trata de nombres, es decir, atributos de las filas. Para comprender mejor esto, hagamos algunos pequeños experimentos.

```
names(c)
```

```
[1] "v1"      "v2"      "v3"      "clase"
```

¡No hay columna **row.names**!

```
c[99, ]
```

```
v1 v2 v3 clase
NA NA NA NA    <NA>
```

Por otra parte, no hay fila **99**, y es normal, pues hemos visto más arriba que este pequeño **data.frame** solo contiene 10 filas.

Pero entonces ¿cómo acceder a los datos mediante este **row.names**?

```
c["99", ]
```

```
v1      v2      v3 clase
99  0.8765281 -1.554613 2.063703     3
```

Bastaría con invocarlo como atributo de la fila, por lo tanto como una cadena de caracteres.

Este **row.names** no afecta a sus manipulaciones de fórmulas, pues no forma parte del **data.frame** propiamente dicho. No se tendrá jamás en cuenta accidentalmente en ninguna función o módulo predictivo.

Si desea desembarazarse de los **row.names** de un **data.frame**, es bastante simple: utilice la siguiente instrucción.

```
row.names(c) <- NULL
```

Para combinar filas, hemos utilizado el número de fila, pero no los valores de los campos de las distintas filas. En la práctica, querrá sin duda procesar o extraer las filas en función de los valores de sus campos (por ejemplo: crear un **data.frame** con los clientes de un país concreto que hayan comprado productos domésticos, partiendo de un **data.frame** con todas las compras de todos los clientes en todos los países...).

Filtrar filas con subset()

Nuestra primera prueba consiste en extraer las filas correspondientes a una clase concreta.

```
c <- subset(df_, clase == "1") # sobre el valor de un factor  
c  
nrow(c)
```

```
v1           v2           v3 clase  
26 -1.28070320 -0.7846984  0.08931045      1  
27 -1.69068657 -0.8296075 -1.48580161      1  
.../...  
[1] 25
```

Existen 25 observaciones en la clase 1.

Como conserva el **row.names** original (**26, 27...**), más tarde podrá establecer la relación con los datos originales (es decir, para los informáticos, una unión basándose en el **row.names**).

Nuestra segunda prueba filtra aquellos datos para los que el campo **v1** es negativo o nulo.

```
c <- subset(df_, v1 <= 0) # sobre el contenido de una variable  
# numérica  
c  
nrow(c)
```

```
v1           v2           v3 clase  
4  -0.16007145  0.4988763  0.16257231      0  
26 -1.28070320 -0.7846984  0.08931045      1  
.../...  
[1] 49
```

Nuestra tercera prueba combina criterios sobre varias columnas.

```
c <- subset(df_, v1 <= 0 &  
          v2 <= 0 &  
          clase != 2) # o más complejo...  
c  
nrow(c)
```

```
v1           v2           v3 clase  
26 -1.28070320 -0.7846984  0.08931045      1
```

```
27 -1.69068657 -0.8296075 -1.48580161      1  
.../...  
[1] 23
```

Cuando se realizan dichas selecciones, uno se siente tentado de estudiar los elementos en función de su pertenencia a las clases conocidas.

c. Aplicación: comparación de elementos de clases y Khi-2

De momento, no hemos obtenido demasiadas estadísticas con R.

Contar las filas nos permite introducir una técnica estadística básica, fácil de utilizar y muy importante.

La identificación de las filas correspondientes a un **criterio** determinado, seguida de la **comparación** entre el reparto de estas **filas por clase** con el reparto del conjunto de **filas por clase para el conjunto** de las filas, le permite verificar si el criterio en cuestión está o no vinculado con la clase.

Imaginemos que tuviéramos un juego de datos con el color preferido de cada miembro de una población (clase: verde, azul, rojo, amarillo...). Por otro lado, dispone de los ingresos familiares anuales. Espera que este reparto del color preferido sea equivalente entre los individuos que dispongan de unos ingresos inferiores a 100 000 o que no dispongan de ingresos.

Si no fuera el caso, de una manera significativa (aquí radica el problema), podría emitir la hipótesis de una dependencia entre el nivel de ingresos anual familiar y las preferencias de los individuos en términos de color.

Para obtener el color neto, nuestro método consistiría en seleccionar las personas cuyos ingresos estén por debajo de 100 000 y por encima de 100 000, y a continuación comparar los elementos de la clase color entre ambas poblaciones.

Apliquemos el mismo método con nuestro segundo ejemplo de **subset()** del párrafo anterior. Vamos a extraer las filas en las que **v1** es negativo o nulo y aquellas en las que **v1** es positivo con el ánimo de determinar si esto depende del valor de la clase.

```
c_neg      <- subset(df_, v1 <= 0)  
c_pos      <- subset(df_, v1 > 0)
```

Es momento de utilizar la instrucción **table** que nos devuelve los individuos por clase.

```
v1_negativo <- table(c_neg$clase)  
v1_negativo
```

```
0  1  2  3  
1 24 24  0
```

```
v1_positivo <- table(c_pos$clase)  
v1_positivo
```

```
0  1  2  3  
24 1  1 25
```

La cuestión es saber si la diferencia de reparto en las clases resulta significativa (nos hemos hecho una idea, ipero hagamos como si aquí no fuera evidente!).

Para ello, en primer lugar vamos a fabricar una tabla con los dos subconjuntos de población utilizando `rbind()` y `as.table()`.

```
m <- as.table(rbind(v1_negativo,
                      v1_positivo)
                    )
m
```

```
      0   1   2   3
v1_negativo 1 24 24  0
v1_positivo 24  1   1  25
```

Ahora podemos utilizar la prueba de χ^2 (de hecho X^2) como prueba de independencia, cuyo objetivo es decirnos si podemos o no rechazar la hipótesis (llamada hipótesis nula) que predice que estas variables son independientes.

```
(Xsq <- chisq.test(m))
```

```
Pearson's Chi-squared test
```

```
data: m
X-squared = 88.4754, df = 3, p_value < 2.2e-16
```

Si **p_value** es inferior a 0.01, se rechaza la hipótesis nula que predice que estas variables son independientes. Observe la terminología, inadie dice que estas variables sean dependientes!

Para una discusión acerca de **p_value**, consulte el siguiente capítulo de este libro.

► El test χ^2 define la posibilidad de ver cómo la distribución constatada se corresponde («fit») con la distribución teórica que habríamos podido imaginar (distribución Normal, distribución de Poisson... o aquí una distribución uniforme en el caso del test de dependencia).

Para convencerse de la diferencia entre los elementos repartidos de manera uniforme y los elementos observados, podemos extraer los valores que habrían sido «normales».

```
Xsq$expected
```

```
      0     1     2     3
v1_negativo 12.25 12.25 12.25 12.25
v1_positivo 12.75 12.75 12.75 12.75
```

En nuestro caso, los elementos de las clases del conjunto del **data.frame** eran iguales, el resultado anterior parece trivial. Pero si hubiéramos tenido una población padre con clases de elementos desiguales, el reparto

teórico no habría resultado tan evidente, de ahí el interés de poder extraer los elementos teóricos.

Llegados a este punto, somos capaces de seleccionar las columnas y las filas, realizar cálculos de medias y de sumas sobre las filas o las columnas y realizar cálculos globales sobre los niveles de ruptura del **data.frame** (es decir, obtener medias, sumas, búsquedas de min/max por clase mediante **aggregate()**). Pero tenemos que ser capaces de generar nuevas columnas (nuevas features) a partir de las columnas precedentes, por ejemplo para adaptar nuestras variables explicativas a los procesamientos algorítmicos que intentamos aplicar para fabricar modelos predictivos o modelos de clustering.

d. Creación de columnas calculadas

Cálculos por columna con within()

Es una instrucción muy sencilla de dominar y que presenta una gran eficacia; sin embargo, no se utiliza demasiado, puesto que sugiere, sin motivo, que hablamos de un bucle clásico.

```
## transformar o agregar columnas por cálculo mediante within ##
## flexible y potente ##

c <- within(df_,
{
  v4 <- v1+v2+v3
})
c
names(c)
```

```
v1          v2          v3      clase      v4
1  0.54946252 1.586331941 2.10823617      0  4.24403063
2  1.40579522 1.597558143 1.13792558      0  4.14127894
...
[1] "v1"       "v2"       "v3"       "clase"    "v4"
```

Ha creado una columna **v4** que contiene fila a fila el resultado de sus cálculos.

Esta instrucción soporta transformaciones muy complejas; en efecto, el código de esta transformación es tan rico como usted quiera, como permite adivinar el hecho de que este código está situado entre dos llaves.

Puede, sin riesgo, codificar su bucle imaginando que el código situado entre las dos llaves es una función anónima. Las variables de esta función son los nombres de las columnas. Esta función devuelve sus valores mediante una o varias columnas nuevas.

Existe una restricción: no debe definir funciones o utilizar funciones anónimas dentro de este código (*closures*), pues se recrearían con cada fila y esto tendría un efecto catastrófico sobre el rendimiento. En la práctica, esta restricción no aporta ninguna limitación importante, basta con declarar sus funciones en el exterior del bloque **within()**, como en el siguiente ejemplo.

```
## demostración de la potencia de within() ##

# para el ejemplo
```

```

f <- function(x,y,z){round(log(x+y+10)/(abs(z)+1))}

c <- within(df_,                               # para cada fila hacer:
{
  vd <- abs(v2-v1)
  vr <- round((v1+v2)/vd) # se invoca a vd
                           # del mismo bucle
  var_global <- v1+v2+v3   # evita crear una columna
  vp <- f(v1,v2,var_global) # ;sin closure aquí!
  vd <- round(vd,2)
}
)

c

```

	v1	v2	v3	clase	vp	vr	vd
1	0.54946252	1.586331941	2.10823617	0	0	2	1.04
2	1.40579522	1.597558143	1.13792558	0	0	16	0.19
.../...							

Estudie los siguientes puntos:

- Se ha fabricado una columna **vd** que se utiliza en un cálculo realizado en la misma iteración para calcular **vr** (sobre la misma fila).
- Se ha aplicado una función de tres variables fila a fila y se han utilizado dos variables creadas en la misma iteración.
- El uso de una variable global ha permitido no tener que crear una columna inútil, dado que no se desea almacenar este resultado en el **data.frame**.
- Se han construido diversas columnas a la vez.

Cuando se encuentre con dificultades a la hora de diseñar la transformación de un **data.frame** mediante el uso de funciones muy compactas y muy vectorizadas, **piense enwithin()**, la navaja suiza de los cálculos sobre columnas. A continuación, si tiene dudas acerca del rendimiento, estudie el uso de técnicas más compactas y verifique atentamente si le aportan una mejora real en términos de rendimiento, ino siempre ocurre así!

Ahora vamos a estudiar **transform()**, que es la función vectorizada y compacta de referencia para la transformación de las columnas de **data.frame**.

Cálculos por columna con transform()

En primer lugar, puede comprobar que todo lo que vamos a realizar con **transform()** puede hacerse con **within()**, aunque a la inversa no ocurre así. El interés de **transform()** reside en su compacidad, su vectorización y su legibilidad en el caso de usos sencillos.

Las siguientes sintaxis producen los resultados esperados y crean las columnas correspondientes sin problema alguno.

```

## Tranformar o agregar una columna con un cálculo mediante transform ##

c <- transform(df_, v1 = v1 + 10)      # transforma una columna
c

```

```
c <- transform(df_, vs = v1 + log(v2+10), # dos columnas
              vd = v1 + v2 +10)
c
```

Pero observe que no puede utilizar una columna creada nueva en el cálculo que permite calcular otra columna. Por otro lado, observe que todos los cálculos crean columnas.

Utilizando conjuntamente **transform()** y **sapply()**, puede aplicar una función anónima de una sola variable, con simplicidad y elegancia, pero sobre una única columna (aquí **v1**).

```
# sapply anónima
c <- transform(df_, vs = sapply(v1,function(x){x+100}))
c
```

Utilizando conjuntamente **transform()** e **ifelse()**, puede fabricar nuevas columnas cuyo cálculo dependa del resultado de una comprobación entre varias columnas. Esta técnica debe recordarse, pues es muy potente y legible.

```
# comprobación y selección
f <- function(x){round(x) %% 3}           # para el ejemplo

c <- transform(df_, vs = ifelse(f(v1)>f(v3),v1-v2,v3-v2))
c
```

	v1	v2	v3	clase	vs
1	0.54946252	1.586331941	2.10823617	0	0.52190423
2	1.40579522	1.597558143	1.13792558	0	-0.45963257
.../...					

Se ha obtenido una columna **vs**, que es resultado de un cálculo según el resultado de una comprobación compleja realizada sobre otras columnas.

Contar con transform(), ifelse() y aggregate()

El uso de **ifelse()** nos ofrece una **técnica para contar eficaz y fácil de depurar**. La idea es la siguiente: basta con fabricar una columna de 0 y de 1, tal que el valor de 1 se corresponda con el hecho de que una comprobación sea verdadera. Contando el número de 1, es decir, sumándolos, podrá contar el número de casos en que la comprobación es verdadera.

Como la columna de 1 y de 0 persiste, puede reutilizarla en otra comprobación sin tener que volver a consumir potencia de máquina y puede verificar manualmente que su prueba devuelva los resultados esperados, lo que simplifica la construcción de los programas.

Por nuestra parte, y para construir un código muy fiable, no dudamos en descomponer una comprobación compleja en comprobaciones intermedias distribuidas en varias columnas que agregaremos mediante una comprobación final.

```
# contar de forma fiable
# utilizando una función auxiliar
```

```

test <- function(clase,v1,v2){(clase ==1)&(v1>v2)}

c <- transform(df_,t = ifelse(test(clase,v1,v2),1,0))

c
sum(c$t)                                # cuenta

```

```

          v1          v2          v3 clase t
1  0.54946252  1.586331941  2.10823617    0  0
.../...
24  0.94318256  0.818294562 -0.48398969    0  0
25  1.29638308  1.223568235  0.80598326    0  0
26 -1.28070320 -0.784698436  0.08931045    1  0
27 -1.69068657 -0.829607455 -1.48580161    1  0
28 -0.58301720 -0.537336047 -2.14113679    1  0
29 -0.67199372 -1.100258099 -2.03583831    1  1
.../...
[1] 11

```

Obtenemos 11 filas para las que la comprobación entre las columnas es verdadera. Para verificar que esta prueba se ha realizado conforme a nuestras expectativas, podemos recorrer las filas del **data.frame** y observar manualmente si **t** posee el valor esperado. Por ejemplo, en la fila **29** se obtiene **clase==1**, **v1** superior a **v2** y el valor de **t** igual a 1.

Esta manera de trabajar nos abre otra perspectiva, podemos considerar esta columna de «1» y de «0» como una nueva feature, una clase sobre la que podemos realizar procesamientos.

Esto nos lleva de manera natural a utilizar **aggregate()** para contabilizar diversos agregados sobre las distintas variables en cada nivel de ruptura conjunto o no de nuestra nueva clase **t** y de nuestra vieja clase **clase**.

```

a <- aggregate(.~ t, data =c, mean)
a[,-5]                                # drop col 5

```

```

      t          v1          v2          v3
1 0  0.1298955  0.2849295  0.05931562
2 1 -0.6657712 -1.1240827 -0.76172084

```

En esta tabla, podemos leer por ejemplo que: *la media de la variable v3 sobre las observaciones que pertenecen a la clase 1 y en las que la feature v1 es estrictamente superior a la feature v2 es igual a -0.76*.

Para obtener agregados teniendo en cuenta a la vez **t** y **clase**, hay que utilizar la sintaxis «fórmula» dentro de la función de agregación.

```

# por clase y vs
a <- aggregate(.~ t+clase, data =c, mean)
a <- a[,-5]
a

```

```

t clase      v1      v2
1 0          0 1.0071463 1.1175014
2 0          1 -1.3514994 -0.6844693
3 1          1 -0.6657712 -1.1240827
4 0          2 -0.9960628 1.2509849
5 0          3 1.2081841 -0.9708346

```

Para una mejor legibilidad, el micro **data.frame** resultante habría ganado si se hubiera ordenado basándose en sus filas y columnas.

e. Ordenar un **data.frame** mediante **order()**

La instrucción **order()**

La instrucción **order()** devuelve una secuencia que se corresponde con la ordenación de lo que queremos ordenar.

```
order(c("a", "b", "d", "c"))
```

```
[1] 1 2 4 3
```

Observe el 4 y el 3 que se corresponden con el hecho de que hay que invertir «d» y «c» para tenerlas en orden alfabético.

Podemos pedirle una ordenación decreciente.

```
order(c("a", "b", "d", "c"), decreasing = TRUE)
```

```
[1] 3 4 2 1
```

También es posible seleccionar cómo clasificar los valores desconocidos.

```
order(c("a", "b", NA, "d", NA, "c"), na.last = FALSE)
```

```
[1] 3 5 1 2 6 4
```

Aquí, los valores **NA**, que tenían **como rango 3 y 5**, se clasifican al principio.

Ordenar filas según una o varias columnas

Volvamos ahora a nuestro pequeño **data.frame** y clasifiquemos sus filas según las columnas **t** y **clase**.

```

# ordenar según columnas
b <- a[order(a$t, a$clase),]
b

```

```

t clase      v1      v2
1 0      0 1.0071463 1.1175014
2 0      1 -1.3514994 -0.6844693
4 0      2 -0.9960628 1.2509849
5 0      3 1.2081841 -0.9708346
3 1      1 -0.6657712 -1.1240827

```

Ahora el **data.frame** está clasificado por valor de **t** y, después, para un mismo valor de **t**, por valor de **clase**.

Ordenar columnas según sus nombres

Conceptualmente inútil, pero en ocasiones práctico para entender mejor visual o prácticamente un **data.frame**, la ordenación alfabética por nombre de columna puede resultarle útil. En ocasiones se seleccionan los nombres de las columnas de manera que será fácil organizarlas mediante una ordenación.

```

# por nombre de columna
d <- b[,order(names(b),decreasing = TRUE)]
d

```

```

v2      v1 t clase
1 1.1175014 1.0071463 0      0
2 -0.6844693 -1.3514994 0      1
4 1.2509849 -0.9960628 0      2
5 -0.9708346 1.2081841 0      3
3 -1.1240827 -0.6657712 1      1

```

Si el resultado no le conviene, y quiere por ejemplo disponer la columna **v1** antes que la columna **v2** sin alterar la disposición de las demás columnas, siempre puede hacerlo de forma manual fabricando su propia secuencia...

```

d <- d[,c(2,1,3,4)]          # manualmente
d

```

```

v1      v2 t clase
1 1.0071463 1.1175014 0      0
2 -1.3514994 -0.6844693 0      1
4 -0.9960628 1.2509849 0      2
5 1.2081841 -0.9708346 0      3
3 -0.6657712 -1.1240827 1      1

```

Aquí hemos intercambiado los rangos de las columnas 1 y 2, que se corresponden con las columnas **v1** y **v2**.

A continuación, vamos a manipular los datos desde la óptica del análisis de datos.

4. Análisis visual de los datos

a. Visualización simple de los datos

Vamos a visualizar los datos numéricos de nuestro **data.frame** de prueba mediante un bucle que va a recorrer

todas las features desde **v1** hasta **v12** (en este **data.frame**, la feature **13** -la última- se llama **clase** y es un **factor**).

```
n_col <- ncol(df)-1          # la última columna no es
                                # numérica, es la clase

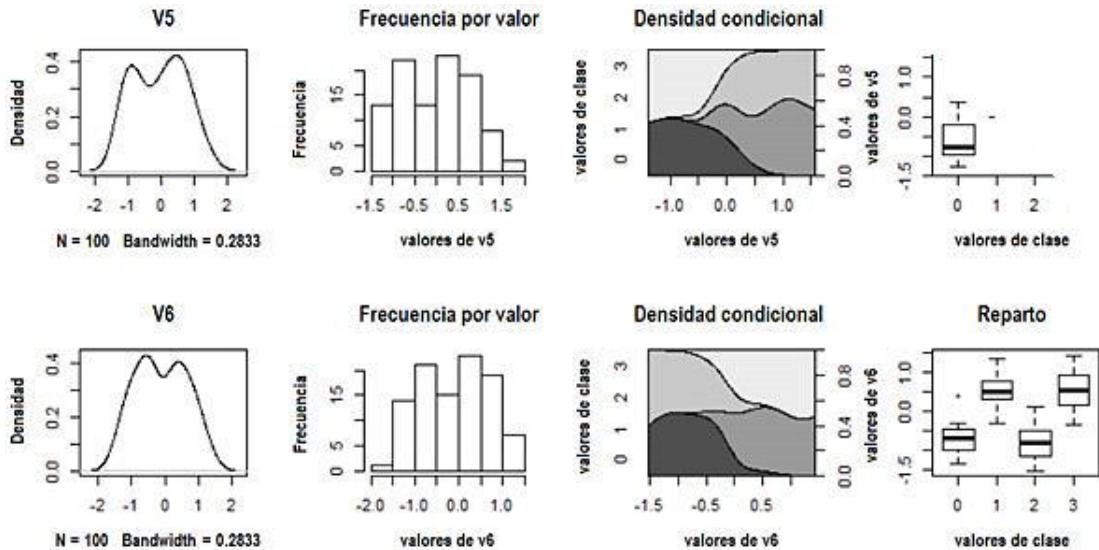
op <- par(mfrow = c(2,4))      # cada gráfico
                                # contiene 2 líneas
                                # de 3 gráficos

for (i in 1:n_col){
  df_i <- df[,i]
  nombre <- names(df)[i]
  plot(density(df_i),main = toupper(nombre))
  hist(df_i, main = "Frecuencia por valor",
        xlab = paste("valores de",nombre))
  cdplot(df_i,df$clase,main = "Densidad condicional",
         xlab = paste("valores de",nombre),
         ylab = "valores de clase"
    )
  plot(df$clase,df_i, xlab = "valores de clase",
        ylab = paste("valores de",nombre),
        main ="Reparto")
}
rm(df_i)  # limpiar el data.frame temporal
par(op)          # IMPORTANTE restaura los
                  # parámetros de presentación
```

Este código funciona de la siguiente manera:

- Se inicializa un parámetro gráfico importante, que es el número de gráficos y su reparto en una agrupación de gráficos, y se almacenan los parámetros que existían antes de realizar esta transformación para poder restablecerlos una vez terminado el trabajo.
- Se almacena el número de columnas sobre las que se desea iterar.
- En el bucle, se almacena la columna en curso y su nombre.
- Se fabrican cuatro gráficos sencillos utilizando la información almacenada.
- Tras ejecutar el bucle, se limpia el **data.frame** temporal y se restablecen los parámetros gráficos originales.

He aquí uno de los gráficos obtenidos.



Información sobre las variables **v5** y **v6** y reparto de sus valores según las clases

Las dos distribuciones parecen bimodales (dos cimas). El reparto en función de las clases es muy selectivo, se percibe visualmente que las features **v5** o **v6** por sí solas no serían sin duda buenos «predictores» de clase, aunque su conjunto sí podría serlo viendo que los diagramas de caja no se pisan (es una configuración llamada «XOR»).

Esto se confirma, por otro lado, observando que las densidades condicionales poseen un aspecto muy discriminante (densidad de los elementos de las clases para un valor determinado de una variable). Para convencerse, imagínese por ejemplo una línea vertical en la posición **v5==1** del gráfico correspondiente; comprobará que solo las clases **1** y **2** están representadas (puede fijarse en que cuanto más clara es la superficie, menor es el número de la clase).

b. Visualización de variables numéricas 2 a 2 con mención de las clases

Programa de visualización

Con la misma idea, el siguiente código permite fabricar las representaciones de las distintas variables dos a dos dibujando las observaciones de las distintas clases con colores y símbolos diferentes.

```
## aspecto de la muestra scatterplot v X v
## n_col <- ncol(df)-1
## op <- par(mfrow = c(2,2))

clase_ <- df$clase
for (i in 1:n_col){
  for (j in 1:n_col) {
    if (i<j ){
      df_i <- df[,i]
      df_j <- df[,j]
      df_ijc <- data.frame(df_i,df_j,clase_) # data.frame
                                         # temporal
      names(df_ijc)<-c("eje_1","eje_2","clase")
      with(df_ijc,
            # para este data.frame hacer
            plot(eje_1,
```

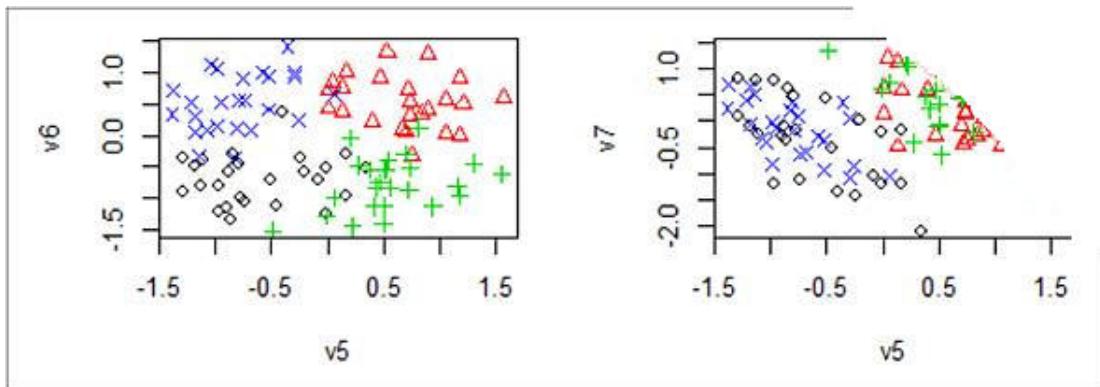
```

        eje_2,
        col=clase,           # color depende de clase
        pch=as.numeric(clase), # símbolo depende de clase
        xlab = names(df)[i],   # verdaderos nombres
        # de los ejes
        ylab = names(df)[j]
      )))
    }
  }
}

rm(df_ijc) # limpia data.frame temporal
rm(df_i)   # limpia data.frame temporal
rm(df_j)   # limpia data.frame temporal
rm(clase_)# limpia data.frame temporal
par(op)

```

He aquí dos de los gráficos obtenidos (el bucle sobre **i** tenía 12 iteraciones, el bucle sobre **j** también, pero solo se han fabricado los gráficos para **i < j**, con el objetivo de no tener duplicados y no tener los gráficos **vi X vi**, que no tendría ningún sentido. Se han fabricado por lo tanto $(12*11)/2$ gráficos, es decir, **66** gráficos **vi X vj**).



*Visualización de las observaciones sobre **v5 X v6** y **v5 X v7**, con sus clases*

Con el siguiente significado:

Significado de los símbolos	
clase 3	×
clase 2	+
clase 1	^
clase 0	○

Leyenda

Nuestras conclusiones anteriores parecen confirmarse, la representación **v5** por **v6** separa bien las cuatro clases. Por su parte, la representación **v5 X v7** separa la mezcla de clase "0" Union "3" de la mezcla de clase "1" Union "2".

En el código utilizado, tenemos pocas novedades, salvo los parámetros del **plot()**, donde usamos el valor de clase para definir el color (**col**) y el símbolo (**pch**) que la representará. Esta forma de trabajar es bastante habitual y conviene memorizarla.

Sin duda, habrá observado que el código anterior no contiene el código correspondiente a la visualización de la leyenda.

Para construir la leyenda

La manera de construir una leyenda para un único gráfico es otra actividad que vamos a estudiar a continuación, pues existe una opción dedicada de la función **plot** (la opción **legend**). Pero aquí se quiere fabricar una leyenda para todos los gráficos, que nos diga qué significan los símbolos. El estudio de este código un poco «simplista» le permitirá descubrir diversas opciones gráficas que se utilizan muy a menudo.

```
## creación leyenda - visualización de símbolos y colores ##

lib <- c("clase 0",
       "clase 1",
       "clase 2",
       "clase 3")
clase <- as.factor(0:3)
v      <- c(0,0,0,0)
w      <- 0:3
plot(v,
      w,
      main = "Significado de los símbolos",
      xlab = "", # sin etiqueta en el eje x
      ylab = "", # sin etiqueta en el eje y
      xaxp = c(0, 1,1), # eje x de 0 a 1 tick 1
      xaxt = "n", # sin eje x
      yaxt = "n", # sin eje y
      bty = "n", # sin marco
      col=clase,           # color depende de clase
      pch=as.numeric(clase)) # símbolo depende de clase
# agregamos lib (con posición y tamaño)
text(v-0.43,w+0.04,lib,cex = 0.8)
```

Observará:

- Que se agrega un título mediante el parámetro **main**.
- La supresión de las etiquetas de los ejes inicializándolas a **""**.
- El juego sobre la escala de las x, mediante **xaxp**.
- La supresión de los ejes y del marco configurando las opciones a **"n"**.
- La selección de colores y de símbolos apoyándose en los valores de clase.
- La agregación de las etiquetas de punto mediante la opción **text**.

Nos gustaría obtener un gráfico sintético que nos permita expresar ciertos aspectos de la naturaleza de la relación entre nuestras features, la más clásica es un gráfico que expresa la correlación entre features.

c. Correlación entre variables numéricas

Tendremos la ocasión de estudiar la noción de coeficiente de correlación más adelante en el libro. De momento, tengamos en mente que esto proporciona cierta idea del hecho de que una variable v_i pueda aproximarse como una combinación lineal de otra variable a través de una relación $v_i \leftarrow a \cdot v_j + b$. Cuanto más cercano a 1 sea el valor absoluto del coeficiente de correlación, mejor será la aproximación. El signo del coeficiente de correlación es el mismo que el signo de a , de modo que si es negativo esto quiere decir que cuanto más aumente la variable v_j más disminuirá la variable v_i y viceversa.

A menudo se utiliza la representación que vamos a construir a continuación para identificar rápidamente dependencias evidentes entre variables (evidentes en cuanto a lineales, aunque pueden existir dependencias reales mucho más difíciles de identificar).

Se utiliza un paquete especializado, el paquete «**corrplot**».

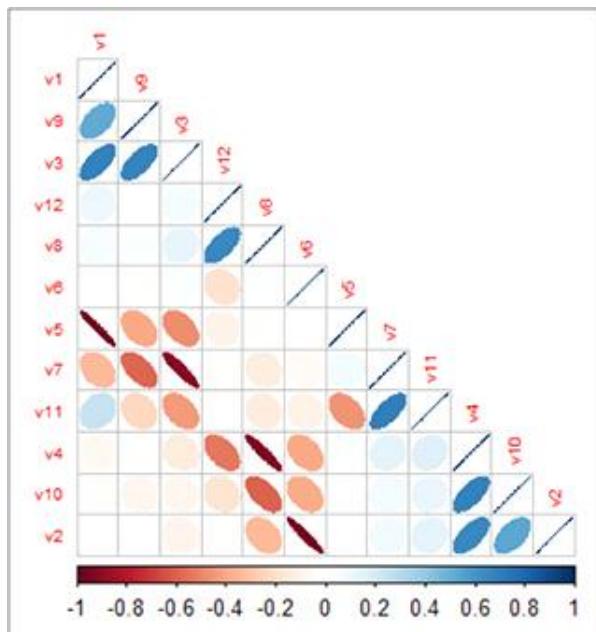
```
## correlaciones entre variables numéricas ##

if(require("corrplot")==FALSE) install.packages("corrplot")
require(corrplot)

num_all <- c(names(df)[1:12]) # las 12 primeras variables

cor <- cor(df[num_all])
corrplot(cor,
         title = "Correlación entre variables numéricas ",
         order = "AOE",
         method = "ellipse",
         type = "lower",
         tl.cex = 0.6 )
```

El resultado es muy sencillo de interpretar en términos de correlación.



Por ejemplo, **v7** y **v3** están fuertemente correladas mediante una correlación negativa.

Observamos que **v5** y **v6** no están correladas en absoluto, ni tampoco **v5** y **v7**; esto se veía bien en las nubes de puntos del párrafo anterior.

Las relaciones están lejos de ser siempre transitivas. Se confirma, por ejemplo:

- **v1** parece correlada a **v3**,
- **v3** parece correlada a **v7**,
- **v1** y **v7** no lo están.

Cuando encuentre una lista de variables muy correladas entre ellas, con una fuerte transitividad «circular» de esta correlación, puede más o menos considerarlas como redundantes y guardar solamente una. Se trata de un método brutal de eliminación de features que funciona en algunas ocasiones. En este caso, conserve la variable menos correlada a las variables de los demás grupos de variables redundantes y cuya distribución de frecuencia no suponga pérdidas, asimetrías o huecos.

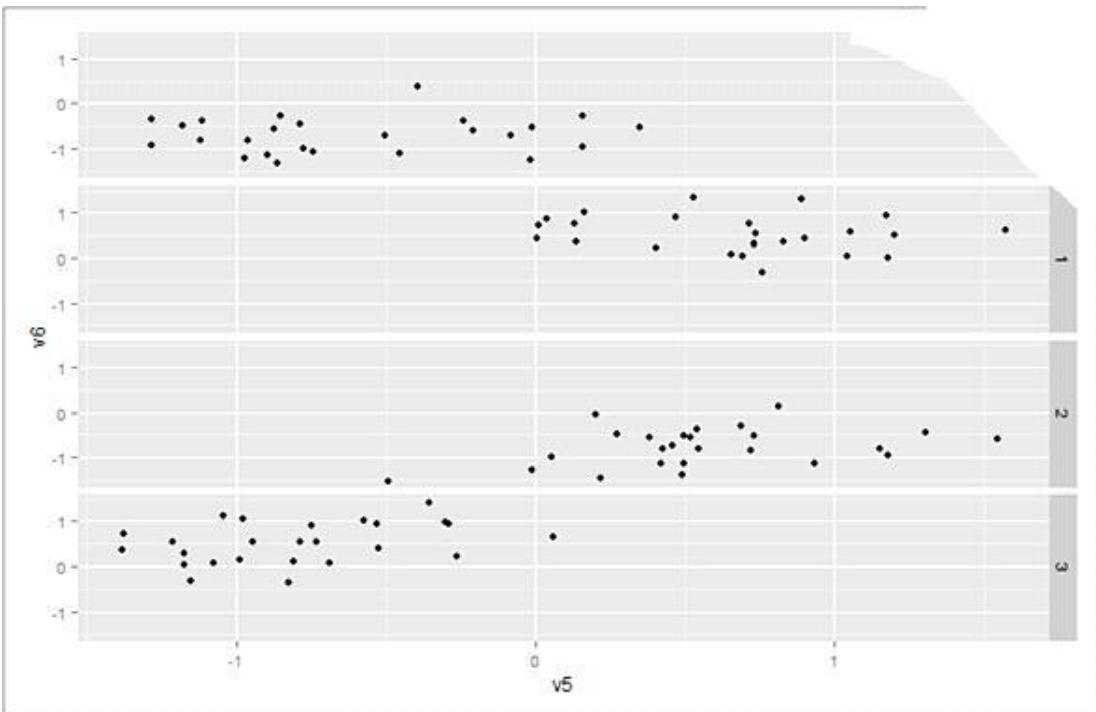
d. Separación por clase, **ggplot2**, **qplot**

Cuando se tienen muchas clases, la representación natural con diferentes símbolos sobre un cruce de variables se vuelve confusa rápidamente. La siguiente visualización es una alternativa eficaz. Nos permite, por otro lado, introducir un paquete gráfico muy útil y elegante, el paquete **ggplot2** y su función gráfica simplificada **qplot()**.

```
## separación de clases en función de v5 y v6
if(require("ggplot2")==FALSE) install.packages("ggplot2")
require("ggplot2") # librería gráfica moderna

qplot(v5, v6, data=df, facets= clase ~.)
```

En una sola línea (pues una vez instalado el paquete e invocado, no tiene que repetir las dos primeras líneas en su sesión de trabajo), obtiene un gráfico de calidad. La opción **facets** es la que permite dividir el gráfico por clase.



Scatterplot v5 x v6 separado por clase

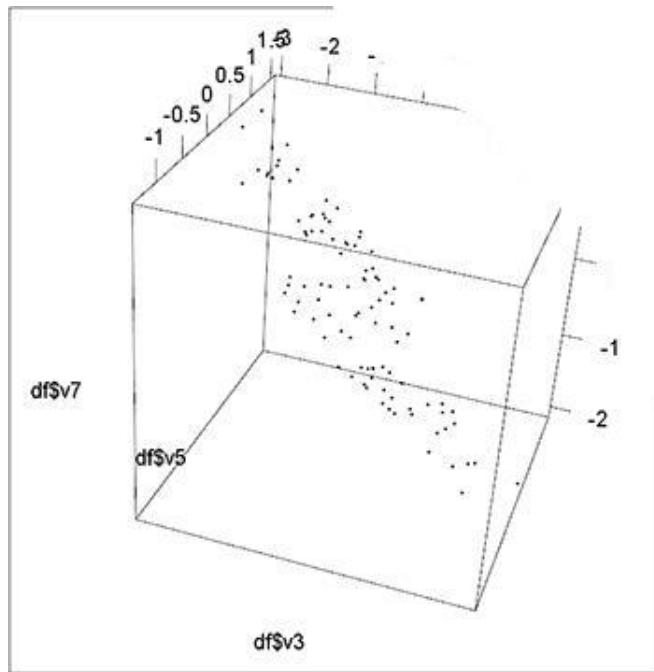
Hasta ahora hemos manipulado gráficos en dos dimensiones, sin visualizar jamás la interacción entre tres variables numéricas.

e. Visualización 3D, relación entre tres variables numéricas

Para visualizar la interacción entre tres variables numéricas, vamos a utilizar un paquete que le permitirá manipular con el ratón su gráfico (es decir, hacer girar el cubo interactivamente). Este paquete se llama **rgl** y sus funciones **open3d()** y **plot3d()** se utilizan una para inicializar el gráfico y la otra para representarlo.

```
## gráfico en 3D, manipulable con el ratón ##  
if(require("rgl")==FALSE) install.packages("rgl")  
require("rgl") # para el 3D  
  
open3d() # abre el canvas  
  
plot3d(df$v3,df$v5,df$v7) # puntos (v3,v5,v7) en 3D
```

El siguiente gráfico aparece en una ventana externa a RStudio (en ocasiones, hay que buscarla por detrás de las demás ventanas abiertas). Puede girar el cubo «agarrándolo» con el botón del ratón.



Visualización 3D de las relaciones entre v3, v5 y v7

También resulta cómodo visualizar varias variables con las clases correspondientes por pares en un único gráfico.

f. Gráficos por pares

Los gráficos por pares resultan muy útiles al inicio del análisis para desarrollar las primeras intuiciones acerca de la interacción entre los datos.

```
v <- c("v3", "v5", "v6", "v7", "v11", "v12", "clase")
df_ <- df[,v]

require(fBasics, quietly=TRUE) # para invocar a basicStats
                                # Estadísticas útiles
lapply( subset(df_, select = v3:v7), basicStats)

                                # gráfico con un color por clase de origen
plot(df_, col = df_$clase,
      pch = as.numeric(df_$clase))
```

Tras obtener algunas estadísticas elementales sobre los datos, este código produce un gráfico por pares.

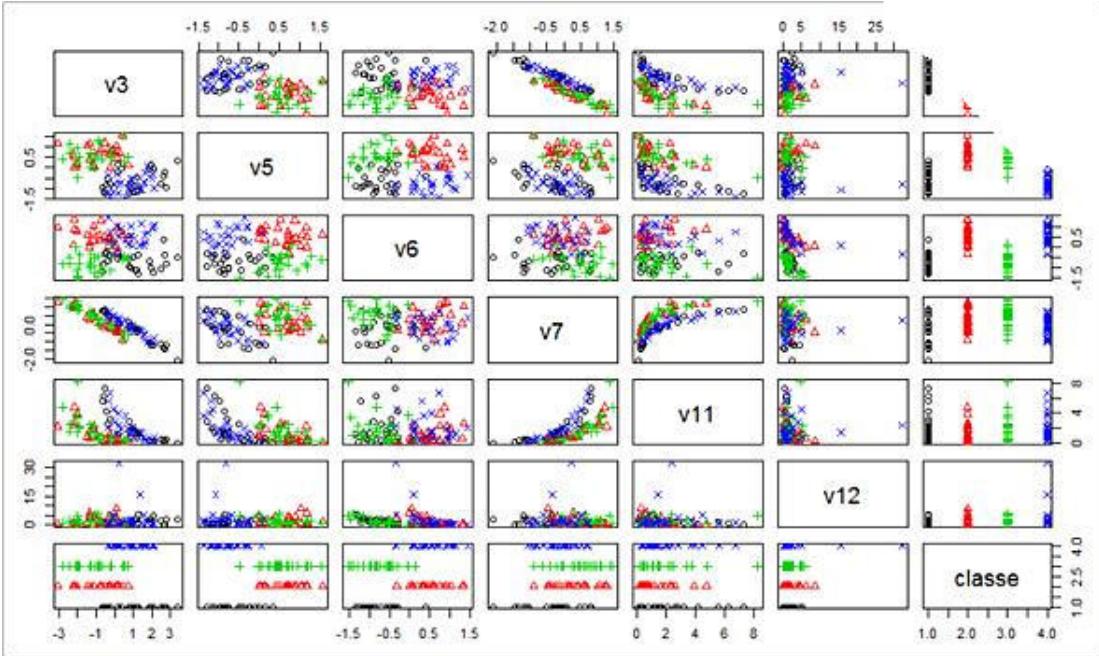


Gráfico por pares

Sobre dicho gráfico, encontramos lo que habíamos percibido anteriormente en cuanto a la calidad de separación de las clases del scatterplot **v5 X v6**. Se confirma también la presencia de puntos muy aislados en **v11 X v12**, estos puntos podrían ser excepciones (en inglés *outliers*) que habría que eliminar del conjunto de datos para evitar desarrollar sobre ellos un modelo sobredeterminado, es decir, un modelo que habría capturado accidentalmente el ruido de fondo de los datos y no la naturaleza profunda y subyacente de estos (*overfitting*).

g. Diagramas de caja y eliminación de outliers

Dibujemos los diagramas de caja y contemos los puntos que parecen extraños.

Los límites de las cajas en los diagramas de caja son los cuartiles Q1 (25 %) y Q3 (75 %), el trazo central se corresponde con la mediana (y no con la media!) y los bordes de las cajas están situados, en este caso, en los cuantiles 5 % y 95 %.

```
## diagramas de caja ##

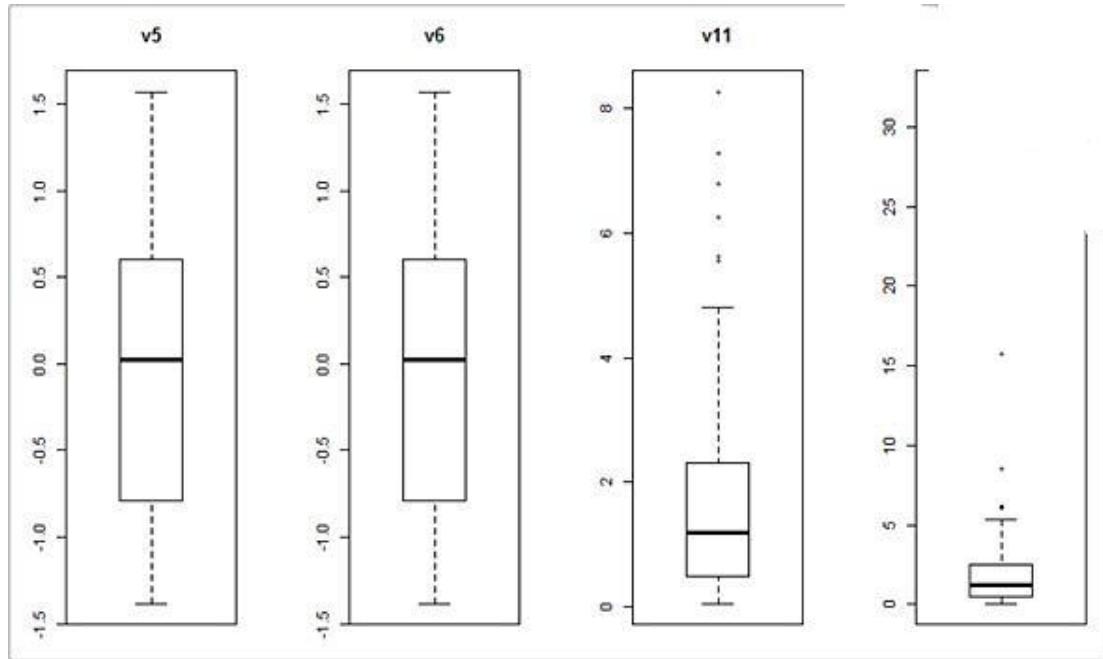
op <- par(mfrow=c(1,4)) # para poner las cuatro cajas sobre la misma página
boxplot(df$v5 , main="v5" , collapse=" ") # diagramas de caja
boxplot(df$v6 , main="v6" , collapse=" ")
boxplot(df$v11, main="v11", collapse=" ")
boxplot(df$v12, main="v12", collapse=" ")

length(boxplot.stats(df$v5 )$out) # número de puntos extraños
length(boxplot.stats(df$v6 )$out)
length(boxplot.stats(df$v11)$out)
length(boxplot.stats(df$v12)$out)
par(op)
```

```
[1] 0
[1] 0
```

```
[1] 6  
[1] 5
```

Imaginemos seis outliers determinados por **v11** y cinco determinados por **v12**. Observe que podrían ser los mismos.



Diagramas de caja

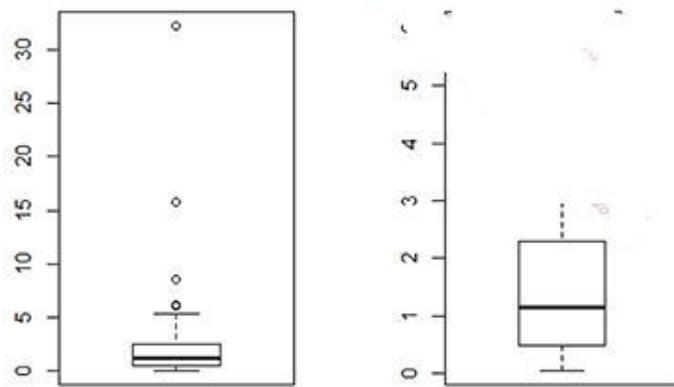
Vemos los potenciales outliers en **v11** y **v12**.

Eliminemos los outliers basándonos en **v12** y veamos el nuevo diagrama de caja correspondiente.

```
## eliminación de los puntos extraños ##  
  
library(dplyr)      # librería de manipulación de datos  
m <- min(boxplot.stats(df$v12)$out)  
df_bis <- filter(df, v12 <=m)    # función filter del paquete dplyr  
  
op <- par(mfrow=c(1,2))  # para tener las dos cajas en la misma página  
  
boxplot(df$v12,      main="v12 original", collapse=" ") #  
boxplot(df_bis$v12, main="v12 sin puntos extraños", collapse=" ")  
  
par(op)
```

Hemos introducido el uso del paquete **dplyr**, muy eficaz en manipulación de datos, y hemos utilizado la función **filter()** de este paquete. Esto no era indispensable, pues hemos visto más arriba otros métodos para filtrar un **data.frame**.

v12 original



v12 antes y después de eliminar outliers

A continuación vamos a dibujar el diagrama por pares, pero con el nuevo **data.frame** sin los **outliers**.

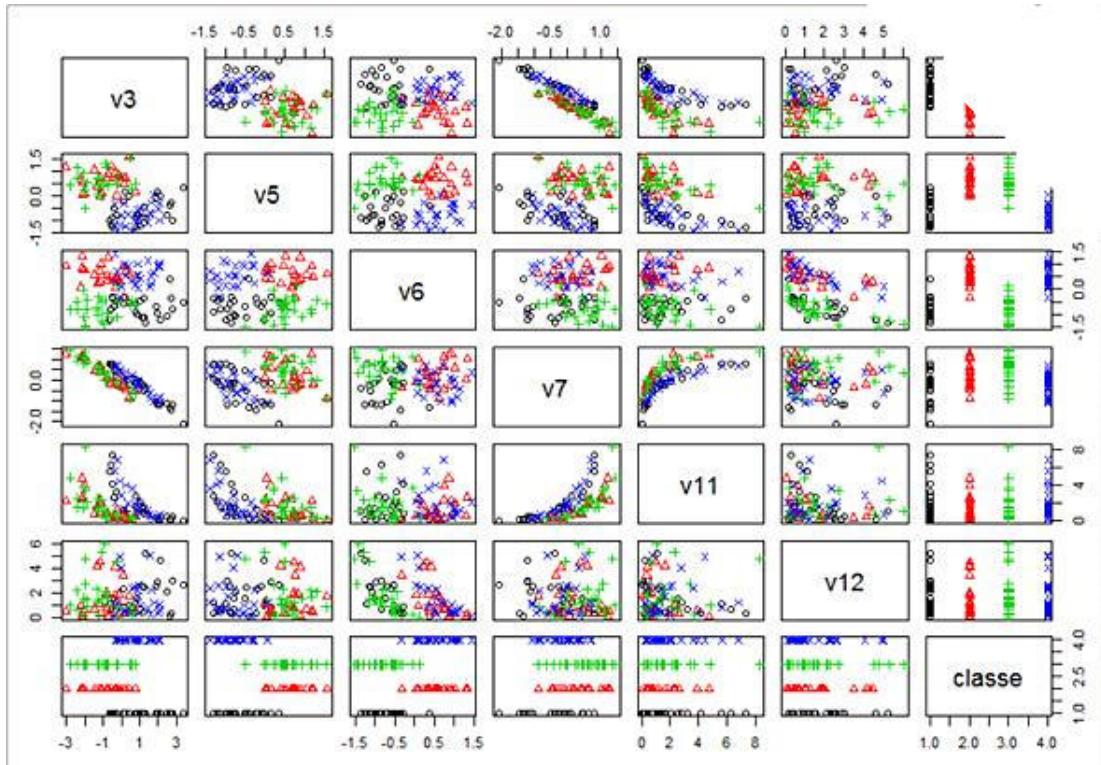


Gráfico por pares sin los outliers

En este diagrama, vemos ahora que estos outliers nos habían enmascarado ciertas concordancias entre **v11** o **v12** y **v3**, **v5**, **v6**, **v7**.

Ahora es momento de tratar de determinar un modelo predictivo sencillo a partir de nuestras observaciones. Como comprobamos visualmente zonas «cuadradas» de clases, podríamos pensar que la tecnología de árboles de decisión va a ser realmente eficaz. En efecto, imaginamos que esta va a utilizar las fronteras que visualizamos para construir su árbol de decisión.

h. Creación de un modelo por árbol de decisión

Creación del modelo

Sobre los datos limpiados de outliers, vamos a fabricar un modelo bajo el formato de un árbol de decisión y a comprobar su eficacia mediante lo que denominamos matriz de confusión. Los datos se separan en dos conjuntos, 60 % para entrenar el modelo, 40 % para comprobarlo. Queremos predecir la **clase** en función de las variables explicativas que nos parecen adecuadas: **v3, v5, v6, v7**.

```
library(rpart)          # árbol de decisión
library(caret)          # nos aporta un particionamiento elegante

# creación partición training/test
p           <- createDataPartition(y=df_$clase,
                                p= 60/100,
                                list=FALSE)
training   <- df_[p,]
test       <- df_[-p,]

# elección de las variables del dataset
X          <- c("v3", "v5", "v6", "v7")
y          <- "clase"          # nuestro objetivo

## construcción del árbol de decisión
d <- training[c(X, y)] # el data.frame de entrenamiento

modelo <- rpart(clase ~ . ,
                 data = d,
                 control=rpart.control(minsplit=1,cp=0)
                 ) # fitting

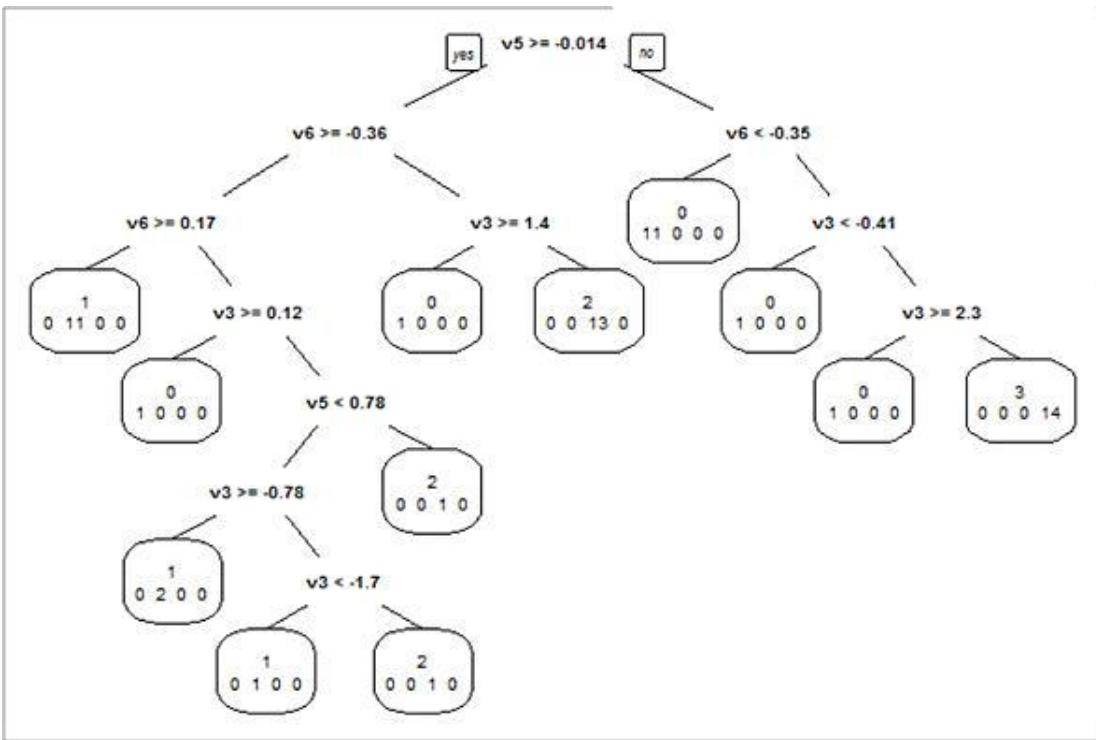
library(rpart.plot)      # para trazar el árbol
prp(modelo,extra=1)      # trazar el árbol

# predecir sobre el conjunto de test
predicciones <- predict(modelo, test, type="class")

print(confusionMatrix(predicciones,test$clase)) # verificar
                                                # mediante matriz
                                                # de confusión
```

Lectura del árbol

El árbol resultante es muy sencillo de leer y ayuda a comprender qué es un árbol de decisión.



Árbol de decisión para determinar los valores de clase

Veamos una rama en su conjunto: si **v5** es superior o igual a **-0.014** y **v6** estrictamente inferior a **-0.36** y **v3** estrictamente inferior a **1.4**, entonces la clase es **2**, que se corresponde con **12** casos del conjunto de entrenamiento. Observe que los *no* están siempre a la derecha.

Matriz de confusión y accuracy

Nuestro cálculo nos ha devuelto también lo que denominamos matriz de confusión, es decir, la tabla que cuenta los fallos y los aciertos para cada clase. Se denomina *accuracy* al porcentaje de éxito.

Confusion Matrix and Statistics					
		Reference			
		0	1	2	3
Prediction		0	1	2	3
0	10	0	1	0	
1	0	9	1	1	
2	0	0	8	0	
3	0	0	0	8	
Accuracy : 0.9211					

Tenemos una tasa de predicciones correctas (*accuracy*) de 92 %, que se calcula realizando la suma de la diagonal de la matriz de predicción dividida por el número total de elementos (suma de todos los términos). Tenemos una predicción a 0 en lugar de 2, una a 1 en lugar de 2 y una a 1 en lugar de 3. Este parece un modelo eficaz.

Ha realizado su primera predicción mediante una técnica de «machine learning» aplicada sobre un conjunto de entrenamiento, tras haber seleccionado features y eliminado outliers. A continuación se ha pronunciado acerca de la calidad de su modelo aplicándolo sobre un conjunto de test/validación. Se ha manejado con R para resolver un

problema de análisis de datos tal y como se da en la práctica en data sciences.

El objetivo del resto del libro será explorar los recovecos de este tipo de procedimiento.

Dominar los fundamentos

Ponerse en armonía con los datos

El error más habitual de las personas recién iniciadas en las data sciences resulta ser el uso directo, sin precaución y sin un análisis preliminar de los algoritmos a su disposición sobre conjuntos de datos incomprendidos.

A lo largo de todo el proceso de elaboración de su estudio y con más razón desde su primer contacto con un problema, el data scientist deberá interesarse por evaluar sus datos mediante diversas herramientas estadísticas y visualizarlos desde diversos puntos de vista. Este precio nos permite emitir hipótesis correctas que trataremos de validar o invalidar a lo largo del proceso.

- El lector interesado que quiera consultar las fuentes encontrará sin duda mucho interés en consultar la obra de Andrei Kolmogorov, a quien debemos numerosas contribuciones respecto a ciertos conceptos subyacentes a nuestra práctica dentro de las data sciences (formalización de la lógica intuicionista como parte del «cálculo sobre problemas», ley fuerte de los grandes números, axiomatización de las probabilidades mediante el lenguaje de las teorías de la medición, upsilon-entropía que nos permite cualificar los estimadores, aplicación del método de Poincaré al estudio de los equilibrios...).

La interpretación de las herramientas estadísticas requiere asimilar ciertas nociones fundacionales relativas a la probabilidad que vamos a abordar a continuación.

1. Algunas nociones fundacionales

a. Fenómeno aleatorio

Se encuentra frente a un **fenómeno aleatorio** si, cuando realiza una **experiencia** varias veces bajo las mismas condiciones, obtiene **resultados** diferentes (y, por tanto, de una cierta manera... imprevisibles).

- Observe que, en esta definición, el hecho de que un fenómeno se identifique como aleatorio no depende de la incapacidad del observador para predecirlo por completo, que le haría declarar de manera perentoria que el fenómeno es intrínsecamente azaroso.

Sin embargo, cuando se repite la experiencia un gran número de veces observamos que los resultados se reparten en función de una ley estable, es decir, con **frecuencias de aparición de los resultados que dependen del valor del resultado**. Por ejemplo, si observa las notas de selectividad de una gran cantidad de alumnos, hallará una cierta frecuencia de notas medias más elevada que la frecuencia de malas o buenas notas. Gracias a que disponemos de esta **ley de los grandes números**, podemos permitirnos trabajar matemáticamente para estudiar el fenómeno.

Un **evento, que se define como una parte de todos los resultados posibles**, es más probable en función de si recubre una gran parte de los resultados posibles.

Por ejemplo, el evento `obtener_una_nota_superior_a_10` tiene una probabilidad mayor que el evento `obtener_una_nota_superior_a_16`.

- Cuando el número de resultados posibles es infinito, el número de eventos posibles es también infinito y aparecen ciertos problemas matemáticos. En este caso, las matemáticas trabajan sobre un subconjunto del conjunto de posibles eventos, que denominan **familia**. Una familia se establece mediante la intersección y la unión contables de eventos (que podemos contar), es decir, intersecciones o uniones que pertenecen a la familia. Una familia incluye el evento vacío y el evento «todos los resultados posibles», el complementario (es decir, el contrario) de un evento de una familia forma parte de la familia. En este caso solo será posible definir una probabilidad sobre los miembros (eventos) de la familia.

b. Probabilidad, variable aleatoria y distribución

Probabilidad

Una **probabilidad es una aplicación** que parte de una familia de eventos a valores comprendidos entre 0 y 1 (incluidos) tal que:

- La probabilidad del evento que comprende todos los resultados posibles es igual a 1.
- La probabilidad de cualquier unión contable de eventos disjuntos de la familia (es decir, sin resultado común) es igual a la suma de las probabilidades de estos eventos.

Por ejemplo, la probabilidad de obtener una nota comprendida entre 0 y 10 vale 1, y la probabilidad de obtener una nota de entre 8 y 9 o entre 9 y 10 es igual a la probabilidad de obtener una nota entre 8 y 10.

Por otro lado, la probabilidad de un evento imposible, por ejemplo obtener una nota inferior a 5 y a la vez superior a 9, es, evidentemente, nula.

Variable aleatoria y distribución

Interesémonos ahora por la noción de variable aleatoria. Esta noción puede prestarse a confusión; en efecto, una **variable aleatoria no es una variable, sino una función** que parte de resultados experimentales de un espacio medible frente a un conjunto cualquiera y que nos permite caracterizar la experiencia.

Imaginemos que estuviéramos interesados en ciertos libros escogidos en una biblioteca y que la variable aleatoria **x** representara el número de páginas. Dispone de una función que, para cada resultado (es decir, cada libro escogido al azar), nos devuelve el número de páginas. Evidentemente, existen otras variables aleatorias, por ejemplo una variable **y**, que expresara en nuestro ejemplo si la tapa es dura o blanda.

Nos podemos dotar de una probabilidad **p_x** llamada **ley de la variable x** (o **distribución de x**, si nos referimos a la estadística) que proporcione una probabilidad para cada valor de **x** (aquí el número de páginas) y para cada evento propio de la familia de los posibles subconjuntos de valores de **x** (por ejemplo, la probabilidad de escoger un libro de entre 100 y 200 páginas). Es fácil saber que el aspecto de la distribución de **x** no tiene nada que ver con la correspondiente a la distribución de **y** (tapa dura o blanda).

Cálculos simples sobre las distribuciones

He aquí algunas propiedades acerca de las distribuciones que conviene conocer.

Tenemos: $p_{x \text{ e } y} + p_{x \text{ o } y} = p_x + p_y$

Si **x** e **y** son disjuntas, es decir, si se excluyen mutuamente, tenemos que: $p_{x \text{ e } y} = 0$ y, por tanto, $p_{x \text{ o } y} = p_x + p_y$.

Estas últimas líneas se comprenden fácilmente si lo vemos desde el punto de vista de los conjuntos, donde la expresión **x e y** significa la **intersección** de los eventos correspondientes, y donde la expresión **x o y** significa la **unión** de los eventos correspondientes.

El ejemplo del siguiente párrafo describe una forma de determinar $p_{x \text{ e } y}$ y le permitirá comprender mejor esta noción.

Condicionamiento y probabilidad condicional

Cuando se han fijado los valores de una variable aleatoria cuya distribución se conoce y si las variables son dependientes, entonces esta información condiciona la probabilidad de la otra variable.

Para simplificar las siguientes explicaciones, vamos a discretizar la variable aleatoria **x**, es decir, a transformarla en un número finito de valores. Aquí, podemos considerar que un libro de menos de 100 páginas es «pequeño», y «grande» en caso contrario.

Consideremos que en general, cuando el libro es «grande», en un 90 % de los casos se utiliza una tapa dura y en caso contrario se utiliza solamente en un 15 % de los casos. A continuación disponemos de una distribución de **probabilidad condicional** que podemos denotar como $p_{y|x}$ y que se enuncia de la siguiente manera: **p_de_y_sabiendo_x**.

Veamos con detalle el aspecto de esta distribución de probabilidad condicional. Expresemos que la probabilidad de que el libro tenga tapa dura **sabiendo que el libro es grande** es el 90 %:

```
py|x(tapa_dura , libro_grande ) = 90 % # además:  
py|x(tapa_blanda , libro_grande ) = 10 %  
py|x(tapa_dura , libro_pequeño ) = 15 %  
py|x(tapa_blanda , libro_pequeño ) = 85 %
```

Parece entonces que, si conocemos la distribución p_x de grandes y pequeños libros en la biblioteca, podríamos deducir la distribución de las tapas duras por tamaño de libro en la biblioteca. Imaginemos que tuviéramos un 80 % de libros grandes, entonces:

```
px(libro_grande ) = 80 % # y:  
px(libro_pequeño ) = 20 %
```

Esto nos permite calcular la distribución de cada tipo de libro, respecto a las dos variables aleatorias en conjunto y sobre todos los libros de la biblioteca: $p_{y|x}$. La probabilidad de que un libro tenga tapa dura y sea grande sería:

```
py y x(tapa_dura,libro_grande)=  
          py|x(tapa_dura,libro_grande) *  
px(libro_grande) =  
          90% * 80% = 72% = 0.72  
  
# además:  
  
py y x(tapa_blanda,libro_grande)=  
          py|x(tapa_blanda,libro_grande) *  
px(libro_grande) =  
          10% * 80% = 8 % = 0.08  
  
py y x(tapa_dura,libro_pequeño)=  
          py|x(tapa_dura,libro_pequeño) *  
px(libro_pequeño) =  
          15% * 20% = 3% = 0.03  
  
py y x(tapa_blanda,libro_pequeño)=  
          py|x(tapa_blanda,libro_pequeño) *  
px(libro_pequeño) =  
          85% * 20% = 17% = 0.17
```

- Se confirma que la suma de las p_{yyx} es igual a 1, lo que se corresponde efectivamente con el valor de la probabilidad cuando consideramos un evento que contiene todos los resultados posibles.

Se demuestra que podemos generalizar de la siguiente manera, para los valores aleatorios discretos (contables), pero también para las distribuciones sobre las variables aleatorias tomando sus valores en conjuntos no contables, como los números reales o los vectores de números reales (o incluso las funciones!):

$$p_{y|y|x} = p_{y|x} \cdot p_x = p_{x|y} \cdot p_y = p_{x \text{ e } y}$$

Esta fórmula es válida suponiendo distribuciones de x e y no nulas.

Con la misma idea tenemos, con tres variables aleatorias:

$$p_{x \text{ e } y \text{ e } z} = p_x \cdot p_{y|x} \cdot p_{z|(x \text{ e } y)}$$

Y así sucesivamente... Podemos generalizar a un número cualquiera de variables aleatorias.

Independencia

La noción de independencia entre variables es muy útil, y a menudo es la condición estricta para poder utilizar ciertos algoritmos. Percibimos intuitivamente que resulta más sencillo interpretar un fenómeno cuando tenemos la posibilidad de verlo vinculado a variables aleatorias independientes. La idea general es que disponer de cierta información sobre alguna de las variables independientes no da ninguna información sobre la distribución de la otra variable aleatoria.

Esto se traduce evidentemente en (suponiendo distribuciones no nulas):

$$p_{x|y} = p_x \text{ que resulta equivalente a:}$$

$$p_{y|x} = p_y \text{ que resulta equivalente a:}$$

$$p_{x \text{ e } y} = p_x \cdot p_y$$

- Preste atención: esta noción de independencia no debe confundirse con la noción de eventos disjuntos, es decir, que no comparten ningún valor de su resultado. Por ejemplo, imagine que en nuestra biblioteca los colores de los lomos de los libros dependen de la nacionalidad de los autores, que solo podrían ser españoles o franceses. El evento { rojo, amarillo, rojo } significaría que este libro tiene al menos un autor español y el evento { azul, blanco, rojo } significa que este libro tiene al menos un autor francés. Ambos eventos tienen una intersección no nula, es decir, el color { rojo }. Sin embargo, el hecho de identificar que un autor es francés mediante los colores { azul, blanco, rojo } no aporta ninguna información acerca del hecho de que exista o no otro autor del libro que sea español e identifiable por sus colores { rojo, amarillo, rojo } y viceversa. Las dos variables aleatorias son independientes y, sin embargo, no disjuntas!

c. Un poco de matemáticas: notaciones y definiciones útiles

Desde el comienzo del capítulo, hemos intentado ahorrar en notación para no sobrecargar la exposición, veamos ahora las notaciones y definiciones que podríamos utilizar.

Lista de notaciones muy básicas

La definición de un conjunto utiliza las llaves; un conjunto A que fuera el conjunto de números enteros pares podría escribirse así:

$$A = \{ \text{enteros pares} \}$$

$$= \{ n \in \mathbb{N} \mid n \text{ par} \}$$

la barra se lee «tales que»

$$= \{ 0, 2, 4, \dots \}$$

$$= \{ n \in \mathbb{N} \mid \exists k \in \mathbb{N}, (n = 2k) \}$$

Esto puede leerse «conjunto de los elementos del conjunto de números enteros naturales, que llamaremos aquí n , tales que existe un entero natural que llamaremos aquí k y que cumple que n es igual a 2 veces k ».

Para designar una experiencia aleatoria, se utiliza a menudo el símbolo \mathcal{E} .

El espacio de estados asociado a la experiencia (es decir, el conjunto de todos los resultados posibles de la experiencia aleatoria \mathcal{E}), se denota con omega en mayúscula: Ω .

Uno de los resultados de este conjunto se escribe (omega en minúscula): ω .

El hecho de que este resultado pertenezca a Ω se denotará como $\omega \in \Omega$.

Si Ω es contable, es posible escribir como un conjunto discreto los posibles resultados $\{\omega_1, \omega_2, \dots\}$ (aquí, hemos indicado de 1 a infinito, es decir \mathbb{N}^*): $\Omega = \{\omega_i\}_{i \in \mathbb{N}^*}$.

Si Ω es finito con un cardinal (es decir, un grupo) de n elementos, podemos escribirlo como conjunto discreto de los posibles resultados $\{\omega_1, \omega_2, \dots, \omega_n\}$: $\Omega = \{\omega_i\}_{i \leq n, i \in \mathbb{N}^*}$.

El conjunto de partes de Ω , es decir, todos los conjuntos que se pueden construir con elementos de Ω al que se añade el conjunto vacío \emptyset , se escribe a menudo $\mathcal{P}(\Omega)$.

Notaciones y elementos vinculados a la teoría de la medida

Detrás de la palabra «medida», comprendemos de manera intuitiva que abordamos una teoría que nos permitirá dominar la manera en la que comprender las cantidades (números reales positivos o nulos) que se corresponden a conjuntos de «cualquier cosa».

Hay que tener en mente que no todo es medible y, por lo tanto, la necesidad de concebir una teoría precisa de la medida se impone a los matemáticos. Los siguientes elementos introducen el vocabulario que permite leer textos manipulando esta noción de medida. No vemos razonable eludir estos conceptos, un poco abstractos, en nuestro libro dedicado a las data sciences, pues los data scientists manipulan probabilidades y estas probabilidades son, de hecho, medidas en el sentido matemático del término.

Empecemos con la definición de un término extraño en este contexto, el término «familia». Una familia \mathcal{A} sobre Ω es un conjunto de subelementos extraídos de partes de $\Omega : \mathcal{P}(\Omega)$, que incluye el conjunto vacío \emptyset , que es cerrada bajo complementos (el complementario en Ω de toda parte pertenece también a Ω), uniones e intersecciones contables (toda unión finita de conjuntos que pertenecen a la familia pertenece también a la familia). Como el complementario del conjunto vacío \emptyset es el propio Ω , de la regla anterior se deduce que toda familia comprende al menos los dos siguientes elementos de la familia trivial siguiente: $\{\emptyset, \Omega\}$.

Resulta útil saber que una unión o la intersección finita de familias de Ω es también una familia de Ω .

Definimos la noción de medida sobre estas familias. Este es un aspecto importante, pues significa que lo que no es una familia no es, sin duda, medible en el sentido de la teoría de la medida. En su práctica de data scientist, a menudo estará tentado de concebir diversas medidas o de probabilizar un conjunto. En ese caso deberá plantearse la cuestión de saber si el conjunto que quiere medir posee efectivamente las características de una familia, en caso contrario este conjunto es, a menudo, incompleto o es infinito y está mal definido.

Una medida μ sobre una familia \mathcal{A} es una función tal que $\mu(\emptyset) = 0$, y tal que la medida de una unión contable de conjuntos dos a dos disjuntos de \mathcal{A} (sin elementos comunes) es igual a la suma de las medidas de estos conjuntos.

Dicho de otro modo, cuando mida el vacío obtendrá 0 y las distintas medidas poseen una cierta coherencia cuando se las suma.

Sea \mathcal{A} una familia extraída de $\mathcal{P}(\Omega)$,

sea p una medida tal que $p(\Omega) = 1$,

p se denomina entonces **probabilidad** y los elementos de esta familia son **eventos** (conjuntos medibles por una probabilidad). El espacio medible se denomina (Ω, \mathcal{A}) y este espacio dotado de esta probabilidad se denomina **espacio probabilístico** (Ω, \mathcal{A}, p) .

Por construcción, la aplicación de una medida (y, por tanto, de una probabilidad), es tal que la medida (o probabilidad) de una unión contable de eventos dos a dos disjuntos de \mathcal{A} es igual a la suma de las medidas (o probabilidades) de estos eventos.

Observemos que una medida es un número real positivo. Como $p(\emptyset) = 0$ comprendemos mejor por qué una probabilidad posee un valor comprendido entre 0 y 1 incluidos, es decir $[0,1]$.

Cuando Ω es un conjunto finito, la familia naturalmente utilizada en probabilidad es $\mathcal{P}(\Omega)$.

Preste atención, esto no es cierto si la dimensión es infinita o para una familia más pequeña (a esto se debe, por otro lado, el hecho de que la noción de familia nos resulte tan útil para poder manipular probabilidades sobre conjuntos Ω infinitos).

Notaciones y definiciones útiles en probabilidad

Al comienzo de este capítulo, utilizamos la siguiente notación para el término probabilidad de una variable aleatoria $x : p_x$. Esta notación es muy agradable y muy compacta y resulta muy rápida de manipular cuando se desea comentar los cálculos sencillos y en contextos no ambiguos, como los que hemos abordado anteriormente. Pero esta notación no cubre todas nuestras necesidades, lo que hace por otro lado que los autores utilicen otra notación más flexible, más exacta, más compleja, pero más verbosa y que vamos a explorar aquí.

En los textos muy matemáticos, designar las variables aleatorias con letras mayúsculas puede ser un buen hábito. Esto recuerda que son funciones X,Y,Z... y las diferencias de las variables habituales x,y,z...

El símbolo utilizado en este capítulo para designar una probabilidad es más «rico», para distinguirla bien de una función habitual en análisis; se trata del símbolo \mathbb{P} .

La notación que vamos a abordar a continuación, falsamente similar a la notación que hemos utilizado más arriba, parte naturalmente expresando la probabilidad de un resultado determinado, aquí nombrado ω .

La probabilidad de un evento constituido por el singleton $\{\omega\}$ se denomina naturalmente:

$$\mathbb{P}(\{\omega\})$$

Un singleton es un conjunto que contiene un único elemento.

 Esto se escribe en ocasiones $\mathbb{P}(\omega)$ en una notación algebraica (relativamente impropia aunque más habitual) o, también algebraica, $p(\omega)$. Notación que no debe confundirse con p_x , donde la x designa la variable aleatoria y no un

evento! La notación matemática, y en particular la notación en estadística, en economía y en física, es una historia de uso, de contexto, de estilo, de autor, de sentido, de país y de disciplina. Los anglosajones y los españoles no comparten exactamente la misma notación y la notación de los informáticos se limita al uso de caracteres ASCII en muchos casos. **Esté muy atento, por tanto, a la eventualidad de un contrasentido cuando aborde un texto matemático y verifique atentamente el sentido de los símbolos que se le presentan.**

Exploraremos más adelante el sentido de la expresión $\mathbb{P}(\{\omega\})$.

Cuando Ω es infinito, el número de eventos singleton es infinito y se confirma que $\mathbb{P}(\{\omega\})$ es, a menudo, nulo, lo que se entiende bien, pues la probabilidad de un resultado concreto frente a una cantidad infinita de resultados posibles parece converger a 0.

Cuando Ω es infinito, los ω tales que $\mathbb{P}(\{\omega\})$ no es nulo se denominan **átomos**; estos átomos se corresponden con lugares particulares de la función de la ley de probabilidad, y que solo podemos encontrar si esta es absolutamente continua.

Y a la inversa, en caso de que Ω sea infinito, tenemos numerosos singletons con probabilidades no nulas. Por ejemplo, echar al aire una moneda no trucada nos devuelve:

$$\begin{aligned}\Omega &= \{\text{cara, cruz}\} \\ \mathbb{P}(\{\text{cara}\}) &= 1/2 \\ \mathbb{P}(\{\text{cruz}\}) &= 1/2 \\ \mathbb{P}(\{\text{cara, cruz}\}) &= 1 = \mathbb{P}(\Omega)\end{aligned}$$

Estas últimas notaciones pueden escribirse de diversas maneras en función de los distintos autores, o según los usos vinculados a la expresión clásica de un teorema o de otro. A continuación le proponemos explorar estas escrituras para que pueda estar seguro de descifrar fácilmente los distintos textos que encontrará en su práctica de data scientist.

Exploración de las notaciones posibles

Destaquemos en primer lugar que la probabilidad de un evento representado por un elemento A de la familia \mathcal{A} se escribe naturalmente:

$$\mathbb{P}(A)$$

Esta notación simple no introduce la noción de variable aleatoria.

Las dificultades de notación aparecen justo en caso de que se quiera describir una probabilidad que haga referencia a una variable aleatoria. Para comprender lo que puede crear la confusión en las distintas notaciones que vamos a abordar a continuación, debe tener en mente, como se ha indicado más arriba, que una **variable aleatoria no es una variable**, sino una función.

Consideremos la variable aleatoria X ; el conjunto de eventos de una familia \mathcal{A} cuya imagen por X posea el valor v se escribirá: $\{a \in \mathcal{A} | X(a) = v\}$.

Si desea dominar este tema, medite esta última expresión.

La probabilidad de este conjunto de eventos, tales que la variable aleatoria aplicada a cada evento devuelve un valor v se escribirá entonces $\mathbb{P}(\{a \in \mathcal{A} | X(a) = v\})$ y, en escritura simplificada, $\mathbb{P}(X = v)$.

En una escritura todavía más simplificada, obtenemos $p_X(v)$.

iNo pierda de vista jamás el sentido profundo oculto detrás de estas escrituras simplificadas!

Imagine una colección de valores: $v_1 v_2 \dots v_n$ que podríamos describir utilizando un índice mudo (es decir, genérico, sin nada que nos recuerde a una colección de valores). Las probabilidades de los distintos valores pueden escribirse, por tanto, de la siguiente manera:

$$p_X(v_i)$$

Si no tenemos ambigüedad sobre X, esta notación se convierte en:

$$p(v_i)$$

Si existe una única colección de valores y una única variable aleatoria, entonces la notación se simplifica todavía más y se convierte en:

$$p_i$$

► Recuerde, utilizando la notación anterior, y considerando que los i valores se corresponden con i eventos independientes cuya unión representa todos los eventos posibles, se confirma que: $\sum_i p_i = 1$

Consideremos a continuación la variable aleatoria X; el conjunto de eventos de una familia \mathcal{A} cuya imagen por X pertenezca al conjunto V se escribe $\{a \in \mathcal{A} | X(a) \in V\}$,

lo que puede expresarse también de la siguiente manera:

$$X^{-1}(V)$$

En este caso la notación simplificada de:

$$\mathbb{P}(\{a \in \mathcal{A} | X(a) \in V\})$$

que vale también:

$\mathbb{P}(X^{-1}(V))$, será muy abusiva, pero muy habitual, y dará $\mathbb{P}(X \in V)$.

Para expresar la misma cantidad podemos invocar a su vez la noción de ley de probabilidad de la variable aleatoria X, que escribimos $p_X \dots$ como habíamos hecho al principio de este capítulo introduciendo la notación muy sencilla siguiente: p_X .

Esta ley es una medida de probabilidad, que es la imagen de \mathbb{P} por X (efectivamente, ¡X es una función!).

Como conclusión, tenemos que:

$$\mathbb{P}(\{a \in \mathcal{A} | X(a) \in V\}) = \mathbb{P}(X^{-1}(V)) = p_X(V) = \mathbb{P}(X \in V)$$

► A menudo se nombra los valores con la letra minúscula de la variable aleatoria: los posibles valores devueltos por la función variable aleatoria X serán x, los valores devueltos por Y serán y...

Veamos a continuación lo que podemos hacer simplemente a partir de estas nociones.

d. Momentos de una variable aleatoria discreta X

Hemos visto más arriba que podemos escribir las probabilidades de distintos valores de la siguiente manera (cuando no existe ambigüedad sobre la variable aleatoria y sobre los eventos considerados): p_i .

Por otro lado, hemos visto que si los i valores corresponden a i eventos independientes cuya unión representa todos los eventos posibles, entonces podemos escribir $\sum_i p_i = 1$.

Imagine que un profesor incompetente puntuara a sus alumnos de 0 a 20 sin utilizar valores intermedios ($n = 21$ notas posibles; $x_1 = 0, x_2 = 1 \dots x_{21} = 20$).

La probabilidad de obtener una nota entre 0 y 20 es evidentemente $\sum_i p_i = 1$ (¡100 % de opciones de tener una nota cualquiera!).

Debido a su incompetencia, el profesor extrae sus notas de un sombrero en el que ha introducido 21 papeles con las 21 notas. Toma la precaución de volver a introducir los papeles en el sombrero después de cada tirada, para dejar la misma probabilidad para cada nota asignada a un alumno (es decir, que cada p_i vale $1/21$).

En cierta manera, este profesor ha creado una máquina de fabricar distribuciones uniformes de notas.

Esperanza matemática: momento m_1

Percibimos que si el supervisor observara la media de las notas de la clase, no vería ninguna anomalía... pues esta media tiene toda la probabilidad de estar alrededor de 10. En efecto, es la **media ponderada por sus probabilidades** respectivas de las distintas notas posibles.

Es lo que denominamos esperanza matemática de la variable aleatoria discreta X y la denotaremos $E(x)$.

La expresión es la siguiente: $E(x) = \sum_i p_i x_i$.

Hagamos el cálculo en R:

```
## cálculo de E(X), distribución uniforme ##  
x <- 0:20      # 21 valores de v[1] ...v[21], notas de 0 ... 20  
n <- length(v) # tenemos efectivamente 21 notas  
  
p <- 1:21       # creación de las 21 probabilidades  
p[1:21] <- 1/n  # asignación de una probabilidad uniforme  
  
          # cálculo de E(X) = sigma de los pi.vi  
E <- 0          # inicialización de E(X)  
for (i in 1:21) {E <- E + x[i] * p[i]}  
E           # esperanza de X
```

Obtenemos $E(x) = 10$.

Comparemos este cálculo con la media de las notas posibles:

```
## re-cálculo de E(X) y comparación con la media m ##  
E <- sum(p*x)    # mismo cálculo más rápido  
                  # es el producto escalar de los vectores
```

```

# p y v
E

m <- sum(x)/n      # cálculo de la media
(m == E)           # comprueba si la esperanza y la media son iguales

```

La expresión ($m == E$) devuelve el resultado TRUE, la esperanza de nuestra distribución uniforme es, efectivamente, su media.

La esperanza matemática es el primer *momento* característico de una distribución. Expresa la tendencia central de la distribución; en ocasiones se denomina simplemente *media de la variable*.

Para una mayor compacidad, es habitual denotar la esperanza matemática μ o m_1 (el $_1$ significa *primer momento*).

Como nuestro profesor ha extraído realmente al azar las notas, sin duda existe una diferencia entre el valor estimado de la esperanza matemática (aquí 10) y el valor constatado. Vamos a pedirle a R que nos fabrique una distribución uniforme discreta aleatoria y vamos a realizar el cálculo anterior para hacernos una idea de la diferencia. A continuación, repetiremos este cálculo 10 000 veces y compararemos el valor de la esperanza aleatoria media de las distintas iteraciones con el valor teórico (10).

```

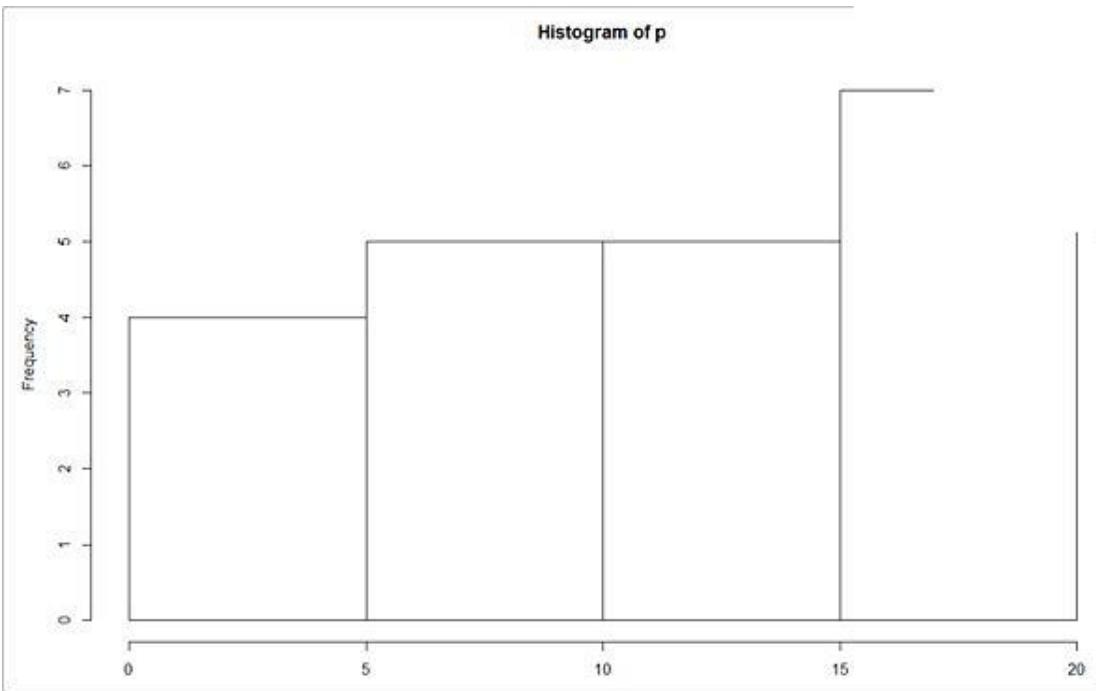
## con una distribución uniforme incluyendo aleatoriedad      ##
## cálculo de la media de E sobre 10000 intentos            ##

E   <- 0      # inicialización de la acumulación de E
num <- 0      # inicialización del contador del número de bucles

for (s in 1:10000) {  # tendremos 10000 sucesos aleatorios
  set.seed(s) # reinicializa el generador de números aleatorios
  num <- num + 1
  p <- sample(1:20, 21, replace=T)      # distribución uniforme
  if (num == 1) hist(p) # dibujemos una distribución sobre 10000
  p <- p/sum(p)
  x <- 0:20    # 21 valores de v[1] ...v[21], notas de 0 ... 20
  E <- E + sum(p*x)
}
E <- E/num # cálculo de la media de las 10000 esperanzas
E

```

Lo que produce el siguiente histograma para la primera iteración, donde comprobamos que la aleatoriedad es menos uniforme que en el cálculo original.



Distribución real de las notas obtenidas con la primera iteración

Sin embargo, tras las 10 000 iteraciones, comprobamos que la esperanza matemática media es igual a 10.00566.

Todo el mundo ha oído hablar de la ley normal con forma de campana, veamos con un pequeño programa en R similar qué obtendríamos con una ley normal.

```
## con una distribución ley normal con aleatoriedad      ##
## cálculo de la media de E sobre 10000 intentos      ##

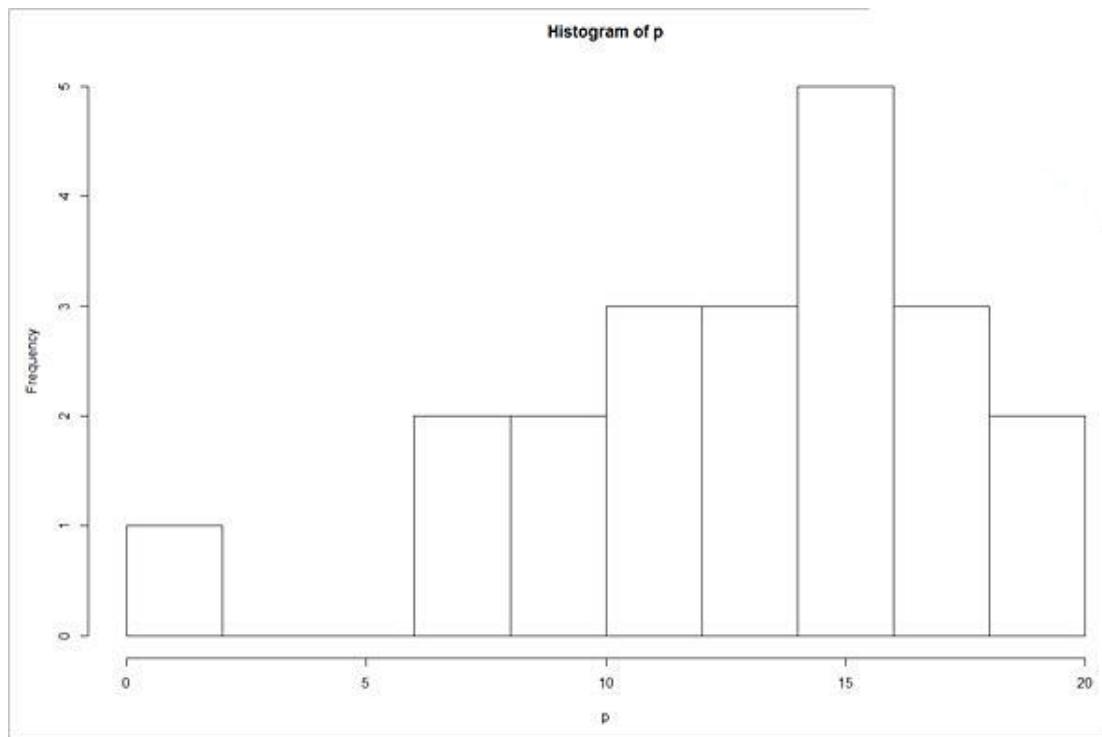
fn <- function (s){# reparto de notas - ley normal de 0 a 20
  set.seed(s)
  p <- rnorm(21, mean = 10, sd = 5) # ley normal
  p <- p - min(p)
  d <- max(p) - min(p)
  p <- (p/d)*20
}

E  <- 0      # inicialización de la acumulación de E
num <- 0      # inicialización del contador del número de bucles

for (s in 1:10000) { # tendremos 10000 sucesos aleatorios
  num <- num + 1
  p   <- fn(s)        # llamada a la distribución
                      # dibujemos una distribución sobre 10000
  if (num == 1) hist(p, breaks = 10)

  p <- p/sum(p)
  x <- 0:20      # 21 valores de v[1] ...v[21], notas de 0 ... 20
  E <- E + sum(p*x)
}
E <- E/num      # cálculo de la media de las 10000 esperanzas
```

Lo que produce el siguiente histograma para la primera iteración, donde comprobamos que para esta primera iteración la nota 10 no es, sin duda, la media.



Distribución real de las notas obtenidas con la primera iteración

Sin embargo, tras las 10 000 iteraciones comprobamos que la esperanza matemática media es igual a 10.00579.

Nuestro profesor habría podido crear fácilmente una distribución de notas centrada en la media, pero con notas más verosímiles (es decir, menos notas extremas) si hubiera utilizado un mecanismo generador de distribuciones normales.

Esperanza matemática (m_1): generalización

La fórmula de la media utilizada más arriba solo funciona sobre distribuciones discretas; la siguiente expresión es la más general:

$$\mathbb{E}(x) = \int_{\mathbb{R}} x \mathbb{P}_X(dx)$$

Esta última expresión se simplifica en el caso de que la variable aleatoria sea continua y posea una densidad de probabilidad f , tal que la probabilidad para cualquier intervalo real se exprese de la siguiente manera en función de f :

$$\mathbb{P}_X([a; b]) = \int_a^b f(x) dx$$

En este caso se obtiene:

$$\mathbb{E}(x) = \int_{-\infty}^{+\infty} x f(x) dx$$

Viendo atentamente esta formulación, comprobamos que no siempre es la media de las x ; en efecto, $f(x)$ es una densidad comprendida entre 0 y 1.

 Preste atención: encontraremos otras notaciones, que utilizaremos en ocasiones cuando resulten más prácticas o tradicionales en algún contexto determinado, o cuando se quiera subrayar algún matiz.

$$\mathbb{E}(x) = \langle x \rangle = \bar{x} = \mu_x$$

Con frecuencia, \bar{x} designa a la media de la distribución (empírica), mientras que μ_x designa a la media de la ley.

La varianza: momento m_2

Ahora que disponemos de una representación que nos permite percibir la tendencia central de una distribución, vamos a construir una representación de su dispersión.

En efecto, en toda primera aproximación, si tiene una idea del «centro» de la distribución y de su espaciamiento, ya tiene una cierta percepción de su distribución.

Se construye una diferencia entre la variable y su media, elevada al cuadrado para evitar que una dispersión a la derecha no se compense con una dispersión a la izquierda y se calcula la media en el sentido estadístico, es decir, su esperanza matemática.

La expresión resultante es la siguiente:

$$m_2 = \mathbb{E}[(X - \mu)^2]$$

Este momento se denomina *varianza*.

Para comprender mejor la dispersión sobre una distribución operacional, hay que poder comparar valores comparables, es decir, que se correspondan con la misma unidad de medida, también se dice la *misma dimensión* (metro, kilo, edad...) que la variable aleatoria. Sin tener en cuenta que la varianza es homogénea al cuadrado de la unidad de la variable aleatoria. Tomamos la raíz, que denominamos desviación típica (*standard deviation* en inglés: stdv).

Por ello, a menudo se escribe la varianza como el cuadrado de la desviación típica: σ_x^2 .

En el caso discreto, esta expresión se convierte en:

$$\sigma_x^2 = \sum_i p_i (x_i - \mu)^2$$

y en el caso continuo:

$$\sigma_x^2 = \int_{-\infty}^{+\infty} (x_i - \mu)^2 f(x) dx$$

De manera cotidiana, la desviación típica se utiliza a menudo directamente para comprender la dispersión. Sin embargo, imagine una distribución con una media de 10 y una desviación típica de 1 y otra con una media de 1 000 y una desviación típica de 2; podría pensar legítimamente que la dispersión de la primera es más consecuente que la segunda, aunque su desviación típica es la mitad. Esta consideración nos lleva al uso de otros indicadores de dispersión, como el *coeficiente de variación* (que no debe confundirse con su homónimo en análisis matemático). El coeficiente de variación es, simplemente, la desviación típica dividida por la media, lo que hace posibles las comparaciones. Observe, sin embargo, que cuando se trabaja con valores negativos la media puede ser nula y el coeficiente de variación no tiene sentido.

Si no existe ambigüedad en el nombre de la variable aleatoria, su expresión es la siguiente:

$$c_v = \frac{\sigma}{\mu}$$

En inglés, hablamos en ocasiones de RSD, de *relative standard deviation*, que se corresponde con el coeficiente de variación en valor absoluto y que a menudo se expresa en porcentaje.

- Otra forma de medir la dispersión, particularmente robusta, consiste en medir la desviación del valor que se corresponde con un cierto porcentaje de la población estudiada (típicamente el 50 %), centrado sobre la mediana tal que 50 % de la población está por debajo y el resto por encima de ella. Hablamos de desviación intercuartil, pues se corresponde de hecho con la diferencia entre los cuartiles Q3 y Q1. En inglés, se denomina IQR por Interquartile Range o en ocasiones Midspread.

Momentos m_3 y m_4

Sus respectivos nombres son los siguientes: *Skewness* (coeficiente de asimetría) y *Curtosis* (coeficiente de aplastamiento).

Son parámetros de forma.

Cuando utilice su implementación en R, a menudo será preferible utilizar formulaciones menos rígidas que las formulaciones de base de estos momentos; veremos esto en el código R que aparece más adelante.

La Curtosis de una ley normal normalizada es igual a 3.

Como la ley normal es una referencia, se calcula a menudo el valor *curtosis* - 3, que denominamos *exceso_de_curtosis*. El *exceso_de_curtosis* de la ley normal vale, por consiguiente, 0.

Una ley más puntiaguda posee un *exceso_de_curtosis* positivo que puede ser de 2, y una ley muy aplanada, como la ley uniforme, posee un *exceso_de_curtosis* negativo inferior a -1.

Un riesgo habitual es aplicar algoritmos que presuponen la normalidad de las leyes, mientras que su coeficiente de aplastamiento es muy diferente al de una ley normal o bien existe cierta deformación («larga cola»), típicamente a la derecha de la media de la distribución.

En ciertos casos, podemos utilizar estos dos momentos para crear nuevas variables (features) para explotar en nuestros modelos predictivos.

He aquí el código en R que permite crear una pequeña tabla resumen de las medias, desviaciones típicas,

skewness y exceso_de_curtosis sobre tres distribuciones clásicas: Uniforme, Normal y Poisson.

```
## cálculo de la media, desviación típica, skewness y curtosis de tres distr ##
## creación de una pequeña tabla resumen ##

library(e1071) # la caja de herramientas
set.seed(10)    # inicialización del generador de números aleatorios
l_uniforme <- runif(10000)      # 10000 valores sobre la ley uniforme
l_normal    <- rnorm(10000, mean = 1)   # sobre la ley normal
l_poisson   <- rpois(10000, lambda = 1) # sobre la ley de poisson

# combinemos las columnas
all <- cbind(l_uniforme,l_normal,l_poisson)

# aplicación de las funciones
media           <- apply(all,2,mean)
desviacion_tipica <- apply(all,2,sd)
skewness         <- apply(all,2,function(x){skewness(x, type = 3)})
xcurtosis       <- apply(all,2,function(x){curtosis(x, type = 3)})

# combinemos las filas
all <- rbind(media,desviacion_tipica,skewness,curtosis)
all <- round(all, digits = 1)      # redondeo
all
```

Lo que devuelve la tabla:

	l_uniforme	l_normal	l_poisson
media	0.5	1	1.0
desviacion_tipica	0.3	1	1.0
skewness	0.0	0	1.0
curtosis	-1.2	0	0.9

Comprobamos que la ley de Poisson es asimétrica con una cola a la derecha (skewness positiva que vale 1) y que la ley uniforme es muy plana (exceso_de_curtosis = -1.2).

 Observe:

- 1) El uso de tipo = 3 en los momentos m_3 y m_4 , para estipular que no se utilicen las formas rígidas de estas funciones.
- 2) El uso de apply para aplicar una función sobre las columnas mediante el parámetro 2.
- 3) El uso de funciones genéricas no nombradas en apply.

e. Primeras consideraciones sobre los errores y estimaciones

Supongamos que ha efectuado un cálculo de media. Si este cálculo se ha realizado sobre una muestra de una población más importante, tendrá sin duda que considerar que esta media representa una estimación de la media real de la distribución. Se planteará, lógicamente, la cuestión de saber cuál es el orden de magnitud del error al que se ve sometido considerando esta media como la de toda la distribución.

La respuesta a esta preocupación en lo relativo a la estimación de la media se denomina *error tipo de la media* (SEM, del inglés *Standard Error of the Mean*) y su expresión es la siguiente:

$$\frac{\sigma}{\sqrt{n}}$$

Comprobamos que, cuando esta muestra es muy grande, el error tipo de la media tiende a 0, lo cual es natural, pues en ese caso se calcula la media sobre una muestra que se aproxima a la población entera.

- En términos de notación, el uso «data science» es, a menudo, denotar \bar{x} a la media calculada sobre la muestra y μ a la media de la distribución de referencia. Además, se denota a menudo s a la desviación típica calculada sobre la muestra, mientras que σ designa la desviación típica de la distribución de referencia.

Para indicar que hablamos de una estimación, el uso suele ser indicar un pequeño sombrero sobre el nombre del objeto considerado. Con esta notación, la estimación de la media μ se escribe $\hat{\mu}$.

Con esta notación, para una **muestra determinada** el error calculado cometido sobre el cálculo de la media resulta ser:

$$e = \hat{\mu} - \mu$$

2. Familiarizarse con los datos

Valiéndonos de nuestros nuevos conocimientos, vamos a utilizar algunas herramientas interactivas para familiarizarnos rápidamente con nuestro juego de datos. Aquí, vamos a observar rápidamente un pequeño juego de datos estadísticos (Longley) relativos a la tasa de paro en EE. UU. que hemos preparado. Este juego de datos está incluido en R dentro del paquete **stats**.

```
require(stats)
longley
```

En el sitio que acompaña a este libro, obtendrá la versión preparada de este juego de datos invocando la función descrita en el siguiente código. Además, este código le presenta una representación de los datos por parejas, que puede consultar en el capítulo Feature Engineering de este libro).

```
# LECTURA Y PREPARACIÓN DEL DATA SET LONGLEY SOBRE PIB EE. UU. Y TASA DE PARO
# uso en modo verbose (default): Z <- f_longley()
# uso en modo mudo : Z <- f_longley(FALSE)
f_longley <- function(verbose = TRUE){
  require(stats)      # estadísticas de base
  require(graphics)   # gráficos de base
  require(dplyr)       # librería de manipulación de datos
  source("pair_panels.R")
  Z <- data.frame(longley[, 1:7], row.names = NULL) # lectura
  Z <- rename(Z,      # inglés -> español
             anyo      = Year,
             índice_PIB = GNP.deflator,
             PIB       = GNP,
             parados   = Unemployed,
```

```

militares = Armed.Forces,
población = Population,
trabajadores = Employed)

orden <- sort(names(Z))      # obtener las columnas por orden alfabético
Z <- data.frame(Z[,orden])  # reclasificar las columnas

if (verbose){
  glimpse(Z)               # ver qué hay en los datos
  pairs(Z,diag.panel = panel.hist,
        upper.panel = panel.cor,
        lower.panel = panel.smooth,
        main ="visualización rápida: data longley")
  y <- data.frame(trabajadores = Z$trabajadores) # regresión
  X <- select(Z, -trabajadores)                  # features
  print(summary(lm(y[,1]~ .,X)))                # REGRESSION TEST
}
Z                                         # data_frame
}

```

```
df1 <- f_longley()      # nuestro data frame se llamará df1
```

En RStudio, si hace clic en **df1** dentro de la ventana superior derecha obtendrá esto:

	año	parados	índice_PIB	militares	PIB	población	trabajado.
1	1947	235.6	83.0	159.0	234.289	107.608	60.323
2	1948	232.5	88.5	145.6	259.426	108.632	61.122
3	1949	368.2	88.2	161.6	258.054	109.773	60.171
4	1950	335.1	89.5	165.0	284.599	110.929	61.187
5	1951	209.9	96.2	309.9	328.975	112.075	63.221
6	1952	193.2	98.1	359.4	346.999	113.270	63.639
7	1953	187.0	99.0	354.7	365.385	115.094	64.989
8	1954	357.8	100.0	335.0	363.112	116.219	63.761
9	1955	290.4	101.2	304.8	397.469	117.388	66.019
10	1956	282.2	104.6	285.7	419.180	118.734	67.857
11	1957	293.6	108.4	279.8	442.769	120.445	68.169
12	1958	468.1	110.8	263.7	444.546	121.950	66.513
13	1959	381.3	112.6	255.2	482.704	123.366	68.655
14	1960	393.1	114.2	251.4	502.601	125.368	69.564
15	1961	480.6	115.7	257.2	518.173	127.852	69.331
16	1962	400.7	116.9	282.7	554.894	130.081	70.551

Veamos a continuación cómo comprender nuestros datos de manera interactiva.

a. R Commander

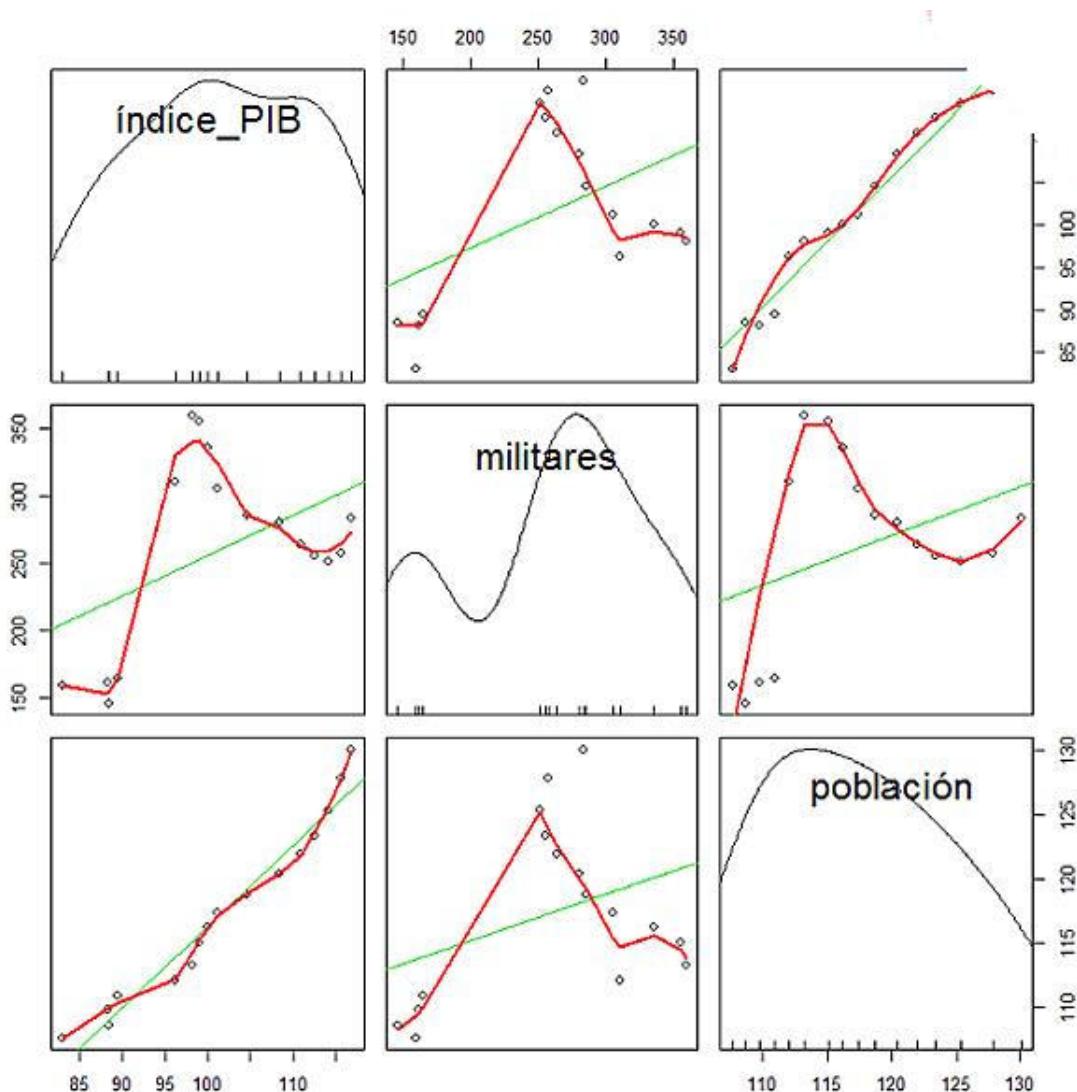
La herramienta **R Commander** permite visualizar cierta información relativa a sus datos.

Para invocarla basta con invocar el paquete.

```
library(Rcmdr)
```

Le dejamos explorar esta pequeña herramienta, pero, para empezar, sepa que debe indicar los datos que va a utilizar mediante el menú **Datos - Juego de datos activo - Seleccionar el juego de datos activo**.

Tras seleccionar el **data.frame df1**, puede seleccionar los datos que desea cruzar entre sí y visualizar las densidades aproximadas de las distintas distribuciones, lo cual resulta muy útil para comprender la muestra mediante los menús **Gráficos - Matriz de puntos**. Seleccionando variables con la tecla [Ctrl] más un clic con el botón izquierdo y, por último, presionando [Intro], obtendrá los gráficos por pares.

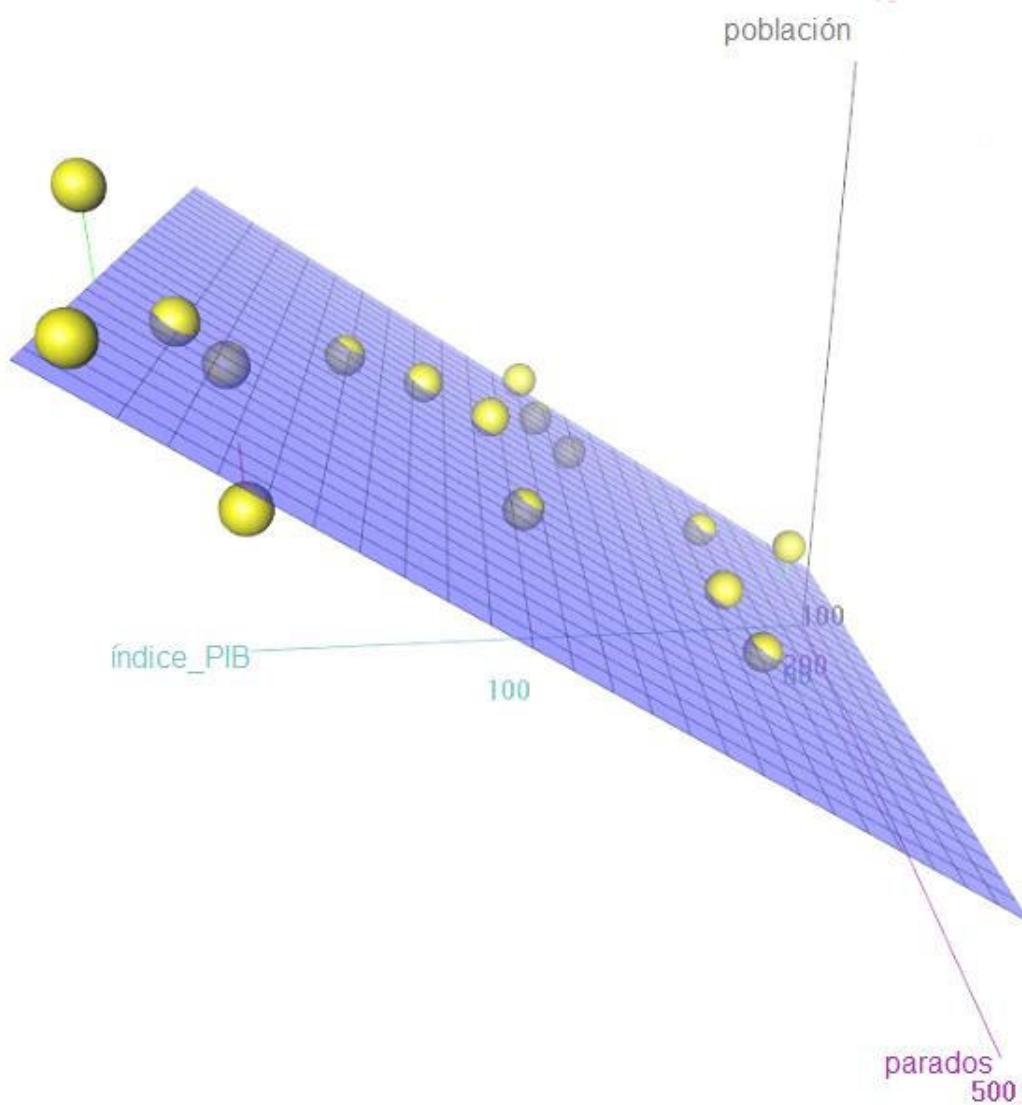


Diagramas por pares (regresión) y distribuciones

Comprobamos que las distribuciones aproximadas (en la diagonal) no tienen un aire muy gaussiano y que la distribución **militares** tiene el mismo aspecto bimodal (dos cimas), lo que podría llevarnos a tomar cierta precaución en cuanto al uso de algoritmos que partan de la hipótesis de la normalidad de las distribuciones.

Nos llama la atención la correlación visual entre **población** e **índice_PIB**.

Esta linealidad puede ser covariante, de modo que nos animamos a observar los gráficos en 3D para comprender mejor nuestros datos. El menú **Gráficos - 3Dgráficos** nos lo permite fácilmente y nos ofrece la capacidad de girar el gráfico utilizando el ratón.



Tres variables próximas en un plano

- Comprobará con agrado que todas las instrucciones de R que han servido para configurar los gráficos están visibles y pueden manipularse en la interfaz de R Commander.

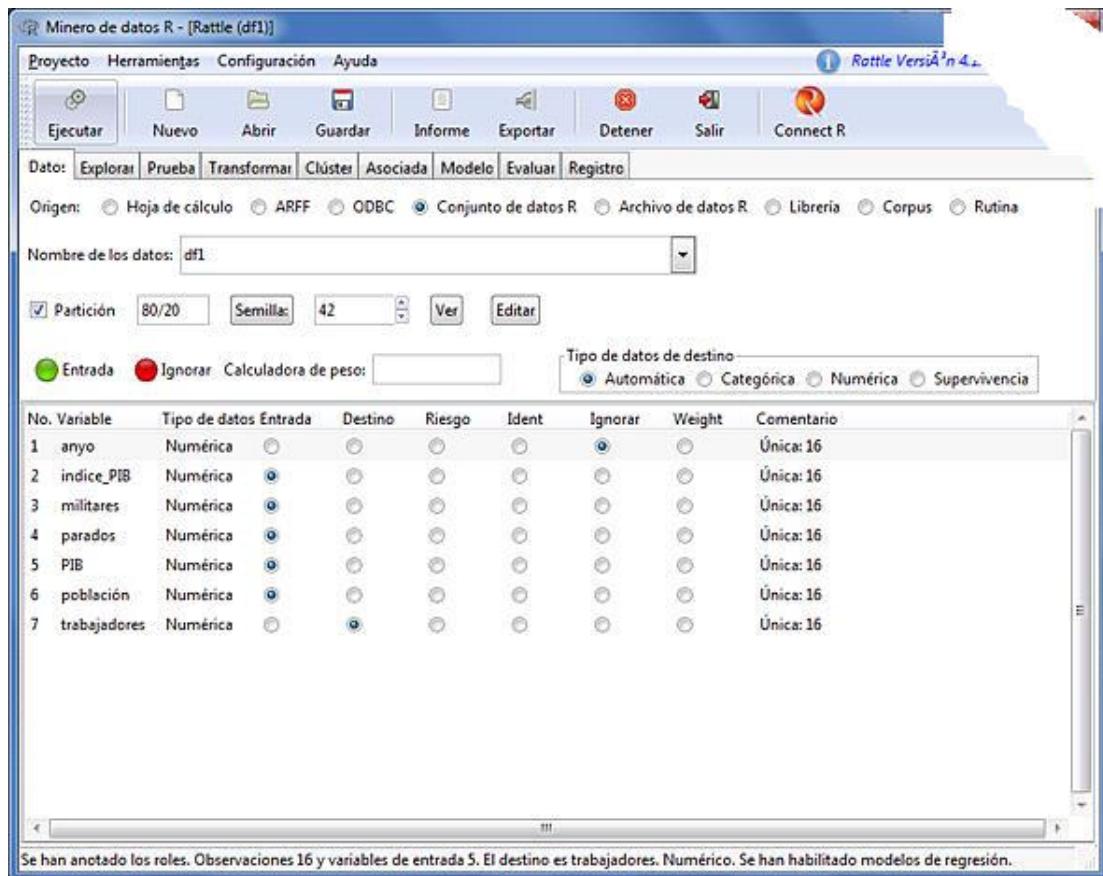
Otra herramienta nos va a aportar un complemento de información; también es interactiva, y se llama **Rattle**.

b. Rattle

Vamos a utilizar esta otra herramienta R muy práctica para explorar nuestros datos, el conocido «dataminer» Rattle. No hay nada más sencillo que invocarlo.

```
# rattle  
library(rattle)  
rattle()
```

Se abre una interfaz que le propongo configurar indicando que desea trabajar con **df1** como juego de datos R, estipulando que su objetivo será **trabajadores** y que va a ignorar el **anyo**. La partición de sus datos es un **80%** para entrenamiento y un **20%** para test/validación (sin separar test de validación aquí). Preste atención, hay que hacer clic en **Ejecutar** tras cada cambio en la interfaz gráfica; en caso contrario Rattle no ejecuta nada.



Interfaz de Rattle

Rattle nos va a permitir explorar nuestros datos. En el archivo **Journal** encontrará el conjunto de comandos de R generados por Rattle, resulta muy útil para progresar estudiándolos. Rattle utiliza otros muchos paquetes y le propondrá instalarlos conforme los vaya a utilizar.

Sin duda, debería echar un vistazo a las correlaciones lineales entre variables numéricas.

Minero de datos R - [Rattle (df1)]

Proyecto Herramientas Configuración Ayuda Rattle Versión

Ejecutar Nuevo Abrir Guardar Informe Exportar Detener Salir Connect R

Datos Explorar Prueba Transformar Clúster Asociada Modelo Evaluar Registro

Tipo: Suma Distribuciones Correlación Componentes principales Interactivo

Organizado Explorar faltantes Jerárquico Método: Pearson

Resumen de correlación con la covarianza 'Pearson'.

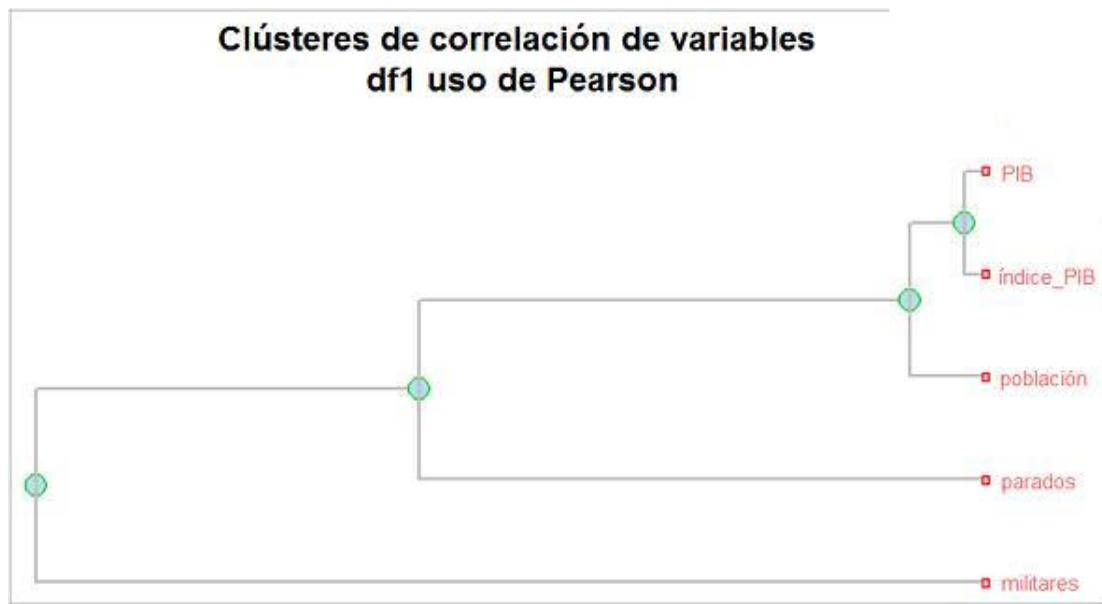
Observe que solo se tienen en cuenta las correlaciones entre las variables numéricas.

	militares	PIB	indice_PIB	poblacion	parados
militares	1.0000000	0.4118257	0.4393722	0.3399045	-0.1383884
PIB	0.4118257	1.0000000	0.9928679	0.9918176	0.6121132
indice_PIB	0.4393722	0.9928679	1.0000000	0.9801487	0.6196313
poblacion	0.3399045	0.9918176	0.9801487	1.0000000	0.6926895
parados	-0.1383884	0.6121132	0.6196313	0.6926895	1.0000000

Clúster jerárquico de correlaciones diagramas.

Correlaciones entre variables numéricas

La ejecución de esta configuración ha creado un **dendograma** de correlaciones en RStudio.



Jerarquía de correlaciones

Empezamos a percibir la estructura de las features; evidentemente, no debería llevarnos a ninguna conclusión, pero estas representaciones nos permiten comprender los datos.

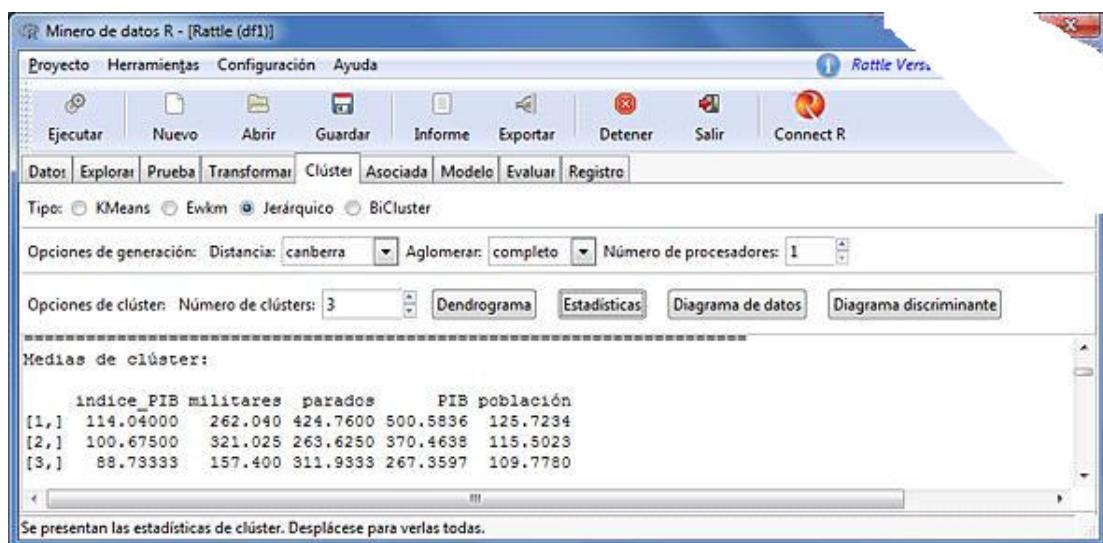
A continuación vamos a observar si esto se «clusteriza» un poco, es decir, si se encuentran grupos de observaciones. Vamos a escoger un mecanismo bastante resistente al hecho de que nuestro conocimiento de los datos sea débil. Vamos a realizar un **clustering jerárquico ascendente**; la mecánica es, por tanto, un poco simplista, aunque no introduce complejidades ni perjudica nuestra comprensión de los datos.

Nos dotamos de una distancia, hemos seleccionado la **distancia de Canberra**. La distancia de Canberra es una distancia de Manhattan que se ha ponderado. Dicho de otro modo, es una distancia basada en la distancia absoluta, que funciona por «pasos» sucesivos, pero que se pondrá en función de los valores de las variables. Esto hace que los «pasos» entre las distintas direcciones tengan un impacto similar.

Consideremos dos líneas cualesquiera de observación y p features. La distancia de Canberra entre dos observaciones es la siguiente:

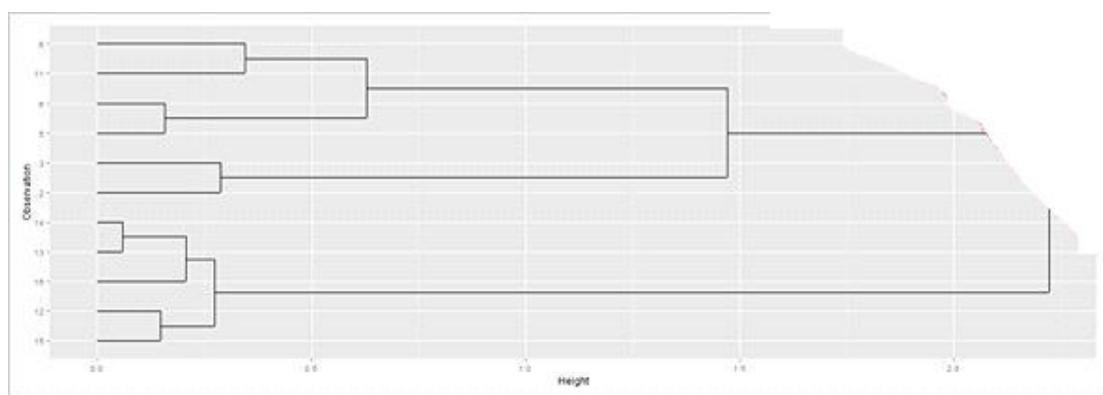
$$d(x_{i1}, x_{i2}) = \sum_{j=1}^{j=p} \frac{|x_{i1,j} - x_{i2,j}|}{|x_{i1,j}| + |x_{i2,j}|}$$

- Brevemente: el algoritmo empieza buscando los dos primeros puntos más próximos y forma el primer clúster. Calcula el centro de ambos puntos, lo que produce un nuevo punto (ficticio) y a continuación se itera. Los puntos se reemplazan por su centro y se vuelve a empezar, cada punto ficticio se convierte en un nivel en el dendograma correspondiente. Se obtiene una clase única. Esta clase se ve como un dendrograma que podemos explotar a continuación. El dendrograma de construcción se dividirá en clústeres, se poda en el nivel en que se obtenga el mejor reparto.



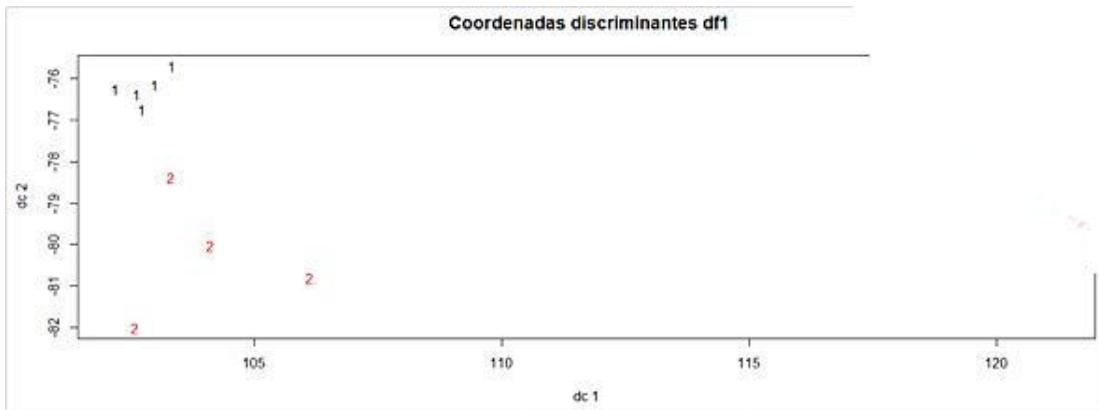
El menú Clúster

Vemos perfectamente los tres clústeres.



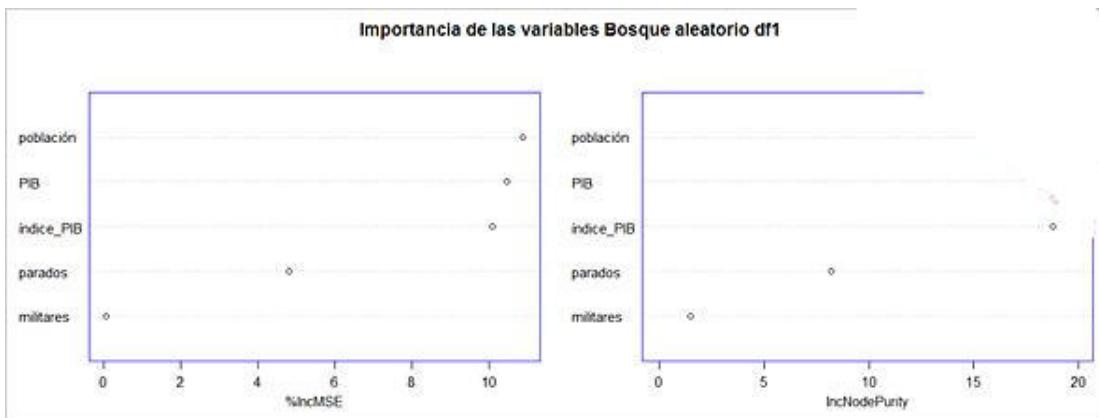
Tres clases en el dendrograma

Esto aparece también en el siguiente diagrama, que muestra que el clúster 3 es el más marcado.



Clústeres en dos dimensiones

En el menú **Model** de Rattle, seleccione **Random forest** y haga clic en el botón **Importance**. Obtendrá un producto derivado muy útil de la aplicación de este modelo que le permite hacerse una cierta idea de la influencia de las distintas variables sobre una eventual predicción.



Veremos en el capítulo Feature Engineering que hay que matizar bastante la interpretación de dicho diagrama en cuanto al hecho de eliminar una u otra feature. Y a la inversa, es probable que tenga que componer un cierto panel de features importantes y menos importantes para realizar una predicción excelente.

Ahora que ha tenido la ocasión de comprender los datos de una manera más descriptiva, vamos a evolucionar hacia un tratamiento más analítico y predictivo. Para comprender los mecanismos, vamos a tener que abordar las matrices y los vectores.

Matrices y vectores

1. Convenciones, notaciones, usos básicos

La pretensión de esta sección es que se familiarice con el uso práctico de las matrices y los vectores, tal y como se presenta con mayor frecuencia en este libro. El aspecto matemático subyacente no se tiene en cuenta (o se tiene en cuenta poco).

Fabriquemos una matriz

Supongamos un dataset compuesto de filas (por ejemplo, que representan eventos o individuos) y, por lo tanto, columnas que representan los valores para cada atributo (en inglés, feature). Una fila que describa un individuo llamado Roberto, de 40 años de edad, con estado civil casado y habitante de la ciudad de Madrid podría codificarse de la siguiente manera en una tabla con otros individuos:

```
!nombre      ! edad ! estado ! ciudad !
!           !       !       !       !
!ROBERTO    !     40!     C ! MADRID !
!FELIPE     !     30!     S ! LEON   !
!RODOLFO    !     20!     S ! LEON   !
!MARIA      !     35!     C ! MADRID !
```

Si memorizamos el número de fila de cada individuo:

```
!nombre      ! fila  !
!           !       !
!ROBERTO    !     1   !
!FELIPE     !     2   !
!RODOLFO    !     3   !
!MARIA      !     4   !
```

Si se codifica el estado civil de la siguiente manera: «C» vale 1 y «S» vale 0, y si se reemplaza a continuación la ciudad por su código postal, podemos obtener una tabla de cifras con el mismo significado:

```
! fila  ! edad ! estado ! ciudad !
!       !       !       !       !
! 1    !     40!     1 ! 28000 !
! 2    !     30!     0 ! 24000 !
! 3    !     20!     0 ! 24000 !
! 4    !     35!     1 ! 28000 !
```

Si, por otro lado, nos acordamos del significado y del orden de cada columna, resulta inútil nombrarlas. Se obtiene:

```
! 1      !     40!     1 ! 28000 !
! 2      !     30!     0 ! 24000 !
! 3      !     20!     0 ! 24000 !
! 4      !     35!     1 ! 28000 !
```

Como vemos que 1 es la fila 1, 2 es la fila 2... Parece evidente que podemos ahorrar todavía más en la notación y no mencionar el número de fila, sin perder información alguna:

```
! 40! 1 ! 28000 !
! 30! 0 ! 24000 !
! 20! 0 ! 24000 !
! 35! 1 ! 28000 !
```

Esta representación contiene cuatro filas y tres columnas de elementos **homogéneos en términos de su naturaleza**. Diremos que es una **matriz**. Es posible tener matrices de números enteros, reales o complejos, de booleanos (verdadero-falso). ¡Incluso podemos imaginar matrices de matrices!

La condición de homogeneidad en términos de naturaleza no implica ninguna homogeneidad en términos de semántica. Cada elemento puede representar cualquier cosa completamente diferente (por ejemplo, la edad no tiene nada que ver semánticamente con el código postal!). En las matrices que manipularemos en data sciences, tendremos siempre una homogeneidad semántica por fila o por columna; la mayoría de los casos por columna (aquí tenemos tres columnas homogéneas semánticamente, la primera representa edades, la segunda el estado civil y la tercera la ciudad).

Nos gusta nombrar las matrices con una letra mayúscula (en ocasiones en negrita), aunque nada nos obliga a ello. Es habitual poner los elementos entre paréntesis, si bien nuestra matriz, que podemos llamar M, se escribirá entonces de la siguiente manera:

$$M = \begin{pmatrix} 40 & 1 & 75000 \\ 30 & 0 & 69000 \\ 20 & 0 & 69000 \\ 35 & 1 & 45000 \end{pmatrix}$$

Utilicemos nuestra matriz

Si queremos hacer referencia a esta matriz, bastará con hablar de M. Si queremos recordar que se trata de una matriz de cuatro filas y tres columnas, podemos llamarla: $M_{4,3}$.

Observe bien este «lema»: **fila por columna** (y no a la inversa!). Aquí tenemos 4 X 3 elementos en la matriz.

Cada elemento de la matriz puede designarse según su posición dentro de la matriz; por ejemplo, el elemento situado en la segunda fila y la tercera columna vale 24000. Es la ciudad de Felipe. Cuando se nombra la matriz mediante una letra mayúscula, es habitual utilizar la misma letra en minúscula para designar un elemento. Evidentemente, hay que indicar aquí su número de fila y su número de columna, que ponemos como índices. La ciudad de Felipe se nombrará entonces: $m_{2,3}$.

Tenemos que $m_{2,3} = 24000$.

Con la misma idea tenemos: $M = \begin{pmatrix} m_{1,1} & m_{1,2} & m_{1,3} \\ m_{2,1} & m_{2,2} & m_{2,3} \\ m_{3,1} & m_{3,2} & m_{3,3} \\ m_{4,1} & m_{4,2} & m_{4,3} \end{pmatrix} = M_{4,3}$

Para definir una matriz con R, basta con indicar el número de filas (o de columnas) y decirle si se quiere llenar fila a fila o columna a columna, lo que produce el siguiente código para nuestra matriz M:

```
## introducir una matriz, por fila, 4 filas ##

M <- matrix (byrow = TRUE,
             nrow = 4, # por fila
             # número de filas
```

```

      c( 40, 1 , 28000 ,
          30, 0 , 24000 ,
          20, 0 , 24000 ,
          35, 1 , 28000 )
)

```

Lo que produce, efectivamente:

	[,1]	[,2]	[,3]
[1,]	40	1	28000
[2,]	30	0	24000
[3,]	20	0	24000
[4,]	35	1	28000

Para obtener $m_{2,3}$ basta con escribir (observe los corchetes en lugar de los paréntesis):

```
M[2,3] # segunda fila  
       # y tercera columna
```

Se obtiene 24000 como se esperaba.

Para obtener toda la segunda fila, basta con no indicar el número de columna (observe la coma seguida de... inada!):

```
M[2, ]                                # segunda fila  
is.matrix(M[2, ])                      # ¡no es una matriz!
```

La segunda fila se extrae correctamente (observe que el resultado es monofila, y R indica [1]), es decir, una fila como resultado:

```
[1]      30      0 24000  
[1] FALSE
```

Pero desafortunadamente el resultado no es una matriz de una fila tal y como desmuestra el resultado **FALSE** para la función **is.matrix()**. **Para obtener una matriz, hay que imponerlo**: este punto es muy importante.

```
m2_ <- matrix(byrow = TRUE,           # una matriz de una fila
              nrow   = 1,
              M[2, ])
```

Podemos verificar las dimensiones:

Es una matriz de una fila y de tres columnas:

```
[,1] [,2] [,3]  
[1,] 30 0 69000  
  
[1] TRUE  
[1] 1 3
```

Con la misma idea, vamos a extraer una columna de la matriz, la columna número 3:

```
m_3 <- matrix(byrow = TRUE,           # una matriz de una columna  
                nrow = 4,  
                M[, 3])  
  
m_3  
is.matrix(m_3)                      # es una matriz  
dim(m_3)                            # una columna de tres filas
```

Se obtiene una matriz de una columna:

```
[,1]  
[1,] 28000  
[2,] 24000  
[3,] 24000  
[4,] 28000  
  
[1] TRUE  
[1] 4 1
```

Extraer vectores

En nuestra práctica cotidiana denominamos **vector** a una matriz que contiene **una sola columna**.

La construcción de vectores es sencilla; en efecto, la función **matrix()** posee un comportamiento por defecto que consiste en fabricar un vector columna si no se le indica ningún parámetro.

Podemos simplificar la creación de un vector columna **m_3** de la siguiente manera:

```
m_3 <- matrix(M[, 3])           # columna por defecto  
m_3
```

```
[,1]  
[1,] 28000  
[2,] 24000  
[3,] 24000  
[4,] 28000
```

Aunque no es el resultado esperado para **m2_**, pues deseamos obtener una fila, de modo que hay que trasponer la columna obtenida para obtener una matriz fila. Esto se realiza utilizando la función de trasposición **t()**.

```

m2_ <- matrix(M[2, ])      # !columna por defecto
m2_
                      # Problema: se obtiene una columna!

m2_ <- t(matrix(M[2, ]))    # trasposición
m2_

```

El primer resultado no nos convence:

```

[,1]
[1,] 30
[2,] 0
[3,] 24000

```

Tras trasponerlo se obtiene una matriz fila:

```

[,1] [,2] [,3]
[1,] 30     0 24000

```

Diferentes notaciones para los vectores y su traspuesto

En España nos gusta decorar los vectores con una pequeña flecha, de la siguiente manera (aquí para un vector « \vec{x} »):

$$\vec{x}$$

► Esta notación es útil cuando puede existir cierta ambigüedad, lo cual es poco habitual en nuestra disciplina. Esta no es la notación que utilizaremos habitualmente. Nos contentaremos con llamarlo x , el vector en cuestión!

Con esta notación, la matriz fila obtenida por la trasposición del vector x se denominará:

$$\vec{x}^T$$

La T significa «traspuesta de» y no es un exponente (ipreste atención!).

Hay quien prefiere escribir:

$${}^T\vec{x}$$

Para indicar la «traspuesta de», algunos autores utilizan también el término «prima»:

$$\vec{x}'$$

► El signo «prima» puede tener otros muchos significados (derivada, primero vs. segundo...); compruebe siempre el sentido de la notación utilizada por un autor.

En ocasiones los autores llaman «vector-fila» a la traspuesta de un vector «normal», que nosotros denominamos a

menudo «vector-columna».

¿Vector (matemático) o vector?

En R, tendremos a menudo la ocasión de manipular datos de tipo **vector**. Preste atención, pues no hablamos estrictamente de vectores, en el sentido de que no es posible realizar naturalmente las operaciones propias que se realizan entre vectores y matrices.

El tipo **vector** es fundamental en R, representa la combinación de varios valores en un agregado que puede manipularse de manera sencilla por diversas funciones. El uso de este tipo de estructura de datos es característico de un lenguaje vectorizado. Dicho lenguaje vectorizado permite ahorrar muchos bucles y simplifica la compacidad del código. Unitariamente, estos *vectores* se comportan como vectores normales frente a algunas operaciones sencillas: la suma de dos vectores suma sus elementos, la multiplicación de un vector por un escalar (es decir, un número) multiplica cada elemento por este número. Ciertas funciones aplicadas al vector se aplican unitariamente a cada elemento.

Creemos un **vector** mediante la función de combinación **c()** y estudiemos el resultado.

```
v <- c(1,2,7,10)      # creación de un vector
v
dim(v)                 # sin dimensión
length(v)               # su longitud
str(v)                 # en detalle
```

```
[1] 1 2 7 10
NULL
[1] 4
num [1:4] 1 2 7 10
```

A la vista de los resultados, el **vector** se ha creado. No tiene dimensión (**NULL**) y por lo tanto no es un vector en el sentido matemático del término. Su longitud es de cuatro elementos. Está compuesto de números (**num**).

Aplicaremos aquí algunas operaciones elementales.

```
2*v                  # producto sobre cada componente
2*v + 100            # todavía mejor
v^2                  # cuadrado sobre cada componente
```

```
[1] 2 4 14 20
[1] 102 104 114 120
[1] 1 4 49 100
```

Se obtienen los resultados esperados, veamos qué ocurre con dos **vectors**.

```
w <- c(-10,-20,-30,-100) # otro vector
v+w                  # suma miembro a miembro
v*w                  # ¡producto miembro a miembro! atención
```

```
[1] -9 -18 -23 -90  
[1] -10 -40 -210 -1000
```

La suma de dos **vectors** ha funcionado correctamente.

La multiplicación de dos **vectors no es una multiplicación vectorial clásica**, sino una multiplicación de los elementos miembro a miembro (esto se denomina producto de Hadamard).

Podemos realizar operaciones entre los elementos de un **vector**, aquí la suma y la media:

```
sum(v)          # suma de los elementos  
mean(v)        # media de los elementos
```

```
[1] 20  
[1] 5
```

A menudo necesitamos realizar un **producto escalar** de dos vectores. Veamos si podemos obtenerlo con dos vectors. Veremos más adelante distintos usos del producto escalar. He aquí dos maneras de calcular este producto escalar. No se sorprenda por la formulación `%*%`; en efecto, R permite definir operaciones específicas a tipos de objetos. En este caso, se enmarca el nombre de la operación con porcentajes. Aquí, la operación `%*%` aplicada a los **vectors** se ha definido por los diseñadores de R como otra forma de multiplicación (el producto escalar):

```
sum(v*w)        # producto escalar v, w  
v%*%w          # producto escalar v, w
```

```
[1] -1260  
[1,] -1260
```

Se confirma una pequeña diferencia en el resultado (la coma). Veamos lo que nos afecta.

```
is.vector(sum(v*w))    # es un vector  
is.vector(1789)         # un escalar simple es un vector  
is.vector(v%*%w)       # no es un vector sino  
is.matrix(v%*%w)        # ;una matriz!
```

```
[1] TRUE  
[1] TRUE  
[1] FALSE  
[1] TRUE
```

El primer cálculo devuelve el producto escalar como un **vector**; observe que un número cualquiera es un vector. El segundo no es un **vector**, sino una matriz con un único elemento que contiene el producto escalar.

A continuación veremos cómo aplicar nuestras propias funciones sobre **vectors** o matrices.

apply() caso 1: vector hacia vector (de vector matemático hacia vector)

A continuación vamos a aplicar una función cualquiera sobre cada elemento de un vector. **Esté muy atento a esta sintaxis**, que puede sorprender. La función **apply()** utilizada aquí forma parte del núcleo del lenguaje R.

En primer lugar, creemos una función cualquiera, aquí una función que realiza ciertas modificaciones sobre los valores.

```
f <- function(x){x^(1.001)-0.001} # una función cualquiera
f(10)                                # test de la función
```

```
[1] 10.02205
```

Esta función transforma un pequeño número de un valor muy próximo, que va a facilitarnos la localización de su aplicación en los resultados que siguen.

Para aplicar **f()** sobre cada elemento de **v**, que es un **vector**, y obtener un nuevo **vector**, utilicemos la siguiente sintaxis:

```
u <- apply(matrix(v),1,f) # aplicación sobre un vector
u
is.vector(u)              # es un vector
```

```
[1] 0.999000 2.000387 7.012635 10.022052
[1] TRUE
```

Veamos cómo funciona esto:

- La función **apply()** posee tres parámetros.
- Aquí hemos escogido aplicarla sobre una matriz, pues es más sencillo de comprender.
- El primer parámetro representa lo que se va a pasar a la función que se ha de aplicar. Hemos decidido aplicar la función sobre una matriz columna que se corresponde con nuestro vector **v**; en efecto, **apply()** es eficaz sobre estructuras que posean una o dos dimensiones, como las matrices y no como los vectores. En nuestra función, **apply()** va a aplicarse sobre el vector **matrix(v)**. **Es una aplicación de un caso: vector matemático hacia vector.**
- El segundo parámetro indica la dimensión, que puede valer **1, 2 o c(1, 2)** en función de si trabajamos sobre la primera, la segunda o las dos dimensiones de una matriz.
- El tercer parámetro representa la función que se va a aplicar, observe que no se indican los paréntesis.

De hecho, la función **apply()** es muy rica y potente, y aquí solo abordaremos una pequeña parte de su uso.

apply() caso 2: vector hacia vector matemático (de vector matemático hacia vector matemático)

Solo cambia el segundo parámetro.

```
u <- apply(matrix(v),2,f) # aplicación sobre un vector
u
is.vector(u)              # no es un vector sino
```

```
is.matrix(u)          # una matriz
```

```
[ ,1]
[1,] 0.999000
[2,] 2.000387
[3,] 7.012635
[4,] 10.022052

[1] FALSE
[1] TRUE
```

Se ha obtenido un vector columna. Estamos, de hecho, en un caso **vector matemático hacia vector matemático** aplicado al vector **matrix(v)**, que es el caso más «normal».

apply() caso 3: matriz hacia matriz

Tratemos aquí la aplicación sobre una matriz, que no es un vector, ni un traspuesto de un vector, es decir, cuya dimensión sea estrictamente superior a 1. Apliquemos nuestra función **f()** miembro a miembro sobre nuestra matriz **M**.

```
N <- apply(M,c(1,2),f)      # aplicación sobre una matriz
N
is.matrix(N)                  # es una matriz
```

```
[ ,1]   [ ,2]   [ ,3]
[1,] 40.14683  0.999 75846.64
[2,] 30.10121 -0.001 69773.09
[3,] 20.05900 -0.001 69773.09
[4,] 35.12366  0.999 75846.64

[1] TRUE
```

apply(): otros casos útiles con funciones genéricas R

A menudo, en la parte inferior de una tabla se incluye una fila y se realiza una suma, una media...

Veamos este tipo de comportamiento utilizando **apply()** sobre una matriz.

Obtengamos la media de cada columna de **M**:

```
t(matrix(apply(M,2,mean))) # media por columna
```

```
[ ,1]   [ ,2]   [ ,3]
[1,] 31.25  0.5 72000
```

Realicemos la media de cada fila de una matriz **T** (hemos escogido **T** como la traspuesta de **M**):

```
T <- t(M)
```

```

T
matrix(apply(T,1,mean))      # media por fila

```

La matriz **T**:

```

[,1]  [,2]  [,3]  [,4]
[1,]    40     30     20     35
[2,]     1      0      0      1
[3,] 75000 69000 69000 75000

```

Media por fila de **T**:

```

[,1]
[1,] 31.25
[2,] 0.50
[3,] 72000.00

```

Podemos utilizar este método para diversas funciones genéricas:

- **sum**: suma de varios números.
- **min**: mínimo de varios números.
- **max**: máximo de varios números.
- **stdev**: desviación típica de varios números.
- **prod**: producto de varios números.

Esto se corresponde con el siguiente código:

```

t(matrix(apply(M,2,sum ))) # suma de cada columna
t(matrix(apply(M,2,min ))) # min de cada columna
t(matrix(apply(M,2,max ))) # max de cada columna
t(matrix(apply(M,2,stdev )))# desviación típica de cada columna
t(matrix(apply(M,2,prod ))) # producto de términos por columna

matrix(apply(T,1,sum )) # suma de cada fila
matrix(apply(T,1,min )) # min de cada fila
matrix(apply(T,1,max )) # max de cada fila
matrix(apply(T,1,stdev ))# desviación típica de cada fila
matrix(apply(T,1,prod )) # producto de términos por fila

```

2. Matrices, vectores: una introducción a la noción de aprendizaje supervisado

En este párrafo, llamaremos **n** al número de filas de nuestra matriz de observaciones (número de hechos observables) y **p** a su número de columnas (número de atributos del problema). Como es un hábito, llamaremos **X** a la matriz de observaciones. Esta matriz es una matriz de dimensión **n x p**.

La forma de esta matriz X es:

$$\begin{pmatrix} x_{1,1} & \cdots & x_{1,p} \\ \vdots & \ddots & \vdots \\ x_{n,1} & \cdots & x_{n,p} \end{pmatrix}$$

El índice de fila varía entre 1 y n, vamos a llamar i a este índice.

El índice de columna varía entre 1 y p, vamos a llamar j a este índice.

Esto puede expresarse declarando que cada elemento de la matriz tiene la forma $x_{i,j}$ con:

$$1 \leq i \leq n \quad y \quad 1 \leq j \leq p$$

De una manera más compacta, escribiremos:

$$X = [x_{i,j}]_{\substack{1 \leq i \leq n \\ 1 \leq j \leq p}}$$

 La traspuesta de la matriz X, que se escribe permutando los respectivos roles de las filas y las columnas, se escribiría entonces:

$$X^T = [x_{j,i}]_{\substack{1 \leq i \leq n \\ 1 \leq j \leq p}}$$

Como los índices son convencionales, se llaman «mudos», esta matriz traspuesta puede escribirse también:

$$X^T = [x_{i,j}]_{\substack{1 \leq i \leq p \\ 1 \leq j \leq n}}$$

La primera columna de la matriz X es el vector:

$$\begin{pmatrix} x_{1,1} \\ \vdots \\ x_{n,1} \end{pmatrix} = x_1$$

La columna j de la matriz X es el vector:

$$\begin{pmatrix} x_{1,j} \\ \vdots \\ x_{n,j} \end{pmatrix} = x_j$$

La última columna de la matriz X es el vector:

$$\begin{pmatrix} x_{1,p} \\ \vdots \\ x_{n,p} \end{pmatrix} = x_p$$

Podemos escribir, entonces:

$$X = (x_1, \dots, x_j \dots, x_p)$$

como compuesta de vectores columnas que representan las columnas de atributos de nuestras observaciones. A menudo se considera una columna $p+1$ llamada por ejemplo y , que representa **la columna sobre la que se va a realizar alguna predicción**.

A continuación, se constatará que la idea general del aprendizaje supervisado es encontrar **una estimación** de la aplicación f , el «modelo predictivo», tal que se pueda deducir el vector y conociendo la matriz X : $y = f(X) + \text{error}(X)$ (tal que el error sea lo más pequeño posible, con una media de cero, por ejemplo).

A menudo se plantea la hipótesis de que el error no depende de X , de modo que es un error irreductible. Se expresará esto con la forma (recuérdelo): $y = f(X) + \varepsilon$

Observemos que realizando esta hipótesis se considera ahora que el vector y es una función de dos variables: la matriz X y el vector ε .

3. Ir más lejos en la manipulación de matrices con R

a. Operaciones básicas

Sumar o restar, como multiplicar por un escalar (es decir, por un número), resulta trivial:

```
M-N                                # resta miembro a miembro
```

```
[,1] [,2]      [,3]
[1,] -0.14682767 0.001 -846.6352
[2,] -0.10120964 0.001 -773.0863
[3,] -0.05900448 0.001 -773.0863
[4,] -0.12365865 0.001 -846.6352
```

```
2*M                                # multiplicación por 2 de los miembros
```

```
[,1] [,2]      [,3]
[1,] 80     2 150000
[2,] 60     0 138000
[3,] 40     0 138000
[4,] 70     2 150000
```

También es posible extraer una submatriz:

```
M[2:3,2:3]                            # extracción de una submatriz
```

```
[,1] [,2]
[1,] 0 69000
[2,] 0 69000
```

Y es posible cambiar los valores de un «rectángulo» de la matriz:

```
M[2:3,2:3] <- 2 * M[2:3,2:3] # reemplazo de variables en una  
M # parte de la matriz
```

```
[ ,1] [ ,2] [ ,3]  
[1,] 40 1 75000  
[2,] 30 0 138000  
[3,] 20 0 138000  
[4,] 35 1 75000
```

Veamos ahora cómo proceder con trabajos algo más pesados sobre matrices y vectores.

b. Algunos trucos útiles sobre las matrices de R

Para los siguientes cálculos, partamos de dos vectores (secuencias de números):

```
x <- matrix( seq( 0, 4 ) ) # los 5 primeros enteros  
print(x)  
y <- matrix( seq( 5, 9 ) ) # los 5 siguientes  
print(y)
```

```
[ ,1]  
[1,] 0  
[2,] 1  
[3,] 2  
[4,] 3  
[5,] 4  
  
[ ,1]  
[1,] 5  
[2,] 6  
[3,] 7  
[4,] 8  
[5,] 9
```

Para ir más lejos, el uso de la librería **matrixcalc** resulta útil.

```
library(matrixcalc) # para realizar cálculos matriciales
```

Esta librería nos ofrece la posibilidad de acceder a matrices clásicas ya compuestas, que nos pueden resultar útiles en diversos casos de uso.

Vamos a crear dos matrices para nuestros ejemplos de cálculos posteriores.

Aquí, una matriz cuadrada **H** con una subdiagonal de números de 1 a 4:

```
H <- creation.matrix( 5 ) # subdiagonal 1 to n-1  
print( H )  
is.matrix(H) # es efectivamente una matriz
```

```
[ ,1] [ ,2] [ ,3] [ ,4] [ ,5]
[1,] 0 0 0 0 0
[2,] 1 0 0 0 0
[3,] 0 2 0 0 0
[4,] 0 0 3 0 0
[5,] 0 0 0 4 0

[1] TRUE
```

Aquí, una matriz de Vandermonde, donde cada vector columna incluye una de las potencias sucesivas, partiendo de 0, de un vector concreto (aquí utilizaremos el vector de los cinco primeros números enteros no nulos; la primera columna está formada por 1, dado que la potencia 0 de cualquier número vale siempre 1).

```
V <- vandermonde.matrix(c(seq(1,5)),5) # matriz de Vandermonde
print( V )
```

```
[ ,1] [ ,2] [ ,3] [ ,4] [ ,5]
[1,] 1 1 1 1 1
[2,] 1 2 4 8 16
[3,] 1 3 9 27 81
[4,] 1 4 16 64 256
[5,] 1 5 25 125 625
```

Producto de Hadamard

El producto de Hadamard de dos matrices se emplea muy poco. No es el producto habitual utilizado entre matrices, aunque puede resultar útil para realizar un filtro o una máscara sobre una matriz (sobre una matriz de píxeles, por ejemplo).

Este producto entre las matrices A y B se denota a menudo A o B

```
# producto de Hadamard
# producto miembro * miembro: z_ij = v_ij * h_ij
Z <- hadamard.prod (V,H)
print(Z)
```

```
# es similar a
Z <- V * H           # producto de Hadamard
print(Z)
```

```
[ ,1] [ ,2] [ ,3] [ ,4] [ ,5]
[1,] 0 0 0 0 0
[2,] 1 0 0 0 0
[3,] 0 6 0 0 0
[4,] 0 0 48 0 0
[5,] 0 0 0 500 0
```

Producto matricial "clásico"

El «verdadero» producto matricial es este; observe que este operador producto se denota con `%*%` en R y a menudo se indica con un punto (o con nada) en matemáticas.

Para multiplicar dos matrices A y B mediante el producto matricial clásico, es preciso que sus dimensiones sean compatibles: si A es de dimensión n, p y B de dimensión m, q , es necesario que **p sea igual a m** para poder realizar la multiplicación. Esta multiplicación matricial consiste en obtener para cada posición de la nueva matriz la suma de los productos miembro a miembro de la fila y de la columna correspondientes de las matrices que se han de multiplicar:

$$AB = [a_{i,k}]_{\substack{1 \leq i \leq n \\ 1 \leq k \leq p}} \cdot [b_{k,j}]_{\substack{1 \leq k \leq p \\ 1 \leq j \leq q}} = \left[\sum_{k=1}^p a_{i,k} b_{k,j} \right]_{\substack{1 \leq i \leq n \\ 1 \leq j \leq q}}$$

```
# producto z_ij = suma_sobre_k(v_ik * h_kj) #
Z <- V %*% H # filas * columnas
is.matrix(Z) # es efectivamente una matriz
print(Z)
```

```
[ ,1] [,2] [,3] [,4] [,5]
[1,]    1     2     3     4     0
[2,]    2     8    24    64     0
[3,]    3    18    81   324     0
[4,]    4    32   192  1024     0
[5,]    5    50   375  2500     0
```

Si A es la matriz de una aplicación lineal f y x un vector que pertenece al conjunto de definición de f y, por tanto, de dimensión compatible: $A.x = f(x)$.

SI A es la matriz de una aplicación lineal f, y B la matriz de una aplicación lineal g, entonces AB es la matriz de una aplicación lineal $f \circ g$, es decir, de la aplicación sucesiva de f y a continuación de f a un vector. Con las condiciones adecuadas en términos de conjunto de definición, tenemos que: $AB.x = f(g(x)) = f \circ g(x)$.

Preste atención: en ciertos casos, las matrices pueden representar otra cosa diferente de las aplicaciones lineales habituales y corresponder con álgebra multilinear.

Este tipo de producto funciona también entre vectores, y entre matrices y vectores, siempre que sus dimensiones sean compatibles.

El producto de un vector columna por un vector fila devuelve una matriz:

```
x%*%t(y) # se obtiene una matriz
```

```
[ ,1] [,2] [,3] [,4] [,5]
[1,]    0     0     0     0     0
[2,]    5     6     7     8     9
[3,]   10    12    14    16    18
[4,]   15    18    21    24    27
[5,]   20    24    28    32    36
```

El producto del traspuesto de un vector por otro se denomina también producto escalar; posee la propiedad de ser nulo cuando ambos vectores son perpendiculares.

```
t(x)%%y          # producto escalar  
t(y)%%x          # ídem
```

```
[ ,1]  
[1,] 80
```

```
[ ,1]  
[1,] 80
```

La raíz cuadrada del producto escalar de un vector por sí mismo es igual a su norma euclíadiana $\|x\|_2$ (vea el final de la sección).

```
sqrt(t(x)%%x)      # norma euclíadiana
```

```
[ ,1]  
[1,] 5.477226
```

Observe que se han obtenido matrices 1×1 compuestas de un escalar único; es de hecho el contenido de esta minimatriz el verdadero resultado devuelto por la operación (i admitamos que en la práctica esto no supone una diferencia!).

 Existen otras operaciones que en ocasiones resultan útiles entre las matrices. Pruebe las tres operaciones siguientes, cuyos resultados son algo engorrosos como para mostrarlos por escrito en este libro.

Suma directa entre vectores

La suma directa de dos vectores se indica, a menudo, con el símbolo \oplus .

```
# suma directa de matrices          #  
# es una operación por bloque, a partir de las matrices A y B #  
# se obtiene una diagonal AB y el resto de los valores a 0    #  
Z <- direct.sum( x, y)      # suma "directa"  
print(Z)                  # dos columnas de dim(10)
```

Suma directa entre matrices

La suma directa de dos matrices se indica, a menudo, con el símbolo \oplus .

```
Z <- direct.sum( V, H)      # suma "directa"  
print(Z)                  # 10 x 10
```

Producto de Kronecker

El producto de Kronecker se indica, a menudo, con el símbolo \otimes .

```
# producto de Kronecker = producto directo (es un p tensorial) #
```

```

# cada valor aij de la primera matriz A se reemplaza          #
# por el producto de este valor y de toda la matriz B          #
# es decir [aij * B]                                         #

Z <- direct.prod( x, y)      # producto "directo"
print(Z)                  # 1 columna de dim(25)
                          # x11 * y
                          # x21 * y ...
                          # x31 * y ...

Z <- direct.prod( V, H )    # producto "directo"
print(Z)                  # 25 x 25
                          # v11 * H , v12 * H ...
                          # v21 * H , v22 * H ...
                          # v31 * H ...

```

c. Normas de vectores y normas de matrices

Las normas son funciones con valores positivos que permiten comparar objetos entre sí.

Normas de vectores

Cuando tenemos varios vectores, o varias matrices sobre un mismo espacio, podríamos querer clasificar estos objetos entre sí, encontrar los extremos, establecer y comparar proporciones...

 Por ejemplo, imagine que ha creado dos modelos predictivos de y a partir de X : $y = f_1(X) + \varepsilon_1 = f_2(X) + \varepsilon_2$

Nos gustaría poder escoger entre ambos modelos. Una primera idea podría ser «tomemos el que tenga un error menor». Sí, pero, ¿cuál entre ε_1 o ε_2 es más pequeño?

 Para hacer esto, se va a utilizar una norma $\|\cdot\|$ y se podrá ver si $\|\varepsilon_1\| < \|\varepsilon_2\|$ o no. La elección de la norma puede influir en nuestra respuesta, pero afortunadamente, respeto a este estricto aspecto de comparación, existe una notación equivalente de normas que hace que ciertas normas -aunque no todas- sean equivalentes y que podamos, en ciertos casos, escoger la más sencilla de manipular sin demasiada pérdida de sentido.

La más conocida de las normas es la «norma euclíadiana de un vector», que se corresponde con la **longitud de este vector** en un espacio euclidiano (por ejemplo, en el espacio geométrico de tres dimensiones en el que vivimos). Cuando se dispone de una base ortonormal (dos vectores unitarios perpendiculares entre sí en geometría plana, por ejemplo), el cálculo de esta norma es una aplicación sencilla del viejo teorema de Pitágoras.

Las normas se indican, a menudo, de la siguiente manera: $\|x\|$

Pero como existen muchas otras normas además de la norma euclíadiana, en ocasiones se agregan índices para evitar cualquier ambigüedad.

La norma euclíadiana de un vector x se escribe también $\|x\|_2$ (en referencia al cuadrado en la expresión que aparece más abajo).

Para un vector $x=[X_i]$ en n dimensiones, cuyas coordenadas estén expresadas en una base ortonormal, la norma euclíadiana es la raíz cuadrada de la suma de los cuadrados de las n coordenadas del vector:

$$\|x\|_2 = \sqrt{\sum_{i=1}^n x_i^2}$$

Existen a nuestra disposición numerosas normas de vector.

La clase más importante de normas, en dimensión finita, a la que pertenece $\|\cdot\|_2$ tiene como expresión general:

$$\|x\|_p = (\sum_i |x_i|^p)^{1/p}.$$

Cuando p es infinito, la expresión cambia y esta norma se convierte en el máximo de los $|x_i|$.

Veamos los distintos valores de estas normas para un vector diagonal de dimensión 3.

```
# normas de vectores

z <- matrix(c(1,1,1))           # vector diagonal 3D
z

n1 <- entrywise.norm( z, 1 )     # p= 1
n2 <- entrywise.norm( z, 2 )     # norma euclidiana p= 2
n10 <- entrywise.norm( z, 10 )    # p= 10
n_i <- maximum.norm(z)          # norma max      p= infinito
print (c(n1,n2,n10,n_i))
```

[1] 3.000000 1.732051 1.116123 1.000000

Comparemos con los valores de estas normas para un vector de base ortonormal.

```
z <- matrix(c(0,1,0))           # vector de la base
z

n1 <- entrywise.norm( z, 1 )     # p= 1
n2 <- entrywise.norm( z, 2 )     # norma euclidiana p= 2
n10 <- entrywise.norm( z, 10 )    # p= 10
n_i <- maximum.norm(z)          # norma max      p= infinito
print (c(n1,n2,n10,n_i))
```

[1] 1 1 1 1

Comprobamos que la aplicación de estas normas cambia la topología del espacio vectorial, ipues las proporciones entre un vector de la base ortonormal y un vector cualquiera no son las mismas según la norma!

Por ejemplo, si quiero obtener todos los puntos del espacio afín situados a una cierta distancia de un punto determinado, no obtengo los mismos puntos de una norma a otra.

Pero, de hecho, ¿qué es una distancia?

Distancias

A partir de las normas podemos definir distancias. Por ejemplo, a partir de la norma euclidiana en el espacio

geométrico 3D en el que estamos sumergidos, podemos definir la distancia entre dos puntos **a** y **b**. Estos dos puntos pueden definirse como vectores construidos vinculando el origen de un punto de referencia y los puntos de nuestro espacio afín. De hecho, para ser exactos, sería más correcto afirmar que todo espacio vectorial puede estar dotado de una estructura de espacio afín por la operación de resta vectorial.

En nuestro espacio tenemos que:

$$\text{distancia } (a, b) = \sqrt{(x_b - x_a)^2 + (y_b - y_a)^2 + (z_b - z_a)^2} = \|b - a\|_2$$

De manera análoga, las distancias inducidas por las normas son de la forma $\|b - a\|$, sea cual sea la definición de la norma y la naturaleza de los objetos **a** y **b**.

En nuestra práctica de data science, la elección de una distancia puede tener bastante impacto. El ejemplo del **clustering**, es decir, la búsqueda de clases de hechos descubriendo montones de observaciones, es manifiestamente sensible a la distancia seleccionada, pues según la naturaleza de las distancias la topología del espacio es diferente.

Las normas utilizadas desde hace tiempo tienen nombres, por ejemplo:

$\|b - a\|_1$ se denomina distancia de Manhattan.

$\|b - a\|_2$ se denomina distancia euclídea.

 Ciertas distancias no se deducen de normas, aunque solo las normas permiten definir distancias. Algunas cantidades se denominan distancias de manera abusiva, pues no poseen todas las características de una distancia. Se las utiliza aun así en ciertos algoritmos, pues demuestran cierta eficacia vinculada a la naturaleza de algún problema particular. Si no posee información particular acerca del contexto, utilice las verdaderas distancias que conservan la desigualdad triangular. Es decir, tales que la distancia entre dos objetos sea siempre más pequeña o igual a la suma de las distancias correspondientes al hecho de pasar por un tercer objeto.

Normas de matrices

Podemos definir fácilmente normas sobre las matrices considerando una matriz sin sus dimensiones y aplicándole una definición de norma vectorial (en efecto, si se considera las matrices como vectores columnas que estarían construidos como una lista monodimensional de todos los elementos de estas matrices, podemos definir un espacio vectorial que justifica este procedimiento).

Para calcular dicha norma de la matriz **X**, aplicamos la aplicación **vec()** a la matriz para obtener un vector, y a continuación aplicamos una norma de vector:

$$\|X\|_p = \|\text{vec}(X)\|_p$$

La función **entrywise.norm()** nos permite calcular estas normas:

```
# normas de matrices #  
  
# Frobenius (generalización: Hilbert- Schmidt)  
print(frobenius.norm( V ))      # devuelve: 696.084  
  
# que es un caso particular de  
print( entrywise.norm( V, 2 ) )  # que también devuelve 696.084
```

```
# norma máxima
print(maximum.norm(V))           # devuelve 625

# que es un caso particular de
print( entrywise.norm( V, Inf ) )# que también devuelve 625
```

Destacaremos que existen otras muchas normas de matrices, como por ejemplo la norma espectral:

```
spectral.norm(V)               # devuelve 695
```

 Preste atención: ciertas normas se escriben de la misma manera por distintos autores o en distintas circunstancias. Compruebe siempre de qué norma se trata antes de implementar un algoritmo. Además, ciertas normas son muy costosas en tiempo de cálculo.

d. Matrices y vectores: diversas sintaxis útiles

Transformar una matriz en vector (en sentido R) o vector (matemático)

Cuando una función no soporta el tipo «matrix» como entrada, hay que transformar a menudo una matriz o un vector matemático en un vector (en sentido R).

```
c <- as.vector(H)
print(c)  # perdemos las dimensiones y ya no es una matriz
```

```
[1] 0 1 0 0 0 0 0 2 0 0 0 0 0 3 0 0 0 0 0 4 0 0 0 0 0
```

En matemáticas, es la aplicación **vec()** la que transforma una matriz en un vector columna, cuya codificación en R resulta trivial:

```
c <- matrix(H)
head(c)          # los cinco primeros elementos del vector columna
```

```
[ ,1]
[1,]    0
[2,]    1
[3,]    0
[4,]    0
[5,]    0
[6,]    0
```

Trasposición de una matriz, de una tabla, de un data.frame

Antes hemos utilizando la función **t()** para realizar una trasposición; esta función también funciona sobre los data.frames. La función **aperm()** es mucho más general, traspone las matrices pero también puede invertir las dimensiones de tablas de más de dos dimensiones.

```
# trasposición de matrices, tablas, data.frames
Tv <- aperm(V, c(2, 1)) # trasposición de dimensiones
Tv <- t(V)           # ídem propio de las matrices o data.frames
print(Tv)
```

Diagonal de una matriz, identidad, triángulos

Consideremos una matriz cuadrada, es decir, cuyo número de columnas sea igual al número de filas; los términos de la matriz tales que el índice de la fila es igual al índice de la columna forman la diagonal de la matriz.

```
d <- diag(V)      # diagonal de la matriz
print(d)          # vector fila en sentido R
is.matrix(d)      # no es una matriz!!!
```

```
[1] 1 2 9 64 625
[1] FALSE
```

Formemos una matriz con esta diagonal:

```
Dv <- diag(diag(V)) # matriz con la diagonal de V
print(Dv)           # ¡¡Por fin!!
```

```
[,1] [,2] [,3] [,4] [,5]
[1,] 1 0 0 0 0
[2,] 0 2 0 0 0
[3,] 0 0 9 0 0
[4,] 0 0 0 64 0
[5,] 0 0 0 0 625
```

Esto nos permite reemplazar la diagonal de V por ceros.

```
V0 <- V-Dv          # reemplazo de la diagonal de V por 0
print(V0)
```

```
[,1] [,2] [,3] [,4] [,5]
[1,] 0 1 1 1 1
[2,] 1 0 4 8 16
[3,] 1 3 0 27 81
[4,] 1 4 16 0 256
[5,] 1 5 25 125 0
```

La matriz identidad, que es el elemento neutro de la multiplicación de matrices ($AI = IA = A$) se obtiene fácilmente para una dimensión determinada:

```
# obtener la matriz identidad
I5 <- diag(1,5)      # matriz de 5 x 5 con 1 en la diagonal
```

```
print(I5)          # de dimensión 5
```

```
[ ,1] [,2] [,3] [,4] [,5]
[1,]    1    0    0    0    0
[2,]    0    1    0    0    0
[3,]    0    0    1    0    0
[4,]    0    0    0    1    0
[5,]    0    0    0    0    1
```

También podemos extraer el triángulo superior o inferior:

```
# triángulos superior e inferior de la matriz

Vl <- lower.triangle(V) # triángulo por debajo de la diagonal
print(Vl)
Vu <- upper.triangle(V) # triángulo por encima de la diagonal
print(Vu)
Vbis <- Vl + Vu - Dv      # reconstrucción de la matriz
```

Triángulo inferior, con diagonal:

```
[ ,1] [,2] [,3] [,4] [,5]
[1,]    1    0    0    0    0
[2,]    1    2    0    0    0
[3,]    1    3    9    0    0
[4,]    1    4   16   64    0
[5,]    1    5   25  125  625
```

También se utiliza a menudo la traza (suma de los términos de la diagonal):

```
# cálculo de la traza
matrix.trace(V) # suma de los términos de la diagonal
```

```
[1] 701
```

Determinantes, valores propios, inversión de matrices

Cuando el determinante de una matriz A no es nulo es posible calcular su inversa A^{-1} tal que $A \cdot A^{-1} = A^{-1} \cdot A = I$:

```
# determinante de una matriz
print(det(V)) # no nulo
```

```
[1] 288
```

Para determinar si una matriz es singular, existe una función muy rápida:

```
# ¿esta matriz es singular?
is.singular.matrix( H )      # sí
is.singular.matrix( V )      # no
```

Calculemos la inversa de una matriz.

```
# inversa de una matriz no singular
Vi <- matrix.inverse(V)
print(Vi)
solve(Vi) # si no se utiliza el paquete matrixcalc
           # se dispone también de una función equivalente
           # que parece menos fiable aquí: solve
```

Podemos comprobar que el producto de la matriz y de su inversa devuelve la identidad, aunque el resultado es aproximado, de modo que vamos a redondear y comprobar que la norma de la matriz a la que se sustrae la identidad es cercana a cero. **Este método debe recordarse**, pues no siempre se puede afirmar que una matriz es igual a otra, pero sí se puede comprobar fácilmente si son más o menos iguales.

```
print(V %*% Vi) # próxima a la identidad
# ¿la norma de la diferencia con la matriz identidad es nula?
print(round(frobenius.norm( V %*% Vi - I5), digits = 11)) # 0
```

```
[,1]      [,2]      [,3]      [,4]      [,5]
[1,] 1.000000e+00 -1.387779e-16  0 5.273559e-16 -2.081668e-17
[2,] 2.331468e-15  1.000000e+00  0 3.108624e-15  1.443290e-15
[3,] 1.776357e-15  1.421085e-14  1 0.000000e+00  3.552714e-15
[4,] 5.329071e-15  2.131628e-14  0 1.000000e+00  5.329071e-15
[5,] 7.105427e-15  2.842171e-14  0 1.421085e-14  1.000000e+00

[1] 0
```

Este cálculo nos permite comprobar que los valores tomados de manera global tienden «colectivamente» a 0 (observación: en el lenguaje R, un valor como **1.42 e-15** significa $1,42 \cdot 10^{-15}$, lo cual es infinitamente pequeño, e incluso no interpretable debido a los límites de codificación de su ordenador; por otro lado, **round()** es la función de redondeo de R).

Diagonalización de matrices, valores propios, vectores propios

En el capítulo Feature Engineering, describimos el principio y uno de los usos de la diagonalización de una matriz, el análisis en componentes principales (o PCA en inglés). Veamos aquí cómo transformar en R una matriz en matriz diagonal encontrando una nueva base (la base de vectores propios) asociada a los valores propios de la matriz.

```
# valores propios y vectores propios de una matriz
V_eigen<- eigen(V) # la función vectores propios
print(V_eigen)       # resultado a descomponer
```

La aplicación de la función **eigen()** produce un objeto compuesto que hay que descomponer:

```

$values
[1] 680.9266583 17.6584861 1.9730603 0.4123561 0.0294392

$vectors
[,1]      [,2]      [,3]      [,4]      [,5]
[1,] -0.002133113 -0.08865214 0.66256386 -0.85603382 .../...
[2,] -0.026814798 -0.33857978 0.65963083 0.27225055
.../...

```

Extraigamos en primer lugar la matriz diagonal ordenada por valor propio:

```

V_valp <- diag(V_eigen$values) # matriz diagonal con los
print(V_valp)                 # valores propios clasificados

```

```

[,1]      [,2]      [,3]      [,4]      [,5]
[1,] 680.9267 0.00000 0.00000 0.0000000 0.0000000
[2,] 0.00000 17.65849 0.00000 0.0000000 0.0000000
.../...

```

A continuación una matriz donde cada columna sea el vector propio correspondiente al valor propio:

```

V_vectp <- V_eigen$vectors # matriz de vectores propios
print(V_vectp)

```

```

[,1]      [,2]      [,3]      [,4]      [,5]
[1,] -0.002133113 -0.08865214 0.66256386 -0.85603382 .../...
[2,] -0.026814798 -0.33857978 0.65963083 0.27225055

```

Ahora que disponemos de las herramientas matemáticas y estadísticas básicas, podemos abordar nuestro problema de estimación con cierto rigor.

Estimaciones

1. Planteamiento del problema de estimación

a. Formulación general del problema

Hemos expuesto el aprendizaje supervisado mediante la siguiente expresión: $y = f(X) + \varepsilon$.

En el caso que nos ocupa, una gran cantidad de elementos del vector y son desconocidos, la aplicación $f()$ debe determinarse y el vector de error debería tener una media de 0 y una norma pequeña respecto a la norma de y .

En estas condiciones, trataremos de producir una estimación de f , que denominaremos \hat{f} , y una estimación de y , que denominaremos \hat{y} , tales que:

$$\hat{y} = \hat{f}(X)$$

Nuestro objetivo es producir una mecánica para encontrar $\hat{f}()$ tal que una «forma» de diferencia entre y e \hat{y} sea mínima.

Para expresar la diferencia global que hay que minimizar, nos dotaremos de una aplicación para medir la diferencia o «pérdida» (*loss* en inglés). La diferencia se calcula entre un valor conocido y un valor estimado para una observación determinada.

Llamaremos $\ell(\cdot, \cdot)$ a esta aplicación que, cuando los y_i son valores de \mathbb{R} , es una aplicación de \mathbb{R}^2 hacia \mathbb{R}^+ .

$\ell(y_i, \hat{y}_i)$ podría tomar naturalmente la forma de una distancia inducida por una norma o una función monótona sobre \mathbb{R}^+ de una distancia inducida por una norma.

Podemos, por ejemplo, tratar de encontrar \hat{f} tal que la media de $\ell(y_i, \hat{y}_i) = \text{distancia}(y_i, \hat{y}_i)$ sea mínima, o que la media de sus cuadrados lo sea (observe que no hemos definido aquí la distancia seleccionada).

► Vamos más bien a buscar una función tal que la esperanza matemática de $\ell(\cdot, \cdot)$ sea mínima.

Esta búsqueda se realizará sobre las observaciones del conjunto de entrenamiento (training set: T). Se acostumbra a llamar «riesgo» (R) a la esperanza matemática de $\ell(\cdot, \cdot)$.

Prácticamente, solo se accede con facilidad al cálculo **empírico** del riesgo cuya esperanza matemática es la media sobre los elementos del conjunto de entrenamiento (en lugar de ser una integral), lo que produce, reemplazando \hat{y} por su valor y denotando a la media de las observaciones del training set con

$$< >_T: R(X_T, y_T, \hat{f}) = < \ell(y_i, \hat{y}_i) >_T$$

Esto se lee: «el riesgo R empírico es una función de las observaciones (X_T, y_T) del "Training" set T y de la estimación de f. R es igual a la media sobre T de $\ell(\cdot, \cdot)$ ».

El método que debemos implementar consiste en **encontrar la función \hat{f} que minimice R** sobre las observaciones del conjunto de entrenamiento T. La ley de los grandes números, bajo ciertas condiciones derivadas del marco de esta obra, garantiza que esta estimación del riesgo R converge hacia la esperanza matemática de la función de pérdidas que representa el «verdadero riesgo». Podemos resumir dicho problema exhibiendo la función **argmin()**, que significa simplemente «encontrar el valor que minimiza esta expresión». Como R depende de hecho del conjunto de entrenamiento T, y vamos a hacer variar f hasta que **consideremos haber encontrado un mínimo para R, f será entonces la \hat{f} buscada**.

Esto se indica de la siguiente manera:

$$\hat{f} = \operatorname{argmin}_f (R_T(f))$$

En resumen: *Para poder comparar dos desviaciones, conviene tomar la media, la esperanza matemática. Así es como definimos la función «riesgo».* Esto resulta útil, pues podemos entrenar el modelo sobre un conjunto de prueba con determinada población para encontrar un modelo óptimo \hat{f} , y probarlo a continuación o utilizarlo de manera operacional sobre poblaciones diferentes.

En función del contexto, encontrará distintos términos con perfiles muy similares a nuestras funciones de riesgo y de pérdida: *loss, cost, reward, objective function...* ¡no se deje impresionar!

 Observación importante: a menudo escogeremos un aspecto particular para \hat{f} , que se corresponde con la selección del «modelo». Podemos, por ejemplo, intentar un modelo multilineal, polinomial... En este caso, el modelo está fijado (provisionalmente), y nuestro problema se reduce a encontrar **una estimación de los parámetros** de \hat{f} . En efecto, cuando dispongamos de estos parámetros, dispondremos de \hat{f} .

Hemos expresado nuestro problema a través **de una aplicación f , que una vez aplicada a una matriz «completa» nos devuelve un vector «completo»**. Esta es una visión muy general y observará **que difiere de la visión que a partir de una fila única de la matriz nos devuelve una componente única del vector**. En efecto, podemos diseñar modelos que utilicen varias filas de X (e incluso todas) para producir una estimación de una componente concreta del vector y .

En muchos casos, encontrará la mención de una función «fila a fila» bajo el nombre de **función hipótesis**. El hecho de que esta función se aplique fila a fila permite realizar numerosos cálculos de derivación parcial respecto a los parámetros de un modelo determinado.

b. Aplicación y reformulación del problema de estimación

La lectura atenta de este párrafo **puede permitirle sobrevivir** a la confusión introducida por las diferencias de formulación en los libros y los artículos que recorrerá en su vida de data scientist. En efecto, el problema que encontrará no se limita al hecho de traducir la notación de un autor a otro; esta notación le dará indicaciones acerca de las hipótesis y la naturaleza de los objetos manipulados por cada autor y que no siempre se corresponde con lo que parece a primera vista.

Introducción sintáctica de la función hipótesis

La traducción fila a fila de $\hat{y} = \hat{f}(X)$ nos obliga a introducir una función \hat{f}_i para cada una de las filas (elimine mentalmente el índice i de f para comprender que este índice es importante, ipues en caso contrario cada fila tendría la misma expresión!):

$$\hat{y}_i = \hat{f}_i(X)$$

En el caso simple donde \hat{y}_i solo depende de la fila correspondiente de X , podemos escribir:

$$\hat{y}_i = h(x_{i,:})$$

donde $x_{i,:}$ significa fila i de X , que se llama a menudo simplemente x_i si sabe que estamos hablando de las filas de X y no de las columnas de X .

En este caso

$$\hat{f}_1 = \hat{f}_2 = \hat{f}_3 = \dots = h,$$

siendo h la función hipótesis.

La tabla de nuestras observaciones resulta ser la matriz $Z = (X, y)$, cada fila z_i es la $(p+1)$.tupla (x_i, y_i) . Podemos redefinir la función de pérdida como dependiente de h y aplicándosela a z_i :

$$z_i : \ell_h(z_i) = \ell(y_i, h(x_i)).$$

Con la misma idea, llamaremos $R(h)$ a la función «verdadera» de riesgo de h , y $\hat{R}(h)$ a la función empírica sobre el conjunto de entrenamiento.

Teorema «No free lunch»

Los problemas de optimización que consisten en encontrar un argmin o un argmax mediante un algoritmo (es decir, de forma puramente algorítmica) se dividen en dos tipos: las optimizaciones de función (por ejemplo: encontrar el mínimo de una función convexa en un espacio de n dimensiones mediante el método de descenso por gradiente) y optimizaciones estocásticas que consisten en buscar un óptimo de la esperanza matemática de un fenómeno (por ejemplo: la búsqueda del mínimo de nuestra verdadera función de riesgo).

Tanto en un caso como en otro, resulta imposible afirmar si existe una correlación real entre la eficiencia de la optimización producida (realizada sobre un conjunto de entrenamiento en el caso estocástico: training set) y la generalización que podríamos realizar de su eficacia (error débil) en el caso general. Si bien todo modelo que funciona bien y que optimiza una función hipótesis sobre un conjunto de entrenamiento está de hecho ligado a una serie de hipótesis implícitas realizadas sobre la naturaleza del fenómeno estudiado.

Esta es una mala noticia para los tecnócratas o los «operativos» que imaginan que es posible abordar todos los problemas con la misma artillería; sin embargo, es una buena noticia para los apasionados como nosotros: cada problema estocástico requiere trabajo y una reflexión sobre la función de riesgo utilizada y sobre el modelo que se ha de implementar.

Esto se opone a la idea común del machine learning, que consistía en trabajar solamente sobre la búsqueda y el condicionamiento de features y en confiar la predicción o la clasificación a un conjunto de algoritmos fijos y determinados por otros conforme avanzaba la tecnología.

No free lunch = nada es gratis.

Aplicación al caso de un modelo de regresión lineal simple

Tomemos el caso de un modelo lineal simple. La matriz X se reduce a una columna. La forma de $h()$ es la siguiente: $h(x) = ax + b$.

Estamos en el caso de que existe una única «feature» como entrada y queremos encontrar una recta de regresión simple que aproxime la nube de puntos sobre el plano.

Los parámetros de $h()$ son a y b , y el esfuerzo de estimación se realizará sobre estos parámetros a y b .

La función \hat{f} , por su parte, tiene **exactamente el mismo aspecto visual que h** (y, por otro lado, se le da a menudo el mismo nombre por abuso del lenguaje), de modo que:

$$\hat{f}(\vec{X}) = \vec{y} = \hat{a}\vec{X} + \vec{b}$$

\vec{b} es un vector columna cuyas componentes valen b.

Cuando piensan que no existe ambigüedad, algunos autores no especifican que \vec{X} , \vec{y} y \vec{b} son vectores y omiten las flechas. Otros, de cultura anglosajona, utilizan distintas tipografías, o escriben los vectores en negrita...

Aquellos que amen el cálculo matricial, entre ellos el autor de este libro, introducen a menudo un vector unidad (un 1 en negrita): **1**. Este vector es una columna de «1». Esto produce algo así:

$$\hat{f}(X) = \hat{y} = \hat{a} \cdot X + \hat{b} \cdot \mathbf{1}$$

o bien:

$$\hat{f}(X) = \hat{y} = \hat{a}X + \hat{b}\mathbf{1}$$

En ocasiones, para evitar cualquier ambigüedad, le recordamos que X y **1** poseen las siguientes formas, en función de las notaciones utilizadas más arriba en este libro; los corchetes son, a veces, paréntesis:

$$\begin{aligned} X &= [x_i]_{1 \leq i \leq n} \\ \hat{y} &= [\hat{y}_i]_{1 \leq i \leq n} \\ \mathbf{1} &= [1]_{1 \leq i \leq n} \end{aligned}$$

Aunque encontrará otras notaciones, que poseen sus ventajas y sus inconvenientes... El karma del data scientist es tal que no se deja perturbar por estos detalles visuales, sino que verifica siempre cuidadosamente el sentido de la notación propuesta, como se le ha recordado en numerosas ocasiones a lo largo de este libro. Observe, sin embargo, que en función del problema se utiliza en ocasiones una notación diferente por motivos históricos o debido a un hábito extendido que ayuda a cada uno a fijarse en el contexto preciso según el asunto abordado.

Aplicación al caso de un modelo de regresión lineal múltiple

En el modelo de regresión lineal múltiple, no tenemos una única columna de features, sino varias: X ya no es un vector, sino una matriz.

$$\hat{f}(X) = \hat{y} = X \cdot \hat{a} + \hat{b}\mathbf{1}$$

$$X = [x_{ij}]_{\substack{1 \leq i \leq n \\ 1 \leq j \leq p}}$$

$$\hat{y} = [\hat{y}_i]_{1 \leq i \leq n}$$

$$\mathbf{1} = [1]_{1 \leq i \leq n}$$

$$\hat{a} = [\hat{a}_i]_{1 \leq i \leq p}$$

Preste atención. **observe que la dimensión del vector \hat{a}** es el número de features, es decir p.

A nivel de cada fila, tenemos la función hipótesis: $h(x_i) = \sum_{j=1}^p x_{ij} \hat{a}_j + \hat{b}$

Si renombramos \hat{b} por \hat{a}_{p+1} , esto da:

$$h(x_i) = \sum_{j=1}^p x_{ij} \hat{a}_j + \hat{a}_{p+1}$$

Que puede escribirse

$$h(x_i) = \sum_{j=1}^{p+1} \hat{x}_{ij} \hat{a}_j$$

definiendo \hat{X} como una matriz formada por una columna de 1 agregada a X :

$$\hat{X} = [x_1, x_2, \dots, x_j, \dots, x_p, 1]$$

con:

$$x_j = \begin{pmatrix} x_{1,j} \\ \vdots \\ x_{n,j} \end{pmatrix}$$

Como ocurre con cualquier matriz, nuestra nueva matriz puede expresarse de forma genérica en función de sus miembros:

$$\hat{X} = [\hat{x}_{ij}]_{\substack{1 \leq i \leq n \\ 1 \leq j \leq p+1}}$$

Con esta nueva matriz aumentada con una columna, se obtiene ahora una formulación más compacta, pero que puede introducir una cierta confusión si confunde X y \hat{X} .

$$\hat{f}(X) = \hat{y} = \hat{X} \cdot \hat{a}$$

con:

$$\hat{a} = [\hat{a}_i]_{1 \leq i \leq p+1}$$

Por otro lado, en la práctica los autores escriben a menudo $y = X.a + \varepsilon$ sin poner de relieve que X no es la matriz de las variables explicativas.

Regresión lineal múltiple y método de los mínimos cuadrados

A continuación vamos a estudiar rápidamente la resolución de nuestro modelo de regresión lineal múltiple. El método de resolución empleado aquí se denomina «método de los mínimos cuadrados» (observe que la regresión lineal simple por el método de los mínimos cuadrados se deduce de manera trivial).

 Importante de recordar: como regla general, encontrar la estimación de nuestras funciones f conformes a un modelo determinado consiste, simplemente, en buscar las estimaciones de los parámetros de esta función tal y

como están definidos por el modelo.

Buscamos: $\hat{f} = \operatorname{argmin}_f (R_T(f))$

Pero como hemos escogido el modelo de f , **esto equivale a buscar una estimación de sus parámetros**, de ahí que nuestro problema se reduzca a buscar:

$$\hat{a} = \operatorname{argmin}_a (R_T(a))$$

con:

$$f(X) = y = X \cdot a + \epsilon$$

y:

$$\hat{f}(X) = \hat{y} = X \cdot \hat{a}$$

Donde hemos reportado el error de estimación de a como estimación del error sobre y .

Sabiendo que el método de los mínimos cuadrados consiste en plantear en primer lugar:

$$\ell(y_i, \hat{y}_i) = |y_i - \hat{y}_i|^2 = (y_i - \hat{y}_i)^2$$

Se obtiene la siguiente expresión que hay que minimizar:

$$\frac{1}{n} \| y - \hat{y} \|_2^2$$

Una pequeña demostración fuera del propósito de este libro da una solución analítica, sabiendo que también es posible buscar esta solución mediante un algoritmo dedicado a la búsqueda de un óptimo, como por ejemplo el método del descenso por gradiente.

$$\hat{a} = (\hat{X}^T \hat{X})^{-1} \hat{X}^T y$$

Esta solución está construida sobre el conjunto de entrenamiento, lo que explica que conozcamos y !

En caso de que los datos estén centrados (es decir, que se les haya restado su media) o reducidos (datos centrados divididos por su desviación típica), el producto de la traspuesta de la matriz de observaciones por sí misma es, respectivamente, la matriz de covarianza o la matriz de correlación entre las features.

- Como regla general, los algoritmos operacionales no implementan directamente las soluciones analíticas cuando estas suponen operaciones bastante costosas o peligrosas, como la inversión de una matriz, aunque esto no le impide utilizar directamente una solución analítica en el caso de un prototipado para «sentir» mejor y comprender las condiciones y el impacto de sus decisiones de cara a la solución. De hecho, antes de manipular una matriz para este tipo de cálculo, a menudo resulta útil realizar diversas comprobaciones, que llamamos el buen **condicionamiento de la matriz**, pues ciertos cálculos pueden tomar valores extremos que deformen la lógica de los siguientes cálculos.

Para saber si su **problema está bien condicionado**, puede injectar pequeñas **perturbaciones aleatorias** negativas y positivas, de valor superior a la precisión de su equipo, sobre sus datos de entrada y comprobar que sus resultados no cambian de manera significativa.

Para recordar: la expresión que se debe minimizar posee **un único mínimo** cuando se han hecho variar todas las componentes de \vec{a} , se trata de una **función convexa** de \vec{a} . Con otra función de riesgo, basada en otra distancia, no se tiene siempre la misma «suerte» y es necesario realizar la búsqueda del mínimo mediante una técnica de optimización numérica.

Por otro lado, este método requiere que se cumplan ciertas hipótesis, tres de las cuales le parecerán evidentes:

- Hace falta un número de observaciones (netamente) más grande que el número de variables.
- No debe haber dos variables colineales.
- La desviación típica del error obtenido no debe depender de los valores de las observaciones: hipótesis de **homocedasticidad** que puede comprobar utilizando el **test de Breusch-Pagan**.

Existe un teorema que sale a nuestro auxilio para limitar la dificultad que supone comprobar estas hipótesis. Su aplicación nos recuerda, por otro lado, que **no hay que buscar conformarse con hipótesis inútiles**: el teorema Gauss-Markov que se aplica a los modelos lineales.

Este teorema nos dice que, si nuestros errores poseen una esperanza matemática nula (media = 0), si la hipótesis de **homocedasticidad se respeta** y si los errores no están correlados, la mejor función de pérdidas es la que se corresponde con los mínimos cuadrados. Bajo estas dos condiciones, **no es inútil suponer o comprobar que los errores se reparten según una ley de distribución normal o que son independientes**.

Por otro lado, sepa que este método es muy resistente al hecho de que ciertas hipótesis no se cumplan.

En inglés, hablamos de «best linear unbiased estimator (BLUE)», lo que expresa que el mejor estimador no condicionado por un modelo lineal es el obtenido mediante el método de los mínimos cuadrados.

Matriz Hat

Teníamos la expresión:

$$\hat{y} = \hat{X} \cdot \hat{a}$$

En el caso de la regresión vista más arriba, tenemos:

$$\hat{a} = (\hat{X}^T \hat{X})^{-1} \hat{X}^T y$$

Reemplazando este valor en nuestra primera expresión, se obtiene:

$$\hat{y} = \hat{X} \cdot (\hat{X}^T \hat{X})^{-1} \hat{X}^T y$$

Si se plantea ahora:

$$H = \hat{X} \cdot (\hat{X}^T \hat{X})^{-1} \hat{X}^T$$

Se obtiene:

$$\hat{y} = Hy$$

Esta matriz H se denomina matriz Hat, pues *hat* significa sombrero en inglés, y a esta matriz se le agrega un sombrero sobre la y (divertido, ¿verdad?).

La matriz Hat es una matriz de proyección que proyecta y sobre el hiperplano del modelo lineal. Dicha matriz Hat se calcula también en el caso de la generalización de modelos lineales, llamada «GLM» (aunque posee una expresión más compleja). La encontrará en otros contextos. Si dispone de la matriz Hat, puede expresar el vector de errores de predicción de manera trivial sin tener que pasar por la función \hat{f} , lo que nos permite ganar un tiempo de cálculo en ocasiones nada despreciable: $e = y - Hy = (I-H)y$

Algunos autores expresan la magia de esta matriz Hat, por ejemplo:

- $(I-H)$ es simétrica respecto a su diagonal (es igual a su traspuesta).
- $(I-H)$ es idempotente (es igual a su cuadrado).

Para recordar: el siguiente aspecto nos parece el más interesante: cuando alguno de los elementos de la diagonal es netamente superior a la media de los elementos de su diagonal (hablamos de dos veces superior), esto indica que la fila de observaciones correspondiente es una **palanca**; dicho de otro modo, que esta observación tiene un fuerte impacto sobre el modelo. **Conviene identificar estas palancas** y verificar cuidadosamente la pertinencia de estas muestras. También es posible intentar comparar el modelo con o sin estos puntos, en el peor caso se creará un modelo sobredeterminado (*overfit*) a partir de alguna de estas palancas, en cuyo caso esta palanca supondrá un error en la recopilación de la información.

Distancia de Cook

Esta última observación puede generalizarse a todos los tipos de modelo, en el sentido de que siempre debería preocuparle **identificar las observaciones que posean un efecto palanca sobre su modelo y que planteen un riesgo de sobredeterminarlo**.

La cuestión es ver cómo evoluciona nuestro vector de predicciones \hat{y} cuando se elimina alguna observación del conjunto de entrenamiento. Supongamos que la observación eliminada fuera la observación k; obtendríamos el vector $\hat{y}_{(-k)}$ resultante:

$$\hat{y}_{(-k)}$$

La desviación entre ambos vectores sería el vector:

$$\delta_{(-k)} = \hat{y} - \hat{y}_{(-k)}$$

Y, como de costumbre, podemos evaluar su tamaño utilizando una norma, aquí la norma euclidiana al cuadrado que obtenemos siempre multiplicando la traspuesta de un vector por sí mismo:

$$\delta_{(-k)}^T \cdot \delta_{(-k)}$$

Para poder comparar esta cantidad en distintos casos prácticos, hay que encontrar una ponderación que normalice

la cantidad que se desea utilizar como evaluador. Cook propone la siguiente ponderación, que se expresa como una relación que utiliza el tamaño de la matriz de los valores predictivos, nuestra matriz X. Los divisores de esta relación son el número de features (columnas de la matriz) y la varianza del error estimado. Esta cantidad se normaliza en función del nivel global de error de nuestra estimación.

La ponderación propuesta es la siguiente: $\frac{\mathbf{X}^T \mathbf{X}}{p \sigma_e^2}$.

La cantidad que hemos obtenido y que nos permitirá evaluar y comparar lo que ocurre cuando se elimina alguna observación del conjunto de datos de entrenamiento se denomina «distancia de Cook» y su expresión sería:

$$D_{(-k)} = \frac{\delta_{(-k)}^T \mathbf{X}^T \mathbf{X} \delta_{(-k)}}{p \sigma_e^2}$$

Llegados a este punto, la formulación de **esta distancia no está específicamente vinculada a los modelos lineales**, siempre podrá calcular dicha distancia (aunque existen riesgos en términos del tiempo de cálculo necesario...).

En el caso lineal, esta distancia se expresa simplemente en función de la matriz Hat y los cálculos son muy rápidos.

A nivel global, recuerde que, cuanto mayor sea la distancia de Cook (típicamente alrededor de 1), más posibilidades existen de que el valor sea una palanca o un valor aberrante, o ambos.

La lista de índices de las observaciones que han generado nuestro modelo lineal llamado **f_lm**, tales que la distancia de Cook sea superior a 0.9, se obtiene de la siguiente manera:

```
which(cooks.distance(f_lm) > 0.9)
```

La búsqueda de una buena estimación de f se lleva a cabo tratando de encontrar una configuración que minimice una función de riesgo, a menudo expresada como un indicador de desviación.

Esta búsqueda se realiza sobre un conjunto de entrenamiento o mediante métodos que veremos más adelante y que dividen el conjunto de entrenamiento y «cruzan» o «recopilan» sus resultados. Por último, cuando se comprueba el modelo, es posible implementar otros indicadores de desviación diferentes a los que nos han servido para definir y afinar el modelo. Además, a menudo se utilizan para comparar la eficacia entre varios modelos!

A continuación vamos a explorar los indicadores de desviación más sencillos de los que disponemos habitualmente.

2. Indicadores de desviación utilizados en machine learning

a. MSE, RMSE, SSE, SST

RMSE, el indicador «rey»

Un primer uso corriente, que tiene en cuenta nuestros últimos comentarios, consiste en utilizar el error cuadrático medio, MSE (*Mean Squared Error*) en inglés. Podemos comprobar que esto se corresponde con el uso de la media de la norma euclíadiana al cuadrado como función de riesgo:

$$MSE(y, \hat{y}) = E(\ell(., .)) = \frac{1}{n} \| y - \hat{y} \|_2^2$$

$$\text{con } \ell(y_i, \hat{y}_i) = |y_i - \hat{y}_i|^2 = (y_i - \hat{y}_i)^2$$

Expresando el MSE mediante una norma vectorial, nos abrimos a la idea de utilizar otras normas para diseñar funciones de riesgo que no estén definidas necesariamente a partir de la esperanza matemática de una función de pérdidas aplicada a cada desviación.



Llamamos a menudo SSE (Sum of Square Errors) a la cantidad: $\| y - \hat{y} \|_2^2 = \sum_{i=1}^n (y_i - \hat{y}_i)^2$

Con esta notación tenemos que: $MSE(y, \hat{y}) = SSE(y, \hat{y}) / n$

No debe confundirse con la SST (Total Sum of Squares), que no utiliza la estimación de y , a saber \hat{y} , sino su media \bar{y} . Esta cantidad solo depende de y : $SST(y) = \| y - \bar{y} \|_2^2 = \sum_{i=1}^n (y_i - \bar{y}_i)^2$

El MSE no es homogéneo respecto a y (es decir, que por ejemplo si y se expresa en metros, el MSE correspondiente se expresaría en m^2). Se utiliza a menudo su raíz para interpretarlo mejor; a esta cantidad se la denomina **root-mean-square deviation (RMSD)** o **root-mean-square error (RMSE)**. El primer nombre nos recuerda que se mantiene cierta relación con una desviación típica, el segundo nos recuerda la forma en la que se construye este indicador de desviación. Tenemos que:

$$RMSE(y, \hat{y}) = \sqrt{MSE(y, \hat{y})}$$

A nivel global, podríamos pensar que, cuanto más débiles sean los MSE o RMSE, mejor será el modelo (es más o menos el caso, aunque esto requeriría una discusión mucho más profunda).

¿Para qué sirven los indicadores de desviación?

Los usos típicos de MSE, RMSE y de la mayoría de los indicadores que veremos coinciden en sus objetivos:

- Primer uso posible: el algoritmo utilizado para construir el modelo está diseñado para mejorar el indicador en cuestión, bien mediante un cálculo directo o bien por iteraciones sucesivas. Hablamos a menudo de método de descenso por gradiente en el caso de iterar haciendo variar los parámetros que se van a optimizar en una dirección que nos ha producido cierta mejora respecto al ciclo anterior. Por ejemplo, el método de los «mínimos cuadrados» consiste precisamente en calcular una solución que minimice el MSE!
- Segundo uso posible: se comparan los valores obtenidos por un indicador (o varios) para determinar el mejor modelo para un conjunto de datos de entrenamiento, o llegado el caso para un contexto determinado si se busca una generalización o la elaboración de un método.
- Tercer uso posible, similar al anterior pero más raro: se busca qué modelo aplicar sobre las distintas partes de un conjunto de trabajo antes de construir una predicción que será la composición de estos modelos (este tipo de método es uno de los que generalmente se clasifican bajo la denominación de métodos «de conjunto»).
- Cuarto uso: en vista de los indicadores, podemos declarar que nuestra estimación es buena, o usable, o exitosa (¡o no!) para concluir acerca de la calidad o empleabilidad de nuestro modelo en un contexto determinado. Este objetivo, muy operacional, que consiste en poder afirmar que se ha obtenido un «buen modelo», es en buena parte un «sueño» para cualquier problema derivado del mundo real. Aunque a los data scientists operacionales se les juzga en estos términos. De hecho, sus indicadores le ayudarán a formarse una convicción que puede compartir a través de explicaciones didácticas, pero que quedan bajo escrutinio, puesto que «incluso el mejor juego de datos solo puede dar lo que contiene...».

b. MAE, ME

Otro uso común, muy utilizado en el caso de las series temporales, consiste en utilizar la media del valor absoluto de los errores, MAE (*Mean Absolute Error*) en inglés:

$$\text{MAE}(y, \hat{y}) = \frac{1}{n} \| y - \hat{y} \|_1$$

- Para obtener la media se divide por el número de elementos **n**, en ciertos casos se decide dividir entre **n-1** o **n-2**. Abordaremos brevemente esto en el capítulo Feature Engineering cuando describamos la PCA. Observe que, como nos situamos en el caso del big data, el número de elementos es muy elevado, y dividir por n, n-1 o n-2 no cambia el resultado de una manera notable. La función **sd()** de R utiliza **n-1** para calcular la desviación típica (standard deviation).

- Preste atención: en ocasiones escuchará hablar de sesgo, bias (B), o bien Mean Error (ME) en inglés, que representa una media de la desviación pero conservando el signo, si bien una desviación negativa podría compensarse con una desviación positiva y viceversa. Esta noción de sesgo resulta interesante, pues permite percibir cierto desfase sistemático del modelo en una dirección o en otra. ¡Aunque es peligroso querer optimizar un modelo basándose en la disminución de este sesgo! En efecto, imagine que posee elementos estimados con grandes errores positivos (o negativos); podría generar durante su estimación grandes errores negativos (o positivos) y disminuir el sesgo.

De manera inversa, trabajar para buscar un modelo cuya optimización se realice basándose en el requisito de disminuir el MAE (o cualquier otro indicador de desviación que resulte de un promedio de valores todos del mismo signo) no tendría este inconveniente, siendo de hecho el MAE una media de los valores absolutos:

$$\text{MAE}(y, \hat{y}) = < |y - \hat{y}| >$$

$$\text{ME}(y, \hat{y}) = < y - \hat{y} >$$

con una de las notaciones propuestas más arriba sobre la esperanza matemática.

Paradójicamente, estos indicadores que parecen muy simples plantean un pequeño problema matemático: no resulta agradable derivarlos, pues contienen un «valor absoluto».

Derivar un indicador permite estudiar cómo evoluciona cuando alguna variable o algún parámetro del modelo evolucionan. Es un elemento muy útil para mejorar la comprensión de lo que se intenta manipular. Imagine, por ejemplo, que un indicador de error en porcentaje evolucionara de manera notable en función de una medida cualquiera realizada sobre una o varias de sus variables; podría quizás deducir una nueva estrategia en cuanto al(a) los modelo(s) que va a utilizar.

En este mismo sentido, nos gustaría que la derivada de una regresión respecto a los valores predichos de una medida de error fuera próxima a cero en cualquier punto (veremos esto en la sección «práctica» de este capítulo), es decir, que fuera próxima a **una recta horizontal**.

Para comprender algunas características un poco más intrínsecas de una cantidad, sin comprender su dependencia respecto a una variable utilizando la noción de derivada, podemos utilizar un mecanismo de relación o de coeficiente de variación respecto a una variable o un parámetro: es este tipo de indicador aplicado al RMSE el que vamos a abordar a continuación.

c. NRMSE/NRMSD, CV_MRSE

Cuando se quiere comparar el cálculo de indicadores de desviación sobre predicciones que corresponden a

diversas naturalezas (y con naturalezas diferentes, peso, edad...) para evaluar por ejemplo la eficacia de un algoritmo en distintas situaciones, o simplemente para interpretar rápidamente el valor de un RMSE o de un MAE, tratamos de eliminar el efecto de la escala normalizando el valor. La cuestión que se plantea es escoger una referencia: esta puede ser la desviación máxima entre dos valores de la variable y y su desviación típica, su media, que expresaremos llegado el caso en porcentaje... Con esta idea obtenemos:

$$\text{NRMSE}(y, \hat{y}) = \text{NRMSD}(y, \hat{y}) = \frac{\text{RMSE}(y, \hat{y})}{y_{\max} - y_{\min}}$$

La "N" de NRMSE significa "Normalized".

Hemos visto antes la noción de coeficiente de variación, que hemos expresado según la siguiente fórmula:

$$c_v = \frac{\sigma}{\mu}$$

En esta fórmula, σ representa la desviación típica de nuestra variable; aquí la variable cuya desviación típica se va a medir es el error $e = y - \hat{y}$. En esta expresión μ representa la media de esta variable, aquí la media o la esperanza matemática de e . Sin embargo, queremos que esta esperanza matemática sea nula. Este coeficiente de variación resulta incoherente. En el caso del RMSE, se define otro coeficiente de variación que no está normalizado sobre la media de aquello sobre lo que se ha calculado la desviación típica (en este caso e), sino sobre la media de la propia variable y , que llamaremos aquí \bar{y} .

Obtenemos:

$$\text{CV_RMSE}(y, \hat{y}) = \frac{\text{RMSE}(y, \hat{y})}{\bar{y}}$$

«CV» significa «coeficiente de variación», pero en un sentido propio de esta expresión (es decir, no generalizable sistemáticamente a otros indicadores, aunque generalizable a otros indicadores de desviación de error).

Hemos visto antes (en la sección de iniciación a las estadísticas y a las probabilidades) que podíamos estudiar distintos «momentos» de una distribución. Cuando se tiene una idea de la tendencia central (media, mediana...), podemos estudiar la dispersión. Esto es lo que veremos con el siguiente indicador.

d. SDR

En ocasiones nos referimos a las desviaciones (errores) con el término **residuos**. Para percibir su dispersión, podemos calcular la desviación típica de estos residuos y la llamamos tradicionalmente «Standard Deviation of Residuals» o SDR en inglés.

$$\sigma_e = \text{SDR}(y, \hat{y})$$

 Se demuestra fácilmente la siguiente aserción: $\text{MSE} = \text{RMSE}^2 = \text{ME}^2 + \text{SDR}^2$

Más claro: **varianza del error = sesgo al cuadrado + varianza de los residuos**.

Podemos actuar sobre el sesgo, que es reducible, pero no se actúa formalmente sobre la varianza de los residuos que podemos considerar como irreductible si no queremos capturar el ruido inherente a nuestros datos mediante el modelo que tratamos de diseñar.

Entre los cuatro usos que hemos visto antes, el último uso, a saber, determinar la calidad de nuestro modelo, se revela como el más polémico. Los siguientes indicadores, que pueden utilizarse con éxito en otros casos de uso, a menudo se usan para formarse una idea, una convicción o una consideración respecto a la calidad del modelo.

e. Accuracy, R2

Los siguientes indicadores también son muy corrientes y ofrecen una rápida percepción del nivel de la «calidad» de la estimación.

Accuracy

En caso de que busquemos predecir la pertenencia a una clase, típicamente cuando y puede valer 0 o 1, utilizamos naturalmente el porcentaje de predicciones exitosas respecto al número total de predicciones. Este indicador se denomina *Accuracy* en inglés. Este indicador es fácil de usar si trata de predecir la pertenencia a una clase. Si trata de predecir un valor real y desea definir una noción de umbral (*threshold*) sobre los valores que se han de predecir, puede extender esta noción para determinar indicadores similares.

La búsqueda de una mejor *accuracy* puede resultar crítica en ciertos casos. En efecto, en ocasiones es preferible optimizar el modelo basándose en la búsqueda de otras buenas tasas de predicción. Por ejemplo, un médico prefiere correr el riesgo de establecer una mecánica de diagnóstico estadístico que le permita encontrar falsos positivos, es decir, personas identificadas como enfermas que realmente no lo están tras realizarles un examen más profundo, mientras que un economista de la Seguridad Social preferirá posiblemente correr el riesgo de utilizar un modelo que fabrique falsos negativos frente al riesgo de obtener falsos positivos, pues desea limitar el número de exámenes costosos e inútiles... Encontrará abundante literatura respecto a estos asuntos y sobre asuntos muy conectados con los términos «matriz de confusión», ROC y AUC que hemos esbozado en el capítulo Introducción.

R2, coeficiente de determinación

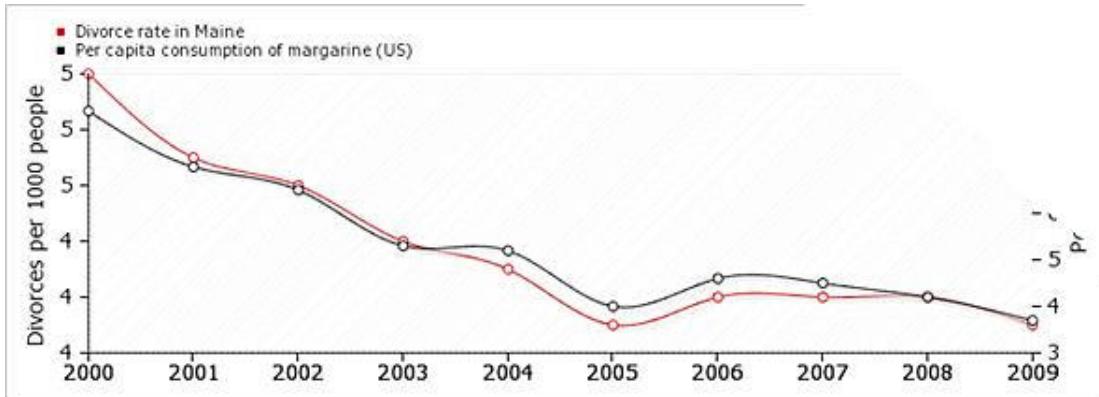
Para evaluar si el modelo representa bien la variación de los datos, podemos utilizar:

$$R^2 = 1 - \frac{SSE}{SST}$$

R2 se denomina coeficiente de determinación, se le llama «R cuadrado», *R squared*, pues para regresiones lineales es igual al cuadrado del coeficiente de correlación. Normalmente, sus valores están comprendidos entre 0 y 1, aunque como su expresión incluye una resta, que se utiliza en diversos contextos bajo diversas definiciones (fórmulas) para R2, en ocasiones obtendrá valores para R2 que no estarán dentro de este intervalo, en función de sus lecturas o cuando utilice diversos paquetes.

A nivel práctico, en lo que respecta a nuestro problema de estimación, cuanto mayor sea el valor de R2, podemos considerar que mayor será la correspondencia entre la aplicación del modelo sobre los datos de prueba frente a los datos reales.

El uso de R2 para afirmar eso o aquello respecto a la veracidad de una relación entre las X e y es una práctica habitual en política y en consultoría, a menudo perniciosa, o en todo caso poco afortunada. En el momento de escribir estas líneas, un sitio ilustra con humor este tipo de conclusiones aberrantes. Le invito a visitarlo para plantearse ciertas cosas: <http://tylervigen.com/spurious-correlations>. Verá, por ejemplo, la siguiente correlación (R2 del orden de 0.98), que podría hacer pensar que la tasa de divorcios en Maine puede predecirse conociendo la tasa de consumo de margarina en los Estados Unidos o viceversa:



Extraña similitud

Destaquemos, en cualquier caso, que estas series son series temporales, contienen pocos puntos, y hablamos de una sola variable explicativa, lo que favorece los «azares» que generan grandes R².

Conviene ser consciente de que el **hecho de agregar variables tiende a aumentar los valores de R²** para estimar un **y** determinado (lo que puede parecer contradictorio según la observación anterior, pero en cualquier caso evoca la probabilidad de obtener un gran R² sin haber fijado el objetivo **y** que se ha de predecir).

Para limitar estos fenómenos, se utiliza a menudo un R² que llamamos «ajustado»: «adjusted R²». La fórmula de este R² tiene en cuenta los grados de libertad mediante la aplicación de una relación $(n-p-1)/(n-1)$. En esta expresión, p es el número de variables y n el tamaño de la muestra. Como se ha indicado en otros lugares en este libro, cuando las muestras son importantes, esta relación tiende a 1 y este tipo de correlación pierde bastante interés.

Si bien esto no se vislumbra a primera vista en su fórmula, R² es muy sensible a la variación de las X. Nadie puede afirmar que un valor de R² es efectivamente «bueno» con certeza (ni tampoco malo, de hecho), salvo en el caso de la prueba de la multicolinealidad de diversas variables.

Conclusión intermedia

Basándonos en los indicadores, utilizados en un contexto operacional, la idea es la siguiente: **mis esfuerzos no mejoran notablemente mis indicadores, y ahora puedo escoger entre considerar mi trabajo como finalizado o cambiar radicalmente los paradigmas de mi estudio** (sin duda, plateándose cuestiones importantes o con un coste o un retraso que conviene evaluar adecuadamente). Como corolario obtenemos que: **la dificultad vinculada a la mejora de mis indicadores puede significar que ya he extraído toda la información disponible en mis datos, y otras etapas no van a mejorar, sino más bien a sobredeterminar mi modelo y a integrar ruido (con el correspondiente riesgo de overfitting)**.

A continuación vamos a realizar dos predicciones sobre los mismos datos utilizando modelos diferentes. Luego compararemos sus respectivos valores de R².

Puesta en práctica: aprendizaje supervisado

Vamos a realizar un pequeño ciclo de construcción de una predicción para poner en práctica las nociones vistas anteriormente. En primer lugar, vamos a dotarnos de un conjunto de observaciones. Este conjunto contendrá 10 000 observaciones, dos features como entrada y una que debemos predecir. Buscaremos el modelo f , la feature objetivo será y , la entrada $X(x_1, x_2)$. El conjunto de observaciones $Z = (X, y)$ está dividido en un conjunto de entrenamiento Z_e de 6 000 filas y un conjunto de validación de Z_v de 4 000 filas.

Vamos a entrenar al modelo sobre el conjunto de entrenamiento Z_e , tratando de estimar el modelo f , lo que nos dará una función «estimación de f » tal que:

$$\hat{y}_e = \hat{f}(X_e)$$

A continuación aplicaremos \hat{f} sobre el conjunto de validación X_v y obtendremos un vector de predicciones \hat{y}_v que compararemos con el vector y_v de los valores reales propios del conjunto de validación.

- Para hacer esto vamos a crear simplemente un conjunto Z «ficticio» donde habremos impuesto, y luego perturbado aleatoriamente, el hecho de que y se sitúe más o menos sobre el plano:

$$2 \cdot x_1 - 7 \cdot x_2 + 1.$$

Nuestra idea es constatar que los algoritmos utilizados encuentran la ecuación de este plano con un error de pequeña amplitud.

1. Preparación

El código de creación del conjunto ficticio de observación es el siguiente (sin ningún interés específico):

```
# un dataset ficticio, creación de las matrices          #

rm(list=ls())

g <- function(x){s <- round(abs(sin(x)*1000 ))
  set.seed(s)                      # randomize
  # pero asegura que g(constante) es una constante
  x*(1- 0.5*rnorm(1)*sin(x))} # para perturbar

set.seed(2); x1 <- matrix(rnorm(10000))
set.seed(3); x2 <- matrix(rnorm(10000))
y     <- matrix(2*x1-7*x2+1)
y     <- apply(y,1,g)
Z     <- data.frame(x1,x2,y)
```

A continuación vamos a dividir el conjunto de observaciones en dos conjuntos, uno de entrenamiento y otro de validación.

En primer lugar comprobaremos el número de columnas, para evitar sorpresas.

```
names(Z)          # X = [x1,x2] ; y = [y]
```

```
[1] "x1" "x2" "y"
```

La función **createDataPartition** de la librería **caret** nos va a permitir extraer una partición de Z. Nos permite construir aleatoriamente un índice de un tamaño determinado correspondiente a una proporción de los números de fila de Z. El vector **index** resultante nos permitirá extraer las filas de nuestro conjunto de entrenamiento, y tomando su opuesto **-index** podremos extraer las filas de nuestro conjunto de validación /test.

```
require(caret)      # nos aporta un particionamiento elegante
set.seed(1)         # inicializa el generador de números aleatorios
index <- createDataPartition(y=Z$y,           # extracción de index
                           p= 60/100,    # que es una serie de
                           list=FALSE)  # 6000 números aleatorios

str(index)          # comprobamos que está ok

Ze   <- Z[ index, ]  # extracción de las 6000 filas
Zv   <- Z[ -index, ]  # extracción de las 4000 filas complementarias
```

```
int [1:6000, 1] 1 3 5 7 9 10 11 12 13 14 ...
```

```
head(index)        # comprobamos - 5 primeros índices
head(Ze)           # 5 primeras filas de estos índices
dim (Ze)           # obtenemos efectivamente 6000 filas
head(Zv)           # 5 primeras filas de los otros índices
dim(Zv)            # obtenemos efectivamente 4000 filas
```

Las seis primeras filas del conjunto de entrenamiento Ze:

	x1	x2	y
1	-0.89691455	-0.96193342	7.3642765
3	1.58784533	0.25878822	2.2410705
5	-0.08025176	0.19578283	-0.5627873
7	0.70795473	0.08541773	1.3363726
9	1.98447394	-1.21885742	15.9955753
10	-0.13878701	1.26736872	-2.3581976

Las cuatro primeras filas del conjunto de validación /test Zv:

	x1	x2	y
2	0.18484918	-0.29252572	3.915574
4	-1.13037567	-1.15213189	6.875471
6	0.13242028	0.03012394	1.143770
8	-0.23969802	1.11661021	-9.811021

Confirmamos que sobre las 10 primeras filas tenemos efectivamente el 60 % de filas en el conjunto de entrenamiento, y el 40 % restante en el conjunto de validación/test, con un aspecto aleatorio sobre la elección del reparto de los números de las filas entre los dos **data.frames**.

Para los siguientes cálculos, necesitamos extraer la columna **y** y sus particiones:

```
y <- matrix(Z$y)           # las y
ye <- matrix(Ze$y)          # las y de entrenamiento
yv <- matrix(Zv$y)          # las y de validación

dim(y)                      # vector 10000 coord
dim(ye)                     # vector 6000 coord
dim(yv)                     # vector 4000 coord
```

2. Comprobar las hipótesis, p_value

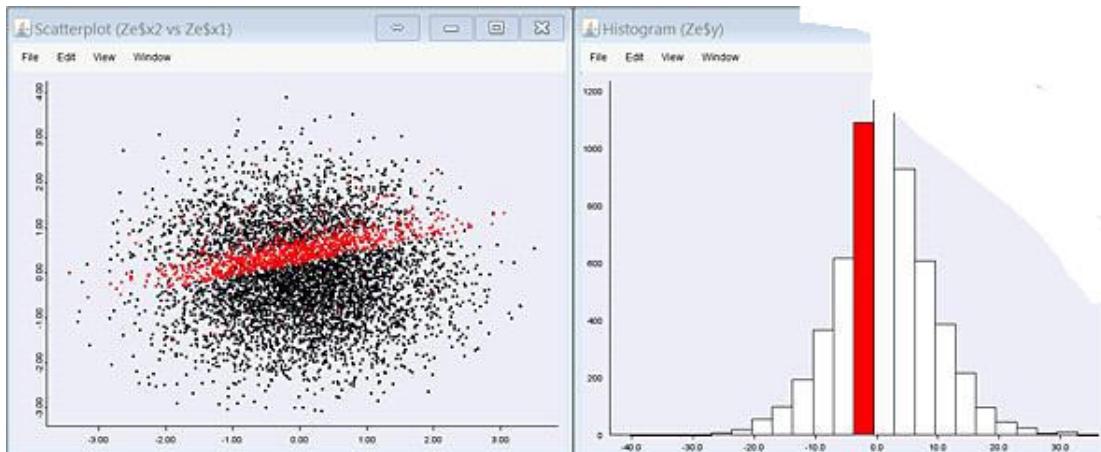
Llegados a este punto, conviene estudiar visual y estadísticamente nuestros datos tal y como hemos indicado en otros capítulos.

a. Análisis gráfico interactivo con iplots

En términos gráficos, vamos a contentarnos aquí con explotar un paquete muy agradable de R, el paquete **iplots**, que permite una visualización interactiva y sincronizada en varias ventanas. Este paquete permite seleccionar con el ratón ciertos datos sobre uno de los gráficos y visualizar los puntos correspondientes sobre los demás gráficos. Aquí hemos seleccionado una banda en la distribución de las **y** y comprobamos los puntos correspondientes en la nube de puntos **x1** por **x2**.

```
library(iplots)

iplot(Ze$x1, Ze$x2)
ihist(Ze$y)
```



Datos sincronizados interactivamente sobre dos gráficos

La banda elegida sobre el histograma de la derecha se corresponde con una serie de puntos alineados sobre el

«scatterplot» de la izquierda; si hacemos lo mismo con las demás bandas, constatamos que se obtiene en la izquierda otra serie de puntos alineados de forma paralela a la primera, lo que le confirma visualmente una cierta linealidad de los datos **x1** y **x2** respecto a **y**.

Podemos (y debemos) utilizar otras representaciones de los datos si queremos percibirlos con agudeza. Por otro lado, las pruebas estadísticas nos van a ayudar a elaborar estrategias de exploración de nuestros datos y a formalizar nuestras primeras intuiciones.

b. Test de Breush-Pagan y zoom sobre p_value

En términos de pruebas estadísticas de hipótesis, para esta pequeña aplicación práctica, podemos contentarnos con realizar el test de **Breush-Pagan** para obtener buenas presunciones de **homocedasticidad de los residuos** de la aplicación de un modelo de regresión lineal. De hecho, el test de Breush-Pagan es un test de heterocedasticidad! (para convencerte de ello, lea la ayuda del test en R: *Performs the Breusch-Pagan test against heteroskedasticity*).

Para comprender el resultado del test hay que conocer la noción de **p-value**, que se utiliza a menudo en estadística (en ocasiones a ciegas!) y que permite pronunciarse con más o menos convicción sobre el resultado de un test.

La idea consiste en comprobar una hipótesis llamada «**hipótesis nula**», que a menudo se escribe H_0 . En la teoría subyacente al uso de p-value, se considera que esta **hipótesis nula no puede validarse jamás**. Preste atención, este punto es muy importante: jamás se puede concluir con esta herramienta que H_0 es verdadera.

Por el contrario, sabemos que a partir de cierto umbral podríamos orientarnos al hecho de invalidar la hipótesis nula.

Los umbrales utilizados son los siguientes:

- Si p_value es **inferior a 0.01, se propone invalidar** la hipótesis nula.
- Si p_value es **superior a 0.1, no se puede invalidar** la hipótesis nula.

El código correspondiente es el siguiente:

```
library(lmtest)          # paquete que contiene los tests para lm
bptest(y ~ x1 + x2, data = Ze) # test
? bptest                  # ayuda del test
```

```
studentized Breusch-Pagan test
```

```
data: y ~ x1 + x2
BP = 50.6275, df = 2, p-value = 1.015e-11
```

Como p_value es muy pequeño, invalidamos la hipótesis nula, que sería la heterocedasticidad de nuestros residuos. Esta es una buena noticia, disponemos de datos cuyos residuos respecto a una regresión lineal poseen una fuerte presunción de homocedasticidad.

Como habrá deducido, resulta vital comprender bien cuál es la hipótesis nula de las pruebas que utiliza. Esto no siempre es sencillo, desconfíe de los numerosos errores de documentación y **realice sus tests sobre los datos** que domine y sepa anticipar los resultados antes de generalizar el uso de una prueba que aprenderá por primera vez.

3. Creación de un modelo (regresión lineal múltiple)

Ahora vamos a construir nuestra estimación de la función de predicción f , utilizando un algoritmo de los más sencillos y más rápidos, que se aplicará sobre el conjunto de entrenamiento: el «fitting linear model». Este entrenamiento, o aprendizaje (machine learning), es el que producirá el modelo \hat{f} .

```
# creación del modelo lineal sobre el conjunto de entrenamiento #
# aplicación sobre el conjunto de validación                      #

f <- lm(y ~ x1+x2,data=Ze)      # creación del modelo y=f(x)
```

Observe la sintaxis: **lm()** nos devuelve un modelo f , elaborado mediante un aprendizaje sobre el **data.frame** **Ze**. La variable que se va a predecir es la columna **y** y esta predicción debe realizarse basándose en efecto combinado de los atributos representados por las columnas **x1** y **x2**. Las sintaxis de las «fórmulas» del tipo **y ~ x1+x2** es característica del lenguaje R, es muy potente y permite describir casos muy variados de potenciales relaciones entre las variables.

4. Establecer una predicción

Ahora podemos realizar una predicción sobre nuestro conjunto de validación. La sintaxis de **predict** se utiliza sobre una gran cantidad de modelos construidos con R.

```
yv_ <- predict(f,newdata = Zv) # f(Xv)
yv_ <- matrix(yv_)
```

Podemos almacenar nuestras predicciones y también nuestro modelo (esto resulta muy útil):

```
                # almacenemos nuestros resultados
                # para un uso futuro
prediccion_lm <- data.frame(Zv[,-3], # se extrae y
                             prediccion_y = yv_,
                             row.names = NULL)
head(prediccion_lm)
tail(prediccion_lm)

f_lm <- f                                # memorización del modelo para una
                                             # comparación futura
```

Algunas filas producto de nuestra predicción:

	x1	x2	prediccion_y
2	0.18484918	-0.29252572	3.421428
4	-1.13037567	-1.15213189	6.713933
6	0.13242028	0.03012394	1.070163
8	-0.23969802	1.11661021	-7.246921
18	0.03580672	-0.64824281	5.591171
19	1.01282869	1.22431362	-5.437667

	x1	x2	prediccion_y
--	----	----	--------------

```

9984  0.88058277  0.4480796   -0.3087302
9987 -0.95327970 -0.4346129    2.0849951
9992 -0.42200874 -0.1059597    0.8842236
9995 -0.97046292  0.7907912   -6.4733440
9999 -0.06147666  0.8628701   -5.1180128
10000 -0.40880672 -2.5715141   18.0602384

```

La predicción empieza en la fila 2 y termina en la fila 10 000, es aleatoria. Recuerde que aquí tiene 4 000 filas de las 10 000 filas originales.

Sí, pero ¿cuál es la calidad de nuestra predicción? Veámosla...

5. Estudio de los resultados y representación gráfica

Hemos obtenido las predicciones sobre nuestro conjunto de validación/test **yv_** y queremos compararlas con los valores reales de **yv**. Una primera forma de proceder consiste simplemente en trazar los puntos de uno respecto a los puntos del otro. Cuanto más próximos estén los puntos de la diagonal, menor será el error. Pero preste atención, conviene tener en mente que parte del error es irreductible porque los datos poseen una aleatoriedad «intrínseca», un ruido. En ocasiones se integra este ruido en el modelo y esto produce una menor capacidad predictiva sobre los nuevos juegos de datos: es el miedo del data scientist, bien conocido con el término de overfitting.

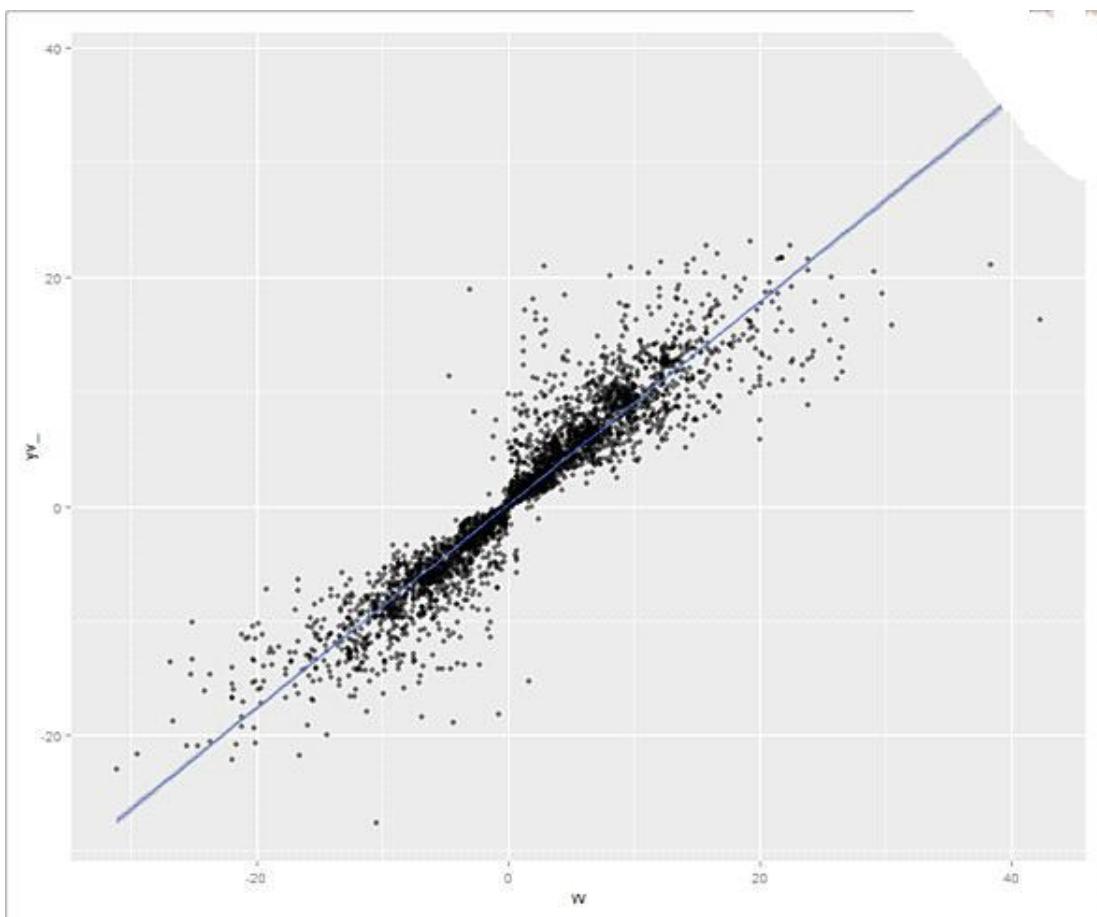
```

require(ggplot2)           # para disponer de qplot()
                           # visualización del resultado
qplot(yv,yv_,              # comparación entre y e y predicha
      geom = c("point", "smooth"),
      method ="lm",
      alpha = I(1/2)
)                         # aspecto satisfactorio

```

Observe los parámetros muy útiles de la función **qplot()**, que permiten introducir una regresión, aquí lineal, y una transparencia (**alpha**) de los puntos con el objetivo de determinar mejor las zonas formadas por muchos puntos o no (intente hacer variar **alpha**).

El resultado muestra que los puntos de la regresión lineal (lm) están, efectivamente, próximos a una recta a 45°; esta es una buena noticia en cuanto a la conveniencia de nuestro modelo.



Predicción VS valores reales

💡 Una falsa y peligrosa «buena idea común» de la que debe desconfiar:

Tenemos dos variables, x_1 y x_2 ; habríamos podido imaginar estimar y como una función de x_1 . Habríamos obtenido $y = f_1(x_1) + e_1$. A continuación, haber estimado e_1 en función de x_2 , $e_1 = f_2(x_2) + e_2 \dots$ y, a final, habríamos obtenido algo del estilo:

$$y = f_1(x_1) + f_2(x_2) + e_2.$$

Efectivamente, esto no es correcto (salvo en casos muy particulares, y a menudo triviales), pues se ha transportado un sesgo de una estimación a otra (y se ha negado el impacto de la interacción entre x_1 y x_2 sobre y).

En la literatura inglesa, cuando encuentre los términos «stepwise blabla» o bien ROR (regression on residual)... esté muy atento, ipuede que se enfrente a un error metodológico común que no sería conveniente repetir!

El estudio de los residuos es una práctica simple que nos permite cualificar mejor el resultado aportado por un modelo. Vamos a construir un vector de residuos e y trazarlo en función de los valores estimados:

```
e <- yv - yv_
# vector de errores (= residuos)

qplot(yv_,e,
      # dispersión de los errores en func. de yv
      geom = c("point", "smooth"),
      method = "lm",
      alpha = I(1/2)
)
```

Lo que produce:

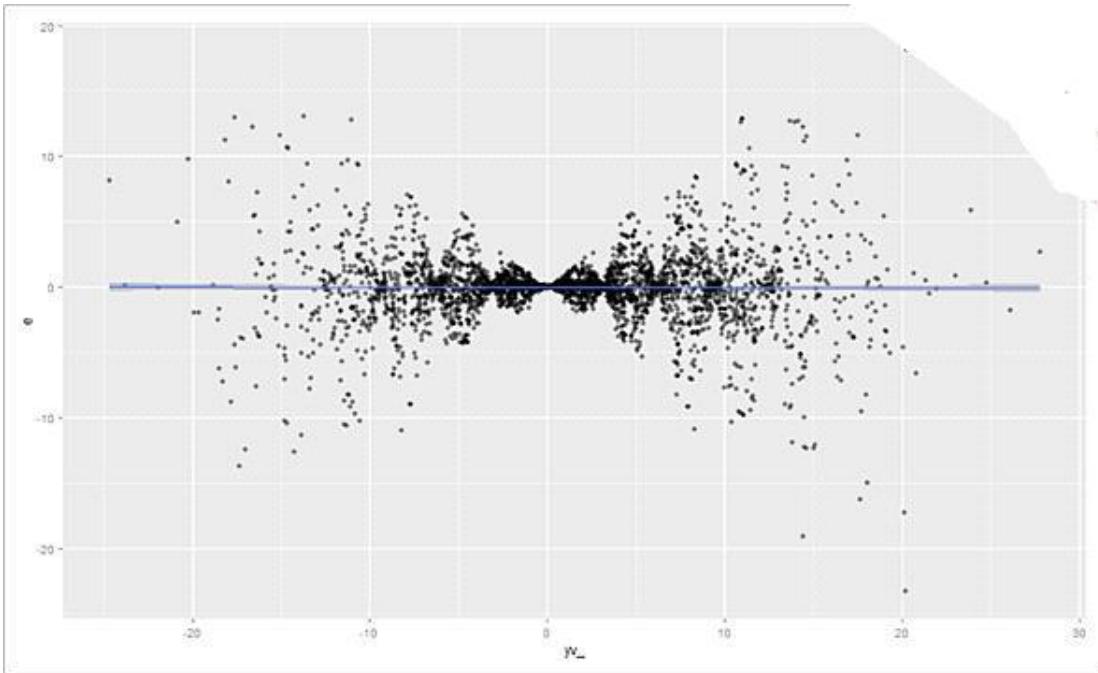


Diagrama de residuos en función de las predicciones

Se constata que la regresión lineal sobre los residuos muestra que «globalmente» los residuos no dependen de los valores predichos. En efecto, la recta es horizontal (si hubiéramos utilizado una aproximación, otro alisado, habríamos podido observar si la derivada era casi siempre nula): esta es otra buena noticia.

Observe que podríamos haber preferido obtener una banda homogénea, horizontal y repartida de manera simétrica respecto al eje horizontal: nuestro modelo posee un pequeño defecto, io bien la muestra está algo sesgada!

Habríamos podido trazar el mismo diagrama en proporción del error:

```
d <- max(yv_)-min(yv_)          # en porcentaje
p <- e/d *100                   # desviación máxima de las predicciones
                                 # proporción de error

qplot(yv_,p,                     # estudio proporción de error
      geom = c("point", "smooth"),
      method = "lm",
      ylab = "porcentaje de error",
      alpha = I(1/2)
    )
```

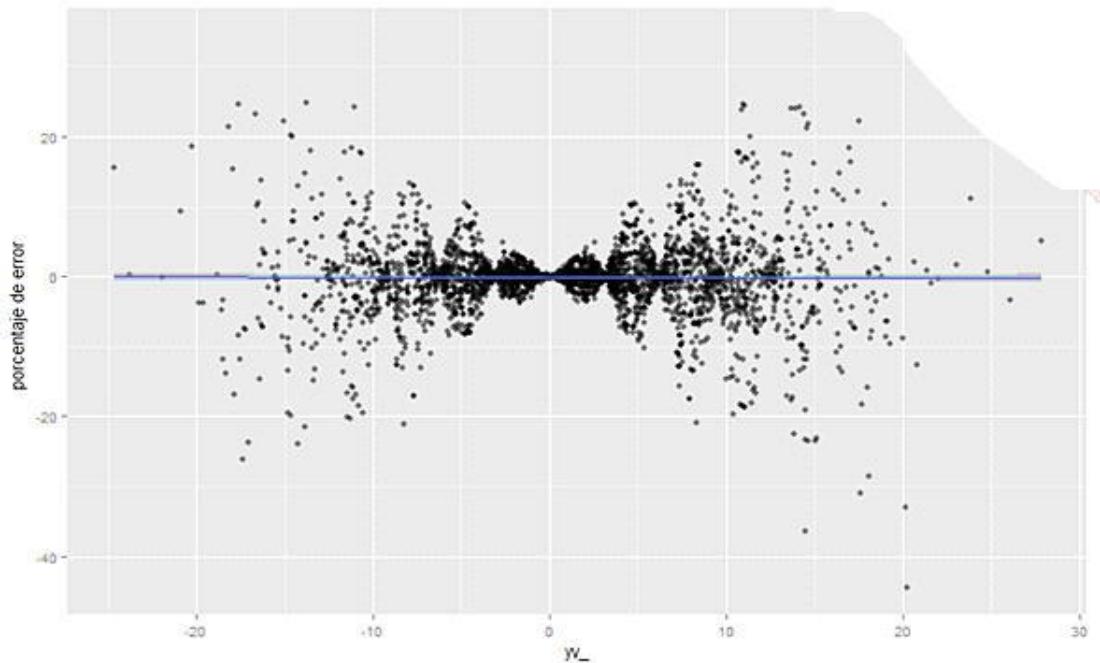


Diagrama de los residuos (en %) en función de las predicciones

De este modo, resulta fácil de interpretar, y se visualiza mejor la proporción de ruido presente.

Para comprender mejor la calidad del resultado, también podemos utilizar los indicadores descritos más arriba.

Es preciso calcularlos; es lo que haremos a continuación.

6. Indicadores habituales - cálculos

Para hacer esto, el código es muy sencillo, aunque no deben confundirse los datos de entrenamiento con los datos de test. Aquí se realizan **cálculos sobre los datos de test/validación** y sobre el vector de error **e**.

Observará también que aquí el cálculo se realiza partiendo de la norma calculada mediante la función **entrywise.norm()** del paquete **matrixcalc** (en ocasiones el autor denomina a los paquetes librerías; es un abuso del lenguaje práctico, pero no del todo comprendido por los ingleses en los foros dedicados a R...). El hecho de referirse a la norma nos abre la puerta a fabricar nuestros propios indicadores a partir de otras normas que estén mejor adaptadas a nuestro problema. Esto en sí es un campo de investigación, y en ocasiones resulta muy interesante. Evidentemente, si salimos del mundo del MSE, habría que darle otros nombres y la fórmula (**MSE**2 + SRD**2**) - **MSE** ya no devolverá cero.

```
# cálculo manual de los indicadores habituales #

library(matrixcalc)           # para realizar cálculos matriciales

n <- length(yv)               # número de elementos de yv
e <- yv - yv_                 # vector de errores (= residuos)
n <- length(yv)               # número de elementos de yv

SSE <- entrywise.norm( e, 2 )**2      # SSE
MSE <- SSE / n    # MSE
RMSE <- sqrt(MSE)             # RMSE
```

```

MAE <- entrywise.norm( e, 1 ) / n          # MAE
ME <- mean(e)                             # ME
NRMSE <- RMSE/(max(yv) - min(yv))        # NRMSE
CV_RMSE <- RMSE/mean(yv)                  # CV_RMSE
SRD <- sqrt(mean(e**2) - mean(e)**2)      # SRD

(ME**2 + SRD**2) - MSE                   # comprobación

MEAN   <- mean(yv)                      # media yv
CENTER <- yv-MEAN                       # centro yv
SST    <- entrywise.norm( CENTER, 2 )**2  # SST
R2    <- 1- SSE/SST                      # R2

idl1 <- data.frame(n,SSE,MSE,RMSE,MAE,ME,NRMSE,CV_RMSE,SRD,SST,R2)
idl1

```

Se comprueba que **(ME**2 + SRD**2) = MSE**, pues su diferencia es nula:

[1] 0

Siempre resulta interesante comprobar la coherencia de nuestros cálculos.

Se obtienen los resultados de nuestros cálculos (almacenados en un mini-**data.frame** para un posible uso futuro). Evidentemente, habríamos podido encapsular todo esto en una función R, aunque aquí no se da el caso.

Resultado de nuestros cálculos:

n	SSE	MSE	RMSE	MAE	ME	NRMSE
4000	26482.29	6.620573	2.573047	1.4765	-0.06345662	0.03505314
	CV_RMSE	SRD	SST	R2		
	2.511445	2.572265	232643.2	0.8861678		

El valor normalizado de la raíz del error cuadrático medio (NRMSE) parece pequeño (3.5 %), y el coeficiente de determinación (R2), próximo a 1. Esto nos tranquiliza.

A continuación vamos a observar con detalle del modelo creado.

7. Estudio del modelo lineal generado

Cada paquete define su propia visualización de los elementos que describen el modelo generado. En muchos casos, es la función **summary()** la que nos permite acceder a esta información.

```
summary(f)           # detalle del modelo
```

Esto nos devuelve la siguiente información (que interpretaremos en función de nuestros conocimientos, llegados a este punto del libro).

```

Call:
lm(formula = y ~ x1 + x2, data = Ze)
```

Residuals:

```

Min      1Q   Median     3Q      Max
-18.9865 -0.6949  0.0014  0.7709  20.9093

```

Coefficients:

	Estimate	Std. Error	t value	Pr(> t)
(Intercept)	1.00921	0.03438	29.36	<2e-16 ***
x1	2.04257	0.03439	59.39	<2e-16 ***
x2	-6.95545	0.03398	-204.72	<2e-16 ***

Signif. codes:	0 **** 0.001 ** 0.01 * 0.05 . 0.1 ' ' 1			

En primer lugar, el resumen nos recuerda la fórmula utilizada. Esto resulta muy útil cuando se ha almacenado el modelo y se quiere utilizar más adelante.

La línea correspondiente a los residuos es muy interesante, pero observe con atención que, como con los demás indicadores, se ha construido utilizando los datos de entrenamiento y no los datos de test y, por tanto, la fiabilidad de esta información es parcial.

El valor medio de los residuos es prácticamente nulo: hay tantos residuos inferiores a cero como superiores, como podríamos haber imaginado observando nuestros diagramas previos. El 50 % de los errores están comprendidos entre -0.69 y 0.77 (del primer cuartil al tercer cuartil hay un 50 % de datos, por definición).

Los valores del primer y del segundo cuartil están muy próximos a la media y muy alejados de los valores extremos (por ejemplo, 0.7709 es muy inferior a 20.9093).

El resto nos da la ecuación del modelo; en efecto, nuestro modelo es lineal y depende de dos variables, de ahí que conozcamos su forma:

$$y = a.x1 + b.x2 + c$$

Podríamos leer:

Coefficients:

	Estimate
(Intercept)	1.00921
x1	2.04257
x2	-6.95545

Por lo que:

$$y = 2.04257 * x1 - 6.95545 * x2 + 1.00921$$

Es la ecuación de un plano en tres dimensiones, el valor de c (**Intercept**) representa el desfase respecto a un plano que pasaría por el origen.

Los *** después de los valores «a, b, c» significan que se puede confiar bastante en ellos.

Vamos a tratar de manipular este modelo nosotros mismos. Como no tenemos la intención de copiar a mano los coeficientes, lo que resultaría una fuente de errores, vamos a utilizar el hecho de que la función **summary()** devuelva de hecho un objeto cuyo contenido podemos leer. Este objeto es **f**.

```

a <- f$coefficients[2]    # extracción de los coeficientes del modelo
b <- f$coefficients[3]

```

```

c <- f$coefficients[1]

f_ <- function(x1,x2){      # recreación de la función del
    a*x1+b*x2+c            # modelo
}

```

Hemos creado una función de dos variables llamada **f_()**, que expresa la ecuación lineal correspondiente a nuestro modelo.

A continuación, podemos tratar de dibujar nuestra función en el espacio: por construcción se trata de un plano en el espacio 3D.

 Un hiperplano es una combinación lineal de una dimensión inferior en una unidad a la dimensión del espacio. Por ejemplo, una línea recta ($y = ax + b$) es un hiperplano en un espacio en 2D, y un plano es un hiperplano en un espacio en 3D.

```

x1_ <- seq(min(x1), max(x1), by=0.2) # malla x1
x2_ <- seq(min(x2), max(x2), by=0.2) # malla x2
y_ <- outer(x1_, x2_, f_)           # un valor por pareja

persp(x1_,
      x2_,
      y_,
      phi=40,                      # ángulos de presentación
      theta=60,
      col= "green",
      ticktype = "detailed",       # eje detallado
      shade=.0001,
      cex.lab=1.0 )                # preste atención a ampliar
                                    # para visualizarlo

```

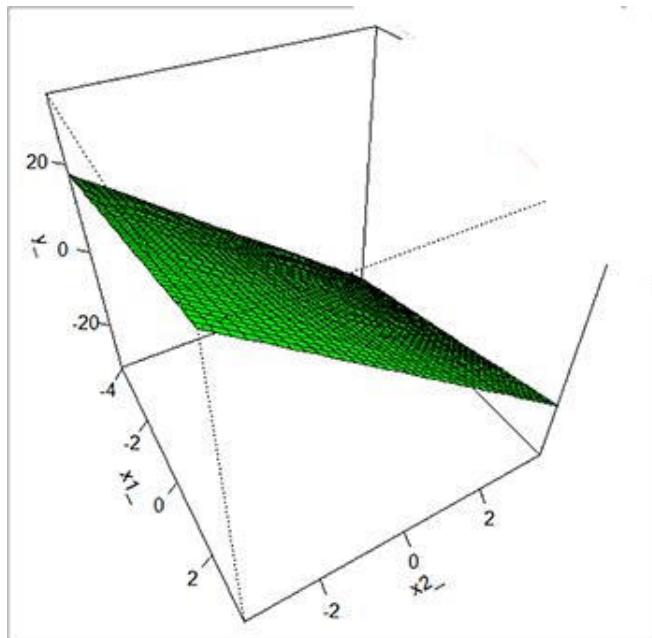
Este código nos permite abordar el uso de la función **persp()** de R, que fabrica representaciones en 3D.

En primer lugar, se fabrican una serie de valores ficticios de **x1** y de **x2** que cubren el espacio máximo de todos los valores efectivos de **x1** y **x2** con un «paso» de 0.2. De hecho, esto nos permitirá fabricar distintas combinaciones de parejas de valores (**x1_**, **x2_**) y, a continuación, aplicarles la función **f_()**.

Se obtienen 39 valores para **x1_** y 38 para **x2_**.

A continuación, aplicando **f_()** sobre el conjunto de las 1482 parejas (**x1_**, **x2_**) mediante la función **outer()**, se obtiene una matriz **y_** de dimensión 39 por 38 que contiene los 1482 valores correspondientes **f_(x1_,x2_)**.

Este trabajo preliminar nos permite configurar las estructuras de datos esperadas por la función **persp()**; los demás parámetros son parámetros de presentación. El resultado podrá parecer minúsculo en su pantalla, de modo que habrá que ampliarlo para obtener la siguiente figura:



Modelo de predicción lineal $y=f(X)$

Hemos obtenido el modelo. Resulta muy interesante en sí mismo, aunque no nos muestra cómo se reparten **realmente** las observaciones en relación con este modelo. Para obtener una visualización, aunque imperfecta, de los puntos del conjunto de entrenamiento, vamos a utilizar el siguiente código. Como existen 6000 puntos, trazaremos solamente un pequeño porcentaje aleatorio de ellos, pues en caso contrario el diagrama será incomprendible. Para distinguir los puntos de arriba de los de abajo, vamos a jugar con los colores y la forma de los puntos, utilizando algunos trucos.

El código transformado es:

```

palette <- colorRampPalette(c("red","yellow"))(100)
diag <- persp(x1_,
              x2_,
              y_,
              phi=40,                      # ángulos
              theta=60,
              col=palette[
                round((y_+max(y_))/(max(y_)-min(y_))* 100)],
              ticktype = "detailed",        # eje detallado
              shade=.0001,
              cex.lab=1.5 )                 # preste atención a ampliar
                                         # para visualizarlo

index_ <- createDataPartition(y=Z$y,      # extracción del índice
                             p= 2/100,    # que es una serie de
                             list=FALSE) # 6000 números aleatorios

u <- as.vector(matrix(Z[index_,1]))
v <- as.vector(matrix(Z[index_,2]))
w <- as.vector(matrix(Z[index_,3]))

nube <- trans3d(u,                  # nube de puntos
                 v,
                 w)

```

```

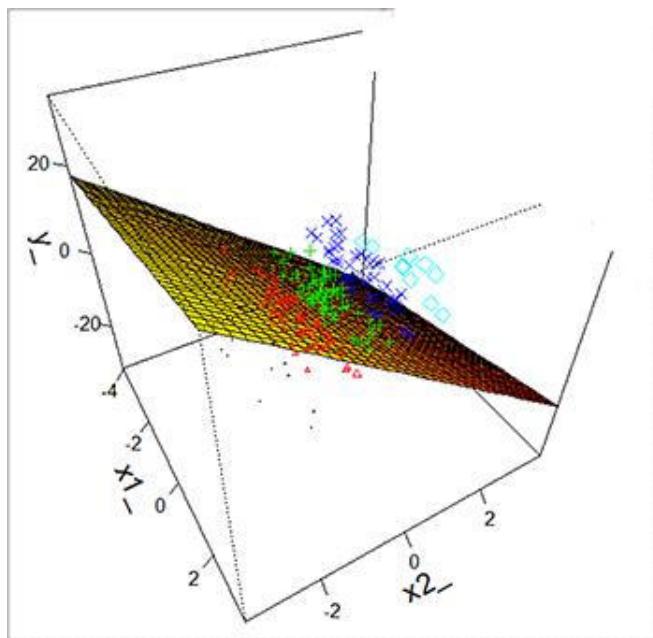
v,
pmat = diag)

points(nube,           # inserción en el diagrama
       pch = v + max(v)+1,    # jugar con la forma de los símbolos
       cex = v/3.5 + max(v/3), # jugar con el tamaño de los símbolos
       col = v + max(v)+1    # jugar con el color de los símbolos
)

```

El código es un poco más rico. En primer lugar, nos dotamos de una paleta de colores, que nos ofrece más de 100 códigos de colores (como los colores en HTML). A continuación, fabricamos el diagrama que almacenamos, pues vamos a mostrarlo a continuación. En los parámetros, el color forma parte de una pequeña fórmula cuyo objetivo es seleccionar un color diferente de la paleta para cada valor de $y_$. La idea de esta expresión es no tener valores negativos y cubrir el centenar de índices de colores de la paleta, y también disponer de valores de índices enteros (sin coma decimal). Después se extrae el 2 % de los puntos de nuestro conjunto de prueba y se declara la nube de puntos aplicando una función especializada. Solo nos queda trazar los puntos en el diagrama almacenado antes haciendo variar el tamaño de los símbolos en función del valor de $f(X)$ y haciendo variar los colores y los símbolos de forma relativamente aleatoria.

Obtenemos el siguiente diagrama.



Modelo predictivo frente al 2 % de las observaciones

En términos de interpretación geométrica y «matemática», ha visualizado un plano afín, a partir del subespacio vectorial $\text{Vect}(\bar{X}_1, \bar{X}_2)$ y los puntos (vectores) del conjunto de entrenamiento. El plano contiene la proyección ortogonal de los puntos correspondientes a sus observaciones, tales que la media entre el plano y los puntos del conjunto de entrenamiento es mínima. Para visualizar este modelo 3D de una manera algo más prosaica y sobre todo que nos permita utilizarlo para comprender su topología o utilizarlo como un ábaco que nos permita hacernos una idea del valor de y en función de x_1 y x_2 , disponemos de otra función en R. Es menos espectacular, aunque resulta muy útil, pues nos permite trazar contornos, es decir, líneas de nivel que se obtienen si observamos el siguiente diagrama.

```

# líneas de nivel
contour(y_,
        method = "edge",

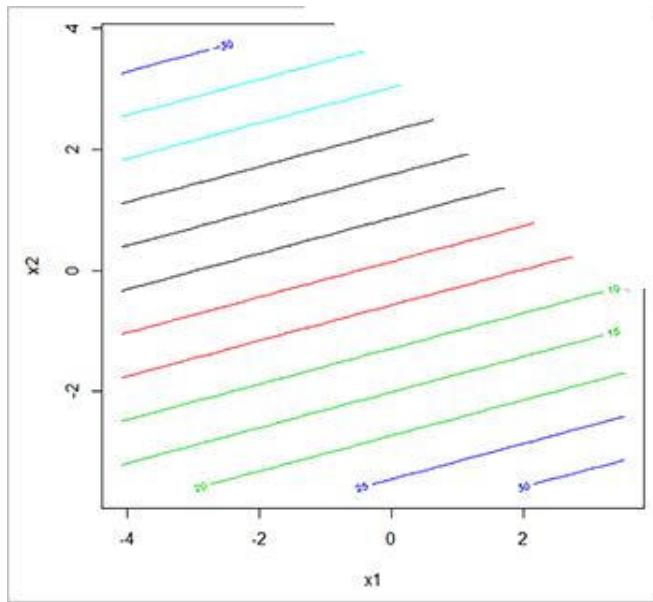
```

```

xlab="x1",
ylab="x2",
col=round(abs(y_))%%5+1) # juego de colores

```

Lo que produce:



Ábaco de los valores de y en función de X - Curvas de nivel

Gracias a esta representación, podemos comprobar que trabajamos sobre un plano (las líneas de nivel son paralelas y equidistantes). Por otro lado, podemos aproximar visualmente el valor estimado de **y** en función de **x1** y **x2**. Por ejemplo, sobre el diagrama podemos leer que para un valor de **x1 = -2** y **x2 = 2** tenemos un valor de **y** comprendido entre -20 y -15. También podemos visualizar la pendiente en cada punto, que no es más que la perpendicular a las líneas de nivel (si no fuera un plano, también habríamos podido deducir un aumento en la pendiente allí donde las líneas se estrecharan).

💡 Viendo esta representación, podemos comprender el uso de la palabra gradiente, utilizada en las técnicas de optimización. Descenso por gradiente aquí sería desplazarse perpendicularmente respecto a las líneas de nivel, representadas por otro lado como un gradiente de colores.

8. Conclusión sobre el modelo lineal

Hemos fabricado un conjunto de datos ficticios a partir de una relación lineal que hemos perturbado con un ruido importante revestido con cierta aleatoriedad (la prueba es el volumen de la nube de puntos). Sin embargo, el algoritmo de regresión lineal múltiple ha sido capaz de extraer la información correctamente y encontrar los coeficientes de regresión con una gran eficacia.

La ecuación utilizada para preparar el conjunto de datos ficticios y notablemente perturbados que nos ha servido para elaborar este conjunto de observaciones era:

$$y = f(X) = 2 \cdot x_1 - 7 \cdot x_2 + 1$$

El modelo resultado de nuestro trabajo es:

$$y = f(X) = 2.04257 * x_1 - 6.95545 * x_2 + 1.00921$$

En cuanto a la linealidad de los datos, el algoritmo de regresión lineal, simple y con muy buen rendimiento, presenta una eficacia formidable para ponerla de relieve. De hecho, encontraremos el uso de los conceptos subyacentes a la linealidad en numerosos algoritmos de machine learning.

Debido al hecho de que conocemos cómo se ha construido el conjunto de datos, podemos presumir una peor eficacia para los demás algoritmos sobre este conjunto de datos, o llegado el caso un cierto riesgo de overfitting que integraría parte del ruido aleatorio que hemos introducido.

Exploraremos ahora lo que obtendríamos utilizando otro algoritmo, más agnóstico en cuanto a la linealidad, sobre estos datos. Para realizar esta prueba, vamos a utilizar Random Forest (un algoritmo que utiliza en «ensemble», es el término consagrado, varias instancias de un algoritmo de árbol de decisión). Este algoritmo posee una gran universalidad en cuanto a sus hipótesis de uso. Veamos en qué medida su rendimiento será peor o igual (que fuera superior sería sorprendente, ipues habíamos preparado un juego de datos basado en la linealidad!).

9. Uso de un modelo «Random Forest»

Para utilizar este algoritmo, vamos a cargar el paquete correspondiente y utilizar un código muy parecido al anterior.

```
# prueba con el algoritmo randomForest          #
# aplicación sobre el conjunto de validación    #

require(randomForest)           # bosque de árboles de decisión
f <- randomForest(y ~ x1+x2,
                  data=Ze,
                  ntree = 30)  # creación del modelo y=f(X)
yv_ <- predict(f,newdata = Zv)  # f(X)
yv_ <- matrix(yv_)
```

Esta es la belleza de R: salvo los parámetros específicos a nuestro modelo, utilizaremos la misma sintaxis.

Para obtener los diagramas de residuo, el código es exactamente el mismo que antes.

Siempre realizando los mismos cálculos, pero almacenándolos en un **data.frame** llamado **id2** y haciendo:

```
# comparación de indicadores
id <- rbind(id1,id2)
id
```

Se obtiene una comparación de nuestros valores calculados. Echemos un vistazo a NRMSE y R2:

```
NRMSE
0.03505314
0.03775296
```

```
R2
0.8861678
0.8679577
```

Los valores están ligeramente degradados, pero esta diferencia no parece particularmente significativa.

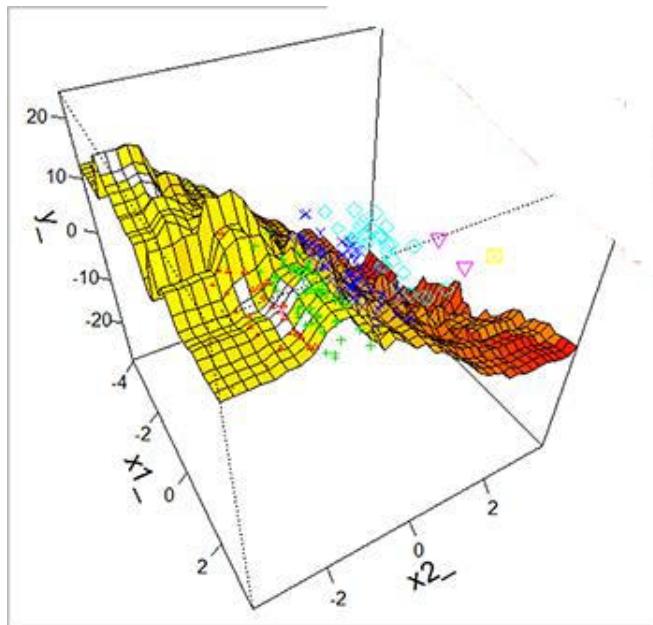
Veamos esto mediante una representación en el espacio utilizando un código similar al usado previamente, pero con una manera de definir la función por representar que será más general (y que podremos **utilizar sobre muchos otros modelos**).

Nuevo código para la función que se ha de representar:

```
f_ <- function(x1,x2){      # creación de la función de predicción
  x1 <- matrix(c(x1))
  x2 <- matrix(c(x2))
  y  <- matrix(c(1))
  linea_X <- data.frame(x1,x2,y)
  predict(f,newdata = linea_X )
}
```

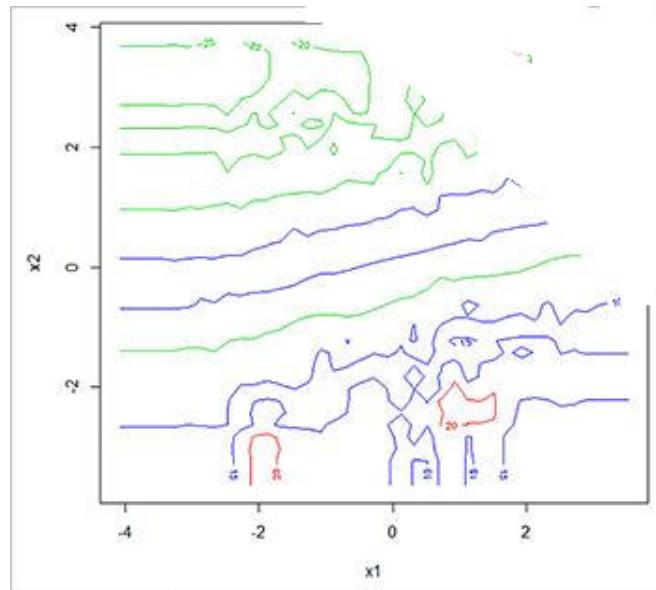
En el código no se utiliza ningún conocimiento acerca del tipo de modelo escogido, nos contentamos con invocarlo.

El siguiente código es siempre el mismo (salvo los parámetros de los colores) y se obtiene esto:



Predicción con Random Forest y puntos reales

Con las curvas de nivel, obtenemos:



Curvas de nivel, Random Forest

El rango de valores de y es menos extenso que el identificado en el modelo lineal, de -25 a 25 en lugar de -30 a 30.

La topología de este modelo es manifiestamente menos lineal fuera de los rangos de y que van de -5 a 10, y sin embargo hemos visto más arriba que su rendimiento no era necesariamente peor observando el valor normalizado de la raíz del error cuadrático medio (NRMSE) y R2.

El nivel de correspondencia de nuestras herramientas de evaluación de modelos en este capítulo del libro no nos da una mayor capacidad para escoger entre ambos modelos. En tal caso, y sin herramientas e investigaciones suplementarias, no dude y tome el modelo que parezca más sencillo, es decir, el que parezca buscar menos los detalles de variación de X (en cierta manera el más fluido). Este modelo será, sin duda, el más generalizable a otros datos, el que tenga menos «overfitting».

Y ahora disponemos del bagaje que nos va a permitir abordar con confianza la comprensión y el uso de las diversas técnicas de machine learning, así como la capacidad para abordar numerosos artículos de investigación.

Técnicas y algoritmos imprescindibles

Construir la caja de herramientas

A modo de introducción a este capítulo, enumeraremos de forma breve aquellos elementos difícilmente prescindibles para poder trabajar con los datos en R y obtener cualquier cosa:

- Un panel de técnicas de visualización para comprender los datos, preparar la acción y controlar visualmente el resultado del trabajo realizado.
- Puntos de referencia teóricos para orientarse entre las distintas opciones técnicas, algunos algoritmos buenos de clasificación, de regresión y de clustering para apoyarse en ellos.

Esto constituye, más o menos, el conjunto de este capítulo.

De hecho, tras abordar los capítulos anteriores, disponemos ahora de un kit de herramientas bastante respetable sobre ciertos asuntos:

- Relativo al aspecto gráfico: plots, diagramas por pares, histogramas, densidad, diagramas de caja, «facet» por clase, 3D, curvas de nivel/contornos, visualización interactiva multiventana...
- Relativo a la clasificación y la regresión: indicadores y pruebas diversas, árbol de decisión, regresión lineal multivariante y Random Forest.

Exploraremos ahora algunas técnicas complementarias a las que ya hemos estudiado. Esta elección es empírica, los elementos se han seleccionado basándose en uno o varios de los siguientes tres criterios:

- Comportar un aspecto didáctico.
- Tener una utilidad práctica.
- Ser un conocimiento compartido con los demás practicantes.

Representación gráfica de los datos

El paquete **ggplot2**, cuya sintaxis pretende ser una «gramática gráfica», se utilizará de manera regular en esta sección. Muchos parámetros de esta gramática son comunes con los del paquete básico de R.

Los ejemplos se realizan sobre juegos de datos ya utilizados antes, y están disponibles en el sitio web que acompaña a este libro.

Vamos a abordar la forma y la sintaxis de diversos gráficos, y también **intentaremos interpretarlos** brevemente: «la práctica hace al maestro».

1. Un gráfico «simple»

La manera de construir un gráfico que vamos a ver a continuación resulta interesante: se trata de un proceso que se basa en una gramática gráfica que vamos a explorar paso a paso.

Estos gráficos van a construirse de forma estructurada, globalmente:

- Cargando los datos en una estructura gráfica.
- Creando un juego de parámetros.
- Agregando representaciones gráficas por sucesivas capas (de geometrías diversas).
- Llegado el caso, agregando un tema que transforme el conjunto.
- Mostrando el gráfico.

Carguemos el paquete.

```
library(ggplot2)
```

Construyamos nuestro primer gráfico, el más útil: un «scatterplot» con una regresión rodeada por su zona de confort. Usaremos un **data.frame** ya utilizado previamente, cuyas variables numéricas van de **v1** a **v12** y que incluye una variable «**factor**» llamada «**clase**».

```
## ejercicio sobre la gramática de ggplot2 ##

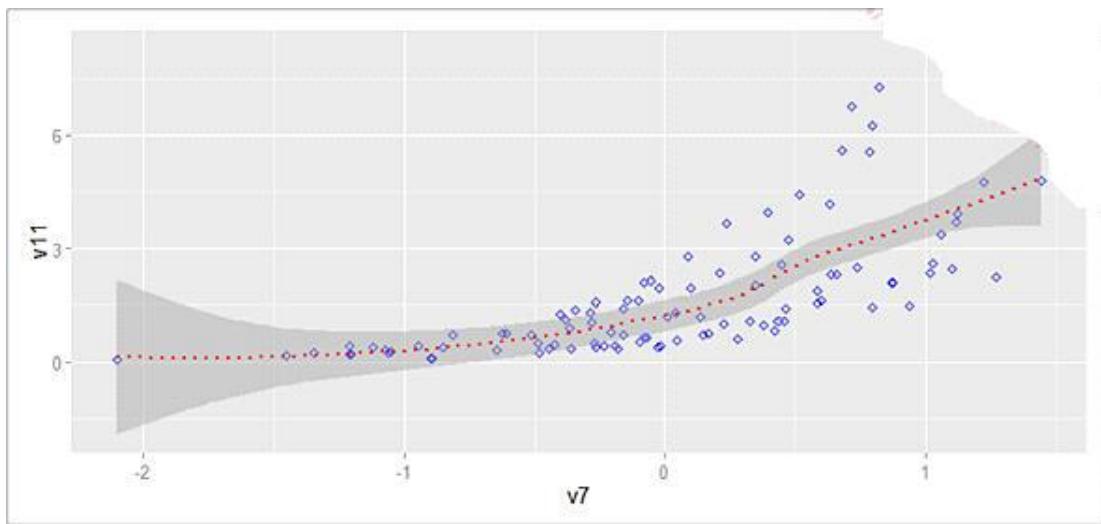
## un gráfico sencillo: v11 en función de v7 con regresión ##

p <- ggplot(data=df2) # escogemos el dataset
al<- aes(x=v7,y=v11) # lista de las variables y parámetros
# creación del plot p1
p1<- p+
# al menos un parámetro de geometría:
  geom_point(al, # queremos puntos
              col = "blue", # azules
              pch = 5) # con forma de rombo
  geom_smooth(al, # y una regresión
              col = "red", # de color rojo
              size = 1, # grosor de la línea
              linetype = 3) # línea con pequeños puntos
```

```
p1<- p1 + theme(text=element_text(size=16)) # tamaño de la letra
p1                                # visualización
```

Observe los siguientes aspectos del código:

- El plot es una estructura de datos **p** (luego **p1**, pero podríamos haber guardado **p**).
- Los parámetros estructurantes son también un objeto: **a1 <- aes()**.
- **aes()** significa «aesthetics», es decir, «estético» en español: son los parámetros...
- Se realizan sucesivas adiciones: **p1 <- p + geom.. + geom + ...+ theme**.



ScatterPlot con regresión y zona de confort

Este tipo de gráfico expresa simplemente el tipo de relación entre dos variables numéricas. Es una de las primeras fuentes de información sobre la realidad que hay que estudiar (caso de variables numéricas).

En un intervalo aproximado de -1.5 a 0, los puntos están en la zona gris, y podemos imaginar que esta regresión está operacional en este intervalo. Por el contrario, tras el 0, la regresión no es más que el resultado de dos componentes diferentes.

En tal caso, se nos presentan diversas opciones de análisis, en función de los datos disponibles. **Buscaremos una variable que nos permita dividir claramente ambas componentes** llamadas con cierta pomposidad «trayectorias» (por ejemplo, la variable clase en este caso, aunque podría ser otra). Si esta variable no existe, habrá que intentar construirla, bien **componiendo distintas variables** y aplicando diversas transformaciones, o bien aplicando un **algoritmo de clusterización** con la idea de separar ambas poblaciones y crear una nueva feature «**tipo de población en función del algoritmo A - entrenado sobre el data set DF - versión del...**».

Evidentemente, si fallan los métodos anteriores, siempre es posible tratar de identificar los puntos de las distintas zonas que parecen correctamente separados en el esquema. Esto permite a continuación asignarles etiquetas para encontrar clústeres «a mano», salvo si alguna de las variables es la variable que se debe explicar, pues este método no sería reproducible sobre los conjuntos de prueba y de validación.

Observe, viendo los conjuntos de prueba y de validación, que cualquier método que cree nuevas features debe apoyarse precisamente en la formulación que se ha realizado sobre los datos de entrenamiento, sin cambiar la configuración, pues el valor de la nueva feature **no debe cambiar en función de los elementos** del conjunto de datos utilizado.

Si cometemos el error de no usar el resultado de la mecánica que se ha utilizado sobre el conjunto de entrenamiento para crear las nuevas features, sino reutilizar el propio mecanismo y aplicarlo sobre los datos de prueba y de validación, el conjunto de elementos de estos datasets influye inevitablemente en el valor de la nueva feature, lo cual resulta particularmente nocivo. Para convencerse, imagine un conjunto de prueba que contuviera una única observación... ¿cómo aplicar un mecanismo de clustering para crear una nueva feature?

Este tipo de mecánica manual solo tiene sentido si está completamente convencido de no haber captado ciertas **variables exógenas** a su conjunto de observaciones y si el número de variables es relativamente pequeño. Sin prohibirlas, evite las «alteraciones manuales», salvo si el problema supera sus capacidades algorítmicas. Y, en tal caso, isea más **prudente** que de costumbre en la interpretación de los resultados y la generalización de los modelos!

2. Histogramas avanzados

a. Distribución multiclasé

Los histogramas representan la distribución empírica de las variables, y su aspecto permite determinar visualmente si será posible manipular de forma fácil la variable correspondiente: si posee una forma unimodal y simétrica como una ley normal, el trabajo se verá facilitado; si no... habrá que pelear.

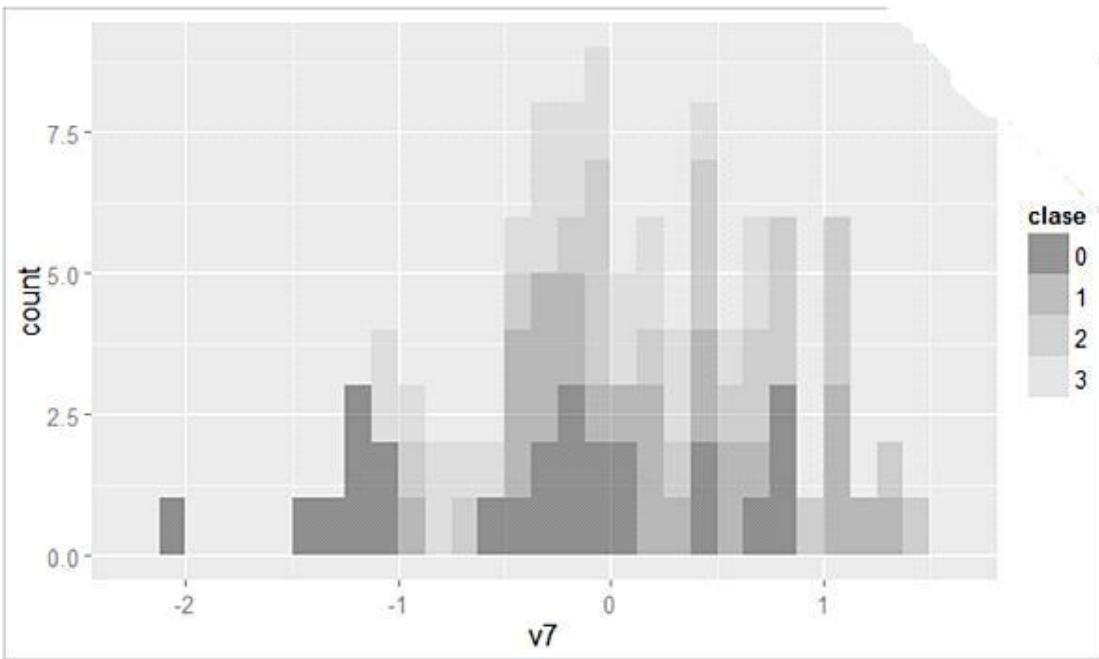
```
## un histograma con varias clases ##

p <- ggplot(data=df2)          # escogemos el dataset
a1<- aes(x=v7, fill = clase)  # lista de variables y parámetros
                                # una densidad por clase

                                # creación del plot p1
p1<- p+
      geom_histogram(a1,        # muestra el histograma
                      binwidth =0.125, # ancho de las bandas
                      alpha = ½) # transparencia del histograma

p1<- p1 + theme(text=element_text(size=16)) # tamaño de la letra
p1                                # visualización
```

Observe el parámetro **fill**, que permite introducir elementos de clase, y el parámetro **binwidth** para ajustar por sucesivos tantos para que el histograma se parezca a algo.



Distribución de v7 y efectivos de clase en v7

Vemos que por debajo de -1 la clase 0 está sobrerepresentada y que por encima de un valor en torno a 0.9 las clases 0 y 3 ya no están representadas. Es muy posible que v7 asociada a otra información pueda ser una feature útil para predecir la clase.

Para pasar un gráfico **ggplot2** a blanco y negro en el contexto de un libro en blanco y negro, utilizamos la siguiente instrucción:

```
p1 <- p1 + scale_fill_grey() # para una edición en blanco y negro
```

b. Mezcla de varias distribuciones por clase

Muy similar en su construcción, y esta es precisamente la belleza de la gramática de **ggplot2**, el siguiente histograma está mejor adaptado a la variable v2.

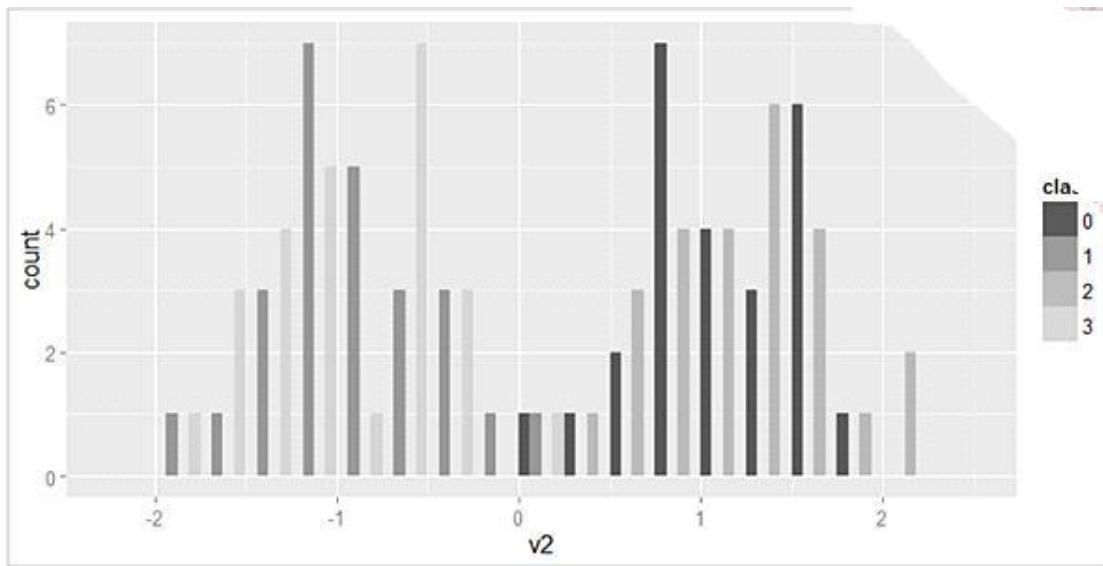
```
## un histograma con varias clases de lado a lado ##

p <- ggplot(data=df2)          # escogemos el dataset
al<- aes(x=v2, fill = clase) # lista de variables y parámetros

                    # creación del plot p1
p1<- p+             # al menos un parámetro de geometría
geom_histogram(al,    # muestra el histograma
              binwidth = 0.25, # ancho de las bandas
              position='dodge', # lado a lado
              alpha = 4/5) # transparencia del histograma

p1<- p1 + theme(text=element_text(size=16)) # tamaño de letra
p1                         # visualización
```

El parámetro **position = 'dodge'** ha transformado radicalmente el aspecto del histograma.



Distribución de v2 y mezcla de clases en el seno de v2

Vemos que **v2** separa completamente el grupo de las clases 1 y 3 del grupo de las clases 0 y 2. Se trata de un tipo de «predictor» que resulta fiable si se utiliza solo para predecir «**clase**» y muy potente si se asocia a uno o n «predictor(es)» que contendría(n) información complementaria (típicamente como «o exclusivo» de cara a **v2**: caso llamado XOR).

c. Visualización de la densidad de una distribución

Tras el «scatterplot», he aquí sin duda el gráfico más importante. Existen muchas maneras de producirlo. A continuación presentamos la versión con **ggplot2**.

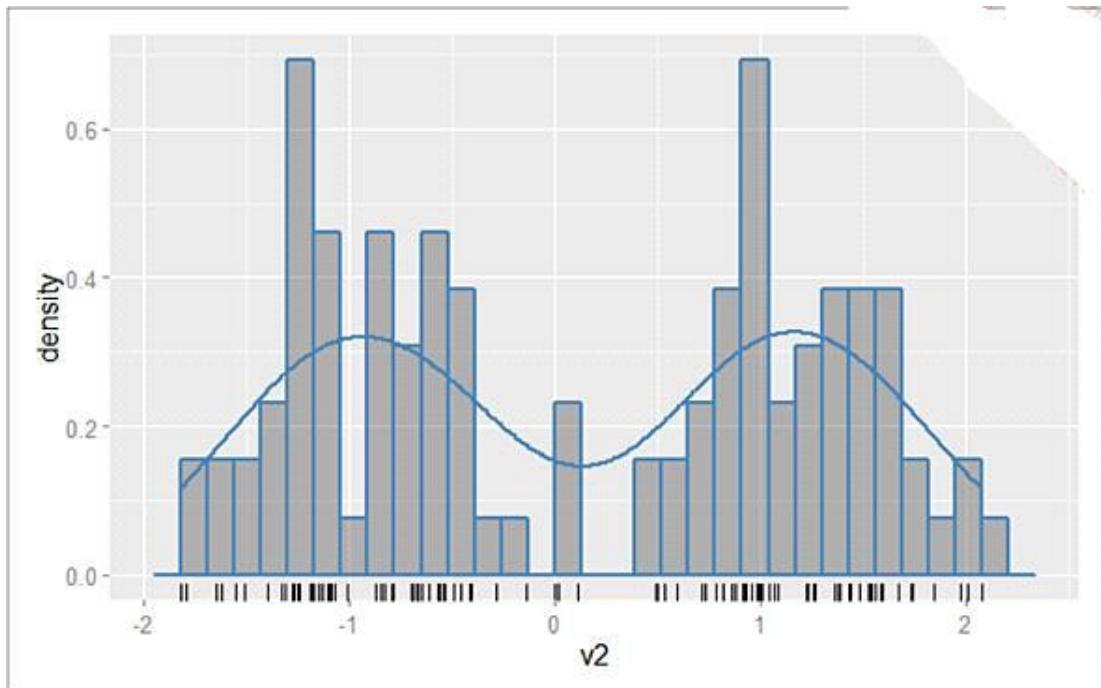
```
## un histograma con density ##

p <- ggplot(data=df2)      # escogemos el dataset
a1<- aes(x=v2,..density..) # variable y parámetros
a2<- aes(x=v2)            # variable

# creación del plot p1
p1<- p+                   # al menos un parámetro de geometría
  geom_histogram(a1,
    size = 1,
    col= "steelblue",
    alpha = 1/3) +
  geom_density(a1,
    size = 1,
    col= "steelblue",
    alpha = 1/3) +
  geom_rug(a2)

p1<- p1 + theme(text=element_text(size=16)) # tamaño de la letra
p1
```

Agregar la geometría **rug()** nos permite identificar las pequeñas variaciones de densidad (visibles según el eje de las x).



Histograma y densidad de **v2**

En vista de su densidad, la distribución de esta variable es bimodal. Observando la banda *rug*, se confirma que el pico izquierdo de la distribución está un poco por debajo de -1, lo cual no se corresponde exactamente con la cima de la densidad que sugiere una información sintética y vinculada al conjunto.

El aspecto de un histograma puede cambiar notablemente en función del ancho de sus columnas (barras). **La densidad y el «rug» le ayudarán y evitarán interpretaciones aventuradas.**

► ¿Había observado que la banda «rug» puede recordar a un tapiz algo tupido... como un felpudo?

d. Otra mezcla por clase

En ocasiones, se quiere destacar las **proporciones de las diferentes clases** en función de los valores de la variable, en cuyo caso hay que utilizar un histograma de tipo «stack».

```
## un histograma "stacked" (apilado)->muestra las proporciones ##
## entre las clases para los distintos valores ##

p <- ggplot(data=df2)           # escogemos el dataset
a1<- aes(x=v2,fill = clase)    # lista de variables y parámetros

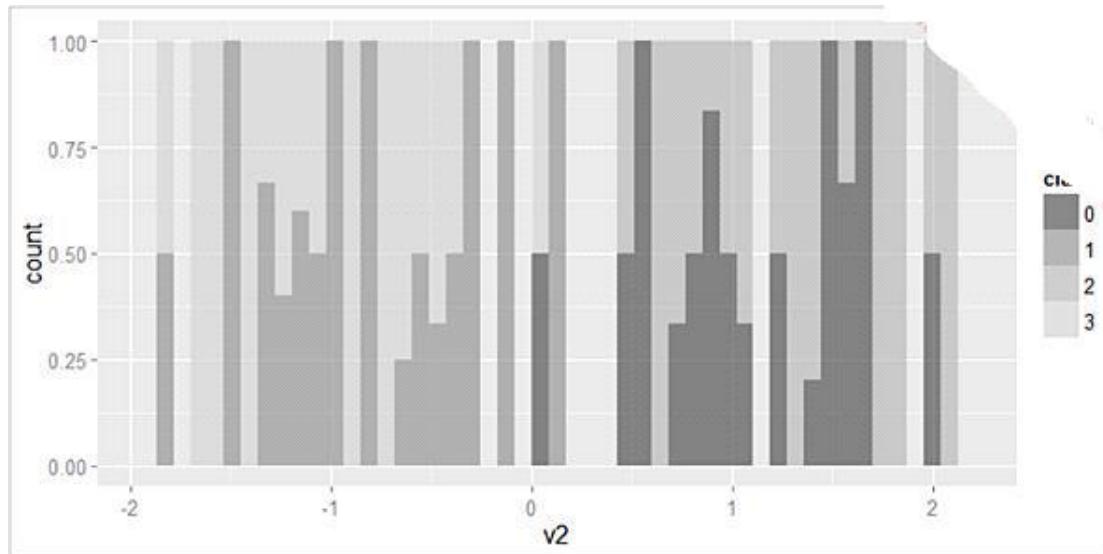
# creación del plot p1
p1<- p+                         # al menos un parámetro de geometría
  geom_histogram(a1,               # muestra el histograma
                 binwidth = 0.085, # ancho de las bandas
                 position='fill',# sin recubrir las barras
                 alpha = 4/7) # transparencia del histograma
```

```

p1<- p1 + theme(text=element_text(size=16)) # tamaño de la letra
p1                                     # visualización

```

Observe la posición **position = 'fill'**.



Histograma de las proporciones por valores y por clases

Encontramos todavía más claramente la capacidad de separación de la variable entre las clases 1 y 3 y 0 y 2.

e. Una variable, pero un histograma para cada clase

Cuando hay que construir un gráfico haciendo variar uno o dos parámetro(s), se utiliza la noción de «facet», como en los cubos OLAP, y se presenta una sucesión de pequeños gráficos. Esto ya lo hemos visto brevemente con anterioridad; veamos ahora cómo hacerlo con **ggplot2**.

► Otro paquete especializado en los «facets» se denomina lattice (= red/malla de facets).

```

## facet: varias curvas en función de la clase      ##

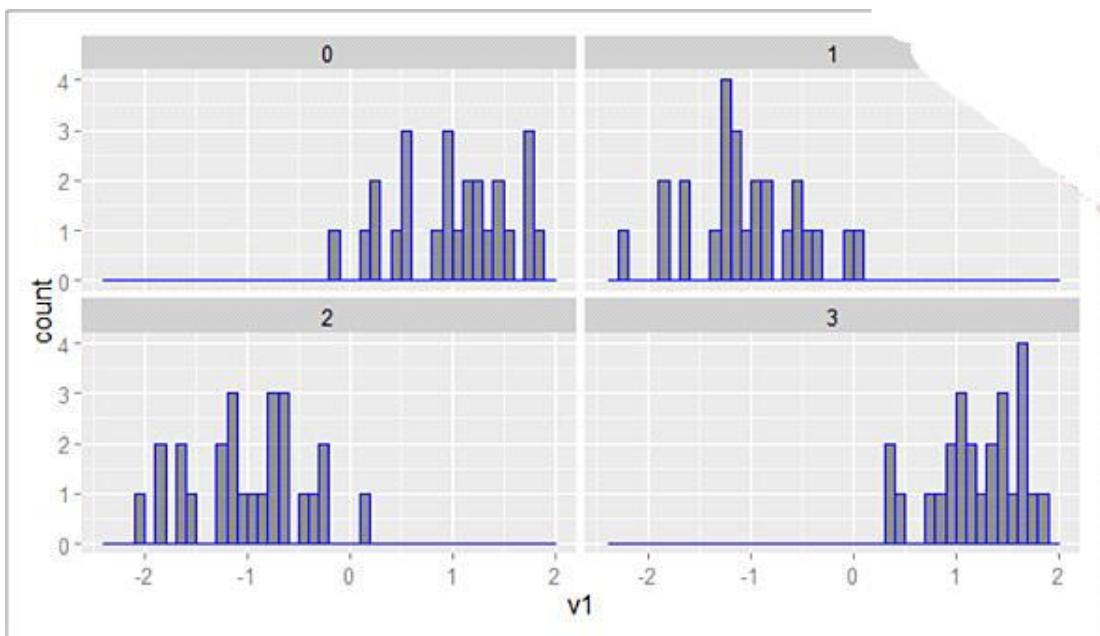
p <- ggplot(data=df2)          # escogemos el dataset
al<- aes(x=v1)                # lista de variables y parámetros

                                    # creación del plot p1
p1<- p+                         # al menos un parámetro de geometría
        geom_histogram(al,          # muestra el histograma
                          binwidth = 0.1, # ancho de las bandas
                          position='dodge',
                          col = 'blue',
                          alpha = ½) # transparencia del histograma

p1<- p1 + facet_wrap(~clase)
p1<- p1 + theme(text=element_text(size=16)) # tamaño de la letra
p1                                     # visualización

```

Agregar un elemento geométrico suplementario resulta decisivo: `+facet_wrap(~clase)` divide el gráfico en función de la clase.



Distribuciones de `v1` divididas por valor de clase

Se confirma la potencia de la implementación de la gramática gráfica de `ggplot2`. No ha sido necesario utilizar ni un solo bucle ni invocar una configuración concreta...

Con numerosas clases, esto resulta una de las raras maneras de proceder.

Hemos cambiado de variable para demostrar que `v1` contiene una mezcla diferente de la de `v2`. Retomaremos más adelante este conjunto de datos en los intentos de clasificación y estas conclusiones nos ayudarán a comprender mejor los resultados asociados a estos ejemplos.

f. Gráfico con una densidad por clase

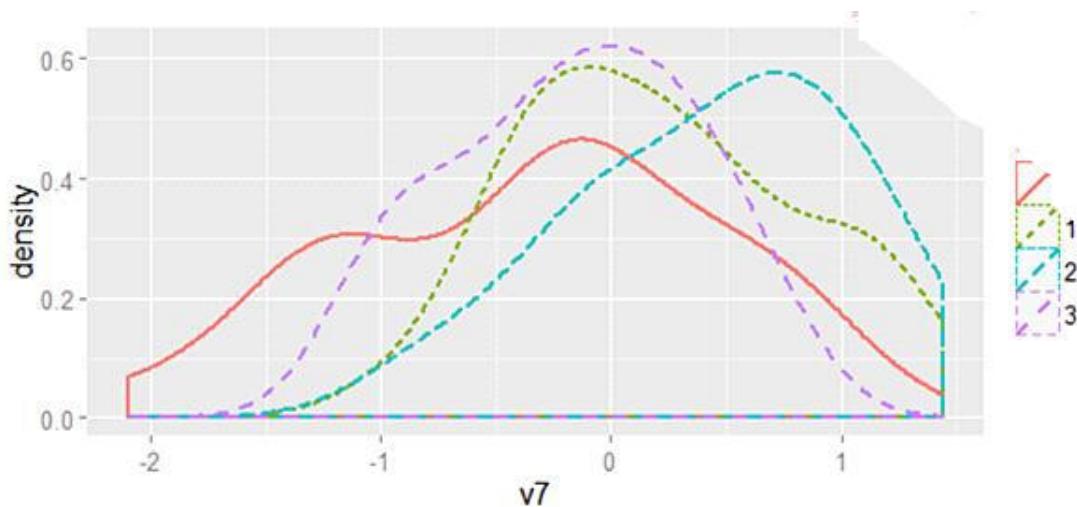
Representa uno de los gráficos más fáciles de interpretar de cara a una mezcla algo más rica de distintas distribuciones dentro de una variable.

```
## un gráfico con densidad por clase ##

p <- ggplot(data=df2)           # escogemos el dataset
a1<- aes(x=v7, col = clase,
          linetype = clase) # lista de variables y parámetros
                      # una densidad por clase

                      # creación del plot p1
p1<- p+
          # al menos un parámetro de geometría
          geom_density(a1,      # muestra density
                        size = 1,    # grosor de la línea
                        )
p1<- p1 + theme(text=element_text(size=16)) # tamaño de la letra
```

Observe el parámetro **linetype**.



Densidades de las clases en el seno de v7

Cuando **v7** es inferior a un valor que parece ser -1.6, tan solo tenemos la clase 0 presente y, por tanto, podríamos escribir una probabilidad condicional como: $\mathbb{P}(\text{clase} = 0 | v7 < -1.6) \approx 1$.

En sí, es un elemento que podría figurar en un árbol de decisión. Esto debería formar parte, por otro lado, de lo que querrá comprobar tras la implementación de cualquier modelo predictivo. Debería **juzgar la calidad de sus modelos mediante dichas comprobaciones realizadas a priori y a posteriori**. Un modelo eficaz debería ser capaz de contener esta probabilidad condicional (si fuera necesario, utilice el método de Monte Carlo como se ha mostrado en el primer capítulo para comprobarlo de cara a un modelo complejo).

➤ Digresión matemática para juzgar la interpretación de dicho gráfico: «Juego con el teorema de probabilidades condicionales (Bayes) abordado en el capítulo Dominar los fundamentos».

Si se cuenta aproximadamente el número de rectángulos sobre la curva de densidad de la clase 0 (cuente los rectángulos enteros y a continuación sume la mitad del número de rectángulos que estén completamente atravesados por la curva), se obtiene cerca de 19 (15 + 4) rectángulos. Bajo la curva en cuestión, a la izquierda de -1.6, se encuentran unos 1.5 rectángulos. De hecho, sería preciso hacer este cálculo bajo la curva de densidad total de **v7**, pero esto no va a cambiar demasiado el resultado. Se obtiene una relación cercana al 8 % : $\mathbb{P}(v7 < -1.6) \approx 0.08$. Por otro lado, conocemos las proporciones de cada clase, contando solamente nuestro **data.frame** de entrenamiento. Hemos visto en los capítulos anteriores que los elementos de las cuatro clases eran iguales, de modo que: $\mathbb{P}(\text{clase} = 0) \approx 1/4$

Hemos visto en el capítulo Dominar los fundamentos que:

$$p(x|y).p(y) = p(y|x).p(x)$$

Con x: (clase = 0)

e y: ($v7 < -1.6$)

se obtiene: $1 \times 0.08 \approx p(y|x) \cdot 1/4$

por tanto $\mathbb{P}(v7 < -1.6 | clase = 0) \approx 0.32$

Imaginando que nuestras cuentas fueran correctas, sabemos que, cuando la clase es igual a 0, tenemos alrededor de una oportunidad sobre tres de que **v7** sea inferior a -1.6.

Esto no resulta nada intuitivo... es la magia del teorema de Bayes.

Evidentemente, en la vida real, dispone de la función «density» para realizar cálculos más precisos; el gráfico le permitirá saber si resulta oportuno realizar un cálculo u otro.

También podemos destacar que el modo de la clase 2 se declara respecto a las demás clases.

Estas observaciones le pueden hacer suponer que la variable **v7** posee también cierto poder de discriminación sobre la variable de respuesta «**clase**». Sería elegible en primera instancia en una lista de features utilizables para prever la clase, pero a condición de implementar un algoritmo bastante eficaz para utilizar las características de **v7** respecto a las demás variables.

3. Diagrama de pares y de facetas

Vamos a retomar los diagramas por pares o de facetas.

Tras la implementación de gráficos sencillos y corrientes, utilizaremos el paquete **ggplot2** para ir más lejos.

a. Diagrama por pares, versión simple

Vamos a visualizar las cuatro primeras columnas del **data.frame** Longley separando visualmente los años inferiores o iguales a 1950 de los demás.

```
## visión general de los datos dos a dos
plot(df1[,1:4],
      col = ((df1$an <= 1950)+1), # el color de los puntos cambia en 1950
      pch = ((df1$an <= 1950)+3) # la forma de los puntos cambia en 1950
      )                         # +1 y +3 para desplazar los
                                # colores y las formas
```

La sintaxis es de una enorme simplicidad y conviene memorizarla. Se definen los colores y los símbolos en función de la columna año y para obtener los colores y los símbolos deseados basta con agregar un desfase sobre las columnas y los símbolos.

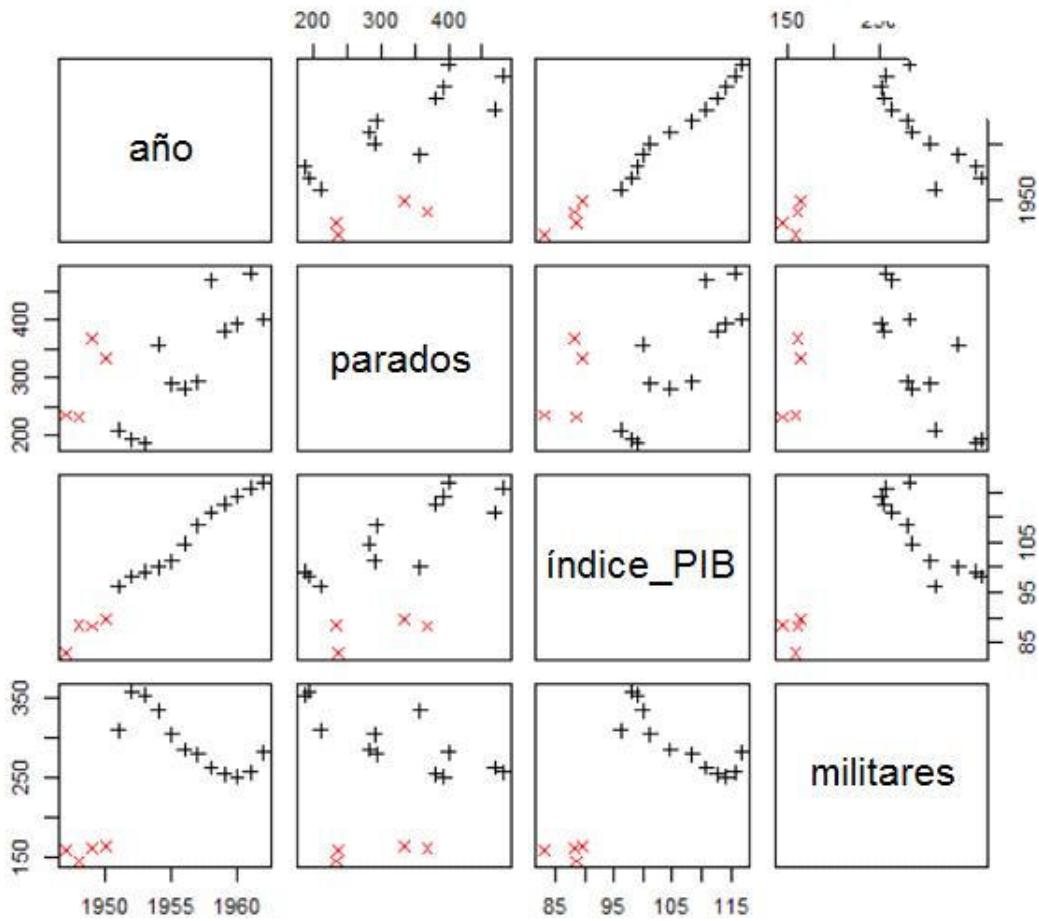


Diagrama por pares, con dos clases y cuatro variables

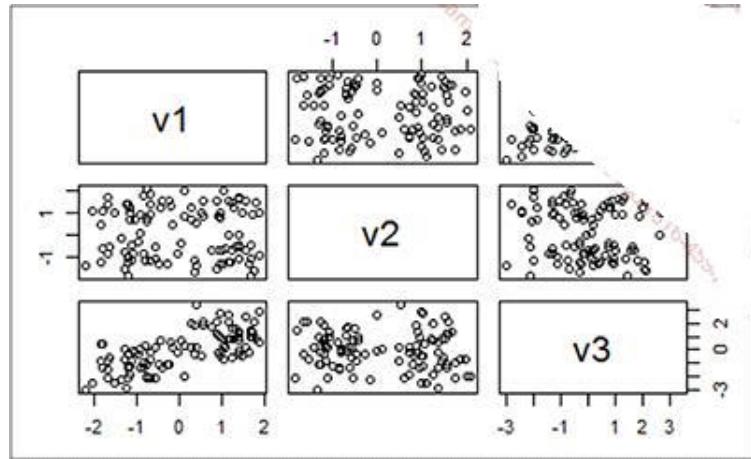
Este tipo de gráfico permite comprender rápidamente las relaciones en un conjunto de variables.

La separación realizada a partir del año 1950 no ha sido al azar. Haciendo variar el año en este gráfico, aparece visualmente una proporción de clustering. Observe, por ejemplo, en la zona inferior izquierda la ilustración del número de militares en función de los años, cómo la separación resulta muy limpia. Como encontramos esta separación en varias ilustraciones, resulta tentador explotar esta hipótesis de clustering y, llegado el caso, crear una feature del tipo: $1 \text{ si } \text{año} > 1950, 0 \text{ si no}$.

b. Clases en configuración XOR

Para convencernos del interés de mostrar símbolos en función de las clases existentes o supuestas, volvamos a los datos que hemos utilizado al comienzo de este capítulo.

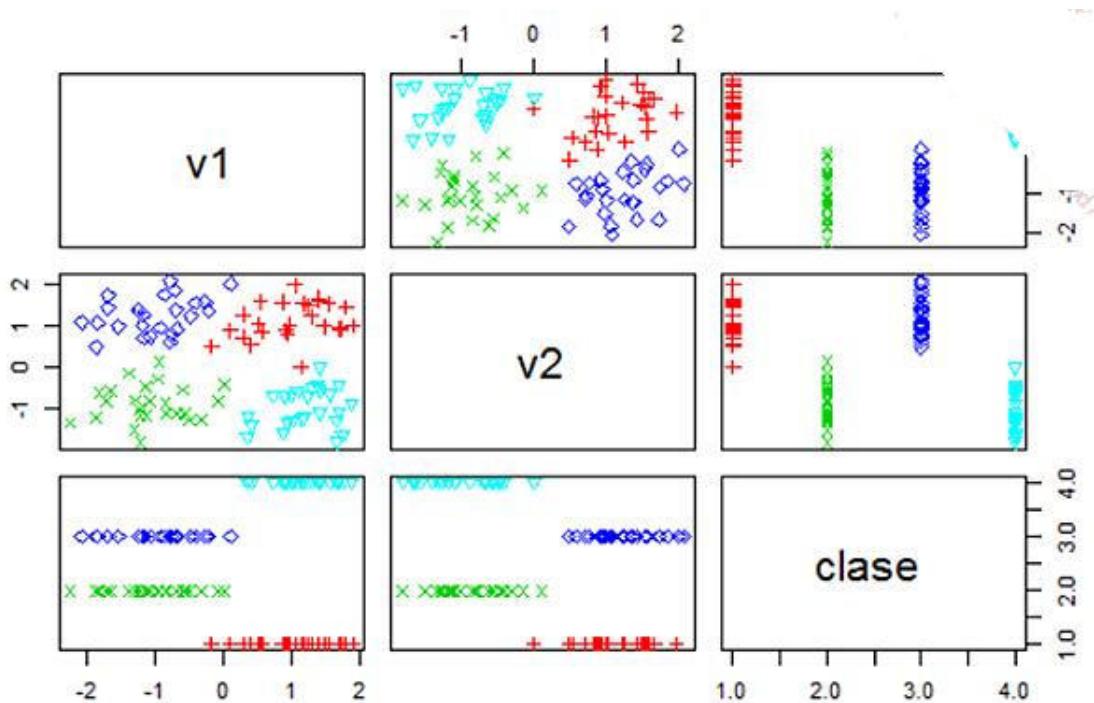
```
plot(df2[,c(1,2,3)])
```



Relaciones v1, v2 y v3... decepcionantes

La relación entre **v1** y **v2** podría parecer completamente estéril en términos de potenciales análisis, pero compare el gráfico anterior con el siguiente.

```
plot(df2[,c(1,2,13)],      # selección de variables
     col = as.numeric(df2$clase)+1, # color función(clase)
     pch = as.numeric(df2$clase)+2 # forma función(clase)
     )
```



Relación v1, v2, clase con símbolos

Las ilustraciones **v1** en función de **v2** son idénticas, con la diferencia de que la pertenencia de los puntos a una clase o a otra salta a la vista, y ahora parece adecuado afirmar que **v1** y **v2** serán excelentes predictores de **clase**.

Observando este gráfico, se confirma que es posible predecir la clase de manera aproximativa utilizando una tabla

de verdad, como la siguiente:

v1<0	v2<0	clase
VERDAD	FALSO	3
VERDAD	VERDAD	2
FALSO	FALSO	1
FALSO	VERDAD	4

Es un medio mnemotécnico para recordar el nombre de esta confirmación, «XOR», es decir, «o exclusivo».

Las variables **v1** y **v2** son i.i.d. (*independent and identically distributed*); solamente conociendo **v1** o **v2** no es posible predecir la clase, pero conociendo ambas se obtiene la capacidad de predecirla: hemos resuelto visualmente un problema de **clasificación**, que es una forma de aprendizaje supervisado.

Cuando abordemos los algoritmos de **clustering**, aprendizaje no supervisado, veremos el caso en el que **no se sabe si existe una clase concreta** y, a través de otras variables, se confirmará (o no) la potencial existencia de estas clases.

c. Diagrama por pares con «factores»

A continuación vamos a crear un diagrama por pares con el paquete **ggplot2** y un paquete complementario a este llamado **GGally**.

Nuestro objetivo es, aquí, visualizar mejor los datos numéricos y los «factores». Para la demostración, hay que agregar una columna ficticia de factores llamada «tipo».

```

set.seed(2)                      # aleatoriedad reproducible
# muestra de 40 enteros tomados de una secuencia de 100
# yendo de 0 a 6 (mediante modulo %% sobe una secuencia)
tipo <- sample(seq(from = 1, by = 3, len = 100)%% 7, 40)
dfy <- data.frame(dfx, tipo)      # agregar una columna
dfy$type=as.factor(dfy$tipo)      # columna de factores

```

Observando el resultado, el código del gráfico es muy compacto.

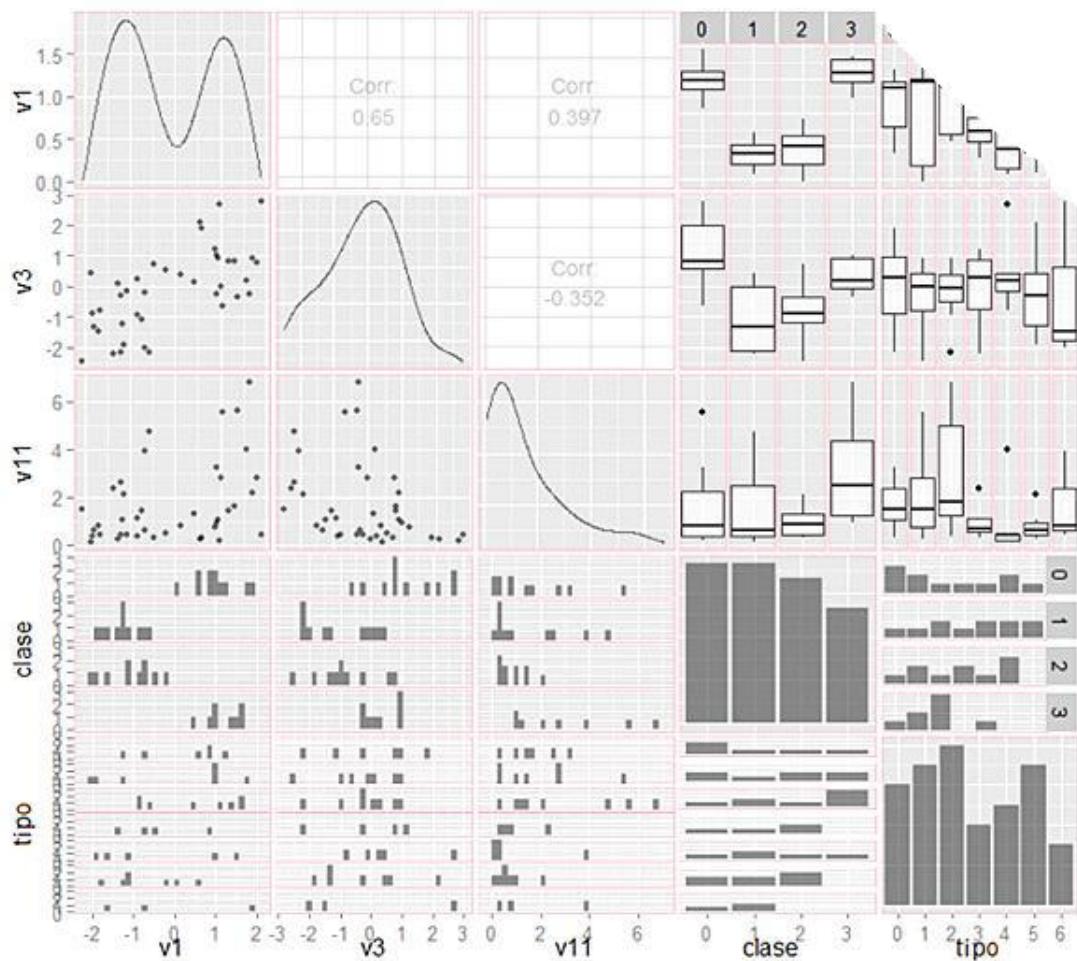


Gráfico por pares, variables continuas y factores

Podemos obtener mucha información de un solo vistazo:

- Las densidades de las variables continuas, representadas en la diagonal del gráfico. Destacamos, por ejemplo, que **v1** es bimodal o que **v11** es muy **asimétrica**, lo que podría llevarnos a aplicar una transformación a esta variable.
- Las distribuciones de los valores de las distintas clases, también en la diagonal.
- Una eventual dependencia lineal (coef. de correlación = 0.65) de las variables **v1** y **v3**.
- El reparto de los elementos por clase para cada variable continua, mediante diagramas de caja. Se plantea, por ejemplo, la cuestión de puntos «outliers», como sugiere el punto que se ha trazado en la clase 0 y que es visible en la línea de la variable **v11**.
- Las distribuciones de cada clase por variable continua. Por ejemplo, se confirma que si **v1** es superior a cero, entonces la clase no es jamás 1 o 2.
- Las clases por tipo (o viceversa). Se confirma que los tipos 5 y 6 no están representados en la clase 3.

El reparto clase por tipo merece un pequeño zoom.

```
p1 <- ggally_ratio(dfy[,c("tipo","clase")])
p1 <- p1 + ggplot2::coord_equal()

p1 <- p1 + theme(panel.background = # ; es más bonito!
```

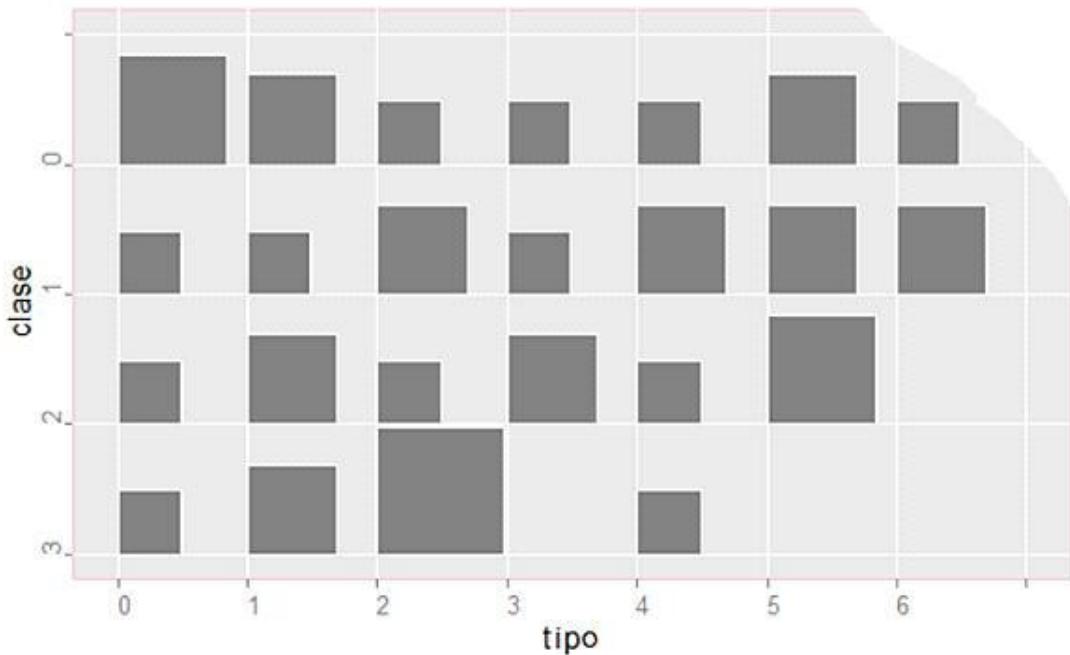
```

element_rect(colour = "pink"))

p1<- p1 + theme(text=element_text(size=16)) # tamaño de la letra
                                         # visualización

```

Los parámetros utilizados permiten visualizar las proporciones de cada intersección de los «factores» **clase** y **tipo** mediante su superficie rectangular. Es un tipo de representación que todavía no habíamos abordado.



Proporciones por clase-tipo

La pareja (clase = 3, tipo = 2) es la más representada; se da aproximadamente 4 veces más que la pareja (clase = 0, tipo = 6). Hay cuatro parejas que no están representadas.

Si tratamos de obtener una previsión a partir de las variables explicativas que serían **clase** y **tipo**, este último punto debería considerarse seriamente: ¿estas uniones resultan imposibles? Esto dejaría entrever una regla que habría que modelar en función de las demás variables. ¿O bien falla la muestra? ¿O simplemente la población no es tan grande y estas uniones no son observables si no se extrae un conjunto de elementos lo suficientemente grande de la población?

Este último caso está lejos de ser imposible, incluso con grandes conjuntos de elementos. Imagine que obtiene 9 clases (**factores**) con 10 posibles niveles por clase, tendría un total de 109 (mil millones) de uniones posibles y necesitaría varios miles de millones de observaciones para estar seguro de disponer de todas las uniones posibles.

d. Facetas y escala logarítmica

Más arriba, observando la relación entre **v7** y **v11**, hemos confirmado que podría resultar conveniente trabajar este gráfico.

En primer lugar, dividamos el gráfico por clase. En segundo lugar, confirmemos la forma exponencial de la curva, y apliquemos una escala logarítmica para comprobar la eventual linealidad de cada trayectoria tras esta transformación.

```

## faceta y escala logarítmica + un título ##

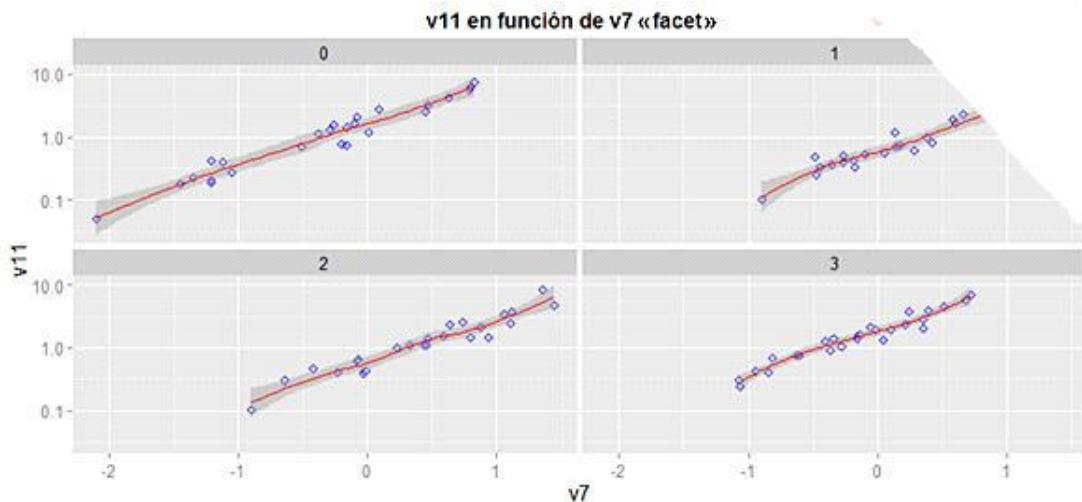
p <- ggplot(data=df2) # escogemos el dataset
al<- aes(x=v7,y=v11) # lista de variables y parámetros
# creación del plot p1
p1<- p+ # al menos un parámetro de geometría:
geom_point(al, # queremos los puntos
            pch = 5, # rombos
            col = "blue" ) + # y azules
geom_smooth(al, # y una regresión
            col = "red", # de color rojo
            size = 0.5 , # grosor de la línea
            linetype = 1) + # línea
            scale_y_log10() # escala logarítmica para y
            labs(title='v11 en función de v7 "facet"' ) # título

p1<- p1 + facet_wrap(~clase)

p1<- p1 + theme(text=element_text(size=16)) # tamaño de la letra
print(p1) # visualización con print (ídem)

```

Observe el uso del parámetro **scale_log10()**.



$\log(v11)$ en función de $v7$ por clase

Para interpretar correctamente las cuatro trayectorias, observe un punto concreto de cada faceta, por ejemplo el valor de la regresión para $v7$ igual a cero. Confirmará entonces que las regresiones para las clases 0 y 3 son similares, como lo son las regresiones para las clases 1 y 2.

Tras dotarnos de las herramientas para visualizar nuestros datos, vamos a completar la paleta de nuestras prácticas algorítmicas.

Machine learning: prácticas corrientes

1. Recorrido teórico acelerado

Ahora estamos familiarizados con una serie de conceptos y notaciones que hemos abordado a través de experiencias y descubrimientos empíricos sucesivos. Vamos a redefinir parte de esta información en un marco teórico simplificado para ofrecerle **una visión rápida y general de la distribución del campo teórico común a los especialistas en machine learning**. Adoptaremos a menudo el punto de vista del aprendizaje supervisado (con una pequeña tendencia a favorecer el aspecto de la regresión del aprendizaje supervisado, pues es más didáctico).

Preste atención: como en las demás partes del libro, la notación matemática se esboza, pero no se describe formalmente como haría un matemático; por ejemplo, las cuantificaciones matemáticas y las definiciones precisas de los espacios a los que pertenecen ciertas variables, ciertas funciones y sus dominios de definición o imágenes no se explicitan formalmente (para evitar, por ejemplo, definir notaciones del tipo espacio de Hilbert, o propias de la teoría de la medida...). Para obtener definiciones «rigurosas», diríjase a los sitios web que citamos en la introducción del libro.

a. Linealidad

Los modelos lineales implementan una **señal** que puede expresarse bajo la forma: $w^T x + b$ (encontramos también esta formulación en el caso de un componente básico de las redes neuronales, el perceptrón, que se basa en una separación lineal del espacio).

Diríjase al capítulo Introducción para consultar los correspondientes esquemas. Encontrará, por ejemplo, una representación de esta última expresión.

También podemos indicar esto poniendo de relieve que el primer término es un producto escalar, lo que permite imaginar otras construcciones similares con otros productos escalares: $(w, x) + b$.

Consideremos $h(x)$ una función hipótesis, tal que esta función se escoja entre numerosas funciones hipótesis y que pueda ser una aproximación de la función $f(x)$ que tratamos de aproximar en un problema de aprendizaje supervisado.

Entonces, en el caso lineal, el problema de clasificación se formula a través de la búsqueda de los **mejores** w y b con $h(x) = \text{sign}(w^T x + b)$. Por `sign()` entendemos el valor -1 o +1 en función del signo de la expresión escalar entre paréntesis (considerando que $w^T x + b = 0$ es la ecuación del hiperplano a partir del cual se espera realizar la separación de las observaciones de ambas clases).

Observe el uso que hemos hecho de la palabra «mejores»: en cierto modo, es el propio objetivo de la teoría del machine learning que describimos en este párrafo el decidir la base sobre la que se va a optimizar nuestro aprendizaje y, por tanto, encontrar el mejor modelo.

Por otro lado, el problema de la regresión lineal se formula mediante la búsqueda de los mejores w y b con $h(x) = w^T x + b$.

Es muy importante comprender que la linealidad que nos interesa más resulta de la combinación lineal de los pesos (w_j) mediante coeficientes constantes producidos por nuestro conjunto de datos (y no al revés, como habíamos visto en el capítulo anterior). Si consideramos esta combinación lineal de los pesos, comprendemos que es posible aplicar cualquier transformación a las columnas (x_j) y permanecer en el marco de un problema lineal donde se buscan los coeficientes (w_j).

A menudo llamamos z al vector obtenido tras la eventual transformación de x por una función de transformación

Φ . Las cosas resultan más fáciles en términos de análisis cuando Φ posee inversa, pues podemos partir de $z = \phi(x)$ para obtener valores de x aplicando la transformación inversa $x = \phi^{-1}(z)$, aunque esto no es indispensable, pues el simple hecho de haber separado las observaciones basta para alcanzar los objetivos básicos de una clasificación.

La aplicación de una transformación Φ no lineal preserva la linealidad de un eventual modelo lineal que queremos ajustar («fit») con nuestros datos, aunque en ocasiones vuelve más complejas las dimensiones del espacio en el que vamos a trabajar (z).

Existen muchas transformaciones no lineales, todas listas y a su disposición. Estas permiten generalizar el modelo lineal a casos en los que habría sido totalmente imposible separar los datos mediante cualquier hiperplano, o incluso con una capa cualquiera (típicamente el caso de paquetes de observaciones convexos que pertenecen a una clase determinada, como los discos o las bolas, contenidas en un continuo de observaciones de otra clase).

Como ejemplo de función Φ , podemos citar la transformación particularmente elegante que vuelve separable a la configuración XOR:

$$\Phi : (x_1, x_2) \rightarrow (x_1, x_2, x_1 \cdot x_2)$$

Los modelos lineales son muy económicos en términos de potencia de cálculo consumida (a diferencia de modelos como los Support Vector Machines, que abordaremos en breve). Estos modelos son, a menudo, excelentes para comenzar con el trabajo de modelización. Lo que aprenda tratando de aplicarlos sobre un problema concreto le será muy útil en su comprensión de sus matices.

b. Errores in y out, noción de dimensión VC

Podemos definir el error propio de una muestra sobre la que se intenta realizar un modelo mediante una función hipótesis como la suma de los errores sobre las distintas observaciones. Este error se denomina, en inglés, Error «in-sample», y es el único error que podemos medir. El error de la población padre se denomina Error «out-sample» y solo está accesible a través de su esperanza matemática:

$$E_{\text{in}}(h) = \frac{1}{n} \sum_{i=1}^n e(h(x_i), f(x_i))$$

$$E_{\text{out}}(h) = \mathbb{E}_x[e(h(x), f(x))]$$

La medida del error que utilizamos es propia del problema funcional sobre el que trabajamos; por ejemplo, ciertas medidas de error pueden ser tolerantes a falsos positivos, pero no a falsos negativos, o a la inversa...

Cuando se escribe $y = f(x)$ la función $f(\cdot)$ está de hecho contaminada por ruido. Este ruido proviene de que nuestros conocimientos respecto al tema son resultado del conjunto de observaciones posibles; dicho de otro modo, reflejan el conocimiento estadístico y empírico de y sabiendo x , lo que se expresa mediante la probabilidad condicional siguiente: $P(y | x)$

La función $f(x)$ traduce una distribución de probabilidad en una función contaminada por ruido. Esto permite sortear una dificultad mayor: si bien se poseen los mismos valores para las diferentes variables explicativas, varias observaciones puede que no desemboquen en el mismo valor de su variable respuesta. Por ejemplo, una persona puede presentar comportamientos diferentes en circunstancias similares sobre las que no se posee una variable explicativa para esta diferencia en el comportamiento.

Para seleccionar el conjunto de entrenamiento (y los conjuntos de prueba o de validación), utilizamos una hipótesis sobre la distribución de estos datos y extraemos una muestra conforme a esta hipótesis (ley uniforme, ley normal...). El algoritmo de aprendizaje se encarga de aprender algo sobre la distribución de la población padre,

aunque su rendimiento puede verse afectado por una distribución padre poco habitual, en particular si su modo de muestreo no respeta estas características básicas.

El muestreo se corresponde con el hecho de extraer parejas (x_i, y_i) generadas a través de la distribución conjunta correspondiente a: $P(x, y) = P(x) \cdot P(y | x)$.

En ocasiones, encontrará la noción de «shattering coefficient», que se traduce en castellano por «coeficiente de pulverización», y que permite deducir el tamaño del conjunto máximo de vectores que es posible crear a partir de un conjunto de puntos x^n y de una función f . Por nuestra parte, habríamos preferido hablar de «coeficiente de separabilidad», siempre que esta noción permita definir otra noción, la «growth function», que caracteriza el crecimiento de la complejidad en función de la muestra para un tipo de modelo determinado.

En el caso del espacio de hipótesis \mathcal{H} , es decir, el conjunto de funciones $h(\cdot)$, esta función es, a menudo, polinomial, y en los casos extremos igual a 2^n . El producto de dicha función de crecimiento y de una exponencial de un término negativo de n tiende a 0 cuando n es muy grande.

En teoría, dicho producto permite deducir la manera en la que un modelo logra rápidamente hacer converger la esperanza matemática de la desviación entre el error in-sample y el error out-sample.

Esto se resume mediante los conceptos de Vapnik y Chervonenkis (VC Bound), y en particular a través del concepto de dimensión VC: d_{VC} .

La diferencia entre el error out-sample y el error in-sample está acotada por un término que nos gustaría ver tender rápidamente a 0 cuando el número de observaciones crece. Esta es una buena noticia, pues quiere decir que, con un número de observaciones lo suficientemente grande, va a ser posible hacer converger ambos errores, aunque por el contrario esto no significa que estos errores vayan a tender a 0.

$$E_{out}(\hat{f}) - E_{in}(\hat{f}) \in \mathcal{O}\left(\sqrt{\frac{d_{VC} \log(n)}{n}}\right)$$

El término \mathcal{O} (notación de Landau) significa que, a partir de un cierto n , existe un $k > 0$ tal que $k \sqrt{\frac{d_{VC} \log(n)}{n}}$

domina todos los $|E_{out}(\hat{f}) - E_{in}(\hat{f})|$. De modo que si este término converge a 0, la desviación entre los errores in y out también va a «comprimirse» a 0 debido a esta dominación; en caso contrario, la desviación entre los errores in y out estará limitada por la función, pero no se verá comprimida hacia 0.

► El término \mathcal{O} también se utiliza en algorítmica para definir la manera en la que un fenómeno converge en función de su tamaño « n » (por ejemplo, un número de filas que hay que tratar). $\mathcal{O}(.)$ expresa el orden de magnitud de su complejidad y ofrece una idea de la rapidez de la convergencia del algoritmo hacia su solución. Por centrar las ideas, digamos que, con nuestras máquinas, un $\mathcal{O}(n)$ con n del orden de un millón se cuenta más bien en milisegundos; un $\mathcal{O}(n^2)$, en horas; un $\mathcal{O}(n^3)$, en años (aunque en horas para $n = 10\,000$), y que, de manera inversa, un $\mathcal{O}(\log(n))$ se cuenta en nanosegundos (sin contar el tiempo fijo de la duración de las entradas-salidas).

De este modo, en nuestro caso, en vista de los términos situados debajo de la raíz, si d_{VC} es una función rápidamente creciente de n , podríamos experimentar dificultades. El aprendizaje no será, evidentemente, consistente.

Observe bien que tomamos la precaución de decir que el aprendizaje no será «evidentemente» consistente: esto se debe a que un gran $\sqrt{\frac{d_{VC} \log(n)}{n}}$ no nos aporta información acerca de la calidad real del aprendizaje que podría, posiblemente, resultar constante debido a otro mecanismo.

Y, de manera inversa, cuando dispone de un algoritmo adaptado a su problema y con un d_{VC} constante o

lentamente creciente en función de n , resulta una muy buena noticia, pues la convergencia a 0 se realizará rápidamente (por ejemplo, según un $\mathcal{O}(\log(n))$).

La eventual elección de una pareja (observaciones, modelo) mediante el análisis de su d_{VC} parece ser una cuestión de equilibrio y de compromiso:

- Cuanto más importante sea d_{VC} , mejor será la calidad del «fitting» sobre los datos de entrenamiento: $E_{in}(\hat{f})$ es cada vez más pequeño, aunque la capacidad de generalización disminuye (riesgo de overfitting).
- Si d_{VC} es pequeño, la generalización es más sencilla: $E_{out}(\hat{f}) \approx E_{in}(\hat{f})$, aunque el «fitting» será peor.

Como punto de atención, conviene saber que la dimensión de un modelo que posee la capacidad de separar el espacio de las p variables mediante hiperplanos o bucles posee una dimensión $p + 1$. Además, la dimensión de un modelo normalizado del tipo:

$$h(x) = \text{sign}(w^T x + 1)$$

es como máximo p .

Respecto a la similaridad entre los hiperplanos y los bucles, piense en una transformación $\phi : (x_1, x_2) \rightarrow (x_1^2, x_2^2)$ que transformara una clase que cubriese un disco centrado envuelto por otra clase en dos clases separables mediante una recta.

Respecto a las redes neuronales, el orden de magnitud de d_{VC} se corresponde con el número de parámetros del modelo (como máximo).

Otro punto de atención resultado de la experiencia parece indicar que utilizar al menos $10 \times d_{VC}$ observaciones produciría buenos resultados. Por nuestra parte, en nuestros contextos Big Data y para d_{VC} menores, trabajamos con conjuntos de entrenamiento del orden de $1000 \times d_{VC}$, sabiendo que los modelos utilizados por los especialistas pero que poseen de manera nativa un gran d_{VC} demuestran su eficacia con otros mecanismos.

Utilizamos, por lo tanto, esta noción para dimensionar nuestras muestras sobre modelos simples (LM, GLM y Logit, neuronal básico...).

El rendimiento de cierto número de algoritmos muy eficaces se modeliza mal mediante d_{VC} . Utilice estas nociones cuando el valor de d_{VC} esté bien documentado y no lo utilice para calificar... lo incalificable.

c. Hiperplanos, separabilidad con márgenes

La mecánica de algoritmos como **el perceptrón** (mecanismo básico e histórico **de las redes neuronales**) permite resolver la inecuación:

$$y_i \cdot (w^T x_i + b) > 0$$

Esto se traduce en que cuando y_i y $(w^T x_i + b)$ son del mismo signo, su producto es positivo y la observación está, potencialmente, bien etiquetada, considerando que $w^T x + b = 0$ representa la ecuación del hiperplano a partir del cual se espera la separación de las observaciones en ambos casos.

El etiquetado de una distribución linealmente separable en etiquetas y («labels») que pueden considerarse como negativas o positivas se realiza mediante una función:

$$f(x) = \text{sign}(w^T x + b)$$

La función característica, definida como la función que indica si una expresión es verdadera o no, permite formalizar ciertos problemas, entre ellos los problemas de optimización. Para representar esta función, es habitual utilizar el símbolo **1** en negrita o \mathbb{I} con la expresión correspondiente. La función devuelve 1 si la expresión es verdadera y 0 en caso contrario.

Esto produce la siguiente expresión:

$$\mathbb{I}_{\{y_i \cdot (w^T x_i + b) > 0\}}$$

Con esta notación, la definición (inocente) del problema de clasificación lineal que trataba de optimizar w y b para obtener el menor error de clasificación se escribe de la siguiente manera:

$$\hat{h}_{w,b} = \operatorname{argmin}_{w,b} (\sum_{i=1}^n \mathbb{I}_{\{y_i \cdot (w^T x_i + b) \leq 0\}})$$

Esta expresión significa simplemente que se busca los valores de w y b , parámetros de nuestra función h , tales que la suma del número de casos de error de clasificación sea mínimo. Observe que esta optimización es demasiado simplista debido a que se realiza brutalmente sobre el espacio de entrenamiento y en vista de lo que hemos aprendido más arriba respecto a la relación entre los errores in-sample y out-sample. Sin embargo, conviene recordar esta formulación, pues es la base de expresiones más complejas pero muy útiles que aparecen en diversos artículos de investigación.

Cuando las nubes de puntos lo permiten, es posible realizar dicho etiquetado introduciendo una noción de margen y considerar hiperplanos que posean un «grosor» (encontramos este enfoque en diversos algoritmos, como los **support vector machines (SVM)**, que veremos más adelante en este capítulo).

Para normalizar esta noción, se introduce la norma del vector w , así como el margen real resulta ser el producto de un margen teórico estrictamente positivo y multiplicado por la norma:

$$\gamma \|w\|$$

Una muestra de observaciones será linealmente separable con un margen si existe un $\gamma > 0$ tal que para toda observación (x_i, y_i) se pueda escribir:

$$y_i \cdot (w^T x_i + b) \geq \gamma \|w\|$$

 Preste atención: encontrará a menudo la expresión $w^T x + b = 0$, que es la ecuación genérica del hiperplano separador. En esta ecuación se invocan los valores teóricos de x que satisfacen la ecuación del hiperplano, cuando x está situado sobre el hiperplano. No hablamos de nuestras observaciones (x,y) , a diferencia de la formulación que acabamos de ver.

Destaquemos que dividir esta ecuación por $\|w\|$ no cambia estrictamente nada, y por lo tanto podríamos obligar a que $\|w\|=1$. Esto significa que encontrará a menudo la siguiente expresión para definir el margen:

$$y_i \cdot (w^T x_i + b) \geq \gamma$$

Podemos modelar el hecho de que la variable de respuesta posea una tasa de ruido uniforme con un parámetro de probabilidad $\eta < 1/2$ multiplicando y por un vector de ruido ϵ tal que $\epsilon_1 = -1$ posea una probabilidad η , y 1 con una probabilidad de $1-\eta$ en caso contrario. Nuestra distribución conjunta se convierte en la distribución de

$$(x, \epsilon^T y).$$

d. Kernel Trick, núcleos, transformaciones, feature space

Hemos visto antes que la complejidad de las clases de funciones que son «separadores lineales» era razonable. Resulta tentador, por tanto, transformar problemas que no sean linealmente separables en problemas linealmente separables. Para ello, la técnica más común se denomina «**kernel trick**», es decir, utilizar una función de transformación basada en los núcleos de Mercer (o siguiendo los pasos de otra teoría que no veremos aquí: los espacios de Hilbert de núcleos reproductores, RKHS- *Reproducing Kernel Hilbert Space*).

K es un núcleo de Mercer *ssi* K es una función de dos vectores sobre el conjunto de números reales que es «simétrica definida positiva». Para ello, es necesario que K sea simétrica y que exista una función Φ tal que K pueda formularse como un producto escalar resultado del producto escalar original y de Φ :

$$K(x_1, x_2) = \langle \Phi(x_1), \Phi(x_2) \rangle$$

En ocasiones, los autores prefieren escribir K haciendo referencia a las coordenadas en el «feature space», lo que produce una expresión más próxima a la expresión matricial común:

$$K_{1,2} = K(x_1, x_2) = \Phi(x_1)^T \cdot \Phi(x_2)$$

Más adelante, vamos a expresar K mediante una formulación compatible con esta definición pero que no exprese a primera vista la función Φ , lo que permite concentrarse en la eficacia de la transformación global por el núcleo y no bloquearse por las características de Φ , sabiendo que K ya posee sus propias restricciones.

Destaquemos, por último, que a menudo la formulación matricial de K será la herramienta práctica que nos permitirá manipular dicho núcleo.

Consideramos una matriz de Gram (en homenaje a Jørgen Pedersen Gram) que contiene todos los valores del producto escalar $K(x_i, x_{i'})$.

Para hacerse una idea del uso habitual de una matriz de Gram, conviene saber que el determinante de dicha matriz es el cuadrado del volumen del paralelogramo (es decir, el paralelepípedo generalizado de n dimensiones) definido por los vectores x_i . Considerando la matriz de Gram que representa K y, por tanto, el resultado de la transformación Φ , se trabaja sobre el hecho de que esta transformación transforma el espacio dejando los volúmenes ocupados por las nubes de observaciones compatibles con la distribución de los volúmenes originales (es decir, mantiene el orden del más pequeño al más grande).

La matriz de Gram del núcleo es naturalmente simétrica, pues el producto escalar es comutativo:

$$K(x_i, x_{i'}) = K(x_{i'}, x_i)$$

Para que sea la matriz de Gram de un núcleo, también debe ser «definida positiva», es decir, que para todo $c = (c_i)$ tenemos:

$$\sum_{i,i'} c_i c_{i'} K(x_i, x_{i'}) \geq 0$$

Lo que también puede escribirse:

$$c^T K c \geq 0$$

si se define K como la matriz $[K_{ij}]$.

Con esta condición, el simple hecho de disponer de K o de su matriz de Gram nos ofrece la **certidumbre de que existe una función de transformación ϕ** correspondiente.

El truco encuentra aquí gran parte de su justificación: puede **aplicar una transformación eficaz a sus datos sin conocer la ecuación**, contentándose con aplicar una función $K(\cdot, \cdot)$, cuya forma es mucho más simple de manipular o de construir que la transformación asociada. Los más valientes pueden incluso tratar de construir sus propios núcleos y probarlos rápidamente. Esto multiplica la capacidad operacional de los algoritmos que tenemos a nuestra disposición.

Es todavía más cierto que podemos fabricar un núcleo a partir de otros núcleos. En efecto:

- Si $\alpha > 0$, es un núcleo.
- La suma y el producto de dos núcleos son también núcleos (definidos positivos).
- Podemos construir núcleos en cascada por composición: $K'(x_1, x_2) = K(\phi(x_1), \phi(x_2))$.

Para poder utilizar diversos algoritmos conocidos sobre los espacios transformados, los «**feature spaces**», a menudo tendrá que calcular la **distancia entre dos vectores z** en el seno de este feature space. Afortunadamente, el cuadrado de esta distancia es muy fácil de expresar:

Escribiendo: $z_1 = \phi(x_1)$ y $z_2 = \phi(x_2)$.

- Aprovechando la norma natural $\|\cdot\|$ del feature space.
- La distancia al cuadrado entre los dos elementos del **z -space** se obtiene mediante una identidad famosa (del nivel de instituto salvo por el formalismo):

$$\|z_1 - z_2\|^2 = K(x_1, x_1) + K(x_2, x_2) - 2 K(x_1, x_2)$$

 Preste atención: el kernel trick puede llevarnos a considerar «feature spaces» demasiado grandes, difíciles de interpretar y sobre los que algunos cálculos pueden resultar demasiado costosos.

Diversos núcleos (p , la dimensión del espacio, siendo al menos igual a 2)

- **Núcleo lineal:** $K(x_1, x_2) = x_1^T x_2$

Las ϕ son aquí transformaciones lineales.

- **Núcleo polinomial:** $K(x_1, x_2) = (b + x_1^T x_2)^d$

Las ϕ son aquí transformaciones polinomiales de orden d .

b puede ser nulo, en cuyo caso hablamos de un **núcleo monomial** de grado d .

- **Traslación y función dos veces derivables y continuas** sobre el espacio \mathbb{R}^p :

Si se dispone de una función $g \in L^2(\mathbb{R}^p)$, entonces $K(x_1, x_2) = g(x_1 - x_2)$ es un núcleo (definido positivo).

- Ejemplo de traslación compuesta con una función g , el **núcleo gaussiano (RBF-Radial Basis Function)**:

$$K(x_1, x_2) = e^{-\alpha \|x_1 - x_2\|^2}$$

- Nos gusta escribir: $\alpha = \frac{1}{2\sigma^2}$

Las Φ son aquí transformaciones de tipo «smooth functions».

- **Núcleo sigmoide**, típico de las redes neuronales:

$$K'(x_1, x_2) = \tanh(K(x_1, x_2) + \theta)$$

- **Núcleo de Laplace**:

$$K(x_1, x_2) = 1/2 e^{-\gamma \|x_1 - x_2\|}$$

- **Núcleo de Tanimoto**, utilizado cuando se dispone de un producto escalar sobre los gráficos:

$$K'(x_1, x_2) = \frac{K(x_1, x_2)}{K(x_1, x_1) + K(x_2, x_2) - K(x_1, x_2)}$$

- Creación de un núcleo a partir de una función $g()$ cualquiera con valor en \mathbb{R} : $K(x_1, x_2) = g(x_1) \cdot g(x_2)$
- Creación de un núcleo a partir de una matriz simétrica S semidefinida positiva de dimensión $n \times n$:

$$K(x_1, x_2) = x_1 S x_2^T$$

 Preste atención: en la práctica del data scientist, existen otras técnicas de núcleos que no se basan en el «kernel trick» que hemos utilizado aquí. Citemos, por ejemplo, los estimadores del método de Parzen-Rosenblatt, que tratan de estimar la densidad de probabilidad de una variable aleatoria (¡muy útil!). Citemos también algunos usos relacionados con times series (que abordaremos más adelante), los «integral kernel» son útiles en la estimación de densidades de probabilidad, y no solamente porque tratamos también la transformación de Fourier, de Laplace... Citemos, por último, los núcleos bayesianos. ¡No los confunda a lo largo de sus lecturas!

Para poder utilizar este «kernel trick», es necesario que el algoritmo que quiera usar pueda escribirse como un proceso que manipule solamente productos escalares. Como los productos escalares permiten crear fácilmente ciertas normas, y las normas permiten crear ciertas distancias, podemos deducir que es posible encontrar la manera de expresar un cierto número de algoritmos mediante productos escalares que harán aparecer los términos de los elementos de una matriz de Gram. Los algoritmos SVM y k-means que veremos más adelante en este capítulo se enmarcan dentro de esta posibilidad.

e. Problemas de la regresión: introducción a la regularización

A modo de introducción, o más bien de «teasing», repitamos lo que declaran numerosos data scientists: «siempre es posible realizar aprendizaje supervisado sin regularización, pero esta óptica es demasiado básica como para poder destacar en este dominio».

Presentación teórica de la regularización

El problema del aprendizaje supervisado tal y como lo abordamos no está, de hecho, bien condicionado, o bien planteado en el sentido de Hadamard. Es decir, no estamos seguros de que exista una solución, de que el número de soluciones sea único y de que el comportamiento de la solución evolucione de manera continua (o casi continua) con cambios en los parámetros iniciales.

Este último problema figura entre aquellos a los que los especialistas se enfrentan a menudo. Por ejemplo, tratando de disminuir la influencia de los puntos extraños (outliers), del ruido ambiental, de la proximidad a las fronteras de decisión de ciertos puntos o de puntos situados en zonas entre dos áreas (puntos en los márgenes).

Sin llegar al fondo de la aserción que vamos a realizar a continuación, debería asimilar que **la propia naturaleza del problema de aprendizaje implica a menudo que este esté mal planteado**. En diversos casos de dimensión infinita del espacio de llegada, el recíproco del operador no es continuo, lo que explica que, cuando el número de dimensiones aumenta, el problema se vuelve inestable debido a las discontinuidades que aparecen (la reciprocidad no se «produce» sobre los puntos originales de la muestra).

Los mecanismos que permiten pasar de un problema mal planteado a un problema bien planteado se conocen, a menudo, bajo el término «regularización», que hace referencia también a la idea de eliminar ciertas irregularidades del problema que se va a tratar.

El método de regularización más conocido y más genérico fuera del mundo de las data sciences se denomina regularización de Tikhonov (o Ridge como «arista»), aunque vamos a utilizar una expresión de regularización más genérica, específica del machine learning antes de introducir el término de regularización «Ridge».

Introducción a la regularización en el uso de la función de riesgo

En el capítulo Dominar los fundamentos habíamos planteado la noción de función de riesgo y definido el *problema del aprendizaje supervisado* en términos de búsqueda de la función que minimice esta función de riesgo.

$$R(X_T, y_T, \hat{f}) = \langle \ell(y_i, \hat{y}_i) \rangle_T$$

$$\hat{f} = \operatorname{argmin}_f (R_T(f))$$

También habíamos expresado una clase de normas cuya expresión genérica era:

$$\|x\|_p = (\sum_i |x_i|^p)^{1/p}$$

Existen otras muchas tipologías de normas y podríamos extender estos conceptos de regularización introduciendo elementos de regularización en función de estas normas (si lo consideramos útil respecto a las características de los algoritmos que vamos a implementar).

La regularización consiste en tratar de sustituir la función de riesgo por una función a la que se agregan uno o varios términos de regularización antes de tratar de minimizarla. Como resultado, se introduce una forma de «margen» que evita ajustarse con demasiada precisión a las observaciones que estén en la frontera de nuestras zonas de decisión y elimina irregularidades o sensibilidades demasiado elevadas respecto a pequeñas variaciones del conjunto de entrenamiento. En efecto, tienen cierta posibilidad de ser «absorbidas» por nuestro «margen».

La optimización de nuestro riesgo se reemplazará por:

$$\hat{f} = \operatorname{argmin}_f (R_T(f) + \lambda_1 L_1(f) + \lambda_2 L_2(f) + \dots)$$

Los $\lambda_r L$ son términos de regularización positivos.

Diversas regularizaciones

En el caso de algoritmos que tratan funciones discriminantes lineales, la regularización se traduce en la búsqueda de los pesos que minimicen la expresión regularizada:

$$\hat{w} = \operatorname{argmin}_w (R_T(w) + \lambda_1 L_1(w) + \lambda_2 L_2(w) + \dots)$$

La expresión más común en los contextos lineales, utilizando dos términos de regularización (llamados respectivamente LASSO y Ridge) es la siguiente:

$$\hat{w} = \operatorname{argmin}_w (\|\tilde{X}w - y\|_2^2 + \lambda_1 \|w\|_1 + \lambda_2 \|w\|_2^2)$$

con $f(X) = \tilde{X}w$; \tilde{X} como la «design matriz» que contiene una columna de 1 agregada a X . En ocasiones encontrará una matriz traspuesta en lugar de esta matriz; simplemente las filas y las columnas de las observaciones se han invertido respecto a nuestra notación en el caso de dicho autor.

En referencia al índice de las normas, en ocasiones se habla de regularización ℓ_1 y de regularización ℓ_2 .

El término de regularización, en particular el término Ridge, puede presentar una formulación más fina que «pondere» cada peso y , de hecho, cada eje del feature space. Para hacer esto se introduce una matriz, a menudo llamada Γ , cuyos coeficientes se aplican al vector de los pesos, si bien la influencia de los distintos pesos no es la misma sobre la regularización. Puede, por ejemplo, jugar con estos coeficientes para disminuir el nivel de regularización sobre los ejes donde sabe que tiene una buena separabilidad debido a alguna característica funcional de su problema. El término de regularización Ridge se convierte en:

$$\|\Gamma w\|_2^2$$

En el caso de los SVM, a menudo se utiliza una regularización ℓ_2 aplicada sobre una función de riesgo muy interesante de estudiar:

$$\hat{w} = \operatorname{argmin}_w \left(\frac{1}{n} \sum_{i=0}^n \max\{0, 1 - \langle w, x_i \rangle\} + \frac{\lambda}{2} \|w\|_2^2 \right)$$

Cuando se aplica un núcleo, el producto escalar y el término de regularización evolucionan ligeramente.

-  Una regularización de distinto tipo pero muy útil: en el contexto del boosting, se trata de encontrar una combinación de hipótesis controlada por un descenso por gradiente exponencial para converger más rápido; la forma de la regularización utilizada es, en este caso, de distinta naturaleza. Se trata de una regularización basada en la entropía que atrae a la distribución de los conjuntos de entrenamiento hacia una distribución uniforme. Su efecto consiste en disminuir los riesgos de ajustarse («fit») al ruido. Como hablamos de entropía, se suman cantidades que presentan logaritmos y que no estudiaremos aquí.

Otro punto de vista

Otra forma de analizar las técnicas de regularización es imaginar que consisten en tener en cuenta el término Ω de la siguiente inecuación, que parte de trabajos sobre la complejidad (ver más arriba el trabajo relativo a d_{VC}):

$$E_{out} \leq E_{in} + \Omega$$

con:

$$E_{out} = \text{sesgo} + \text{varianza}$$

Filosofía de barra libre en torno a la regularización

De hecho, el interés de estos conceptos solo se comprende realmente tras haber sufrido las primeras

frustraciones reales relacionadas con el machine learning: tras semanas de trabajo sobre sus features, sobre el uso de modelos y técnicas inteligente y honestamente seleccionadas, implementa su sistema para probarlo y observa rendimientos netamente inferiores a los que había supuesto dado el cariño depositado en la «feature selection», en sus cálculos, en el control de sus iteraciones, de sus validaciones y de sus pruebas.

El data scientist afirma entonces a sus clientes, a sus colegas y a sus eventuales referentes divinos o científicos: «es injusto... si yo lo he hecho todo bien!»

¿Realmente todo?... ¿Se ha dotado de un esquema de regularización o de alguna técnica alternativa con los mismos objetivos y adaptada a su problema?

La cuestión es perniciosa, pues nada le probará jamás que existe dicho esquema, que sea óptimo o incluso que esté documentado en algún sitio... Y además resulta difícil... y hasta contrario a la intuición; pero sería un crimen intelectual no intentarlo...

De este modo, resulta delicado introducir la regularización demasiado temprano en su «mindset». Pero deberá volver a este punto tras algunos meses de experiencia.

Afirmamos que podemos clasificar o predecir en función de criterios propios de las observaciones pasadas o simplemente accesibles. Suponemos que emergen patrones de todo esto y que es posible identificarlos, o al menos reproducirlos.

Bajo este enfoque nos resistimos a aceptar el caos, el azar, las disruptpciones y la sabiduría ancestral: «lo mejor es enemigo de lo bueno».

Como héroe de los tiempos modernos del Big Data, el data scientist está más motivado por el rendimiento o por la búsqueda de una realidad subyacente y oculta que por la modesta búsqueda de una solución viable entre varias.

Este data scientist se precipita a la hora de trabajar con los últimos algoritmos del momento, en ocasiones sin ni siquiera piensa en el sentido del problema que trata de abordar y en la naturaleza real de los datos sobre los que va a trabajar.

Este data scientist trata de utilizarlo todo de su lado aplicando toda la potencia mecánica a su disposición sobre los datos.

Este data scientist se va a resistir a aplicar mecanismos como la regularización a su problema, pues tiene miedo de dejar escapar parte de la información.

De hecho, una parte de su problema se resume en lo que se niega a aceptar, que: $E_{out} \leq E_{in} + \Omega$ y que es como si: $E_{out} = E_{in} + \sigma(\Omega)$; siendo σ una función que tiende a 0 cuando n crece, ilo cual es FALSO la mayor parte del tiempo!

Se resiste a «aceptar su pérdida» y a utilizar una técnica de regularización como la que hemos visto antes.

Puede que tenga miedo de dejarse llevar por la duda de cara a las numerosas dificultades del problema de aprendizaje:

- $E_{out} \leq E_{in} + \Omega$
- $E_{out} = \text{sesgo} + \text{varianza}$ (es decir, se necesita un compromiso sesgo vs. varianza).
- La distribución $P(y | x)$ no es una función y, por tanto, sabiendo x es posible tener varias y.
- $P(y | x)$ no puede deducirse para todas las x (demasiados casos prácticos posibles que no se dan en

la realidad: la maldición de las grandes dimensiones).

- La forma empírica de $\mathbb{P}(x)$ se ve afectada por el número de dimensiones del problema.
- Pueden existir variables exógenas a los datos.
- Sin ser exógenas, ciertas variables latentes pueden no ser razonablemente accesibles a través de nuestros datos.
- Los procesos del mundo real cambian y, por lo tanto, los patrones que hay que identificar evolucionan, en ocasiones conforme se recoge información, que no siempre es instantánea.
- Los patrones encontrados no resisten a una disrupción importante del contexto; ¿conoce la probabilidad de dicha disrupción?
- El proceso de recogida de información puede ser imperfecto.
- Existen varias fuentes de ruido.
- Lo falso puede implicar lo verdadero.
- Existen lógicas alternativas, en el sentido de que lo contrario del contrario de una verdad no siempre es una verdad (véase Kolgomorov): ciertos algoritmos se sitúan dentro del marco de esta lógica alternativa.
- El problema planteado puede no tener solución, o como mínimo estar mal planteado!
- La potencia de cálculo y el tiempo del que dispone son recursos limitados.
- Tratamos de deducir las causas a partir de las consecuencias, mientras que las consecuencias se deducen de las causas (ien el mejor de los casos!): $\mathbb{P}(x|y)$ versus $\mathbb{P}(y|x)$.
- Por último, una predicción realizada por una máquina se compara a menudo con la que habría podido realizar un humano... y a este respecto, no siempre se supera al experto humano. Si bien el modelo económico de su intervención puede verse invalidado.

2. Práctica por práctica

a. Cross validation: k-fold CV

La ambición de esta técnica es ayudarnos a producir modelos correctamente validados, poco sujetos al overfitting y robustos frente a pequeñas variaciones en las variables explicativas.

Tras haber dividido aleatoriamente nuestras observaciones en varias partes (k folds) de tamaños equivalentes a nivel global, uno de los grupos se considera como conjunto de validación y los $k-1$ grupos restantes se utilizarán como conjuntos de entrenamiento sucesivos. Esta forma de trabajar se efectuará k veces designando cada vez a un grupo como conjunto de validación. El error se evaluará a través de su MSE y se calculará cada vez, para luego realizar una media.

Este método es famoso por proporcionar un excelente compromiso entre el sesgo y la varianza del error de test a condición de escoger un k razonable (como $k = 5$ o $k = 10$).

 Para comprender simplemente las nociones de sesgo y de varianza de los errores de predicción, resulta práctico referirse a la «Dartboard analogy» de Moore&McCabe (2002), que representa una diana del juego de dardos:

Si sus tiros están agrupados en cualquier lugar de la diana, la varianza del error es baja (esté lejos o no del centro de la diana).

Si sus tiros están distribuidos en torno al centro de la diana (estén agrupados o no), el sesgo es bajo.

Cuando la varianza y el sesgo son bajos, todo se sitúa en torno al centro de la diana.

Cuando la varianza y el sesgo son altos... ientonces es un mal tirador!

Cuando realice su tirada, puede dar preferencia a obtener una varianza pequeña, para tratar de agrupar sus impactos en alguna zona, y a continuación tratar de disminuir el sesgo para reconducir el conjunto de impactos hacia el centro de la diana.

El compromiso entre sesgo y varianza refleja de hecho el MSE, es decir, la suma del sesgo al cuadrado y de la varianza. El dilema es que, para un valor de MSE constante, si mejora uno está degradando el otro. Cuando tenga que escoger entre dos modelos cuyos MSE sean similares, resulta conveniente identificar si conviene tenerle más miedo a una varianza importante o a un sesgo importante; esto guiará su elección.

El método k-fold CV, que normalmente se utiliza en problemas de regresión, también puede aplicarse a problemas de clasificación. En este caso, no se utiliza el MSE, sino simplemente el número de observaciones mal cualificadas.

Diversos paquetes, como el paquete **caret**, que vamos a utilizar a menudo en los siguientes ejemplos, permiten invocar fácilmente esta técnica.

```
require(caret) # ;superpaquete!
```

```
# CV k-fold
set.seed(2)
fitControl <- trainControl( ## 10-fold CV
  method = "repeatedcv",
  number = 10,
  ## 10 veces
  repeats = 10)
```

El paquete **caret** encapsula un gran número de algoritmos e invoca su función de modelización a través de una función llamada **train()** en la que puede introducir a la vez los parámetros de origen del algoritmo que se invocará y los parámetros propios de **caret** como **trControl**.

```
m <- train(v12~,data = d,
  method = "glm",
  trControl = fitControl
  ## opciones particulares
  )
```

Aquí, se ha fabricado el modelo **m**, con una fórmula que permite predecir **v12** a partir de los datos de entrenamiento **d**, aplicando el algoritmo "**glm**". El paquete **caret** utilizará la definición de CV-fold construida más arriba. Habríamos podido agregar parámetros propios al algoritmo «**glm**» donde se encuentra el comentario **## opciones particulares**.

b. Naive Bayes

Ya hemos invocado el teorema de Bayes, que puede verse como la expresión de una **probabilidad condicional** de una causa cuando se conoce la realización de un evento.

Fundamentos y principios de los algoritmos de tipo Naive Bayes

Hemos visto su formulación en el caso discreto para dos o tres eventos. Ahora vamos a mostrar una formulación algo más general del teorema.

Consideremos un evento x de probabilidad no nula, que llamamos conocido **a priori**, e y_k , con valores de k de 1 a c , otros eventos con probabilidad no nula, se construye la probabilidad **a posteriori** de asignación de y_k :

$$\mathbb{P}(y_k | x) = \frac{\mathbb{P}(y_k) \cdot \mathbb{P}(x|y_k)}{\sum_{j=1}^c \mathbb{P}(y_j) \cdot \mathbb{P}(x|y_j)}$$

Existe una generalización en el caso de una variable conocida mediante su función de densidad $f(\cdot)$, que llamaremos función de **densidad a priori** de la variable aleatoria que consideramos y que se expresa a través de la siguiente fórmula:

$$f(y | x) = \frac{f(y) \cdot \mathbb{P}(x|y)}{\int_{-\infty}^{\infty} f(y) \cdot \mathbb{P}(x|y) dy}$$

Esta última función de densidad $f(y|x)$ se denomina función de **densidad a posteriori**.

En el marco de un problema de clasificación, se considera que nuestro evento a priori es un vector fila correspondiente a una observación de nuestras variables explicativas (y , por lo tanto, conocido a priori), y que buscamos saber si esta observación pertenece a la clase y_k que denominaremos simplemente k .

Para simplificar la notación, considerando una observación $x = (x_1, \dots, x_p)$, llamaremos $p(k|x)$ a la probabilidad de que x pertenezca a la clase k .

Esto significa la probabilidad de que la clase sea k , conociendo x .

Podemos conocer fácilmente tanto una probabilidad a priori como que una observación cualquiera pertenezca a una clase k , como hemos hecho algunos párrafos más arriba, pues esto se corresponde simplemente con el porcentaje de observaciones del conjunto de entrenamiento que pertenece a la clase k . Llamaremos $p(k)$ a esta probabilidad a priori.

Para simplificar la explicación, nos vamos a limitar al caso en que solo haya dos clases, si bien podría generalizarse fácilmente a un caso multiclase. De modo que a partir de ahora k vale 0 o 1.

Nuestro objetivo es estimar $p(k|x)$, es decir $p(0|x)$ o $p(1|x)$, pues esto se corresponde con el hecho de estimar la probabilidad de estar en el caso 0 o 1 conociendo x .

Una forma habitual de proceder es decidir que, cuando la probabilidad $p(k|x)$ sea superior a 50 %, entonces la observación pertenece a la clase k considerada.

$$p(0|x) = \frac{p(0) p(x|0)}{p(0) p(x|0) + p(1)p(x|1)} = \frac{p(0) f(x|0)}{p(0) f(x|0) + p(1)f(x|1)}$$

donde $f(\cdot)$ representa las distribuciones correspondientes, además:

$$p(1|x) = \frac{p(1) f(x|1)}{p(0) f(x|0) + p(1)f(x|1)}$$

El término del denominador se elimina estableciendo la relación:

$$\frac{p(1|x)}{p(0|x)} = \frac{p(1) f(x|1)}{p(0) f(x|0)}$$

y evidentemente tenemos que $p(0|x) + p(1|x) = 1$,

de modo que, para encontrar nuestras dos probabilidades condicionales sabiendo x , «basta» con encontrar $\frac{f(x|1)}{f(x|0)}$

Si se impone la **condición trivial** de que las variables que forman las x son independientes, entonces la probabilidad de x sabiendo k es igual al producto de la probabilidad de cada una de sus componentes *sabiendo k*, lo que se traduce en:

$$\frac{p(1|x)}{p(0|x)} = \frac{p(1) f(x|1)}{p(0) f(x|0)} = \frac{p(1)}{p(0)} \prod_{j=1}^J \frac{f(x_j|1)}{f(x_j|0)}$$

- La expresión « x sabiendo y » puede sorprender, pero es la expresión utilizada para indicar que el conocimiento de uno está condicionado por el conocimiento del otro.

Podemos estimar las distribuciones marginales del tipo $f(x_j|1)$ simplemente contando en el conjunto de entrenamiento la proporción de un valor x_j determinado entre los demás valores posibles en el seno del conjunto de las observaciones correspondientes a la clase 1 (existen otras maneras de estimar estas distribuciones marginales, aunque esto no aporta nada a lo expuesto hasta el momento).

Trabajando de manera similar para la clase 0, se obtienen todos los elementos necesarios para el cálculo de $\frac{p(1|x)}{p(0|x)}$, y como se sabe que $p(0|x) + p(1|x) = 1$, se obtienen fácilmente $p(0|x)$ y $p(1|x)$.

Evolución del algoritmo

Existe una serie de evoluciones y de mejoras a este algoritmo. Una de ellas conviene conocerla, se denomina **corrección de Laplace (Laplace smoothing = 1)** y se aconseja utilizarla cuando se sospecha que el conjunto de elementos de ciertas distribuciones marginales será muy pequeño.

Cuando el algoritmo está implementado de manera más sofisticada, se trata de escoger la mejor manera de calcular las distribuciones marginales a priori y llegado el caso se utilizan distintas hipótesis sobre las distribuciones de sus distribuciones (uniformes, normales...). Es posible aplicar distintos núcleos de transformación a los datos. Además, en ocasiones se trabaja liberando más o menos condiciones de independencia.

Para seleccionar el mejor «**clasificador**» de cara a los distintos parámetros del algoritmo, se intenta maximizar una cantidad que se corresponde con un producto de probabilidades calculadas **a posteriori**. Esto produce algo como:

$$\hat{y} = \text{argmax}(\dots)$$

Destacaremos que la construcción de este algoritmo no se basa en la minimización de una función de riesgo como habíamos visto en el capítulo Dominar los fundamentos.

- Preste atención: los términos «clasificación bayesiana» y «redes bayesianas» corresponden a clases de modelos de los que forma parte Naive Bayes, pero que pueden conllevar relaciones mucho más ricas entre las variables.

Producto derivado del algoritmo

Aplicando un algoritmo neperiano sobre cada miembro de la generalización de la fórmula:

$$\frac{p(1|x)}{1-p(1|x)} = \frac{p(1)}{p(0)} \prod_{j=1}^{j=p} \frac{f(x_j|1)}{f(x_j|0)}$$

Y tras realizar algunos cálculos elementales, se obtiene una expresión lineal cuya forma es muy común:

$$z_k = w_k^T \cdot x + b$$

con:

$$z_k = \ln(p(k|x))$$

Reconocemos aquí una combinación lineal, donde los pesos se denominan «*weights of evidence*» y representan la intensidad de la contribución de cada variable a cada clase. El conocimiento de estos pesos podría incitarle a tomar diferentes decisiones, como la eliminación de features debido a su poca contribución al modelo.

Uso de estos algoritmos

Incluso en su versión más sencilla, este algoritmo es muy eficaz por varios aspectos:

- Tiene muy buen rendimiento y es muy rápido.
- Es tolerante al hecho de que la condición de independencia de las variables no se cumpla del todo.
- Es capaz de abordar casos en los que las superficies de decisión no sean lineales, es decir, donde las clases no estén separadas por hiperplanos, sino por hipersuperficies mucho más complejas.

Entre las aplicaciones más comentadas de estos algoritmos, encontramos el «scoring» de consumidores, el filtrado de correos no deseados («spam») y el diagnóstico de enfermedades.

Puesta en práctica con R

Vamos a trabajar sobre los datos que habíamos manipulado antes en los gráficos, **tras haber aplicado una estandarización** de aquellos y un escalado para que repartan sus valores entre 0 y 1. Es una opción que debe considerar cuando teme que un algoritmo deje que una variable supere a otra debido al ámbito de sus valores y usted desea dar preferencia a la influencia de los valores más representados. Observe que, en ocasiones, esta puede ser una opción que «esterilice» la eficacia de sus algoritmos y, por lo tanto, conviene saber cuestionar esta práctica.

```
z_norm <- function(x){  
  if((class(x) != "factor") & (class(x) != "character")) {  
    x <- ((x-mean(x))/sd(x)) # sd = desviación típica  
    x <- (x - min(x)) /abs(max(x)-min(x))  
  }  
  x  
}
```

```

        }
df <- data.frame(lapply(df,z_norm))

```

La primera transformación se llama z-score, la segunda es un escalado. Es habitual utilizar solo una o la otra.

Como de costumbre, se separa los datos en dos conjuntos (o mejor dicho en tres: entrenamiento/validación/test).

```

# creación partición entrenamiento/test
require(caret)

set.seed(2)
p           <- createDataPartition(y=df$clase,
                                 p= 60/100,
                                 list=FALSE)
training    <- df[p,]
test        <- df[-p,]

```

Tras seleccionar las variables que nos interesan, se utiliza el paquete e1071, que contiene una implementación de **naive Bayes**.

```

# naive Bayes                               #

set.seed(2)

X      <- c("v3", "v5", "v6", "v7") # explicativa
Y      <- "clase"                      # respuesta

d <- training[c(X, Y)] # data.frame de entrenamiento
t <- test[c(X, Y)]     # data.frame de test

if(require("e1071")==FALSE) install.packages("e1071")
library(e1071)

m <- naiveBayes(clase ~ . ,
                 data = d,
                 laplace = 0)

predictions <- predict(m,t, type= "class")# predecir sobre test
c <- confusionMatrix(predictions,          # matriz de confusión
                      t$clase)
cat("Accuracy del test naiveBayes: ",c[[3]][1], "\n")

print(c)                                # matriz de confusión

```

Accuracy del test naiveBayes: 0.925

Confusion Matrix and Statistics

		Reference			
		0	1	2	3
Prediction	0	10	0	2	0

```

1 0 9 0 0
2 0 1 8 0
3 0 0 0 10

```

La matriz de confusión muestra que las clases 1 y 2 no se han predicho perfectamente, de modo que el valor de *accuracy* de 0.925 parece bastante digno.

En ocasiones resulta útil estudiar las observaciones que han sido mal clasificadas.

```

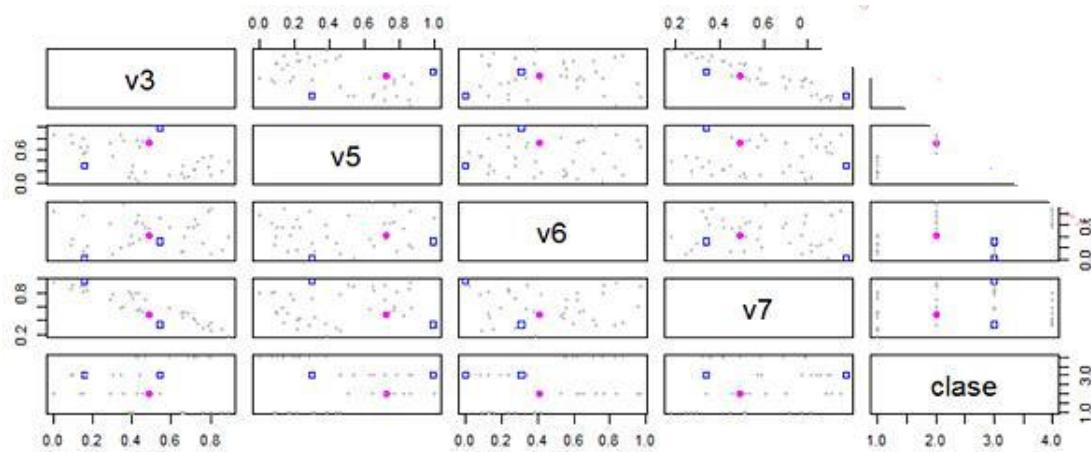
pred <- as.numeric(levels(predictions))[predictions]
real <- as.numeric(levels(t$clase))[t$clase]
error <- pred-real
tt <- data.frame(t,error)
plot(tt[,1:5], col = 8 - error*2, pch = 20-error)

```

Observe y **memorice la sintaxis utilizada** para transformar un **factor** en numérico; es una de las pocas que funcionan correctamente:

as.numeric(levels(v))[v]

siendo **v** un vector de **factors**.



Evidencia de los lugares de los errores de clasificación

La ubicación de los errores de clasificación de la clase 3 (que se corresponde con la clase 2 en términos de nivel en los factores) tal y como se muestra en la figura **v3 X v5** (situada sobre el límite de una de las dos nubes de puntos) podría llamar su atención. Para entrenarse, intente comparar esta representación con el contenido de la matriz de confusión.

c. C4.5 y C5.0

Bajo estos nombres «salvajes» se esconden dos algoritmos propios de los árboles de decisión creados por Ross Quinlan. C5.0 es una evolución comercial de C4.5, que puede utilizar mediante una licencia open source en versión monothread (es decir, sin cálculo paralelo).

Principios básicos

El árbol de decisión se construye mediante el control de la evolución de la entropía que podemos esperar con cada

decisión (test) formando así las ramas del árbol. Se trata de un algoritmo exigente.

Observe que una de las principales diferencias con otro algoritmo de clasificación basado en árboles de decisión, llamado CART, reside en el hecho de que C4.5 utiliza un control que se realiza mediante un índice de entropía, mientras que CART utiliza un índice particular, el índice de Gini. Ambos índices poseen propiedades similares en los extremos: son valores comprendidos entre 0 y 1 (incluidos) y su máximo se sitúa en $\frac{1}{2}$. Con cada decisión, se utiliza el test que más hace progresar la clasificación de los datos de entrenamiento respecto al valor del índice.

El crecimiento del árbol se controla mediante la noción de pureza de los nodos. El árbol se termina cuando las ramas son puras o cuando el número de instancias correspondiente a cada hoja es pequeño (es uno de los parámetros).

A las hojas del árbol se les asigna un valor de clase (la variable de respuesta) que se corresponde con la clase de la mayoría de las hojas que crecen de una misma rama.

Un algoritmo de poda mejora el resultado, así como una mecánica que posea la capacidad de compensar los datos incompletos. Por último, se realiza una traducción en reglas (parecido a lo que sucede en un sistema experto) que densifica el resultado producido.

C5.0 agrega, a los mecanismos ya explorados en C4.5, un algoritmo de «boosting» y gestiona varios tipos de datos, entre ellos fechas. El mecanismo de reglas es más parecido al de los sistemas expertos y, por lo tanto, los conjuntos de reglas pueden mezclarse y aplicarse sobre todo el campo de predicción, lo que aumenta la precisión («accuracy»). La versión comercial está construida para trabajar en modo multithreading sobre varias CPU.

Puesta en práctica con R

Vamos a confirmar que C5 está integrado en el ecosistema de R.

```
set.seed(2)
if(require("C50") == FALSE) install.packages("C50")
library(C50)
m <- train(clase ~ .,
            data = d,
            method = "C5.0",
            importance = TRUE)
predictions <- predict(m, t, type= "raw") # predecir sobre test
c <- confusionMatrix(predictions,           # matriz de confusión
                      t$clase)
cat("Accuracy C5.0 : ", c[[3]][1], "\n")
```

Accuracy C5.0 : 0.95

El valor de accuracy es mejor que con **naive Bayes** (pero no generalice esta afirmación: cada caso tiene su herramienta).

Observe la introducción del parámetro **importance**, que no es específico de C5.0, sino que vamos a utilizar aquí para comparar la influencia de las variables explicativas en la construcción del árbol.

```
importances <- varImp(m, scale = FALSE)
importances
```

```
C5.0 variable importance
Overall
v5      100
v6      100
v3      100
v7      30
```

d. Support Vector Machines (SVM)

Las máquinas de vectores de soporte representan una clase de algoritmos extremadamente interesante. Se trata de algoritmos sólidos, basados en la teoría, que requieren un pequeño número de observaciones para su entrenamiento y capaces de resistir a un gran número de dimensiones. El uso intermedio de una «programación cuadrática» puede limitar su uso en grandes datasets.

Fuera de su formulación matemática, cuya resolución utiliza una función bastante sofisticada (el lagrangiano), el principio general es bastante sencillo.

Caso lineal: objetivos similares al problema del perceptrón

En primer lugar, consideremos el algoritmo en el caso de un espacio de dos clases linealmente separables.

Buscamos un plano separador con un margen, como hemos definido antes en este capítulo. Este plano separador, «espesado» por su margen, está restringido por un número limitado de observaciones que definen los bordes. Algunos autores traducen SVM por «Separador de Vasto Margen». Estas observaciones se denominan vectores de soporte.

Para ajustar este plano, debemos minimizar una cantidad que es función creciente de la norma del vector de pesos:

$$\frac{1}{2} \|w\|^2 + \dots$$

con la restricción de separación:

$$y_i \cdot (w^T x_i + b) \geq 1 + \dots$$

Los «...» se corresponden con cantidades que representan las distintas maneras de ajustar el algoritmo en términos de flexibilidad del margen y del tipo de regularización.

El valor 1 para el margen se ha escogido de manera empírica y establece la norma de las proporciones del conjunto de datos, aunque habría podido guardarse un valor de margen explícito.

La resolución de este sistema se realiza a través de su transformación usando un objeto matemático llamado lagrangiano.

$$\mathcal{L}(w, b, \alpha) = \frac{1}{2} \|w\|^2 - \sum_{i=1}^n \alpha_i y_i (w^T x_i + b) + \sum_{i=1}^n \alpha_i$$

Las α se denominan «multiplicadores lagrangianos» o «valores duales».

El objetivo es minimizar el lagrangiano respecto a w y b y maximizarlo respecto a α . Esto corresponde a lo que

denominamos un punto «silla» de este lagrangiano (se denomina así porque visto en 3 dimensiones este tipo de problema se asemeja a una «silla» de montar a caballo, donde buscaríamos el punto sobre el que se sienta el jinete).

La búsqueda de los óptimos nos lleva a considerar que las derivadas parciales deben ser nulas respecto a w y b (condición necesaria pero no suficiente, la ausencia de variación respecto a las variables indica al menos la presencia de una «meseta»). Se agrega una condición complementaria cuya justificación se sale del marco de lo expuesto y eso produce la expresión:

$$\sum_{i=1}^n \alpha_i y_i = 0$$

y :

$$w = \sum_{i=1}^n \alpha_i y_i x_i$$

En términos de resultado, se comprende fácilmente que **muchos α_i son nulos**, y los demás se corresponden a x_i , que son nuestros «vectores de soporte» y que, por su parte, determinan los hiperplanos y el margen. Podemos eliminar las w y las b de las ecuaciones y obtener un problema, que llamaremos «problema dual», que consiste en buscar las α_i que maximizan una expresión que solo incluye el producto escalar (reconoceremos los elementos de la matriz de Gram):

$$W(\alpha) = \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_{i,j=1}^n \alpha_i \alpha_j y_i y_j \langle x_i, x_j \rangle$$

con las α_i positivas o nulas y $\sum_{i=1}^n \alpha_i y_i = 0$.

A continuación debemos resolver este sistema de «programación cuadrática», llamada así dado que incluye a la vez cuatro ecuaciones e inecuaciones (como en la «programación lineal» evocada en la introducción de este libro, pero con valores correspondientes a polinomios generalizados de grado 2, de ahí el término cuadrático).

Tras realizar los cálculos, obtenemos nuestro modelo:

$$f(x) = \text{sign}(\sum_{i=1}^n \alpha_i y_i \langle x, x_i \rangle + b)$$

 Conociendo la formulación dual del problema, si no queremos utilizar un paquete SVM ya construido, bajo la óptica de introducir características que nos serán útiles de cara a resolver un problema concreto, podemos contentarnos con invocar el paquete R de «programación cuadrática» que prefiramos para terminar nuestro SVM.

Los **vectores de soporte**, cuando todo va bien, existen en un **número limitado**. Aquí reside el interés de este método.

Esta afirmación es bastante evidente si pensamos en un hiperplano del plano, es decir, una recta. El hiperplano «espesado» consiste en dos rectas y basta con tres puntos para definir dos rectas paralelas.

Esta dimensión reducida del problema dual limita la d_{vc} , pues, incluso con un gran número de observaciones, el número de parámetros, que son los vectores de soporte, es muy bajo (parafraseando a uno de nuestros colegas data scientist: *That's the true beauty of SVM!*).

En los casos sencillos de no-separabilidad del problema donde nos molesta tener un número pequeño de observaciones que se mezclan entre ambas clases, típicamente un fenómeno de ósmosis en la frontera de ambas clases, se introduce una configuración que disminuye la restricción del margen. Hablamos entonces de clasificación de «margen débil». Aparece un nuevo parámetro de optimización, de modo que el objetivo es controlar el

compromiso entre la tasa de errores de clasificación y el espesor del margen.

Caso no lineal

Aunque no es en el caso lineal donde los SVM revelan lo mejor de sus cualidades, sino que son todavía más interesantes para abordar casos no lineales.

La expresión dual, incluso con su regularización integrada y el margen débil, utiliza una matriz que contiene los miembros de una matriz de Gram. La aplicación del «kernel trick» a este algoritmo resulta bastante práctica.

Según los datos, la configuración y el núcleo utilizado, se obtendrá un número más o menos importante de vectores de soporte, aunque a menudo muy razonable.

Por otro lado, una parte del ajuste operacional del modelo consiste en encontrar un equilibrio entre sus capacidades de clasificación y el número de vectores de soporte.

Un gran número de vectores de soporte induce una presunción de overfitting.

Extensión del uso de SVM

La maquinaria SVM se adapta al caso de una **clasificación sobre más de dos clases**, pero a través de iteraciones un poco más costosas si el número de clases es importante (si se tienen c clases, el método *one_versus_all* itera c veces, y el método *one_versus_one* itera $c(c - 1)$ veces).

También existen adaptaciones de este algoritmo para los **problemas de regresión**. Para ello nos basaremos en el uso de una función de pérdidas (loss) que determina una forma de pertenencia a una clase «bien o mal» cualificada:

$$\max\{0, |y - f(x)| - \varepsilon\}$$

Imagínese ε como una especie de poder o de límite de separación de los valores de respuestas.

Utilizar SVM con R

El paquete **kernlab** nos aporta numerosas opciones para trabajar con SVM y los núcleos. Mediante **caret** es posible invocar este paquete sin cambiar ninguno de nuestros hábitos.

```
set.seed(2)
if(require("kernlab")==FALSE) install.packages("kernlab")
library(kernlab)
m <- train(clase ~ .,
            data = d,
            method = "svmPoly",
            importance = TRUE)
predictions <- predict(m,t)# predecir sobre test
c <- confusionMatrix(predictions,          # matriz de confusión
                      t$clase)

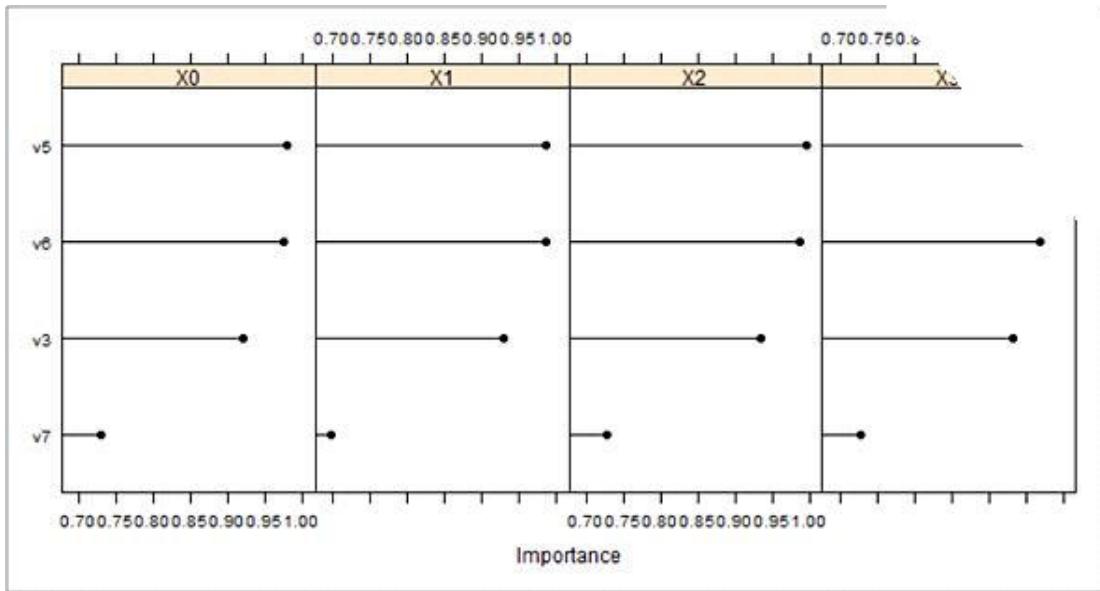
cat(" Accuracy SVM : ",c[[3]][1], "\n")
print(c)                                # matriz de confusión
```

Accuracy SVM : 0.975
Confusion Matrix and Statistics

Reference				
Prediction	0	1	2	3
0	10	0	0	0
1	0	9	0	0
2	0	1	10	0
3	0	0	0	10

Veamos la influencia de las variables de origen.

```
importances <- varImp(m, scale = FALSE)
importances
plot(importances, top = 4)
```



Contribución de las variables

La variable **v7** contribuye poco al modelo, sería interesante estudiar el impacto consistente en no utilizarla en la construcción de este. Si la sustracción de esta feature crea una mala clasificación en las zonas identificables de nuestro espacio, sin duda habrá que conservarla. De manera inversa, si crea errores de clasificación poco numerosos y dispersos, podremos pensar que esta variable puede ser realmente un factor de overfitting. Observe que esta forma de seleccionar las features es un poco trivial: veremos en el capítulo Feature Engineering que en ocasiones resulta conveniente combinar predictores débiles entre sí o, llegado el caso, mezclar predictores débiles y fuertes. Por tanto, nuestro análisis trivial sigue siendo adecuado si no queremos implementar las técnicas de selección de features que veremos en el capítulo Feature Engineering.

e. Clusterización, k-means

A continuación vamos a abordar el tema de la clusterización. En este caso, no disponemos de variable de respuesta, estamos en el caso de un aprendizaje no supervisado. De hecho, queremos hacer emerger agrupaciones sobre el conjunto de datos.

El algoritmo k-means

El algoritmo k-means es un algoritmo «natural», descubierto simultáneamente por varios investigadores a mediados del pasado siglo.

Al algoritmo se le da el número k de clústeres que se desea identificar.

La inicialización del algoritmo requiere que se seleccionen varios puntos de partida, que se propondrán como centros provisionales. Para ello, existen varios métodos; el más sencillo es una determinación aleatoria de estos primeros puntos.

Desde la primera iteración, cada punto se asigna a uno de los centros provisionales en función de su proximidad a este.

A continuación, se recalcula el centro de cada clúster provisional en función de los puntos de cada clúster provisional.

El algoritmo converge cuando, de una iteración a la siguiente, los puntos permanecen en el mismo clúster.

En términos de distancia, los especialistas se basan frecuentemente en la distancia euclídea, aunque podrían utilizarse otras distancias o medidas de similaridad si se adaptan bien a la naturaleza de nuestro problema.

Es un algoritmo exigente en recursos, que puede estabilizarse sobre una configuración de clústeres que puede que solo represente un óptimo local. El algoritmo es muy sensible al método utilizado para escoger los primeros centros.

En ciertos aspectos, el algoritmo se comporta naturalmente como si buscara sus clases como una mezcla de k distribuciones gaussianas en torno a sus k centroides respectivos. Esta afirmación está en el origen de una idea complementaria que consiste en reemplazar la definición de las distancias por nociones más probabilísticas. Es, entre otros, la base de los algoritmos «model based k-means», que son capaces de «clusterizar» datos complejos.

Por nuestra parte, utilizaremos a menudo k-means asociado al *kernel trick*, lo que consiste en buscar los clústeres «lineales» en un feature space implícito de una dimensión mayor para resolver las posibles anidaciones complejas que existan entre las nubes de puntos (esto se abordará en el siguiente ejemplo práctico).

Cuando los centroides seleccionados son puntos reales deducidos de las observaciones, se obtiene un algoritmo llamado «k-medoid» o «Partitioning Around Medoids», cuyo acrónimo es PAM. Este método es más robusto que k-means en lo relativo al ruido y los outliers, y se utiliza a menudo con definiciones de distancias o de similaridades diferentes a la distancia euclídea (la distancia de Manhattan, por ejemplo). Observe atentamente que la noción de robustez no prejuzga la pertinencia de los clústeres, sino la estabilidad de estos frente a pequeñas perturbaciones.

También podemos introducir la noción de difuso (fuzzy) en este tipo de algoritmo. Hablamos en este caso de «fuzzy k-means». Consulte el capítulo Otros problemas, otras soluciones en este libro para comprender los detalles e implicaciones de la lógica difusa.

k-means en la práctica

Vamos a utilizar la función **kkmeans** del paquete **kernlab**, que nos ofrece la capacidad de aplicar un núcleo antes de realizar nuestra clusterización k-means.

Los datos utilizados son las cuatro primeras features de nuestro conjunto de test habitual, donde las dos primeras están en configuración XOR respecto a una clase que habíamos utilizado antes como variable de respuesta.

Aquí, el aprendizaje no es supervisado, de modo que no daremos ninguna información al algoritmo acerca del

hecho de que una observación pueda o no corresponderse con una clase concreta. Nuestra idea, aplicando un kernel polinomial, es que el algoritmo encuentre clases sobre las que no le habíamos dado ninguna pista, lo que demostrará la potencia del algoritmo. Escogemos $k = 4$; en realidad debería probar diversos valores de k antes de encontrar el valor que discrimine los datos de la forma más útil para usted. Pero como aquí tratamos de encontrar cuatro clases, es el número que indicaremos.

```
# clusterización k-means con kernel
set.seed(2)
X     <- c("v1", "v2", "v3", "v4") # explicativa
y     <- "clase"      # respuesta

d <- training[c(X, y)] # data.frame de entrenamiento
t <- test[c(X, y)]      # data.frame de test
nb_v <- ncol(d)         # número de columnas de d
m2 <- kkmeans(as.matrix(d[, -nb_v]),
              centers=4,
              kernel= "polydot", # núcleo polinomial
              degree = 2 )       # grado 2

centers(m2) # coordenadas del centro de los clústeres
size(m2)    # tamaño de los clústeres
kernelf(m2) # tipo de kernel
```

```
[,1]      [,2]      [,3]      [,4]
[1,] 0.8118809 0.2001848 0.5870953 0.3719407
[2,] 0.2947667 0.2875266 0.3280735 0.4519852
[3,] 0.8137515 0.7027082 0.6728157 0.6508371
[4,] 0.4018759 0.7613900 0.3536118 0.7004308

[1] 15 15 12 18
```

```
Polynomial kernel function.
Hyperparameters : degree = 1 scale = 1 offset = 1
```

Hemos obtenido las coordenadas de nuestros centroides, los elementos de nuestros cuatro clústeres y los parámetros utilizados por el algoritmo respecto al kernel usado.

Ahora vamos a preparar los datos para representarlos mediante un gráfico. El siguiente código no posee ningún interés conceptual, pero esté atento a la forma en la que extraemos las etiquetas (`label` = otra manera de llamar a una clase) y nos tendrá que perdonar, si es un programador experimentado, por la no-genericidad del truco que utilizamos para renombrar las etiquetas propias de `kkmeans()`. Observe que, de vez en cuando, cuando se producen cambios, incluso ínfimos, en los parámetros o en la semilla aleatoria, los identificadores de las etiquetas cambian: no hay ninguna razón por la que el algoritmo, no supervisado, «adivine» los identificadores de las clases!

```
labels <- unlist(m2)[1:length(m2)]# extracción de labels
                                # resultantes (es decir, clústeres)

prev <- data.frame(d[,1:nb_v-1],
                    labels)      # no es útil sino agradable

x <- as.matrix(prev[,1:nb_v-1]) # ejes x para plot
```

```

# valores de y
# alineación brutalmente trivial
# de los valores de labels
# con los valores de clase.
p <- prev$labels*100+100
p <- ifelse(p ==500,0,p)
p <- ifelse(p ==300,1,p)
p <- ifelse(p ==400,2,p)
p <- ifelse(p ==200,3,p)
y_new <- factor(p)           # valores y
y_old <- d$clase             # inútil, pero estético

```

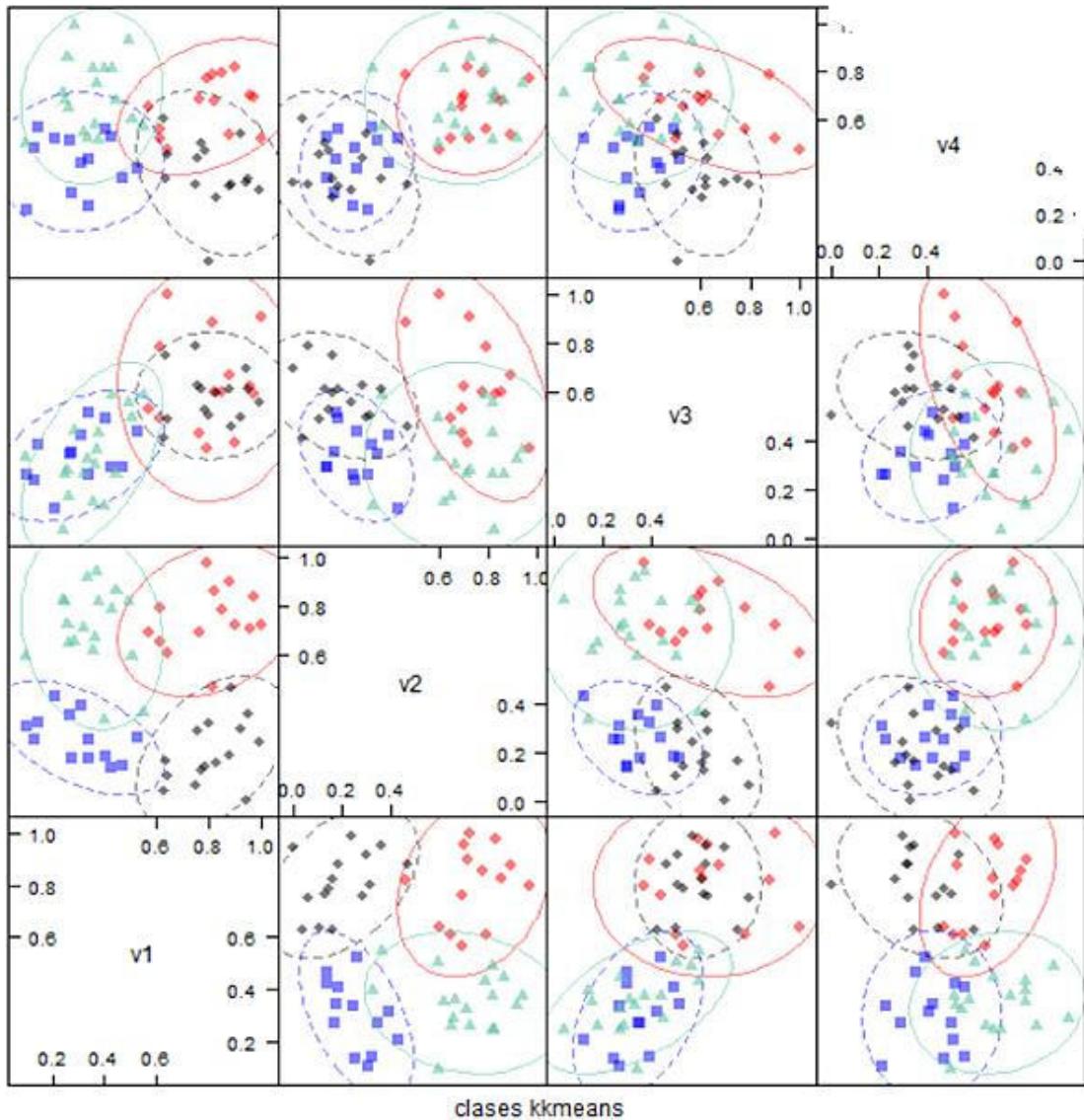
A continuación vamos a utilizar una función gráfica del paquete **caret**, que encapsula las funciones gráficas del paquete **lattice**, que representa una alternativa interesante a **ggplot2**. Esta función dispone de poca capacidad de customización, aunque produce de manera nativa buenos resultados visuales integrando de manera nativa el trazado de una elipse por etiqueta.

```

trellis.par.set(theme = col.whitebg(), warn = FALSE)
transparentTheme(trans = .5)
featurePlot(x, y_new ,
            "ellipse",
            jitter = TRUE,
            xlab= "clases kkmeans")

```

El resultado es bastante estético y fácil de interpretar.



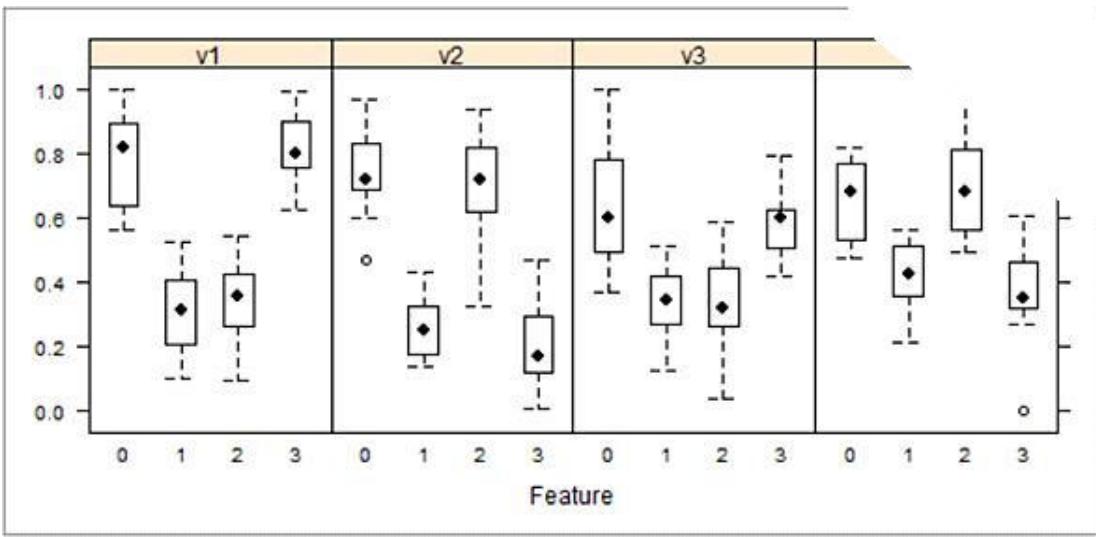
Clústeres determinados por cuatro features vistos en 6x2 lados

Los clústeres parecen estar bien definidos, pues existen lados en los que las elipses se recubren poco (o se recubren dos a dos). Observe atentamente que estas elipses no son las superficies de decisión de **kkmeans**, sino elipses que se agregan después para ayudarnos a visualizar mejor el conjunto.

Además, tenemos el placer de encontrar nuestra configuración XOR sobre **v1** y **v2**.

Veamos ahora los diagramas de caja correspondientes.

```
featurePlot(x, y_new, "box")
```



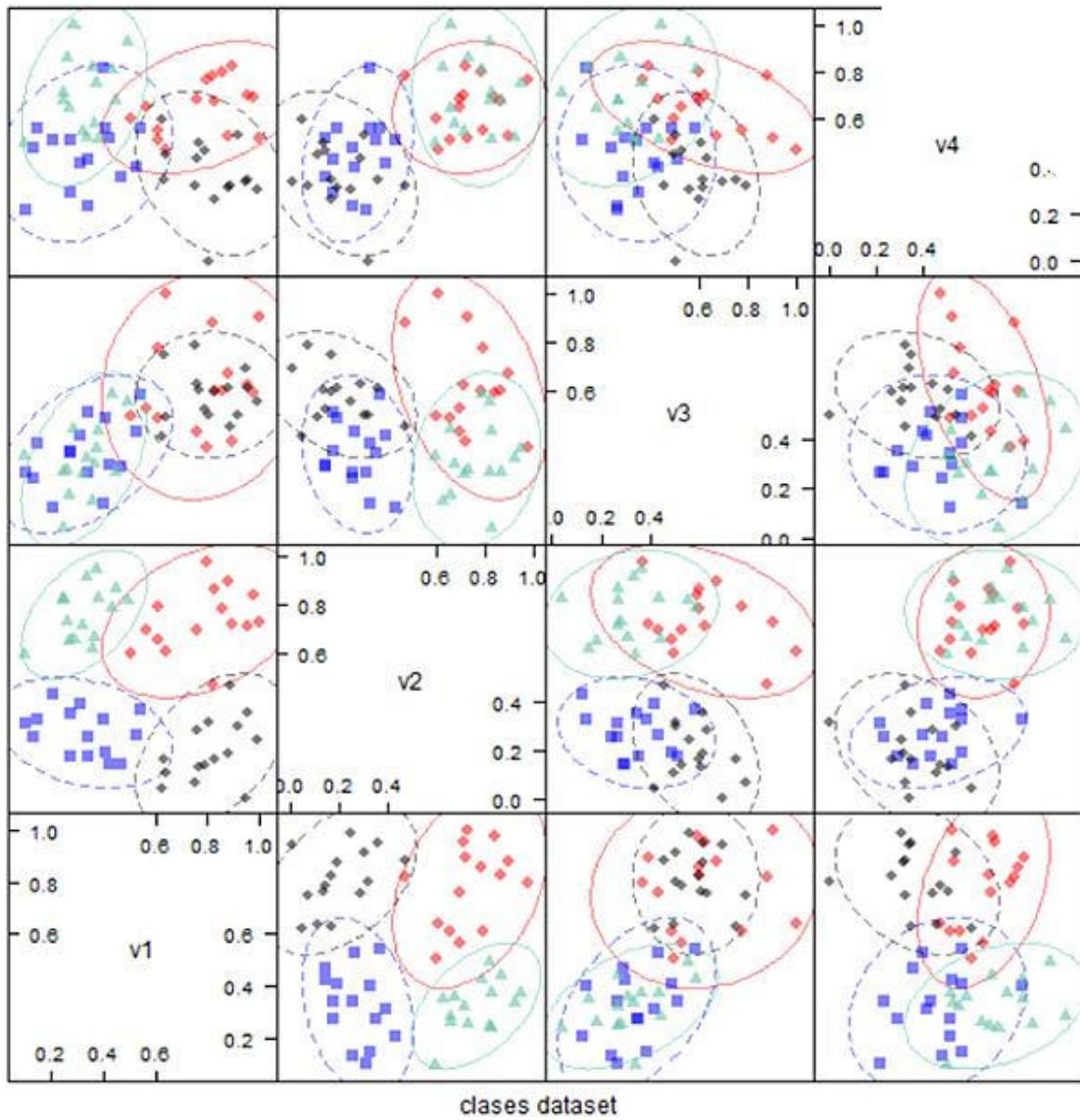
Diagramas de caja de los cuatro clústeres en función de cuatro features

Las distribuciones de los clústeres se reparten efectivamente en función de las cuatro features.

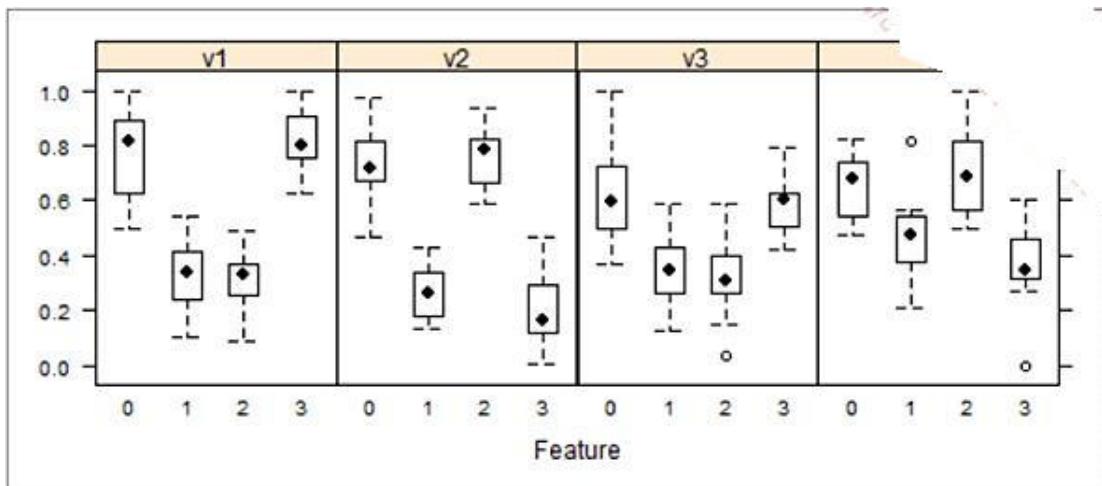
Comparemos con el reparto de las clases para identificar si «por azar» encontramos una superposición entre estas clases conocidas y los clústeres identificados sin supervisión.

```
transparentTheme(trans = .5)
featurePlot(x, y_old ,
            "ellipse",
            jitter = TRUE,
            xlab="clases dataset")

featurePlot(x, y_old, "box")
```



Clases originales



Diagramas de caja originales

Comparando las representaciones de las etiquetas encontradas por la clusterización con la representación de las

clases originales, se confirma una gran superposición de las distribuciones de los puntos. Las distribuciones encontradas por la clusterización son algo más extensas, esto se debe a que se ha producido una correcta generalización.

Observe que no aconsejamos fabricar estimadores de clases utilizando una clusterización (es decir, un método no supervisado), pues nuestro propósito era mostrarle la eficacia del algoritmo de clusterización.

En el caso clásico, no dispone de clases a priori y su objetivo es construirlas.

- Cuando la feature «clase» no es su variable de respuesta buscada, puede utilizar este método para identificar una forma de dependencia entre otras variables y la clase. Se plantea una opción interesante para contraer todas estas features: reemplazarlas (incluso la clase en cuestión) por la etiqueta de los clústeres. Esta forma de trabajar está adaptada al caso práctico de datos fijos, pero no a la construcción de un modelo predictivo, pues sus clústeres son variables en función del conjunto de elementos y de la muestra con la que se trabaja.

La elección de k

En muchos casos prácticos, no dispone de ningún argumento para escoger el valor de k. Afortunadamente, existen diversos métodos a su disposición para no ir a ciegas.

El más común se denomina método de la «silhouette» (silueta en español). Se corresponde con la construcción de un indicador de consistencia de los clústeres. Este método no está implementado en todos los paquetes (cuando lo está, basta con invocar la función correspondiente), de modo que propondremos otro método «manual» que podrá utilizar en todos los casos prácticos desde el momento en que sepa realizar la suma de distancias al cuadrado de cada punto respecto a su clúster.

En el paquete **kkmeans**, como ocurre en otros, cuando la distancia escogida es la distancia euclidiana, la suma de los cuadrados interiores al clúster se denomina «withinss» (es decir, «squared sum»: «dentro»).

Si fuera necesario, siempre puede crear una función similar de forma trivial recorriendo todos los puntos y sumando los cuadrados de las distancias. El método es bastante genérico y conviene recordarlo.

```
# test con distintos números de núcleos
#
k      <- 10          # número máximo de núcleos
sw    <- (1:k)*0  # vector de 0
psize_min <- 0.1        # proporción mínima de elementos
psize     <- 1          # inicialización proporción elementos 100 %
for (i in 2:k)
{
  if (psize >= psize_min){
    set.seed(2)
    m2 <- kkmeans(as.matrix(d[,-nb_v]),
                  centers=i,
                  kernel= "polydot", # núcleo, polinomial
                  degre = 2 )        # grado 2)
    psize <- min(size(m2)/sum(size(m2)))
    swi   <- sum(withinss(m2)) # suma los within-cluster sum of squares
    sw[i] <- swi
  }
}
```

No hay ningún misterio en esta primera sección de código, salvo que se limita el número de elementos mínimo de

un clúster al 10 % del número total de elementos. Se obtiene un vector **sw** de las *sumas de sumas de los cuadrados de las distancias* buscadas.

A continuación hay que crear un gráfico para el análisis. Observe la sintaxis que le permite controlar la escala imponiendo un valor sobre la escala para cada k:

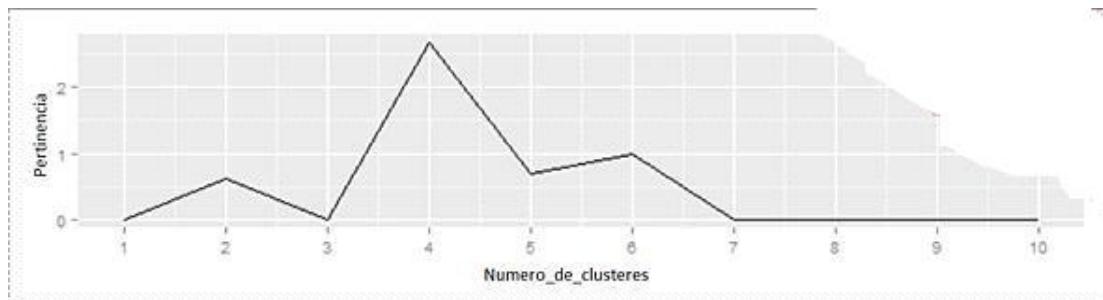
```
scale_x_continuous(breaks=c(1:k))
```

Por otro lado, puede entretenerte descifrando el pequeño truco de programación del principio de este código que trata de organizar y escalar los valores gráficos (intente trabajar sin esto y verá como el resultado es menos atractivo!).

```
# plot resultado
# 1) deselecciona los nulos con
#     una seguridad de 1e-10
# 2) mínimo de no nulos
# 3) distancia de este mínimo
# 4) guarda los no nulos
# observación: si ha escogido sw
# ahora debería haber hecho sqrt
sw <- sw*(sw - min(ifelse(sw <= 1e-10, NA, sw), na.rm = TRUE))

# valor óptimo de k
require("ggplot2")      # paquete gráfico
Pertinencia <- scale(sw, center = FALSE) # no centrado en cero
Numero_de_clusteres <- 1:k                # 1 to k
p <- qplot(Numero_de_clusteres,Pertinencia,geom = "line")
p <- p + scale_x_continuous(breaks=c(1:k)) # escala regular
p
```

A continuación basta con interpretar el resultado.



Selección del número de clústeres en función de las sumas de los cuadrados de las distancias de los puntos a su centroide

El número k sugerido es efectivamente 4.

Un único clúster... no tiene sentido alguno. A partir de siete clústeres incluidos, los conjuntos de elementos son demasiado pequeños. Con tres clústeres no existe ninguna clusterización inteligente (utilizando este núcleo), resultado muy interesante de anticipar debido al hecho de que sabíamos que existía una configuración XOR.

En lo que respecta a más de cuatro clústeres, sin duda resultaría una torpeza seleccionar cinco clústeres, pues esto podría llevar al algoritmo a mezclar puntos de diferentes clústeres de la descomposición óptima en cuatro

clústeres.

Escoger $k = 6$ es la única segunda opción válida.

¿Dónde nos encontramos en nuestro aprendizaje?

Llegados a este punto, nos parece interesante realizar una **evaluación de los conocimientos adquiridos** a través de un eventual estudio lineal de este libro hasta el final de este capítulo.

1. Sus conocimientos operacionales

Ahora dispone de la capacidad de abordar todos los casos de uso «mainstream» del machine learning, en el sentido de que dispone de ejemplos prácticos y operacionales sobre los algoritmos que utilizan actualmente de manera cotidiana muchos profesionales, ya sea la regresión, la clasificación o la clusterización. Conoce bastante bien R para trabajar con sus datos hasta el infinito, crear todas las funciones auxiliares que podrían serle útiles y realizar sus análisis visuales o valorar su trabajo mediante gráficos de calidad.

Su vocabulario y su conocimiento de los conceptos básicos son suficientes como para abordar los modos de uso de muchos paquetes y descifrar el sentido de muchos textos escritos por investigadores. Puede integrarse en un equipo y dialogar. Evidentemente, todavía tiene que abordar muchas otras técnicas, pero en la mayoría de los casos podrá interpretarlos a través del filtro de sus conocimientos adquiridos hasta el momento.

Para perfeccionar este esquema básico que le permitirá convertirse en un data scientist «mainstream» y operacional capaz de integrarse en una empresa, tan solo le queda abordar el contenido del capítulo Feature Engineering dedicado a la selección de features. Si no dispone de tiempo y tiene que comenzar un proyecto de data scientist en pocos días, empiece a estudiar el capítulo Feature Engineering. En caso contrario, le proponemos un segundo recorrido que le llevará hasta el capítulo Feature Engineering, que cerrará el bucle.

2. Las posibles lagunas que es preciso cubrir ahora

El siguiente capítulo le permitirá hacer una pausa. Se trata de un capítulo metodológico, que le dotará de un resumen rápido acerca de la manera clásica de abordar un problema de data science.

Además, de capítulo en capítulo, abordaremos nuevos tipos de datos y nuevos algoritmos o nuevos conceptos que le permitirán ejercer su creatividad de data scientist a todos los niveles.

No desciende las partes de este libro dedicadas al lenguaje natural ni la parte centrada en las series temporales, ipues sin duda se cruzará con ellas algún día!

Marco metodológico del data scientist

El problema metodológico a nivel del proyecto

Los data scientists operacionales proceden de la voluntad de satisfacer las aspiraciones, las necesidades o las exigencias de un cliente interno a una organización.

Abordaremos aquí las características del desarrollo de un proyecto que incorpore data sciences entre su conjunto de actividades.

Abordaremos a continuación la parte del desarrollo del proyecto propia de los aspectos de data sciences.

El objetivo es proveer una especie de «checklist» comentada para los managers, jefes de proyecto y data scientists.

1. La expresión de una necesidad

La formulación de la necesidad requiere a menudo un arduo trabajo de soporte. Esta necesidad se expresa, con frecuencia, en formas que no son directamente interpretables en términos de problema de data sciences:

- Me gustaría ir más lejos en el análisis de nuestros datos de clientes/pacientes/administrados/contrapartes/competidores/mercados/riesgos/fraudes/seguridad/proveedores/logísticos/producciones/países/tendencias/sociedades/medioambientales...
- Querría obtener el valor de estos datos (aprovechar la fiebre del oro de los datos).
- No quiero «perder el tren» del Big Data.
- Querría proveer nuevos servicios y productos a mis clientes.
- Quiero revolucionar mi oferta (disrupción).
- Quiero digitalizar mi organización.
- Querría optimizar o transformar mis procesos.
- Querría anticiparme frente a los movimientos del mercado, de los competidores, del *churn* (fidelidad de los clientes)...

El enfoque de la definición de la necesidad es, de hecho, un enfoque de comprensión mutua. La idea es establecer un ciclo que permita al «cliente interno» y al data scientist (aquí un *business analyst* o un *director de proyectos* con competencias en data sciences) compartir al menos los siguientes cinco aspectos fundamentales:

- Criterios de éxito comunes que permitan juzgar el éxito del proyecto de data sciences (KPI: *key business/process/product/project indicators*).
- Una comprensión común de lo que puede esperarse tras el tratamiento de estos datos (control, comprensión, nuevos conocimientos, nuevas peticiones, optimización, decisión, alerta, anticipación, predicción, supervisión, realidad aumentada, automatización...).
- Casos de uso de lo que producirá el proyecto.
- Una comprensión común de los datos necesarios para la implementación del proyecto (naturaleza, volumen, exhaustividad, proveniencia, temporalidad, fiabilidad, propiedad, coste, confidencialidad...).
- Los elementos lógicos necesarios para el proyecto (presupuesto, arquitectura, planificación).

Como mínimo, el resultado de este análisis debería formar parte de un acta y de una presentación resumida para los decisores y los principales actores.

2. La gestión del proyecto

Una vez lanzado el prototipo o el proyecto, el jefe de proyecto toma la iniciativa necesaria para hacer converger el conjunto respetando de la planificación.

Como la propia naturaleza de un proyecto de data sciences implica numerosas entregas y muchas evoluciones, los elementos de la planificación podrán en ocasiones modificarse y volverse a trabajar. El jefe de proyecto debe encontrar las formas de gestión del proyecto que permitan que los distintos actores puedan implicarse en las distintas evoluciones (y como mínimo que estén informados!). El control del proyecto debería integrar procesos de decisión compatibles con la variabilidad intrínseca de este tipo de proyecto.

Una excelente forma de proceder es trabajar bajo un marco **iterativo de compromiso progresivo**. Con cada iteración, se precisa el conjunto de iteraciones siguientes y evidentemente el contenido preciso de la siguiente iteración.

Se trata de un ciclo ágil, pero **no es conveniente** implementar todas las «mejores técnicas» de la agilidad propias de las prácticas del desarrollo informático: la psicología del data scientist no debería ser igual que la psicología de un desarrollador eficaz o del creador de una startup.

Como jefe de proyecto de un equipo de data scientists, evite recurrir sistemáticamente a los stand-up meetings, al Scrum, al peer-programming y, en general, a las técnicas de socialización sistemática del equipo. Son excelentes métodos, pero el ciclo de reflexión I+D no se presta a esto, al menos durante las primeras iteraciones. Deje a los data scientists llegar lo más al fondo posible, muestre interés en sus reflexiones y sus prácticas, deje al equipo explicarse y dudar...

El jefe de proyecto no debería centrarse en los entregables de los data scientists salvo en **dos hitos** precisos en cada iteración: al comienzo de la iteración para darle un pequeño impulso (*kick-off*) y antes del último tercio de esta para recoger los compromisos definitivos respecto a lo que se entregará realmente (*commitments*) y asegurar que los data scientists acompañarán sus entregables con una documentación de calidad y con los componentes bien referenciados (véase más adelante en este capítulo).

Evidentemente, conviene favorecer la transferencia de competencias de los séniors hacia los júniors, los diseños colaborativos e inducir sesiones de intercambio sobre una pizarra en blanco, rotulador en mano. Todo esto debe realizarse desde el respeto al ritmo de reflexión de cada uno y sin intención de controlar. No se controla la I+D y la creatividad, ise las favorece!

Para compensar la naturaleza aleatoria de este proceso abierto, científico y creativo, debería gestionarse un **plan de proyecto sencillo** con rigor y pragmatismo. Sin un exceso de formalismo y particularmente compacto, permite a **cada miembro del equipo** tener en mente los elementos importantes de la iteración en curso y recordar las reglas de **gestión de la configuración**. Todo miembro del equipo **debería haber leído y comprendido la planificación**, y en particular todo lo relativo a sus responsabilidades en la siguiente iteración: ide nada sirve ignorar la ley!

El plan de proyecto pone de relieve los grandes hitos del proyecto y precisa eventuales hitos importantes de la iteración. En este plan de proyecto, cada información que aporte incertidumbre estará marcada con una imprecisión bien clara, por ejemplo: «dicha acción se realizará en 10 a 20 días/persona y producirá como mínimo esto y en el mejor caso esto otro». Los objetivos de la siguiente iteración incluyen la lista de entregables principales en un estadio de definición identificado en una matriz RACI mínima (*Responsible = Doer, Accountable, Consulted, Informed*). Cada entregable principal posee una priorización propia y sus incertidumbres propias (es decir, opcional o no en el próximo ciclo, requisitos claves del entregable, estimación de carga del entregable, elementos de recepción mínimos del entregable). El plan de iteración se **discute y modifica por el equipo** al inicio de cada iteración. Los objetivos, los resultados esperados (incluso los psicológicos) y los **compromisos de cara al cliente** interno se recuerdan durante el *kick-off*.

- 💡 Los entregables pueden tener una naturaleza operacional (entregar tal elemento técnico o algorítmico), informacional (por ejemplo, identificar y cualificar las fuentes de datos necesarias para alcanzar tal objetivo) o decisional (típicamente la definición de los elementos de gestión del proyecto, la respuesta a las preguntas clave y el reporte de las dudas técnicas).

Con cada iteración, conviene precisar las nuevas prioridades e integrar los conocimientos adquiridos previamente para reorganizar el proyecto. En este marco, cada hipótesis que haya conducido a una decisión debería registrarse cuidadosamente y ser revisada con regularidad.

Esta gestión de hipótesis es esencial, pues los proyectos de data sciences se llevan a cabo en entornos cambiantes, con una gran competencia, innovadores y experimentales.

En sus revisiones y tras su toma de decisiones, el cliente interno, el director de informática y el jefe de proyecto deberían constantemente **tener presentes los tres factores siguientes**:

- ¡Nada de data science sin datos!
- A priori no se conoce la calidad y la pertinencia de lo que se va a encontrar en los datos: habrá que reconfigurar el proyecto con frecuencia, incluidos los objetivos y los *use cases* (casos de uso)!
- La tecnología es compartida, está disponible y es open source. El factor diferenciador de cara a la competencia reside, por lo tanto, en la calidad de los equipos: un dominio real de data sciences, un dominio metodológico, capacidad de innovación, un dominio del proyecto, la gestión del cambio, competencias funcionales, un dominio de las arquitecturas actuales... Esta última lista debería tenerse en cuenta más o menos en este orden, pues el dominio de data sciences es algo infinitamente más disruptivo que el dominio de Hadoop o de Spark; al fin y al cabo, estas tecnologías solo encuentran su utilidad a través de lo que hacen.

El ciclo interno de data sciences

Dentro del ciclo de proyecto, el data scientist va a gestionar las iteraciones específicas de un ciclo que le pertenece, pero que tendrá que sincronizar cuidadosamente con los ciclos de producción y de decisión del proyecto.

A partir de un cierto estado «funcional» concreto del proyecto, a saber una definición de sus objetivos, diversas exigencias de negocio, cuestiones planteadas, casos de uso, datos disponibles y equipos materiales, el data scientist va a comenzar una nueva iteración.

Tendrá que planificar cuidadosamente esta iteración y tratar de no dejarse llevar por un perfeccionismo o un laxismo matemático, algorítmico o técnico incompatible con los objetivos de su planificación. Planificar una actividad I+D es algo particularmente delicado y requiere mucha atención.

En primer lugar, el data scientist que haya cualificado una necesidad y un contexto va a realizar un estudio más o menos analítico de estos.

1. Revisión detallada del problema planteado

La primera tarea consiste en expresar el problema en términos de técnicas propias de data sciences. Preste atención a no centrarse exclusivamente en las técnicas estadísticas o de machine learning, sino en recorrer la gama de algoritmos a su disposición basándose en los objetivos funcionales enunciados.

En lo relativo a los aspectos *machine learning*, hay que identificar rápidamente si el problema planteado se traduce en términos de aprendizaje supervisado o no. En los casos más complejos, habrá que interesarse en los ciclos de técnicas iterativas del *deep learning*.

La semántica general de los datos ideales a manipular debe abordarse con precaución. ¿Resulta útil manipular conceptos, palabras, expresiones concretas (nombres de personas, de organizaciones), eventos, información agregada (cifras de negocio), relaciones (redes), series temporales (cotizaciones en bolsa), imágenes, vídeos, sonidos, datos geográficos...?

Conviene identificar rápidamente si la capacidad con la que tendremos que responder al problema planteado requiere el dominio de un «modelo de datos»:

- Están claros los criterios de pertenencia de estos datos: típicamente, ¿poseen un identificador, referencias mutuas?
- Existen relaciones semánticas o funcionales entre estos datos (un coche es un tipo de medio de transporte, compuesto de una carrocería, un motor... poseído por un individuo, ensamblado en una fábrica).
- ¿Es posible imaginar una vista multidimensional de estos datos desde diversos ejes?
- ¿Existen claves de relación (individuos, lugares, fechas, temas, frecuencias, colores, volúmenes, densidades, idiomas, orientaciones, velocidades)?
- ¿Existe una referencia temporal (fecha de creación, de identificación, duración, caducidad...)?
- ¿Cuál es la parte de incertidumbre, de elementos cualitativos, de flujo, de incompletitud de la información?
- ¿La existencia (o no) de ciertos datos es, en sí misma, información?

Una vez planteados los principales ejes de trabajo, ahora es posible arrancar el proceso sobre bases sólidas.

2. Trabajos previos sobre los datos

De la etapa anterior arranca un primer cuaderno de carga relativo a los datos útiles para la iteración.

a. Exigencias sobre los datos

Se identifica una gama de datos que es preciso recopilar, en términos de proveniencia, de naturaleza, de atributos, de volumen, de temporalidad (de tal fecha a tal fecha).

Llegados a este punto, hay que señalar cuidadosamente qué se espera de tal o cual dato, priorizar los requisitos en función de la esperanza de éxito vinculada al acceso o a la transformación de tal o cual conjunto de datos.

Hay que asegurar que los requisitos sobre los datos son compatibles no solo con la percepción que tenemos del problema, sino también con los criterios económicos y logísticos de la iteración.

Por ejemplo, ipuede resultar inútil recopilar datos en un idioma que no sabe procesar en esta iteración, o recopilar datos sobre volúmenes tales que no podrá manipular en esta iteración, o almacenar datos de dispositivos que simplemente no tendrá tiempo de analizar!

b. Recogida, limpieza y comprensión de los datos

En lugar de comprender el conjunto de datos inicial de una sola vez, a menudo es oportuno concentrarse en un primer flujo de datos para estudiar la estructura y el significado semántico aparente.

Los datos pueden presentarse bajo diversos formatos (estructurados, semi-estructurados, no-estructurados) y su coste de transformación puede no ser despreciable, en términos de retardo o de tiempo pasado. Una transformación inadecuada puede comprometer futuros proyectos «esterilizando» el potencial de los datos, y a menudo se da el caso cuando se trata de recolocar los datos según un modelo relacional, en estrella o en copo de nieve (modelos característicos de los proyectos de datawarehouse). Transformando los datos para poder manipularlos, en ocasiones se pierde mucha información útil, de modo que no es una decisión que pueda tomarse a la ligera.

A partir de este punto, es útil formalizar rápidamente nuestras hipótesis y nuestras cuestiones acerca del sentido de estos datos y su semántica (por ejemplo: pienso obtener una cifra de negocios anual, pero ¿es sobre un periodo contable o de enero a diciembre?).

El data scientist va a elaborar, ahora, una comprensión estadística y visual de los datos recopilados. Va a analizarlos en términos de distribución, de densidad, de linealidad, de calidad de las muestras, y va a dotarse de numerosas representaciones gráficas. Llegados a este punto, sin pretensión alguna de obtener un resultado, es habitual utilizar ciertos algoritmos sencillos y, sobre todo, cuyos resultados son fácilmente interpretables (LM, k-NN, k-Means, Tree...) para «sentir» los datos. Pero en ningún caso hay que buscar adaptarse a un modelo demasiado rápido.

Es bastante útil charlar con los interlocutores de negocio de la organización, observar sus reacciones frente a las representaciones gráficas, con el objetivo de identificar rápidamente diversas anomalías evidentes (por ejemplo: «es curioso, no hay datos acerca de tal producto en tal año, mientras que...»).

En este punto se realiza una primera limpieza de los datos. No existe ninguna vocación semántica y se articula basándose en la identificación de los datos inválidos, de datos incompletos que se podría recopilar, de formatos básicos para acceder a los datos y de pertenencia a datasets siempre que estas pertenencias sean evidentes semánticamente (por ejemplo, recopilar los datos por individuo, si no existe ninguna ambigüedad acerca de la clave de asociación).

Cuando llegue el momento, habrá que «cruzar el Rubicón» y comenzar con la preparación fina de los datos en el sentido *feature engineering* para esta iteración. Es una decisión que debería tomarse en el momento adecuado dentro de cada iteración: demasiado pronto, empezaremos a trabajar sobre datos que no comprendemos o con demasiadas lagunas; demasiado tarde, no tendremos el tiempo necesario para realizar un buen trabajo de data

science de aquí al final de la iteración.

A menudo, el data scientist debutante pasa a la siguiente fase... demasiado pronto! Implementa los algoritmos más potentes a su disposición sobre datos «no aptos para su consumo» o incomprendidos semánticamente, y en ocasiones demasiado sesgados.

3. El ciclo de modelado

A menudo se percibe como el ciclo más «noble» por el data scientist matemático o técnico. Comprende cinco subciclos que se realizan de manera iterativa, sabiendo que cuando todo va bien no hay necesidad de iterar el test ni la interpretación:

- Feature engineering (que abordaremos también en el capítulo Feature Engineering).
- Modelado.
- Evaluación (modelo + features).
- Test (modelo + features).
- Interpretación y confrontación con el cliente interno.

a. Feature engineering

A partir de los datos brutos, vamos a realizar diversas transformaciones, selecciones, asociaciones, creación de nuevos datos a partir de los datos disponibles. La idea consiste en mejorar estos datos para poder utilizarlos en nuestros algoritmos y alcanzar nuestros objetivos.

En términos de tipos de transformaciones, podemos citar:

- Textos que se convierten en «opiniones positivas o negativas», conceptos, acciones, valores, fechas, duraciones...
- Imágenes que se convierten en contornos, distribuciones de colores...
- Sonidos que se convierten en distribuciones de frecuencias, de volumen, de referencias a objetos del mundo real, de palabras...
- Películas que se convierten en series temporales, conceptos, grafos...
- Curvas que se convierten en valores: curtosis, skewness, cuartiles, desviación típica, media...
- Etiquetas de archivos .xml que se convierten en nombres de columnas.
- Uniones, proyecciones, agrupaciones, filtros realizados sobre los datos.
- Atributos numéricos que se convierten en factores.
- Individuos que se convierten en grupos, centroides...
- Clústeres que se convierten en clases.
- Composiciones de columnas que se convierten en otras columnas.
- Transformaciones de columnas mediante diversas funciones, entre ellas la normalización de los valores de las columnas.
- Divisiones de columnas en varias columnas.
- Puntos extraños que desaparecen (outliers).
- Valores que se convierten en NA.
- Filas con NA que desaparecen.

- Tablas que se convierten en grafos, y viceversa.
- Conjuntos de documentos, de correos electrónicos, de tweets que se convierten en grafos.
- Aserciones u opiniones que se convierten en reglas de un sistema experto asociadas a una probabilidad (consulte el capítulo Otros problemas, otras soluciones).

b. Modelado y evaluación

La elección de modelos surge en primer lugar de las características de los datos y de la cuestión planteada, aunque el problema es que esta relación está, en ocasiones, íntimamente vinculada a los propios datos y no está accesible a primera vista.

Otras características pueden influir en la elección del data scientist, como su dominio del modelo, la potencia de cálculo de las máquinas a su disposición o la facilidad de interpretar los resultados del tipo de modelo seleccionado.

El modelo puede ser más o menos difícil de configurar, y por otro lado un modelo potente es, a menudo, difícil de gestionar.

Por modelo, no debemos entender únicamente los algoritmos de machine learning o de optimización, sino toda una cadena que comprende: la transformación y la selección de features, el entrenamiento del modelo y la construcción del modelo basándose en el método de evaluación.

Este proceso es muy iterativo y se realiza bajo el control de una o varias medidas de desviación/de riesgo como las que hemos visto previamente.

En su versión más simple, el trabajo sobre las features y la construcción de los modelos y de su configuración general se realiza sobre los datos de entrenamiento; la evaluación del rendimiento y la puesta a punto definitiva de los parámetros del conjunto se realizan sobre los datos de validación.



Llegado el caso, un algoritmo «maestro» actúa como un proceso de orquestación de las acciones paralelas o segmenta la resolución del problema en diversos procesos antes de consolidarlos (lo que denominamos, de manera global, algoritmos de conjunto o «ensemble»).

Pero ¿cuándo detenerse? La respuesta estará ligada a la lógica de iteración, de modo que hay que detenerse cuando el tiempo dedicado a esta fase finalice, típicamente a dos tercios de la iteración a nivel del proyecto.

La parada no es brusca, y consiste en planificar y ejecutar las últimas tareas indispensables antes de seleccionar un modelo.

En términos de planificación, resulta adecuado prever un margen del 10 % al menos para poder reaccionar si el modelo seleccionado fracasa en la fase de test, en cuyo caso habrá que repetir un microciclo completo (a partir de las features) para darse una última oportunidad de completar la iteración.

c. Escoger el mejor modelo

Para poder escoger el mejor modelo, habrá hecho falta previamente almacenar con cuidado cada versión de nuestra experiencia (datos de entrenamiento y de validación, features, modelo, parámetros, elementos de evaluación...). Para cada una de estas experiencias, a continuación habrá que indicar en una tabla los valores de evaluación obtenidos sobre los datos de entrenamiento y sobre los datos de validación. Seguidamente podríamos clasificar los valores de evaluación de entrenamiento por «error» decreciente y constar un óptimo (mínimo) sobre la curva de evaluaciones para los datos de validación correspondientes a la misma versión del modelo. Este óptimo se corresponde con nuestro mejor modelo no sobreajustado («overfitted») (consulte el capítulo Introducción).

Para problemas complejos, es conveniente realizar esta identificación de modelo sobre varios identificadores de desviación, de riesgo o de error. Seleccionando un modelo un poco menos óptimo sobre un indicador concreto, pero no demasiado lejos de ser un óptimo sobre varios indicadores, es más probable escoger un modelo que no esté sobreajustado a sus indicadores.

Aquí reside una gran diferencia entre las data sciences del mundo real y las data sciences académicas, o de concursos, como Kaggle.com.

En el mundo real, es mucho más difícil afirmar si un modelo es adecuado o no, pues es tarea de los data scientists escoger sobre qué indicadores va a medirse la calidad de su propia predicción.

d. Test, interpretación y confrontación con negocio

El método implementado para la etapa de test es similar a la mecánica de evaluación, salvo por el hecho de que no se trata de iterar o ajustar el modelo o sus parámetros, que son útiles tal cual (*as is*), y porque se utiliza un conjunto de test diferente de los conjuntos de entrenamiento y de validación.

Cuando todo funciona con normalidad, la comprobación sobre los datos de test obtiene los mismos resultados que sobre los datos de validación. En caso contrario, hay que utilizar el margen de tiempo restante para rehacer un microciclo completo y tener así una segunda oportunidad de éxito. Preste atención: la situación se vuelve delicada, pues podemos estar tentados de sobreajustar el modelo sobre los datos de test o incluso terminar haciéndolo.

 Por otro lado, uno de los malos hábitos que en ocasiones adquieren los data scientists que participan en concursos famosos como el que plantea Kaggle.com es que, de propuesta en propuesta, el test se transforma en validación.
¡El que no haya caído alguna vez en este error que tire la primera piedra!

Los resultados y los modelos estabilizados se estudian a continuación bajo el punto de vista de la interpretación: se intenta darle sentido a la construcción de los resultados y a la distribución del proceso que terminará por industrializarse.

El contacto con negocio, los expertos funcionales de la empresa y las últimas pruebas permitirán perfilar el método definitivo.

Los intercambios con los equipos técnicos y los equipos encargados del despliegue y de la gestión del cambio permitirán preparar la puesta en producción, que en ocasiones supone un proyecto en sí mismo.

Dependiendo del caso, el modelo creado será objeto o no de un despliegue operacional. En ocasiones, este modelo y el «debriefing» de la iteración desembocan únicamente en la definición de las próximas iteraciones.

A veces resulta útil reescribir de una manera más fiable el modelo (refactoring), o incluso realizar una proyección en un entorno técnico o en un lenguaje diferente, si encontramos problemas de arquitectura o de rendimiento. La buena calidad de la gestión de la configuración y de la documentación aporta mucha tranquilidad, pues desembocará en una acción de refactoring eficaz y correctamente probada (ver más adelante en el capítulo el contenido de la documentación).

Respecto a la sucesión de iteraciones y de puestas en producción, nuestro consejo es el siguiente: es conveniente gestionar en paralelo nuevas iteraciones de I+D y de explotación operacional de un modelo, incluso aunque sea muy parcial. De esta doble experiencia se obtienen bastantes lecciones útiles y esto «bootstrapa» el proceso, con la única limitación de no decepcionar a los usuarios ni hacerles perder el tiempo trabajando con modelos todavía ineficaces o difíciles de implementar. También conviene asegurar que los costes de integración informática o de formación generados por muchas versiones sucesivas no sean demasiado elevados.

4. Preparación de la industrialización y despliegue

Más allá de los aspectos puramente arquitecturales y técnicos del proyecto informático correspondiente y de los aspectos organizativos o humanos vinculados a la incorporación del producto terminado en el ciclo de vida de la empresa, el data scientist conserva sus responsabilidades particulares durante las últimas etapas.

El workflow operacional que va desde la captación y recogida de datos hasta el tratamiento y el posterior análisis de los datos debe estabilizarse en la organización y validarse por los data scientists antes de su puesta en producción. Esta validación debe realizarse una vez liberada la aplicación (**«go live»**) y durante cierto periodo de observación (VSR: verificación de servicio regular).

Los data scientists deben haber evaluado y diseñado con precisión el ciclo de vida de los datos, de los modelos y de sus parámetros. La organización deberá implementar los procesos y los medios del **mantenimiento operacional** de los modelos «vivos». iPreste atención, pues muchos proyectos prometedores **fracasan operacionalmente debido a descuidos en estas etapas!**

Se plantean cuestiones importantes al equipo que afectan en gran medida al data scientist:

- ¿Cómo hacer evolucionar los modelos y la configuración identificando los puntos siguientes: existencia de versiones mayores y menores, puesta en marcha concurrente de varias versiones de los modelos, evolución y tuning de los parámetros en el corto plazo, comprobación del rendimiento predictivo, evolución de los conjuntos de entrenamiento/validación?
- ¿Cómo asegurar y mantener el rendimiento técnico del modelo?
- ¿Cómo asegurar la gestión de la configuración (y la gestión de las versiones) del conjunto? Esta debería ser ejemplar, pues el conjunto corre el riesgo de hundirse debido a una mala gestión o a la obsolescencia. Conviene mantener el vínculo lógico entre las versiones de los conjuntos de datos (entrenamiento, validación y test), de los modelos, de los conjuntos de parámetros, de los workflows y de los componentes informáticos afectados.
- ¿Cómo recopilar el feedback de los usuarios e integrar sus futuras necesidades?

5. Preparación de las siguientes iteraciones

a. Elementos que es preciso tener en cuenta

Todas las etapas han planteado nuevas cuestiones, nos han llevado a construir nuevas hipótesis, nos han permitido identificar nuevas necesidades, nos han aportado nuevos conocimientos... Ciertos aspectos se han dejado provisionalmente de lado, algunas fuentes de datos no se han explotado, ciertos modelos no se han probado...

Sobre estas bases se construye la continuación del proyecto.

Debe elaborarse y compartirse un acta de cada iteración. A partir de esta acta y de las nuevas perspectivas del cliente interno, se revisarán, precisarán y especificarán las futuras iteraciones.

El método es ágil, aunque su **documentación debe actualizarse escrupulosamente** y el conjunto de componentes debe formar parte de una **gestión de la configuración eficaz**. Una buena documentación no es verbosa, sino que debe ser fácil de mantener. Se parece más a un sistema de gestión del conocimiento que a un sistema de producción de bellos documentos. Para comprobar la calidad de la pareja documentación/gestión de versiones, basta con alternar las tareas entre los miembros del equipo y someter el conjunto a los nuevos miembros (data scientists recién incorporados, por ejemplo), a especialistas de Big Data o a profesionales de la gestión de la calidad, para observar sus reacciones y recoger sus observaciones y sugerencias.

b. Documentación gestionada por los data scientists

No hay que limitar la documentación a los siguientes elementos, aunque estos deberían formar parte de ella:

- Un pequeño resumen didáctico del objetivo de la versión dirigido a personas no técnicas y no científicas (no dude en realizar esquemas sencillos).
- Una *release note* de la versión (lista de componentes y versión de los componentes).
- La lista de requisitos del cliente y técnicos implementados en la versión.
- Una breve nota científica (con el formato de un artículo de investigación) que puedan utilizar otros data scientists internos al proyecto y que deberán leer atentamente el jefe de proyecto y todo el equipo de data scientists. Para garantizar una trazabilidad completa, este artículo científico debería incluir en su anexo la mención precisa y sin ambigüedades de **todas** las fuentes de inspiración utilizadas específicamente en la iteración e indicar su **aportación al proyecto** (otros documentos internos, entre ellos los artículos científicos previos y las actas de las reuniones, páginas de sitios web, código fuente transformado o estudiado, artículos de investigación, artículos de libros, MOOC, lectura de *white papers*, pósteres científicos, conferencias...).
- Todos los elementos de **ciencia reproducible**: los elementos mínimos que permitan a otro equipo reproducir los resultados clave de la iteración y obtener **exactamente** el mismo comportamiento sobre los datos de test (iincluidas las eventuales anomalías!).
- Un acta de la iteración (dificultades, éxitos, respeto a los indicadores y a los objetivos, cuestiones resueltas, cuestiones abiertas, hipótesis de trabajo y recomendaciones para el futuro).

Nuestra recomendación es clara: hay que dedicar al menos el 15 % del coste de la iteración a la gestión de la configuración y el mantenimiento de la documentación. Todo el equipo debería estar implicado en el mantenimiento de esta base de referencia. Típicamente, cada uno debería dedicar de un 10 % a un 20 % de su tiempo a la documentación y a la gestión de la configuración de los componentes bajo su responsabilidad, y deberá planificar tiempo para llevar a cabo esta actividad, que en ningún caso es opcional.

Complementos metodológicos

1. Clasificar sus objetivos

Con el fin de organizar su trabajo, el data scientist debe identificar cuidadosamente la naturaleza real de los objetivos que se le asignan.

Para evaluar la naturaleza de cada uno de los objetivos o requisitos, inspírese con una tipología como la que se muestra a continuación, pero transfórmela en función de sus conocimientos y costumbres.

Los siguientes requisitos están más o menos clasificados por nivel de dificultad. En todo caso, no requieren el mismo espectro de técnicas de data sciences:

- Describir los datos (por ejemplo: 13 % de... son...).
- Interpretar los datos (por ejemplo: en 2015 los electores de... votaron... porque...).
- Optimizar el proceso logístico o mecánico.
- Predecir evoluciones de las tendencias.
- Dar apoyo a los actores en su toma de decisiones.
- Identificar comportamientos anómalos (fraudes, ataques...).
- Identificar el impacto de una evolución potencial de la tendencia sobre otra tendencia.
- Identificar la estrategia para cambiar las tendencias (es decir, encontrar sucesiones del tipo causa-efecto, por ejemplo para influir sobre los que tienen influencia).
- Identificar el impacto de un potencial evento sobre actores precisos.
- Cambiar el comportamiento puntual de los actores (es decir, determinar o reaccionar: por ejemplo, generar un acto de compra directa a través de una petición expresa).
- Identificar señales débiles de evolución de tendencia (cambios iniciados vs. cambios potenciales).
- Decidir en lugar de un actor humano en un contexto «normal».
- Simular el comportamiento de actores de principio a fin sobre un cierto espacio de tiempo.
- Identificar señales débiles de cambio de comportamiento de un actor (acciones iniciadas vs. acciones potenciales).
- Identificar señales débiles de que se produzca una catástrofe (hundimiento de un mercado, crisis, catástrofes naturales, avería en una infraestructura, conflictos...: procesos iniciados vs. procesos potenciales).
- Decidir en lugar de un actor humano en un contexto disruptivo.

2. Trucos y argucias

Incluso cuando disponga de muchos datos, empiece estudiando muestras de tamaño reducido, lo que facilita su representación visual y los primeros intentos de interpretación, acelera los algoritmos y disminuye el sobreajuste («overfitting»).

Resulta muy satisfactorio confirmar haber encontrado un primer modelo que se generaliza bien partiendo de una pequeña muestra, antes de tener en cuenta el conjunto de datos completo a nuestra disposición.

De manera inversa, a menudo resulta más conveniente trabajar sobre la colección completa de un gran volumen de datos que refinar un modelo alimentado solo con datos parciales.

Un buen trabajo de feature engineering a menudo es más eficaz que la búsqueda de un mejor algoritmo.

Un trabajo cuidadoso de visualización de los datos y de los resultados nunca resulta inútil, intente anticiparse siempre a los resultados posibles o plausibles de una representación gráfica antes de ejecutarla. Esto le ayudará a descubrir sus prejuicios respecto a la naturaleza del problema.

Contemple seriamente la posibilidad de que sus datos estén sesgados por ciertas características del proceso de recogida de información. Estudie cuidadosamente todas las distribuciones de sus datos, las dependencias y la posibilidad de que estén manipulados.

No sea trivial en la exploración de los datos textuales, de las encuestas, de la recogida de opiniones, de sentimientos, de puntos de vista. Todo lo que es declarativo refleja una intención y está influido directamente por el contexto de la recogida de la información.

Cuando elementos temporales (fechas, duraciones...) o una secuencia de acciones o de eventos formen parte de sus features (o puedan depender de ellos), doble su atención sobre las interpretaciones erróneas y sobre las hipótesis de uso de sus algoritmos.

El tiempo es un generador importante de sesgo en la recogida de información que debería ser «intemporal». Cuando se recogen datos discretos sobre un eje temporal, hay que estar muy atento al ritmo de la captura, que puede estar sincronizada accidentalmente con los armónicos del proceso. Más bien al contrario, podemos querer organizar la recogida de información en función de un ritmo determinado para evitar un sesgo semántico (recoger las cotizaciones de la bolsa a mitad o a final de temporada no tiene el mismo sentido).

Por otro lado, es importante trabajar en el marco de las prácticas básicas de «data science reproducible»:

- Gestionar versiones del conjunto de componentes (incluidas las aplicaciones y los conjuntos de parámetros).
- Realizar un script de todas las acciones, prohibiendo los cálculos manuales.
- Documentar los análisis intermedios realizados sobre las representaciones gráficas, o traducirlas en pruebas estadísticas objetivas.
- Reiniciar sistemáticamente los generadores de números aleatorios (set.seed).
- Salvaguardar los resultados intermedios, en particular antes y después de las transformaciones de los datos.
- Comentar nuestro código.
- Integrar pruebas unitarias en nuestro código.

Procesamiento del lenguaje natural

Definición del problema

La importancia de las aplicaciones de *text mining* no ha dejado de aumentar en estos últimos años. La emergencia de las redes sociales ha intensificado este fenómeno. La principal característica de los datos textuales, y que los diferencia de los datos semiestructurados (archivos XML o JSON) o estructurados (bases de datos), reside en la necesidad de tener que inspeccionar de forma no determinista dentro de cada ítem de datos.

Los datos textuales pueden tratarse a diversos niveles de profundidad:

- Identificación de palabras que pertenecen a listas de palabras (*bag of words*).
- Identificación de cadenas de palabras (frases o expresiones).
- Identificación de elementos semánticos (reconocer palabras en función de su significado).

Otra de las características, típica de un texto, reside en la débil proporción que existe entre el número de palabras de un texto y el número total de palabras posibles. Esto crea estructuras de datos muy dispersas, en particular cuando se trata de representar la información como una estructura en forma de tabla (*sparse matrix*).

En este capítulo, vamos a centrar nuestra atención en un método potente, pero que puede resultar algo disuasivo en cuanto a su formulación: el análisis semántico latente.

Existen bastantes otras técnicas en el contexto NLP muchas son técnicas muy útiles, pero que intervienen sobre todo al inicio de los procesamientos de data sciences extrayendo features. Trabajaremos con ellas de la forma «habitual».

Análisis semántico latente y SVD

1. Aspectos teóricos

Llamado LSA (*Latent Semantic Analysis*) en inglés, el análisis semántico latente se basa en la construcción de una matriz que incluye los valores de una función particular calculada a partir de las ocurrencias de los distintos términos (palabras) presentes en los diferentes documentos.

Los documentos pueden ser textos, correos electrónicos, tweets, artículos escritos en blogs, CV...

Cada término forma parte de una fila de esta matriz, y cada columna de la matriz se corresponde con un documento.

En la intersección de filas y columnas se encuentra el resultado del cálculo de una función. Este resultado es tanto mayor cuanto mayor sea el número de ocurrencias del término en el documento y si el término es extraño en general. Los paquetes se proporcionan con diversas funciones clásicas, aunque nuestro problema requerirá posiblemente refinar nuestra propia función. Dicha función siempre debe reflejar el hecho de que **un término extraño es más significativo que un término corriente**.

La función que mide habitualmente la importancia de un término en un documento en función del corpus se denomina TF-IDF (*Term Frequency-Inverse Document Frequency*). Se basa en la ley de Zipf, que trata de la frecuencia de las palabras en un texto y cuya interpretación teórica hace referencia a la noción de entropía.

El uso de la entropía es un excelente signo de objetividad. También es buena noticia respecto al número de hipótesis empíricas que es preciso verificar sobre la calidad del corpus.

Por tanto, debería considerar seriamente el construir su propia función si su problema es algo particular.

 Cuando se dispone de varios corpus y de varias «funciones de importancia», puede resultar extremadamente eficaz realizar análisis cruzados sobre estos diversos aspectos y quedarse con unas pocas features tras cada cruce.

Las features obtenidas son, a veces, complementarias, relativamente independientes y altamente interpretables.

En muchos casos de implementación, los términos considerados son palabras que se han desprovisto de sus características de contexto (indicaciones de plural y de género, mayúsculas al comienzo de la frase, elementos de conjugación...).

a. SVD: generalidades

La matriz X se descompone mediante una técnica que se denomina «descomposición en valores singulares» (SVD).

Esta técnica posee muchos usos en data science, en particular para reducir el número de dimensiones.

Obtenemos la siguiente descomposición:

$$X = U\Sigma V^T$$

Las matrices U y V son matrices ortogonales y, por tanto, tales que $U^T U = I$ y $V^T V = I$, el cuadrado de su determinante es igual a 1.

La matriz Σ es una matriz diagonal (de modo que todos sus términos que no están en la diagonal son nulos).

Observe que siempre se obtiene una descomposición y que esta es única.

Observación para los matemáticos: una matriz ortogonal siempre se parece a una matriz compuesta de una diagonal de matrices de rotaciones planas y de 1 y -1. Por «similitud», entendemos que existe una matriz de cambio de base que permite transformarla de esta manera. En el caso de una matriz ortogonal, la matriz de cambio de base es la propia matriz ortogonal!

b. Una justificación de la descomposición SVD

Consideremos la descomposición:

$$X = U\Sigma V^T$$

Imagine que la matriz X que se ha de descomponer tenga como dimensión 10 000 términos x, 100 000 documentos y que la matriz diagonal Σ sea una matriz cuadrada de 5 000 x 5 000.

La matriz U tendría como dimensión 10 000 x 5 000 y la traspuesta de V tendría como dimensión 5 000 x 100 000.

De modo que nuestra matriz X contendría 10^9 celdas, mientras que la suma del número de celdas de las tres matrices sería: 575.10^6 .

En tal caso, hemos reducido el volumen de datos en un 50 % sin perder ninguna información. Esto es habitual cuando la matriz X es muy dispersa (sparse matrix: con muchos ceros, que no se codifican para ganar espacio).

De hecho, veremos más adelante que la aplicación de este método nos ofrece otra oportunidad de ganar volumen. En efecto, podríamos suprimir las filas y las columnas menos significativas de las matrices resultado de la descomposición sin perder información.

c. SVD en el contexto LSA

En un sentido funcional, en este contexto, cada **fila** de la matriz U es un **término** y cada **columna** de la matriz U puede interpretarse como un **tema**, es decir, una **combinación de términos** (palabras) con un coeficiente de ponderación que exprese el peso de cada término en el documento.

La matriz Σ es una matriz diagonal en la que cada valor se corresponde con el peso del tema respecto al conjunto de temas del corpus de los documentos estudiados.

La matriz V^T nos proporciona filas de temas cuyas columnas representan los documentos.

Podemos **disminuir las dimensiones del problema eliminando los temas que tengan los pesos más bajos**, lo que permite eliminar las filas correspondientes de U y V y las filas y columnas correspondientes de la matriz diagonal Σ (todas las últimas filas, pues la matriz diagonal se clasifica en función de los valores de los pesos).

d. Interpretación

Para percibir mejor el sentido de todo esto, hay que integrar la idea de que la matriz U está compuesta de varios vectores (columnas) que representan nuestros temas (es decir, nuestros conceptos) y que se ha fabricado **un nuevo espacio cuyos ejes son los temas** (los que hemos conservado perdiendo muy poca información).

Podemos **representar cada documento en este nuevo espacio** de menor dimensión y mejor adaptado a la topología real de los temas/conceptos subyacentes.

La nueva dimensión del problema es más razonable, y este espacio adquiere sentido, de modo que resulta viable

utilizar algoritmos clásicos de clasificación o de aprendizaje sobre este nuevo espacio.

En matemáticas, se denomina vectores singulares a los vectores (columnas) de U , y valores singulares a los valores de la diagonal de Σ .

La descomposición SVD minimiza la distancia euclíadiana de los documentos con los vectores singulares. El cuadrado del valor singular correspondiente a un vector singular se percibe a menudo como una «energía» latente. Para escoger qué vectores se van a eliminar, es posible aplicar un método de Pareto 80/20 a esta energía. Dicho de otro modo, se suma sucesivamente los cuadrados de los valores singulares, desde el más grande hasta el más pequeño, y, cuando esta suma alcance el 80 % de la suma total de los cuadrados, se considera haber encontrado el índice del último vector singular. Esta mecánica elimina todos los vectores columnas que siguen (volveremos a ver esta técnica en otro contexto de reducción de dimensión en el capítulo Feature Engineering). El umbral de 80 % es indicativo, es un parámetro como cualquier otro del modelo que convendrá optimizar en función de la calidad de los residuos y del rendimiento esperado.

SVD capta las «correlaciones» lineales, de modo que, en caso de problema medianamente lineal, la cuestión es saber si conviene transformar sus features o la variable explicada antes de utilizarlas para trabajar sobre un problema más lineal. Cuando el contexto no sea realmente lineal, es posible contemplar el algoritmo **Isomap**.

Existen otros tipos de descomposición de matrices con usos similares:

- «CUR matrix decomposition», que devuelve un resultado no único y aproximado, pero que en ocasiones es más sencillo de interpretar.
- «Eigen-decomposition», que es el método de referencia que la esposa del autor (Sra. Ghislaine Laude-Dumez) ya utilizaba en sus investigaciones relativas a la **construcción de clústeres utilizando distancias de similaridades lingüísticas en sus trabajos de doctorado en 1986...** sus trabajos nos han inspirado desde entonces y hasta ahora. Abordaremos uno de los requisitos previos en el capítulo Feature Engineering cuando hablemos de PCA (*Principal component analysis*: análisis de componentes principales).

e. Alternativa no lineal, Isomap (MDS, geodésico, variedad, manifold)

Isomap se basa en la exploración «geodésica» de una variedad (*manifold*) utilizando la mecánica de escalamiento multidimensional (MDS).

MDS, que hemos abordado indirectamente cuando hemos hablado antes de «similaridad», permite abordar una relación de proximidad que puede basarse en distancias cuya topología no conocemos ni dominemos necesariamente.

La idea de una exploración geodésica es sencilla: en lugar de considerar el espacio en el que está ubicado el objeto que se desea explotar y considerar que la línea recta es el camino más corto, se acepta una topología espacial sobre la que nos desplazamos. Considere este ejemplo: sobre tierra, el camino más corto operacional entre dos puntos alejados es un arco de circunferencia, y no la cuerda que enlaza ambos puntos pasando por el sol!

Una variedad (manifold) se parece a una superficie ubicada en una dimensión mayor, como lo es por ejemplo la superficie de la Tierra en el espacio de 3 dimensiones que la rodea. Esta noción permite comprender propiedades topológicas no evidentes: por ejemplo, la variedad llamada banda de Möbius, es una superficie que posee una sola cara!

Planteándose una pregunta como: «¿cómo aplicar el método del gradiente sobre las geodésicas de una variedad como la banda de Möbius?», **accederá a la verdadera naturaleza del problema de un data scientist** que adorase un mundo lineal y que descubriese mundos complejos cuando el número de dimensiones aumentara.

2. Puesta en práctica

a. Inicialización

Para realizar este ejemplo práctico, hemos copiado seis archivos de texto en inglés (.txt), libres de derechos, en una carpeta de nuestro equipo.

El volumen total de este minicorpus es de 2.4 MB.

La codificación de los archivos es UTF-8. Para transformar un archivo de texto manualmente en UTF-8, basta con un editor de texto que permita realizar esta opción en el momento de guardar el documento (por ejemplo: WordPad).

Vamos a utilizar un paquete LSA que encapsula la descomposición SVD, aunque es posible utilizar otros paquetes de descomposición SVD.

```
rm(list = ls())
library(lsa)                                # análisis semántico latente
getwd()                                         # archivos in /clásicos
dir <- paste(getwd(),"clasicos", sep = "/")
```

La ruta donde se encuentra el archivo se construye agregando la ruta en curso y el nombre de la carpeta.

En procesamiento del lenguaje natural, es habitual eliminar ciertas palabras antes de realizar los procesamientos. La razón más común es que estas palabras no son significativas, aunque existen otras muchas razones para hacerlo (por ejemplo, para eliminar nombres de empresas, de personas...).

Se llama «stop words» a estos conjuntos de palabras. Los paquetes le proporcionan stop words para diversos idiomas, aunque los distintos laboratorios gestionan sus propios paquetes de palabras para eliminar en función de los contextos y de los idiomas.

```
data(stopwords_en)                           # stopword en inglés
stopwords_en[1:5]
```

```
[1] "a"      "about"   "above"   "across"  "after"
```

El paquete **lsa** nos permite fabricar una matriz de términos de nuestros archivos que incluye todas las palabras del corpus **salvo las stop words**.

```
                                # todas las palabras
M_palabras= textmatrix(dir, stopwords=stopwords_en,
                       stemming=FALSE,
                       minGlobFreq=0)
dim(M_palabras)
M_palabras.df <- data.frame(M_palabras[,])
```

```
[1] 14208      6
```

La matriz está en un formato propio del paquete, que incluye muchos atributos de listas; veremos su manipulación más adelante.

Para echar un vistazo rápido al contenido de esta matriz de 14 208 filas de términos y 6 columnas de documentos, hemos creado un **data.frame** temporal que no nos servirá para nada más. En RStudio puede visualizar fácilmente las 1 000 primeras filas.

Existen muchos algoritmos para limpiar las palabras, de modo que palabras como «loves», «loved»... se conviertan en una misma entidad.

Preste atención: el resultado no está formado por palabras, sino por **radicales (stem)**. Por ejemplo, a una palabra importante de nuestro conjunto de datos, happy, le corresponde la expresión «happi», que es su *stem*.

```
# los radicales/stem
M= textmatrix(dir, stopwords=stopwords_en,
              stemming=TRUE,
              minGlobFreq=0)
dim(M)
```

```
[1] 9321      6
```

Todavía son bastantes radicales (9 321). A continuación vamos a configurar nuestro código para eliminar las impurezas, el valor del parámetro es empírico y requiere varios ensayos.

```
# menos stems
M= textmatrix(dir, stopwords=stopwords_en,
              stemming=TRUE,
              minGlobFreq=6)
dim(M)
M.df <- data.frame(M[, ])
```

```
[1] 349      6
```

Veamos a qué se parece esta «textmatrix».

```
attributes(M)$dim           # estudio estructura de M
attributes(M)$class
attributes(M)$dimnames$terms
attributes(M)$dimnames$docs
```

```
[1] 349      6
[1] "textmatrix"
[1] "abl"   "act"   "actual"  "ad"   "age"   "ago"   ...
[1] "Hemingway - Garden of Eden.txt"
```

```
[2] "Hemingway - Green Hills of Africa.txt"
[3] "Lewis Carroll - AliceTLG.txt"
[4] "Lewis Carroll - AliceWonderLand.txt"
[5] "Nietzsche - Beyond Good n Evil.txt"
[6] "Tolkien - The Fellowship Of The Ring.txt"
```

b. En el núcleo de LSA

A continuación hay que abordar un aspecto clave: la implementación de dos funciones. Una de ellas dará preferencia a los términos más numerosos y la otra a los términos raros: TF-IDF (*Term Frequency-Inverse Document Frequency*). Puede trabajar sobre el algoritmo redefiniendo estas funciones.

```
X <- lw_logtf(M) * gw_idf(M) # weight functions
```

A continuación se calculan todos los valores singulares de esta matriz y se selecciona una porción mínima.

Siempre es mejor empezar nuestras pruebas por el menor número de valores singulares, siendo dos a menudo un valor ideal cuando el problema se preste a ello, pues la interpretación de los resultados resulta más fácil. Como todos los parámetros, esto deberá ser objeto de una optimización en función de su problema y de la eficacia comprobada sobre el conjunto.

Tras determinar el número de valores singulares (en esta iteración), se aplica simplemente sobre nuestra matriz **termsXdocuments** y se obtiene una estructura compleja llamada **SVD**, que contiene nuestras tres matrices **U**, **S** (es decir, Σ) y **V**.

```

SVD <- lsa(X, dims=dimcalc_raw())           # todos los valores
                                                # singulares

S     <- SVD[3] # diagonal valores singulares
S     <- unlist(S)
n_s   <- sum((cumsum(S)/sum(S))<0.8)        # Pareto
n_s

SVD <- lsa(M, dims=n_s)                      # filtro energía
S     <- SVD[3] # diagonal valores singulares
S     <- unlist(S)                            # viejo truco
S     <- diag(unlist(S))
S

```

```
[1] 2
      [,1]      [,2]
[1,] 1713.596  0.0000
[2,]    0.000 507.8745
```

Como tenemos dos valores singulares, la matriz S es una matriz diagonal de dimensión **2x2**. Los dos valores singulares son 1714 y 508.

Vamos a formar la matriz U de términos por tema (o concepto) y visualizar sus 6 primeras filas.

```
head(U)
```

	tk.1	tk.2
abl	-0.016965571	0.0001335879
act	-0.002980738	-0.0141344135
actual	-0.015012501	0.0021472618
ad	-0.018008667	0.0105258476
age	-0.022767656	0.0175027865
ago	-0.048814008	0.0615980721

Los valores de la matriz **U** indican **la contribución de estos stems a cada uno de los dos temas**.

El análisis detallado del contenido de los temas resulta, a menudo, muy ilustrativo, aunque solo sea para interpretar los resultados o confirmar algún defecto en el modelo.

c. Resultados

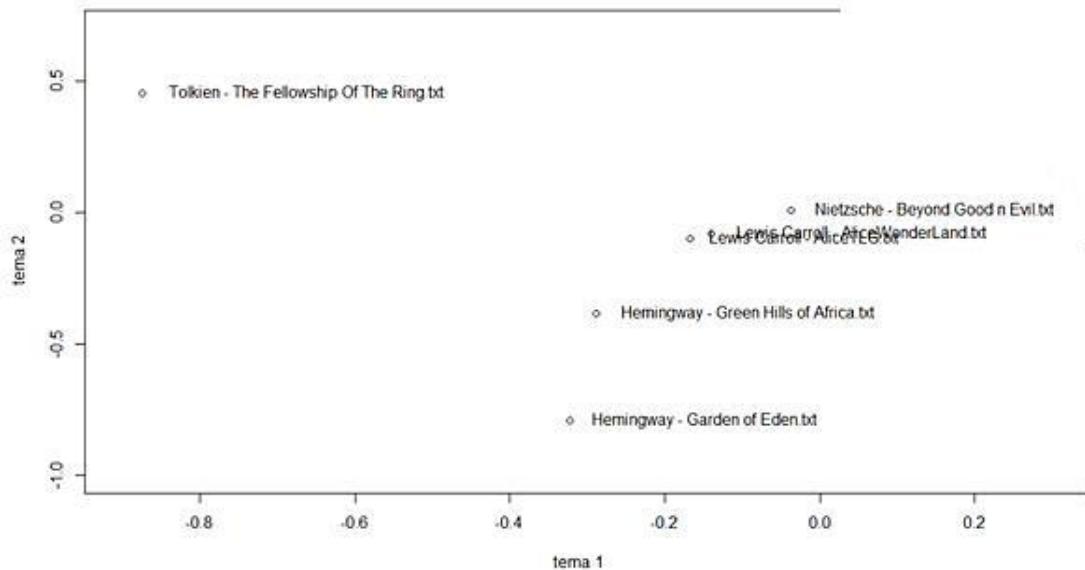
La matriz V es muy importante, pues nos devuelve la posición de los documentos respecto a los dos nuevos ejes de temas. Decimos que se van a visualizar los documentos en **el espacio de conceptos**.

```
V <- SVD[2] # V documento / temas (vectores singulares izquierdos)
V <- data.frame(V)
V <- as.matrix(V)
head(V)
```

	dk.1	dk.2
Hemingway - Garden of Eden.txt	-0.32281603	-0.79405444
Hemingway - Green Hills of Africa.txt	-0.28823778	-0.38469400
Lewis Carroll - AliceTLG.txt	-0.16717706	-0.10215457
Lewis Carroll - AliceWonderLand.txt	-0.14035622	-0.07993375
Nietzsche - Beyond Good n Evil.txt	-0.03757907	0.00551130
Tolkien - The Fellowship Of The Ring.txt	-0.87386988	0.45236350

```
plot(V,      # plot en el espacio de conceptos / temas
      xlab = "tema 1",
      ylab = "tema 2",
      xlim = c(-0.9,0.3),  # ajuste manual
      ylim = c(-1, 0.7))  # ajuste manual

text(V, labels = row.names(V),  # etiquetas
     adj    = c(-0.1,0.3))  # ajuste manual
```



Representación de los documentos en el espacio de temas (conceptos)

El resultado resulta agradable:

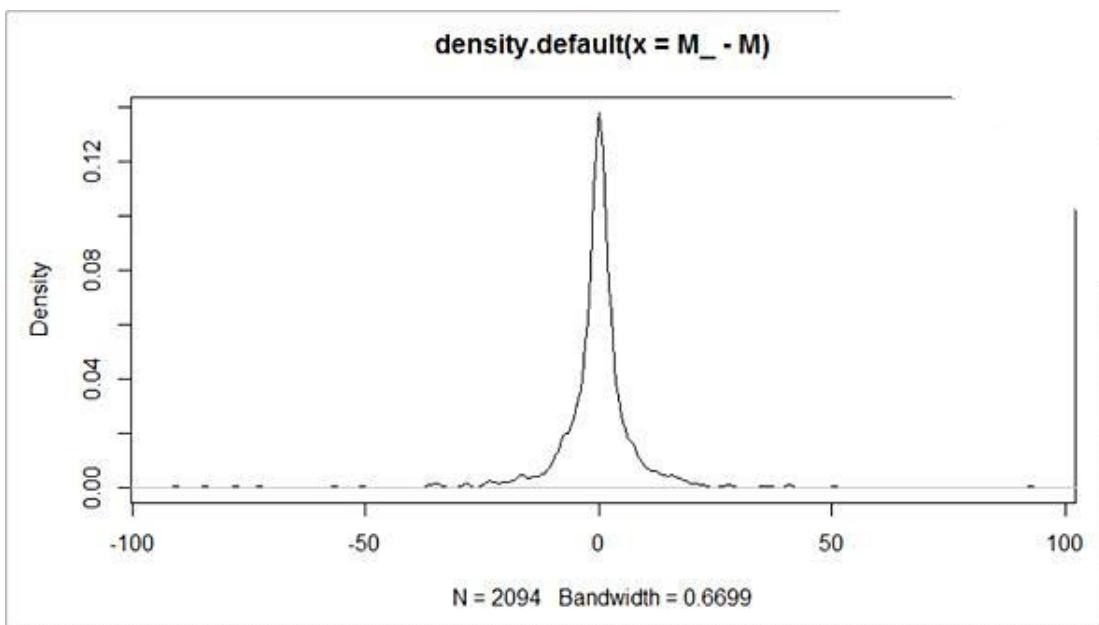
- Los dos libros de Lewis Carrol están juntos, lo que nos satisface, pues hablan acerca de dos desarrollos de la misma historia.
- Los dos libros de Hemingway se encuentran en la misma zona.
- El libro de Tolkien aparece alejado de todo esto.
- El espíritu ácido de Nietzsche se ha encontrado con el espíritu atormentado de Lewis Carrol...

Antes de delirar sobre las interpretaciones, una primera preocupación debería ser comprobar la calidad de la reconstrucción de la matriz M.

Para ello vamos a comparar la matriz original y la matriz reconstruida a partir de dos temas multiplicando en conjunto las tres matrices que resultan de la descomposición (sin olvidarnos de trasponer \mathbf{V}).

```
M_ <- U %*% S %*% t(V)           # reconstrucción y test
dens <- density(M_-M)
plot(dens)
sd_dens <- sd(M_-M)
sd_dens
```

[1] 8.326368 [desviación típica]



Densidad de los residuos entre la matriz de términos original y la matriz reconstruida

La forma de esta curva nos tranquiliza por el hecho de que las distancias sean próximas a cero (haría falta investigarlo, como de costumbre, aunque no es el objetivo de este ejemplo).

d. Manipulaciones, interpretaciones recreativas y no fundadas

El resto de este ejemplo tiene como único objetivo manipular los datos para apelar a su imaginación; no lo generalice todo en términos de «método», pero recuerde ciertas prácticas que le serán, sin duda, útiles.

El producto de los dos últimos términos de la descomposición devuelve los temas por documento.

```
StV <- S %*% t(V)           # temas por documento
head(StV)
```

```
Hemingway - Garden of Eden.txt Hemingway
[1,]          -553.1761
[2,]          -403.2800
.../...
```

El producto de los dos primeros términos de la descomposición devuelve los términos por tema.

```
US <- U %*% S           # términos por tema
head(US)
```

	[,1]	[,2]
abl	-29.072127	0.06784587
act	-5.107779	-7.17850810
actual	-25.725356	1.09053949
ad	-30.859571	5.34580952
age	-39.014555	8.88921888
ago	-83.647468	31.28408977

Vamos a estandarizar cada columna de la matriz U utilizando la técnica **muy sencilla y útil del z-score**: para normalizar un vector de esta manera, se sustrae su media de cada componente y a continuación se divide cada resultado por la desviación típica. De este modo, se vuelve a centrar la variable y se normaliza su ámbito.

```
# normalización z-score
# d las dos columnas de U
z_score_1 <- (U[,1] - mean(U[,1]))/sd(U[,1])
z_score_1[1:5]

z_score_2 <- (U[,2] - mean(U[,2]))/sd(U[,2])
z_score_2[1:5]
```

	abl	act	actual	ad	age
0.4614357	0.8098867	0.5100991	0.4354456	0.3168690	
	abl	act	actual	ad	age
0.02787277	-0.23837930	0.06544950	0.22180047	0.35199561	

Los rangos de valores son ahora comparables y permiten intentar poner de relieve aquellos términos caracterizados por las mayores distancias entre ambos temas (para tratar de interpretar «funcionalmente» de qué se trata).

La distancia en valor absoluto nos permite definir la amplitud de la diferencia. De este modo, resulta sencillo extraer la lista de términos caracterizados por una gran diferencia (superior a tres veces la desviación típica). Se recuperan estos términos expresando la distancia con signo (es decir, negativa o positiva).

```
# distancia entre temas
# búsqueda de los max

delta_U <- z_score_2 - z_score_1
delta_U_ <- abs(delta_U)
sd(delta_U_)
which(delta_U_ > 3*sd(delta_U_))
lista_delta_grande <- delta_U_[which(delta_U_ > 3*sd(delta_U_))]
sort(lista_delta_grande) # palabras disjuntas XOR

sl[1:4]
sl[(length(sl)-5):length(sl)]
```

[posición en la matriz]

am	answer	call	day	don	door	eye	fear	found	
11	13	32	56	69	70	93	96	109	.../...

[4 stems netamente más fuertes en el tema 1 que en el tema 2]

love	don	pleas	read
-4.025184	-3.328342	-3.268420	-3.189584

[5 stems netamente más fuertes en el tema 2 que en el tema 1]

look	am	day	time	eye	sudden
4.531457	4.650500	4.655193	4.822226	5.160135	5.423675

Ahora estamos en posición de tratar de interpretar rápidamente los temas, y a continuación de clasificar estos documentos respecto a los temas.

¿El tema_1 sería una especie de medida de asuntos relacionados con la **empatía** y el tema_2 una medida de asuntos relacionados con la **inmediatez**?

Le dejamos que juzgue usted mismo, pero para alimentar su reflexión, veamos qué está vinculado a «love» en nuestra base de términos y qué está vinculado a «sudden».

```
associate(M, "love") [1:10]
associate(M, "sudden") [1:10]
```

```
[love]
happi    wine drink write   read
0.991   0.986 0.986 0.985 0.984
```

```
[sudden]
set      expect   save    feet   glanc
0.999   0.998 0.998 0.997 0.997
```

Resulta inquietante imaginar que los temas de inmediatez y de empatía separaban así de bien a Tolkien de los demás autores, que Hemigway utiliza un vocabulario más estático que Tolkien y aborda menos temas relacionados con la empatía que Lewis Carrol o Nietzsche...

Grafos y redes

Introducción

El estudio de los grafos nos lleva a abordar diversos aspectos. Las principales diferencias o similitudes se estudian bien entre los grafos o partes de grafos, o bien entre los nodos del grafo (típicamente en un grafo o en una parte de un grafo).

El capítulo Introducción le ha presentado la noción de grado, que es una feature simple, aunque por desgracia insuficiente, para cualificar los grafos y los nodos.

Existen núcleos (kernels) aplicados a los grafos que permiten capturar y manipular a continuación diversas características de la estructura de los grafos a través de nuestras técnicas de machine learning habituales, que no abordaremos aquí. Este capítulo se centra en aquellos aspectos verdaderamente específicos de los grafos.

Primeros pasos

La teoría de grafos comprende muchos conceptos, teoremas y notaciones muy precisas, pero algo preocupantes, pues estas definiciones varían en función de los autores. No definiremos matemáticamente todos los conceptos subyacentes a nuestro uso de los grafos, pues no bastaría con un capítulo. Aquí encontrará una selección de los términos clave y de definiciones «en español» bastante fiables que nos permitirán manipular los grafos en un contexto operacional. En caso de duda, diríjase a las fuentes citadas en la introducción del libro.

1. Algunas nociones y notaciones complementarias básicas

Le conviene empaparse de las siguientes nociones y tratar de reconocerlas en los grafos que encuentre.

Un grafo finito no orientado sin bucle (*loop*), es decir, un **grafo simple** (*undirected simple*) se escribe $G(V, E)$. Preste atención, en ocasiones se llama de la misma manera a los grafos de otra naturaleza (no simples, orientados, hipergrafos...).

Sus nodos o *vertices* se escriben $V(G) = \{v_1, \dots, v_n\}$.

 Nota de vocabulario: en inglés, *vertices* es el plural de *vertex*.

Sus aristas o *edges* se escriben $E(G) = \{e_1, \dots, e_n\}$.

Cuando no existe ambigüedad, resulta inútil escribir la (G) .

Las aristas son elementos del producto cartesiano $V \times V$.

Dos nodos diferentes son adyacentes cuando comparten la misma arista.

Dos aristas diferentes son adyacentes cuando comparten un mismo nodo.

Una cadena (*walk*) entre dos nodos diferentes, sus extremos, es un grafo de nodos adyacentes tales que solo los extremos poseen un grado igual a 1 dentro de este grafo. En ocasiones se habla de cadena abierta u *open walk*.

Una cadena elemental (*trail*) es una cadena cuyo grado máximo de un nodo es 2: no vuelve a pasar jamás por el mismo nodo (preste atención: en ciertos textos el término cadena designa una cadena elemental).

Un ciclo (*closed walk, cycle*) es un grafo de nodos adyacentes de grados al menos iguales a 2 dentro de este grafo.

Un ciclo elemental es un ciclo cuyos nodos poseen exactamente un grado igual a 2. Preste atención: en ciertos textos el término ciclo designa un ciclo elemental.

Un ciclo de orden n puede escribirse C_n .

Un grafo completo, que vincule todos los nodos, de orden n , puede escribirse K_n .

El orden (*order*) del grafo que se corresponde con el número de nodos se escribe $|G|$.

El tamaño (*size*) del grafo que se corresponde con el número de aristas se escribe $||G||$.

Una ruta (*path*) entre dos nodos distintos, sus extremos, es un grafo orientado de nodos adyacentes por sus arcos, tales que solo uno de los extremos posee un grado entrante igual a 0 dentro de este grafo; el otro extremo es entonces el único nodo del grafo que posee un grado saliente igual a 0.

Una ruta elemental es una ruta cuyos nodos poseen grados entrantes y salientes inferiores o iguales a 1: la ruta no vuelve a pasar jamás por el mismo nodo (preste atención: en ciertos textos el término ruta designa una ruta elemental).

Una ruta o *path* de orden n se escribe P_n .

Un circuito es un grafo orientado de nodos adyacentes por sus arcos de grados entrantes y salientes al menos iguales a 1 dentro de este grafo.

Un circuito elemental es un circuito cuyos grados entrantes y salientes son iguales a 1.

El grado de un nodo se escribe $d_G(v)$. Con la menor ambigüedad, habrá que hacer referencia al grafo sobre el que se ha definido el grado.

El grado entrante, o semigrado interior, de un nodo se escribe: $d_G^-(v)$.

El grado saliente, o semigrado exterior, de un nodo se escribe: $d_G^+(v)$.

Evidentemente: $d_G(v) = d_G^-(v) + d_G^+(v)$.

La ruta más corta (o la cadena más corta) designa una geodésica (*geodesic*) del grafo respecto a una distancia determinada (por ejemplo, el número de aristas o de arcos que deben recorrerse para ir de un nodo a otro, o la suma de los valores indicados por los arcos que hacen referencia a alguna cantidad con características de distancia o de medida de similaridad en otro espacio).

Una geodésica es la generalización de la noción de línea recta en una superficie.

El hecho de comprender la noción de ruta más corta en referencia a esta noción nos permite abrir la mente a representaciones muy sofisticadas del espacio y, en particular, a espacios en los que se inscriben nuestros grafos.

2. Manipulaciones simples de grafos con R

Primeros grafos e ilustración de nociones básicas

El paquete **igraph** permite manipular los grafos con un excelente rendimiento. También existe en Python y en C++.

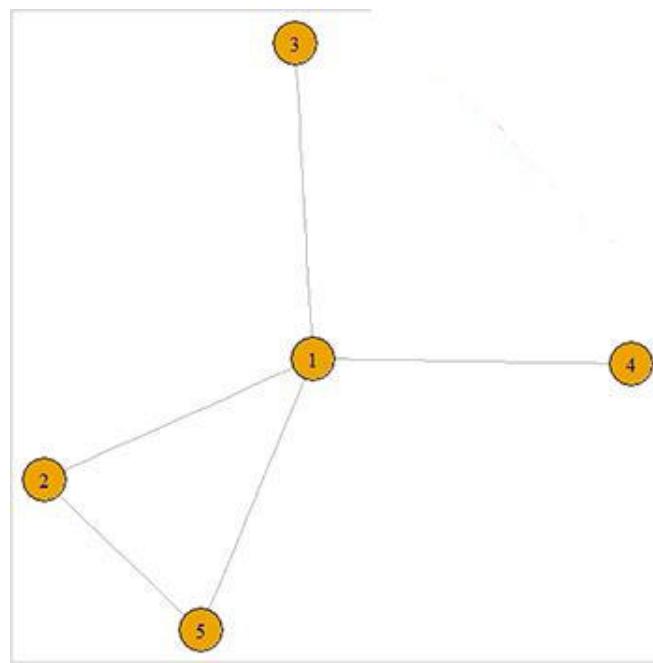
El uso básico de este paquete es muy simple. Ha seducido a muchos desarrolladores, y existen además muchos otros paquetes de R que se basan en él: ¡es un valor seguro!

```
if(require("igraph") == FALSE)
  install.packages("igraph",
                   dependencies=c("Depends", "Suggests"))
library(igraph)

G<- graph.formula(1-2, # un grafo simple
                  1-3,
                  1-4,
                  2-5,
                  5-1)
plot(G)                 # plot sin ningún parámetro
V(G)                   # vertices (nodos)
E(G)                   # edges (aristas)
```

```
+ 5/5 vertices, named:
[1] 1 2 3 4 5

+ 5/5 edges (vertex names):
[1] 1--2 1--3 1--4 1--5 2--5
```



Un grafo simple con un ciclo elemental de orden 3

Este grafo simple (no orientado) contiene un ciclo elemental de orden 3: {"1--2", "2--5","5--1"}

Este ciclo elemental es exactamente igual a: {"1--5","1--2","5--2"}

Contiene, entre otras, dos cadenas elementales que van del nodo "5" al nodo "4": {"5--2","2--1","1--4"} y {"5--1","1--4"}

El grado del nodo "1" es igual a 4, el del nodo "4" igual a 1.

También es posible crear un grafo orientado.

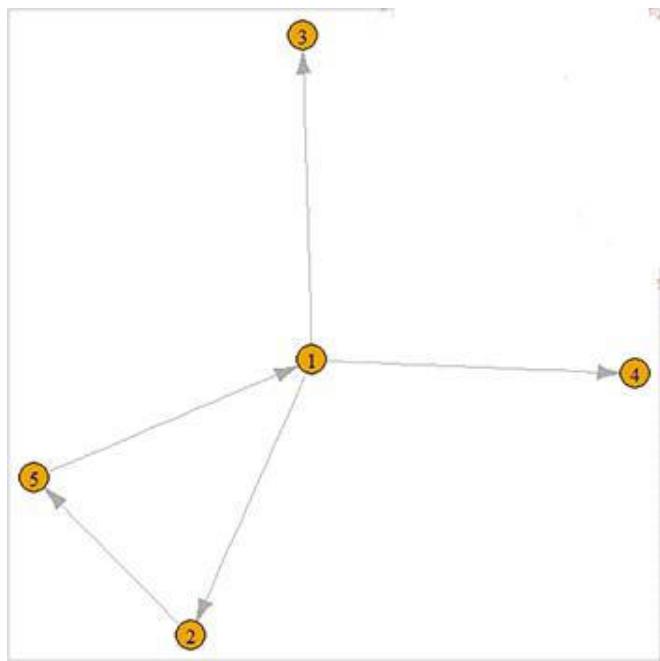
```
igraph.options(vertex.size = 10,
              edge.arrow.size =0.2)

G<- graph.formula(1->2, # directed G
                   1->3,
                   1->4,
                   2->5,
                   5->1)
plot(G)                  # plot sin ningún parámetro
V(G)                     # vertices (nodos)
E(G)                     # edges (arcos!!)
```

```
+ 5/5 vertices, named:
[1] 1 2 3 4 5

+ 5/5 edges (vertex names):
[1] 1->2 1->3 1->4 2->5 5->1
```

Las primeras líneas de código permiten implementar las opciones generales de los plots, imponiendo el diámetro visual de los nodos y el tamaño de las puntas de las flechas. En la fórmula del grafo, los + indican la ubicación de las puntas de las flechas.



Grafo orientado con un circuito elemental de orden 3

Este grafo orientado contiene un circuito elemental de orden 3: {"1->2","2->5","5->1"}

Contiene una ruta elemental que va del nodo "2" al nodo "4": {"2->5","5->1","1->4"}

Todos los nodos poseen un semigrado interior igual a 1. El nodo "4" posee un semigrado exterior nulo. El nodo "1" posee un semigrado exterior igual a 3 y un grado total de 4. Encontramos que:

$$d_G("1") = d_G^-("1") + d_G^+("1") = 1 + 3 = 4$$

Opciones gráficas de igraph

El siguiente código muestra las opciones gráficas principales de un grafo **igraph**. El grafo contiene nodos con nombres completos.

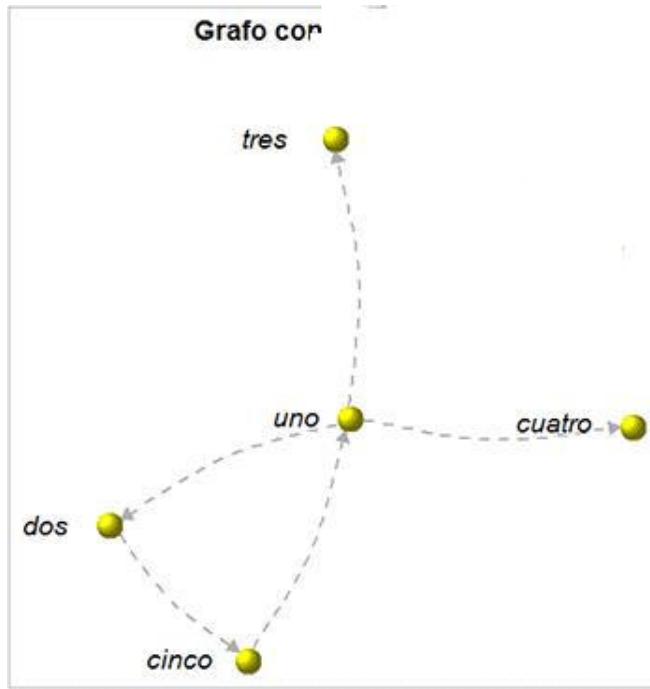
```
set.seed(1)
G<- graph.formula(uno->dos, # directed G
                  uno->tres,
                  uno->cuatro,
                  dos->cinco,
                  cinco->uno)
```

```

V(G)                      # vertices (nodos)
E(G)                      # edges (arcos!!)
set.seed(1)
plot(G,
      main= "Grafo con texto y opciones", # título
      vertex.label.dist = 1.5,           # desplaza las etiquetas
      vertex.color = "yellow",
      vertex.frame.color = "blue",     # color de los trazos de los nodos
      vertex.label.color = "black",    # color de las etiquetas
      vertex.label.font = 3,           # tipo de letra
      vertex.label.cex = 1.5,          # tamaño de la letra
      vertex.shape = "sphere",        # forma
      vertex.label.degree = - pi,     # etiqueta a la izquierda
      edge.arrow.size = 0.1,          # tamaño de puntas de flechas
      edge.arrow.mode = 2,            # flecha normal
      edge.curved = -0.2,            # curva de las aristas
      edge.lty = 2,                  # tipo de línea
      edge.width = 2,                # grosor de la línea
      edge.arrow.width = 2,           # grosor de la punta de flecha
      margin=0.1)                   # margen del cuadro

```

Es posible configurar estas distintas opciones a nivel del propio grafo, a veces a nivel de cada nodo o arco, o como valor por defecto.



Grafo transformado

El siguiente comando permite obtener ayuda sobre las opciones gráficas del paquete.

```
help(igraph.plotting)
```

Es posible guardar un gráfico, y también la definición de un grafo, para utilizarlo en aplicaciones externas (o para

recuperarlo más adelante en nuestro entorno de trabajo).

```
png(filename="mi_grafo.png",
     height=800,
     width=600)           # abre dispositivo de salida png
plot(G)
dev.off()              # preste atención, ¡hay que CERRARLO!

          # exportar un grafo
write.graph(G, file=mi_grafo.dl', format="pajek")
          # importar un grafo
G_ <- read.graph(file='mi_grafo.dl', format="pajek")
plot(G_)
```

Este código nos permite abrir un dispositivo dedicado al formato de una imagen **.png** y, a continuación, «imprime» sobre esta salida virtual que construye un archivo. No hay que olvidarse de cerrar la salida virtual. Esta técnica **funciona para todos los gráficos** que aparecen en la zona «plot» de RStudio.

El código guarda también el grafo en un formato de archivo compatible con la aplicación **Pajek** y lee el archivo para demostrar que el grafo se ha guardado correctamente. Existen muchos otros formatos disponibles, entre ellos graphML, que es una representación XML compartida entre varias aplicaciones, como **Gephi**.

Recorrido y explotación trivial del grafo

Es posible calcular el conjunto de vecinos.

```
neighbors(G,5, mode ="total") # vecinos
```

```
+ 2/5 vertices, named:
[1] uno    dos
```

```
neighbors(G,5, mode ="in")   # vecinos antes
neighbors(G,5, mode ="out")  # vecinos después
```

```
+ 1/5 vertex, named:
[1] dos
```

```
+ 1/5 vertex, named:
[1] uno
```

En el grafo anterior, se comprueba efectivamente que estos nodos preceden o siguen al nodo "5".

Con la misma idea, el siguiente código extrae los arcos incidentes.

```
incident(G,5, mode = "total") # edges incidentes
incident(G,5, mode = "in")    # edges entrantes
incident(G,5, mode = "out")   # edges salientes
```

```
+ 2/5 edges (vertex names):
[1] cinco->uno    dos->cinco

+ 1/5 edge (vertex names):
[1] dos->cinco

+ 1/5 edge (vertex names):
[1] cinco->uno
```

Se obtiene el mismo resultado con los operadores dedicados, propios del paquete:

E(G)[inc(5)]	# arcos del nodo 5
E(G)[from(5)]	# arcos desde el nodo 5
E(G)[to(5)]	# arcos hacia el nodo 5

Conviene observar atentamente el resultado del siguiente código.

are_adjacent(G, 1, 2)	# adyacentes orientados
are_adjacent(G, 2, 1)	# adyacentes orientados

```
[1] TRUE
[1] FALSE
```

Estos resultados ponen de relieve que la noción de adyacencia en un grafo orientado tiene en cuenta la orientación de los arcos.

Obtengamos la matriz de adyacencia del grafo.

A <- get.adjacency(G) # matriz de adyacencia
A # sparse matrix

```
5 x 5 sparse matrix of class "dgCMatrix"
      uno  dos  tres  cuatro  cinco
uno    .    1    1    1    .
dos    .    .    .    .    1
tres   .    .    .    .    .
cuatro  .    .    .    .    .
cinco   1    .    .    .    .
```

Cada arco está representado por un **1**. Como no es la matriz de un grafo no orientado, la matriz no es simétrica. Como no hay ningún bucle, los elementos de la diagonal no valen **1**.

Pero lo que resulta más sorprendente es que los demás valores de la matriz no parecen representar **0**. De hecho, este tipo de matriz se denomina *sparse matrix*, es decir, una matriz dispersa, gestionada inteligentemente de tal manera que ciertas intersecciones pueden estar vacías para no consumir memoria.

Este mecanismo es muy potente, pues un grafo con un grado intermedio no demasiado elevado que contenga muchos nodos se codificará de una forma muy compacta. Esto ocurre a menudo con los grafos de redes sociales, donde un millón de personas dialogan, ipero no todas entre sí!

Con este tipo de sparse matrix, hasta un simple PC puede manipular grandes grafos. El paquete **igraph** se basa en el paquete **Matrix**, que le provee las sparse matrix. Este paquete dispone de algoritmos y de representaciones en memoria de matrices específicas, matrices dispersas o, por el contrario, matrices densas.

El paquete permite crear un grafo a partir de su matriz de adyacencia, lo que permite realizar procesamientos matriciales pesados sobre grafos con muchísima simplicidad.

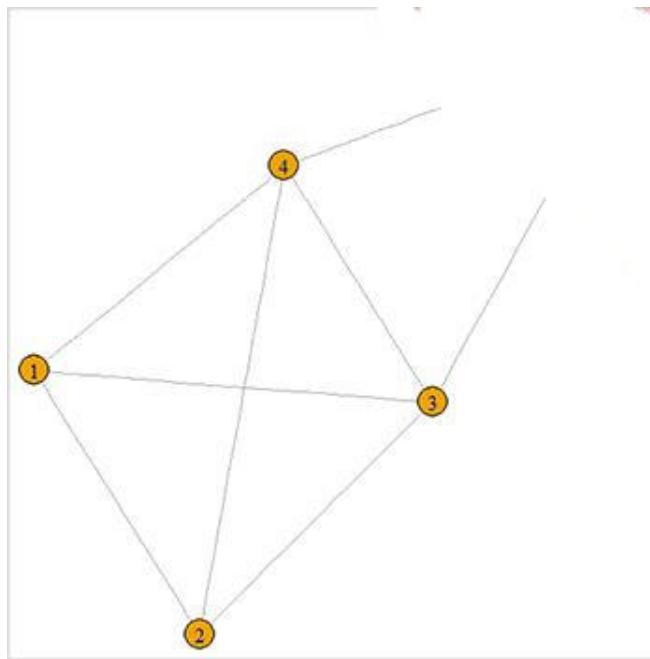
El paquete **igraph** contiene muchos generadores de grafos.

💡 El siguiente grafo que se parece a una casa con una cruz es un «clásico», pues, a diferencia de lo que es posible hacer con la casa sin cruz, podemos recorrer todo el grafo con un lápiz sin levantararlo del papel y sin volver a pasar por el mismo sitio dos veces.

```
# manipulaciones básicas
set.seed(15)          # semilla aleatoria
G <- make_graph("HouseX") # casa con cruz
vcount(G)              # número de nodos
ecount(G)              # número de aristas
set.seed(15);plot(G)
```

```
[1] 5   [ 5 nodos]
[1] 8   [ 8 aristas]
```

Para obtener dos veces seguidas el mismo grafo, no debemos olvidar inicializar la semilla de números aleatorios.



Casa con una cruz

Gestión de los atributos del grafo

Es posible trabajar sobre los atributos del grafo, así como asignar nombres (etiquetas) a las aristas.

```

vertex_attr(G)           # vacío pues no hay ningún atributo
edge.attributes(G)       # ídem para las aristas

# actualización atributos edge
# para todos los edges
# uso del pipe: %>%
G <- G %>%
  set_edge_attr("label", value = letters[1:ecount(G)])
set.seed(15);plot(G)
edge.attributes(G)
str(G)

```

```

list() [al principio, ningún tributo]
list()

```

```

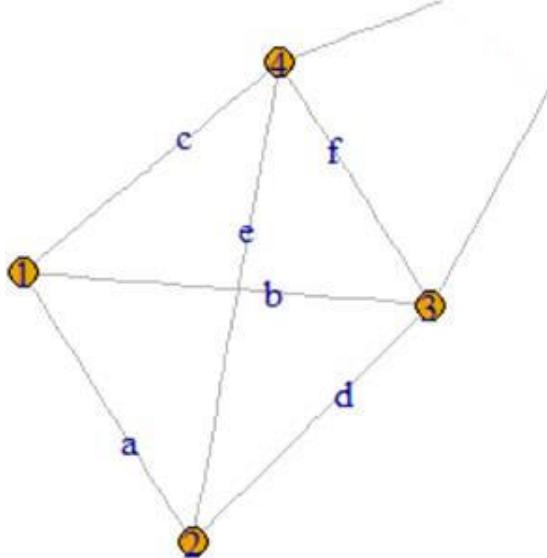
$label [letras para cada arista]
[1] "a" "b" "c" "d" "e" "f" "g" "h"

```

```

IGRAPH U--- 5 8 -- Housex
+ attr: name (g/c), label (e/c)
+ edges:
1 -- 2 3 4      2 -- 1 3 4      3 -- 1 2 4 5      4 -- 1 2 3 5      5 -- 3 4

```



Grafo con etiquetas

También es posible trabajar sobre los atributos nodo a nodo o por arista. Es posible, por ejemplo, cambiar el color de los nodos en función de su naturaleza, o hacer evolucionar su nombre.

Observe atentamente la sintaxis de la parte izquierda de la asignación de los colores en el siguiente código, así como el uso de una función anónima para realizar la transformación de nombre.

```

# manipulación de atributos por

```

```

# asignación y $
V(G)$color <- "green"      # color de los nodos
                            # solo para estos nodos :
V(G)[c(1,3,5)]$color <- "white"
V(G)$size <- 20             # tamaño de los nodos

                            # cambiar las etiquetas
x <- "N_"; y <- sapply(V(G),(function(z) paste0(x,z)))
V(G)$label <- y
vertex_attr(G)              # muestra los atributos
set.seed(15);plot(G)

```

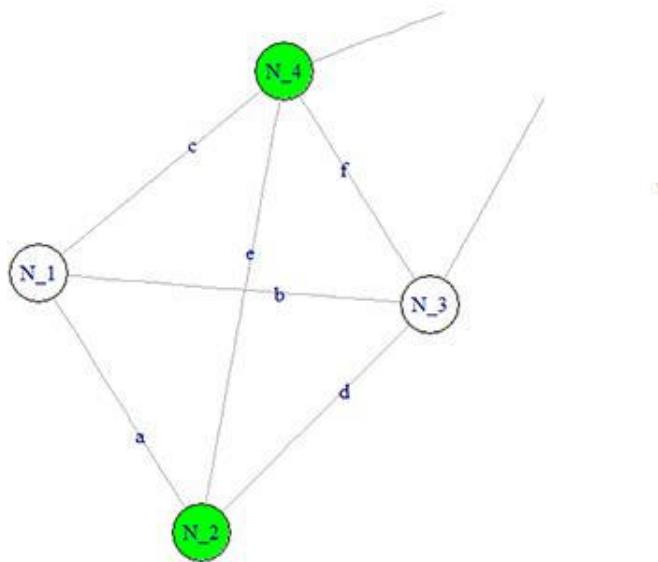
```

$color
[1] "white" "green" "white" "green" "white"

$size
[1] 20 20 20 20 20

$label
[1] "N_1" "N_2" "N_3" "N_4" "N_5"

```



Atributos diferenciados por nodo

La creación y manipulación de atributos no gráficos también es muy importante en el uso de grafos. Mediante los atributos situados sobre los nodos o las aristas, resulta fácil gestionar datos complejos y ricos. La referencia a una imagen, a otro grafo o cualquier nueva feature pueden asociarse fácilmente a los nodos y a las aristas. Vamos a realizar una pequeña manipulación simplista para comprender hasta qué punto resulta trivial agregar un atributo e identificar los nodos correspondientes a un valor de este atributo.

```

# creación de un nuevo atributo
V(G)[c(1,3,5)]$nuevo_atributo <- "top"
V(G)$nuevo_atributo          # contiene NA
                            # buscar según una condición
top <- which(V(G)$nuevo_atributo == "top")

```

Observe la forma de asignar un valor al nuevo atributo que no existía previamente, sin tener que asignar un valor sobre todos los nodos, y a continuación el uso de **which()** para extraer los nodos correspondientes a un valor del atributo.

```
# complementarios de top
no_top <- setdiff(1:vcount(G),top)
no_top
# actualización de los complementarios
V(G)[no_top]$nuevo_atributo <- "no_top"
V(G)$nuevo_atributo

V(G)[]] # descripción exhaustiva de los nodos
E(G)[]] # descripción exhaustiva de las aristas
```

```
[1] 2 4

[1] "top"   "no_top" "top"   "no_top" "top"

+ 5/5 vertices:
  color size label nuevo_atributo
1 white  20    N_1          top
2 green   20    N_2         no_top
3 white  20    N_3          top
4 green   20    N_4         no_top
5 white  20    N_5          top

+ 8/8 edges:
  tail head tid hid label
1     2     1     2     1      a
2     3     1     3     1      b
3     4     1     4     1      c
4     3     2     3     2      d
5     4     2     4     2      e
6     4     3     4     3      f
7     5     3     5     3      g
8     5     4     5     4      h
```

Distribuciones visuales, layout

La disposición de nodos en el plano puede controlarse mediante la opción **layout**, que invoca a los algoritmos clásicos de presentación de grafos.

Esto no es anecdótico: la presentación de grafos afecta, por desgracia, a la interpretación que se realiza de ellos.

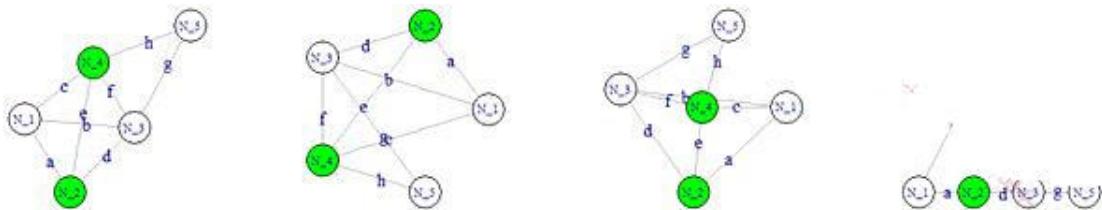
```
# diferentes aspectos del mismo grafo
op <- par()                      # almacena el anterior valor de disposición
par(mfrow = c(1,4))                #
# diferentes aspectos del mismo grafo
# tamaño de la letra edges
G <- G %>%
  set_edge_attr("label.cex",value = 1.6)  %>%
  set_vertex_attr("label.cex",value = 1)    %>%
```

```

set_vertex_attr("size", value =40)

set.seed(15);plot(G)
set.seed(15);plot(G, layout=layout.circle)
set.seed(15);plot(G,
                  layout=layout.reingold.tilford(G,circular=T))
set.seed(15);plot(G, layout=layout.reingold.tilford) # CUIDADO!
par(op)

```



Cuatro representaciones de un mismo grafo

La segunda representación (circular) pone de manifiesto el hecho de que solo faltan dos aristas para obtener un grafo completo, mientras que las dos últimas representaciones ponen de manifiesto un rol central de N_4 que también habría podido tener N_3 . Vemos hasta qué punto resulta útil, pero también peligroso, comentar el aspecto de un grafo, pues puede inducir representaciones mentales erróneas respecto a la realidad del fenómeno observado.

3. Estructura de los grafos

Se dice que un grafo es **conexo** (*connected*) si siempre es posible encontrar una cadena entre dos nodos diferentes o si solo contiene un nodo. Esta definición también es válida para los grafos orientados, pues nos referimos al grafo no orientado subyacente. Un grafo orientado es fuertemente conexo si siempre existe un camino de ida y un camino de vuelta entre dos nodos del grafo.

Un grafo H es un subgrafo (*sub-graph*) de un grafo G si todos sus componentes (nodos, aristas o arcos) están incluidos en G . En ocasiones hablamos de un «grafo parcial generado por tales nodos o tales aristas». El grafo G es un *supergrafo* de H .

Un grafo es maximal para una propiedad si no existe ningún supergrafo que posea la misma propiedad.

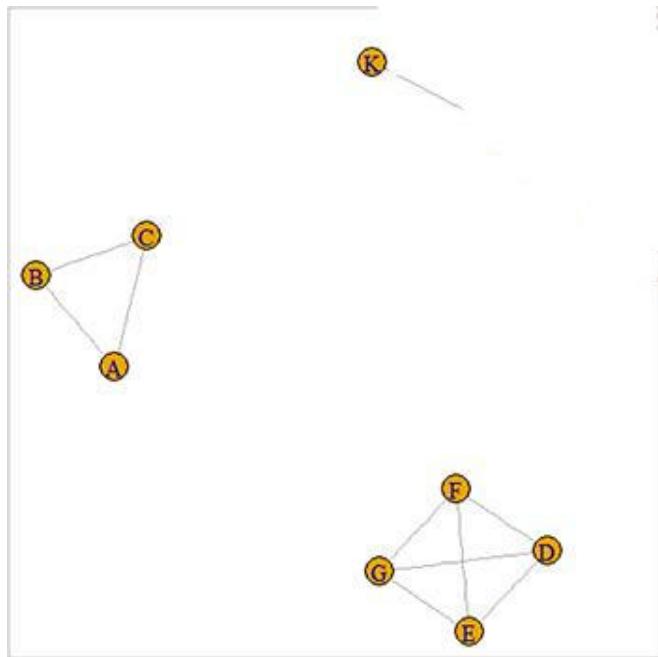
Un componente (*component*) de un grafo es un subgrafo conexo maximal (es decir, maximal para la propiedad de conexidad). Un grafo es p -conexo si posee p componentes (un grafo conexo es 1-conexo). Estos componentes permiten crear una relación de equivalencia cuya partición se corresponde con el grafo. Los p componentes se denominan componentes conexos.

El siguiente código presenta otra manera de producir un grafo con **igraph** y nos permite visualizar un grafo 3-conexo.

```

# un grafo 3-conexo
G <- graph_from_literal( A:B:C -- A:B:C ) +
  graph_from_literal( D:E:F:G -- D:E:F:G ) +
  graph_from_literal( I-- K, I--L, I--M)
plot(G)

```



Grafo 3-conexo

El diámetro de un grafo es la longitud de la mayor de las cadenas más cortas entre dos nodos de un grafo (cuando no existe ninguna cadena posible entre dos nodos, es decir, si los nodos se encuentran en componentes conexos diferentes, se considera que la longitud en cuestión es infinita). En una red social conexa, piense en el diámetro como la «distancia máxima» entre dos actores.

```
diameter(G)
```

```
[1] 2
```

Compruebe este resultado visualmente para convencerse.

Es fácil extraer un subgrafo de un grafo indicando los nodos correspondientes en el subgrafo.

```
# subgrafo a partir de nodos
Ha <- induced.subgraph(G, V(G)[c("D", "E", "F")]) # subgrafo
V(Ha)
E(Ha)
```

```
+ 3/3 vertices, named:
[1] D E F
```

```
+ 3/3 edges (vertex names):
[1] D--E D--F E--F
```

Grafos y redes (sociales)

Las nociones que vamos a abordar a continuación no son específicas del estudio de las redes sociales, aunque se utilizan ampliamente en este marco.

Una red social está compuesta por actores (humanos, grupos de humanos, personas morales) que están relacionadas por vínculos que denominamos vínculos sociales, y son típicamente interacciones, síncronas o no.

Evidentemente, es habitual representar los actores mediante nodos y las relaciones mediante aristas o arcos, lo que denominamos en ocasiones sociograma, cuando las relaciones suponen directa o indirectamente conceptos de afinidad entre los actores (quién ha sido escogido por quién, quién es líder...).

Las relaciones pueden ser estructurales (empleado por..., compañero de...), factuales (dialoga con..., participa en... con...), o declarativas (like..., le gusta...). A menudo estas relaciones se extienden a los objetos que rodean a los actores: «a fulanito le gusta tal película».

Cuando las relaciones son simétricas, como el hecho de estar vinculado en LinkedIn, se obtienen grafos no orientados. Cuando los grafos representan la expresión de opiniones (sobre alguien o algo) o cuando representan suscripciones a servicios o envíos de tweets o de correos electrónicos, esto da lugar a grafos orientados.

Si la relación está cuantificada, como el número de tweets enviados a alguien, el número de estrellas que indican si nos ha gustado algo, el número de lugares comunes, de smileys... se produce un grafo evaluado (que incluye pesos, *weights* en inglés).

Los tipos de relaciones están etiquetadas en los propios arcos o en las aristas (colega, follower...).

1. Análisis de las redes sociales: conceptos básicos

Las nociones más comúnmente utilizadas en el análisis de las redes sociales son la densidad de los vínculos de la red (*density of links*) y la centralidad (*centrality*).

La **densidad de los vínculos** representa simplemente el número de vínculos reales frente al número de vínculos posibles. Se vislumbra cuando ciertas partes del grafo contienen numerosos vínculos (se habla a menudo de comunidades), y los vínculos entre dichas comunidades son relativamente escasos. Es una manera habitual de definir los clústeres de un grafo.

La **centralidad** es una noción más sutil que trata de poner de manifiesto los actores más «importantes» de la red. Evidentemente, la noción de importancia es bastante difusa y, según sus objetivos, podemos definir diferentes nociones para modelarla.

La interpretación de esta centralidad depende totalmente de la semántica de las relaciones expresadas (aristas o actos): por ejemplo, analizar las relaciones de transferencia de información entre personas que interactúan con interlocutores habituales o analizar cómo se organiza la opinión de distintos consumidores sobre diversos hoteles no presentan el mismo sentido en términos de centralidad.

El propio grafo, o uno de sus subgrafos, puede estar caracterizado por su nivel de centralización.

Para construir features, por ejemplo para utilizarlas en machine learning, existen varias estrategias, típicamente:

- Fabricar las features calculadas directamente a partir de las características de los nodos (o de las aristas/arcos).
- Asignar features a los nodos en función de su pertenencia a una parte del grafo (por ejemplo: «pertenencia al subgrafo de personas interesadas por...») haciéndolas heredar de las características de este subgrafo, como su nivel de centralidad. Por ejemplo: «pertenencia a un grupo muy influenciado por un líder, nivel de influencia normalizado

del líder = 10.45».

- Asignar la feature al nodo como si fuera su «posición» en el subgrafo (por ejemplo: «influir sobre las compras de un libro de música» o «nivel de influencia sobre su grupo = 1.7»).

Centralidad

La centralidad se declara en función de las necesidades de cada uno; citemos por ejemplo los tres tipos de medida de centralidad más simples:

- La **centralidad respecto a un grado** (*degree centrality*), que calcula los nodos que poseen un alto $d_G(v)$ o $d_G^-(v)$ o $d_G^+(v)$. Los nodos se consideran importantes si poseen fuertes interacciones directas con los demás nodos. Cuando $d_G^-(v)$ es importante, es decir, cuando posee muchos arcos entrantes, hablamos en ocasiones de «prestigio» y cuando $d_G^+(v)$ es importante hablamos en ocasiones de «gregarismo». Con la misma idea, también podríamos evaluar la interacción de los nodos del conjunto de una comunidad con otras comunidades, para modelar aquellos casos en los que el «líder» de una comunidad ejerce su influencia a través de los miembros de su comunidad, sin ejercerla de manera directa sobre el exterior de la comunidad. Tratamos, evidentemente, de normalizar este valor en función de los grados de los demás nodos o del grado del nodo que posee un grado máximo. Un grafo o un subgrafo puede obtener un grado de centralización. Como punto de referencia sobre el grado de centralización de un grafo, basta con recordar que este es **máximo para un grafo en estrella**. Este tipo de medida responde a menudo a la cuestión de saber cuáles son las personas que acceden a muchas otras personas.
- La **centralidad respecto a la proximidad** (*closeness centrality*), que utiliza la longitud de la cadena más corta (o ruta, introduciendo o no las longitudes sobre las aristas o los arcos) para definir una distancia. La centralidad respecto a la proximidad de un nodo es, en este caso, la inversa de una función como la suma de las distancias de este nodo respecto a los demás (o como una suma de inversos). Podemos introducir transformaciones de las distancias (como 2^d), ponderaciones, convenciones (como declarar infinita la distancia entre nodos no conexos). Como una primera aproximación, podemos considerar que este tipo de medida de centralidad muestra si un nodo puede **alcanzarse o no fácilmente** por los demás.
- La **centralidad respecto al intermediario** (*betweenness centrality*), que define el número de veces que un nodo (o una arista, o un arco) sirve de intermediario (*bridge*). A menudo basamos esta medida en el número de veces en las que un nodo figura en la cadena más corta o la ruta más corta entre los demás nodos del grafo. Si se suprime de una red los nodos o aristas que poseen una fuerte *betweenness centrality*, se aísla gráficamente aquellos subgrafos más o menos conexos del grafo. Podemos percibir los nodos que poseen una fuerte *betweenness centrality* como intermediarios, brokers de información.

Existen otras **medidas de centralidad más sofisticadas**, pero todas comparten al menos una de las dos siguientes afirmaciones:

- La importancia de un nodo puede serle atribuida por otros nodos (por ejemplo, para la búsqueda de personas bien conectadas a otras personas bien conectadas).
- Las características y las transformaciones de la **matriz de adyacencia** de un grafo permiten implementar medidas teniendo en cuenta el conjunto del grafo o de todas sus **geodésicas**. O, por ejemplo, imaginemos calcular medidas tras haber realizado una diagonalización de esta matriz a partir de una base de vectores propios (*eigenvalues*), o medidas que utilizan la traza como una norma. El célebre algoritmo de Google (*PageRank*) forma parte de estas técnicas.

No dude en consultar los textos de los principales autores, como Freeman, Bonacich (*Bonacich centrality*), Seeley y Katz (*Katz prestige*) para profundizar.

¿Cómo escoger una medida?

Evidentemente, hay que empezar identificando lo que se busca determinar a partir de un grafo y emitir diversas hipótesis acerca de qué podría ser o no importante.

Partiendo de ello, uno de los métodos más fáciles de implementar consiste en identificar manualmente sobre el grafo aquellos nodos (o aristas) conocidos por su importancia respecto al tema correspondiente, y a continuación aquellos nodos manifiestamente no importantes y, llegado el caso, los nodos medianamente importantes.

Dispondremos entonces de un conjunto de test que nos permitirá trabajar sobre matrices de confusión. La idea es la siguiente: vamos a probar diversas medidas y a dotarnos de diversos umbrales para estas medidas (*si medida_x > umbral_1_x entonces clase_x = importante*). Para cada modelo de medida, es decir, una pareja formada por una medida y sus umbrales, resulta fácil trabajar sobre una matriz de confusión aplicando la medida y los umbrales a los nodos sobre los que se ha construido una opinión manualmente.

Todas las parejas que presenten un buen rendimiento en al **menos una de las casillas de su matriz de confusión son candidatas** en términos de medida. Por buen rendimiento podemos entender: poseer uno de los valores diagonales elevado o poseer uno de los valores no situados en la diagonal que sea bajo.

En un mundo ideal, al final de este proceso, se tendría que haber seleccionado un panel de medidas en el que todas las casillas de la matriz de confusión hubieran funcionado al menos una vez en alguna de las parejas (medida, umbral).

2. Puesta en práctica

Tras habernos dotado de un nuevo grafo simple pero con bastantes nodos como para demostrar nuestras medidas, vamos a calcular su diámetro y a continuación calcularemos una medida de centralidad para cada uno de los cuatro tipos de medidas de centralidad explicadas previamente. El conjunto se almacena en un pequeño **data.frame**.

```
set.seed(26); G <- make_graph("Zachary") # grafo conocido
diameter(G) # su diámetro

nodos <- as.vector(V(G)) # etiquetas de los nodos
df <- data.frame( # almacenamiento en data.frame
  nodos,
  degree(G), # grado de los nodos
  closeness(G), # proximidad de los nodos
  betweenness(G), # intermediaridad de los nodos
  eigen_centrality(G)$vector # eigen centralidad nodos
)
names(df) <- c("nodos", # cambio de nombre de las columnas
  "degree",
  "closeness",
  "betweenness",
  "eigen_centrality")
```

[1] 5

El diámetro total de este grafo es igual a 5.

Las primeras filas del **data.frame** son las siguientes.

	nodos	degree	closeness	
1	1	16	0.01724138	
2	2	9	0.01470588	28.4
3	3	10	0.01694915	75.850793
4	4	6	0.01408451	6.2880952

Cuatro medidas de centralidad

Las escalas de estas medidas no son similares. Para representar esto visualmente y trabajar sobre escalas similares, vamos a «normar» los valores de 0 a 20. El valor 20 se corresponde con el diámetro visual máximo de nuestros nodos (es tan solo una opción gráfica!). Para poner de relieve los nodos de fuerte centralidad respecto a cada una de estas medidas, basta con conservar el cuantil más elevado de los valores; aquí nos hemos puesto un límite a 85 % (es decir, solo el 15 % restante). El código incluye un pequeño truco para eliminar los NA que aparecen tras el corte por encima del cuantil seleccionado.

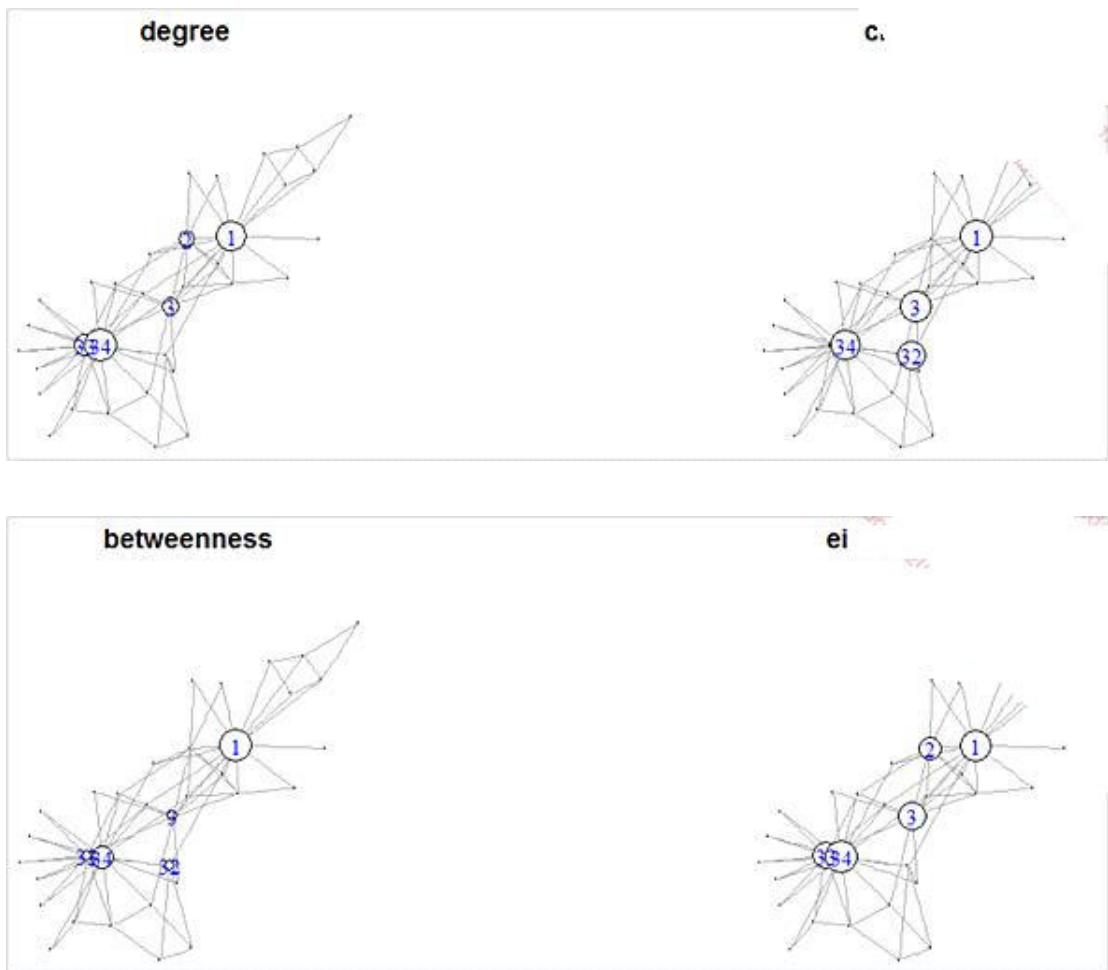
```

op <- par()                      # almacena el anterior valor disposición
par(mfrow = c(1,2))                # dos gráficos por línea
for (j in 2:5) {                   # para los 4 tipos de centralidad
  nodos <- as.vector(V(G))        # etiquetas de los nodos
  c   <- df[,j]                  # normalización de valores
  Mc <- max(c)
  mc <- min(c)
  c   <- 20 * (c - mc) /abs(Mc-mc)    # escalado

  w   <- cut(c,                  # guardamos el 15 % de los nodos
             breaks = quantile(c, probs =c(0.1,0.85,1)),
             labels = FALSE)
  w   <- ifelse(is.na(w),1,w)    # eliminación de NA
  c   <- ifelse(w == 2, c,0)      # tamaño nodos: 0 o proporción
  nodos[which(c == 0)]<-NA       # eliminación pequeños valores
  set.seed(16);
  plot(G,
        main = names(df)[j],      # título = nombre de la medida de c
        layout=layout.fruchterman.reingold,# organización visual
        vertex.size = c,           # tamaño según nodo
        vertex.label=nodos,        # etiqueta de nodos
        vertex.label.color= "blue",# color etiquetas
        vertex.color = "white",    # color círculos
        edge.width=1)              # grosor línea
  }
  par(op)                         # restauración de parámetros
}

```

Lo que nos permite comparar las medidas de centralidad mediante el tamaño de los nodos más centrales para cada uno.



Nodos destacables en función de la medida de centralidad

Observamos que los nodos escogidos por **degree** y **eigen_centrality** son idénticos y que los nodos escogidos por **closeness** se encuentran entre los escogidos por **betweenness**. Por otro lado, los nodos 1, 3 y 34 son escogidos por las cuatro medidas. En un grafo social simple, las cuatro medidas son coherentes, mientras que sus subyacentes algorítmicos son diferentes. La noción de centralidad implícita en ellos es, por lo tanto, más genérica de lo que podría indicar lo variado de sus definiciones. Evidentemente, para ciertos grafos más complejos, las medidas expresan nociones más diferenciadas, sin que por ello nadie se haya tomado la molestia de codificar las cuatro en un mismo paquete!

3. Detección de comunidades

El *partitioning & clustering* de los grafos es uno de los retos clave en las redes sociales. «¿Quién es similar a quién?» es la cuestión clave que se ha de resolver.

El sentido de la similaridad de los nodos de un grafo solo puede definirse a través del conocimiento de la semántica implícita en las relaciones del grafo. En el resto del párrafo, las nociones de similaridad y de comunidad se referirán a aspectos más genéricos, basados únicamente en la topología de la comunidad y en la ubicación de un nodo dentro de esta comunidad.

En 2005 los Srs. Pascal Pons y Matthieu Latapy presentaron un algoritmo llamado Walktrap que utilizaremos aquí. Encontrará su descripción, clara y agradable de leer, en su artículo de investigación correspondiente, accesible en el excelente sitio web arXiv.org.

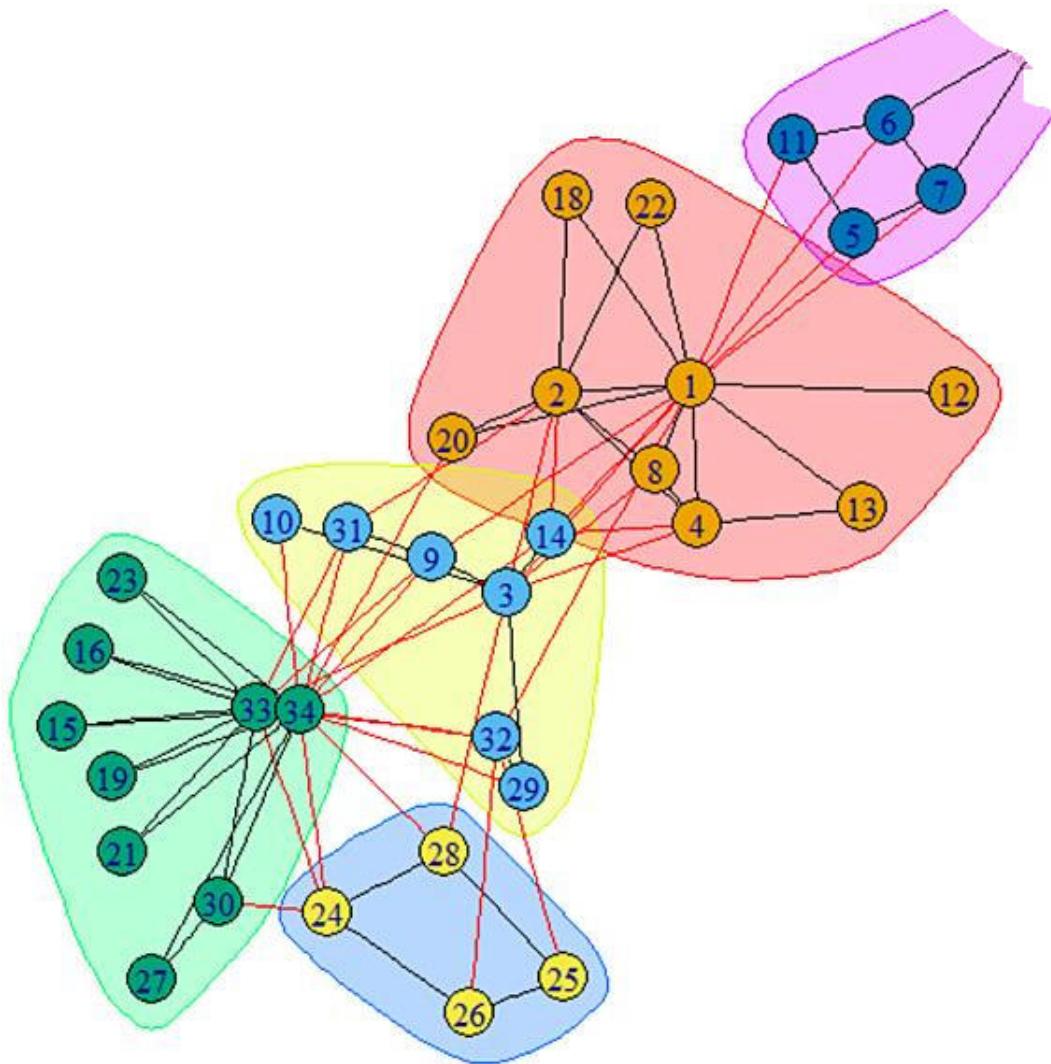
Este algoritmo se basa en un paso aleatorio dentro del grafo y en una medida de similaridad (una distancia) que se

basa en el «conocimiento» de un nodo por parte de otro, es decir, la probabilidad de alcanzar un nodo a partir de un paso aleatorio a partir de otro nodo. Este algoritmo presenta un excelente rendimiento y está implementado en **igraph**.

💡 Veremos con más detalle la noción de paso aleatorio cuando abordemos las series temporales.

Para invocar Walktrap, la sintaxis es trivial.

```
# determinación de comunidades  
CG_1 <- cluster_walktrap(G)  
set.seed(16);plot(CG_1, G)
```



Este algoritmo deduce el número de comunidades en función de la topología de la red, a diferencia de k-means, que habíamos estudiado antes en el marco de la clusterización de conjuntos de observaciones.

Podemos destacar una cierta coherencia entre las comunidades y la elección de los nodos 1, 2, 3, 32, 33 y 34 como poseedores de cierta centralidad (consulte la sección anterior). Para convencerse de ello, intente eliminar

mentalmente los vínculos intercomunidades que se basan en estos nodos: comprobará que las comunidades se encuentran ahora prácticamente aisladas unas de las otras.

Veamos los diámetros de los diferentes clústeres. Tendremos también la ocasión de aprender a aislarlos extrayendo sus miembros.

```
num_cluster <- max(membership(CG_1)) # número de clústeres
for (i in 1:num_cluster){           # para cada clúster
  l <- V(G)[membership(CG_1) == i]  # lista de sus nodos
  H <- induced.subgraph(G,l)        # el clúster!
  d <- diameter(H)                # su diámetro
  cat("Clúster",i,"diámetro",d,"con los nodos",l, "\n")
}
```

```
Clúster 1 diámetro 2 con los nodos 1 2 4 8 12 13 18 20 22
Clúster 2 diámetro 4 con los nodos 3 9 10 14 29 31 32
Clúster 3 diámetro 2 con los nodos 15 16 19 21 23 27 30 33 34
Clúster 4 diámetro 2 con los nodos 24 25 26 28
Clúster 5 diámetro 2 con los nodos 5 6 7 11 17
```

Algunas observaciones empíricas:

El clúster 2 se caracteriza por su gran diámetro igual a 4, que puede compararse con el diámetro del conjunto del grafo, que sería igual a 5.

El clúster 1 se caracteriza por la presencia del nodo 1, que posee una gran centralidad y que es el único bridge con el clúster 5, que no posee ningún nodo con una gran centralidad. El nodo 2 posee también un rol delegado interesante en este clúster.

Como ocurre con el clúster 5, el clúster 4 es periférico. Los nodos 3 y 32 del clúster 2 son los bridges de su conexión más corta al otro «lado» del grafo. Si no, habría que «pasar» a través del clúster 3.

El clúster 3 está casi totalmente representado por los nodos 33 y 34, el nodo 33 tiene vocación de delegación del nodo 34, que asegura más conexiones con el exterior.

Vamos a comparar estos resultados con los de otros dos algoritmos con menos de 10 años de edad (*cluster_spinglass*, *cluster_leading_eigen*).

El primero utiliza una analogía energética y trata de encontrar una configuración de clústeres que minimiza la energía de la red imaginando que esta red (nuestro grafo) pudiera asemejarse a un *spinglass*. Un spinglass es una aleación que contiene impurezas magnéticas, y es también el paradigma de referencia de una cierta manera de construir un entorno desordenado en el que ciertos elementos poseen acoplamientos con otros elementos más o menos alejados (lo que representan nuestras aristas).

El segundo se basa en la diagonalización de matrices simétricas (no está, sin duda, adaptado para trabajar con grafos orientados).

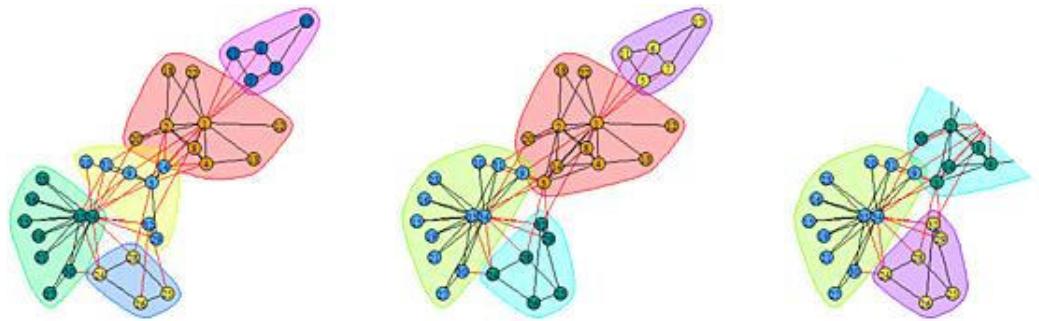
Puede accederse a las referencias científicas de ambos algoritmos mediante la ayuda de estos en **igraph**.

```
CG_2 <- cluster_spinglass(G)
CG_3 <- cluster_leading_eigen(G)
op <- par()          # almacena el anterior valor de la disposición
```

```

par(mfrow = c(1,3))    # tres gráficos por línea
set.seed(16);plot(CG_1, G)
set.seed(16);plot(CG_2, G)
set.seed(16);plot(CG_3, G)
par(op)                 # restauración de los parámetros

```



Comparación Walktrap, SpinGlass, LeadingEigen

Nuestros dos nuevos algoritmos han encontrado un número inferior de comunidades (una comunidad menos). Los roles de los nodos «centrales» siguen poniéndose de relieve. De hecho, el único punto que difiere entre SpinGlass y LeadingEigen, es la interpretación de la gran centralidad del nodo 1, que en LeadingEigen extiende la comunidad 17-6-7-11-5 (que también se había identificado como tal en Walktrap) al nodo 1 y de manera consecuente al nodo 12.

Tratemos de verlo con mayor claridad. Vamos a objetivar las diferencias entre los resultados midiendo la información mutua entre los modelos y las diferencias en el etiquetado. Los resultados se almacenarán en un pequeño **data.frame**.

```

t1 <- c(                      # comparación de información mutua
        compare(CG_1, CG_2, method="nmi"),
        compare(CG_1, CG_3, method="nmi"),
        compare(CG_2, CG_3, method="nmi"),
        compare(CG_1, CG_1, method="nmi") # truco:max
      )

t2 <- c(                      # comparación contenido
        compare(membership(CG_1), membership(CG_2)),
        compare(membership(CG_1), membership(CG_3)),
        compare(membership(CG_2), membership(CG_3)),
        compare(membership(CG_1), membership(CG_1)) # truco:min
      )

t0 <- c("información mutua","nodos diferentes")
df <- data.frame(cbind(t0, rbind(round(t1,2),round(t2,2))))
names(df) <- c("Indicador",
               "Walk-Spin",
               "Walk-Eign",
               "Spin-Eign",
               "referencia 100% similar")
row.names(df)<-NULL

```

	Indicador	Walk-Spin	Walk-Egin	Spin-Egin	ref.
1	información mutua	0.76	0.68	0.9	1
2	nodos diferentes	0.69	0.94	0.28	0

Cohesión de los resultados de los tres algoritmos

La información mutua entre SpinGlass y LeadingEigen es próxima a 1 y el indicador que expresa el número de nodos diferentes es mínimo, ambos algoritmos producen resultados similares. Por otro lado, la diferencia entre WalkTrap y LeadingEigen es la diferencia máxima de la tabla, de modo que si buscáramos el mejor consenso habría que seleccionar la clusterización propuesta por el algoritmo SpinGlass.

Para mejorar este análisis, vamos a ver la calidad de la separación aportada por cada uno de los tres algoritmos.

```
modularity(CG_1) # calidad de la separación
modularity(CG_2)
modularity(CG_3)
```

```
[1] 0.3532216
[1] 0.4197896
[1] 0.3934089
```

El segundo algoritmo (SpinGlass) aporta la mejor separación (0.42).

Ambos puntos de vista son concordantes, de modo que resultaría normal escoger este algoritmo.

A pesar de ello, vamos a adoptar un último punto de vista: la búsqueda de cliques en el grafo.

En inglés, una de las definiciones de la palabra **clique** es: «grupo de personas que se asocian y no dejan a otros entrar». En castellano diríamos «pandilla» o «banda». Resulta algo peyorativo; podríamos decir, por ejemplo: «el señor fulanito y su clique (pandilla) son unos estafadores». En la teoría de grafos, un clique es un subgrafo completo de un grafo no orientado (según la notación vista al inicio de este capítulo: un grafo K_n). Un clique de n nodos posee $n(n - 1)/2$ aristas.

El pequeño código siguiente muestra los 8 primeros K_n . Observe cómo se crea un grafo a partir de su matriz de adyacencia, que puede ser de tipo **matrix** o **Matrix**.

```
op <- par()                      # almacena el antiguo valor
par(mfrow = c(2,4),pty = "s")    # 2 filas, 4 columnas, square
for (i in 1:8) {                  # ocho  $K_i$ 
  a_G <- -diag(1,i)+1           # matriz de 1 salvo la diagonal
                                # grafo a partir de la matriz
  K <- graph_from_adjacency_matrix(a_G,mode = "undirected")
                                # estética
  t <- paste0("K",i," , con num edge(s) = ",i*(i-1)/2)
  t <- ifelse(i == 1, paste('Singleton: ',t),t)
  t <- ifelse(i == 2, paste('Recto: ',t),t)
  t <- ifelse(i == 3, paste('Triángulo: ',t),t)
  s <- ifelse(i < 3 , 30,100/i)
  plot(K, main = t,
       vertex.size = s,
```

```

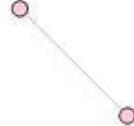
    vertex.label = NA,
    vertex.color = "pink",
    layout=layout.circle)
}
par(op)                                # restauración de parámetros

```

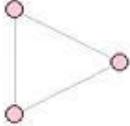
(Singleton:) K1, con num edge(s) = 0



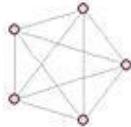
(Recto:) K2, con num edge(s) = 1



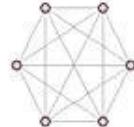
(Triángulo:) K3, con num edge(s) = 3



K5, con num edge(s) = 10



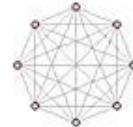
K6, con num edge(s) = 15



K7, con num edge(s) = 21



K8, con num edge(s) = 28



Los ocho primeros K_n

Existen muchas maneras de utilizar esta noción en el análisis de grafos. Conviene tener en mente que la búsqueda de cliques en grandes grafos en ocasiones resulta costosa en términos de rendimiento.

Nos interesaremos por cliques de tamaño al menos igual a 4, que en primer lugar contaremos. A continuación, contaremos el número de cliques de tamaño al menos igual a 4 y que sean «maximales», es decir, que no estén contenidos en otros cliques. Por último, buscaremos atentamente la composición de los cliques más extensos (extensos en cuanto al número de nodos, no por su diámetro, pues un clique no degenerado posee un **diámetro igual a 1**).

```

c <- cliques(G, min = 4)      # cliques de tamaño al menos igual a 4
length(c)
c <- max_cliques(G, min = 4) # ídem pero cliques maximales
length(c)
largest_cliques(G)           # el clique o los cliques más grandes

```

```

[1] 13
[1] 4

[[1]]
+ 5/34 vertices:
[1] 8 1 2 3 4

[[2]]
+ 5/34 vertices:
[1] 4 1 2 3 14

```

Podría interesarnos trabajar con detalle sobre los 13 cliques de tamaño superior a 4 y observar cómo se reparten dentro de los 4 cliques maximales. Vamos a contentarnos con observar la composición de los dos cliques más grandes.

Son dos K_5 que poseen cuatro nodos comunes. Comparando el conjunto de sus nodos (1, 2, 3, 4, 8, 14) con el de

nuestras potenciales comunidades, confirmamos que solo el algoritmo SpinGlass ha creado una comunidad que engloba estos dos cliques. Es un último motivo para favorecer este algoritmo debido a la selección de la partición de comunidades.

 Hasta ahora habíamos tratado de dividir las comunidades de tal manera que no existía ningún elemento en común. A partir de estas divisiones y de la identificación de cliques o de otras estructuras de relación que se corresponden con la naturaleza del problema, en ocasiones resulta conveniente mezclar estos resultados para crear comunidades que posean un cierto recubrimiento, pero con una gran coherencia. También es posible descubrir una subcomunidad dentro de una comunidad. En ocasiones, tras haber despejado un poco el trabajo con un algoritmo rápido sobre un grafo muy grande, resulta conveniente utilizar otros algoritmos por comunidad.

La teoría de grafos está muy extendida; la búsqueda en grafos sociales está en plena expansión. En las siguientes etapas de sus trabajos, no dude en dedicar algo de tiempo estudiando, por ejemplo, las nociones de small world y de grafo «gigante», muy de moda. Las herramientas vistas en este capítulo le permitirán realizar numerosos proyectos interesantes y de actualidad. Trabajando los grafos a este nivel, verá cómo evolucionan los requisitos cuando quiera ir más allá en su estudio, y llegará el momento de ir más lejos: dedique algo de tiempo a practicar y asimilar los conceptos básicos de las redes antes de iniciarse en teorías demasiado sofisticadas.

Otros problemas, otras soluciones

Series temporales

Cuando las observaciones evolucionan en el tiempo, las modelamos mediante una serie temporal y comprobamos si su secuencialidad influye en los valores de las observaciones.

1. Introducción

Las series temporales tienen como variable subyacente una variable temporal. Podemos considerarlas como la extensión de una lista de observaciones indexadas por una variable temporal. En el límite, si los intervalos de tiempo tienden a 0, la variable tiempo que era discreta puede definirse como continua. En este caso, se considera la serie discreta correspondiente como una serie de observaciones posibles de la serie continua, que a menudo representa un sistema físico (una oscilación, una señal electrónica o biológica o, por extensión, el valor de un índice bursátil...).

El estudio de las series temporales se basa en que a priori cada valor de la serie depende de otros valores de la serie, a menudo en los valores precedentes. En este último caso, se intenta expresar naturalmente esta dependencia mediante una función. Esta función puede percibirse como una **función generatriz** de la serie que se aplica sobre un conjunto de primeros elementos.

En este caso, es comprensible que se denomine a estas series «**procesos**».

Un **proceso estocástico indexado sobre un conjunto T** es una colección de variables estocásticas $(X_t)_{t \in T}$ definidas sobre un mismo espacio probabilístico.

En la práctica cotidiana, t es un índice de tiempo y T es igual a enteros, a enteros positivos, a valores reales, a valores reales positivos o incluso a una potencia de \mathbb{R} .

Evidentemente, el caso que vamos a abordar más a menudo es que T sea igual a un conjunto de enteros negativos y positivos \mathbb{Z} , lo que expresa a su vez que no se prejuzga la existencia de un inicio del proceso. En la práctica, se restringe T a un conjunto finito de índices I incluidos en el conjunto de valores enteros naturales positivos \mathbb{N}^* . Se realizan, efectivamente, observaciones entre dos momentos y el primero puede indexarse a «1».

$X = (X_t)_{t \in T}$ puede estudiarse, de manera aproximada, basándose en diversos aspectos. ¿Su media móvil es constante o puede modelarse por una función identificable que exprese una tendencia? Respecto a esta media o esta función, ¿los residuos poseen un aspecto particular (como una oscilación respecto a esta media), globalmente el proceso es estacionario?

 Observación para aquellos que descubran el término «serie» y que se pregunten si existe algún vínculo con las series que hayan estudiado en otras ocasiones: utilizaremos el vocablo «serie», pero en matemáticas «clásicas» (es decir, hasta matemáticas superiores) hablaríamos de sucesiones U_n ; el término S_n designa en la enseñanza secundaria las sucesiones cuyos términos son sumas de los n o $n + 1$ primeros términos de una sucesión determinada.

2. Modelo estacionario

Es el concepto fundamental del estudio de las series temporales. Se dice que un proceso es estrictamente estacionario si algunas de sus propiedades son **invariantes tras una traslación** cualquiera realizada sobre el conjunto T de sus «índices».

Esta definición expresa una intuición sencilla: imagine el aspecto de un electrocardiograma; cuando todo va bien, si está en reposo, siempre encuentra las mismas formas con cada pulsación, es decir, una traslación de tiempo igual a la duración entre dos latidos del corazón.

a. Proceso estacionario: los fundamentos

En la mayoría de los casos prácticos, cuando los autores hablan de procesos estacionarios, hacen referencia a los

procesos estacionarios llamados de «segundo orden». No hay equivalencia entre ambas nociones de estacionariedad, salvo por los procesos gaussianos.

Sin entrar en la explicación matemática precisa de estas nociones, conviene recordar que:

- Los procesos de **segundo orden** (no necesariamente estacionarios) poseen una especie de esperanza matemática de su cuadrado **delimitada** (jamás infinita) y que se les suele caracterizar parcialmente mediante dos funciones: la media en función del índice, $m(t)$, y la covarianza de la serie sobre dos índices diferentes, aquí llamada $C(s,t)$.
- Los procesos gaussianos están completamente caracterizados mediante sus $m(t)$ y $C(s,t)$. De este modo, si los conoce y afirma que el proceso es gaussiano, sabrá expresar la formulación precisa de este proceso. Un proceso es gaussiano si cualquier combinación lineal finita del proceso sigue una ley normal. Se deduce que un proceso gaussiano es siempre de segundo orden.
- Un proceso es estacionario de segundo orden si $m(t)$ posee un valor constante y si $C(s,t)$ es una función de la diferencia de los índices $C(s,t) = \gamma(s-t)$. Esta función $\gamma(h)$ se denomina función de autocovarianza (en inglés acvf).
- En el caso de procesos estacionarios de segundo orden, se define una función de autocorrelación (en inglés **ACF**, AutoCorrelation Function) cuya expresión es: $\rho(h) = \frac{\gamma(h)}{\gamma(0)}$. Evidentemente, h representa una desviación entre dos índices.

De manera intuitiva podemos imaginar que si existe cierto grado de autocorrelación en una serie de observaciones del proceso, resulta más sencillo prever la evolución futura del proceso.

Cuando los diversos valores de una serie temporal son independientes, tenemos que hacer frente a lo que llamamos un **ruido**. Este ruido puede tener una estructura global en términos de distribución, es decir, que ciertos valores son más probables que otros, aunque somos incapaces de decir para qué valor de t tendremos un valor determinado de este ruido (incluso de manera aproximada).

Cuando una serie temporal es **i.i.d.** (variables independientes e idénticamente distribuidas = variables aleatorias que poseen todas la misma ley de probabilidad y que son independientes) y esta serie está **centrada**, como la ley normal, decimos que un ruido está centrado.

Cuando el ruido es i.i.d. y está centrado, posee las características de un proceso estacionario de segundo orden y la covarianza entre dos índices de este ruido es nula, se dice que existe un **ruido blanco** de baja intensidad.

En función de los modelos estudiados, se espera que los **errores de predicción** sobre los procesos temporales sean ruidos blancos débiles, o si no ruidos centrados normales, o si no ruidos solamente centrados pero que poseen al menos una covarianza débil o nula.

A menudo, se modelan los procesos mediante una función recursiva que utiliza a su vez los valores previos de la función, pero también mediante la introducción de un ruido para introducir la variabilidad en el modelo. **La naturaleza del ruido de la variable de respuesta no es necesariamente similar al ruido introducido en la fórmula:** ¡no los confunda!

► La función recursiva más conocida genera la serie de Fibonacci: 1 1 2 3 5 8 13 21 34 55 89 144... (es posible empezarla en cero).

► Para $t > 0$, está definida por (observe los dos valores utilizados para inicializar el proceso):

$$X_t = X_{t-1} + X_{t-2}$$

$$X_1 = 1$$

$$X_2 = 1$$

Si a dicha definición de X_t le agrega un ruido blanco, obtiene lo que se denomina un proceso autorregresivo de orden 2. Aquí la palabra «orden» indica el número de valores antecedentes que necesita para definir el valor de un índice determinado.

Gracias a agregar este ruido blanco, obtiene una serie temporal que contiene un factor aleatorio débil y que se parece a un proceso natural.

La fórmula generadora de un proceso autorregresivo de orden 2, que se escribe a menudo como un AR(2), es:

$$X_t = \varphi_1 X_{t-1} + \varphi_2 X_{t-2} + \varphi_0 + \varepsilon_t$$

Los φ son constantes cuyo valor es un número real, y ε_t es un ruido blanco débil.

El siguiente párrafo merece la pena ser estudiado con atención, pues no solo detalla los AR(r), sino también porque la manera de hacer evolucionar la notación de una expresión matemática es muy instructiva en la manera en la que se trabaja en las ramas matemáticas más operacionales como la nuestra. Los que no estén familiarizados con las notaciones matemáticas de ciclo superior tienen aquí la ocasión de familiarizarse con técnicas injustamente aburridas. Esto facilitará en gran medida su comprensión de ciertos textos que pueden parecer, a primera vista, difíciles de interpretar.

b. Proceso autorregresivo AR: ir más lejos

Podemos generalizar la expresión anterior de manera «evidente» para los demás órdenes:

$$X_t = \sum_{s=1}^{s=r} \varphi_s X_{t-s} + \varphi_0 + \varepsilon_t$$

Invocando el orden r (como en «orden»).

La expresión X_{t-s} es un poco delicada de manipular en matemáticas convencionales e impide la «vectorización», como dirían los informáticos, de esta expresión. De modo que se utiliza un operador para indicar: «encontrar el valor de un índice desplazado por detrás de una posición». Esta operación se realiza sobre toda la serie temporal.

Este operador se llama, en inglés, *BackShift operator* o *Lag operator*. En algunos paquetes de R se utiliza la palabra «lag», de modo que vamos a utilizar la expresión «Lag operator» y la llamaremos $L()$. En español, hablamos de **operador de retardo**.

Tenemos: $L(X_t) = X_{t-1}$

Tenemos: $L(L(X_t)) = X_{t-2}$

Y tenemos: $L^s(X_t) = X_{t-s}$

Observe con atención que la potencia s en L^s significa que se aplica s veces la función $L()$ y realmente no se trata de una potencia (como x al cuadrado)... ¡Es una notación cómoda y utilizada a menudo que no debería llevarle a confusión!

Nuestra formulación se convierte en:

$$X_t = \sum_{s=1}^{s=r} \varphi_s L^s (X_t) + \varphi_0 + \varepsilon_t$$

Con una escritura todavía más condensada que hiciera pasar diversos términos a la izquierda tendríamos:

$$X_t - \sum_{s=1}^{s=r} \varphi_s L^s (X_t) - \varphi_0 = \varepsilon_t$$

Para condensar esta expresión, llamamos Φ (la mayúscula de φ) a una función polinomial (es decir, de tipo $\Phi(z) = \sum_{s=0}^{s=r} \varphi'_s z^s$) donde la mayoría de coeficientes φ'_s son iguales a nuestros φ_s , aunque no todos (haga sus cálculos si quiere convencerse recordando que z^s es igual a $z(z(z(\dots)))$, aplicado s veces).

Sabiendo que Φ se aplica sobre L y que el resultado se aplica sobre la serie X_t , aparece una notación muy condensada.

Esta notación expresa que, en el caso de los fenómenos autorregresivos, la aplicación de un operador polinomial bien escogido de la función de retardo, es decir $\Phi(L)$, sobre la X_t serie es un ruido blanco débil: $\Phi(L) X_t = \varepsilon_t$

c. Consideraciones (muy) útiles

Introducción a las medias móviles

La **media móvil simple** se denomina SMA en inglés (*Simple Moving Average*):

$$\text{SMA}(r) = (X_t + X_{t-1} + \dots + X_{t-(r-1)}) / r$$

Cuando se aplica esta operación sobre todos los términos de una serie (se dice que se **aplica un filtro** sobre la serie), se **liman** las asperezas de la serie.

Evidentemente, y hablando con más rigor, para indicar el nombre de la serie y el índice que se ve afectado habría que escribir: $\text{SMA}((X_t), t, r) = (X_t + X_{t-1} + \dots + X_{t-(r-1)}) / r$

y habría que indicar que solo puede calcularse para los índices t tales que $X_{t-(r-1)}$ existe.

Esta noción de media móvil **no debería confundirse con la noción de MA()**, que se corresponde con otro modelo de serie temporal diferente a AR() y que se denomina «Moving Average Model». La filiación de estos nombres no es producto del azar, de modo que el riesgo de confusión es todavía mayor.

Las MA() son series que expresan la respuesta a uno o varios hechos aleatorios. Se calculan los términos de la serie temporal como la **suma de un valor medio de la serie**, de ahí la filiación con las medias móviles simples, y de un polinomio que se aplica sobre los r últimos valores de una serie temporal que resulta ser un **ruido blanco débil**.

La forma de una MA(r) es la siguiente (con μ un valor medio):

$$X_t = \mu + \varepsilon_t + \sum_{s=1}^{s=r} \theta_s \varepsilon_{t-s}$$

MA y AR... conjuntos y ARMA

Un proceso autorregresivo de orden 1, AR(1) tal que

$$X_t = \varphi_1 X_{t-1} + \varphi_0 + \varepsilon_t$$

es estacionario a condición de que

$$|\varphi_1| < 1$$

 Observación para los matemáticos: en el caso de procesos autorregresivos de orden superior (AR(r)), el hecho de que la serie temporal sea estacionaria está condicionado por una restricción sobre Φ , que es que todas sus raíces deben estar ubicadas por debajo de una «esfera» unidad.

Cuando un proceso AP(r) es estacionario, es posible exhibir su media móvil infinita: MA(∞).

El modelo ARMA(p,q) considera series temporales que asocian los dos modelos anteriores:

$$X_t = \mu + \varepsilon_t + \sum_{s=1}^{s=p} \varphi_s X_{t-s} + \sum_{s=1}^{s=q} \theta_s \varepsilon_{t-s}$$

siendo p el orden de la parte autorregresiva del modelo, y q la parte media móvil.

Hemos visto antes la función polinomial de L llamada $\Phi(L)$ para las AR(p). Esta función es en sí misma un operador sobre la serie temporal. Además, podemos definir un operador $\Theta(L)$ para el modelo media móvil MA(q) que se aplica sobre los ruidos blancos en lugar de aplicarse sobre las series temporales.

Con la notación utilizada más arriba, la formulación de un ARMA sería:

$$\Phi(L)X_t = \Theta(L)\varepsilon_t$$

Procesos vectoriales: VARMA

Se dice que un proceso es **vectorial** cuando, en lugar de depender de una serie de índices temporales, depende de varias series de índices temporales.

Sería más claro hablar de **series temporales multivariadas**.

Por ejemplo, la cifra de negocios de un ostricultor depende del ciclo de las mareas (ciclo no sincronizado con el año civil) y del ciclo de las estaciones (durante los meses sin «r» no se consumen ostras, y en Navidad se consumen muchas).

Tenemos aquí dos vectores temporales (dos variables explicativas temporales) que influyen en la variable explicada.

La extensión del modelo ARMA correspondiente a las series temporales se denomina modelo Vector AutoRegressive Moving-Average: VARMA.

Los modelos VARMA deberían cubrir el caso más sencillo en el que las variables explicativas son independientes, y también el caso en que existe alguna dependencia entre las variables explicativas. Esto presenta un gran interés, pues le permite explotar la «covarianza» entre estas variables explicativas temporales.

Hasta aquí hemos trabajado sobre modelos estacionarios, ¿cuáles son los métodos cuando el modelo no es estacionario?

3. Procesos no estacionarios

a. El modelo ARIMA

A menudo se utiliza una generalización de ARMA(p,q) llamada ARIMA(p,d,q), siendo d un nivel de «diferenciación», para tratar los modelos no estacionarios. ARMA trata los modelos estacionarios y buscamos trabajar sobre modelos no estacionarios. ¿Cómo podemos pasar de uno al otro?

Se define una función ∇ , Nabla, aquí operador diferencia, que de hecho es la función $(1-L)$.

El operador ∇ **elimina las tendencias**, tanto más cuantas más veces se aplica. De este modo, aplicando un cierto número de veces este operador, **esperamos obtener una serie estacionaria**, lo que nos permitirá aplicar la modelización ARMA anterior.

Un proceso es ARIMA cuando existe un número entero tal que el proceso $\nabla^d X_t$ es un ARMA(p,q) y que:

$$\phi(L) \nabla^d X_t = \theta(L) \varepsilon_t$$

Para los matemáticos, diremos que a menudo sucede que $\phi(L)$ y $\theta(L)$ tienen sus raíces estrictamente en el exterior de la «esfera» unidad y no comparten ninguna raíz. Si las raíces están todas en el exterior de la esfera unidad, entonces decimos que el proceso es un proceso **causal** y, si las raíces están todas estrictamente en el interior, se dice que el proceso es **inversible**. La prueba estadística se denomina KPRESS.

Observación sobre la notación:

un ARIMA(p,0,q) es un ARMA(p,q)

un ARIMA(p,0,0) es un AP(p)...

b. Procesos estacionales: SARIMA

Los procesos estacionales tienen su propia modelización ARIMA: Seasonal-ARIMA.

La idea es que un desfase de S índices, que se corresponde con una «estación», crea una estructura similar. Evidentemente, identificar esta estacionalidad resulta más complicado cuando el proceso no es estacionario, pues esta estructura se aplica sobre la tendencia. Imagine variaciones estacionales en torno a una cifra de negocios creciente cada año, esto no impide que la cifra de negocios durante las vacaciones de verano de ciertas empresas disminuya en la misma proporción todos los años.

El principio de este modelo es tratar de eliminar la influencia de las estaciones aplicando un operador: $(1 - L^S)$ un cierto número de veces; este número se llama D, en alusión a las d tendencias generales. Se aplica este operador $(1 - L^S)^D$ sobre la serie obtenida aplicando la limpieza de la tendencia.

Al mismo nivel que un ARIMA(p,d,q), un proceso SARIMA se caracteriza parcialmente por su D, pero también por sus P y Q, que son respectivamente el orden de la parte autorregresiva del modelo estacionario y el orden de la parte media móvil del modelo estacionario.

De este modo, algunos autores y ciertos paquetes de R describen un SARIMA de la siguiente manera: SARIMA (p,d,q,S,P,D,Q).

Tras haber implementado cada uno de estos modelos, es normal estudiar el aspecto de sus residuos. Hemos visto en un capítulo anterior que la hipótesis de **homocedasticidad** de los residuos era importante, es decir, que la varianza de los errores era constante independientemente del valor de t. Tras utilizar un modelo ARIMA en ciertos contextos perturbados, comprobará a menudo que este no es el caso, por ejemplo en el caso de los mercados financieros. ¿Qué podemos hacer?

Aquí es donde intervienen los siguientes modelos, especializados en el tratamiento de aquellos casos donde los errores son **heterocedásticos**.

c. Modelos ARCH y GARCH

ARCH significa *AutoRegressive Conditionally Heteroscedastic*.

GARCH significa *Generalized AutoRegressive Conditional Heteroskedasticity*.

Modelo ARCH(q)

Es un modelo autorregresivo en el que se aborda el problema de la **heterocedasticidad de los residuos**.

Estos modelos son característicos del mundo financiero y de la econometría.

Imagine que P_t representa la evolución del precio de un «asset» de los mercados financieros en el tiempo («securities», índices...).

En la modelización original de los mercados financieros, nos referíamos a la idea de que la evolución de este precio debería ser aleatoria, y todo quedaba igual por otro lado. Si se quiere describir que este precio es aleatorio, se modela con lo que se denomina un mercado aleatorio: $P_t = P_{t-1} + \varepsilon_t$, ε_t es un ruido blanco débil normal y los intervalos de t son supuestos débiles en vista de la realidad abordada.

 De hecho, los financieros se interesan sobre todo en la evolución instantánea del rendimiento de los «assets». La serie temporal que siguen atentamente se corresponde con la fórmula: $Y_t = \ln(P_t / P_{t-1})$, que recuerda a un logaritmo neperiano.

O más «old school», pero correspondiente a la práctica de los mercados:

$Y_t = 100 \log(P_t / P_{t-1})$, que utiliza un logaritmo en base 10 y que se percibe como un porcentaje (sorprendentemente).

Esta serie se denomina «continuously compounded rates of return», es decir, en español: «rendimiento anual continuamente compuesto».

El uso del logaritmo disminuye el impacto sobre la curva de las grandes diferencias de precio y expresa efectivamente un valor negativo si el rendimiento baja, pues la relación de precios será inferior a 1 y, por tanto, su logaritmo será negativo.

Hemos escrito antes que los errores poseen, desafortunadamente, una varianza que depende del tiempo. Para modelar esta variabilidad, se escribe que los errores pueden representarse como el producto de un **ruido blanco fuerte** que expresa una aleatoriedad de gran amplitud y una serie temporal de desviaciones típicas:

$$\varepsilon_t = \sigma_t z_t$$

No nos interesaremos por el signo del error, sino por su amplitud, y se modela entonces la varianza como un polinomio de errores al cuadrado.

Observe con atención que siempre podemos **tratar de expresar una función como un polinomio sobre un intervalo de definición acotado, si esta función es continua y derivable un cierto número de veces y si los valores de las derivadas están acotados en el intervalo en cuestión**. Evidentemente, cuanto más pequeño sea el intervalo, menor será el error máximo entre el polinomio y la «verdadera» función. El método no tiene nada de extraño:

$$\sigma_t^2 = \phi'(\varepsilon_t^2)$$

El resto le resultará más o menos familiar:

- Se estima un modelo AR del tipo: $\phi(L) Y_t = \varepsilon_t$
- Esto nos procura una estimación de $\varepsilon_t : \hat{\varepsilon}_t$
- A continuación encontramos una estimación de los coeficientes de ϕ' realizando una regresión lineal sobre $\phi'(L) \hat{\varepsilon}_t^2 = \hat{\varepsilon}_t^2$
- Para hacer esto, se escoge un nivel de lag (retardo), que llamaremos q .
- En este proceso, se realizarán pruebas estadísticas para comprobar que los coeficientes de ϕ' son significativos y no ocultan otros. Llegado el caso, si algún coeficiente no es significativo, descubriremos que no hemos logrado construir un modelo ARCH(q).

Existen muchas variantes y evoluciones de este modelo que se salen del marco de este libro y a los que debería enfrentarse si trabaja sobre los mercados financieros.

Modelo GARCH(p,q)

GARCH(p,q) es un modelo que generaliza ARCH de manera similar a ARMA, pero respecto a las varianzas:

$$\sigma_t^2 = \phi'(\varepsilon_t^2) + \phi(X_t^2)$$

que proviene de la observación de que esta varianza puede que no siempre tienda a 0 (el mundo financiero se renueva siempre...).

Durante estos últimos años, el modelo GARCH era uno de los más populares en el **mundo financiero**.

d. Convolución y filtros lineales

De manera muy operacional, ciertas operaciones que hemos visto más arriba pueden comprenderse, como la aplicación de filtros sobre los datos.

Hay que tenerlo en mente, pues muchas herramientas y autores le hablarán de filtros sobre las series temporales y es conveniente relacionar esto con lo que hemos visto antes.

Filtros

¿Qué es un filtro? Nuestra respuesta será un poco simplista, ipero trata de desmitificar el asunto!

Como una primera aproximación, imagine una criba que dejara pasar únicamente algunos valores y otros no:

Tomemos la serie (10,15,20,30) y multipliquémosla miembro a miembro por (1,0,1,0): obtenemos (10,0,20,0). Hemos filtrado la primera serie mediante este filtro. Por extensión, toda multiplicación u operación miembro a miembro de una serie por otra puede percibirse como un filtro aplicado por una de las dos series sobre la otra.

El operador de retardo L es un filtro.

Componer filtros da como resultado nuevos filtros.

Filtro y convolución

Volvamos sobre este tipo de expresión:

$$\sum_{s=1}^{s=p} \varphi_s X_{t-s}$$

Esta operación puede percibirse como la aplicación de un retardo L^s , seguida por la aplicación de un filtro formado por los φ_s .

Las matemáticas utilizan una operación similar que tiene la siguiente forma:

$$\phi * \phi' (t) = \sum_{s=-\infty}^{s=\infty} \varphi_s \phi'_{t-s}$$

y que se llama producto de convolución de dos series ϕ y ϕ' .

Cuando los valores de los coeficientes de las dos series son nulos para los índices inferiores a 1 y a partir de ciertos índices positivos, estamos delante de un producto de convolución fácil de calcular. Es el caso más habitual cuando se recogen datos operacionales que se sitúan sobre rangos de tiempo delimitados.

Existe una versión continua de este producto de convolución, que nos permite extender la noción de filtrado a las series temporales que estuvieran definidas por su función generatriz continua:

$$\phi * \phi' (t) = \int_{-\infty}^{\infty} \phi(s) \phi' (t - s) ds .$$

El producto de convolución es commutativo, distributivo sobre la suma y, a menudo, asociativo (bajo ciertas condiciones de integración).

El producto de convolución de dos distribuciones da como resultado una dilatación (o una contracción) de una en proporción de los valores de la otra sobre la que se compone una traslación del conjunto.

Distribución de Dirac y producto de convolución

La distribución de Dirac o delta de Dirac δ_a es una función que vale cero en todo el rango salvo en a, donde vale infinito (y su integral, es decir, su superficie, vale 1). Se trata de una distribución que resulta muy práctica para modelar un impulso fuerte.

De este modo, se trata de una distribución cuya representación gráfica es un pico vertical infinito, que podemos representar como una raya vertical.

La delta de Dirac 0: δ_0 , a menudo representada simplemente por δ , es el elemento neutro de la convolución, es decir, que la convolución de una distribución por δ no cambia nada.

Cuando realiza la convolución de una distribución por una delta de Dirac δ_a obtiene simplemente la **misma distribución trasladada en a** (esto debería recordarle en cierta medida a la función de retardo).

Filtros lineales: lo que hace falta saber

En esta sección, vamos a llamar a nuestros filtros $F(\cdot)$.

Siendo X la serie definida por X_t , la aplicación del filtro F devuelve una serie Y definida por Y_t tal que:

$$Y_t = F(X)_t$$

El significado importante de esto es **que hace falta conocer toda la serie X para calcular cada valor de la serie Y**. Esto resulta un **verdadero problema**, que explica la necesidad de tener que utilizar a menudo nociones intermedias matemáticamente más «ricas» para realizar los cálculos operacionales que afortunadamente podrá llevar a cabo con ayuda de los paquetes de R correspondientes (funciones de transferencia + convolución... + transformadas diversas, entre ellas transformadas de Fourier).

Un filtro lineal posee tres características importantes:

$$F(\alpha X) = \alpha F(X)$$

$$F(X + X') = F(X) + F(X')$$

$$F(L^s X) = L^s F(X)$$

Esta última característica es muy importante; el operador de retardo puede aplicarse indistintamente antes o después del filtro.

El operador de retardo es un filtro lineal.

La formulación general de un filtro lineal es la siguiente:

$$Y_t = \sum_{s=-\alpha}^p \varphi_t X_{t-s} + \sum_{s=1}^q \varphi'_t Y_{t-s}$$

Si $\alpha > 0$, hace falta conocer los valores futuros para calcular la serie, un filtro de este tipo es un filtro no causal. No se puede producir la serie sobre la marcha, pues se necesitan los valores futuros de la serie.

Con esta escritura, he aquí la fórmula de la media móvil:

$$Y_t = \frac{1}{2k+1} \sum_{s=-k}^k X_{t-s}$$

Se trata de un filtro **paso-bajo, no causal**, que preserva las bajas frecuencias y que elimina las altas frecuencias.

Dos ejemplos de filtros lineales simples:

- *smoothing*: $Y_t = \frac{1}{4}X_{t-1} + \frac{1}{2}X_t + \frac{1}{4}X_{t+1}$
- *differentiation*: $Y_t = X_t - X_{t-1} = (1-L)X_t = \nabla X_t$

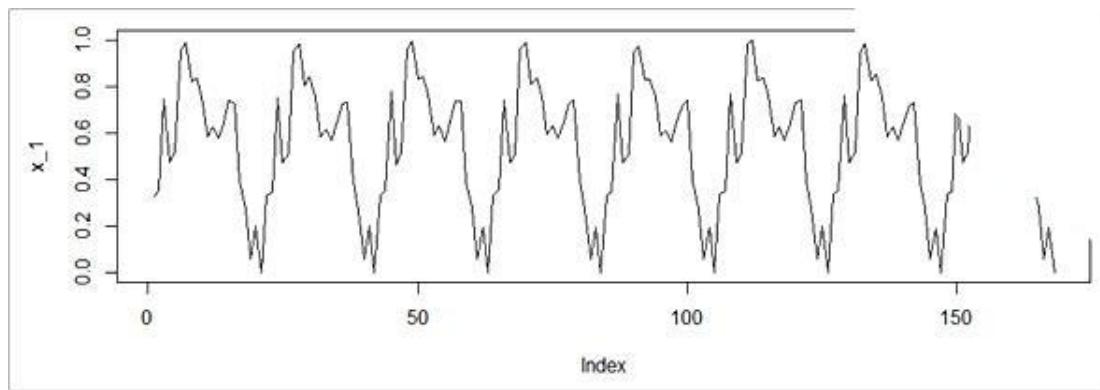
Este último es un **filtro causal, paso-alto**. Elimina las bajas frecuencias y mantiene las altas frecuencias, si bien hace emerger lo que sale de la tendencia de una serie.

4. Puesta en práctica

Una serie cualquiera de números puede verse como una serie temporal. Tomemos, por ejemplo, la siguiente serie y representémosla como gráfico de tipo línea.

```
plot(x_1, type = "l")
```

Esto produce el siguiente gráfico:



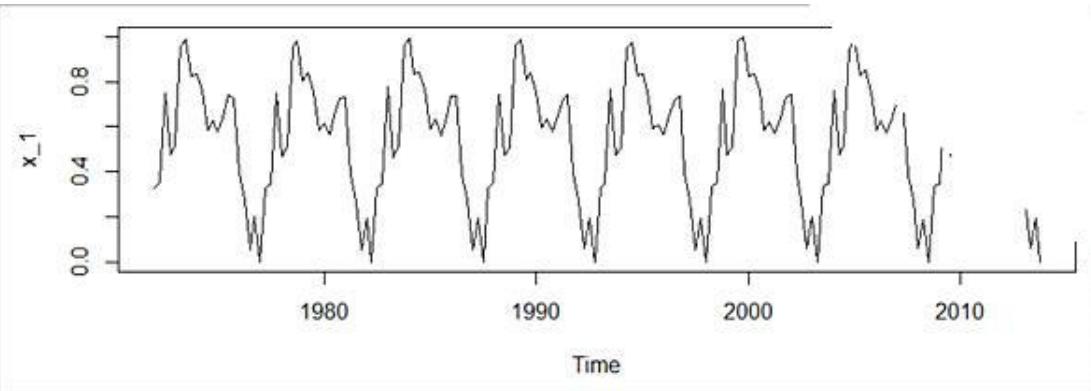
Una serie basada en su índice, sin noción de fecha

a. Los fundamentos de la manipulación de las series temporales en R

Estos índices no se parecen realmente ni a fechas ni a tiempos. Además, nos gustaría utilizar el tipo de R especializado en las series temporales. Habrá que pasar a la función dos datos importantes: la fecha de inicio y la frecuencia de las observaciones por unidad de tiempo.

```
x_1 <- ts(x_1,
            frequency = 4,
            start = c(1972, 1)
          )
plot(x_1)
```

El resultado pone de relieve las fechas convencionales:



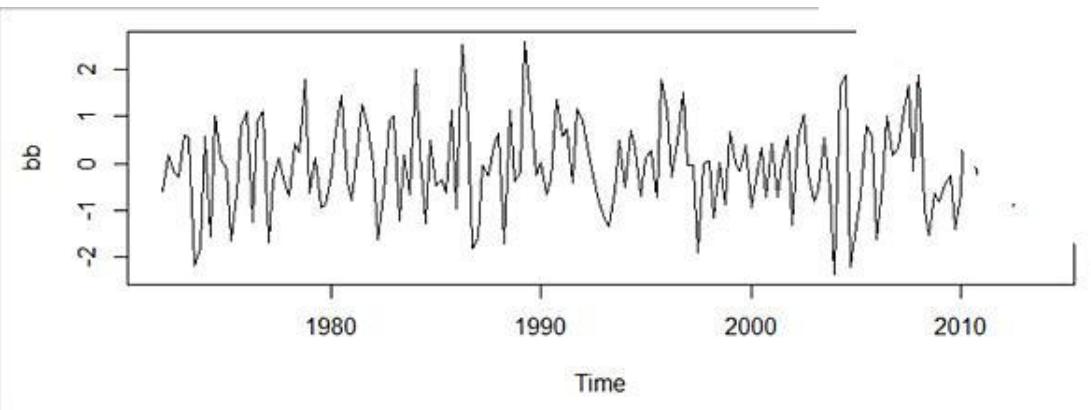
Serie temporal, con sus fechas

La función **plot** ha reconocido el tipo «serie temporal» y ha creado automáticamente un grafo con líneas.

A continuación vamos a crear un ruido blanco sobre el mismo intervalo de tiempo.

```
bb <- rnorm(length(x_1),0,1)
bb <- ts(bb,
        frequency = 4,
        start = c(1972, 1)
      )
plot(bb)
```

Lo que produce un resultado con un aspecto muy aleatorio:



Ruido blanco

Dispone de diversas funciones para extraer información de una serie temporal.

```
# información acerca de la serie
start(x_2)      # inicio de la serie
end(x_2)        # fin de la serie
frequency(x_2)  # número de datos por periodo
deltat(x_2)     # cuya frecuencia es...
```

```
[1] 1972      1  
[1] 2013      4  
[1] 4  
[1] 0.25
```

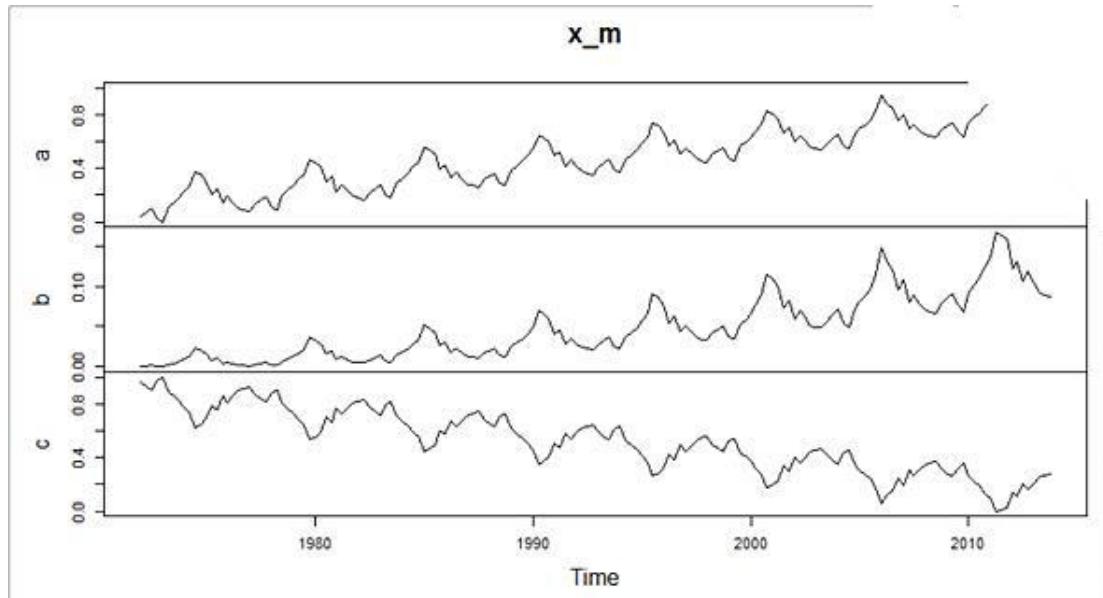
También podemos extraer una parte de la serie **x_2**.

```
# extraer valores de una serie temporal  
window(x_2, start = c(1974,3), end = c(1976,3))
```

```
          Qtr1      Qtr2      Qtr3      Qtr4  
1974           0.37435007 0.35533755  
1975 0.30239055 0.20597054 0.24675331 0.13751311  
1976 0.19132147 0.14054889 0.09941033
```

Es posible visualizar fácilmente varias series temporales sobre un mismo gráfico.

```
# series múltiples  
# 3 series temporales cualesquiera  
  
a <- x_2  
b <- x_2**2/6  
c <- 1-x_2  
  
# bind de las series  
x_m <- ts(cbind(a,b,c) ,  
           frequency = frequency(a), # no perder la frecuencia  
           start = start(a))         # no perder el inicio  
  
plot(x_m)                      # sobre 3 líneas
```



Tres series temporales sobre un mismo gráfico

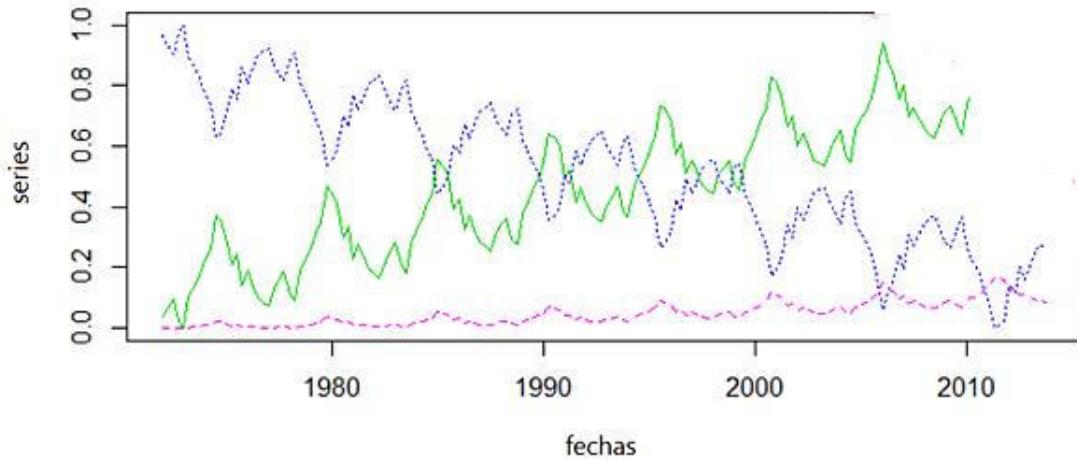
También podemos mostrar series temporales codificadas de diversas maneras (series, series múltiples, matrices)

en la misma ventana.

```
plot(x_m,
  plot.type = "single",
  lty = 1:3,
  col = c(3,6,12),
  ylab ="series",
  xlab ="fechas")

class(x_m)
```

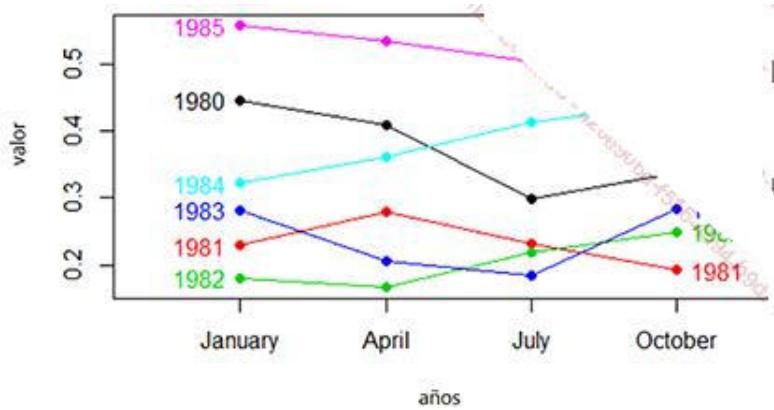
```
[1] "mts"      "ts"       "matrix"
```



Tres series temporales en la misma ventana

Veamos a continuación una pequeña función que le permite darse cuenta rápidamente si podría existir cierta estacionalidad anual (en años verdaderos).

```
# estaciones anuales o no
seasonplot(window(x_2,
  start = c(1980,1),
  end = c(1985,4)),
  ylab="valor",
  xlab="años",
  main="",
  year.labels=TRUE,
  year.labels.left=TRUE,
  col=1:20,
  pch=19)
```



Los años no se parecen, de modo que hay pocas posibilidades de que exista una estacionalidad anual manifiesta.

b. Estudio de las series temporales

La primera herramienta que nos viene a la mente es, sin duda, el cálculo de una media móvil.

Vamos a comparar los comportamientos sucesivos:

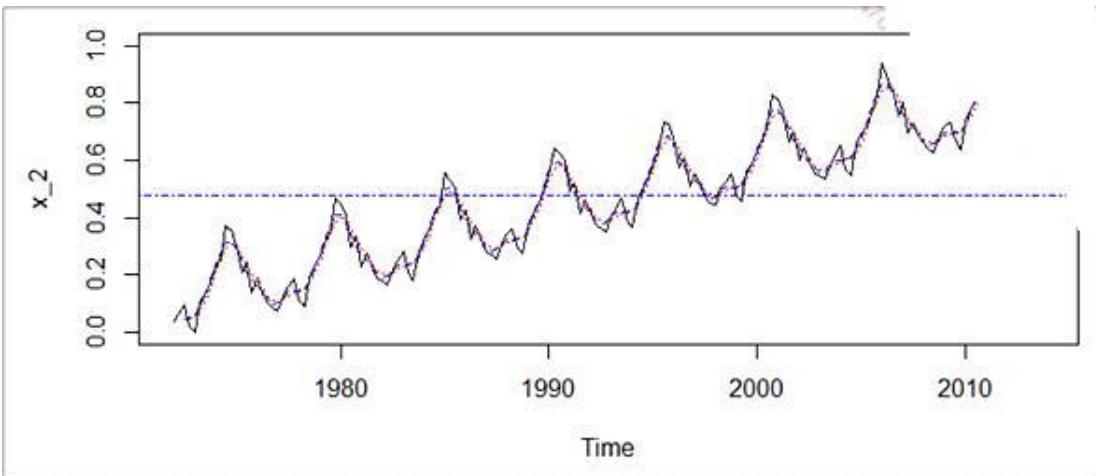
- Un filtro compuesto de funciones **filter()** que proceden por convolución para calcular la respuesta (los coeficientes creados mediante la función **rep()**, aunque también habríamos podido definirlos a mano o de algún otro modo). Estas funciones se utilizan aquí para calcular una media móvil robusta.
- La función **ma()**, media móvil.

```
x_2.df <- data.frame(x_2)           # data.frame fácil de visualizar
ml <- lm(x_2~. ,data =x_2.df )    # regresión lineal

                                # moving average por media
                                # de 2 convoluciones
orden <- 4
mal  <- (filter(x_2, rep(1/orden, orden), side = 1) +
          filter(x_2, rep(1/orden, orden), side = 2))/2

                                # moving average de R
ma2  <- ma(x_2, order = orden)

                                # gráficos
plot(x_2)                      # la serie temporal
abline(ml, col = "blue", lty = 4)      # la regresión
lines(mal, col = 2 , lty = 3)        # ma manual
lines(ma2, col = 4 , lty = 2)        # ma R
```

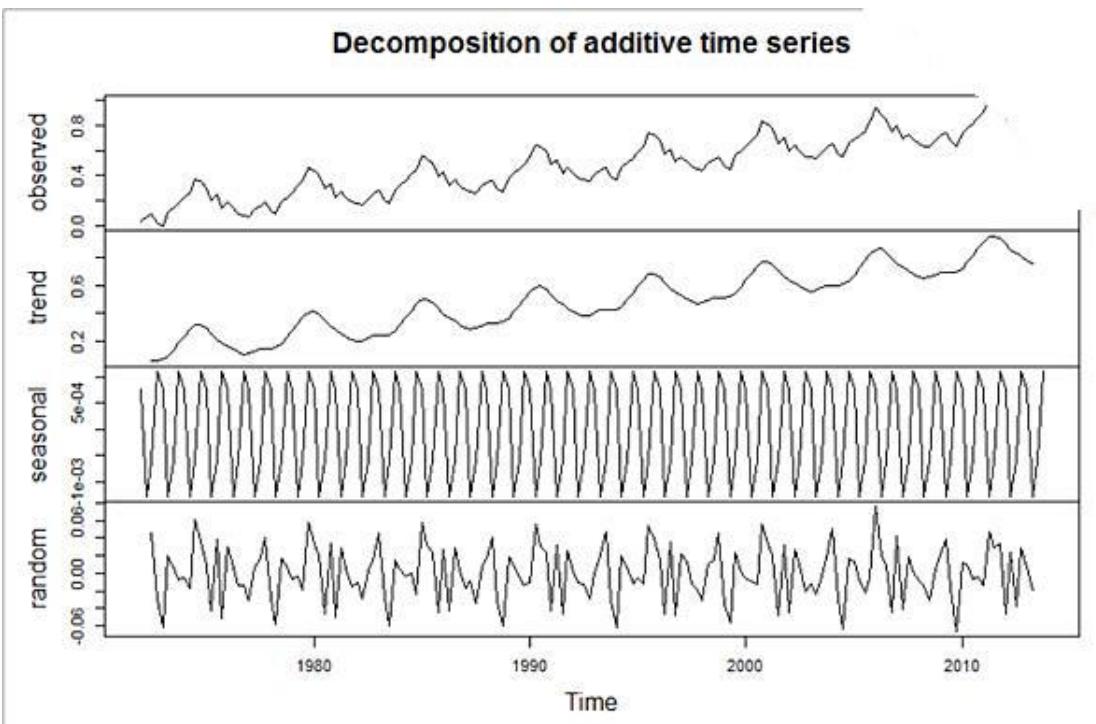


Regresión y medias móviles

Los dos cálculos de medias móviles han producido resultados idénticos. Si no hubiéramos realizado la media de los parámetros side, habríamos tenido un ligero desplazamiento a la derecha o a la izquierda de la curva.

A continuación vamos a tratar de realizar una descomposición aditiva de la serie temporal (para descomponer de forma multiplicativa, utilice la opción **multiplicative**).

```
x_2_m <- decompose(x_2)
plot(x_2_m)
```



Descomposición aditiva en 3 series

Se ha obtenido una tendencia, una estacionalidad y un ruido. Este ruido está compuesto por una sucesión de secuencias de ruidos idénticas, lo que nos permite suponer que esta serie ha sido construida. Es, de hecho, un indicio de fraude sobre los datos: no se trata de un ruido natural. También podría ser la señal de un problema del

algoritmo de descomposición aditiva respecto a las características de nuestra serie.

Podríamos querer recomponer la serie sin su ruido; aquí esto no aporta gran cosa, pues la estacionalidad es despreciable (observe el gráfico, posee una amplitud de 4/1000, mientras que la tendencia posee una amplitud de 1).

Para recomponer, inos contentamos con sumar sin el ruido!

```
x_2_ <- x_2_m$trend + x_2_m$seasonal  
plot(x_2_)
```

Sabiendo que no disponemos de datos antes del tercer trimestre de 1972, ni de datos que cubran el final del último año, estas operaciones generan valores desconocidos en el inicio y el final de la serie temporal.

```
which(is.na(x_2_m$trend + x_2_m$seasonal+x_2_m$random))
```

```
[1] 1 2 167 168
```

Encontrará otras funciones de descomposición de series temporales:

- **stl()** en el paquete **stats**.
- **decomp()** en el paquete **timssac**.
- **tsr()** en el paquete **ast**.

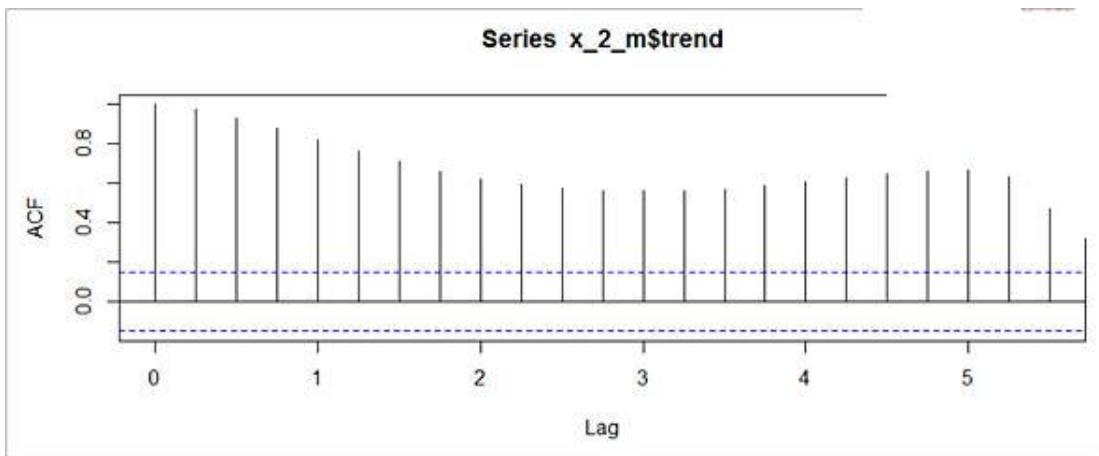
Pruebe esta a modo de ejemplo:

```
stl(x_2, t.window=21, s.window="periodic", robust=TRUE)
```

Observamos las autocorrelaciones de la descomposición:

```
acf(x_2_m$trend, na.action = na.pass)
```

Utilizamos la función **acf()**.



Autocorrelaciones de la tendencia

Para el lag nulo, es decir, sin retardo, se obtiene una autocorrelación perfecta, iuna serie siempre está correlada consigo misma! En este tipo de gráfico, este lag solo le sirve como recordatorio... olvídelo.

La autocorrelación no decrece muy rápido, el proceso no es o es poco estacionario.

La curva de autocorrelación oscila, lo que permite pensar que existe cierta estacionalidad de la tendencia (no hablamos aquí de la estacionalidad débil en torno a la tendencia estudiada más arriba, sino a la variación de la propia tendencia).

Realicemos la prueba KPSS, *Kwiatkowski Phillips Schmidt Shin* (basada en la presencia de raíces en la esfera unidad, ver más arriba).

```
library(tseries)
kpss.test(x_2_m$trend)           # H0 estacionaria
kpss.test(x_2_m$trend, null = "Trend") # H0 tendencia estacionaria
```

```
.....
p-value = 0.01
p-value smaller than printed p-value
```

Primera prueba: hipótesis nula rechazada --> serie sin duda no estacionaria

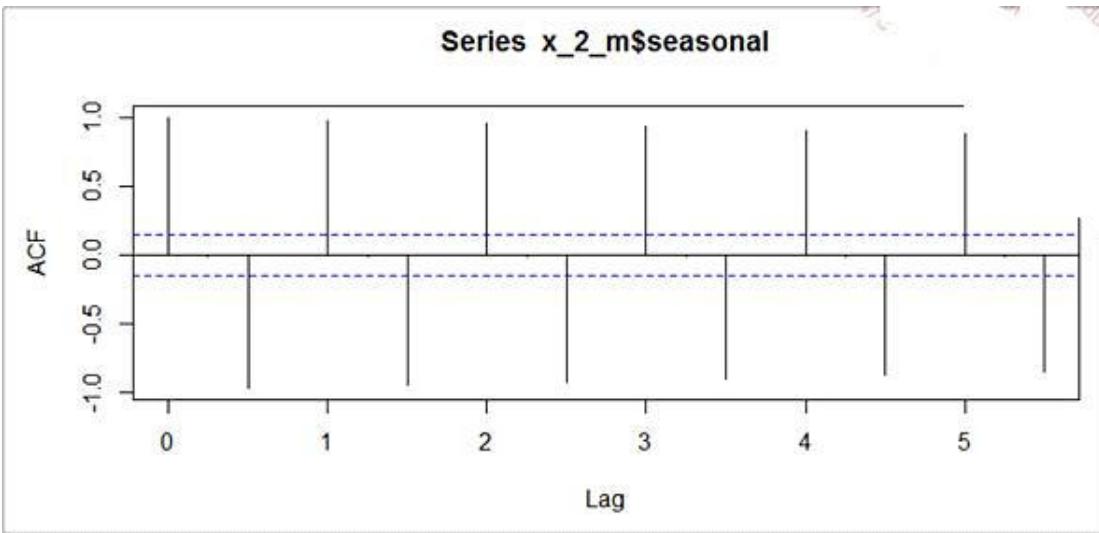
```
.....
p-value = p-value = 0.1
p-value greater than printed p-value
```

Segunda prueba: hipótesis nula no rechazada --> tendencia sin duda estacionaria en torno a la curva de tendencia.

Estudiemos a continuación la estacionalidad:

```
acf(x_2_m$seasonal, na.action = na.pass)
```

Las autocorrelaciones de la estacionalidad son típicas, próximas a -1 o a 1 de manera alterna.



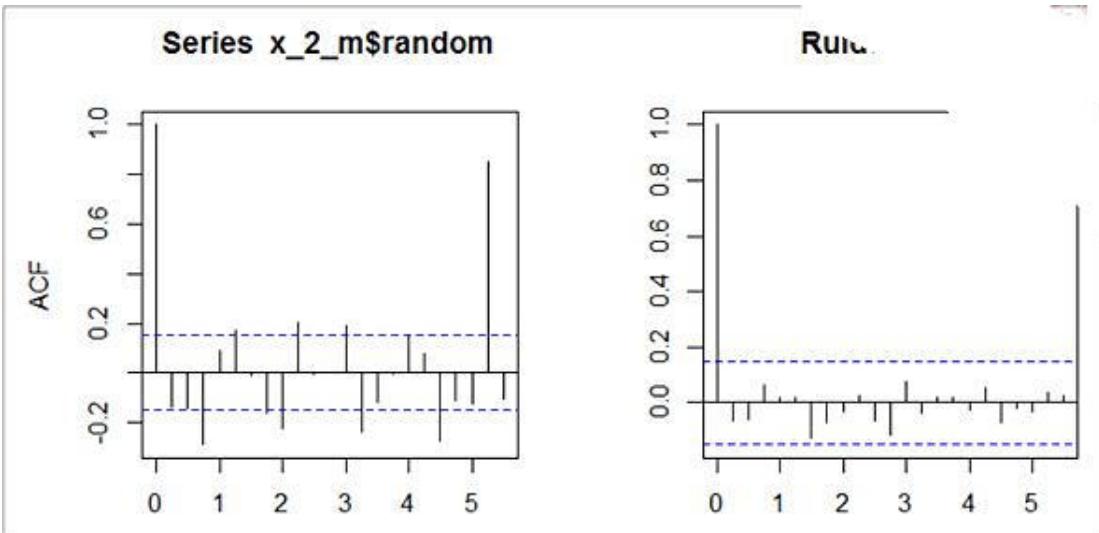
Autocorrelaciones de la estacionalidad

Se trata de una estacionalidad anual (cuatro trimestres). Hemos confirmado más arriba que su amplitud es muy débil, pero está presente. Sin embargo, sería difícil utilizarla, pues su amplitud es netamente más débil que el ruido.

Estudiemos este ruido un poco extraño:

```
op <- par()                      # 2 por gráfico
par(mfrow = c(1,2))
                                # acf ruido VS ruido blanco
acf(x_2_m$random, na.action = na.pass, xlab = "")
acf(bb, main = "Ruido blanco", xlab = "", ylab = "")
par(op)
```

Vamos a comparar este ruido con el ruido blanco fabricado más arriba.



Ruido extraído frente a ruido blanco

Como habíamos percibido, nuestro ruido extraído posee muchos pequeños defectos y un gran defecto de periodicidad.

Resulta interesante realizar el test de Ljung-Box aumentando el número de lags.

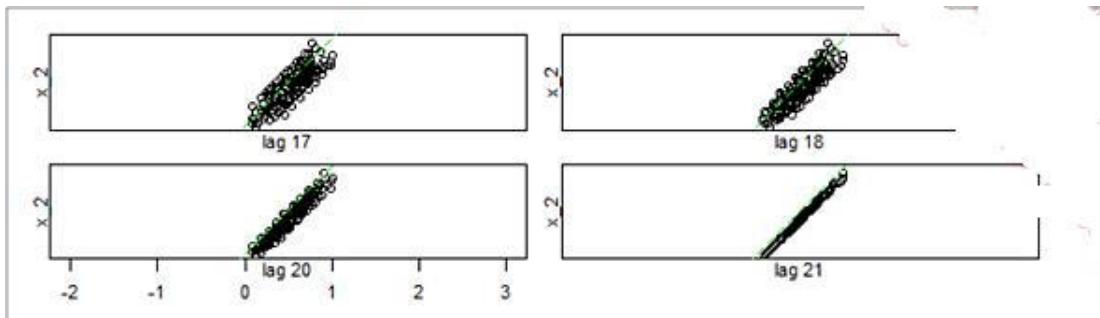
```
# H0: test de independencia  
Box.test(x_2_m$random, lag=3, fitdf=0)
```

.../... df = 3, p-value = 0.0001603

A partir de tres lags, la hipótesis H0 se rechaza, el ruido posee una autocorrelación. Esto confirma nuestro análisis acerca de la mala calidad del ruido.

Una observación detallada de los lags puede hacerle dudar.

```
# no estacionario  
lag.plot(x_2, lags=21,  
         do.lines=FALSE,  
         diag.col = "green",  
         layout = c(7,3))
```



Aspecto de los lags de una serie

Este aspecto «muy alineado» de cada lag es característico de una serie temporal no estacionaria. El lag 21 es el más característico y se corresponde con la periodicidad de la tendencia que puede constatar en su gráfico (5 años + un trimestre).

Estudiar las series temporales está bien... pero tras haberlas estudiado, a menudo se desea predecir su evolución.

c. Predicciones sobre ARIMA (AR MA SARIMA)

El siguiente ejemplo es una predicción de una serie ARIMA, cuyos coeficientes desconocemos. Estudiando la documentación de arima(), comprobará que es sencillo trabajar sobre AR, MA o SARIMA de la misma manera. Se trata tan solo de una historia de parámetros.

Nuestra serie se llama: **a**.

En primer lugar, pidamos a R el tipo de ARIMA que mejor se corresponde con nuestra serie.

```
auto.arima(a) # búsqueda de los mejores coeficientes
```

.... / ... ARIMA(1,1,1) ... / ...

Vamos a probar con ARIMA(1,1,1) y crear el modelo **m**.

Recordemos que: ARIMA(1,0,0) = AR(1), ARIMA(0,0,1) = MA(1) o que el índice central es el nivel de diferenciación.

```
titulo = "arima 1 1 1"      # un título para el plot
m <- arima(a, c(1,1,1))    # creación del modelo m
nivel_alisado_daniell <- 1 # escoger el nivel de alisado
```

Hemos escogido, además, el nivel de alisado para alimentar una función «kernel» que utilizaremos en el plot. Este valor no se utiliza en la modelización, de modo que puede jugar con él para adaptar mejor las características de su serie.

A continuación vamos a utilizar el modelo **m** para fabricar una serie predicha **p** sobre **12** nuevos índices y calcular tres series para comprender mejor visualmente el resultado de nuestro trabajo:

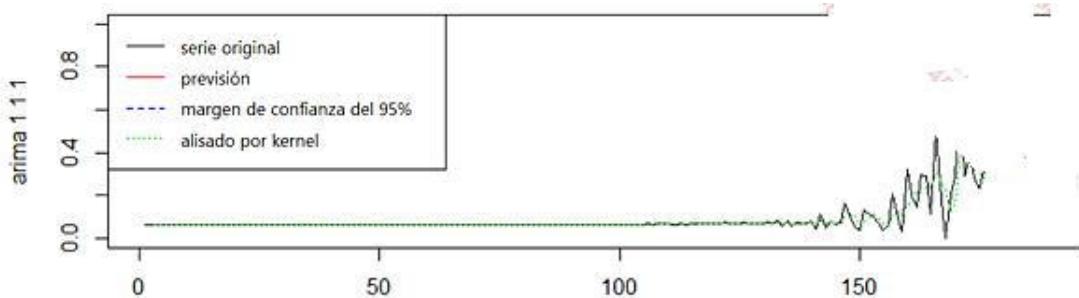
- Una serie de dos desviaciones típicas sobre la predicción.
- Una serie de dos desviaciones típicas por debajo de la predicción.
- Una serie que es el alisado de nuestra serie **a**.

```
# predicción y creación de series
p <- predict(m, n.ahead=12)      # predicción de valores ahead
min_p <- p$pred-2*p$se          # serie a -2 desviaciones típicas
max_p <- p$pred+2*p$se          # serie a +2 desviaciones típicas
                                # serie con alisado
ak <- kernapply(a, kernel("daniell", nivel_alisado_daniell))
```

Puede utilizar estas series temporales para realizar diversos cálculos; de momento visualizaremos el resultado.

```
# PLOT
ts.plot(a,p$pred, min_p, max_p, # traza las series
        col=c(1,2,4,4),
        lty = c(1,1,2,2),
        xlab = "",ylab = titulo)
lines(ak, col = 3, lty = 3)      # traza el alisado
legend("topleft",
       c("serie original",
         "previsión",
         "margen de confianza del 95%",
         "alisado par kernel"),
       col= c(1,2,4,3),
       lty =c(1,1,2,3),
       cex =0.8)
```

Observe el uso de la función **ts.plot()**.



Predicción 12 periodos ARIMA(1,1,1)

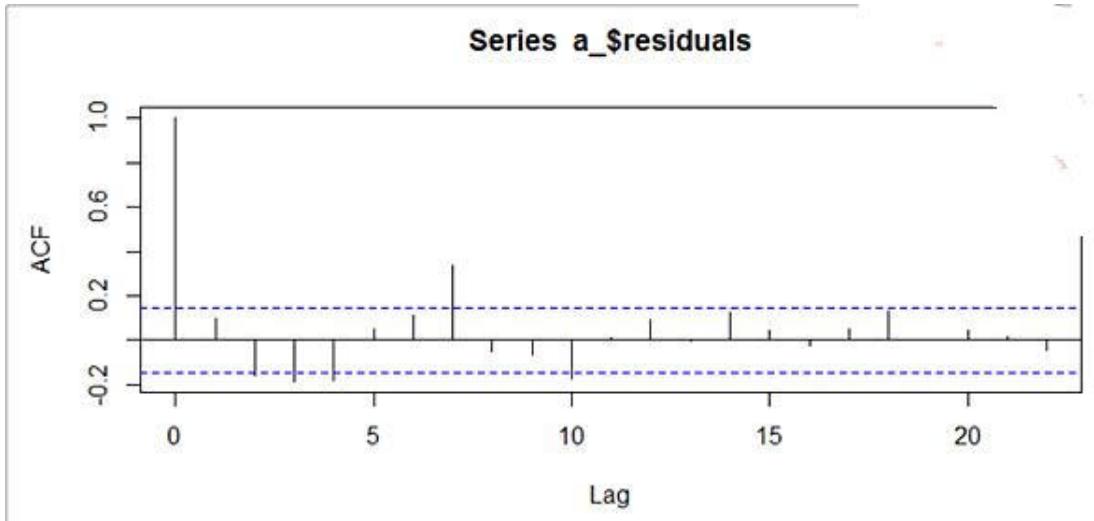
Observe el aspecto de la previsión y de los márgenes de confianza, la forma indica que hemos anticipado bien el descenso de la serie y su remontada posterior, pero con cierta incertidumbre. En los primeros períodos predichos (digamos cuatro o cinco), la previsión tiene todas las probabilidades de ser exacta, aunque el aspecto un poco explosivo de esta curva nos incite a intentar niveles de diferenciación más importantes o a echar un vistazo utilizando otro modelo.

Cuando su objetivo es una previsión a corto plazo, un pequeño error de modelo tiene consecuencias pequeñas. Crear una ley más general, que se proyecte más lejos en el tiempo, requerirá mucho más trabajo y preferentemente la recogida de más instancias del fenómeno que hay que describir.

Veamos ahora cuál es la calidad del ajuste del modelo creado.

```
# estudio residuos
require(forecast) # paquete de predicciones:
a_ <- forecast(m) # funciona sobre objetos propios de
# arima, fracdiff, auto.arima
# ar, arfim
acf(a_$residuals)
```

Nuestra idea es obtener residuos correspondientes a un ruido blanco.

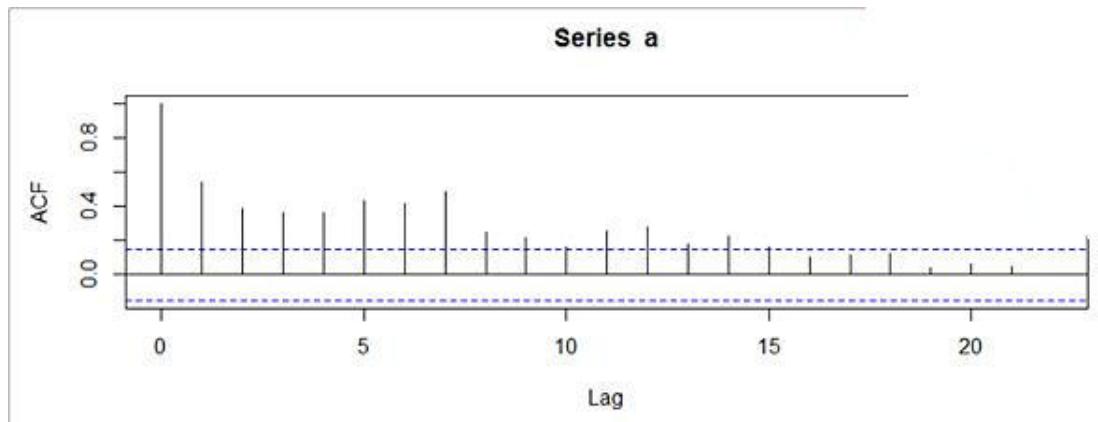


Comprobamos un defecto sobre el lag 7. Esto plantea el problema de la estacionalidad, que no había propuesto **auto.arima()**.

Para estar seguros, observemos la ACF de la serie original.

```
acf(a)
```

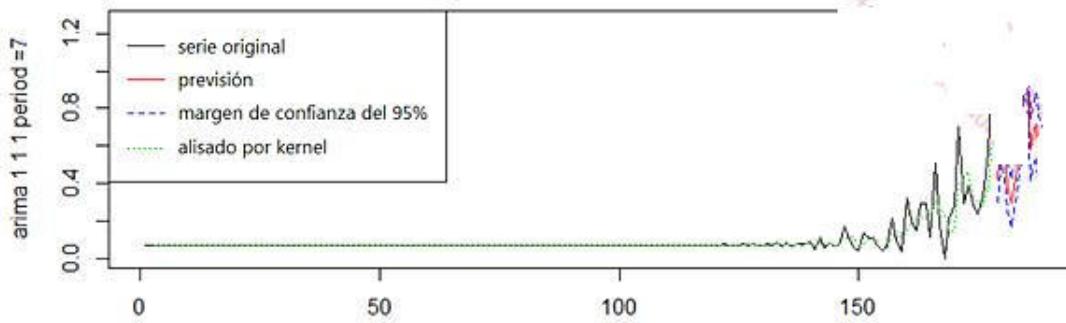
```
# acf serie original
```



ACF de la serie original

Comprobamos un pico de estacionalidad en lag 7. A continuación vamos a intentar introducir cierto periodo en el modelo. Vamos a utilizar el mismo código que antes, pero cambiando el modelo.

```
titulo = "arima 1 1 1 period =7"      # un título para el plot
m <- arima(a, c(1,1,1),            # creación del modelo m
           seasonal = list(order = c(1,1,1), period =7))
```



Serie ARIMA(1,1,1) periodo = 7

La estacionalidad se ha tenido en cuenta, de modo que el resultado es agradable a la vista. La desviación típica está considerablemente acotada.

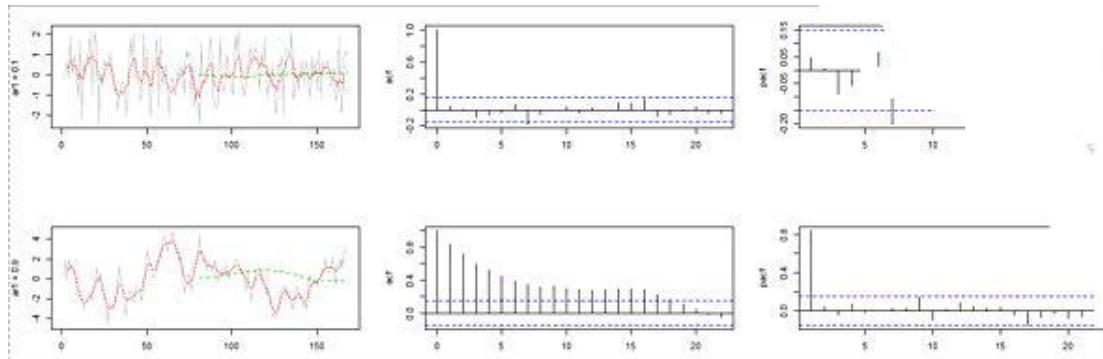
Disponemos de funciones similares para tratar las series temporales ARCH y GARCH y una manipulación similar (**garch()** y también **garchFit()** del paquete **fGarch**). El ajuste es algo más complicado, pues las variaciones de varianza de los errores son más delicadas de comprender visualmente y no le bastará con su buen criterio para

ayudarle.

Para ayudarle a desarrollar su «percepción» de las series temporales, le proponemos visualizar algunas series temporales y sus parámetros en un pequeño bestiario complementario.

5. Minibestiario ARIMA

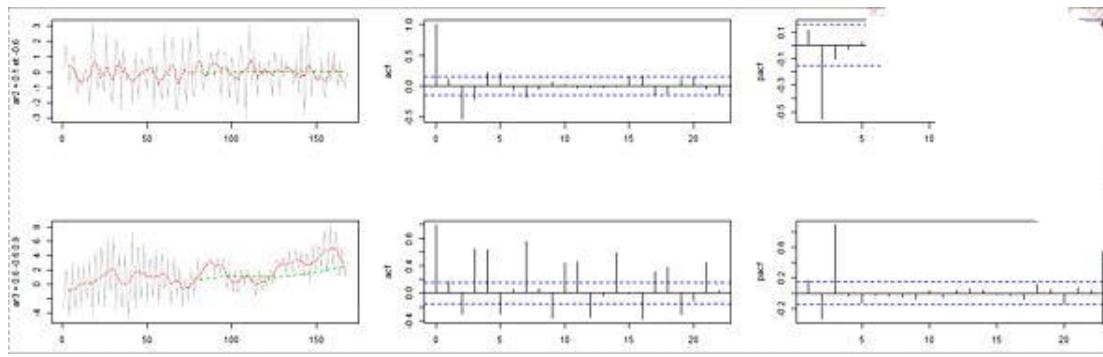
Vamos a hacer variar los parámetros ARIMA y observar el aspecto de las medias móviles, del alisado y de las curvas, pero sobre todo el aspecto de las autocorrelaciones y de las autocorrelaciones parciales. Las autocorrelaciones parciales (PACF, *Partial AutoCorrelation Function*) son las autocorrelaciones sin tener en cuenta los lags intermedios. Le permiten (entre otros) visualizar el orden de las AR (preste atención: observe que los gráficos ACF empiezan en lag 0 y los PACF en lag 1).



AR(1) 0.1 y 0.9

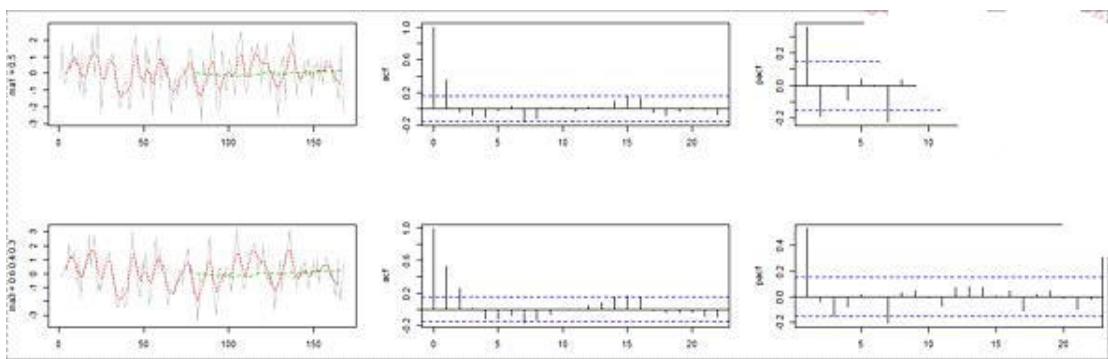
En AR(1)(0.1) la ACF es muy típica: se observa un decrecimiento instantáneo de los lags. Por el contrario, con un pequeño coeficiente, la PACF no le da ninguna pista ni previsión acerca del orden AR de la serie (es decir, que no se ve ningún pico en lag 1).

En AR(1)(0.9) el coeficiente próximo a 1 genera un comportamiento más próximo a una MA y los lags no decrescen tan rápido, existe también una pequeña estacionalidad hacia el lag 15, pero el orden 1 se ve bien sobre la PACF.



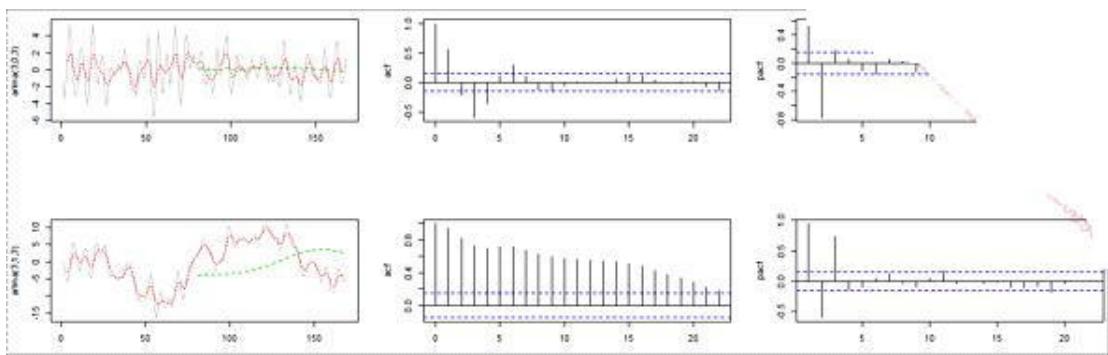
AR(2)(0.1, -0.6) y AR(3)(0.6, -0.6, 0.9)

Tenemos coeficientes positivos y negativos que se compensan. Por defecto, las series ya no serán estacionarias. AR(2)(0.1, -0.6) tiene su ACF decreciente y el orden 2 es visible por un pico negativo en la PACF. AR(3)(0.6, -0.6, 0.9) posee una ACF decreciente en el primer lag, pero una forma de estacionalidad; el orden 3 se observa perfectamente en la PACF.



$MA(1)(0.5)$ y $MA(3)(0.6,0.4,0.3)$

Las dos MA, cuyos coeficientes son todos del mismo signo (aquí positivos), poseen una ACF característica con un decrecimiento de los lags y una oscilación de estos (cuando se mezclan coeficientes positivos y negativos, esto es mucho menos visible). Las PACF poseen un lag 1 fuerte.



$ARIMA(3,0,3)$ y $ARIMA(3,1,3)$

$ARIMA(3,0,3)$ posee una ACF que se parece bastante a una ACF de MA, la PACF es un poco más contrastada. $ARIMA(3,1,3)$ ilustra la potencia de la diferenciación: con una sola diferenciación, la curva es mucho menos oscilante y se approxima a la media móvil, la ACF es la de una tendencia.

Ahora posee el suficiente conocimiento para acometer seriamente el estudio de las series temporales y abordar casos de uso más complejos estudiando la literatura correspondiente.

Sistemas difusos

Esta sección presenta las nociones que encontrará en diversos algoritmos y diferentes paquetes. No encontrará aplicaciones prácticas dentro de este capítulo. Esto le permitirá abordar diversos asuntos en otros contextos, en particular los sistemas expertos, que no trataremos debido a la falta de espacio en este libro. Estos sistemas parecen estar pasados de moda fuera del mundo industrial, pero tenemos la convicción de que volverán con fuerza entre las herramientas de los futuros data scientists.

El hecho de que los sistemas difusos (*fuzzy system*) resulten eficaces en el reconocimiento de patrones y en la aproximación de funciones «complejas» proviene en parte de su capacidad para soportar una codificación eficaz del conocimiento.

Los sistemas difusos se fundamentan en sistemas expertos basados en reglas (*rule-based systems*) con el siguiente aspecto: «SI esto ENTONCES aquello».

Como deja adivinar el término «fuzzy», estas reglas están además empañadas por una lógica difusa (mediante la incorporación de la noción de conjuntos difusos, los «*fuzzy sets*», que abordaremos más adelante).

Un conjunto difuso puede verse como la pareja formada por un conjunto «clásico» (en inglés «crisp») y una función sobre los elementos de este conjunto tomando los valores en cierto intervalo, típicamente [0,1]. Esta función cuantifica la verosimilitud de una aserción (es, de hecho, el nivel de pertenencia al conjunto, una extensión de la noción de «función característica» que vale 0 cuando un elemento no está en el conjunto y 1 cuando el elemento está en el conjunto).

La facilidad de la interpretación de las reglas por parte de los humanos ayuda a ajustar estos sistemas, en particular si el número de reglas es razonable. Es posible codificar un conocimiento incierto, lo que mejora el diálogo con los expertos, pues no es necesario captar todo su conocimiento acerca de un determinado asunto a través de una sola regla muy compleja y expresada una sola vez. Por el contrario, se acumularán reglas unitarias indicando su incertidumbre, si bien la validación individual de estas se realizará fácilmente. La base que contiene las reglas podrá evolucionar por incrementos sucesivos, sin desestabilizar el conjunto.

El primer inconveniente de estos sistemas reside en la necesidad de disponer de expertos a los que consultar para definir las reglas, lo que introduce cierto sesgo y una especie de rigidez de cara a las situaciones imprevistas.

El segundo inconveniente resulta ser el rápido crecimiento del número de reglas cuando aumenta el número de variables de estos sistemas. La solución consiste en tratar de dividir el conocimiento en silos de reglas con poca interacción entre sí.

 Se denomina «blackboard» a los espacios de memoria a través de los cuales los silos de reglas comparten información, que denominamos «hechos».

Sin embargo, estos sistemas merecen figurar entre las herramientas del data scientist, pues además existen mecanismos que permiten descubrir automáticamente nuevas reglas no expuestas a primera vista por los expertos (a quienes a menudo resulta útil consultar para una validación final!).

Algunos elementos de lógica

Para codificar de forma correcta el conocimiento, es necesario disponer de una cierta lógica, en particular si queremos producir automáticamente nuevos hechos por un sistema de reglas. En efecto, este sistema no podrá suplir las carencias de nuestro diseño (a diferencia de un humano, que en ocasiones puede identificar incoherencias en las aserciones acerca de diversos asuntos sobre los que todavía no posee competencias o está mal informado).

El universo de un discurso correspondiente a los conjuntos o a las aserciones que queremos manipular debería

definirse, o al menos comprenderse:

- ¿Estamos manipulando un conjunto finito de palabras, el conjunto de todas las palabras de un vocabulario, conceptos identificados en una ontología precisa, enteros positivos (y solo enteros positivos), enteros positivos vistos como elementos particulares del conjunto de números reales...?
- ¿Estamos manipulando conceptos instanciados en una cierta realidad, es decir, objetos existentes en esta realidad? Por ejemplo, es distinto razonar sobre los atributos de las naves espaciales en la literatura de ciencia-ficción (instanciadas, numerables, con pocos atributos), que razonar sobre la representación que realizan los niños (no instanciadas, no numerables, con muchos atributos mal determinados) o razonar sobre el universo de naves ya lanzadas por la NASA (instanciadas, numerables, con infinidad de atributos posibles pero técnicamente bien determinables).

Supongamos que hayamos identificado dos conjuntos «clásicos» dentro del universo del discurso del que disponemos. Llamemos a los conjuntos A y B.

Estos conjuntos y sus interacciones pueden tener diversas modalidades; las más simples son:

- Un conjunto puede estar vacío (\emptyset).
- Un conjunto puede ser igual a todo el universo del discurso (que llamaremos \mathbb{D}).
- Un conjunto puede poseer una definición o una descripción por extensión (lista de elementos), lo que le confiere una «existencia» y, por lo tanto, justifica el uso de sus nombres respectivos A y B.
- Es posible definir el conjunto de lo que no está en un conjunto determinado, intersecciones y uniones de conjuntos...

Si abordamos cada uno de estos conjuntos a través de su definición, del tipo:

$$A = \left\{ \begin{array}{l} \text{conjunto de elementos del universo del discurso tales que} \\ \text{la aserción aserción}_A \text{ sea verdadera} \end{array} \right\}$$

entonces es fácil dirigirse a las propias aserciones y enunciar definiciones como:

$$A \cap B = \left\{ \begin{array}{l} \text{conjunto de elementos del universo del discurso tales que} \\ \text{la aserción aserción}_A \text{ sea verdadera} \\ \text{y} \\ \text{la aserción aserción}_B \text{ sea verdadera} \end{array} \right\}$$

$$A \cup B = \left\{ \begin{array}{l} \text{conjunto de elementos del universo del discurso tales que} \\ \text{la aserción aserción}_A \text{ sea verdadera} \\ \text{o} \\ \text{la aserción aserción}_B \text{ sea verdadera} \end{array} \right\}$$

$$\neg A = \left\{ \begin{array}{l} \text{conjunto de elementos del universo del discurso tales que} \\ \text{la aserción aserción}_A \text{ sea falsa} \end{array} \right\} = \text{not } A$$

Avanzamos de manera natural hacia una notación booleana que se refiere directamente a las aserciones, y la aserción_A se convierte en A «simplemente».

Por otro lado, $A \cap B$ se convierte en $A \wedge B$: $A \wedge B$

y $A \cup B$ se convierte en $A \vee B$: $A \vee B$

Algunos ejemplos de aserciones:

x tiene los ojos azules (que el atributo «ojos» del elemento x de \mathcal{D} es igual a «azul»).

x es un entero par no nulo: $\exists k \in \mathbb{N}^* \mid x = 2k$.

La función f aplicada a x produce un resultado superior a 7 (es decir: $f(x) > 7$).

Tras recopilar la información correspondiente a las dos aserciones, las distintas modalidades que deberíamos considerar deberían determinarse entre las modalidades evidentes siguientes:

$A, B, \neg A, \neg B, A \wedge B, A \vee B, \neg A \wedge B, A \wedge \neg B, \neg A \vee B, A \vee \neg B, \neg A \wedge \neg B, \neg A \vee \neg B$.

a las que no debemos olvidar agregar las dos modalidades siguientes:

A si y solo si B (que corresponde a la equivalencia \Leftrightarrow).

A xor B (el O exclusivo, que excluye que A y B sean ambos verdaderos a la vez, que en ocasiones se escribe $A \oplus B$).

 Para ser completos, también habría que evocar los dos casos límites:

$\neg A \vee A, \neg B \vee B$ siempre verdaderos en lógica clásica y que valen \top .

$\neg A \wedge A, \neg B \wedge B$ jamás verdaderos en lógica clásica y que valen \emptyset .

La mecánica calculatoria y un cierto número de teoremas de esta lógica «clásica» se respetan a nivel global en las expresiones de lógica difusa que manipulan los conjuntos difusos definidos de la siguiente manera:

$$A = \{(x, \mu_A(x)) \mid x \in \mathcal{D}\}$$

En esta definición, μ_A es una función que permite determinar el nivel de pertenencia al conjunto difuso (en inglés, «membership function»).

Características y vocabulario de la lógica difusa

A partir de las funciones de pertenencia, es posible definir mecánicas propias de la lógica difusa para implementar los conceptos fundamentales de la lógica: la disyunción («o»), la conjunción («y»), la negación, la implicación, la equivalencia entre conjuntos o aserciones difusas... todo esto sin perder toda la simplicidad de uso de estas aserciones y sin renunciar a todos nuestros hábitos calculatorios.

Esta es la belleza de la lógica difusa: vamos a poder mantener gran parte de nuestros hábitos de lógica y utilizar una mecánica calculatoria parecida a la de la lógica clásica, aunque de hecho estemos manipulando una noción mucho más rica y compleja.

El hecho de no contentarse con una lógica binaria {0,1}, sino trabajar sobre un intervalo continuo, nos permite modular las nociones de pertenencia a un conjunto o de veracidad de una aserción.

Cuando el intervalo-imagen de la función de pertenencia es [0,1], los tres niveles de pertenencia posibles a un conjunto difuso A se definen de la siguiente manera (consideremos un elemento x de \mathcal{D}):

- Sin pertenencia si $\mu_A(x) = 0$.

- Miembro difuso si $0 < \mu_A(x) < 1$.
- Miembro soporte (se llama *core member* o perteneciente al *kernel*) si $\mu_A(x) = 1$.

La notación más habitual para designar un miembro de un conjunto difuso es un poco extraña, habríamos esperado una evocación de la pareja $(x, \mu_A(x))$. De hecho, la notación para un miembro del conjunto difuso A es $\mu_A(x)/x$, la barra no representa en absoluto una división!

Para evaluar el aspecto de la función de verosimilitud μ_A , es habitual recopilar y estudiar una colección de elementos de A, preguntándose por ejemplo para cada elemento: ¿es un miembro que verifica la aserción del conjunto difuso?

- ¿Lo es por naturaleza/definición, es una verdad medible, es una probabilidad, una aserción plausible... es una convicción, una creencia?
- ¿Este nivel de pertenencia es nulo, débil, medio, fuerte, certero?

Para recrear una lógica que respete más o menos las formas de la lógica habitual, los diferentes expertos proponen mecanismos de composición de funciones de pertenencia que son similares, pero sensiblemente diferentes. Para evitar interpretaciones incorrectas de nuestros resultados, es muy importante identificar los mecanismos subyacentes a la lógica difusa utilizada en las soluciones empaquetadas que queremos implementar.

Si queremos mantenernos cerca de la lógica clásica, el sistema utilizado más habitualmente es este:

$$\mu_{A \cap B}(x) = \min(\mu_A(x), \mu_B(x))$$

$$\mu_{A \cup B}(x) = \max(\mu_A(x), \mu_B(x))$$

$$\mu_{\neg A}(x) = 1 - \mu_A(x)$$

Si queremos tener un comportamiento clásico pero próximo a cómo se componen las probabilidades, optamos a menudo por el siguiente sistema:

$$\mu_{A \cap B}(x) = \mu_A(x) \cdot \mu_B(x)$$

$$\mu_{A \cup B}(x) = \mu_A(x) + \mu_B(x) - \mu_A(x) \cdot \mu_B(x)$$

$$\mu_{\neg A}(x) = 1 - \mu_A(x)$$

Para ilustrar el aspecto que podría tener una función de pertenencia, he aquí un ejemplo de definición de dicha función.

Consideramos aquí un caso donde:

- mientras el valor de x no alcance el valor «a» (excluido), se estima que la aserción es falsa y que por lo tanto el elemento no pertenece al conjunto difuso,
- para x de «a» (incluido) a «b» (excluido), se considera que el elemento es un miembro difuso,
- a partir de «b» (incluido), el elemento es cierto, es decir, forma parte del conjunto de soporte del conjunto difuso.

$$\mu_A(x) = \begin{cases} 0 & x < a \\ \frac{x-a}{b-a} & a \leq x < b \\ 1 & x \geq b \end{cases}$$

Por ejemplo, esta función de pertenencia podría modelar la noción de *adulto responsable*, con $a = 14$ años (edad mínima para trabajar en prácticas) y $b = 24$ años (edad mínima para ser senador).

Podemos hacer crecer o decrecer más o menos rápidamente la tasa de pertenencia entre 0 y 1 (es decir, en nuestro ejemplo, se consideraría más o menos rápidamente responsable entre 14 y 24 años) utilizando una función menos lineal que $\frac{x-a}{b-a}$. Un uso habitual consiste en concentrar o dilatar $\mu_A(x)$ considerando otro conjunto difuso B, tal que $\mu_B(x) = \mu_A(x)^2$ o tal que $\mu_B(x) = \sqrt{\mu_A(x)}$.

El siguiente código traza esta función de pertenencia:

```
# ejemplo de función membership ##

mu <- function(x,a,b){
  if      ((a >= b) || (x < a)) { return (0) }
  else if (x >= b)    { return (1) }
  else                  { return ((x-a)/(b-a)) }
}

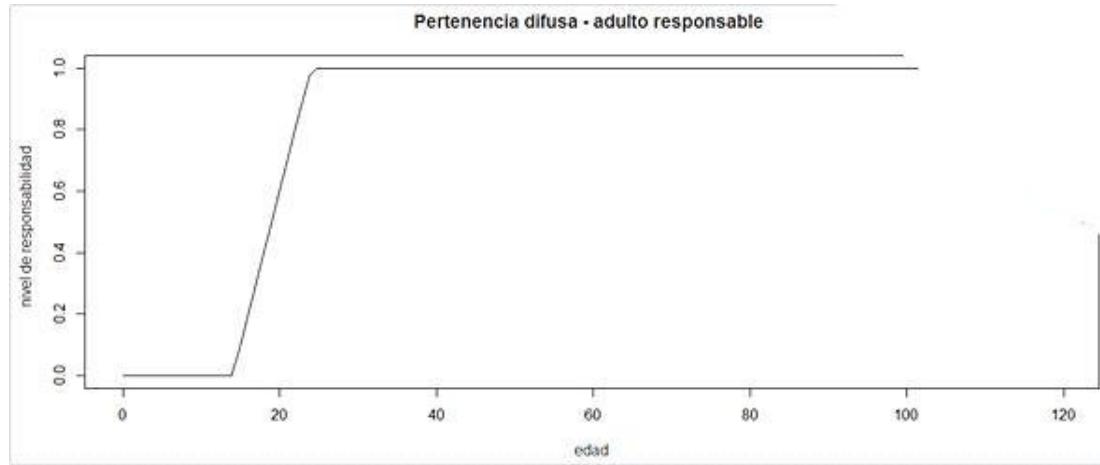
mu(14 ,14,24)    # test dato 0
mu(14.1,14,24)   # test dato 0.01
mu(23.1,14,24)   # test dato 0.91
mu(24 ,14,24)    # test dato 1

#-----#
# plot de la función membership para a=14 y b=24 ##

x <- seq(0,120,0.99)          # edad de 0 - 120 años
y <- sapply(x,
            function(x) {mu(x,14,24)})  # imagen de x

plot(x,y,
      main = "Pertenencia difusa - adulto responsable",
      xlab = "edad",
      ylab = "nivel de responsabilidad",
      type = "l",
      col  = "black")
```

Lo que se corresponde con una curva utilizada con mucha frecuencia:

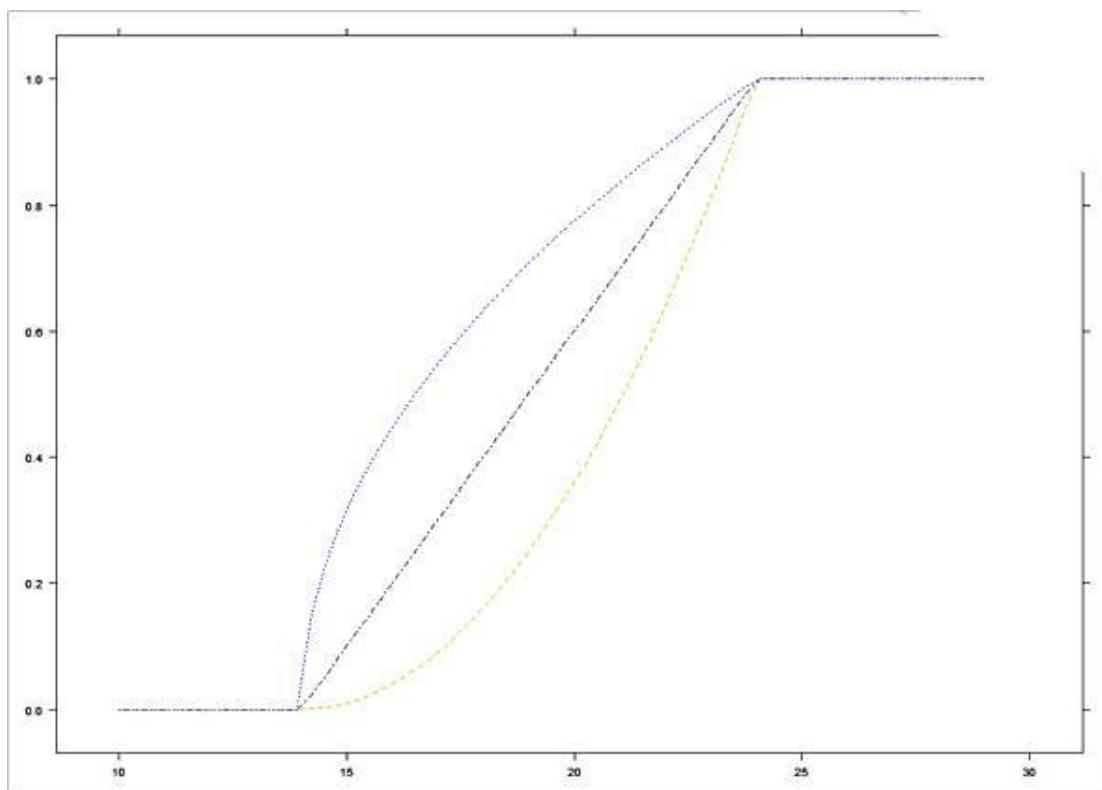


Función de pertenencia con transición lineal

Como hemos indicado en la observación anterior, podemos deformar esta función para que se corresponda mejor con nuestra percepción de la manera en la que se pasa de la no-pertenencia a la parte miembro difuso, o de la parte miembro difuso a la parte de soporte. En el siguiente código, se utiliza la librería *lattice* para realizar representaciones una vez realizada una concentración o dilatación:

```
## plot de la función membership sobre 10-30 años ##  
## comparación dilatación normal y concentración ##  
  
library("lattice") # librería gráfica  
x <- seq(10,30,0.3) # edad de 10 - 30 años  
  
normal <- sapply(x,  
                  function(x) {mu(x,14,24)})  
concentracion <- sapply(x,  
                        function(x) {mu(x,14,24)^2}) # concentración: ^2  
dilatacion <- sapply(x,  
                      function(x) {mu(x,14,24)^(1/2)}) # dilatación: ^^(1/2)  
  
mis_datos <- data.frame(x,normal,concentracion,dilatacion)  
  
#-----#  
## construcción de los gráficos ##  
t0 <- xyplot(normal ~ x ,data = mis_datos,  
              col="black", type = "l",lty = 4,  
              xlab="",ylab="",  
              aspect =0.7)  
t1 <- xyplot(concentracion ~ x ,data = mis_datos,  
              col="orange", type = "l",lty = 2,  
              xlab="",ylab="",aspect =0.7)  
t2 <- xyplot(dilatacion ~ x ,data = mis_datos,  
              col="blue", type = "l",lty = 3,  
              xlab="",ylab="",aspect =0.7)  
  
plot(t0) # plot gráfico normal  
plot(t1,newpage = FALSE) # superponer concentración  
plot(t2,newpage = FALSE) # superponer dilatación
```

Obtenemos las tres curvas sobre un mismo gráfico:



Función de pertenencia concentrada, normal, dilatada

La curva superior representa una transición brutal a los 14 años entre el estado no-adulto-responsable y un nivel rápidamente creciente de pertenencia al conjunto de los adultos responsables. A la inversa, la curva inferior representa una transición suave pasados los 14 años, y a continuación una aceleración cada vez más rápida a partir de los 18 años. Como ocurre a menudo, se confirma que las elecciones matemáticas del data scientist están lejos de ser neutras.

Enjambre (swarm)

Las analogías con la naturaleza inspiran la creación de algoritmos. Cuando se observa una colonia de hormigas (*ant*) que van a buscar alimento sobre un mantel tras un almuerzo en el campo, nos sorprende la «inteligencia colectiva de las hormigas». Estas se organizan en filas que van hacia los puntos donde encuentran la mayor cantidad de recursos, exploran con eficacia el territorio, cambian su comportamiento y su organización cuando se agregan obstáculos y cambian su organización cuando un recurso escasea en algún lugar del mantel.

La idea de los algoritmos que utilizan esta analogía es sencilla: no existe ninguna inteligencia centralizada en estos comportamientos, pero el conjunto de comportamientos fáciles de codificar (es decir, poco inteligentes) más los breves intercambios de información le proporcionan una inteligencia colectiva al enjambre (*swarm*).

1. Swarm y optimización: el algoritmo PSO

El uso más habitual de estos algoritmos es la búsqueda de óptimos en el caso de problemas muy difíciles, típicamente cuando existen muchos mínimos locales. Observe que el hecho de encontrar estos óptimos locales puede formar parte de nuestros objetivos, del mismo modo que nuestras hormigas solo buscan la fuente de alimento más grande, pero todas las fuentes de alimento notables deben estar accesibles a cierta distancia de su base.

a. Presentación de PSO

El algoritmo PSO (*Particle Swarm Optimization*) implementa una población de partículas que se desplazan de manera estocástica en el espacio de búsqueda. Cada individuo se caracteriza por su mejor rendimiento en este espacio (por ejemplo, allí donde la hormiga encuentra la mayor cantidad de alimento, y no allí donde se encuentra realmente la mayor cantidad de alimento). El individuo comunica esta experiencia a parte de la población (en función de sus medios de comunicación y, de hecho, según su voluntad de programador). En ocasiones resulta más conveniente comunicar la información solo a un conjunto restringido de la población para no deformar el comportamiento de todo el enjambre.

Esta información influye en el comportamiento de los demás individuos informados, que van a compararla con la información proporcionada por cada uno de sus interlocutores y con su propia información. La dirección y la velocidad de su próximo movimiento se deducen de ello.

En la introducción, hemos evocado una búsqueda de óptimo (método del gradiente) dando como ejemplo un niño que busca huevos de pascua. Aquí tenemos una tropa de niños que colaboran anunciando lo que han encontrado y dónde.

b. Descripción de PSO

Se trata de un juego por turnos: con cada turno pasa el tiempo de t a $t + 1$, de modo que un delta de tiempo dt siempre será igual a 1.

El vector posición de la partícula i en el instante t es x_t^i . Esto se corresponde simplemente con los vectores de nuestras variables explicativas para la partícula i en el turno t .

La velocidad de la partícula en el instante $t + 1$ será: $v_{t+1}^i = \frac{x_{t+1}^i - x_t^i}{dt} = x_{t+1}^i - x_t^i$.

Preste atención: es un **vector** de velocidad, que contiene la velocidad sobre cada uno de los ejes del problema y que representa de hecho la **dirección** en la que se desplaza la partícula.

De hecho, hablamos más a menudo de *search velocity* (velocidad de búsqueda) que de velocidad, en referencia al

problema que tratamos de resolver.

En el turno anterior, la partícula ya había tomado una decisión, que no estaba necesariamente desprovista de sentido. Se asigna una **inercia** a la partícula que genera una nueva dirección como la resultante de su trayectoria anterior y de la trayectoria inducida por la nueva información. El coeficiente en cuestión evolucionará en el tiempo (a la baja) para permitir a la partícula «aterrizar» sobre un óptimo: en cada turno está más atenta a la información de los demás; además, en cada turno la información transmitida por el grupo debería ser más adecuada, pues el grupo ha explorado una superficie mayor del territorio.

Este coeficiente de inercia se escribe: w_t (*weight* = peso). Observe que aquí hemos seleccionado una función de inercia común a todas las partículas.

Si la partícula se contentara con seguir siempre el mismo camino, sin ninguna inteligencia, su trayectoria estaría determinada por:

$$v_{t+1}^i = x_{t+1}^i - x_t^i = w_t v_t^i$$

Lo que significaría que el vector velocidad en $t + 1$ es w_t veces el vector velocidad en t y que la partícula irá siempre en línea recta, pero con una velocidad decreciente, pues w_t es decreciente.

Para introducir el conocimiento que posee la partícula, hay que agregar a este vector velocidad una componente de velocidad (y, por tanto, de dirección) que tiene en cuenta su conocimiento y que va a deformar su trayectoria para aumentar su probabilidad de encontrar un óptimo. Para ello, se orienta esta nueva componente de la velocidad en la dirección del mejor rendimiento pasado de nuestra partícula (donde la hormiga haya encontrado más alimento hasta el momento). Llamaremos p^i a esta posición, que se almacenó cuando la variable era la mejor; en aquel momento teníamos: $p^i \leftarrow x_{turno}^i$.

El componente de velocidad (y, por tanto, de dirección) correspondiente es:

$$\frac{p^i - x_t^i}{dt} = p^i - x_t^i$$

Para que la partícula tenga en cuenta la información recibida de las demás partículas respecto a su mejor rendimiento, se construye una componente de velocidad similar, pero utilizando p^{grupo} , es decir, el lugar donde el grupo ha encontrado un mayor rendimiento.

$$p^g - x_t^i$$

La partícula debe integrar ambas informaciones, y a continuación podemos ponderar sus respectivas influencias. Es habitual llamar a estas dos ponderaciones *cognitive scaling parameter* y *social scaling parameter*, que expresa que la primera componente se corresponde con la confianza de la partícula en su propio conocimiento, y la segunda, con su confianza en el conocimiento adquirido por el grupo. Llamaremos a estos dos coeficientes c_1 y c_2 respectivamente.

Podemos introducir algo de azar a todo esto para que la partícula pueda tener cierta posibilidad de encontrar recursos nuevos y también para que no se quede «atascada» en el mismo lugar. Para ello, multiplicamos nuestras componentes de velocidad vinculadas al conocimiento por dos números aleatorios comprendidos entre 0 y 1, que cambian con cada turno y que llamaremos $r_{1,t}$ y $r_{2,t}$ (de *random*).

Sumando todas estas componentes de velocidad (y, por tanto, de dirección), la trayectoria de la partícula i de un turno al otro viene determinada por la ecuación:

$$x_{t+1}^i = x_t^i + v_{t+1}^i$$

con :

$$v_{t+1}^i = x_{t+1}^i - x_t^i = w_t v_t^i + c_1 r_{1,t} (p^i - x_t^i) + c_2 r_{2,t} (p^g - x_t^i)$$

Para obtener nuestro algoritmo PSO, basta con implementar todo esto en un doble bucle (los turnos y las partículas), agregar la inicialización de las posiciones, calcular los valores de la función que hay que optimizar y sus máximos para cada individuo en cada turno, y encontrar a continuación los criterios de parada del algoritmo y una función w decreciente inteligente.

2. Puesta en práctica de PSO

Vamos a buscar el mínimo de una función clásica: la función de Rastrigin (por el nombre de su creador). Esta función comprende muchos óptimos locales y resulta difícil de optimizar mediante ciertos algoritmos de optimización.

En primer lugar se crea la función y a continuación se adapta su formato al paquete **pso**.

```
library(pso)
library(ggplot2)

d=2                                # función a optimizar
                                    # 2 dimensiones
f_ <- function(x1,x2){            # función de Rastrigin
  d * 10+x1^2 + x2^2-10*(cos(2*pi*x1)+cos(2*pi*x2))
}

f <- function(x = c(0,0)){        # su versión vectorial
  f_(x[1],x[2])
}
f(c(5,5))                          # ejemplo de uso
```

El algoritmo requiere la definición de un cierto número de parámetros de control, y a continuación su ejecución se realiza de manera sencilla.

```
# los extremos
# clásicos de Rastrigin
min_x1 <- - 5.2
min_x2 <- - 5.2
max_x1 <- 5.2
max_x2 <- 5.2
maxit=100 # número máximo de iteraciones
s=40      # número de partículas

control =list(trace=1,      # modo traza
             maxit=maxit,
             REPORT=1,      # número de informes
             trace.stats=1,
             s = s
           )
```

```

set.seed(15)
          # optimización
p <- psoptim(rep(NA,d), # tantos NA como d
              fn=f,
              lower=c(min_x1,min_x2),
              upper=c(max_x1,max_x2),
              control=control)

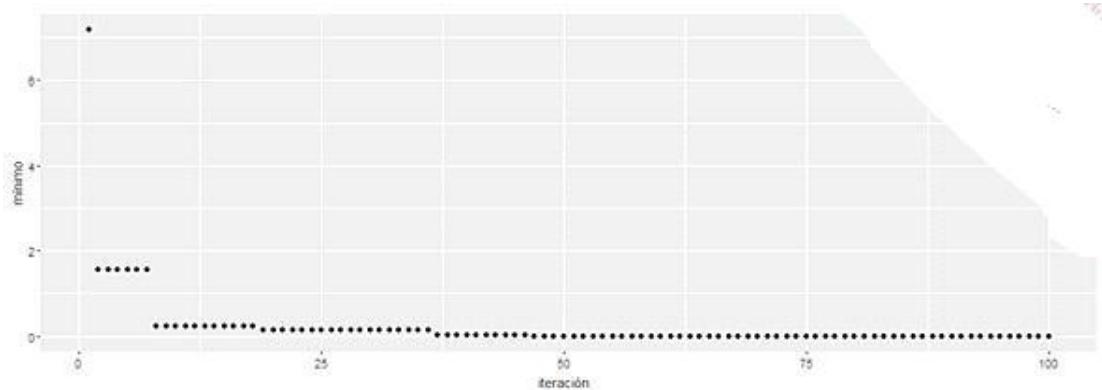
qplot(p$stats$it,p$stats$error,
      xlab= "iteración",ylab = "mínimo")

cat("Coord óptimo:",round(p$par,2),
    "valor de la función correspondiente",
    round(p$value,2),"\n")

```

Coord óptimo: 0 0 valor de la función correspondiente 0

Las coordenadas del óptimo son (0,0) y su valor es 0, lo que se corresponde con los valores teóricos.



Evolución hacia el mínimo en función del número de iteraciones

El valor mínimo se obtiene tras menos de 50 iteraciones. Visualicemos la función y el mínimo encontrado.

```

x1_ <- seq(min_x1, max_x1, by=0.2) # mallado x1
x2_ <- seq(min_x2, max_x2, by=0.2) # mallado x2
y_ <- outer(x1_, x2_, f_)           # un valor por pareja

palette <- colorRampPalette(c("red","yellow"))(100)
diag <- persp(
            x1_,                                # perspectiva 3D
            x2_,
            Y_,
            phi=-10,                             # ángulos de presentación
            theta=60,
            col=palette[round((y_+max(y_))/(max(y_)-min(y_))* 55)],
            ticktype = "detailed",                # eje detallado
            shade=.001,

```

```

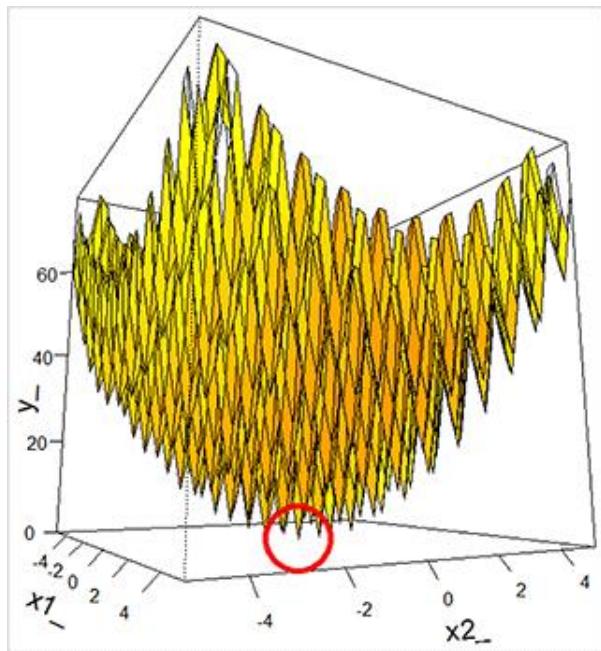
cex.lab=1.5 )

u <- as.vector(p$par[1])
v <- as.vector(p$par[2])
w <- as.vector(p$value)

nube <- trans3d(u,
                  v,
                  v,
                  pmat = diag)

points(nube,
        # inserción en el diagrama
        pch = 1,      # símbolo
        cex = 8,      # tamaño
        lwd = 5,      # grosor del trazo
        col = 2       # color
)

```



Mínimo de la función de Rastrigin

La función de Rastrigin contiene efectivamente numerosos mínimos locales, pero el algoritmo ha convergido con éxito hacia el lugar de su mínimo global.

Feature Engineering

Feature Engineering, los fundamentos

Con menos de 100 000 entradas en Google, el Feature Engineering, que podríamos traducir con un poco de imaginación como *Ingeniería de los datos*, podría parecerse al pariente pobre del machine learning y de sus 20 millones de enlaces. Sin embargo, representa a menudo más de la mitad del esfuerzo que debe realizarse en un proyecto de data sciences.

1. Definición del problema

El Feature Engineering, que hace referencia a las técnicas de ingeniería utilizadas para trabajar sobre los atributos del problema que estudiamos, representa uno de los aspectos más delicados de las data sciences.

En efecto, las características de un problema determinado no siempre se presentan de manera natural y evidente.

Hace falta bastante esfuerzo para imaginar cuáles podrían ser los atributos adecuados, crearlos, seleccionarlos, probarlos y transformarlos para hacerlos relevantes y asimilables por nuestros algoritmos. Las buenas prácticas requieren buenos atributos (en inglés, *predictors* o *features*).

El trabajo en la elección y la parametrización de los modelos, la medición de los errores de predicción (o de clasificación) y la atenta interpretación de los resultados son tres elementos indisociables de la disciplina llamada Feature Engineering.

La idea general es ajustar y transformar los datos brutos (*data*) para que nos aporten una mayor información. Matemáticamente, esta ganancia se corresponde con una disminución de la entropía. De algún modo, la adquisición de información significativa y utilizable disminuye el desorden del ambiente mediante nuestra comprensión del fenómeno estudiado.

El primer escollo es, con frecuencia, la determinación del propio objetivo del estudio! En efecto, la cuestión que se plantea no siempre es completamente explícita y se traducirá a menudo el objetivo del estudio en términos reduccionistas, pero bastante simples para realizar una predicción (por ejemplo, sería difícil pretender predecir la satisfacción de los clientes sin tener la menor idea de cómo medirla).

2. Sobre qué hay que estar muy atento

La eficacia de los algoritmos depende en gran medida de la naturaleza y de la calidad de los datos que se utilizan. En cierta manera, la negligencia es particularmente contraproducente!

a. La calidad de la distribución

Sea \mathbf{X} , una matriz de m columnas (las m features) y un vector \mathbf{y} (el objetivo que se ha de predecir); se espera entonces que las n filas ($\mathbf{x}_i, \mathbf{y}_i$) sean independientes y estén idénticamente distribuidas (decimos **i.i.d.**), en particular respecto a los problemas de clasificación.

Esto significa que provienen de la misma ley de probabilidad (de ahí el término **idénticamente distribuidas**).

¿Qué hacer si los datos no son i.i.d.?

A menudo se indica que se dispone de una mezcla de varias muestras extraídas de contextos muy diferentes. La mención de esta diferencia de contexto puede convertirse en una nueva feature. Por ejemplo, si estudiamos la producción de tornillos para máquinas de diferentes tecnologías, resultará natural crear una feature **tipo_de_maquina**.

¿Qué hacer si las filas no son independientes?

Cuando las filas (X_i, y_i) están condicionadas por otras filas, se deduce que no son independientes. Creando un orden entre los eventos dependientes podemos, a menudo, obtener una feature que hará que las filas sean independientes, pues ya no dependerá la una de la otra, sino por el contrario del valor del nuevo parámetro externo (por ejemplo, un tiempo, una secuencialidad o una causalidad).

En ocasiones, simplemente tenemos una serie temporal que no habíamos identificado como tal, y bastará con dotarse de las herramientas correspondientes a las **Times Series** (series temporales).

b. La naturaleza de las features

Hay que entender la naturaleza de las features y codificarlas correctamente respecto a los algoritmos que se pretende utilizar. Por ejemplo, no habría que explotar accidentalmente un orden ficticio entre los tres colores **Celeste, Magenta y Amarillo** por haberlos codificado respectivamente **1,0, 2,0 y 3,0** y utilizar a continuación un algoritmo que explotara de manera falaz este aparente orden de relación.

Los distintos tipos de atributos que hay que codificar se reparten de la siguiente manera:

- Variables categóricas nominales, por ejemplo:
 - mujer, hombre
 - celeste, magenta, amarillo
 - sí, no
- Variables categóricas ordinales, por ejemplo:
 - no está de acuerdo, indiferente, está de acuerdo
 - sobre el pódium, en la cabeza del pelotón, en la cola del pelotón
- Variables cuantitativas de intervalos, por ejemplo:
 - años (de -2000 a + 3000)
 - nivel respecto al mar
- Variables cuantitativas de razón, por ejemplo:
 - recuento de la población de las provincias españolas
 - consumo de electricidad en kW

El uso de estas naturalezas de features aparece rápidamente con la práctica, pero merece la pena que lo observemos con atención.

Variables categóricas nominales

Se trata de clases con nombre y un número finito de ocurrencias (por ejemplo: género, país...).

Hay que verificar que no existe ninguna relación de orden implícita entre las ocurrencias (un ejemplo de relación de orden implícito sería: frío, tibio, caliente...), pues podría perderse información respecto a una mejor codificación realizada con alguno de los siguientes tipos (ordinales).

Variables categóricas ordinales

Hablamos aquí de variables ordenadas mediante un orden total, es decir, que cada valor puede compararse con los demás (por ejemplo, los sentimientos amorosos: mucho - un poco - en absoluto).

Hay que entender bien que conocer la medida precisa de la distancia entre dos valores sucesivos no debería tener ningún significado, y que no deberíamos comparar dos distancias.

En caso de que el orden no sea total, puede existir un orden parcial. Habrá que volver a enunciar y utilizar variables nominales, o bien crear nuevas variables ordinales para cada rama, que esté totalmente ordenada.

Para ilustrar este punto, consideremos la siguiente jerarquía: **jefe de grupo - jefe de la entidad 1 - adjunto jefe de la entidad 1 - jefe de la entidad 2 - adjunto al jefe de la entidad 2**.

En esta jerarquía, no es posible saber si alguno de los dos adjuntos es superior al otro. Se creará una feature **nivel_jerárquico_rama_1** y una feature **nivel_jerárquico_rama_2**. En esta segunda feature, el adjunto de la rama 1 no aparecerá (se dice que el valor de su feature es **no disponible** o **NA** en inglés).

Variables cuantitativas de intervalos

Se trata de variables ordinales cifradas que se refieren a una posición identificada en una recta cuyo origen es convencional (por ejemplo: 1 000 años antes de JC - JC - 2017 después de JC).

Aquí, las distancias entre dos valores sucesivos tienen el mismo significado, o al menos el mismo sentido. Es posible calcular y comparar las distancias. Estas variables pueden ser discretas (por ejemplo, fechas) o continuas (por ejemplo, una posición en una ruta).

Si duda entre intervalo y razón, plantee la siguiente comprobación:

¿Tendría sentido multiplicar este valor por 2?

Si la respuesta es que sí, esta variable cuantitativa estará mejor codificada como variable cuantitativa de razón.

Variables cuantitativas de razón

Representan cantidades medibles y disponen de un cero absoluto (por ejemplo, la masa de un objeto).

En ocasiones, este cero absoluto es un límite (es decir, que no es estrictamente alcanzable). Estas variables pueden ser discretas (por ejemplo, un número de personas) o continuas (por ejemplo, una distancia, una frecuencia).

Cuando se calcula el valor absoluto de las distancias de una variable de intervalos, se obtiene una variable de razón (por ejemplo, la distancia entre fechas se convierte en una duración, que es una variable cuantitativa de razón).

Si aceptamos una pérdida de información, una variable de razón puede manipularse como una variable de intervalos.

3. Dominar la dimensionalidad

El hecho de estar atento a la naturaleza y a la calidad de nuestros datos es importante, pero resulta, por desgracia, insuficiente en muchos casos prácticos. La realidad que tratamos de analizar se disimula con frecuencia detrás de

las debilidades de nuestro muestreo y de los límites de nuestra tecnología de cara a las dimensiones de nuestro modelo.

- En el contexto del Big Data, el analista dispone a menudo del conjunto de la población padre, que permite pensar que esta muestra es más o menos perfecta. Esto genera una peligrosa ilusión en todos los casos donde el fenómeno estudiado es temporal. En efecto, nuestra población completa no refleja necesariamente de manera homogénea todos los casos posibles significativos que caracterizan la realidad de nuestro estudio. Es decir, ¿que ciertos eventos interesantes y significativos simplemente todavía no han tenido lugar?

El problema planteado

El hecho de disponer de un gran número de features es bastante más molesto de lo que podríamos llegar a pensar. La dificultad de los algoritmos que permiten trabajar con el conjunto de dimensiones no aumenta, en efecto, de manera proporcional con el número de dimensiones.

Por otro lado, un número demasiado grande de features genera, en ocasiones, un sobreajuste (overfitting).

The curse of dimensionality: sparse matrix

Esto se ha enunciado como una expresión muy imaginativa: **la maldición de la dimensionalidad** o, en inglés: **the curse of dimensionality**.

Como primera aproximación, el problema proviene de que hace falta una muestra cada vez más grande para realizar una predicción sobre un número de dimensiones también creciente, pues existen menos posibilidades de encontrar en la naturaleza todos los casos posibles que ilustren la combinación de las distintas dimensiones. Nuestra muestra se vuelve rápidamente incompleta respecto al número de dimensiones, un poco como ocurre con las estrellas dispersas en el espacio (en inglés, se habla de *sparse matrix*).

Agregar features es, en ocasiones, contraproducente, pues rápidamente faltarán ejemplos disponibles que permitan entrenar nuestro algoritmo de predicción o de clasificación. Encontramos discusiones detalladas acerca del número mínimo de filas necesarias (n) para un número de features (m) con el objetivo de entrenar un algoritmo determinado.

The curse of dimensionality: hiperespacio

Otro aspecto de este problema proviene de la topología de los hiperespacios que comprenden numerosas dimensiones.

Imaginemos una superficie cuadrada, de longitud de lado igual a 1, sobre la que estuviera diseminada toda nuestra población en función de sus valores sobre dos ejes (es decir, dos features).

Supongamos ahora que un individuo central pudiera identificar a todos sus vecinos realizando una única acción cuando están a una distancia inferior a 0,5... o pongámonos en el lugar de un pescador que tuviera a su disposición una pequeña redecilla con un mango de longitud 0,5 metros situado en el centro de un mar cuadrado de 1 metro de lado.

Los vecinos en cuestión están en un círculo circunscrito al cuadrado. Mediante esta acción, el individuo central es capaz de identificar una gran parte de la población ($\text{superficie_del_círculo}/\text{superficie_del_cuadrado} = \pi/4$), lo que equivale a cerca del 79 % de la población. La redecilla no le permitirá encontrar aquellos vecinos escondidos en las esquinas, $100-79 = 21$ % de los individuos se le escaparán.

Si el individuo se plantea el mismo problema en dimensión 3, es decir, una esfera circunscrita en un cubo, la relación

entre ambos volúmenes es del 53 %. Con la misma redonda, el pescador deja ahora escapar a $100-53 = 47\%$ de sus vecinos.

En siete dimensiones (siete features), nos quedamos en un 5%! La relación entre el volumen del hipercubo y el de la hiperesfera disminuye rápidamente y tiende a 0!!!

¡El pescador deja escapar $100-0 =$ a casi el 100 % de sus vecinos!

Rápidamente, nuestro individuo tendrá enormes problemas para encontrar a sus vecinos con una única acción... incluso con varias.

Si recordamos el nombre de uno de los algoritmos insignia de las data sciences, a saber, la **Búsqueda de los k vecinos más cercanos/KNN**, nos entran escalofríos y comprendemos de repente por qué ciertas librerías de R bloquean el número máximo de features a 32.

Feature ranking y selección

La idea aquí es muy simple, se clasifican las features por eficacia predictiva y mantenemos únicamente las primeras. La gracia está en encontrar el equilibrio adecuado entre el número de features conservadas y la eficacia total del modelo. Pero esto plantea una pequeña complicación, en efecto, ciertas features no funcionan solas, sino de manera combinada con otras 2, 3, ..., k features.

A menudo se utiliza el algoritmo Random Forest, que como producto derivado proporciona una lista muy interesante de las features clasificadas según su nivel de importancia... que se traduce por **importance** en inglés.

Las personas que se ven presionadas utilizan también un cálculo rápido de coeficiente de correlación y eliminan todas las features que poseen correlaciones muy débiles con las demás. Este método trivial y aparentemente seductor resulta a menudo erróneo debido a la complicación mencionada más arriba: ciertas features participan de una manera no despreciable en la covarianza del modelo.

La idea del siguiente párrafo es muy diferente. En lugar de eliminar features de buenas a primeras, se van a construir otras nuevas, tales que su ranking resulte más sencillo, y a continuación se realizará la selección.

4. Una solución práctica: el PCA

Desde hace más de treinta años, encontramos análisis y reducciones de dimensiones a través del análisis en componentes principales (que ya explotó en un contexto de aprendizaje automático Dumez-Laude en 1986) llamado *Principal Component Analysis* en inglés, o **PCA**. La relación utilidad/tiempo_de_cálculo de este método es excelente, como ocurría en 1986... y todavía sigue siendo válido en la época del Big Data debido al hecho de trabajar con inmensos volúmenes de datos que queremos comprender.

La idea general consiste en encontrar una notación de nuestro espacio que muestre nuestras nubes de puntos de manera que ciertos ejes (dimensiones) resulten casi inútiles para describir esta nube de puntos. Por ejemplo, un platillo volador representado en un espacio de tres dimensiones puede representarse de manera muy aproximada por un disco sin espesor en un espacio de dos dimensiones: hemos logrado reducir el número de features de tres a dos con una pérdida mínima de información (aunque ya no tenemos representada una variación del espesor del platillo).

Globalmente, la rotación del espacio que nos interesa se corresponde con la que diagonaliza una matriz de pseudodistancias propias de los coeficientes de correlación de nuestra nube entre cada feature. Esto significa que el modelo funciona cuando existe cierta correlación lineal entre los factores. Si no es el caso, habrá que tratar de aplicar diversas transformaciones antes del PCA, con el ánimo de aumentar la linealidad de los factores.

5. Un ejemplo simple del uso del PCA

Este capítulo se basa en el dataset Longley, un pequeño dataset de test que contiene información acerca de los parados en EE. UU. Nuestra primera tarea será visualizar las relaciones simples entre cada columna de datos. El siguiente código implementa la función **pairs**, aplicada a la tabla **c(x,y)**, que combina las features **X** y el objetivo **y**.

```
## ANÁLISIS RESUMIDO DE FEATURES
## 
source("pair_panels.R")      # funciones parámetros de la función pairs
source("f_longley.R")        # función de acceso a los datasets Longley
require(stats)                # estadísticas básicas
require(graphics)             # gráficos básicos
require(dplyr)                 # librería de manipulación de datos
require(caret)                 # para las predicciones y transformaciones

z <- f_longley(verbose =FALSE)      # carga muda de los datos
y <- data.frame(trabajadores = Z$trabajadores) # y es el objetivo
X <- select(Z, - trabajadores, - anyo) # features - año / overfit
pairs(c(X,y),diag.panel = panel.hist, # visualización de correlaciones
      upper.panel = panel.cor,
      lower.panel = panel.smooth,
      main = "RELACIONES 2 A 2 ENTRE LAS FEATURES")
```

El gráfico obtenido es fácilmente explotable y conviene estudiarlo con atención para descubrir toda su riqueza.

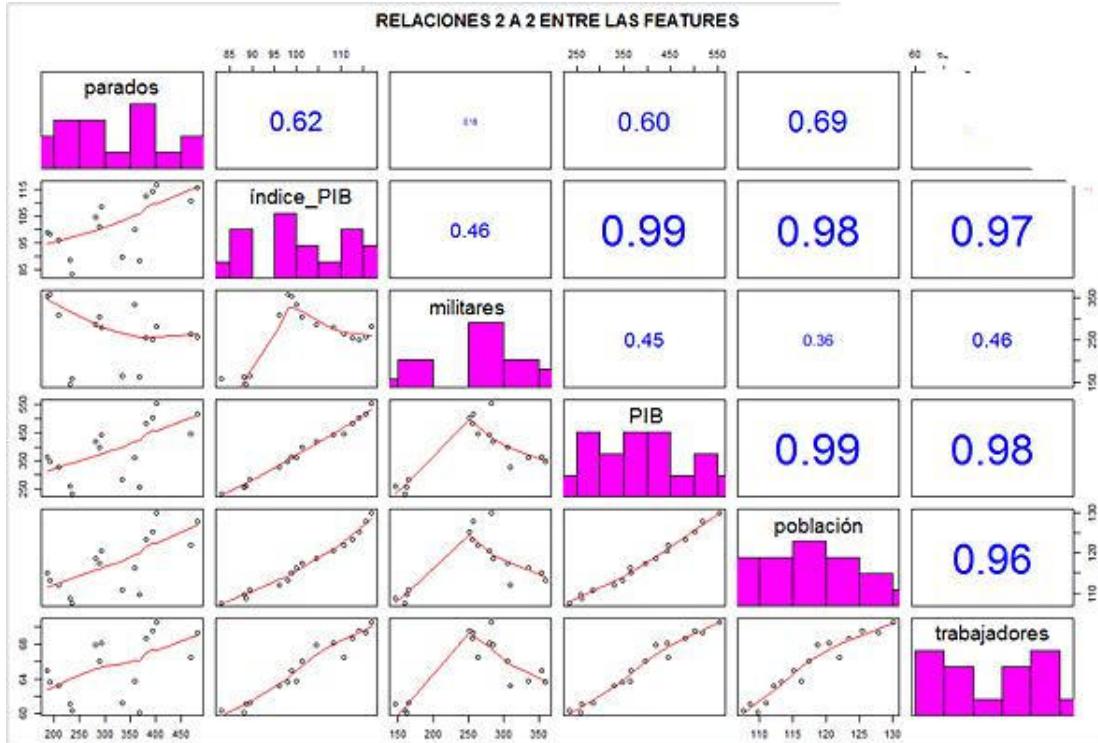


Gráfico obtenido con la función **pairs**

Las primeras conclusiones son las siguientes:

- El gráfico muestra una fuerte correlación entre ciertas features (por ejemplo: 0,99 entre **población** y **PIB**).
- La feature **militares** está poco correlada linealmente con las demás, pero sobre las cinco nubes de puntos que le corresponden (en fila y en columna) se observan distintamente dos poblaciones que, tomadas de forma individual, podrían resultar homogéneas (observe las líneas quebradas).

 Una práctica errónea (pero habitual) sería eliminar inmediatamente esta feature bajo el pretexto de que su correlación con todas las demás, incluidas aquellas por predecir, es pequeña.

El análisis por componentes principales parece ser un buen candidato:

- Muchas dimensiones están correladas linealmente.
- Algunas dimensiones discriminan claramente paquetes de datos poco numerosos y con un contorno simple.

Antes de lanzar cálculos precisos y quizás costosos, comprobaremos rápidamente la eficiencia potencial del PCA sobre nuestro problema de reducción de dimensiones.

Las features **X** son objeto aquí de un PCA a través de una función **prcomp** que genera el modelo **modelo_pca**. A través de la función **pareto.chart** de la librería gráfica de control de calidad **gcc**, se muestra un diagrama de Pareto que clasifica las varianzas **v** obtenidas elevando al cuadrado las desviaciones típicas **sdev** del modelo **modelo_pca**.

```
## ANÁLISIS VISUAL TRIVIAL EN COMPONENTES PRINCIPALES (MUY RÁPIDO)##
X_<- data.frame (scale (X)) # normalización de datos entre -1 y 1
                                # no cambia las correlaciones

modelo_pca <- prcomp (X_) ; # análisis en componentes principales
modelo_pca                         # el "modelo" obtenido
summary(modelo_pca)                 # información sobre modelo_pca

library (gcc)                      # librería "quality control charts"
v <- modelo_pca$sdev^2            # varianza = cuadrado de la desviación tipica

pareto.chart (v,xlab = "Componentes",      # diagrama de Pareto
              ylab = "Varianza",
              col=rainbow(length(v)),
              main = "Pareto de las nuevas dimensiones")
```

La primera tabla resultado devuelve la matriz de la rotación en el espacio que permite pasar de las dimensiones originales a las nuevas dimensiones. Esta tabla puede explotarse por quienes quieran representar precisamente la transformación hecha para pasar de las features a las nuevas features candidatas (PC1, PC2...).

Rotation:

	PC1	PC2	PC3	PC4	PC5
parados	0.3658062	0.59532321	0.7100763	-0.004614581	-0.086870665
índice_PIB	0.5210129	-0.05808997	-0.1889153	-0.776958379	0.292946852
militares	0.2296424	-0.79831473	0.5511572	0.078584283	-0.002874243
PIB	0.5199086	-0.05345522	-0.3174971	0.135947010	-0.779455948
población	0.5212397	0.04529867	-0.2356355	0.609637027	0.546878225

La segunda tabla devuelve la varianza acumulada de las diferentes componentes. Muestra que más del 95 %

(exactamente un 0.9570) de la varianza explicativa se agrega desde la segunda componente (PC2).

Importance of components:

	PC1	PC2	PC3	PC4	PC5
Standard deviation	1.8999	1.0841	0.44627	0.12199	0.03088
Proportion of Variance	0.7219	0.2351	0.03983	0.00298	0.00019
Cumulative Proportion	0.7219	0.9570	0.99683	0.99981	1.00000

El diagrama de Pareto muestra el histograma de las contribuciones de cada nueva dimensión a la varianza total y la curva acumulada de esta contribución. Esto permite anticipar el número de dimensiones que es posible conservar sin perder mucha varianza explicativa. Típicamente, queremos conservar las dimensiones que explican un acumulado de al menos el 95 % de la varianza del modelo.

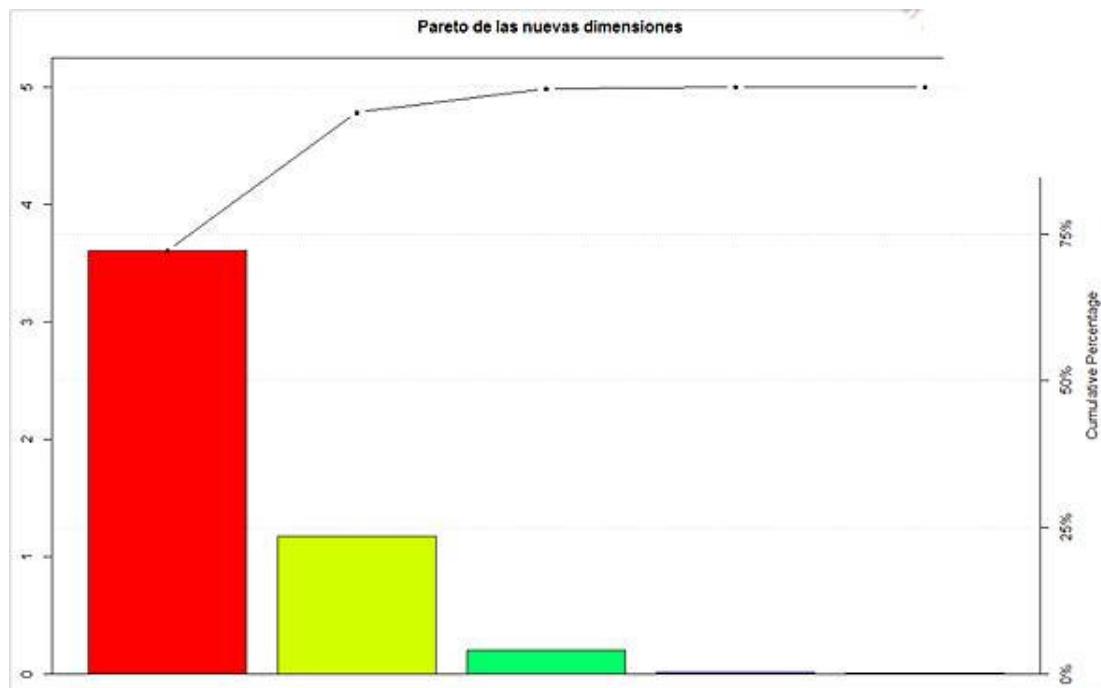


Gráfico obtenido con la función `pareto.chart`

La curva por encima de la segunda barra muestra lo que ya habíamos identificado en la tabla anterior, es decir, que el 95 % se alcanza con PC2. El gráfico permite apreciar mejor el impacto general del PCA y las proporciones entre PC1 y PC2.

Tan solo queda realizar el PCA efectivo, y será la ocasión de utilizar la librería **caret**. Aquí se suceden tres transformaciones sucesivas, los resultados de la transformación anterior son la entrada de la transformación siguiente. Las transformaciones sucesivas de la tabla de features **X** son **X_1**, **X_2**, **X_3**. Evidentemente, no es útil dar un nombre y conservar estas tablas intermedias en una versión definitiva del código. Sin embargo, la puesta a punto del código requiere estudiar atentamente el resultado de cada etapa de cálculo y, por tanto, ver a qué se parece cada transformación.

La transformación YeoJohnson (primera transformación realizada en la sección de código siguiente) no se justifica formalmente aquí. Su finalidad es normalizar los datos (en el sentido de una distribución normal).

Se le propone aquí para memorizarla y por cuatro motivos didácticos importantes:

- Siempre hay que abordar la eventualidad de una transformación original monótona (es decir, siempre creciente o

decreciente) para aplanar las nubes de puntos que se espera utilizar en una regresión lineal (por ejemplo: si la curva que se ajusta a la nube de puntos se parece a un logaritmo, entonces puede contemplarse la posibilidad de realizar una transformación relativamente exponencial para conseguir... iuna recta!).

- Este tipo de transformación debería, evidentemente, realizarse antes del PCA.
- Conviene prestar atención al dominio de aplicación de las transformaciones realizadas (por ejemplo, YeoJohnson soporta valores de x negativos y reacciona bien en 0).
- Aquí, la transformación se aplica sobre todas las features, lo cual no es sin duda oportuno, y conviene aplicar estas transformaciones previas de forma inteligente tras haber estudiado el aspecto de las distintas nubes de puntos que representan las features dos a dos.

► El método de transformación YeoJohnson, igual que el método Box-Cox, no aplica parámetros fijos a priori, sino que busca los mejores parámetros llamados habitualmente λ (Lambda) en la literatura correspondiente, y a continuación aplica la transformación final optimizada (que puede ser tanto de base exponencial como logarítmica). Esto permite ahorrar un tiempo considerable en el análisis preliminar del problema.

```
## REDUCCIÓN DE DIMENSIÓN efectiva mediante PCA
## 3 transformaciones sucesivas sobre los datos (librería caret)
t1 <- preProcess(X,method = "YeoJohnson");t1 # soporte x < 0
X_1 <- predict(t1, X) # transformación 1
t2 <- preProcess(X_1,method = "pca",thresh = 0.95);t2 # pca
X_2 <- predict(t2, X_1) # transformación 2
t3 <- preProcess(X_2,method = "range");t3 # norma entre 0 & 1
X_3 <- predict(t3, X_2) # transformación 3

pairs(c(X_3,y),diag.panel = panel.hist, # análisis visual
      upper.panel = panel.cor,           # /!\ NO SOBREINTERPRETAR
      lower.panel = panel.smooth,
      main = "RELACIONES 2 A 2 ENTRE LAS NUEVAS FEATURES")
```

La transformación YeoJohnson devuelve la siguiente información:

```
Lambda estimates for Yeo-Johnson transformation:
0.44, 1.63, 1.61, 0.71, -2.25
```

La definición de esta transformación en la literatura declara que la fórmula correspondiente a un **Lambda positive e inferior a 0.5 es $\log(x+1)$** .

De modo que se aplicará esta transformación a la feature **X\$paro**, la prueba es que su Lambda es igual a 0.44 (ver la primera columna de la tabla correspondiente a la primera feature de **X**).

Es posible y deseable estudiar a posteriori las transformaciones que se han realizado: por su parte, evite la magia sin control.

La transformación PCA devuelve la siguiente información:

```
PCA needed 2 components to capture 95 percent of the variance
```

Esto resulta coherente con el análisis anterior y devuelve el gráfico por pares siguiente:

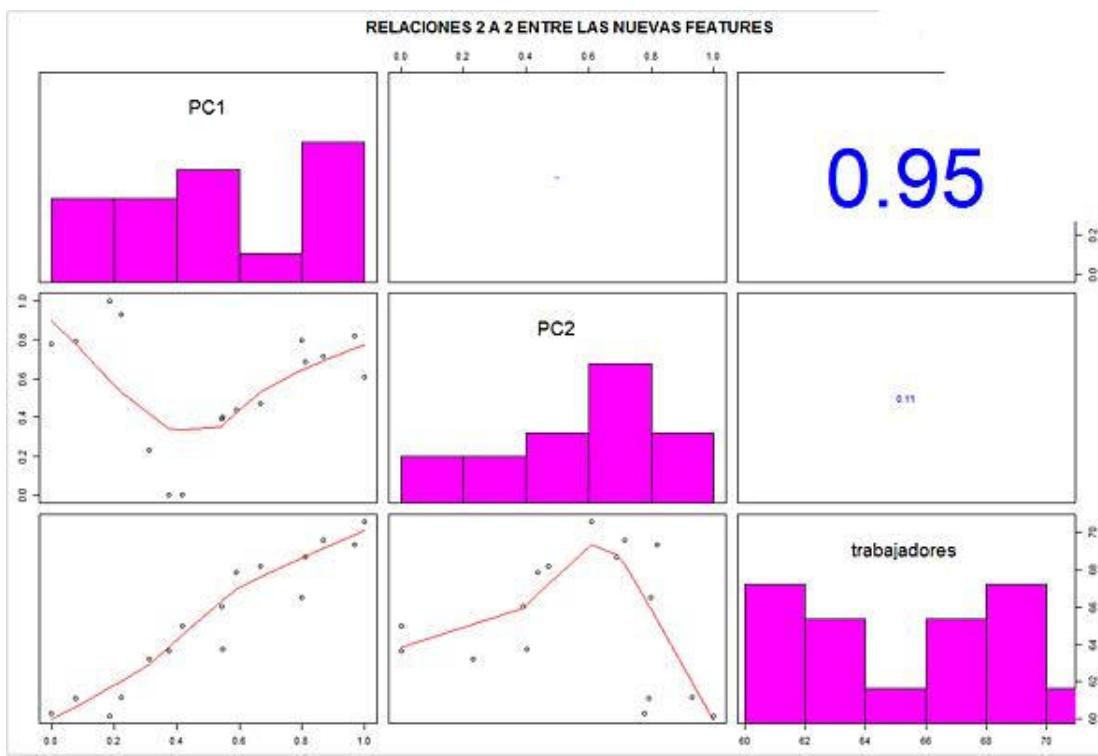


Gráfico obtenido con la función **pairs**

► Preste atención: el 0.95 no tiene nada que ver con nuestro umbral de 0.95 en el PCA, aquí es el valor absoluto del coeficiente de correlación entre la nueva feature PC1 y nuestro objetivo **y\$trabajadores**.

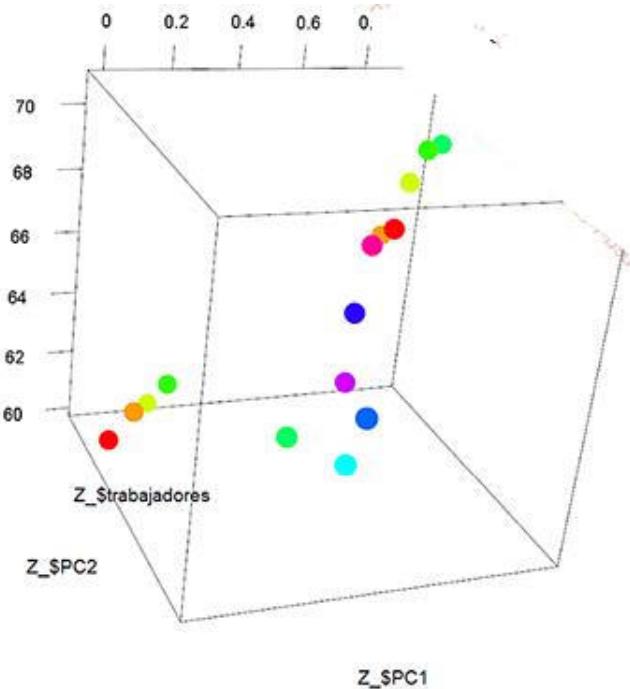
PC1 y PC2 no están correladas, lo cual resulta conveniente según el objetivo fijado. PC2 no está correlada con la variable objetivo, pero no conviene infravalorar esta información pues la combinación de PC1 y PC2 explica las variaciones de **trabajadores**.

La siguiente visualización 3D permite apreciar mejor la calidad de la reducción de dimensiones. Tras haber ordenado nuestras filas por precaución utilizando la función **orderBy** de la librería **doBy**, es posible manipular interactivamente el cubo [PC1,PC2,trabajadores] para comprobar según determinados ángulos la compacidad de las nubes de puntos obtenidas. Esta acción valida visualmente la eficacia de la reducción de dimensiones.

```
## creación de una representación 3D manipulable interactivamente
library(doBy) # para poder ordenar los datos
# Buena práctica: ordenar antes de las representaciones 3D
Z_ <- data.frame(X_3,y,row.names = NULL)
Z_ <- orderBy(~ PC1+PC2,data = Z_)

# MANIPULACIÓN INTERACTIVA CON EL RATÓN en 3D
library(rgl)
plot3d(Z_$PC1,Z_$PC2,Z_$trabajadores,
       type="s", size=1.75, lit=FALSE,col = rainbow(10))
```

Haciendo girar el cubo con el ratón, observamos en efecto dos zonas coherentes; la reducción de dimensiones ha funcionado correctamente y los puntos no están dispersos de forma aleatoria en el espacio:



*Cubo obtenido con la función **plot3d***

Estas conocidas transformaciones posibles se aplicarán a todos los datos futuros antes de su introducción en los modelos de clasificación o de predicción: datos de **entrenamiento**, datos de **test**, datos de **validación** y de **producción**.

Los algoritmos basados en el álgebra lineal, como el PCA, manipulan en primer lugar matrices y vectores. Son estructuras homogéneas en términos de contenido, que contienen números (enteros, reales, y llegado el caso complejos). Conviene asegurarse de proporcionarles datos completamente homogéneos y limpios de cualquier residuo. Es lo que vamos a estudiar a continuación.

6. Los valores desconocidos y las features mal condicionadas

El problema de la codificación del cero (que hemos abordado brevemente a través de la diferencia entre las variables de intervalos y de razón) está conectado a un problema sobre el que el data scientist debe trabajar con atención: ¿cómo tratar los valores desconocidos?

Un valor puede faltar porque no esté informado, o bien porque no tenga sentido para un tipo de ocurrencia (por ejemplo: entre los medios de transporte de una ciudad, el atributo **consumo_de_carburante** no estaría informado para las bicicletas...). En los archivos nos encontramos con frecuencia con valores nulos o infinitos (**Inf** o **-Inf**) que deberían considerarse como valores que faltan... ¡o no!

En R, para indicar que una variable está ausente, hay que asignarle el valor **NA** (sin comillas) que significa **Not Available**. Es posible saber si un valor es **NA** mediante la función **is.na**, que devuelve **TRUE** si el valor es **NA** y **FALSE** en caso contrario. Con la misma idea, **anyNA** indica si una estructura contiene al menos un valor **NA**. Como de costumbre con R, es posible aplicar la función al conjunto de elementos de una estructura (vector, data.frame...). He aquí un ejemplo:

```
## COMPROBAR SI NA (NOT AVAILABLE) ##

y <- c(NA,1,NA) # fabricamos un vector con NA
is.na(y)          # determinar dónde están los NA en el vector
```

```

# esto devuelve: TRUE FALSE TRUE

is.na(c("D", "NA", Inf, NaN, 0, "", NA)) # aquí solo el último es NA!
## esto devuelve: FALSE FALSE FALSE FALSE FALSE TRUE ##

anyNA(c(1,2,3)) # devuelve FALSE: no hay NA aquí

anyNA(c(NA,1,NA)) # devuelve TRUE: hay al menos un NA

```

La lectura de un archivo con valores no disponibles o no significativos se ve facilitada por el parámetro **na.strings**, bien adaptado a la carga de archivos.

En el archivo **f_NA.csv** siguiente, se encuentran valores que faltan y valores anormales ("NA", "#DIV/0!", ""):

```

Id,VAR1,VAR2,VAR3,VAR4
1,10,10.5,"tip"
4,20,10.5,"tip"
3,20,"NA","tip",1.1
5,30,10.5,"top",1.2
9,30,, "tip",1.2
12,30,"","tip",0
7,20,10.5,"top",1.1
10,"#DIV/0!","", "tip",1.8

```

Este código va a cargar el archivo de manera correcta:

```

## Leer un archivo indicando el tipo de los valores desconocidos ##
## estos datos se convertirán en NA ##

Z <- read.csv("f_NA.csv",
               header = TRUE,
               na.strings = c("NA", "", "#DIV/0!"))

head(Z)      # muestra las 5 primeras filas
tail(Z)      # muestra las 5 últimas filas

str(Z)       # ;devuelve los primeros valores y las clases!

```

El código anterior ha cargado el archivo satisfactoriamente:

```

Id VAR1 VAR2 VAR3 VAR4
1 1 10 10.5 tip NA
2 4 20 10.5 tip NA
3 3 20 NA tip 1.1
4 5 30 10.5 top 1.2
5 9 30 NA tip 1.2
6 12 30 NA tip 0.0
7 7 20 10.5 top 1.1
8 10 NA NA tip 1.8

```

La función **str()** permite comprobar la naturaleza de las distintas features: **int** para entero natural, **num** para real, **Factor** para una categoría de distintos niveles (**levels**).

```
'data.frame':      8 obs. of  5 variables:  
 $ Id : int  1 4 3 5 9 12 7 10  
 $ VAR1: int  10 20 20 30 30 30 20 NA  
 $ VAR2: num  10.5 10.5 NA 10.5 NA NA 10.5 NA  
 $ VAR3: Factor w/ 2 levels "tip","top": 1 1 1 2 1 1 2 1  
 $ VAR4: num  NA NA 1.1 1.2 1.2 0 1.1 1.8
```

He aquí algunos ejemplos de aplicación de **is.na()**:

```
is.na(Z)      # una tabla booleana que dice dónde están los NA
```

```
Id  VAR1  VAR2  VAR3  VAR4  
[1,] FALSE FALSE FALSE FALSE  TRUE  
[2,] FALSE FALSE FALSE FALSE  TRUE  
[3,] FALSE FALSE  TRUE FALSE FALSE  
[4,] FALSE FALSE FALSE FALSE FALSE  
[5,] FALSE FALSE  TRUE FALSE FALSE  
[6,] FALSE FALSE  TRUE FALSE FALSE  
[7,] FALSE FALSE FALSE FALSE FALSE  
[8,] FALSE  TRUE  TRUE FALSE FALSE
```

```
is.na(Z[8,])  # la fila 8 de la tabla
```

```
Id  VAR1  VAR2  VAR3  VAR4  
8 FALSE  TRUE  TRUE FALSE FALSE
```

```
is.na(Z$VAR2) # la columna de la feature VAR2 de la tabla
```

```
[1] FALSE FALSE  TRUE FALSE  TRUE  TRUE FALSE  TRUE
```

Cuando una feature está mal condicionada, por ejemplo una columna de enteros que deberían estar codificados en categorías (**Factor** con varios **Levels**), afortunadamente es posible rehacerla:

```
as.factor(Z$VAR1) # transforma la feature VAR1 en categorías  
is.na(Z$VAR1)      # esto no pierde los NA
```

```
[1] 10    20    20    30    30    30    20    <NA>  
Levels: 10 20 30
```

```
[1] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
```

 R permite manipular otros tipos de datos raros, que no hay que confundir con **NA** (**NaN** / Not a Number e **Inf** / Infinito).

```

## no confundir NA con NaN o Inf
## 
1/0      # división por 0 devuelve Inf (infinito)
1L/0L    # división por 0 entero corto devuelve también Inf
1/Inf    # devuelve 0 (cool!)
Inf/Inf  # devuelve NaN: Not a Number
0/0      # devuelve NaN
is.nan(1i) # FALSE: 1i es el número complejo i
is.infinite(1/0)  # TRUE
is.infinite(1i/0) # TRUE (bravo R!!!)

```

Por trabajar con los datos con precaución y haber reflexionado cuidadosamente acerca de su naturaleza, el data scientist habrá adquirido ahora una percepción más clara de su problema y podrá arriesgarse a una cierta creatividad, incluso a concebir nuevas features!

7. Creación de nuevas features

En ocasiones las features contienen poco poder predictivo, mientras que intuitivamente el analista está persuadido de que los datos manipulados poseen cierto vínculo con el objetivo que hay que predecir.

Paradójicamente, a menudo es conveniente obligarse a realizar una fase de creación de nuevos atributos que precederá a una nueva fase de reducción de las dimensiones, ahora realizada sobre bases sanas.

Existe por ejemplo un método avanzado de PCA (Kernel_PCA) muy eficaz que se implementa en distintas librerías en R siguiendo este principio:

- Descomposición de las features existentes en nuevas features.
- PCA y reducción de dimensiones.

Las transformaciones clásicas que permiten generar o adaptar nuevas features son las siguientes:

- Normalización por extensión o reducción del rango de valores (por ejemplo, el **z-score**).
- Normalización de la distribución (la transformación Yeo-Johnson que hemos visto brevemente más arriba es de este tipo).
- Normalización de los textos en números (Ngram, puntuación de sentimientos... consulte el capítulo Procesamiento del lenguaje natural - NLP).
- La creación de variables ficticias/dummies por descomposición (value mapping) de features ya presentes (veremos un ejemplo más adelante).
- La creación de variables ficticias por clustering (la fila i se asignará a un clúster concreto tras el uso de un algoritmo de clasificación realizado en pre-processing).
- La creación de variables ficticias por estudio de un grupo de features que representan una secuencia causal o temporal:

Típicamente, si hay una serie temporal repartida en varias features o si diferentes features están ordenadas, es posible calcular las características de estas distribuciones, las más simples son la media, la desviación típica, el mínimo y el máximo y los factores de forma (curtosis: factor de aplandamiento, skewness: coeficiente de asimetría).

- La discretización de variables (pertenencia a un intervalo, pertenencia a un cuantil...):

Esta acción puede resultar delicada, pues en ocasiones los umbrales tienen fuertes impactos, aunque a menudo es muy eficaz si se tiene el tiempo suficiente para estudiar la distribución de la variable con objeto de determinar los umbrales y si se utiliza el resultado en un algoritmo adaptado (por ejemplo: árboles de decisión).

Podemos considerar el posicionamiento de los umbrales como un hiperparámetro que se tratará de optimizar (la

optimización de un hiperparámetro es uno de los métodos básicos de aplicación de un modelo genérico, como por ejemplo la búsqueda del parámetro Lambda óptimo en la transformación de Yeo-Johnson vista más arriba).

- En general, todas las combinaciones y transformaciones simples de una a n variables: mínimo, máximo, suma, producto, potencias, raíces, distancias y todas sus combinaciones lineales: el número de combinaciones resulta infinito, de modo que hay que reflexionar bien acerca de su sentido, en las posibilidades de mejorar y en la manera de realizar dichas combinaciones antes de lanzarse.

El siguiente código, que opera sobre el archivo cargado en nuestros ejemplos anteriores, va a transformar **VAR1** en categorías (**Factors**), crear features **dummies** por una descomposición realizada a partir de las features categóricas **VAR1** y **VAR3**, generar una combinación simple **Id*VAR4** y, por último, agregar el conjunto en un nuevo **data.frame**.

```
## creación de nuevas features respetando los NA      ##

library(caret) # para obtener la función dummyVars

# transforma la feature VAR1 en categorías (es decir, factors)
# que contienen un nivel (level) por valor de entero
Z$VAR1 <- as.factor(Z$VAR1) # "obliga" un numérico como factor
str(Z)                      # ver el resultado

# creación de una nueva columna producto de Id*VAR4
t1 <- dummyVars( ~ Id*VAR4 , data = Z)
new_feature1 <- predict(t1, newdata = Z)
head(new_feature1)

# creación de tantas columnas como el producto del número
# de niveles (levels) de categorías (factors) VAR1 y VAR3
t2 <- dummyVars( ~ VAR1:VAR3 , data = Z)
new_feature2 <- predict(t2, newdata = Z)
head(new_feature2)

## construcción del nuevo data.frame resultante
new_Z <- data.frame(new_feature1,
                     new_feature2,
                     Z,
                     row.names = NULL)

str(new_Z)                  # ver el resultado
```

Que produce las 14 features siguientes:

```
'data.frame':      8 obs. of  14 variables:
 $ Id             : num  1 4 3 5 9 12 7 10
 $ VAR4           : num  NA NA 1.1 1.2 1.2 0 1.1 1.8
 $ Id.VAR4        : num  NA NA 3.3 6 10.8 0 7.7 18
 $ VAR1.10.VAR3.tip: num  1 0 0 0 0 0 0 NA
 $ VAR1.20.VAR3.tip: num  0 1 1 0 0 0 0 NA
 $ VAR1.30.VAR3.tip: num  0 0 0 0 1 1 0 NA
 $ VAR1.10.VAR3.top: num  0 0 0 0 0 0 0 NA
 $ VAR1.20.VAR3.top: num  0 0 0 0 0 0 1 NA
 $ VAR1.30.VAR3.top: num  0 0 0 1 0 0 0 NA
```

```
$ Id.1      : int  1 4 3 5 9 12 7 10
$ VAR1      : Factor w/ 3 levels "10","20","30": 1 2 2 3 3
$ VAR2      : num  10.5 10.5 NA 10.5 NA NA 10.5 NA
$ VAR3      : Factor w/ 2 levels "tip","top": 1 1 1 2 1 1
$ VAR4.1    : num  NA NA 1.1 1.2 1.2 0 1.1 1.8
```

Ahora habría que:

- Eliminar las features duplicadas, como Id e Id.1.
- Eliminar las features constantes o casi constantes (ZVF: *Zero Variance Features*), como la feature **VAR1.10.VAR3.top**.
- Llegado el caso, mejorar el número de features (por ejemplo, utilizando la librería **dplyr**).
- Iterar todo el proceso de análisis con estas nuevas variables.

8. A modo de conclusión

Los procedimientos tan operacionales descritos aquí se enmarcan en el día a día del data scientist y garantizan a menudo resultados satisfactorios.

Es posible realizar muchas otras manipulaciones (muy útiles), y existen además numerosas librerías que encapsulan diversas técnicas de feature engineering. Sin embargo, intentar utilizarlas sin poseer una experiencia previa en los procedimientos descritos más arriba no es, sin duda, conveniente.

Como para el ajuste de los algoritmos de machine learning, uno de los medios para converger reside en parte en la evaluación del error cometido en cada iteración de nuestro proceso de mejora y de análisis. Sin embargo, aquí las dificultades son más importantes, pues el número de opciones a nuestra disposición es muy elevado.

Llegados a este punto, los artículos de investigación son, a menudo, una gran ayuda, pues requieren la imaginación del profesional, a condición de que no se deje disuadir o distraerse por el formalismo matemático. El feature engineering se basa tanto en las matemáticas como en la intuición, y una interpretación superficial de los aspectos matemáticos puede bastar para alimentar la creatividad.

Evidentemente, los más experimentados podrán ahondar en los aspectos teóricos que a nosotros solo se nos ocurrirán más adelante.

PCA clásico, elementos matemáticos

 La asimilación de esta sección no es indispensable para la implementación operacional de los conceptos descritos más arriba.

Esta pequeña sección debería permitirle leer y descifrar sacándole provecho a la literatura acerca del PCA y le dará una visión global de algunas técnicas matriciales habituales (entre ellas el cambio de base).

Con las siguientes convenciones:

- n : número de filas de datos.
- p : número de features (es decir, de dimensiones).
- q : número (reducido) de dimensiones, que debería ser inferior a p !
- i : índice, de 1 a n .
- j : índice, de 1 a p .
- k : índice, de 1 a q .
- X : matriz (x_{ij}) de datos **centrados**, o **centrados y reducidos** (hemos aplicado una transformación previa, *centrado* significa haber sustraído a cada columna su media, *reducido* significa que se ha dividido este resultado por la desviación típica de la columna; en la literatura inglesa *centrado* y *reducido* es el **z-score**). Preste atención, dos columnas no deberían ser colineales (es decir, una columna no debería ser totalmente proporcional a la otra).
- X' : matriz (x'_{ij}) de datos transformados en una nueva base.
- X'' : matriz (x''_{ik}) de datos representados en una base truncada de q dimensiones, es el objetivo!

Podemos calcular:

$$C = \frac{1}{p} X^T X$$

que es la matriz de covarianza -si X solo estuviera centrada- o la matriz de coeficientes de correlación si X estuviera centrada y reducida. Parece que, si no se reduce la matriz, entonces las variables con una gran varianza influyen más en el modelo. C es una matriz simétrica real (e inversible, pues no había vectores colineales en X).

 Si reemplazamos p por $(p-1)$ en la fórmula del cálculo de la matriz de covarianza, podemos aplicar la corrección de Bessel, cuyo objetivo es eliminar el sesgo de la varianza. De manera inversa, en ocasiones se aplica en ciertos cálculos una corrección p sobre $(p+1)$ para mejorar el error cuadrático medio (MSE).

 El uso de la matriz de covarianza indica que la PCA es apropiada a las distribuciones gaussianas. En efecto, esta matriz traduce relaciones de primer y de segundo orden. Los órdenes más elevados se abordarán mediante otros métodos como ICA (Independent Component Analysis) o Kernel PCA (PCA con transformación de los datos mediante lo que se ha acordado llamar el «kernel trick» en la literatura anglosajona).

 Si se quiere asignar un peso a cada dimensión, se considera una matriz diagonal de pesos W , tal que la suma de p pesos sea igual a 1. Tenemos entonces $C = 1/p X^T W X$.

Vamos a escoger una nueva base tal que la varianza sobre cada eje esté maximizada (y, por lo tanto, estos ejes tenderán hacia la independencia). Parece que las varianzas obtenidas son los valores propios de la diagonalización de esta matriz (la demostración queda fuera de nuestro propósito aquí, palabras clave: optimización, problema de maximización con multiplicador de Lagrange).

Estamos en el caso de un endomorfismo de espacio vectorial, que es una aplicación lineal de un espacio vectorial en sí mismo y, por tanto, hay una sola matriz de paso P para determinar entre la nueva y la antigua base, tal que:

$$C' = P^{-1} C P$$

La diagonalización de esta matriz puede proporcionarnos esta nueva base. El teorema espectral para las matrices simétricas reales nos dice que pueden diagonalizarse y que existe una matriz O ortogonal y una matriz C' diagonal tales que:

$$C = O C' O^T$$

Recuerde, una matriz ortogonal posee una inversa igual a su traspuesta con:

$$O^T O = I_p$$

(matriz identidad de p dimensiones)

$$O^{-1} = O^T$$

¡Hemos encontrado nuestro cambio de base!

Es la matriz diagonal cuyos valores diagonales son los valores propios de la matriz C . Estos valores son positivos, pues son varianzas, es decir, cuadrados de las desviaciones típicas. Hemos llevado cuidado para ordenar los vectores propios según un orden decreciente de los valores propios, lo que nos va a simplificar el trabajo para pasar de p a q dimensiones; la dimensión más significativa es, por tanto, la primera. Queremos, en efecto, que la variabilidad sobre la primera componente es máxima, así como sobre la segunda teniendo en cuenta la primera con la que nos hemos hecho independientes, y así sucesivamente...

La traza de la matriz C' representa la varianza total. Si consideramos una reducción de dimensiones truncando nuestra nueva base, obtenemos una matriz C'' . En esta base truncada, la traza (suma de valores de la diagonal) es la varianza todavía presente en nuestra nueva base. Es la relación entre estos dos valores la que habíamos representado como umbral en el Pareto dibujado más arriba (0.95).

- ➊ Si observamos las dos o tres primeras dimensiones, resulta cómodo dibujar elipsoides cuyos semiejes tengan una longitud de $1/2$ desviación típica para obtener una escala visual que permita juzgar la distancia entre diversos puntos o nubes de puntos (la desviación típica aquí es la raíz del valor propio sobre el eje).

Reducción de los datos (data reduction)

Este pequeño párrafo no concierne, estrictamente hablando, al «Feature Engineering», aunque hace referencia a la limpieza de los datos antes de su uso en un algoritmo, a menudo vinculado con el machine learning. Como la confusión es habitual, echemos un vistazo a esta noción. Aquí el problema es el contrario: la preparación no afecta a las «columnas», sino a las «filas».

Esta operación consiste en extraer y, a continuación, utilizar un subconjunto coherente de las filas contenidas en un dataset de gran volumen. En ocasiones resulta útil para ahorrar recursos de máquina, aunque también sirve para mejorar la eficacia de ciertos algoritmos o para disminuir la exposición al «ruido».

Esta noción de ruido es, de hecho, intuitiva en el caso de la interpretación de imágenes o de sonidos, donde estamos habituados a la idea de que los sensores son menos eficaces de lo que desearíamos e introducen cierto «ruido». Este problema de introducción de ruido no es exclusivo de los sensores «físicos»: también puede introducirse «ruido» accidentalmente en cualquier tipo de datos en función del proceso de captura de la información.

Una de las técnicas más habituales para reducir los datos consiste en seleccionar ciertos «puntos» del dataset, y a continuación seleccionar, de forma iterativa, otros puntos próximos cuyo valor resulte cercano al de sus vecinos más cercanos (es decir, que posean una misma clase o una clase cercana a través de una distancia propia al modelo). Reconocemos aquí una implementación del algoritmo kNN (*k next neighbors*).

La tasa de reducción de los datos se mide a través de una relación (en inglés: *reduction rate*). Una idea sencilla consiste en realizar un cierto número de intentos con un método de reducción de datos, y a continuación utilizar estos datos como entrada a un algoritmo de clasificación y de predicción que nos parezca eficaz. Entonces es posible optimizar la tasa de reducción comparando la eficacia de la predicción entre los distintos intentos (por ejemplo, consultando los elementos de una matriz de confusión, como *accuracy*).

Los algoritmos más utilizados pertenecen a las siguientes familias: *Instance Based Learning* (IBL, llamados a veces *memory base learning*; IBL comprende, entre otros, los algoritmos *kNN* y *Kernel Machines*) o *Clustering Based Learning* (CBL, que no hay que confundir con *Case based learning*).

Estas elecciones distan bastante de ser neutras, pues condicionan la eficacia de los modelos entrenados y son, a menudo, buenos limitadores del sobreajuste (overfitting).

Reducción de la dimensionalidad y entropía

Puede parecer molesto basar toda nuestra estrategia de reducción de la dimensionalidad en el álgebra lineal, como hacemos con el PCA, pues estas técnicas podrían involucrar ciertas hipótesis simplificadoras acerca de la naturaleza de los datos sobre los que el data scientist tiene algo de prisa.

El concepto de «entropía e información compartida» descrito en la introducción de este libro parece más general y en apariencia resulta legítimo construir parte de su estrategia sobre estos conceptos.

En el capítulo Introducción, hemos expresado la información mutua de dos variables como representable mediante la siguiente expresión (en función de la entropía):

$$I(X,Y) = H(X) + H(Y) - H(X,Y)$$

Esta expresión se anula en el caso de dos variables dependientes. Aquí, la noción de dependencia no supone una dependencia lineal o incluso una dependencia lineal tras cualquier transformación.

1. Descripción teórica del problema

Vamos a llamar S (como set), al conjunto de nuestras variables explicativas x_i e y a nuestra variable respuesta.

Nuestro objetivo es seleccionar un subconjunto de S que contenga p features, es decir, ciertas variables explicativas (features); llamaremos s a este subconjunto y m a su número de features.

Ahora nos hace falta un criterio de selección.

En una primera aproximación, podemos imaginar rápidamente que hay que encontrar el conjunto de features s que posea **la mayor dependencia con el objetivo** y . En efecto, si un conjunto de features posee más información mutua que otro con el objetivo, este conjunto será el más adecuado para predecir el objetivo.

Como hemos decidido utilizar la información mutua, nuestra medida de dependencia será

$$D(s,y) = I(s,y) = I(\{x_1, \dots, x_m\}, y).$$

La idea será, entonces, explorar todas las posibilidades de conjunto s incluidas en S , calcular esta dependencia y recordar la más elevada.

Como p es algo grande, nos lleva a calcular un gran número de combinaciones de información mutua entre cada subconjunto posible s e y .

Resulta más difícil que la expresión matemática correspondiente a $I(\{x_1, \dots, x_m\}, y)$; aquí en el caso continuo es un poco ruda:

$$I(\{x_1, \dots, x_m\}, y) = \iiint \dots \int p(x_1, \dots, x_m, y) \log \frac{p(x_1, \dots, x_m, y)}{p(x_1, \dots, x_m) p(y)} dx_1 \dots dx_m dy$$

De modo que hacer esto un gran número de veces para encontrar el máximo puede resultar impráctico en la realidad.

El cálculo de la **dependencia máxima** que consiste en buscar s tal que $s = \operatorname{argmax}_{s \in S} D(s, y)$ es demasiado costoso.

Nos vemos tentados a realizar un cálculo más sencillo, que consiste en no calcular todas las grandes integrales

múltiples, sino en calcular únicamente las dependencias miembro a miembro entre cada feature de cada conjunto s candidato y la variable de respuesta y , y a continuación sumar y promediar el conjunto. El problema es que esta variable introduce cierta redundancia en el cálculo. Llamamos a esta cantidad relevancia. Como reemplazará más adelante la dependencia, vamos a llamarla D' .

$$D'(s, y) = \frac{1}{|s|} \sum_{x_i \in s} I(x_i, y)$$

- La notación $|s|$ significa «cardinal de conjunto s », aquí el valor m, pero como m cambia en función de s , es más adecuado utilizar esta notación en esta expresión.

El cálculo de la **relevancia máxima**, que consiste en buscar s tal que:

$$s = \operatorname{argmax}_{s \in S} D'(s, y)$$

es mucho más rápido, aunque integra parejas de features redundantes, lo cual es una muy mala noticia cuando se quiere disminuir el número de features!

Pero no importa, basta con calcular la redundancia 2 a 2 de las variables (no es toda la redundancia, aunque es fácil de calcular) y buscar a continuación una configuración que minimice esta **redundancia**.

La expresión de la redundancia R consiste simplemente en sumar la información mutua de todas las parejas de variables y realizar una media (es, por tanto, una doble suma sobre s).

$$R(s) = \frac{1}{|s|^2} \sum_{x_i, x_j \in s} I(x_i, x_j)$$

Vemos una redundancia mínima y buscamos ahora una s tal que:

$$s = \operatorname{argmin}_{s \in S} R(s)$$

Nuestra solución s es el subconjunto de features que mejor corresponde a estas dos condiciones (simultáneamente).

El nombre de este método es, naturalmente:

Minimum Redundancy - Maximum Relevance

Es habitual abreviar este nombre de la siguiente manera: **mRMR**

2. Implementación en R y discusión

Vamos a utilizar el paquete **mRMRe**, que implementa varias versiones de cálculo de información mutua a través del uso de diversas funciones auxiliares (de fórmulas de correlación). Según la naturaleza de su problema, será conveniente inclinarse por estas funciones, típicamente para escoger entre una implementación basada en una correlación en el sentido de Pearson, de Spearman, o una medida de frecuencia.

Si llamamos ρ a esta función de correlación, entonces el cálculo de información mutua entre dos variables a y b es:

$$I(a,b) = -\frac{1}{2} \ln(1 - \rho(a,b)^2)$$

Sobre estas dos variables, utilizaremos el conjunto de datos en XOR usado en el capítulo Técnicas y algoritmos imprescindibles, aunque con todas sus features.

Con sus conocimientos actuales, el comienzo del código no presenta ningún secreto para usted.

```
# Leer los datos

data <- read.csv("datatest1.csv",
                 sep=";",
                 dec=",",
                 na.strings=c(".", "NA", "", "?"),
                 strip.white=TRUE,
                 encoding="UTF-8")

data$clase<-as.factor(data$clase) # es adecuado transformar
# los valores discretos
# en factors

#-----
set.seed(1)
library(caret)
p       <- createDataPartition(y=data$clase,
                             p= 60/100,list=FALSE)
training <- data[p,]
desconocido <- data[-p,]
q       <- createDataPartition(y=desconocido$clase,
                             p= 50/100,list=FALSE)
test     <- desconocido[q,]
validation <- desconocido[-q,]

# tipos de variables y lista

X_all    <- c("v1", "v2", "v3", "v4",
            "v5", "v6", "v7", "v8", "v9",
            "v10", "v11", "v12")

num_all   <- c("v1", "v2", "v3", "v4",
            "v5", "v6", "v7", "v8", "v9",
            "v10", "v11", "v12")

y        <- "clase"           # nuestro objetivo

#-----
X_4      <- c("v1", "v2", "v3", "v4")
plot(training[c(X_4, y)],
     col = training$clase) # visión general

#-----
## ANÁLISIS RESUMIDO DE FEATURES ##
```

```

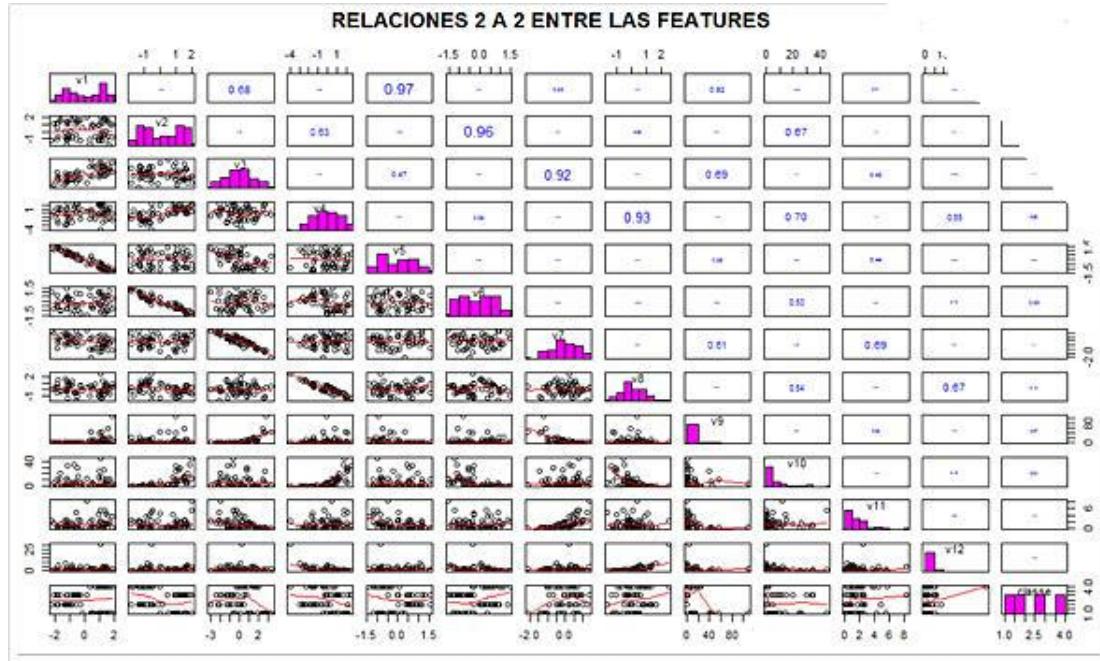
source("pair_panels.R") # funciones parámetros de la función pairs
require(stats) # estadísticas básicas
require(graphics) # gráficos básicos
require(dplyr) # librería de manipulación de datos
require(caret) # para las predicciones y transformaciones

#-----#
pairs(training[c(x_4, y)], # ver las correlaciones
      diag.panel = panel.hist,
      upper.panel = panel.cor,
      lower.panel = panel.smooth,
      main = "RELACIONES 2 A 2 ENTRE LAS FEATURES")

#-----#
pairs(training[c(X_all, y)], # ver las correlaciones
      diag.panel = panel.hist,
      upper.panel = panel.cor,
      lower.panel = panel.smooth,
      main = "RELACIONES 2 A 2 ENTRE LAS FEATURES")

```

El último plot presenta el siguiente aspecto. En la práctica tendrá que hacer zoom varias veces, por ejemplo por grupos de 4 x 4 features, como en el plot anterior del código.



Sin duda, un número de features demasiado grande

La invocación del paquete no plantea ninguna dificultad particular. Es posible definir el número de procesos paralelos que va a utilizar para gestionar el rendimiento del algoritmo en su máquina.

Se calcula la matriz de información mutua entre las variables y se expone a través de una representación de tipo *heatmap*, con colores oscuros y calientes o claros y más bien fríos.

La matriz de información mutua se denomina habitualmente **MiM**.

```

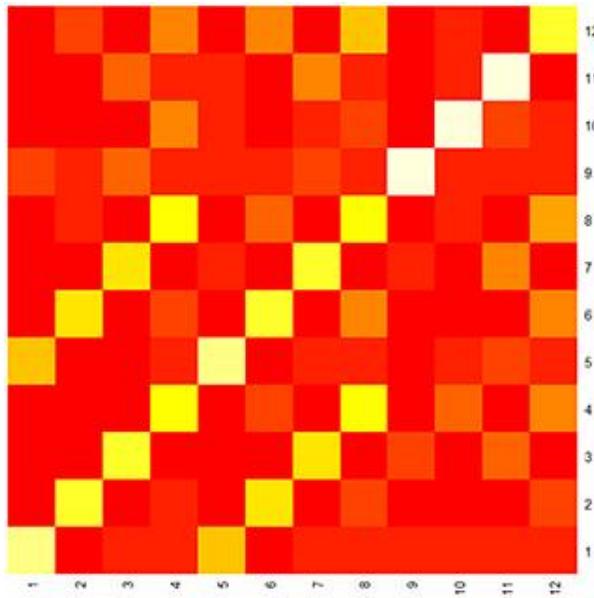
if(require("mRMRe") == FALSE) install.packages("mRMRe")
library(mRMRe)      # mRMRe permite la paralelización de mRMR
set.thread.count(4) # nivel de paralelismo

df <- data.frame(data) # el paquete necesita data.frames

df$clase=as.ordered(df$clase) # mRMR solo trata
                                # factores ordenados
dd <- mRMR.data(df)
#-----#
# utilizando la función de acceso a la información del paquete
# podemos obtener la matriz de información mutua entre
# las variables numéricas

M <-matrix(mim(subsetData(dd, 1:12, 1:12)), nrow = 12, ncol =12)
print(M)
diag(M) <- 0
diag(M) <- max(c(M))
print(M)
heatmap(M, Rowv = NA, Colv = NA) # heatmap

```



Heatmap de la MiM, 12 features

Cuando la clase es oscura y caliente, significa poca información mutua.

La diagonal es clara: una variable comparte una gran información mutua consigo misma!

Se pone de relieve cierta redundancia entre las series de variables **v1 v2 v3 v4 y v5 v6 v7 v8**.

A continuación vamos a entrar de lleno en el asunto y a pedir obtener **una composición de features óptima y reducida**: nuestro subconjunto s.

El algoritmo no pretende encontrar sistemáticamente el óptimo, de modo que debería pedirle varias soluciones posibles y realizar su propia elección utilizando las medidas de distancia y los algoritmos de predicción que elija.

Vamos a pedirle que nos construya seis soluciones (seis conjuntos s candidatos).

Nos corresponde a nosotros determinar el número de features que nos gustaría mantener. A continuación tendremos que proceder como con k-means o k-NN para optimizar este número en función de los resultados de las predicciones.

Vamos a ser ambiciosos y a tratar pasar de 12 features a 3 features, esperando conservar un máximo nivel de calidad en nuestras predicciones.

Observe que disminuir el número de features sin perder demasiada accuracy disminuye enormemente el riesgo de sobreajuste (overfitting).

```
num_sol <- 6
num_feature <- 3
ensemble <- mRMR.ensemble(data = dd,
                           target_indices = c(13),
                           solution_count = num_sol,
                           feature_count = num_feature)

solutions(ensemble)
```

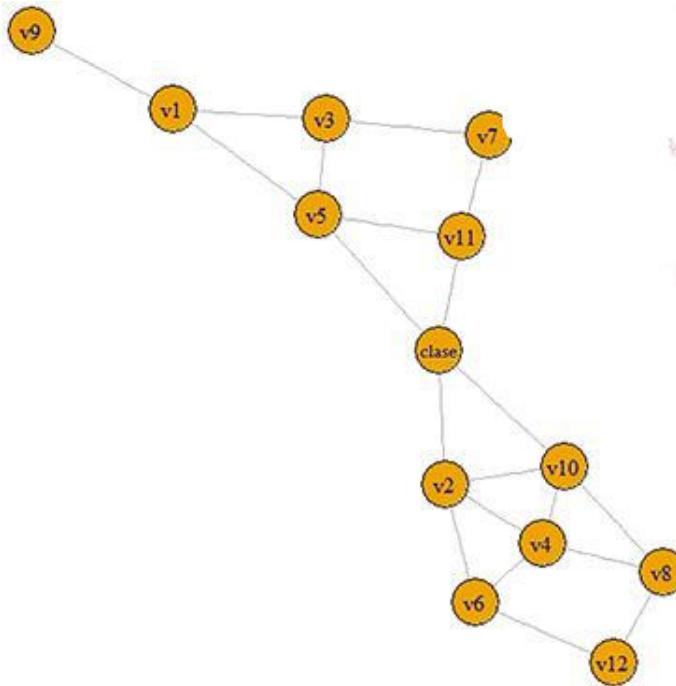
```
[,1] [,2] [,3] [,4] [,5] [,6]
[1,]   11    8    6    4    2   10
[2,]   10    5    8    5    5    5
[3,]    6    2    5    6    7    6
```

Obtenemos seis soluciones propuestas que contienen tres features cada una.

A continuación vamos a observar las relaciones directas entre las features.

```
net      <- new( "mRMRe.Network",
                 data = dd,
                 target_indices = c(13),
                 levels = c(2, 1),
                 layers = 3)

visualize(net)
```



Red de dependencia

Se confirma la existencia de dos subredes de dependencia en torno a la variable objetivo.

En las seis soluciones propuestas, se confirma siempre una mezcla entre ambas subredes, lo cual resulta intuitivamente comprensible.

A continuación vamos a realizar predicciones a través de las seis composiciones de features propuestas, para seleccionar la más eficaz, empezando por definir una 10Fold-CrossValidation para controlar el overfitting.

```
library(randomForest)
fitControl <- trainControl( ## 10-fold CV
  method = "repeatedcv",
  number = 10,
  ## 10 veces
  repeats = 10)
```

Hemos escogido el modelo RandomForest para limitar tener que realizar hipótesis sobre los datos.

```
for (i in 1:num_sol) {
  set.seed(10)
  features_selection <- solutions(ensemble)[[1]]
  ft <- c(features_selection[,i],13)

  m <- randomForest(clase ~ ., data = training[ft],
                     importance=TRUE, cv = fitControl)
  pred_1 <- predict(m,newdata = test[ft])
  c <- confusionMatrix(pred_1,# calificación de la predicción
                        validation[ft]$clase)
  print(features_selection[,i])
  print(c[[3]][1]) # extrae accuracy de la matriz de confusión
```

}

```
[1] 11 10  6
Accuracy
0.65
[1] 8 5 2
Accuracy
0.9
[1] 6 8 5
Accuracy
0.8
[1] 4 5 6
Accuracy
0.85
[1] 2 5 7    <-- la mejor accuracy
Accuracy      <--
0.95      <--
[1] 10 5  6
Accuracy
0.9
```

El conjunto de features v2,v5,v7 obtiene una accuracy de 0.95.

Si hacemos el mismo cálculo con RandomForest y las 12 variables, obtenemos una accuracy de 0.9, que hemos mejorado con solo tres variables!

Nuestra reducción de dimensiones ha funcionado perfectamente.

Cuando estudiamos la clasificación de la importancia de features realizada por RandomForest con las 12 variables, lo cual es una **técnica muy común** que conviene recordar, encontramos la siguiente clasificación.

```
round(sort(importance(modelFit)[,"MeanDecreaseGini"]),
decreasing = TRUE),2)
```

v2	v1	v9	v10	v6	v5	v3	v4	v8	v7	v11	v12
7.48	6.99	6.65	6.11	4.70	4.15	2.62	2.08	1.04	1.02	0.84	0.61
----	----	----	----	----	----	----	----	----	----	----	----

La comparación de ambos resultados resulta muy esclarecedora. La variable más importante (v2) está presente en nuestra tripleta ganadora (v2,v5,v7), aunque las variables v5 y v7 están lejos en términos de importancia.

Un **buen panel de una variable fuerte (strong) y de dos variables débiles (weak)** supera no solo un panel con todas las variables, sino también paneles compuestos con las variables más fuertes.

Conviene que memorice esta conclusión, pues numerosos data scientists juniors cometan sistemáticamente el error de utilizar únicamente las variables más fuertes.

Complementos útiles

GAM: generalización de LM/GLM

Para ir más lejos con el modelo lineal vamos a estudiar su generalización, que forma parte, por motivos evidentes, del conjunto de herramientas de muchos data scientists.

El modelo aditivo generalizado (GAM) va más allá que el modelo GLM en la generalización de los modelos lineales, de modo que debería tenerlo en cuenta desde el inicio de su trabajo.

Se trata de un modelo muy manejable, con grandes posibilidades, eficaz en términos de rendimiento, cuyas cualidades pueden incitar a comportamientos de sobreajuste, pero que permite hacer lo que podríamos denominar un «traje a medida» sobre los datos. Aunque es también un modelo que nos incita a comprender y analizar nuestros datos y que resulta muy interesante desde una óptica audaz y exploratoria.

En el modelo lineal común, imaginamos que la variable de respuesta y es una ley normal de media μ y de varianza σ^2 con: $y \sim \sum_{j=1}^p \beta_j x_j + \beta_0$

En el modelo GLM, se presentan otros conceptos, pero manteniendo la linealidad sobre los coeficientes β , que son los parámetros que hay que descubrir!

En este modelo, la variable de respuesta no puede corresponderse con una distribución normal, sino que debe formar parte de la «gama» de distribuciones de la «familia exponencial» que comprende, entre otras, las siguientes distribuciones: Normal, Normal multivariante, Log-Normal, inversa Gaussiana, Gamma, Gamma-Normal, Gamma-inversa, Exponencial, Khi-cuadrado y Khi-cuadrado inversa, Béta, Dirichlet, Bernoulli, Multi-Bernoulli, Poisson y Geométrica. Preste atención, pues ciertas distribuciones no pertenecen a esta familia, salvo si bloquean un parámetro. Este es el caso por ejemplo de la distribución multinomial. La distribución uniforme no forma parte de esta familia, ni las distribuciones logísticas e hipergeométricas o las mezclas de Gaussianas.

En el caso del modelo GLM, se exhibe una función *link* llamada clásicamente $g(\cdot)$, con:

$$g(y) \sim \sum_{j=1}^p \beta_j x_j + \beta_0$$

La idea consiste simplemente en estimar la transformada de y , en lugar de y .

En relación con el caso LM, basta con considerar que g es la identidad. A menudo se llama a η la transformada de la media (esperanza matemática): $\eta = g(\mu)$

y, por tanto: $E(y) = \mu = g^{-1}(\eta)$

g^{-1} es la función link inversa.

El modelo GAM generaliza GLM introduciendo funciones de transformación a nivel de las variables explicativas, típicamente funciones de alisado (*smoothing*) $s_j(\cdot)$.

$$g(y) \sim \sum_{j=1}^p \beta_j s_j(x_j) + \beta_0$$

Las funciones $s_j(\cdot)$ pueden ser de naturalezas muy diferentes, debido a que las variables no son necesariamente similares.

La invocación del algoritmo se realiza de manera sencilla, con una sintaxis del tipo:

```
## código ficticio:  
library(mgcv)  
model <- gam(y~ s(x1) + s(x2) + s(x3), data = df)
```

Podemos influir en la naturaleza de las funciones `s` imponiendo, por ejemplo, el hecho de utilizar una función de regresión cúbica si comprobamos que las variaciones de las variables se prestan: `s(x1, bs = "cr")`.

Puede resultar adecuado crear una fórmula de regresión a la vez con una función, una variable y su transformación:

$$y \sim s(x1) + x1$$

También podemos utilizar un alisado de varias variables:

$$\sim te(x1, x2)$$

Conviene consultar la ayuda del paquete en la siguiente entrada: **Smooth terms in GAM**, que muestra el ámbito y el uso de las funciones de alisado disponibles.

Tenga en mente un caso práctico interesante correspondiente al caso de los **datos geográficos sobre una esfera (nuestro planeta)**: `bs = "sos"`.

Vamos a abordar a continuación los objetos y los problemas que incluyen dimensiones espaciales (1D, 2D, 3D...). De momento, recuerde que la noción de alisado (smoothing) se aplica de manera natural a este tipo de contexto, de modo que es, por ejemplo, posible imaginar que la temperatura entre dos puntos cercanos en un mapa del tiempo puede expresarse como el resultado de un alisado, de una interpolación entre las temperaturas de estos dos puntos.

Manipulación de imágenes

La manipulación de imágenes permite extraer diversas features de estas o preprocesarlas antes de aplicar diversos algoritmos (típicamente, un procesamiento **a través de una red neuronal**).

Vamos a abordar estas manipulaciones de manera sintáctica y sin florituras; la idea consiste en incitarle a ir más lejos descubriendo la simplicidad de herramientas que ya tiene a su disposición.

Manipular imágenes consiste, también, en evolucionar en un mundo **espacial** en dos y tres dimensiones, como en geografía, donde dos dimensiones son las variables explicativas de la tercera (intensidad, color). Otra similaridad con los usos geográficos y geofísicos es que las dimensiones espaciales deberían considerarse de manera conjunta a través de su covarianza, por ejemplo: la distancia entre dos ojos es más representativa de las características de una cara que la distancia de los valores de sus coordenadas respectivas.

En una imagen se ocultan, como en geografía, otras variables latentes y, en ocasiones, dimensionales (por ejemplo: la profundidad). Del mismo modo que el agua se encuentra a menudo en el fondo de un barranco, las sombras de una cara aparecen más a menudo en los huecos de la cara que sobre la punta de la nariz.

Nuestro trabajo sobre las dimensiones 2D y 3D empieza con un trabajo sobre la imagen, y a continuación nos encadenaremos con otros aspectos de la espacialidad. Recuerde a su vez que existe todo un continuo de pensamiento entre las dimensiones 1D/2D de las series temporales y las dimensiones 2D/3D de la imagen, de la geografía, de la propagación del sonido o de las ondas. Además, las técnicas de modelado, de optimización o de predicción adquiridas en algunos de estos campos de investigación resultarán útiles en otros campos (modulando algunos cambios de vocabulario y sus restricciones).

1. Creación, visualización, lectura y escritura de imágenes

Una imagen en blanco y negro puede almacenarse como una matriz de píxeles (altura y anchura de la imagen). Cada píxel posee una intensidad más o menos fuerte (su intensidad de gris), típicamente codificada por un valor que va de 0 a 255 (incluido) en el caso de imágenes utilizadas en los ejemplos que encontrará en el tratamiento de imágenes en data sciences.

Una imagen en color RGB es una tabla que contiene tres matrices similares a la de una imagen en blanco y negro. Cada matriz representa un plano de colores, cuyas celdas contienen la intensidad de uno de los tres colores: Rojo (*Red*), Verde (*Green*), Azul (*Blue*). El píxel final se muestra como el color resultante de la mezcla de los tres colores con sus respectivas intensidades.

Existen otros modelos de colorimetría, aunque este es el utilizado con más frecuencia en el contexto de las data sciences. Los demás modelos son, en ocasiones, más adecuados en términos de espacio colorimétrico y de capacidad para corresponderse a colores tal y como los percibimos en distintos soportes (papel, pantalla, ropa, fotografía, visión en el interior o en el exterior...).

El paquete que vamos a utilizar aquí se llama **EBImage**, y resulta interesante por diversos motivos: facilidad de uso, existencia de funciones avanzadas de filtrado y existencia de tratamientos útiles en data sciences. De hecho, este paquete y sus «paquetes amigos» están particularmente adaptados al procesamiento de imágenes que representan objetos como células vivas. Preste atención: en este paquete los valores que se muestran de los píxeles están codificados basándose en intensidades de 0 a 1, aunque se aplican filtros sobre el conjunto de valores numéricos.

Nos limitaremos a un uso básico de estos paquetes focalizándonos en el tratamiento de imágenes en lugar de las data sciences específicas de las imágenes. Las funciones más sofisticadas de estos paquetes muestran, de hecho, técnicas que se abordan en otros campos, pero que pueden aplicarse a imágenes antes que a otras cosas.

Como ejemplo de creación de imagen, vamos a fabricar dos imágenes de referencia: un ruido blanco sobre la gama de los grises, y un ruido gaussiano sobre la gama de los colores. El término ruido blanco posee el mismo significado que en el caso de las series temporales que hemos estudiado previamente, o que en el caso de un sonido. ¡Evidentemente, estos ruidos blancos no son imágenes blancas!

El siguiente código dimensiona una imagen en blanco y negro, y a continuación fabrica una matriz de valores reales repartidos siguiendo una ley uniforme de 0 a 1.

```
getwd()           # comprobemos la ubicación de los archivos
library (EBImage) # librería de tratamiento de imágenes
help(EBImage)     # para obtener la ayuda sobre el paquete

## creación de una imagen de niveles de gris - ruido uniforme      ##
ancho      <- 333
alto       <- 111
planos     <- 1  # gris: 1 plano de color
dimension   <- c(ancho , alto , planos)

                  # creación de los píxeles
num        <- ancho * alto * planos
data       <- array(rnunif (num),
                  dim =dimension)

                  # creación de la imagen
im <- Image(data,
            dim = dimension,
            colormode='Grayscale')
display(im)      # visualización en nuestra aplicación
```

R invoca a su aplicación por defecto y, tras aceptar *leer un objeto local* a través de él, le permite ver su imagen.

Utilizando la opción "**raster**", visualiza la imagen directamente en la zona de plot gráfica de RStudio.

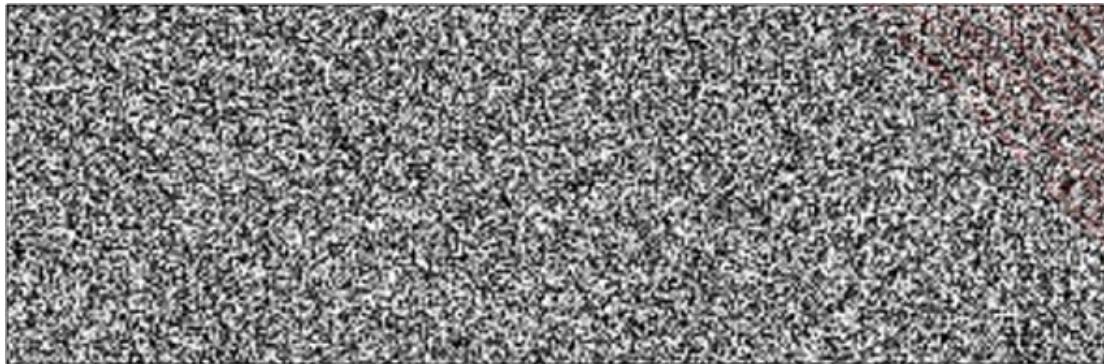
```
display(resize(im, 2*ancho, 2*alto),  # escala de visualización
       method = "raster",          # visualización en R
       all = TRUE)
print(im)      # características im
```

```
Image
colorMode    : Grayscale
storage.mode : double
dim         : 333 111 1
frames.total : 1
frames.render: 1

imageData(object)[1:5,1:5,1]
 [,1]      [,2]      [,3]      [,4]      [,5]
[1,] 0.5529527 0.28992810 0.05774661 0.8922491 0.6057984
[2,] 0.3754047 0.08185318 0.24690566 0.2676992 0.2488014
[3,] 0.8961590 0.31552319 0.59251919 0.2728073 0.9613687
[4,] 0.5960973 0.03378383 0.21456323 0.4144032 0.6904279
[5,] 0.7260245 0.70670889 0.35211329 0.9671756 0.8944150
```

Comprobamos cómo hemos fabricado una imagen sobre una gama de grises, de las dimensiones deseadas, con un único plano. Se muestran los valores de 25 píxeles sobre los 333x111 píxeles de la imagen.

Un ruido blanco tiene el siguiente aspecto:



Ruido blanco

Podemos crear un ruido gaussiano sobre la gama de los colores con un código similar.

```
## Ídem, pero creación de una imagen en color - ruido gaussiano ##

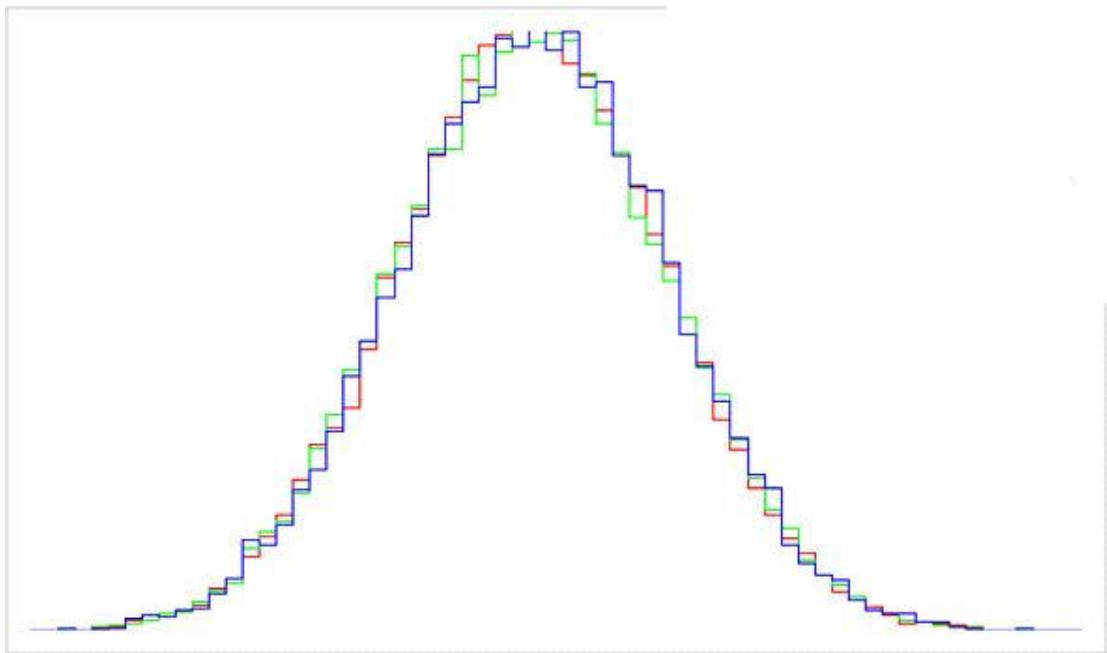
ancho      <- 200
alto       <- 111
planos     <- 3      # número de planos de colores
dimension  <- c(ancho , alto , planos)

# creación de los píxeles
num        <- ancho * alto * planos
data       <- array(rnorm(num,mean = 0.5, sd =0.15),
                  dim =dimension)

# creación de la imagen
im <- Image(data,
            dim = dimension,
            colormode='Color')

# visualización
display(resize(im, 2*ancho, 2*alto),    # escala de visualización
        method = "raster",           # visualización en R
        all = TRUE)
print(im)      # características im
hist(im)
```

Obtenemos una mezcla de puntos de colores cuyas características merecen que nos detengamos un momento, pero que en una primera aproximación poseen el reparto deseado de los colores primarios (la implementación de la función **hist()** sobre los objetos de la clase **Image** es bastante minimalista, pero no importa).



Reparto gaussiano de intensidades de colores

Resulta práctico salvar nuestra imagen, el paquete adapta automáticamente el formato real con que se guarda la imagen a la extensión de su nombre de archivo.

```
## escritura de imágenes en el disco ##  
writeImage(im, 'test.tiff') # guardar una imagen sin compresión  
writeImage(im, 'test.jpeg', quality=80) # guardar una imagen jpg
```

Como sin duda sabrá, el formato **.tiff** es fiel, mientras que el formato **.jpg** está comprimido, lo que produce una imagen diferente con cierto alisado.

A continuación vamos a leer una foto codificada en **.jpg**. El animal de compañía del autor ha participado en la producción de este libro gracias su apoyo psicológico incondicional: llamada Mika, va a mostrarse en RStudio.

```
## lectura de una imagen en el disco ##  
im <- readImage("mika.jpg")  
print(im)  
dim(im)  
ancho <- dim(im)[1]  
alto <- dim(im)[2]  
planos <- dim(im)[3]  
dimension <- c(ancho, alto, planos)  
display(resize(im, 1.5*ancho, 1.5*alto), # escala de visualización  
       method = "raster", # visualización en R  
       all = TRUE)
```

```
Image  
colorMode : Color  
storage.mode : double  
dim : 3456 3456 3
```

```
frames.total : 3  
frames.render: 1
```

En disco, esta imagen a color de 3456 x 3456 píxeles está comprimida en un tamaño de 2.45 MB, pero en memoria esta imagen, llamada **im**, ocupa 273.4 MB!

Cuando manipula un número importante de imágenes en un contexto de machine learning, siempre hay que evaluar los volúmenes y el rendimiento correspondientes antes de decidir realizar un eventual preprocesamiento para hacerlas compatibles con nuestras restricciones de hardware.



iMika pesa 273.4 MB en memoria!

Para manipular dicha imagen fuera del paquete, basta con transformar la imagen en una tabla R clásica.

```
data <- imageData(im) # un array compatible con cualquier programa  
str(data)           # dimensión y primeros valores de data
```

```
num [1:3456, 1:3456, 1:3] 1 1 1 1 1 1 1 1 1 1 ...
```

Su tamaño en memoria es idéntico.

Para comparar imágenes antes y después del procesamiento, vamos a crear una pequeña función de doble visualización que se utilizará a menudo en el siguiente paquete.

```
## función display comparación antes VS después      ##  
## implementación trivial, si la imagen 2 no es del mismo tamaño ##  
## entonces solo se muestra esta                      ##
```

```

f_display2 <- function (ima, imb) {
  imf <- imb                                # muestra imb por defecto
  ancho <- dim(imb)[1]
  alto <- dim(imb)[2]

  combinar <- (dim(ima)[1] == ancho) &      # ¿mismo ancho?
             (dim(ima)[2] == alto)        # ¿mismo alto?
  if (combinar) {imf <- combine(ima,imb)} # si sí, se combina

  ## fabricación de una imgen con 2 frames y visualización

  display(resize(imf, 2*ancho, 2*alto), # escala de visualización = 2
          method = "raster",           # visualización en R
          all = TRUE)
}

```

Esta función utiliza la función **combine()**, que resulta muy útil para asociar dos imágenes.

Preste atención: la opción **raster**, que exige mostrar la imagen en la zona de plot de R o RStudio, produce una visualización muy lenta para imágenes pesadas: suprima esta opción si su equipo es muy lento en visualización (esta se mostrará en la aplicación por defecto, pero rápidamente).

2. Transformaciones de imágenes

La sintaxis utilizada en este párrafo es muy simple, aunque debería estudiarse con atención, pues lo que sigue a continuación no es trivial, ya que vamos a aplicar diversos filtros cuyos efectos pueden no parecer triviales de anticipar. No dude en jugar un poco más con estas funciones para comprender su comportamiento real.

a. Ejemplos de manipulación del color y de las intensidades

La transformación se realiza con una sola operación «aritmética» (vectorizada).

Cambio de luminosidad

```

## luminosidad: + o -
im <- readImage("mika.jpg")
im1 <- im - 0.3
f_display2(im,im1)

```



Imagen ensombrecida

Debido a esta transformación, los píxeles están más sombreados y poseen valores diferentes, pues parecen haberse ensombrecido o desplazado hacia el negro. Realizando la transformación inversa, se llevará la sorpresa de obtener la información que parecía haberse perdido: esta transformación es reversible, la pérdida de información solo afecta a la visualización. Hemos obtenido valores negativos en al menos uno de los tres planos de colores que ensombrecen la imagen, pues al menos una de las componentes de color es, ahora, negativa (preste atención: salvar esta imagen en un formato de imagen le puede llevar a perder información no representable).

Observe con atención que, para que un píxel se vuelva de color negro, es preciso que sus tres valores sean inferiores o iguales a 0.

El siguiente código tiene como objetivo llamar su atención sobre la codificación de los píxeles y la manera de filtrarlos, de modo que invierta cierto tiempo para comprender mejor todo esto.

```

ancho  <- dim(im)[1]
alto   <- dim(im)[2]
planos <- dim(im)[3]
dimension <- c(ancho , alto , planos)

sum(im < 1e6 )/3      # número de píxeles
ancho*alto            # ídem
sum(im < 0.3 )/3      # número de píxeles con sombra a oscurecer
sum(im1 < 0  )/3      # número de píxeles ensombrecidos

                                         # creación de una imagen blanca
data  <- array(1, dim =dimension) # 3*1 = blanco
im_blanca <- Image(data,
                     dim = dimension,    # dimensiones de mika
                     colormode='Color')
im_negra  <- im_blanca - 1      # creación de una imagen negra
                                         # (inútil salvo para el ejemplo)
im2 <-im_blanca
im2[which(im < 0.3 )] <- 0

f_display2(im,im2)

```

```
[1] 11943936  
[1] 11943936  
[1] 3327928  
[1] 3327928
```



Píxeles con una componente oscurecida

Cambio de contraste

Aumentemos el contraste de una imagen.

```
## contraste: *  
im <- readImage("aix.jpg")  
im1 <- im * 1.5  
f_display2(im,im1)
```



Contraste aumentado

Corrección gamma

Esta corrección la conocen bien los fotógrafos, y permite trabajar sobre los contrastes de manera no lineal.

```
## corrección gamma: ^
im <- readImage("jardin.jpg")
im1 <- im ^ 2
f_display2(im,im1)
```



Corrección gamma

b. Ejemplos de manipulación de la geometría de la imagen

Empezaremos recortando un rectángulo de la imagen.

```
## recorte/cropping, evidentemente:
im <- readImage("gatito.jpg")
dim(im)
ancho    <- dim(im)[1]
alto     <- dim(im)[2]
planos   <- dim(im)[3]
im1 <- im[1:200, 1:200, ] # observe la segunda coma
f_display2(im,resize(im1,ancho,alto))
```



Cropping de una parte de la imagen

El paquete proporciona también inversiones verticales y horizontales.

```
## inversión
im <- readImage("gaviotas.jpg")
im1 <- flip(im)
f_display2(im,im1)

im <- readImage("laguna.jpg")
im1 = flop(im)
f_display2(im,im1)
```



Flip



Flop

Es posible realizar traslaciones y rotaciones.

```
## translación
im1 <- translate(im, c(100, 100))
f_display2(im,im1)
## rotación
im <- readImage("parajo.jpg")
im1 <- rotate(im,45)
f_display2(im,im1)
```



Rotación sentido horario 45°

Podemos realizar una transformación afín controlada por una matriz.

```
#transformación afín cualquiera
im <- readImage("rosas.jpg")
trans_afin <- matrix(c(
    0.5, 0.1,
    0, -0.3,
    0.5, 3
```

```
) ,  
nrow=3)  
  
im1 <- affine(im, trans_afin)  
f_display2(im,im1)
```



Transformación afín

c. Aplicación de filtros sobre las imágenes

Hay disponibles muchas maneras de procesar imágenes. He aquí ejemplos para animarle a descubrirlas...

La ecualización, del inglés *equalization*, tiene como objetivo un mejor reparto de las intensidades (aunque impacta en los colores).

```
## análisis de la imagen y ecualización posterior ##  
  
hist(im) # análisis de los colores de la imagen  
im1= equalize(im)  
hist(im1)  
f_display2(im,im1)
```



Ecuación

Para reducir el ruido, a menudo se utiliza otro procesamiento, como la búsqueda de contornos, o se aplica un filtro de paso medio.

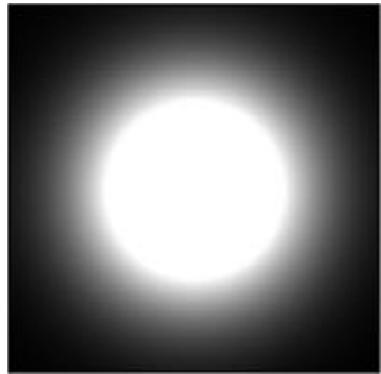
```
## filtro de paso medio y suprimir las intensidades de píxel ##  
im2 <- medianFilter(im1, 1.1) > 0.6  
f_display2(im1,im2)
```



Filtro de paso medio y conservación de los píxeles intensos

Se nos abren muchas otras posibilidades, como la fabricación de todo tipo de filtros, por ejemplo mediante la función **makeBrush**, que produce matrices a nuestro antojo y que puede aplicar sobre sus imágenes. Aquí, una matriz con valores repartidos de forma circular gaussiana.

```
display(5000*makeBrush(101, shape='Gaussian', sigma=20))
```



«*Brush*» gaussiano

Utilizando alguno de los tipos de técnicas vistos en estos últimos párrafos, puede empezar a manipular sus imágenes para realizar diversos preprocesamientos o extraer las features a su conveniencia.

Cómo crear una muestra: LHS (hipercubo latino)

Para crear una muestra eficaz, en particular cuando el problema posee numerosas dimensiones, es tentador fabricar una muestra donde se encuentre un mínimo de interacciones entre las variables. Evidentemente, este tipo de muestra resulta tanto más útil cuando se sospecha cierta covarianza entre ellas, pues este modo de muestreo no va a permitir capturar fácilmente el efecto de las interacciones entre las variables.

En el caso de problemas de pocas dimensiones (2D, 3D) pero con muchos puntos por paquete de observaciones (imagen, geografía...), estos métodos de muestreo adquieren todo su sentido.

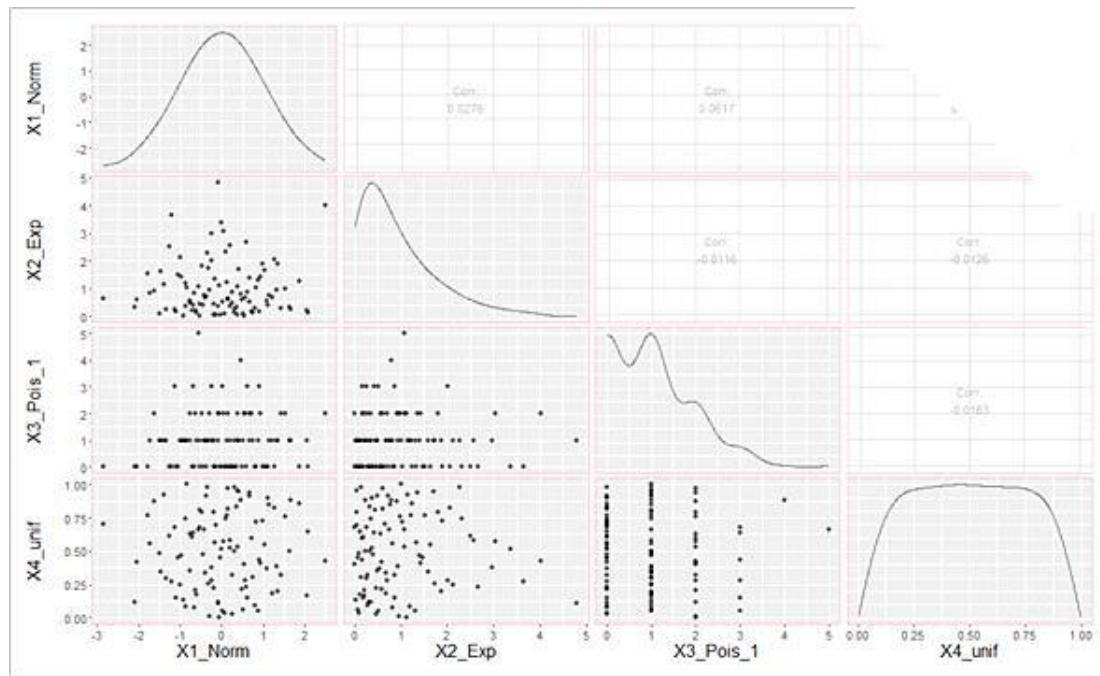
Los jugadores de ajedrez conocen bien el problema de las ocho reinas sobre un tablero, que consiste en situar sobre el tablero a ocho reinas sin que se ataquen entre ellas. Los cuadrados mágicos, los Sudokus y las ocho reinas son problemas de construcción de un reparto de objetos bajo restricciones precisas y, a menudo, mínimas.

LHS (*Latin Hypercube Sampling*) recuerda al problema de las «torres» de un juego de ajedrez, donde cada torre no debe atacar a las demás.

La idea es sencilla: se segmenta cada variable y se autoriza una sola observación por cada intersección (cada baldosa del hipercubo general) compuesta por una arista y que corresponde con un segmento de una dimensión. La observación se ubica aleatoriamente en algún lugar dentro de su propio «pequeño» hipercubo (su baldosa). Cuando se desea obtener un número mayor de observaciones, se reitera este proceso.

El conjunto puede ponerse bajo el control de una función que optimiza la ubicación. En nuestro ejemplo, la media de las distancias entre los puntos se maximiza para garantizar un reparto en el conjunto del espacio.

Tras fabricar el cuadrado con una distribución uniforme por columna, resulta **indispensable** transformar las distribuciones de cada variable para simular la distribución y la amplitud del valor original de la variable (si no se conoce la distribución, al menos hay que corregir la amplitud: imagine que desea obtener notas entre 0 y 20; como la distribución producto está situada entre 0 y 1, nos vemos obligados a multiplicar por 20).



Muestra LHS con distribuciones reintroducidas

El resultado parece satisfactorio; no se introduce ninguna dependencia lineal accidental (pequeños coeficientes de correlación), podríamos por ejemplo trabajar sobre la distribución normal.

Se trata de una técnica que puede resultar muy útil para crear distribuciones de variables antes de aplicar un método de Monte-Carlo. Si el tiempo de procesamiento parece demasiado largo para un importante número de observaciones, será preciso iterar cierta cantidad de veces con un número menor de observaciones (que vamos a agregar en conjunto con `rbind`) antes de transformar las variables (en este orden, si no aplicará la ley de los grandes números y sus distribuciones se normalizarán).

Trabajar sobre datos espaciales

1. Variograma

a. Campo y variable regionalizada

Durante la secuencia del «tiempo» en la televisión, el presentador comenta un mapa sobre el que se anotan las temperaturas, le indica un **campo escalar** (*scalar field*).

En cada ubicación, definida por sus coordenadas en 2D sobre el mapa, existe una temperatura (un valor escalar). Sobre el mapa del tiempo, solo ciertos puntos muestran una temperatura; se trata de **una muestra**: habríamos podido escoger estos puntos utilizando LHS, pero sin duda es preferible indicar las temperaturas en aquellos puntos correspondientes a las principales ciudades.

Para deducir la temperatura que hará en su casa, en el caso de que no habite en una gran ciudad, tendrá que **interpolar** (realizando un **alisado**). Dispone de una infinidad de estrategias más o menos eficaces: un alisado brutal utilizando las dos ciudades más próximas y realizando una media, el mismo alisado realizando una media ponderada por el inverso de las distancias, el uso de tres ciudades que forman un triángulo a su alrededor y el cálculo de los coeficientes de ponderación como el inverso de los coeficientes del baricentro de su ubicación vista como un baricentro de tres ciudades...

Cuando el presentador del tiempo le describe eficazmente la situación de los vientos sobre el mapa, le indica la velocidad del viento y también su dirección: ha definido vectores «viento» en diversos lugares del mapa. En ocasiones se incluyen pequeñas flechas más o menos largas que representan estas velocidades en todos los puntos del mapa. Hablamos de un **campo vectorial** (*vector field*).

El piloto de un globo aerostático no se concentrará, sin duda, en los campos en 2D sobre el mapa, sino que deseará conocer la fuerza del viento en su dirección en 3 dimensiones, por ejemplo: el viento a 100 m de altitud en tales coordenadas GPS es ascendente en N-Norte-Oeste, 60 km/h.

La diferencia entre los valores de un campo escalar entre dos puntos puede generar un campo vectorial inducido. Si este campo vectorial es «proporcional» a la variación del campo escalar en todas sus dimensiones, hablamos de **gradiente**: como en el caso de nuestro niño, que buscaba sus huevos de Pascua y que intentaba orientar su velocidad de desplazamiento en el sentido donde «quemaba» cada vez más.

En geografía, meteorología, prospección minera, hablamos a menudo de **variable regionalizada** (VR) para indicar los campos escalares o vectoriales.

El gradiente de un campo escalar está dirigido en cada punto en la dirección de la mayor pendiente, es decir, perpendicularmente a las líneas del iso-valor (es decir, las curvas de nivel en geografía). El **módulo** (la intensidad) del vector en cada punto es el **valor de la pendiente en este punto**. Preste atención: el gradiente apunta desde los valores bajos hacia los valores altos, y a menudo se introduce un signo «-» para invertir la dirección si el fenómeno físico va en el otro sentido (como en el caso de una canica que rueda por una pendiente).

Gradiente, formulación matemática

Consideremos un campo escalar en un espacio 3D: $f(X)$

Recordemos que un vector X se expresa como $X = (x_1, x_2, x_3)^T$ en una base ortonormal de dicho espacio, base compuesta por tres vectores: e_1, e_2, e_3 .

con, recordemos, estos tres vectores expresados en su propia base:

$$e_1 = (1, 0, 0)^T$$

$$e_2 = (0, 1, 0)^T$$

$$e_3 = (0, 0, 1)^T$$

Vamos a introducir la noción de **derivada parcial**. Si no es matemático, imagine la noción de derivada parcial como la derivada de una función considerando que todas las variables, salvo la variable sobre la que se deriva, poseen valores constantes (considerando la función como completamente expresada en función de estas variables).

Por ejemplo, sabiendo calcular las derivadas respecto a las x siguientes:

$$(3x^2)' = 6x$$

$$(7x)' = 7$$

$(5c x)' = 5 c$ para todo c que sea un valor constante que no dependa de x

$(c)' = 0$ para todo c que sea un valor constante que no dependa de x

y planteando la función $g(x_1, x_2, x_3) = 3x_1^2 + 7x_2 + 5x_1 x_2 + 7x_3$, las derivadas parciales de g respecto a x_1 , x_2 y x_3 son respectivamente:

$$\frac{\partial g}{\partial x_1}(x_1, x_2, x_3) = 6x_1 + 5x_2$$

y

$$\frac{\partial g}{\partial x_2}(x_1, x_2, x_3) = 7 + 5x_1$$

y

$$\frac{\partial g}{\partial x_3}(x_1, x_2, x_3) = 7$$

El gradiente de un campo escalar de dimensión 3 se expresa de la siguiente manera:

$$\text{grad}(f) = \frac{\partial f}{\partial x_1}(x_1, x_2, x_3) e_1 + \frac{\partial f}{\partial x_2}(x_1, x_2, x_3) e_2 + \frac{\partial f}{\partial x_3}(x_1, x_2, x_3) e_3$$

Que sería el vector con las tres componentes siguientes:

$$\left(\frac{\partial f}{\partial x_1}(x_1, x_2, x_3), \frac{\partial f}{\partial x_2}(x_1, x_2, x_3), \frac{\partial f}{\partial x_3}(x_1, x_2, x_3) \right)^T$$

A menudo se llama: ∇f

∇ se llama operador nabla (*del* en inglés); podemos imaginarlo como un falso vector cuyas componentes son operadores de derivada parcial:

$$\nabla \approx \left(\frac{\partial}{\partial x_1}, \frac{\partial}{\partial x_2}, \frac{\partial}{\partial x_3} \right)^T$$

De modo que, por ejemplo, sin necesidad de recordar que g es una función de tres coordenadas:

$$\nabla g = \left(\frac{\partial}{\partial x_1}, \frac{\partial}{\partial x_2}, \frac{\partial}{\partial x_3} \right)^T g = \left(\frac{\partial g}{\partial x_1}, \frac{\partial g}{\partial x_2}, \frac{\partial g}{\partial x_3} \right)^T = (6x_1 + 5x_2, 7 + 5x_1, 7)^T$$

Este operador se generaliza a las demás dimensiones.

 Este operador permite definir otras operaciones; nosotros no las necesitaremos en este capítulo, de modo que no nos vamos a extender sobre ello:

Planteando F , campo vectorial: $F(X) = (F_1(X), F_2(X), F_3(X))^T$, la divergencia del campo vectorial (es un campo escalar) se escribe: $\text{div}(F) = \nabla \cdot F$ y el rotacional del campo (es un campo vectorial, en inglés «curl»): $\text{rot}(F) = \nabla \wedge F$

El signo \wedge significa «producto vectorial».

El laplaciano de un campo escalar es un campo escalar que se obtiene aplicando dos veces el operador nabla:

$$\Delta f = \nabla^2 f = \frac{\partial^2 f}{\partial x_1^2} + \frac{\partial^2 f}{\partial x_2^2} + \frac{\partial^2 f}{\partial x_3^2}$$

Es una suma de segundas derivadas parciales.

Una aplicación práctica del laplaciano

Del mismo modo que una simple derivada en un punto representa la pendiente de una curva en este punto, el gradiente en un punto representa la mayor pendiente de una hipersuperficie en este punto. La segunda derivada de una función continua proporciona información acerca de la curva y del aspecto de sus cambios de pendiente; es también el caso del laplaciano.

Si consideramos una imagen en gama de grises, es decir, un campo escalar en dos dimensiones, el cálculo del laplaciano en cualquier punto de la imagen devuelve información acerca de los cambios brutales de intensidad de la imagen. Los puntos donde el laplaciano cambia de signo se corresponden más o menos con los puntos límites de los contornos de la imagen. Es el espíritu del filtro laplaciano que vamos a ver a continuación.

Desafortunadamente, una imagen puede tener ruido y el cálculo del laplaciano en un contexto discreto no es más que una aproximación.

Este último punto resulta importante de comprender: el cálculo de la variación, y por tanto de la gradiente, se realiza «por etapas» sucesivas. Escribamos una «aberración» matemática, pero que puede ayudarnos a

comprenderlo: en las expresiones $\frac{\partial}{\partial x_1}$ y $\frac{\partial}{\partial x_2}$ tendríamos $\partial x_1 \approx \partial x_2 \approx 1$ si calculamos la variación

saltando sobre otro píxel adyacente vertical u horizontalmente, y $\partial x_1 \approx \partial x_2 \approx 2$ si calculamos la variación entre los dos píxeles adyacentes a la derecha y a la izquierda o arriba y abajo del píxel con el que trabajamos.

Si escogemos una aproximación en un único salto ($\partial x_1 \approx \partial x_2 \approx 1$), entonces las derivadas parciales se aproximan de la siguiente manera, en el caso discreto de las imágenes:

$$\frac{\partial f}{\partial x_1}(x_1, x_2) \approx \frac{f(x_1+1, x_2) - f(x_1, x_2)}{1} = f(x_1 + 1, x_2) - f(x_1, x_2)$$

$$\frac{\partial f}{\partial x_2}(x_1, x_2) \approx \frac{f(x_1, x_2+1) - f(x_1, x_2)}{1} = f(x_1, x_2 + 1) - f(x_1, x_2)$$

No es una aproximación demasiado satisfactoria, pues su fórmula no es simétrica, aunque funciona bien para calcular un laplaciano creíble (aplicándolo dos veces, tras realizar algunos cálculos):

$$\Delta f(x_1, x_2) = \nabla^2 f = \frac{\partial^2 f}{\partial x_1^2} + \frac{\partial^2 f}{\partial x_2^2} \approx$$

$$f(x_1 + 1, x_2) + f(x_1, x_2 + 1) + f(x_1 - 1, x_2) + f(x_1, x_2 - 1) - 4 f(x_1, x_2)$$

Para calcular la derivada discreta en un punto sobre un eje, también podemos realizar la media del punto anterior (izquierdo o inferior) y del punto posterior (derecho o superior), con $\partial x_1 \approx \partial x_2 \approx 2$

Los matemáticos expresarán esto diciendo que basta con *desarrollar la fórmula de Lagrange de primer orden*. En este caso se obtienen, en el caso discreto de las imágenes, las siguientes aproximaciones:

$$\frac{\partial f}{\partial x_1}(x_1, x_2) \approx \frac{f(x_1 + 1, x_2) - f(x_1 - 1, x_2)}{2}$$

y

$$\frac{\partial f}{\partial x_2}(x_1, x_2) \approx \frac{f(x_1, x_2 + 1) - f(x_1, x_2 - 1)}{2}$$

Estas últimas aproximaciones conviene **recordarlas de memoria**, pues numéricamente hablando, es decir, algorítmicamente hablando, es fácil utilizarlas para simular las derivadas parciales en un espacio de dos dimensiones y resulta trivial generalizarlas a **n dimensiones**. Cuando encuentre un lagrangiano o un laplaciano, siempre podrá aproximarlos rápidamente realizando dicho cálculo discreto.

A continuación vamos a expresar nuestra aproximación del laplaciano en dos dimensiones como un **filtro**.

Teníamos:

$$\Delta f(x_1, x_2) \approx f(x_1 + 1, x_2) + f(x_1, x_2 + 1) + f(x_1 - 1, x_2) + f(x_1, x_2 - 1) - 4 f(x_1, x_2)$$

Vamos a realizar el cálculo en cada punto (x_1, x_2) y obtener una nueva malla con todos los valores aproximados del laplaciano.

Lo que se corresponde con la aplicación del filtro de convolución, que llamaremos K_Δ (como «kernel del laplaciano»), según:

```
0 1 0  
1 -4 1  
0 1 0
```

La operación de aplicación del filtro es una convolución, ya hemos visto esta noción en el caso de las series temporales. Para el caso discreto, realizar una convolución es muy sencillo: en cada punto de la imagen se superpone el centro del filtro sobre el punto y se calculan todos los productos correspondientes miembro a miembro, y a continuación se suman. La operación se indica a menudo con el signo *. Por ejemplo, calculemos la convolución de nuestro filtro **en el** punto marcado por el valor a:

```
1 7 1 2 1 5  
4 8 b 7 6 5      0 1 0  
1 c a d 3 7    (*)  1 -4 1 = b+c+d+e - 4a  
1 5 e 6 4 2      0 1 0  
5 4 2 3 7 8  
1 5 4 2 3 7
```

La operación de convolución se realiza en todos los puntos de la malla, es decir, del campo escalar discreto de los valores de gris de la imagen (olvidemos los bordes para simplificar la exposición).

Tenemos:

$$\Delta f \approx (K_\Delta * f) = (f * K_\Delta)$$

El resultado puede considerarse como una nueva imagen. Para que esta imagen sea correcta visualmente, hay que normalizar todos los valores tras la convolución (por ejemplo, dividiéndolos por el número de celdas del filtro, aquí 9 o 6... o normalizando los valores de gris, del valor que se corresponde con el blanco al valor que se corresponde con el negro).

La aplicación de un **filtro laplaciano extrae los contornos de la imagen**.

Existe toda una literatura acerca de la búsqueda del mejor filtro para extraer los contornos según las condiciones y el uso que se hace de estos contornos. En los anexos encontrará toda una gama de filtros que puede probar.

Aproximando la derivada con la segunda serie de aproximación vista más arriba, se obtiene un filtro más complejo. Como regla general, la suma de los coeficientes de los filtros laplaciános es nula.

En el paquete utilizado más arriba para las imágenes, la creación de filtros estaba apoyada por la función **makeBrush**, cuyo nombre hace referencia al efecto que obtiene aplicando una convolución parcialmente, como cuando se pasa un pincel por una acuarela...

b. Determinación del variograma

Consideremos una variable regionalizada escalar (un campo escalar) sobre dos dimensiones: $z(x)$, con x un vector del espacio.

En lugar de llamarla f , lo habitual es que los autores la llamen z (pues z nos recuerda a la perpendicular a un plano, y por tanto al valor de un escalar en cada lugar del plano).

Esta variable regionalizada puede verse como la «realización» de un proceso estocástico $Z(x)$, variable aleatoria de dos dimensiones, dependiente de un índice doble.

Llamemos t y s a dos lugares del plano (dos vectores-puntos del espacio de las variables explicativas $t = (t_1, t_2)^T$ y $s = (s_1, s_2)^T$).

Podemos querer expresar para estos dos lugares las esperanzas matemáticas y varianzas de Z_t y Z_s , así como su covarianza:

- $E(Z_t)$, que escribiremos $\mu(t)$, como «media en t ».
- $E(Z_s)$, que escribiremos $\mu(s)$.
- $\sigma_{Z_t}^2$, que podemos escribir σ_t^2 o $\text{var}(t)$.
- $\sigma_{Z_s}^2$, que podemos escribir σ_s^2 o $\text{var}(s)$.
- Covarianza entre Z_t y Z_s , que llamaremos: $C(s, t)$ (que puede no estar definida).

Pero, si reflexionamos bien, en el caso de un campo escalar espacial como la altitud por encima del nivel del mar en un lugar determinado de un mapa, estaríamos más interesados en conocer las **variaciones de esta altitud** (desnivel) que en la propia altitud. Ocurre lo mismo en las prospecciones mineras que van a determinar la presencia de un filón en función de la variación de un campo de gravedad (ídem para el filtrado sobre la evolución del gradiente, o la «sacudida» del laplaciano que devuelve el contorno del filón visto desde arriba).

De modo que nos concentraremos en la **varianza de la variable aleatoria entre dos puntos**:

$$\text{var}(Z_t - Z_s)$$

Es habitual plantear:

$$2\gamma(t, s) = \text{var}(Z_t - Z_s)$$

y llamar a esta cantidad variograma; $\gamma(t, s)$ sería el semivariograma.

Esto permite trabajar en cuestiones como: ¿cuál es la desviación típica de las variaciones de desnivel entre dos puntos de un mapa?

Los variogramas en cada punto son herramientas habituales que saben interpretar visualmente los geógrafos, los prospectores mineros...

En las series temporales (procesos estocásticos, índice monodimensional), habíamos estudiado la noción de estacionariedad temporal. En los procesos estocásticos espaciales, existe la misma noción de estacionariedad, pero espacial.

► Ejemplo aproximado que le permitirá convencerse de la relación entre ambas nociones: imagine que ilumina la rueda de un carro con una luz estroboscópica... Cuando esta luz posee una frecuencia proporcional a la frecuencia

correspondiente a la que los radios de la rueda se sustituyen unos a otros, la rueda le parecerá que está inmóvil y la estacionariedad física de la forma de la rueda se pone de relieve.

Como con la función de autocovarianza $C(s, t) = \gamma(s - t)$ en la estacionariedad temporal, que solo depende de la diferencia h entre ambos índices, podemos escribir $h = \|s - t\|$. Se confirma entonces que 2γ solo depende de h en caso de que el espacio sea isotrópico, es decir, que las normas no dependan de las direcciones tomadas (no es preciso exigir aquí una estacionariedad de orden 2).

En el caso estacionario tenemos que:

$$2\gamma(t, s) = C(t, t) + C(s, s) - 2C(s, t)$$

Como de costumbre, llamamos a la estimación del variograma con el nombre variograma **empírico**.

Existen modelos de variogramas (fórmulas tipo), en particular para diferenciar topologías (por ejemplo: el modelo esférico...).

Para terminar, hablemos de lo que hace soñar, el efecto «pepita»:

Y (0) debería tener un valor nulo, aunque de hecho diversos factores como el ruido blanco que envuelve al fenómeno, el sesgo y la varianza de los instrumentos de recopilación de información crean pequeñas variaciones.

Paradójicamente, estos ruidos se ocultan por una verdadera pepita y cuando usted no está en presencia de ella los ruidos reaparecen: estaban ocultos por una pepita...

2. Krigeage (kriging)

También conocido con el nombre: *Wiener-Kolmogorov prediction*

a. La teoría, brevemente

Esta pequeña sección debería leerse paso a paso; hace referencia a ciertas nociones que ya hemos estudiado, pero una lectura demasiado rápida sería, sin duda, infructuosa.

La analogía entre los datos espaciales y los datos temporales AR(1) se encuentra también en la manera de modelarlos como procesos gaussianos (GP) de la siguiente manera:

$$y \sim \sum_{j=1}^p \beta_j x_j + \beta_0 + ?$$

Un kriging model (D. Krige, G. Matheron...) posee la siguiente formulación general, que supera el campo de los datos espaciales:

$$y \sim \sum_{j=1}^p \beta_j x_j + \beta_0 + GP$$

GP es un proceso gaussiano. Ya sabemos que los procesos gaussianos están caracterizados por su media y su covarianza (aquí, esta última se llamará $K(s, t)$, y no $C(s, t)$, para introducir la noción de núcleo). Esto se escribe: $GP(m, K)$.

En el caso del krigeage, se hace referencia a un $GP(0, K)$.

En 2D o 3D, hablamos también de *Gaussian Random Field*.

El kernel (la función de covarianza) encapsula toda la información sobre las dependencias espaciales, que pueden tratarse a través de él. Como exhibe una covarianza, el modelo de variograma y el modelo de núcleo están íntimamente ligados (ver la fórmula anterior).

Típicamente, los núcleos presentan el siguiente aspecto (aunque no siempre):

$$K(s, t) = \sigma^2 e^{-\psi(s-t, \theta)}$$

Se confirma que nos encontramos en el condicionamiento habitual del variograma, pues se utiliza una función ψ (creciente) de $s - t$.

ψ es una función decreciente del parámetro θ (*range*).

Los dos parámetros de K son σ^2 y θ .

Como ejemplo de ψ , se utiliza a menudo el caso isotrópico gaussiano:

$$\psi = \|s - t\|^2 / \theta$$

Cuando las observaciones parecen ruidosas, se agregan valores positivos sobre la diagonal de la matriz de covarianza (kernel) para evitar por ejemplo anomalías u observaciones demasiado cercanas correspondientes a valores muy diferentes, lo que perturba el "alisado". El GP se convierte en un *GP + ruido gaussiano de media nula*.

Esto corresponde el útil parámetro **nugget** de los paquetes de *krigeage*.

LHS y krigeage hacen una buena mezcla, los encontramos juntos en los paquetes.

El aprendizaje supervisado se llama aquí *Gaussian vector conditioning* en el sentido de que permite buscar la media del GP y se busca un núcleo cuya varianza sea resistente a las variaciones de la variable de respuesta (como en el caso del condicionamiento de las matrices antes de utilizarlas en ciertos algoritmos).

b. Implementación en R

En el data.frame creado con LHS, vamos a aplicar una función un poco perturbada y a realizar nuestro entrenamiento y a continuación un test.

La fórmula empleada es la fórmula por defecto del paquete.

Resulta interesante saber que este código utiliza un **algoritmo genético para la optimización**, lo que lo vuelve resistente a ciertas funciones que contienen óptimos extraños.

Con la misma idea, se utiliza el parámetro **nugget** para el ruido, lo que vuelve al código resistente.

La predicción se hará con el modelo «UK», universal, que es el más completo.

```
# kriging                                #
require(DiceKriging)                      #
                                         # una función extraña
y <- abs((10*df[,1])+7*(df[,2])+3*sin(df[,3])+sin(df[,4]))#
df.train <- df[1:70,]#
y.train  <-  y[1:70]
```

```

df.test <- df[71:100,]
y.test <- y[71:100]

m <- km(~ 1,
        # fórmula por defecto
        design = df.train,
        response = y.train,
        optim.method = "gen",      # algoritmo genético
        coef.cov = 1,
        nugget.estim=TRUE,         # para evitar el ruido
        control = list(trace=T,    # para tener todo
                       maxit = 1e4)) # menos de 10000 iteraciones
coef(m)                      # parámetros efectivos

y.pred <- predict(m, newdata = df.test, type = "UK")

qplot(y.test,
      y.pred$mean,
      geom = c("point", "smooth"),
      alpha = I(1/2)
)

```

.... / ... FUNCIONAMIENTO DEL ALGORITMO GENÉTICO

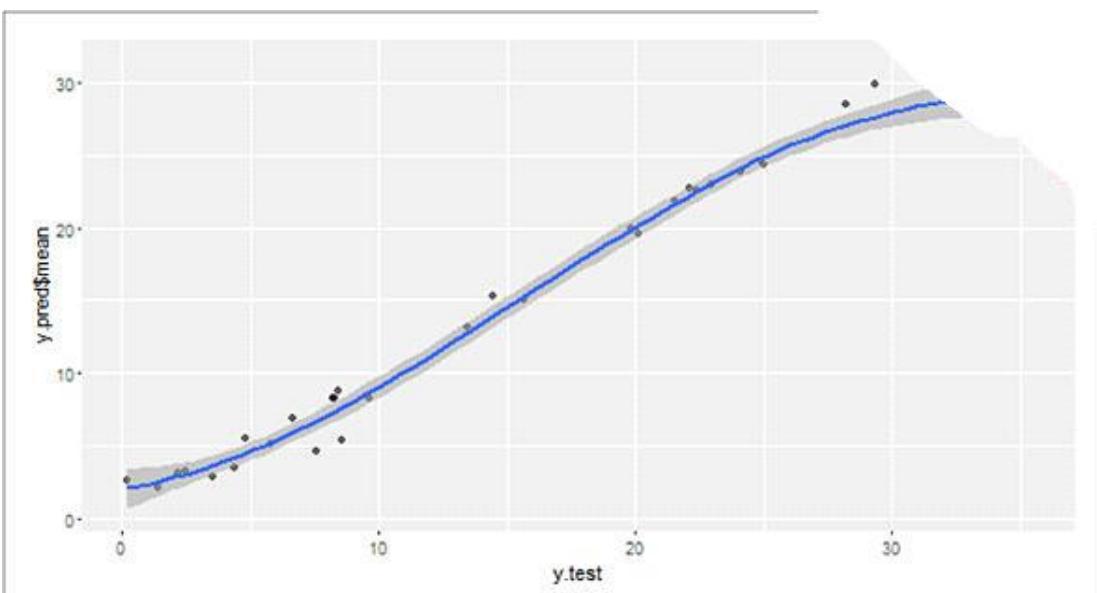
Operators (code number, name, population)

(1) Cloning.....	0
(2) Uniform Mutation.....	1
(3) Boundary Mutation.....	1
(4) Non-Uniform Mutation.....	1
(5) Polytope Crossover.....	1
(6) Simple Crossover.....	2
(7) Whole Non-Uniform Mutation.....	1
(8) Heuristic Crossover.....	2
(9) Local-Minimum Crossover.....	0

.... / ...

Solution Found Generation 1: SÚPER RÁPIDO!

Number of Generations Run 4



La predicción parece correcta

Este método se utiliza en numerosos contextos y su interpretación se comprende, a menudo, entre los ingenieros, lo que le permite dialogar con ellos sobre los variogramas!

Naturalmente, cuando piensa en un campo escalar natural, incluso con ruido, se convierte de facto en un excelente candidato.

Buenas prácticas útiles

1. Trazar una curva ROC

Hemos introducido la curva ROC y la noción de AUC en el primer capítulo. El concepto no es muy complicado, pero en ocasiones existen dificultades para trazar estas curvas de buenas a primeras. Son las cuervas esperadas por sus colegas; he aquí un código «hiperclásico» para producir una.

Para fabricar nuestro conjunto de datos, hemos retomado el código LHS visto más arriba, pero con 300 observaciones en lugar de 100.

Nuestro ejemplo utiliza árboles de decisión. Hemos fabricado una función perturbada para producir clases binarias. Esta parte de código no presenta sorpresas.

```
require(caret)
require(party)

set.seed(1)
y <- 10*df[,1] +                      # una función extraña
    7* df[,2] * rnorm(nrow(df)) +
    3*sin(df[,3]) +
    2*sin(df[,4])*runif(nrow(df))
y <- ifelse(y < 7,0,1)                  # fabricación de clases
sum(y)/length(y)                         # % clases
y <- factor(y)                          # creación factores

df_<- data.frame(df,y)                 # más simple para:
p           <- createDataPartition(y=df_$y,
                                  p= 50/100,
                                  list=FALSE)
training   <- df_[p,]
test       <- df_[-p,]

#-----#
# ctree
m <- ctree(y~.,data=df_)                # árboles
m
y.pred <- predict(m,test)                # predicción
y.test <- test$y
confusionMatrix(y.test,y.pred)           # test
```

Confusion Matrix and Statistics

		Reference
Prediction	0	1
0	92	6
1	14	37

Accuracy : 0.8658

Para mostrar el aspecto clásico de una curva ROC, nos hace falta una accuracy correcta pero no demasiado, como ocurre aquí.

El código es bastante simple, pero hace referencia a los datos de trabajo y a funcionalidades del paquete que en ocasiones vale la pena investigar: este es el interés de este pequeño ejemplo que le hará ganar tiempo.

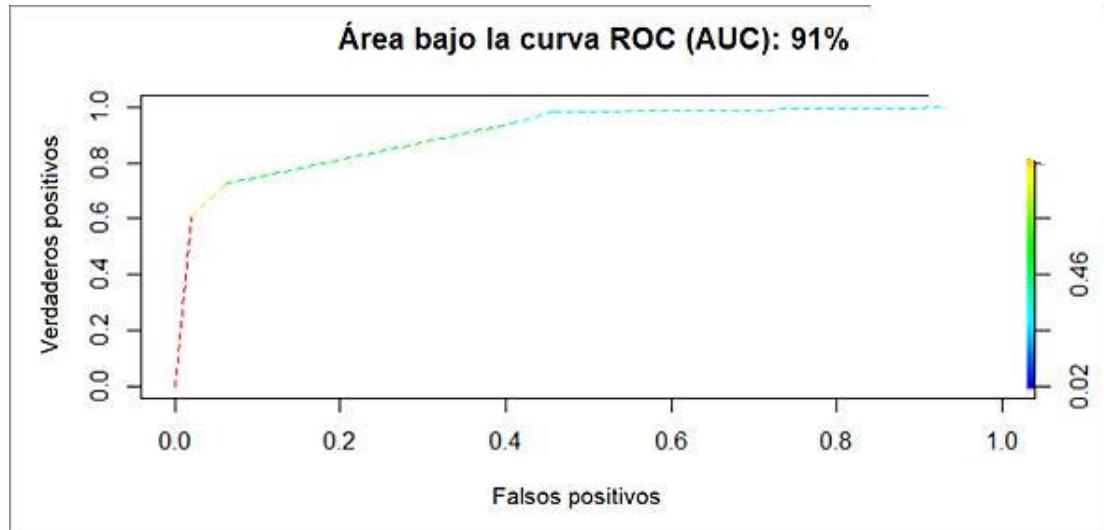
```
## trazar una curva ROC ##

require(ROCR)

y.prob = 1- unlist(treeresponse(m, test), # las probabilidades
                    use.names=F)[seq(1,nrow(test)*2,2)]
y.rocr = prediction(y.prob,test$y)          # estandarización
y.perf = performance(y.rocr, "tpr","fpr") # T y F positive rates

y.auc.perf = performance(y.rocr, measure ="auc", # auc
                         x.measure = "cutoff")
auc <- round(unlist(y.auc.perf@y.values),2)*100 #

plot(y.perf,colorize=T, lty =2,
      main=paste("Área bajo la curva ROC (AUC):",auc,"%"),
      xlab ="Falsos positivos",
      ylab ="Verdaderos positivos")
```



Una curva ROC de aspecto clásico

Recuerde: trazando las curvas ROC de modelos diferentes o de configuraciones diferentes, el mejor de ellos, en el sentido de esta curva ROC, se corresponde con la curva que englobe mejor a los otros (a menudo se corresponde también con la mayor AUC).

2. Una red neuronal (primeros pasos hacia el deeplearning)

A lo largo de este libro, hemos citado en varias ocasiones las redes neuronales. En la parte teórica, nos hemos preparado, de hecho, para abordar estas técnicas, pero sin detenernos demasiado en ello para no alterar el ritmo de la exposición. Como las redes neuronales vuelven en la actualidad con fuerza a través del *deeplearning*, aquí le proponemos realizar unas primeras pruebas antes de atacar al «monstruo» deeplearning en sus investigaciones futuras.

Para comprender el deeplearning, en primer lugar deberá practicar con las redes neuronales sobre diversos conjuntos de datos y confirmar sus fortalezas y sus debilidades.

Entonces estará preparado para abordar la teoría de las redes neuronales del deeplearning, pues comprenderá con detalle la naturaleza de los problemas a los que se dirigen estas soluciones: esto permite no dejarse llevar por la «magia» utilizando los algoritmos más potentes sin idea alguna de sus fortalezas y debilidades... lo cual produce grandes decepciones operacionales.

Vamos a entrenar una red neuronal sobre los datos utilizados en el párrafo anterior (con una única diferencia: por motivos técnicos, la variable de respuesta y no se transformará en factor).

Para el ejemplo, vamos a crear una red neuronal con dos capas ocultas (*hidden layer*) de tres nodos cada una.

Utilizamos la función tangente hiperbólica como función de decisión «difusa» S. Es una curva en S, con el mismo tipo de rol que la función **membership** que hemos visto en la sección dedicada al álgebra difusa del capítulo Otros problemas, otras soluciones.

Observe que estas opciones no están justificadas para los datos que se han de tratar: una red con una capa y la función utilizada por defecto habrían funcionado bien en términos de accuracy. Por el contrario, esta configuración es, sin duda, menos sensible al sobreajuste y equilibra más fácilmente el número de falsos positivos y de falsos negativos. Tendrá, no obstante, que jugar con todos estos parámetros.

En el código observará un hiperparámetro, que le permite definir la frontera de valor entre la clase 0 y la clase 1. Es el tipo de valor que hay que ajustar en la fase de entrenamiento o, llegado el caso, de validación, pero jamás durante la verdadera fase de test.

Es importante destacar que el resultado de los cálculos de predicción no es una propuesta de valores del objetivo, sino un vector normalizado de probabilidad-proporción sobre el que va a decidir la clase que hay que escoger (de ahí los pequeños cálculos de escalado y el umbral de decisión «hiperparámetro»).

```
.... not run: poner el código del párrafo anterior antes
set.seed(1)
y <- 10*df[,1] +                      # una función extraña
    7* df[,2] * rnorm(nrow(df)) +
    3*sin(df[,3]) +
    2*sin(df[,4])*runif(nrow(df))
y <- ifelse(y < 7,0,1)                  # fabricación de las clases
sum(y)/length(y)                         # % clases
df_<- data.frame(df,y)                  # más simple para:
p           <- createDataPartition(y=df_$y,
                                  p= 50/100,
                                  list=FALSE)
training   <- df_[p,]
test       <- df_[-p,]
```

A continuación vamos a entrenar a nuestra red neuronal.

```
library(neuralnet)
m.training <- apply(as.matrix(training),c(1,2),as.numeric)

# ejemplo de función membership          ##
```

```

nn <- neuralnet(y~ X1_Norm + X2_Exp + X3_Pois_1 + X4_unif ,
                 m.training,
                 hidden=c(3,3),
                 act.fct="tanh")
nn

```

1 repetition was calculated.

```

Error Reached Threshold Steps
1 4.972394821    0.009542404099 18010

```

Con tan pocos datos, el tiempo de cálculo es casi instantáneo (tenemos, sin embargo, 18 010 steps). Esta es la fuerza de las neuronas de la red: consumen individualmente muy poca energía.

```

p <- compute(nn,m.test[,1:4])
b <- p$net.result
a <- m.training[, "y"]          # inútil aquí pero para recordar
c <-(max(a)-min(a))*b+min(a) # qué resultados están en proporción
                                # para renormalizar

hyperparam <- 0.5             # JUGAR sobre este valor
y.pred <- ifelse(c < hyperparam, 0,1)
y.test <- test$y
confusionMatrix(y.test,y.pred) # test

```

Confusion Matrix and Statistics

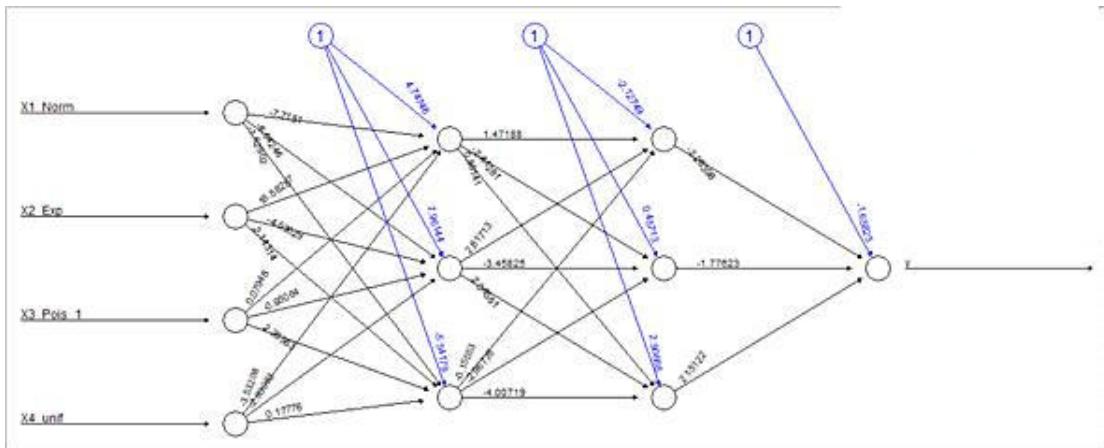
		Reference
		0 1
Prediction	0	86 11
	1	16 37

Accuracy : 0.82

Llegados a este punto, los resultados son un poco peores en términos de accuracy que con el árbol «básico» del capítulo anterior, aunque el equilibrio entre los falsos negativos y los falsos positivos es mejor (ajustar el modelo no era nuestro propósito, y no tenía otra pretensión que mostrarle una sintaxis eficaz para atacar el uso de las redes neuronales).

Veamos a qué se parece nuestra red.

```
plot(nn)
```



Red neuronal 2 capas ocultas

Vemos los pesos sobre los arcos de la red, pero también los nodos suplementarios anotados con «1», y he aquí la explicación:

Basta con referirse a la parte teórica del capítulo Técnicas y algoritmos imprescindibles, donde recordábamos la ecuación de separación (respecto a un hiperplano) que era $y_i \cdot (w^T x_i + b) > 0$. Aquí, el valor de b en esta última ecuación está, de hecho, normalizado a 1 a nivel de la entrada (*input*) de cada neurona de la red.

El paquete que utilizaremos aquí se configura automáticamente para la *backpropagation* mediante el uso de un algoritmo «robusto» (aquí *resiliente*). Pero puede escoger otros tipos de backpropagation; por ejemplo, el algoritmo **grprop** destacado por los autores (en 2005) y que parece más eficaz.

Sin entrar en los detalles de la backpropagation, sepá que tiende a minimizar los errores sobre los pesos utilizando un método de gradiente (similar al que hemos visto más arriba en este capítulo):

$$w^{k+1} = w^k - \eta \nabla E(w^k)$$

El vector de pesos depende del vector de pesos de la iteración anterior a través de un desplazamiento a lo largo de un gradiente. La función $E(w^k)$ sobre la que se aplica ∇ es la SSE (*Sum of Square Errors*, ver el capítulo Dominar los fundamentos) del conjunto de pesos de la iteración anterior.

En las versiones no adaptativas, η que denominamos *learning rate*, es una constante situada entre 0 y 1 por motivos de convergencia o de estabilidad del algoritmo. Sin embargo, ocurre que estos límites reducen la eficacia del algoritmo. Algunos investigadores han creado versiones donde η es un valor que varía durante las iteraciones (en función de las derivadas parciales de $E(w^k)$). Orienta el sentido del gradiente componente a componente.

Encontrará esta técnica en las redes neuronales utilizadas en el marco del deeplearning.

El aprendizaje no supervisado de las redes neuronales es una de las componentes típicas de una implementación deeplearning.

Es una variedad muy interesante de aprendizaje no supervisado, pues no está basado en el clústering, sino en una forma de autoaprendizaje: de capa en capa, se implementa un ciclo donde una capa aprende a dejarse supervisar por otra, se denomina *unsupervised pretraining*.

Una de las principales características comunes de estos sistemas es la implementación de la noción de recursividad (o, al menos, de la noción de reentrada, pues la codificación real no siempre es recursiva). La idea de la recursividad reside en la llamada de una función por sí misma; la reentrada se limita al hecho de reinyectar los datos producidos por un algoritmo previamente en este mismo algoritmo. En ambos casos, la cuestión de la convergencia del conjunto es crucial.

Poder asegurar que el aprendizaje progresá, saber cuándo detenerse, o si existe un punto de equilibrio... constituyen los diversos aspectos de esta misma cuestión.

Para identificar los estados de equilibrio en un sistema físico, posiblemente dinámico, resulta cómodo modelarlo por sus características energéticas (energía potencial vs. energía cinética...). En el marco de las redes neuronales, esta analogía se implementa a menudo a través de las **máquinas de Boltzmann** o de técnicas de **recocido simulado** (*simulated annealing*) que simulan la búsqueda de un estado de equilibrio «térmico» de la red.

La búsqueda del equilibrio energético, incluso simulado, y de la convergencia de un algoritmo determinado puede, en muchos casos, considerarse como resultado de la misma problemática.

Bajo esta óptica, la distribución de Boltzmann (que también se denomina *Gibbs distribution*) posee una propiedad interesante: la relación de dos estados de la máquina solo depende de la «temperatura» y de la diferencia de energía entre ambos estados. Podemos imaginar que la «ruta» entre dos estados importa poco y deducir que el sistema convergerá de forma natural si no se controlan artificialmente las evoluciones intermedias de los estados de la red (es decir, de la máquina de Boltzmann).

Gradient Boosting y Generalized Boosted Regression

1. Los grandes principios

Los principios del boosting y de los métodos de gradiente ya se han presentado en los capítulos anteriores (desde el primer capítulo). A continuación vamos a centrarnos en una implementación práctica, particularmente fácil de manipular para datasets de un tamaño razonable como los que encontrará en los concursos de data scientist: la *Generalized Boosted Regression*.

El objetivo de una regresión se formaliza de la siguiente manera (consulte el capítulo Dominar los fundamentos):

$$\hat{f} = \operatorname{argmin}_f (R_T(f))$$

En el caso «paramétrico», esto equivale a encontrar un vector de parámetros $\hat{\beta}$ estimados de la función tales que:

$$\hat{\beta} = \operatorname{argmin}_{\beta} \sum_{i=1}^n \ell(y_i, f(x_i, \beta))$$

En el caso no paramétrico, nos quedamos con la primera formulación en función del riesgo y se aplica el método del gradiente directamente sobre el riesgo.

Tras cada iteración tenemos en ambos casos:

$$\hat{f}^{k+1} = \hat{f}^k - \eta \nabla R(f)$$

Pero en el caso paramétrico, es posible aplicar el método del gradiente directamente sobre los parámetros, lo cual no nos privamos de hacer.

Todo esto plantea problemas prácticos. En efecto, el modelo tiende a sobreajustarse sobre el conjunto de observaciones (en lugar de crear una función resultante generalizable). Aquellos casos donde existen observaciones diferentes con los mismos valores explicativos que desembocan en una respuesta diferente pueden desestabilizar el descenso por gradiente (icon razón hemos desarrollado otros métodos de optimización como los que hemos visto más arriba!).

Una conclusión práctica (consulte la obra de Friedman) es basar el descenso por gradiente en funciones poco sensibles al valor de un punto (y, por tanto, no apoyarse directamente en el valor de la función f en cada punto).

En el caso de las redes neuronales, la función E en cuestión era la SSE (consulte la sección anterior).

El boosting se aplica sobre esta función.

Como lo explica con sencillez Greg Ridgeway, el creador del paquete `gbm` que vamos a utilizar más adelante, la descomposición funcional ANOVA (*ANalysis Of VAriance*, análisis de la varianza) puede expresarse en función de las distintas variables explicativas:

$$f(x) = \sum_j f_j(x_j) + \sum_{jk} f_{jk}(x_j, x_k) + \sum_{jkl} f_{jkl}(x_j, x_k, x_l) + \dots$$

Esto deja entrever que es posible trabajar a distintos niveles de profundidad de covarianza entre las variables explicativas (en ocasiones llamadas *covariate* en inglés). Si hay poca covarianza (o información mutua compartida

entre dos, tres, ... variables), la modelización resulta mucho más simple (como tras un PCA).

- Observación importante relativa a los árboles de decisión: todo esto implica que, cuando se utiliza una tecnología de árbol que solo toma sus decisiones sobre los valores unitarios de cada variable, no se modela correctamente la covarianza de las n-tuplas de 2 a n variables.

2. Los parámetros y los usos (paquete GBM)

a. Covarianza

Para utilizar el paquete, habrá que seleccionar con esta idea el nivel de covarianza máximo que desea tratar, y este será el parámetro **interaction.depth**.

También habrá que definir una función de riesgo o una función de pérdidas; esto se realizará escogiendo la distribución correspondiente. Es uno de los grandes logros del método: **ipuede escoger cómo se evaluará el rendimiento predictivo del algoritmo!**

b. Loss

El parámetro correspondiente se llama: **distribution** (no olvide que este parámetro define indirectamente su **loss function o su función de riesgo**).

El autor del paquete recomienda seguir la estrategia siguiente: los problemas de clasificación se tratan con las distribuciones **bernoulli** o **adaboost**; las variables continuas, con **gaussian** (norma euclíadiana) o **laplace** (valor absoluto) o por último **quantile regression**; los datos censurados (tiempos de vida), con **coxph**; los resultados de un proceso temporal, con **poisson** (o **gaussian** o **laplace**).

Si le interesan **las redes sociales/grafos y el NLP**, le proponemos estudiar el uso de **pairwise**, que se corresponde con el algoritmo **lambdaMARK**, que es una evolución de **RankNet**, algoritmo que se encuentra entre los favoritos del autor de este libro.

Podría tratar los problemas que consisten en encontrar un orden (una clasificación ascendente o descendente) total (o parcial) entre diversos objetos. Imagine un grafo orientado que describa los rumores que ciertas personas comunican a otras; el que posea el menor *ranking* sería un buen candidato a ser designado como el originador del rumor.

El número de aplicaciones es infinito. Citemos por ejemplo:

- Las herramientas de búsqueda full text y los *web engines*.
- La búsqueda de documentos multimedia.
- El filtrado colaborativo (típicamente el uso de opiniones o hábitos para desarrollar un producto o un servicio).
- El análisis de sentimientos.
- La orientación de la publicidad (por ejemplo, el *web advertising*).

c. Optimización del algoritmo

El **learning rate** se corresponde con el parámetro **shrinkage**. Cuanto más exigente es este parámetro (pequeño), mayor es el número de iteraciones, y más podrá adaptar su modelo (con más o menos riesgo).

La tasa de separación en diversas muestras para gestionar el **bagging** se denomina: **bag.fraction**.

Por último, puede definir el **número de iteraciones** (lo que evita tener que iterar eternamente). Tras una ejecución del algoritmo para un parámetro determinado, obtiene un número de iteraciones óptimo, de modo que no es descabellado configurar el siguiente uso del algoritmo basándose en este número (con un margen de + 20 % por ejemplo). El parámetro se llama: **n.trees**.

3. Puesta en práctica

Para crear el objetivo, vamos a utilizar los mismos datos y la misma función que habíamos utilizado en nuestro ejemplo sobre las redes neuronales. Tendrá que invocar al código correspondiente antes de ejecutar este ejemplo.

La sintaxis es trivial.

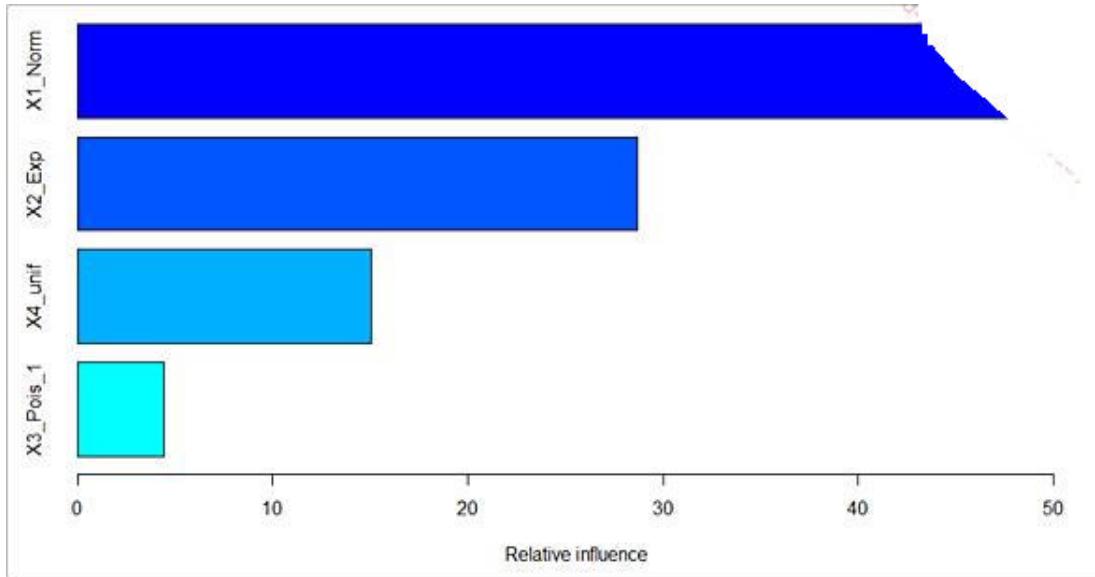
```
library(gbm)
set.seed(1)

m <- gbm(formula = y ~ .,
           distribution = "bernoulli",
           data = training,
           n.trees = 1000,
           interaction.depth = 4,
           shrinkage = 0.01,
           cv.folds=10)

summary(m)
```

	var	rel.inf
X1_Norm	X1_Norm	51.819075458
X2_Exp	X2_Exp	28.721191133
X4_unif	X4_unif	15.021199284
X3_Pois_1	X3_Pois_1	4.438534125

El algoritmo ha percibido correctamente que nuestras variables eran independientes (recuerde su construcción mediante LHS). La importancia relativa de las variables es coherente con la forma en la que hemos construido los datos (para convencerse de ello, estudie la función utilizada para generar el objetivo).

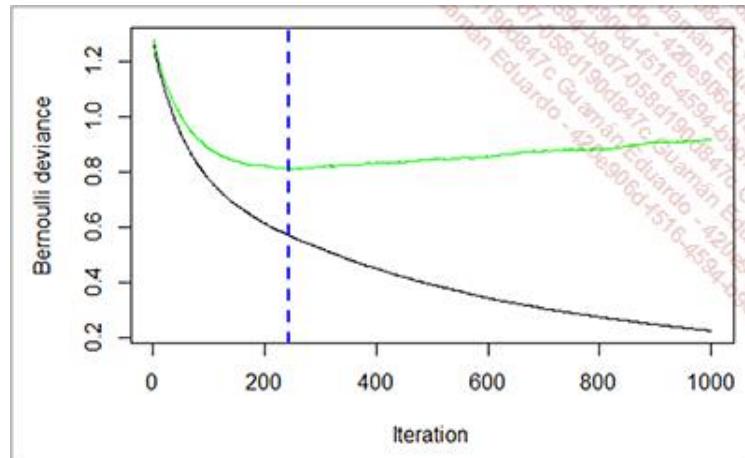


Importancia relativa de las covariates

Veamos el número de iteraciones propuesto por el algoritmo.

```
top = gbm.perf(m,method="cv")      # curva óptima iteración
top                                # número óptimo de iteraciones
```

[1] 243



Número óptimo de iteraciones

En la siguiente llamada al algoritmo con parámetros similares (sobre todo el **shrinkage**), bastará con ajustarse a un valor un poco por debajo de 243 en términos de número de iteraciones (**n.trees**).

Anexos

Acerca de la utilidad de estos anexos

Los anexos aportan diversos complementos al libro y **se integran realmente en él**: recórralos e identifique ahora su contenido, cuya utilidad le resultará evidente cuando aborde un nuevo problema de data science.

Están diseñados para proporcionarle brevemente algunas claves sobre diversos asuntos importantes o prácticos para comprender su campo de investigación respecto a caminos todavía por explorar.

Fórmulas

Dominar la expresión de las *fórmulas* ofrece perspectivas que permiten ahorrar tiempo: en efecto, en lugar de realizar múltiples transformaciones antes de aplicar sus modelos, las transformaciones se realizan al vuelo en el momento de la inyección en el modelo. Esto permite **ganar en memoria, en tiempo y en comodidad**. Inspírese con los siguientes ejemplos.

- Regresión normal con dos variables, con valor en el origen (b): **y~ x1 + x2**
- Regresión sin valor en el origen (b=0): **y~ x1-1**
- Regresión con todas las interacciones posibles entre tres variables: **y~ x1*x2*x3**
- Ídem, pero sin las interacciones de tres variables: **y~ x1*x2*x3 - x1:x2:x3**
- Busca los términos cuadráticos (cuadrados) para x1 y x2; observe el uso de **I()**: **y~x1+I(x1^2)+x2+I(x2^2)**
- Polinomial de grado 3 para x1 y lineal para x2: **y~poly(x1, 3)+x2**
- GLM: **log(y) ~ x1 + x2**
- GAM simple con smooths: **y~s(x1)+s(x2)+s(x3)**
- GAM simple con smooths covariantes: **y~s(x1)+s(x2,x3)**

Trate de utilizar **plot.gam()** para visualizar sus funciones de smoothing y **vis.gam()** para representarlas en 3D (que no muestran una visión en 3D con la variable respuesta, sino una visión en 3D con el predictor).

Preste atención: la función **I()** permite imponer la forma real de la fórmula liberándose de la sintaxis de las fórmulas, por ejemplo para expresar una verdadera potencia de 2 (y no una interacción) en una fórmula, y un fitting sobre **1/x1**:

y~ I(1/x1) + x1 + I(x1^2)

Es posible crear objetos *formula*, lo cual resulta muy útil cuando desea iterar y comparar diversos ejemplos. Para construir un objeto *formula*, intente:

```
x <- paste0("x", 1:3)
formul <- as.formula(paste("y~", paste(x,collapse="+")))
```

Por último, puede resultar útil tratar o no explícitamente los NA:

```
m <- lm(formul, na.action = na.fail)
```

En ocasiones pueden realizarse proezas jugando con las fórmulas y utilizando solamente **lm/glm/gam**, asociado a los paquetes **boot** y a su función de bootstrap: **boot()**.

Si la varianza es una función creciente de la media, los errores no están distribuidos de manera normal o los datos no son reales sino enteros, evalúe utilizar formulaciones como:

```
glm(y~x1, poisson)
glm(y~x1, quasipoisson)
```

Piense también en: **binomial**, **Gamma**, **inverse.gaussian**.

Estrategias según la naturaleza de los datos

Las primeras estrategias descritas aquí se corresponden con el uso de modelos lineales (`lm`, `glm`, `gam`...).

1. Recuentos

Aquí los datos son recuentos (de enteros positivos): número de accidentes, número de días de ola de calor, número de impactos...

El valor 0 es un valor como cualquier otro, de modo que no hace falta cambiarle el sentido mediante una transformación.

Los modelos lineales pueden producir valores negativos, la varianza de la variable de respuesta es una función creciente de la media.

Hemos visto antes que hay que pensar en «Poisson» en este caso.

El trabajo sobre las frecuencias puede resultar fructífero como complemento (ver las proporciones).

Si la **variable de respuesta** es una cuenta, podemos tener grandes problemas de dispersión, y podemos probar **quasipoisson** (o **negativ binomial**).

2. Proporciones

Aquí, se manipulan relaciones: tasa de mortalidad, tasa de respuesta, tasa de curación, reparto de una clase en una población...

Observe que en este caso no se conoce el número de eventos (eso crea parte de la dificultad).

En este caso, pensamos en **binomial** o **quasibinomial** (aunque el problema no es para nada sencillo).

En ocasiones, las transformaciones (como **log**) resultan muy útiles aquí.

3. Variable de respuesta binaria

Aquí tenemos solamente dos clases: hombre/mujer, 0-1...

Hemos visto que la referencia era la distribución de **Bernoulli**.

Pensamos en las opciones **binomial** o **quasibinomial**.

Si utilizamos una función **link**, será a menudo **log-log**.

Se utiliza el test Khi_2 para comparar los resultados de los distintos modelos.

4. Datos que inducen un modelo mixto (mixed effect)

Estos datos se corresponden a menudo (aunque no siempre) con **estudios longitudinales**: seguir a un grupo de individuos, una cohorte, durante un cierto periodo de tiempo.

Típicamente en este caso, la variable de respuesta se mide varias veces en diversos momentos sobre los mismos individuos y cada individuo genera un grupo de datos.

Aquí los datos explicativos son clases con dos naturalezas, fija y aleatoria:

$$y \sim X\beta + Z\theta$$

X es la *design matrix* ligada a los efectos fijos.

Z es la *design matrix* ligada a los efectos aleatorios.

Con una esperanza matemática (la media) que depende únicamente de efectos fijos: $E(y) = X\beta$
y una esperanza matemática de efectos aleatorios que es nula: $E(\theta) = 0$

Para fijar las ideas, formulemos algunos ejemplos:

- ¿Existe un efecto «barrio» (efecto fijo) sobre la delincuencia (respuesta) de los jóvenes (efecto aleatorio)?
- Imagine el estudio de un tratamiento sobre diversos pacientes: el efecto fijo es el tratamiento, el efecto aleatorio es el paciente.
- El tiempo es una covarianza de efecto fijo en el caso de que se observe lo que ocurre antes y después de aplicar un tratamiento.

Piense en calcular coeficientes de correlación, o similar, intraclasses/grupos.

Cree representaciones visuales intraclasses/grupos.

Preste atención: el efecto fijo podría no ser más que una constante, y uno de los efectos aleatorios, la variación de esta constante.

Piense en las siguientes comprobaciones para abordar los efectos aleatorios (o no):

- Cochran-Mantel-Haenszel.
- Woolf.

Piense también en el test de Wald.

Considere **logit** y la regresión logística.

Refiérase, llegado el caso, a su conocimiento acerca de las time series ARMA().

Evalúe el interés de volver a codificar ciertas clases en modelo binario:

clase	codificacion_habitual	codificacion_binaria
<hr/>		
jovenes	1	1 0 0
maduros	2	0 1 0
viejos	3	0 0 1

Trabaje agregando sucesivamente variables en el modelo.

Pruebe los paquetes **lme4** y **nlme**.

Ejemplo de sintaxis:

```
m <- lme(fixed = y~x1, random = ~1|x2/x3 )
```

Estudie el impacto del tiempo e infórmese acerca de la noción de pseudorrereplicación si el análisis visual devuelve curvas temporales similares para varias clases/grupos.

5. Datos espaciales

Los hemos estudiado a lo largo del libro, aunque no hemos estudiado paquetes específicos como **spatstat**, **spdep**, **spatial**, **geoRglm**, **fields**, **grasper**, **pastecs**, **splancs** o **gstat**.

Estos paquetes le proporcionan muchas funciones útiles, incluidas representaciones de mapas, variogramas...

6. Grafos

He aquí algunos paquetes complementarios para estudiar:

Rgraphviz

sna

ergm

dynamicGraph

7. Análisis de supervivencia (survival analysis)

¿En cuánto tiempo, tras cuántos ciclos, una máquina de tal o cual tipo en tal o cual circunstancia se estropeará?

Este tipo de preguntas se plantea en la industria, la medicina, la sociología, las compañías aseguradoras...

La función de supervivencia se escribe: $S(t) = \mathbb{P}(T > t)$

Como podemos adivinar, t representa el tiempo y T una variable aleatoria que caracteriza la ocurrencia de la avería.

Uno de los problemas que se deben resolver aquí reside en el hecho de que la varianza del error no es constante (aumenta con el tiempo).

Podemos utilizar los paquetes **survival** (y modelización **survfit()** o **coxph()**).

Podemos tener que trabajar con **datos censurados**, es decir, tales que no se conoce la fecha de la avería para todos los individuos (de hecho, ciertos individuos todavía no se han averiado, aunque pueden existir otros motivos). Habrá que interesarse por **survreg()** con, por ejemplo, una parametrización simple correspondiente a un valor aleatorio constante (**dist = "exponential"**).

Filtros (sobre imágenes)

Observación previa: tras aplicar o durante la aplicación de los siguientes filtros, a menudo hay que aplicar una normalización de la imagen.

Realizar paso-alto: más contraste (filtro + constante para evitar los valores negativos)

```
0 -1 0    c c c  
-1 5 -1  + c c c  
0 -1 0    c c c
```

Paso-bajo, ejemplo: suavizado, reduce el ruido

```
1 1 1  
1 4 1  
1 1 1
```

Media (es un paso-bajo)

```
1 1 1  
1 1 1  evidentemente en 3 x 3 la normalización es de 1/9  
1 1 1
```

Derivada delta=2 según fila

```
0 0 0  
-1 0 -1  
0 0 0
```

Derivada delta=2 según columna

```
0 -1 0  
0 0 0  
0 -1 0
```

Otros gradientes según fila y columna (no isótropos)

```
0 1 -1  horizontal  
0 1 -1  
0 1 -1
```

```
-1 1 0  horizontal  
-1 1 0  
-1 1 0
```

```
0 0 0 vertical  
1 1 1  
-1 -1 -1
```

```
-1 -1 -1 vertical  
1 1 1  
0 0 0
```

Otros gradientes, contornos: Kirsch según fila y columna (no isótropos)

```
-3 -3 5      podríamos invertirlo y empezar por las 5 en fila  
-3  0 5      como más arriba  
-3 -3 5
```

```
-3 -3 -3      podríamos invertir y empezar por las 5 en columna  
-3  0 -3  
 5  5  5
```

Laplacianos (contornos)

```
0  1  0  
1 -4  1  
0  1  0
```

```
-1 -1 -1  
-1  8 -1      laplaciano de gaussiana 3x3  
-1 -1 -1
```

```
-1  0  0 -1  
 0  1  1  0      se aplica centrado sobre uno de los 4 "1"  
 0  1  1  0  
-1  0  0 -1
```

Filtro mediano: no es un producto de convolución!

sobre un cuadrado (por ejemplo 3 X 3), el filtro reemplaza los valores por el valor mediano de los valores.

Filtro de Sobel(contorno): utiliza dos núcleos laplacianos y a continuación los combina

ejemplo de 2 núcleos combinados (horizontal + vertical):

```
-1 -2 -1  
 0  0  0  
 1  2  1
```

+

```
-1  0   1  
-2  0   2  
-1  0   1
```

Filtro gaussiano (difuminar)

Máscara propia de un gaussiano de dos dimensiones, a menudo sobre un número de píxeles al menos de 5 x 5 caracterizado por la desviación típica; si esta es inferior a 1 se reduce el ruido, si es superior a 1 se difumina.

Sucesión de máscaras. ejemplo de aumento de los detalles

```
imagen + c.(imagen - (imagen * difuso_gaussiano))

es un coeficiente (típicamente 5)
la desviación típica del difuso gaussiano es superior o igual a 1, por ejemplo 3
```

Otros filtros (sofisticados) de detección de contorno:

- Los filtros de Marr = Laplaciano de Gaussiana (LoG) = «sombrero mejicano» (pero mayor que el 3x3 visto antes).
- Filtros diferencia de Gaussianos (Dog).
- Filtros de Canny.
- Snakes.

Distancias

En función de la naturaleza del problema que se ha de tratar, a menudo resulta conveniente utilizar una distancia específica para tratar los datos considerados. No dude en ser creativo a este respecto. Las siguientes líneas le ayudarán a obtener inspiración. Sepa que existen muchas distancias sobre temas muy variados. Cuando se aborda un nuevo problema, a menudo resulta útil realizar una búsqueda sobre las distancias (y las distribuciones, por otro lado) que otros han utilizado para resolver problemas similares.

La base

d no negativa

$$d(x,y) \geq 0$$

$$d(x,y) = d(y,x)$$

$$d(x,x) = 0$$

$$d(x,z) \leq d(x,y) + d(y,z)$$

En ocasiones, estas condiciones no se cumplen al 100 %, se definen entonces «casi» distancias o métricas (pseudo..., semi...) cuyos detalles se salen del marco de este anexo.

Distancias y similaridades

s no negativa es una similaridad:

$$s(x,y) = s(y,x)$$

$$s(x,y) \leq s(x,x)$$

$$s(x,y) = s(x,x) \text{ssi } x = y$$

supongamos s(..) comprendida entre 0 y 1, entonces podemos crear diferentes distancias:

$$d = 1 - s$$

$$d = \frac{1-s}{s}$$

$$d = \sqrt{1-s}$$

$$d = \sqrt{2(1-s)^2}$$

$$d = \arccos(s)$$

$$d = -\ln(s)$$

Jaccard

Distancia (y similaridad) de Jaccard entre dos conjuntos de elementos A y B:

$$s(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

$$\text{dist}(A, B) = 1 - s(A, B)$$

Resulta muy útil para el NLP.

Distancia propia de una norma

Existen muchas normas y, por tanto, muchas distancias naturales asociadas. Siempre tenemos la posibilidad de crear una distancia natural asociada:

$$d(x, y) = \|x - y\|$$

También podemos crear una distancia normalizada asociada:

$$d(x, y) = \frac{\|x - y\|}{\|x\| + \|y\|}$$

Diferencia simétrica semimétrica

Se supone que disponemos de una medida finita μ sobre los conjuntos (como la cardinalidad de estos conjuntos indicada con $|.|$).

$$d(A, B) = \mu(A \Delta B)$$

Δ es la diferencia simétrica de ambos conjuntos.

Esta definición está en el origen de un cierto número de definiciones de distancias.

La distancia de Tanimoto, por ejemplo: $d(A, B) = \frac{|A \cap B|}{|A \cup B|}$

Distancia de Hausdorff

Sean A y B dos subconjuntos de un conjunto Ω dotado de una distancia d:

$$d_p(A, B) = (\sum_{x \in \Omega} |d(x, A) - d(x, B)|^p)^{1/p}$$

Esta distancia puede inspirarnos en el caso del procesamiento de imágenes.

Es una forma de construir una distancia que puede utilizarse en diversos contextos.

La distancia utilizada es habitualmente la distancia de un punto al conjunto:

$$d(x, A) = \inf\{d(x, y) | y \in A\}$$

Distancia entre dos variables aleatorias

$$d(X, Y) = (\mathbb{E} [|X - Y|^p])^{1/p}$$

Distancia entre dos leyes de probabilidad

$$d(P_1, P_2) = (\sum_x (|p_1(x) - p_2(x)|)^p)^{1/p}$$

donde la métrica uniforme de Kolgomorov-Smirnov, que permite verificar la calidad del fitting entre dos distribuciones:

$$d(P_1, P_2) = \sup_x |P_1(X \leq x) - P_2(X \leq x)|$$

Distancias en un grafo

La cadena más corta, la ruta más corta, ponderadas o no son distancias naturales.

Distancia entre dos grafos

La distancia de Fernandes-Valiente entre dos grafos G_1 y G_2 está definida mediante la identificación de su subgrafo máximo $H_M(V_M, E_M)$ y de su subgrafo mínimo $H_m(V_m, E_m)$, y a continuación con el siguiente cálculo:

$$d(G_1, G_2) = (|V_m| + |E_m|) - (|V_M| + |E_M|)$$

Distancia de Mahalanobis

Es la distancia natural propia de la norma (A es una matriz positiva semidefinida y x un vector):

$$\|x\|_A = \sqrt{x^T A x}$$

De modo que podemos crear diferentes distancias de este tipo haciendo variar la matriz en función de su problema.

Trucos y pequeños consejos

1. Acerca de los tests

Los tests no paramétricos son menos potentes, pero también menos exigentes en términos de condiciones sobre las distribuciones de variables.

Antes de aplicar el R de Pearson (test paramétrico) o el R de Spearman (test no paramétrico), compruebe que al menos una de las dos variables sigue una ley normal.

Piense en el test de Kruskal Wallis (no paramétrico) para definir una eventual dependencia entre una variable numérica y una variable cualitativa multi-clase.

Piense en el test de Kolmogorov-Smirnov: dos distribuciones empíricas son similares o no, una distribución empírica se corresponde con una distribución determinada (este último punto resulta particularmente útil).

El test de normalidad de Shapiro-Wilk es, a menudo, una excelente elección.

2. Gestión de las variables

Los coeficientes de una regresión dependen de las unidades, **normalizando** sus datos podrá comparar mejor la influencia de una variable sobre el conjunto. Si tiene **problemas de condicionamiento** de matriz (sensibilidad a pequeñas variaciones, matrices singulares), puede que esto le ayude a salir del paso.

Selección paso a paso de variables: agregando las features paso a paso a su modelo, se hará una mejor idea de su influencia. Si utiliza métodos dedicados a ello, evitará problemas de colinealidad (caso de las regresiones lineales).

3. Análisis y manipulación de resultados

a. Residuos

Existe una forma de dependencia normal (correlación, información mutua...) entre los residuos y las variables de respuesta.

Trace y estudie sus residuos en función de todas las variables: explicativas, respuesta, externas no utilizadas en el modelo.

La forma de nube de residuo, en torno a su recta de regresión, le ofrece información acerca de las transformaciones que puede aplicar a las variables: intente encontrar transformaciones naturales que corrijan esta forma (polinomiales, log, exp, sin, cos...).

En caso de ruptura neta de la nube de residuos, trate de introducir variables externas.

b. Manipulación de los modelos

Utilice la función **summary()** sobre todos los modelos u objetos producidos por los paquetes que utilice. El uso de **unlist()** puede, a su vez, permitirle acceder a elementos de los modelos que parecen inaccesibles.

Tras observar el resumen de un modelo generado, en ocasiones resulta adecuado acceder directamente a su

contenido mediante funciones disponibles en el paquete, como **coef(m)**, **fitted(m)** o **resid(m)** (por lo tanto, investigue si existen estas funciones auxiliares para su paquete/modelo):

En ocasiones es posible **acceder a los datos del modelo generado** con sintaxis del tipo:

```
summary(m)[[1]][2]
```

a las que habrá que aplicar:

```
as.numeric(unlist())
```

En ocasiones se tiene la suerte de poder acceder a los datos del modelo generado por sintaxis como:

```
m$el_dato_deseado
```

Cuando se han creado varios modelos (típicamente del mismo tipo), nada nos impide crear una **lista de estos modelos para recorrerla en bucle** y aplicarle diversas extracciones de información y de procesamiento; esto nos ayudará a ahorrar bastante código.

El tipo de modelo mejor adaptado es, a menudo, un modelo mínimo, y resulta conveniente escoger un modelo menos ajustado.

Paquetes y temas para estudiar

1. Creación de gráficos JavaScript con R

En los dos casos siguientes puede obtener bonitos gráficos con JavaScript:

- Gráficos (o mapas): <http://rcharts.io/>
- Mapas interactivos: <https://rmaps.github.io/>

2. Crear uniones como en SQL

Cuando se dispone de dos tablas, y una contiene una referencia a los registros de otra (clave foránea), en ocasiones hace falta construir una tercera tabla donde se combinan los datos de la segunda, fila a fila, con la primera: es un ejemplo de unión.

El paquete **dplyr** permite hacer esto particularmente bien en excelentes condiciones de rendimiento, por ejemplo:

```
nueva_tabla <- leftjoin(x=tabla1, y=tabla2)
```

Los datos de **tabla2** se proyectan en **tabla1** mediante los valores iguales de las columnas con el mismo nombre. El número de filas resultante en **nueva_tabla** es igual que el de **tabla1**.

3. Reglas de asociación

Un gran clásico, para determinar las reglas de asociación (a menudo se demuestra mediante el ejemplo archiconocido del estudio de los supervivientes del Titanic):

```
library(arules)
mis_reglas <- apriori(datos_titanic)
inspect(mis_reglas)
.....
library(arulesViz)
plot(mis_reglas) # ¡plots interesantes!
plot(mis_reglas, method="grouped")
plot(mis_reglas, method="graph")
```

4. Exportar un modelo

El *Predictive Model Markup Language* (PMML) es un formato de archivo XML cuyo objetivo es compartir modelos entre aplicaciones de machine learning. El número de objetos disponibles es bajo, pero resulta interesante conocer los modelos que son exportables si trata de prototipar en R y a continuación explotarlos en otros entornos. El paquete correspondiente se llama **pmml**.

5. Tensores

Si se siente atraído por las formulaciones matemáticas de alto nivel, si los **arrays** de R le parecen interesantes de manipular, si desea procesar imágenes de manera muy genérica, resolver problemas físicos, o simplemente es curioso y determinado, entonces le conviene conocer los tensores. Los cálculos de índice sobre los tensores pueden preocupar en ocasiones. Los tres paquetes siguientes nos ayudarán con ello:

rTensor

svcm

tensorA

6. SVM para la detección de novedades (novelty detection)

Novelty detection es un problema importante del machine learning. La idea es identificar observaciones del conjunto de test que no se corresponden con el conjunto de entrenamiento y sobre las que pueden existir dudas en cuanto a la capacidad del modelo entrenado para reaccionar correctamente.

Uno de los algoritmos utilizados para esta tarea es SVM en modo «one-classification».

La sintaxis básica podría parecerse a esto:

```
require(e1071)
m <- svm(x,
          type="one-classification",
          kernel="polynomial",
          nu=0.1,
          scale=FALSE)
```

Vocabulario y «tricks of the trade»

Sin duda, no puede ahorrarse el conocimiento de los siguientes temas durante su carrera como data scientist.

1. Complementos sobre las bases del machine learning

OLS: *Ordinary least square (regression)*, es otra manera de llamar a la regresión lineal ordinaria por el método de los mínimos cuadrados que hemos abordado (LM) desde el principio del libro.

RM VS SRM: *Empirical risk minimization* designa la minimización de la función de riesgo sin regularización, *Structural risk minimization* con regularización. L1 y L2 designan los tipos de regularizaciones que hemos visto en el libro.

La función indicatriz, que hemos introducido en el capítulo Técnicas y algoritmos imprescindibles, permite una gran compacidad en la expresión de ciertos problemas por ejemplo, k-NN tiene como objetivo:

$$y = \operatorname{argmin}_c (\sum_{x_i} \mathbb{I}_{\{y_i=c\}})$$

LOOCV: *Leave-one-out cross validation*, es un k-fold CV con k=n (preste atención, observe la variable del modelo atentamente).

PMF VS PDF: *Probability mass function VS Probability density function* (densidad de probabilidad): caso discreto vs. caso continuo para la densidad de una ley de probabilidad.

CDF: *Cumulative distribution function*, es simplemente la función de reparto. Joint CDF (... conjunta) puede identificarse por:

$$F(x,y) = \mathbb{P}(X \leq x, Y \leq y) = \mathbb{P}(X \leq x \cap Y \leq y)$$

LSE: *LogSumExp*, utilizada en el LSE «trick», es una función de smoothing que consiste en hacer el logaritmo de la suma de las exponenciales de los argumentos que se le pasan. Posee una propiedad muy interesante que es:

$$\max\{x_i\} \leq \text{LSE}\{x_i\} \leq \max\{x_i\} + \ln(n)$$

El «truco» correspondiente para evitar overflow (desbordamiento de la memoria/valores extremos) consiste en transformar las expresiones introduciendo el LSE y, a continuación, utilizar otra manera de calcular que evite los overflows:

$$\text{LSE}\{x_i\} = \max\{x_i\} + \ln(\sum_i(e^{x_i - \max\{x_i\}}))$$

En efecto, sustrayendo su máximo a los valores, se disminuye la amplitud del argumento en las exponenciales.

DCM: *Dirichlet Compound Multinomial*, en NLP se utiliza esta función de probabilidad que reemplaza la expresión habitual del modelo multinomial utilizado, para contrarrestar el problema del *burstiness*, es decir, una mala reacción a los cambios bruscos y aleatorios de frecuencias (aquí de frecuencias de palabras en los documentos).

2. Complementos sobre los aspectos bayesianos

Inferencia bayesiana: prácticamente, consiste en inferir probabilidades basándose en teorema de Bayes, como hemos visto en varios lugares de este libro:

- De etapa en etapa, se mejora la inferencia mediante la recogida de datos (observaciones, naturaleza de las distribuciones...) que precisa la probabilidad. En cada etapa, es importante mantener una definición precisa del universo del discurso correspondiente.
- La inferencia bayesiana no es, de hecho, una disciplina simple debido al rigor que implica.
- La noción de probabilidad difiere entre el mundo bayesiano y la manera en la que hemos introducido las probabilidades. En el mundo bayesiano, la probabilidad no es más que un valor numérico asociado al conocimiento o al nivel de confianza sobre el que hemos construido una hipótesis.
- En ocasiones, este conocimiento es resultado de una medida de frecuencia o del número de eventos posibles sobre el número de eventos probables, pero en otras ocasiones, no.
- La probabilidad de un evento puede conocerse a priori, sin conocer otro evento.

Ev: *weight of evidence*, es la medida de la evidencia (del peso de la prueba, o del peso de una aserción fundada):

- Su expresión utiliza un logaritmo (neperiano, natural o decimal, esto solo cambia las unidades). Es posible concebirlo como la inversa de la función sigmoide (una de las funciones en S que asignábamos en las redes neuronales o en lógica difusa) llamada logit.
- $\text{Ev}(p) = \log \frac{p}{1-p} = \log p - \log(1-p) = \text{logit}(p)$
- Es una cantidad aditiva, lo cual resulta práctico. Cuando se mejora el conocimiento, se puede calcular la desviación del Ev obtenido.

MAP VS MLE: Máximo a posteriori vs. Máxima verosimilitud (*Maximum likelihood*). Aquí se encuentra en el corazón de una forma de uso clásica de la estadística bayesiana, que hemos abordado de manera indirecta en ciertos capítulos y de manera más práctica en el capítulo Técnicas y algoritmos imprescindibles, en la descripción de «Naive Bayes». Para abordar técnicas utilizando estos conceptos, debe tener en mente la **importante** distinción siguiente:

- La **verosimilitud** (*likelihood*) de un vector de parámetros θ , conociendo un resultado x , es igual a la probabilidad de la variable observada x conociendo los valores de θ :
$$\mathcal{L}(\theta | x) = \mathbb{P}(x | \theta) \quad [\text{observe la inversión del orden entre } x \text{ y } \theta].$$
- La **probabilidad a posteriori** (*posterior probability*) de un vector de parámetros θ es la probabilidad condicional de este evento tras haber puesto de relieve un cierto hecho x . Hablamos de a posteriori pues se determina (en esta definición) el valor de θ en vista de la observación de los hechos. Esta cantidad se escribe: $\mathbb{P}(\theta | x)$

LR: regresión logística, la hemos abordado indirectamente en los modelos lineales generalizados:

- Aquí hablamos más precisamente de una regresión logística binomial (multinomial si se generaliza).
- Aquí se tiene una variable de respuesta de Bernoulli (0 o 1) y se pueden construir expresiones como $p(x|1), p(x|0), p(0|x), p(1|x)$, como hemos visto algunas líneas antes.
- En el modelo de regresión logística LOGIT, la cantidad $\text{logit}(p(1|x))$ se estima como una combinación lineal de las variables explicativas.
- Observe que esta regresión posee un aspecto «fundamental», pues es también, de hecho, la búsqueda de una regresión de la función de evidencia bayesiana que habíamos evocado más arriba, y que está en el núcleo de su construcción mental e histórica.
- Esto se corresponde con la elección de la función link = logit en GLM o GAM.

3. Vocabulario (en inglés) de los modelos gaussianos

MVN: *MultiVariate Normal*, ley normal multivariable (una «campana» en n dimensiones).

GDA: *Gaussian Discriminante Analysis*, que no habíamos abordado en el libro, permite entre otros crear un clasificador de clases (basado en la distancia de Mahalanobis) del tipo «centroides más cercanas». Esta clasificación permite crear clases que se intersectan, lo cual puede resultar interesante.

LDA VS QDA: *Linea VS Quadratic Discriminant Analysis*: son dos técnicas de clasificación (por lo tanto, de aprendizaje supervisado). El análisis discriminante lineal puede realizarse de forma paramétrica y se supone entonces que estamos frente a una MVN (ver el siguiente párrafo) o no paramétrica si no se cumple esta última condición. La versión cuadrática no utiliza una separación basada en la definición de un hiperplano que conocemos bien: $h(x) = w^T x + b$, sino una separación de clase que hace aparecer una superficie cuya definición contiene términos al cuadrado (**recuérdela**):

$$h(x) = x^T A x + w^T x + b$$

Se obtienen, por ejemplo, clases separadas por hipérbolas, lo que permite en ciertos casos separar regiones no separables linealmente.

Algoritmos para estudiar

Incluir los siguientes algoritmos en su caja de herramientas le resultará un complemento útil a los que ya hemos abordado en este libro:

- MVN & EM: *multivariante mixture of Gaussians* & *Expectation-Maximization*, cuando se encuentran distribuciones multimodales.
- Tabu search, TS: optimización, muy eficaz para los **datos discretos** o las optimizaciones combinatorias.
- Algoritmo genético, GA: estrategia **adaptativa**, optimización.
- Learning Classifier System, LCS: posee capacidades de **refuerzo** del aprendizaje en machine learning.
- Self-organizing Map, SOM: red neuronal **no supervisada** que utiliza una estrategia «competitiva», muy eficaz en feature extraction, no paramétrica.
- ... y la teoría de juegos.

Algunas formulaciones de álgebra lineal

Hemos escogido aquí algunas expresiones matemáticas que encontramos en ciertas demostraciones de nuestro dominio y que permiten traducir rápidamente una expresión en otra más fácil de utilizar. Las condiciones de aplicación y el significado de las expresiones no se describen; para utilizarlas de manera práctica diríjase a las fuentes citadas al comienzo del libro. Las «fórmulas» más simples no se presentan.

Traza

$$\text{Tr}(v^T v) = v v^T$$

$\text{Tr}(X)$ = suma de sus valores propios

$$\partial \text{Tr}(X) = \text{Tr}(\partial X)$$

Determinante

$\det(X)$ = producto de sus valores propios

$$\det(I + u v^T) = 1 + u^T v$$

$$\det(I + \varepsilon X) \cong 1 + \det(X) + \varepsilon \text{Tr}(X) + \frac{1}{2} \varepsilon^2 (\text{Tr}(X)^2 - \text{Tr}(X^2)) \text{ con } \varepsilon \text{ pequeño}$$

$$\frac{\partial \det(X)}{\partial X} = \det(X) (X^{-1})^T$$

Derivadas parciales

$$\frac{\partial \text{Tr}(X^k)}{\partial X} = k (X^{k-1})^T$$

$$\partial \det(X) = \det(X) \text{Tr}(X^{-1} \partial X)$$

$$\frac{\partial \det(X)}{\partial y} = \det(X) \text{Tr}(X^{-1} \frac{\partial X}{\partial y})$$

$$\frac{\partial v^T c}{\partial v} = \frac{\partial c^T v}{\partial v} = c \text{ con } c \text{ vector constante}$$

$$\frac{\partial \text{Tr}(X^T C)}{\partial X} = \frac{\partial \text{Tr}(C^T X)}{\partial X} = C \text{ con } C \text{ matriz constante}$$

$$\frac{\partial X_{kl}}{\partial X_{ij}} = \delta_{ik} \delta_{lj}$$

Esperanza matemática y varianza

$$E[Ax + b] = A\mu + b \text{ con } A \text{ matriz constante, } \mu \text{ vector de medias, } b \text{ vector constante}$$

$$E[AXB + C] = A E[X]B + C \text{ con matrices } A \ B \ C \text{ constantes}$$

$$\text{var}(Cx) = C \text{var}(x) C^T \text{ con } C \text{ matriz constante}$$

$$\text{cov}(Ax, By) = A \text{cov}(x, y) B^T \text{ con matrices } A \ B \text{ constantes}$$

Normas

$$\|cA\| = |c| \|A\|$$

$$\|AB\| \leq \|A\| \|B\|$$

$$\|A + B\| \leq \|A\| + \|B\|$$

Conclusión

Conclusión

Ahora le queda encontrar datos para manipular, nuevos problemas para entrenarse. Puede apoyarse sólidamente en estos nuevos conocimientos para descifrar el sentido de los artículos científicos y de los documentos que acompañan a los paquetes R.

No dude en echar un vistazo a nuestros anexos, puede que encuentre información complementaria útil en el día a día.

La elección de temas para tratar en un libro es siempre un asunto delicado. Hasta la última página, hemos querido ayudarle a ordenar su aprendizaje de manera sólida y que le conduzca inexorablemente a un nivel de conocimiento tal que sea capaz de formar parte de un equipo de data scientists.

Con algo de tenacidad y la lectura atenta del código de otras personas, podrá no solo utilizar los paquetes de sus colegas y crear el «pegamiento» entre las distintas funcionalidades que se proponen, sino también escribir su propio código más fundamental.

Incluso aunque le cueste la teoría y las matemáticas, no sea tímido e intente codificar sus propias ideas en R. No se contente con utilizar el código de los demás («en la estantería», en inglés: *commercial off-the-shelf*). Se progresará rápidamente enfrentándose a problemas reales, siempre que esto dé sentido a las soluciones de los demás.

Esperamos encontrarle en la vida real, o descubrir sus futuros paquetes en CRAN o GitHub. En caso de que este libro le haya resultado útil, no dude en hacer referencia a él, iesto nos animará a seguir trabajando con pasión!

Henri Laude

Con el apoyo operacional de los apasionados en inteligencia artificial y computer sciences que han alimentado este libro a través de numerosas conversaciones o que han tenido la gentileza de aportar múltiples correcciones de orden técnico o científico:

Ghislaine Laude-Dumez

Eva Laude

Audrey Laude

Fréderic Fourré

David Pothier

Lionel Combes

Emmanuel de la Gardette