



INTELIGENCIA ARTIFICIAL PARA PROGRAMADORES CON SPRISA

UNIVERSO
de LETRAS

MARCO ANTONIO ACEVES FERNÁNDEZ

Inteligencia Artificial para Programadores con Prisa

Marco Antonio Aceves Fernández



Inteligencia Artificial para Programadores con Prisa

Marco Antonio Aceves Fernández

Esta obra ha sido publicada por su autor a través del servicio de autopublicación de EDITORIAL PLANETA, S.A.U. para su distribución y puesta a disposición del público bajo la marca editorial Universo de Letras por lo que el autor asume toda la responsabilidad por los contenidos incluidos en la misma.

No se permite la reproducción total o parcial de este libro, ni su incorporación a un sistema informático, ni su transmisión en cualquier forma o por cualquier medio, sea éste electrónico, mecánico, por fotocopia, por grabación u otros métodos, sin el permiso previo y por escrito del autor. La infracción de los derechos mencionados puede ser constitutiva de delito contra la propiedad intelectual (Art. 270 y siguientes del Código Penal).

© Marco Antonio Aceves Fernández, 2021

Diseño de la cubierta: Equipo de diseño de Universo de Letras

www.universodeletras.com

Primera edición: 2021

ISBN: 9788418854613

ISBN eBook: 9788418856723

*Quiero dedicar el presente libro a mi madre
M.^a del Socorro Fernández Valadez (q.e.p.d.),
mi mentora, quien me enseñó que todo logro
requirió esfuerzo y alcanzar
cada sueño, sacrificio.*

Índice general

[**Acerca de este libro 23**](#)

[**Capítulo 1. Introducción 25**](#)

[**Capítulo 2. Conceptos básicos de Python 31**](#)

[2.1. Jupyter Notebooks 32](#)

[2.2. Instalación de Python 32](#)

[2.3. Funciones básicas 35](#)

[2.4. Tipos de datos 37](#)

[2.5. Estructuras de datos 37](#)

[2.6. Ciclos y condicionales 37](#)

[**Capítulo 3. Conceptos básicos de Inteligencia Artificial 41**](#)

[3.1. Inteligencia artificial 41](#)

[3.1.1. Ramas de la inteligencia artificial 43](#)

[3.2. Reglas 45](#)

[3.3. Entropía 46](#)

[3.4. Determinismo 47](#)

[3.5. Heurística y metaheurística 48](#)

[3.6. Espacio de búsqueda 50](#)

[3.7. Instancias 53](#)

[3.8. Atributos 53](#)

[3.9. Observaciones 58](#)

[3.10. Hiperparámetros 60](#)

[3.11. Medidas de distancia 60](#)

[3.11.1. Distancia euclídea 62](#)

[3.11.2. Distancia Manhattan 63](#)

[3.11.3. Distancia Chebyshev 64](#)

[3.12. Tipos de algoritmos de inteligencia artificial 65](#)

[3.12.1. Algoritmos de acuerdo al estilo de aprendizaje 65](#)

[3.12.2. Aprendizaje supervisado 66](#)

[3.12.3. Aprendizaje automático 66](#)

[3.12.4. Aprendizaje semisupervisado 67](#)

[3.12.5. Algoritmos de acuerdo a su similitud 68](#)

[3.12.6. Algoritmos de regresión 68](#)

[3.12.7. Algoritmos basados en árboles de decisión 69](#)

[3.12.8. Algoritmos de agrupamiento \(clustering\) 70](#)

[3.12.9. Algoritmos basados en instancias 72](#)

[3.12.10. Algoritmos basados en reducción de dimensionalidad 73](#)

[3.12.11. Algoritmos de ensamble 74](#)

[3.12.12. Algoritmos basados en redes neuronales artificiales 75](#)

[3.12.13. Algoritmos basados en redes neuronales profundas 75](#)

[**Capítulo 4. Manejo de Datos 77**](#)

- [**4.1. Análisis de distribución de los datos 80**](#)
- [**4.2. Distribución normal 82**](#)
- [**4.3. Identificación de problemas de calidad en los datos 83**](#)
- [**4.4. Manejo de valores faltantes 87**](#)
- [**4.5. Imputación de datos 88**](#)
 - [**4.5.1. Técnicas por información externa o deductiva 89**](#)
 - [**4.5.2. Técnicas deterministas de imputación 89**](#)
 - [**4.5.3. Técnicas estocásticas de imputación 90**](#)
 - [**4.5.4. Métodos de imputación múltiple 91**](#)
- [**4.6. Determinar relaciones entre atributos de los datos 92**](#)
 - [**4.6.1. Gráfica de dispersión \(scatter\) 93**](#)
 - [**4.6.2. Gráfica de barras apilada \(stacked bar\) 94**](#)
 - [**4.6.3. Gráfica de caja \(boxplot\) 95**](#)
 - [**4.6.4. Consideraciones de las relaciones entre variables 97**](#)
- [**4.7. Simplificación de datos escasos 98**](#)
- [**4.8. Cantidades y descripciones de los datos 99**](#)
- [**4.9. Normalizar y escalar los datos 102**](#)
 - [**4.9.1. Codificación uno-a-n 103**](#)
 - [**4.9.2. Normalización de valores ordinales 104**](#)
 - [**4.9.3. Normalización de valores cuantitativos 105**](#)
- [**4.10. Representación de los datos 107**](#)
 - [**4.10.1. Tablas de decisión 107**](#)
 - [**4.10.2. Árboles de decisión 108**](#)
 - [**4.10.3. Reglas de clasificación 109**](#)
 - [**4.10.4. Agrupaciones 110**](#)
- [**4.11. Metadatos 111**](#)

Capítulo 5. Aprendizaje máquina 113

- [**5.1. Aprendizaje inductivo 116**](#)
- [**5.2. Aprendizaje analítico o deductivo 118**](#)
- [**5.3. Aprendizaje evolutivo 118**](#)
- [**5.4. Generalización 118**](#)
- [**5.5. Aprendizaje máquina 119**](#)
 - [**5.5.1. Aprendizaje por refuerzo 120**](#)
 - [**5.5.2. Aprendizaje basado en similitud 121**](#)
 - [**5.5.5. Aprendizaje basado en probabilidad 124**](#)
 - [**5.5.6. Aprendizaje basado en error 126**](#)
 - [**5.5.7. Aprendizaje Supervisado 129**](#)
 - [**5.5.8. Aprendizaje automático 130**](#)
- [**5.6. Metodología propuesta para el desarrollo de algoritmos de IA 131**](#)
 - [**5.6.1. Recolectar 131**](#)
 - [**5.6.2. Preparar 132**](#)
 - [**5.6.3. Analizar 135**](#)
 - [**5.6.4. Entrenar 136**](#)
 - [**5.6.5. Probar 137**](#)

- [5.6.6. Usar 137](#)
- [5.7. Aprendizaje por árboles de decisión 138](#)
- [5.8. Bosques aleatorios 146](#)
- [5.9. Entrenamiento 149](#)
 - [5.9.1. Particionar los datos de entrenamiento 150](#)
- [5.10. Evaluación del modelo 153](#)
 - [5.10.1. Sesgo 159](#)
 - [5.10.2. Sobreaprendizaje 161](#)
 - [5.10.3. Subaprendizaje 161](#)
 - [5.10.4. Regularización 161](#)
 - [5.10.5. Generalización 164](#)
- [5.11. Predecir el rendimiento de un modelo 164](#)
- [5.12. Validación cruzada 167](#)
- [5.13. Validación en línea 169](#)
- [5.14. Validación a una instancia 170](#)
- [5.15. Pruebas de cobertura 170](#)
- [5.16. Métricas de exactitud de un modelo 173](#)
 - [5.16.1. Métricas para datos categóricos 173](#)
 - [5.16.2. Métricas para datos continuos 177](#)
- [5.17. Precisión y exactitud 182](#)
- [5.18. Metaaprendizaje 183](#)

[Capítulo 6. Redes Neuronales 187](#)

- [6.1. Redes neuronales convolucionales 190](#)
- [6.2. Redes neuronales recurrentes 194](#)
- [6.3. Funciones de activación 196](#)
 - [6.3.1. Función de escalón 197](#)
 - [6.3.2. Función sigmoide 198](#)
 - [6.3.3. Función tanh \(tangente hiperbólica\) 199](#)
 - [6.3.4. Función de rectificación lineal \(ReLU\) 200](#)

[Capítulo 7. Factores que afectan el rendimiento en un algoritmo de IA \(factores a considerar\) 203](#)

- [7.1. Preparación de los datos 204](#)
 - [7.1.1. Datos faltantes 204](#)
 - [7.1.2. Valores atípicos 206](#)
 - [7.1.3. Escalar y normalizar 207](#)
 - [7.1.4. Lidiar con el ruido 209](#)
- [7.2. Velocidad de la predicción 211](#)
- [7.3. Capacidad de reentrenamiento 212](#)
- [7.4. Configuración de hiperparámetros 213](#)
- [7.5. Seleccionar el algoritmo adecuado 217](#)
- [7.6. Elección de la función de activación 223](#)
- [7.7. Mejorar la exactitud y el rendimiento del algoritmo 225](#)
 - [7.7.1. Mejorar el rendimiento mediante datos 226](#)
 - [7.7.2. Mejorar el rendimiento mediante algoritmos 228](#)
 - [7.7.3. Mejorar el rendimiento mediante algoritmos de ensamble 230](#)

[Capítulo 8. Prácticas de IA en Python](#) 233

[8.1. Normalización](#) 235

[8.1.1. Min-max](#) 236

[8.1.2. Z-score](#) 237

[8.1.3. Normalización por medias](#) 237

[8.1.4. Normalización por vector unitario](#) 238

[8.1.5. Comparación entre métodos](#) 239

[8.2. Cálculo de distancia euclídea](#) 245

[8.3. Cálculo de distancia Manhattan](#) 253

[8.4. Cálculo de distancia Chebyshev](#) 260

[8.5. Relaciones entre atributos](#) 270

[8.5.1. Gráfica de scatter en 3D](#) 270

[8.5.2. Gráfica de dispersión tipo matriz](#) 276

[8.5.3. Gráfica de barra apilada \(stacked bar\)](#) 280

[8.5.4. Gráfica de cajas \(boxplot\)](#) 284

[8.6. Detección de errores de cardinalidad](#) 288

[8.7. Detección de valores atípicos \(outliers\)](#) 295

[8.8. Análisis de componentes principales \(PCA\)](#) 298

[8.9. Creación de datos sintéticos](#) 314

[8.10. Técnica de imputación múltiple MICE](#) 330

[8.11. Regresión lineal y polinomial](#) 339

[8.12. Gradiente descendente](#) 346

[8.13. Método de validación cruzada \(*K-fold*\)](#) 362

[8.14. Algoritmo K-medias \(K-means\)](#) 368

[8.15. Algoritmo K-vecinos más cercanos \(KNN\)](#) 383

[8.16. Árboles de decisión](#) 395

[8.17. Algoritmo de bosques aleatorios](#) 403

[8.18. Red neuronal simple \(perceptrón\)](#) 411

[8.19. Perceptrón multicapa \(MLP\)](#) 419

[8.20. Red neuronal convolutiva \(CNN\)](#) 427

[8.21. Red neuronal recurrente \(RNN\)](#) 457

[Apéndice. Datos Usados en este Libro](#) 475

[A.1. Lirios](#) 475

[A.2. Enfermedad coronaria](#) 477

[A.3. Gorriones](#) 478

[A.4. Distancia ciudades en américa](#) 481

[A.5. Condiciones climáticas](#) 484

[A.6. Síntomas de meningitis](#) 485

[A.7. Datos de partículas contaminantes PM10](#) 486

[A.8. Otras bases de datos](#) 487

[A.8.1. Datos de pasajeros del Titanic](#) 487

[A.8.2. Datos aleatorios de edades](#) 488

[Glosario de Términos](#) 489

Listado de figuras

- [Figura 1.1. Comparación entre sistemas expertos, sistemas difusos, redes neuronales y algoritmos genéticos 29](#)
- [Figura 2.1. Instalación de Jupyter Notebook 33](#)
- [Figura 2.2. Ejemplo de interface Jupyter Notebook 34](#)
- [Figura 2.3. Ejemplo de interface Jupyter Notebook 34](#)
- [Figura 2.4. Ejemplo de «Hello World» 35](#)
- [Figura 2.5. Tipos de variables 36](#)
- [Figura 2.6. Ciclos y condicionales 38](#)
- [Figura 2.7. Resultado de la ejecución del código de la figura 2.6 39](#)
- [Figura 3.1. Taxonomía general de la inteligencia artificial 45](#)
- [Figura 3.2. Ejemplo de un espacio de búsqueda para algoritmos de optimización 51](#)
- [Figura 3.3. Ejemplo de exploración vs explotación 52](#)
- [Figura 3.4. Diferencia entre instancia, característica, observación y atributo 58](#)
- [Figura 3.5. Representación de la distancia euclídea 62](#)
- [Figura 3.6. Ejemplo de cálculo de la distancia euclídea 63](#)
- [Figura 3.7. Representación de la distancia Manhattan 64](#)
- [Figura 3.8. Representación de la distancia Chebyshev 65](#)
- [Figura 3.9. Ejemplo de algoritmos de aprendizaje supervisado 66](#)
- [Figura 3.10. Ejemplo de algoritmos de aprendizaje automático 67](#)
- [Figura 3.11. Ejemplo de algoritmos de aprendizaje semisupervisado 67](#)
- [Figura 3.12. Ejemplo de algoritmos de regresión 69](#)
- [Figura 3.13. Ejemplo de algoritmos basados en árboles de decisión 70](#)
- [Figura 3.14. Ejemplo de algoritmos de agrupamiento 71](#)
- [Figura 3.15. Ejemplo de algoritmos basados en instancias 72](#)
- [Figura 3.16. Ejemplo de algoritmos basados en reducción de dimensionalidad 73](#)
- [Figura 3.17. Ejemplo de algoritmos de ensamble 74](#)
- [Figura 3.18. Ejemplo de redes neuronales profundas 76](#)
- [Figura 4.1. Histogramas ejemplo de seis diferentes tipos de distribución de los datos. a\) Uniforme, b\) normal \(unimodal\), c\) unimodal sesgado izquierda, d\) unimodal sesgado derecha, e\) multimodal, f\) exponencial 81](#)
- [Figura 4.2. Tres distribuciones gaussianas. \(a\) Distribución con desviación estándar idéntica pero diferentes medias. \(b\) Distribución con media idéntica, pero diferente desviación estándar 83](#)
- [Figura 4.3. Tendencia de los datos cuando es modificada por un dato atípico 86](#)
- [Figura 4.4. Gráfica de dispersión 2D. a\) Dispersión X1 \(longitud total\) vs X2 \(extensión del ala\). b\) X1 vs X5 \(longitud del esternón\) 93](#)
- [Figura 4.5. Gráfica de dispersión tipo matriz 94](#)
- [Figura 4.6. Gráfica de barras apilada 95](#)
- [Figura 4.7. Gráfica de caja. Característica X1 - longitud total 96](#)
- [Figura 4.8. Gráfica de caja. Característica X2 - extensión del ala. 97](#)
- [Figura 4.9. Cuarteto de Anscombe 98](#)
- [Figura 4.10. Ejemplo de una tabla de decisión 108](#)
- [Figura 4.11. Ejemplo de un árbol de decisión 108](#)
- [Figura 4.12. Regla de clasificación 109](#)

- [Figura 4.13. Diferentes maneras de visualizar las agrupaciones](#) 110
[Figura 5.1. Tipos de aprendizaje automático](#) 115
[Figura 5.2. Ejemplo de un algoritmo de aprendizaje por refuerzo](#) 121
[Figura 5.3. Ejemplo de algoritmo de vecinos más cercanos K-NN](#) 123
[Figura 5.4. Ejemplo de regresión lineal](#) 127
[Figura 5.5. Ejemplo de regresión polinomial de grado 3](#) 128
[Figura 5.6. Tipos de aprendizaje por interacción hombre-máquina](#) 130
[Figura 5.7. Nodo raíz para un árbol de decisión con el algoritmo ID3](#) 143
[Figura 5.8. Árbol de decisión con el algoritmo ID3, decisión clima soleado y nublado.](#) 144
[Figura 5.9. Árbol de decisión con el algoritmo ID3 terminado](#) 145
[Figura 5.10. Estructura de un bosque aleatorio](#) 146
[Figura 5.11. Diferentes tipos de entrenamiento](#) 150
[Figura 5.12. Partición recomendada del conjunto de datos en datos de entrenamiento, validación y pruebas](#) 152
[Figura 5.13. Matriz de confusión para un ejemplo de cinco clases](#) 158
[Figura 5.14. Sesgo de una función de activación](#) 160
[Figura 5.15. Comportamiento típico de la pérdida del entrenamiento con respecto al número de iteraciones](#) 162
[Figura 5.16. Diferencia entre pérdida del entrenamiento y pérdida de la validación, representando una mala generalización](#) 162
[Figura 5.17. Pérdida del entrenamiento y pérdida de la validación mostrando el momento de sobreaprendizaje](#) 163
[Figura 5.18. Diferencia de la pérdida del entrenamiento y pérdida de la validación antes y después de la regularización](#) 164
[Figura 5.19. Ejemplo de la distribución de las pruebas para una validación cruzada «10-fold»](#) 169
[Figura 5.20. Ejemplo de un árbol de decisión para pruebas de cobertura](#) 171
[Figura 5.21. Ejemplo de una prueba de cobertura](#) 172
[Figura 5.22. Ejemplo de datos reales vs datos predichos por una red recurrente tipo LSTM](#) 181
[Figura 5.23. Diferencia entre exactitud y precisión](#) 182
[Figura 6.1. Topología de una red neuronal simple](#) 189
[Figura 6.2. Imagen vista como una matriz](#) 192
[Figura 6.3. Imagen vista como un vector](#) 192
[Figura 6.4. Ejemplo de una red recurrente](#) 195
[Figura 6.5. Función de activación de escalón](#) 197
[Figura 6.6. Función de activación sigmoide](#) 198
[Figura 6.7. Función de activación tanh](#) 199
[Figura 6.8. Función de activación ReLU](#) 200
[Figura 7.1. Consideración de análisis para datos con outliers](#) 207
[Figura 7.2. Simulación de ruido en bases de datos basado en k-fold](#) 211
[Figura 7.3. Forma genérica de una red neuronal](#) 218
[Figura 7.4. Forma de un clasificador tipo «Naive Bayes»](#) 219
[Figura 7.5. Forma genérica de un árbol de decisión](#) 220
[Figura 7.6. Forma del algoritmo K-vecinos cercanos](#) 221
[Figura 7.7. Forma del algoritmo máquina de soporte de vectores](#) 222
[Figura 7.8. Tipos de algoritmos de aprendizaje máquina](#) 222
[Figura 8.1. Distribución de datos con la normalización min-max](#) 241
[Figura 8.2. Distribución de datos con la normalización z-score](#) 241
[Figura 8.3. Distribución de datos con la normalización por medias](#) 241
[Figura 8.4. Distribución de datos con la normalización por vector unitario](#) 242

- [Figura 8.5. Comparación de distribución de datos utilizando normalización](#) 242
[Figura 8.6. Conjunto de datos normalizado con min-max](#) 243
[Figura 8.7. Conjunto de datos normalizado con min-max2](#) 243
[Figura 8.8. Conjunto de datos normalizado con z-score](#) 244
[Figura 8.9. Conjunto de datos normalizado con normalización por medias](#) 244
[Figura 8.10. Conjunto de datos normalizado con normalización por vector unitario](#) 244
[Figura 8.11. Conjunto de datos de gorriones](#) 246
[Figura 8.12. Datos e índices de la base de datos de gorriones X1 vs X2](#) 247
[Figura 8.13. Resultado de distancia euclíadiana](#) 249
[Figura 8.14. Resultado del segundo cálculo de la distancia euclíadiana](#) 250
[Figura 8.15. Resultado tres de la distancia euclíadiana](#) 251
[Figura 8.16. Distancia euclíadiana para varios índices simultáneos](#) 252
[Figura 8.17. Cuadrícula para representar la distancia Manhattan](#) 254
[Figura 8.18. Cuadrícula para representar la distancia Manhattan mostrando puntos de inicio y de fin](#) 255
[Figura 8.19. Cuadrícula para representar la distancia Manhattan con puntos de inicio y de fin](#) 256
[Figura 8.20. Cálculo de distancia Manhattan por número de píxeles](#) 257
[Figura 8.21. Cálculo de distancia Manhattan con dimensiones de cuadros](#) 258
[Figura 8.22. Cálculo de la distancia Manhattan con resultado de la distancia](#) 260
[Figura 8.23. Tablero de ajedrez para la distancia Chebyshev](#) 262
[Figura 8.24. Distancia Chebyshev mostrando el tablero de ajedrez y el rey](#) 265
[Figura 8.25. Cálculo de la distancia para una imagen ejemplo](#) 269
[Figura 8.26. Gráfica que muestra la relación entre tres atributos](#) 275
[Figura 8.27. Gráfica que muestra la relación entre los tres atributos seleccionados](#) 276
[Figura 8.28. Matriz de dispersión que muestra las clases en diferentes colores](#) 280
[Figura 8.29. Gráfica de barra apilada](#) 283
[Figura 8.30. Gráfica de cajas de ejercicio de gorriones](#) 288
[Figura 8.31. Listado de base de datos de edad.csv](#) 289
[Figura 8.32. Gráficas de dispersión y de caja para detección de errores de cardinalidad](#) 293
[Figura 8.33. Listado de errores de cardinalidad](#) 294
[Figura 8.34. Listado de errores de cardinalidad para datos no numéricos](#) 294
[Figura 8.35. Gráficas de dispersión y de caja para detección de valores atípicos](#) 298
[Figura 8.36. Representación gráfica de un componente principal](#) 300
[Figura 8.37. Listado de base de datos codificado para análisis de componentes principales](#) 302
[Figura 8.38. Normalización de datos para preprocesamiento de análisis de componentes principales](#) 305
[Figura 8.39. Cálculo de covarianza para PCA](#) 306
[Figura 8.40. Muestra de la base de datos de gorriones para datos sintéticos](#) 318
[Figura 8.41. Distribución estadística de los datos por atributo para el ejemplo de datos sintéticos](#) 320
[Figura 8.42. Ejemplo de un histograma para la distribución de datos para el ejemplo de datos sintéticos](#) 321
[Figura 8.43. Instancias con observaciones únicas para el ejercicio de creación de datos sintéticos](#) 325
[Figura 8.44. Ejemplo de ruleta para la selección de datos aleatorios de las observaciones de datos sintéticos](#) 327
[Figura 8.45. Extracto de los datos creados sintéticamente en la base de datos](#) 329
[Figura 8.46. Comparación entre la distribución de datos original y cuando se agregan datos sintéticos](#) 329
[Figura 8.47. Extracto de la base de datos ambiental del apéndice A.7](#) 332
[Figura 8.48. Síntesis de porcentaje de valores inválidos de la base de datos](#) 333

- [Figura 8.49. Ejemplo de una distribución de cada característica de la base de datos de partículas contaminantes.](#) 333
- [Figura 8.50. Diferencia de distribuciones de datos originales e imputados con MICE](#) 339
- [Figura 8.51. Muestra de datos utilizadas para el ejercicio de regresión](#) 341
- [Figura 8.52. Muestra de datos filtrada por columnas utilizada para regresión](#) 341
- [Figura 8.53. Regresión lineal con datos de contaminantes ambientales](#) 345
- [Figura 8.54. Regresión polinomial de grado 3 con datos de contaminantes ambientales](#) 345
- [Figura 8.55. Regresión polinomial de grado 5 con datos de contaminantes ambientales](#) 346
- [Figura 8.56. Función para representar el ejemplo de gradiente descendente](#) 350
- [Figura 8.57. Función para representar el ejemplo de gradiente descendente mostrando los contornos de la función](#) 352
- [Figura 8.58. Extracto de los resultados por iteración del algoritmo de gradiente descendente](#) 356
- [Figura 8.59. Ejecución del algoritmo de gradiente descendente mostrando su convergencia](#) 359
- [Figura 8.60. Función graficada en 3D mostrando los resultados de gradiente descendente](#) 360
- [Figura 8.61. Resultados finales del algoritmo gradiente descendente](#) 361
- [Figura 8.62. Función Himmelblau para el ejercicio de descendiente al gradiente](#) 362
- [Figura 8.63. División de datos para prueba de validación cruzada cuando k = 5](#) 363
- [Figura 8.64. Resultado de la división de k-fold para k = 4](#) 366
- [Figura 8.65. Extracto de la creación de listas para las particiones con k = 4 para validación cruzada](#) 367
- [Figura 8.66. Extracto de la apertura de archivo iris.csv para K-medias](#) 376
- [Figura 8.67. Listado de valores nulos por atributo \(columnas\)](#) 376
- [Figura 8.68. Estadísticas de la base de datos por atributo \(columnas\)](#) 377
- [Figura 8.69. Separación de datos para K-medias por el método 80-20](#) 377
- [Figura 8.70. Ejecución del algoritmo de K-medias, mostrando las iteraciones 0 y 1](#) 378
- [Figura 8.71. Ejecución del algoritmo de K-medias mostrando las iteraciones 8 y 9](#) 378
- [Figura 8.72. Ejecución del algoritmo de K-medias mostrando el entrenamiento](#) 379
- [Figura 8.73. Ejecución del algoritmo de K-medias mostrando las pruebas](#) 380
- [Figura 8.74. Resultados del algoritmo de K-medias, el porcentaje de predicción correcta](#) 382
- [Figura 8.75. Distribución de datos de enfermedad coronaria para ejercicio de KNN](#) 385
- [Figura 8.76. Base de datos para el ejercicio de KNN, mostrando la normalización de los datos](#) 386
- [Figura 8.77. Resumen de la clase para el ejercicio de KNN](#) 391
- [Figura 8.78. Resultados del entrenamiento para el ejercicio de KNN](#) 392
- [Figura 8.79. Datos de predicción vs datos reales para el algoritmo KNN](#) 394
- [Figura 8.80. Ejemplo de un árbol de decisión mostrando sus atributos y las ganancias por atributo](#) 396
- [Figura 8.81. Datos de la BD de Titanic para el algoritmo de bosques aleatorios](#) 404
- [Figura 8.82. Listado de atributos de la BD de Titanic](#) 405
- [Figura 8.83. Conversión de atributos numéricos mediante la función get_dummies](#) 406
- [Figura 8.84. Información de los atributos de la BD de Titanic con la observación de si existen datos nulos](#) 406
- [Figura 8.85. Información de los atributos de la BD de Titanic sin instancias nulas](#) 407
- [Figura 8.86. Salida del algoritmo de bosque aleatorio](#) 411
- [Figura 8.87. Diagrama de un perceptrón simple de una capa](#) 412
- [Figura 8.88. Diagrama de una compuerta OR de dos entradas](#) 412
- [Figura 8.89. Error por número de épocas de entrenamiento de una compuerta lógica OR por medio de un perceptrón simple](#) 415
- [Figura 8.90. Diagrama de una compuerta XOR de dos entradas](#) 416
- [Figura 8.91. Error por número de épocas de entrenamiento de una compuerta lógica XOR por medio de un perceptrón simple](#) 418
- [Figura 8.92. Ejemplo de un perceptrón multicapa \(MLP\)](#) 420

[Figura 8.93. Función de Rosenbrock](#) 422

[Figura 8.94. Error de aprendizaje para el ejercicio MLP de la función de Rosenbrock](#) 425

[Figura 8.95. Comparación entre el entrenamiento \(izquierda\) y la prueba \(derecha\) de una función de Rosenbrock con un MLP](#) 426

[Figura 8.96. Ejemplo de canales en una imagen](#) 430

[Figura 8.97. Ejemplos de kernel para una red profunda convolutiva](#) 437

[Figura 8.98. Ejemplo de una zancada 2x2 para un kernel 4x4 en una red profunda convolutiva](#) 438

[Figura 8.99. Ejemplo de una zancada 4x4 para un kernel 4x4 en una red profunda convolutiva](#) 438

[Figura 8.100. Ejemplo de un padding de 1x1 y 2x2](#) 439

[Figura 8.101. Ejemplo de MaxPooling para una CNN](#) 440

[Figura 8.102. Ejemplo de dilatación = 1 para una CNN](#) 440

[Figura 8.103. Pérdida de entrenamiento y validación para una CNN](#) 450

[Figura 8.104. Época vs exactitud para una CNN para clasificación de imágenes](#) 451

[Figura 8.105. Pérdida de entrenamiento y validación por número de épocas para una red recurrente](#) 471

[Figura 8.106. Error RMSE para una red recurrente RNN](#) 472

[Figura 8.107. Datos reales vs datos predichos por una red recurrente RNN](#) 473

[Figura A.1. Mapa de las ciudades en el continente americano usada para ejercicios de optimización](#) 483

[Figura A.2. Ejemplo de datos continuos de partículas contaminantes PM10](#) 486

Lista de tablas

[Tabla 3.1. Ejemplo de instancias, atributos y observaciones 59](#)

[Tabla 4.1. Atributos de datos y las operaciones que se pueden realizar con cada uno 101](#)

[Tabla 5.1. Base de datos para caso de estudio de condiciones climáticas 140](#)

[Tabla 5.2. Atributos para observación de clima nublado 143](#)

[Tabla 5.3. Atributos para observación de clima soleado 144](#)

[Tabla 5.4. Atributos para observación de clima lluvioso 145](#)

[Tabla 5.5. Matriz de confusión para cálculo del error 156](#)

[Tabla 5.6. Probabilidad que una variable aleatoria X se encuentre dentro de un rango de confianza 166](#)

[Tabla 5.7. Ejemplo de datos utilizados para calcular las métricas para datos continuos 180](#)

[Tabla 8.1. Error con todos los atributos para comparación con PCA 312](#)

[Tabla 8.2. Error con los componentes principales usando PCA 314](#)

[Tabla A.1. Ejemplo de cada clase BD de lirios 476](#)

[Tabla A.2. Lista de supervivientes para la BD de gorriones 479](#)

[Tabla A.3. Lista de no supervivientes para la BD de gorriones 480](#)

[Tabla A.4. Tabla de distancias de ciudades de América 482](#)

[Tabla A.5. Condiciones climáticas para la base de datos clima 484](#)

[Tabla A.6. Tabla de la BD de meningitis 485](#)

Acerca de este libro

Este libro de fundamentos de inteligencia artificial para programadores con prisa se pensó con el objetivo de presentar de manera sencilla el conocimiento adquirido que se remonta prácticamente desde que comencé la maestría en el año 2001, antes de que la inteligencia artificial fuera tan popular. Posteriormente, cuando terminé mi doctorado en el área de Sistemas Inteligentes, también en la Universidad de Liverpool (Inglaterra) en 2005, noté que hay un crecimiento enorme, pero también un gran desconocimiento de por dónde empezar de manera sencilla.

El área de inteligencia artificial (IA) es muy amplia y está teniendo en los últimos años un crecimiento exponencial. Sin embargo, parte de las razones por las cuales se realiza este libro en específico se debe a que habiendo tantos métodos, algoritmos, problemas y aplicaciones que pueden ser abordados desde muchas perspectivas y, por un lado, se vuelve complejo saber por dónde empezar y qué conocimientos se deben de tener al respecto, mientras que, por otro lado, en ocasiones solo se descarga un código y se ejecuta sin saber a ciencia cierta si lo que se está haciendo o si es lo correcto para ese problema en específico.

Este libro introductorio refleja algunos temas importantes mismos que se abordan con un lenguaje accesible para poder iniciar al lector en esta fascinante área que es la inteligencia artificial, y que considero se deben de explicar. Existen más algoritmos que no se abordan en el presente libro por cuestiones de espacio y número de prácticas. Así mismo, se muestran de manera simple algunas prácticas sencillas utilizando uno de los lenguajes de programación más usados en la actualidad: Python 3.x.

Existen muchos recursos en el internet que pueden ser muy útiles conforme el lector avance en la lectura del presente libro y de la serie de libros que se tienen contemplada. Aunque hay recursos como Mendeley®, DataBrief®, GitHub, entre otros, recomiendo un repositorio de bases de datos de muchas áreas diferentes para poder explorar los temas que se vierten en el presente libro; esta base de datos es la conocida del repositorio de UCI que puede ser consultada y sus bases de datos descargadas de manera gratuita en la siguiente dirección: <http://archive.ics.uci.edu/ml/index.php>

Algunos de los datos de esta base de datos –y de otras– han sido recopiladas por el autor a manera de ejemplo práctico para la implementación y aprendizaje de los temas de esta serie de libros, mismos que son explicados en el apéndice A y que pueden ser encontrados en la siguiente dirección: http://www.amese.net/libro_ia/datos/

Por otro lado, además de plasmar las prácticas y explicarlas en este libro, pueden ser descargadas de la siguiente página: http://www.amese.net/libro_ia/Prog/

Por último, me gustaría aclarar que las prácticas, ejercicios y tips vertidos en este libro son solo una forma de abordar los temas de manera sencilla. Hay muchos otros estilos de programación, otras librerías y otras metodologías para explicar y programar los temas. Al respecto, quiero comentar que traté de realizarlo de manera práctica y clara y las prácticas están realizadas utilizando diferentes estilos de programación con ese mismo propósito, espero haber logrado mi objetivo de atraer tu interés por esta fascinante área. Si tienes un proyecto, negocio o aplicación en mente y no estás seguro por dónde empezar, envíame un correo a: marco.aceves@gmail.com

Espero que disfruten este libro tanto como yo he disfrutado hacerlo.



Marco A. Aceves Fernández

CAPÍTULO 1

Introducción

Comúnmente, cuando doy alguna plática o comento de manera informal que mi área de experiencia son los sistemas inteligentes, la reacción de la gente es muy diversa, aunque casi siempre obtengo la misma imagen mental de mis interlocutores y esta es la de un investigador que está haciendo un «terminator®» en su laboratorio esperando a que algo salga mal y la inteligencia artificial tome control sobre todo y todos en este mundo.

Esta visión apocalíptica de la mayoría de la gente está muy alejada de la realidad —por lo menos de momento— con la inteligencia artificial tomando un rol cada vez más importante en nuestra sociedad, pero quedándose aún en pañales en relación a la visión de ciencia ficción de las películas de Hollywood. Cuando uno se adentra en el cada vez más vasto mundo de la inteligencia artificial, uno de los mayores problemas es saber a qué algoritmo, tipo de procesamiento, variantes, etcétera, adentrarse.

También cuando se me pregunta: **¿cuál es el mejor algoritmo de inteligencia artificial?**, mi respuesta es siempre: **depende**. Hay muchos tipos de algoritmos para muchos tipos diferentes de problemas. No existe una receta de cocina que permita modelar cualquier tipo de problema, bajo cualquier circunstancia con un solo algoritmo. En este sentido, cada vez los algoritmos son más complejos para hacer frente al incremento de la complejidad de los problemas que existen hoy en día.

Estos algoritmos «inteligentes» incluyen algoritmos basados en redes neuronales, cómputo evolutivo, inteligencia colectiva —también llamada inteligencia de enjambre—, lógica difusa, entre otros. Estas técnicas son parte de lo que se conoce como inteligencia artificial (IA). De esta forma, IA es una combinación de diversas disciplinas como las ciencias computacionales, la sociología, biología, matemáticas, etcétera.

A todo esto, ¿cómo se define la inteligencia? En este sentido, la definición no está muy clara y siempre está sujeta a debate. Los diccionarios definen inteligencia como la habilidad de comprender, de entender y de tener la capacidad de pensar y razonar. Otras palabras que suelen definir la inteligencia incluyen: creatividad, conciencia, habilidad, intuición, emoción, entre otros.

Eso nos lleva a la pregunta: ¿las computadoras pueden ser inteligentes? Esta es una pregunta que al día de hoy es objeto de un intenso debate.

Uno de los primeros trabajos en abordar la inteligencia artificial específicamente en relación con la era digital moderna fue escrito en 1950 por el matemático británico Alan Turing, llamado «Maquinaria de cómputo e inteligencia». Es por esto que a Turing se le conoce como el padre de la inteligencia artificial, principalmente, por su contribución en la teoría actual de cómputo. Fue el primero que hizo las preguntas y las pruebas de si las máquinas podían ser creadas para aprender.

El llamado test de Turing mide la capacidad de una máquina supuestamente inteligente contra una persona basada en comportamiento inteligente. Esta prueba, la cual Turing llamó el juego de imitar, consiste en una máquina y una persona, cada una en un cuarto separado y otro humano al que se le llamó el interrogador, quien no puede ver en qué cuarto está la máquina ni en cuál está la persona. La prueba consiste en que el interrogador pueda realizar preguntas cuya respuesta sea a modo de texto, es decir, escrito. Esto se realiza de tal forma que el interrogador no pueda sesgar su juicio de acuerdo con el sonido de una voz o la apariencia de una máquina o una persona. De esta forma, se le pregunta al interrogador si puede distinguir entre la

máquina y la persona de acuerdo con las respuestas que recibió. Si el interrogador no es capaz de distinguir entre la máquina y la persona, entonces se asume que la máquina es inteligente.

Las características principales de las pruebas de Turing son:

- Trata de dar una noción objetiva sobre el término inteligencia de máquinas –comúnmente llamado en la actualidad inteligencia artificial–. El conocer la naturaleza del término inteligencia de un ser por medio de un particular set de preguntas puede proveer el comportamiento estándar para poder responder si una máquina es realmente inteligente.
- Provee un medio para no desviarse de algunos confusos métodos y si son o no apropiados los procesos internos por los cuales se podría considerar una máquina inteligente o si es «consciente» de sus acciones.
- Elimina cualquier sesgo a favor de un organismo viviente al forzar al interrogador a evaluar la inteligencia únicamente por el contenido de las respuestas.

A pesar de que aún se utiliza el test de Turing en muchas aplicaciones y algoritmos diferentes de inteligencia artificial, es vulnerable a muchos problemas inherentes a las pruebas. En primer lugar, los problemas a los que nos enfrentamos y la complejidad de estos ha cambiado radicalmente desde 1950. Así mismo, una máquina que haya sido entrenada para alguna habilidad en específico podría dar respuestas atinadas del test de Turing, pero si se hacen preguntas en un tema diferente, es probable que la máquina no pase el test.

En mi opinión, el test de Turing tiene un problema fundamental en lo que se refiere a tratar de medir la inteligencia de una máquina: el parámetro para poder decir si una máquina es inteligente es el de la inteligencia humana, lo cual me parece un error en muchas aplicaciones.

Es decir, se obliga a una máquina a llenar un molde de inteligencia humana para determinar si es inteligente, generalmente, por medio de tareas o problemas simbólicos. Existen muchos tipos de problemas en los cuales se requiere habilidades motrices, matemáticas, destreza manual, percepción de colores, texturas, formas, entre otros, que, si bien es cierto, no tienen que ver completamente con habilidades cognitivas, sí se puede considerar un componente de la inteligencia humana.

Turing también comentó acerca de la posibilidad de construir un programa inteligente en una computadora digital y las bases para delimitar dicho programa en términos de su capacidad, complejidad computacional, diseño, entre otros, para diseñar dicho programa. Existen principalmente dos objeciones en contra de dicha posibilidad de las máquinas inteligentes que valen la pena comentarlas en este apartado.

La primera ya ha sido refutada por algunos algoritmos de hoy en día, pero se mantuvo vigente por muchos años. Consiste en la premisa de que las máquinas solo pueden hacer exactamente lo que se les programa a hacer, por lo tanto, no puede realizar acciones originales –y, por lo tanto, inteligentes–. Existen, como se comentó, algunas excepciones, como lo son algunos sistemas expertos principalmente para áreas de diagnóstico o las redes profundas, entre muchos otros.

La segunda es llamada el «argumento de la informalidad del comportamiento». Este argumento contempla la imposibilidad de crear un set de reglas que puedan decirle a un individuo exactamente qué hacer bajo cualquier posible circunstancia. En el caso de cada individuo, existe la flexibilidad que permite que reaccione a una gran cantidad de situaciones que se presentan, lo que hace un comportamiento inteligente. Tratar de modelar o predecir cada una de estas posibles reacciones en una máquina es prácticamente imposible, incluso para los avances tecnológicos y la capacidad de cómputo de hoy en día.

A pesar de que la mayoría de los programas muestran una cierta estructura y no demuestran una gran flexibilidad u originalidad, mucho del trabajo de la inteligencia artificial en los últimos quince años ha

demostrado que ciertos modelos, lenguajes, plataformas, etcétera, pueden vencer esta deficiencia. En la figura 1.1 se muestra algunas de las familias de algoritmos que serán abordadas en el presente libro y las habilidades para procesar información.

Comparación de Sistemas Expertos (SE), Sistemas Difusos (SD), Redes Neuronales (RN) y Algoritmos Genéticos (AG)

	SE	SD	RN	AG
Representación del Conocimiento	○	●	□	■
Tolerancia a la Incertidumbre	○	●	●	●
Tolerancia a la Imprecisión	□	●	●	●
Adaptabilidad	□	■	●	●
Habilidad de Aprendizaje	□	□	●	●
Facilidad para Explicarlo	●	●	□	■
Minería de Datos	□	■	●	○
Mantenibilidad	□	○	●	○

Los grados como se evaluaron son: □ malo, ■ relativamente malo, ○ relativamente bueno, y ● bueno

Figura 1.1. Comparación entre sistemas expertos, sistemas difusos, redes neuronales y algoritmos genéticos

CAPÍTULO 2

Conceptos básicos de Python

El lenguaje de programación **Python** está, hoy en día, dentro de los tres lenguajes más populares para la producción de código. Esto es debido a la gran cantidad de desarrollo que existe por parte de la comunidad que contribuye en la implementación de soluciones en una gran variedad de ámbitos, como lo son:

- Frameworks de desarrollo web.
- Manejo y análisis de datos.
- Visualización de datos.
- Matemáticas.
- Inteligencia artificial.

De esta forma, existe una gran gama de aplicaciones que se puede desarrollar por medio de este lenguaje.

Adicionalmente, **Python** se encuentra, junto con R, dentro de los dos lenguajes de programación más usados para análisis de datos. También se ha reportado que **Python** es el lenguaje favorito, en los últimos tiempos, para el desarrollo de aprendizaje máquina (*machine learning*).

Todo esto es debido a las siguientes características:

- Propósito general.
- Multiplataforma.
- Multiparadigma.
- Orientado a objetos.
- Teclado dinámico.
- Legible.

Python es un lenguaje importante al que se debe prestar atención si se quiere entrar en el mundo del análisis de datos y la inteligencia artificial. En esta práctica, se realizará una pequeña introducción a este lenguaje de programación para brindar una rápida adaptación a esta plataforma digital.

2.1. Jupyter Notebooks

Desde 2015 se inició una organización llamada **Project Jupyter**, encargada del desarrollo de software libre de regalías –llamado usualmente «open-source»–, la cual creó un ambiente computacional interactivo basado en web al cual llamaron **Jupyter Notebooks**. Estas libretas **Jupyter Notebooks** son compatibles con varios lenguajes de programación, dentro de los cuales se encuentra Python, para desarrollar código interactivo ejecutado en aplicaciones web.

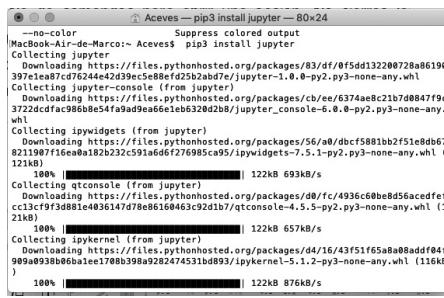
Estas libretas se ejecutan bajo **IPython** (Interactive **Python**) dando versatilidad al momento de programar, combinando texto, código, imágenes y otros medios, haciendo muy dinámico el desarrollo de código. Todo esto puede verse mejor en la práctica. Por lo cual, se procederá a realizar la instalación e introducción al uso de **Python** con **Jupyter Notebooks**.

2.2. Instalación de Python

El proceso de instalación es muy sencillo y aquí resumiremos los pasos a seguir para que tengas lo necesario para seguir los ejercicios que realizaremos.

Ahora, independientemente del sistema operativo que utilices, haz lo siguiente:

- Entra a <https://www.python.org/> y descarga e instala la versión más reciente de Python 3 para tu sistema operativo.
- Abre una consola de comandos.
- Escribe **python -version** en la consola de comandos para asegurarnos de que tenemos una versión 3.XX.
- Escribe **pip3 -version** en la consola de comandos para asegurarnos de que tenemos **pip** instalado.
- Escribe **pip3 --install jupyter notebook** en la consola de comandos, como se muestra en la figura 2.1. Para instalación en MAC, basta con escribir **pip3 install jupyter**.



```
MacBook-Air-de-Marco:~ Aceves$ pip3 install jupyter
Collecting jupyter
  Downloading https://files.pythonhosted.org/packages/df/0f/fdd132300728a86190397e1ea87cd76244e42d39ec5e8efef2d2b5ab7e/jupyter-1.0.0-py3-none-any.whl
Collecting jupyter-console (from jupyter)
  Downloading https://files.pythonhosted.org/packages/cb/e/e/5374ae8c21b7d8847f9c3722ddcfa96bb8e54fa9a9d9a661eb632d2b8/jupyter_console-6.0.8-py3-none-any.whl
Collecting ipywidgets (from jupyter)
  Downloading https://files.pythonhosted.org/packages/56/a/0/dbcf581bb2ff51e8db67821097f16ea0a182b232c591aa6d6f276985c95/ipywidgets-7.5.1-py3-none-any.whl (121kB)
  100% |████████████████████████████████| 122kB 693kB/s
Collecting qtconsole (from jupyter)
  Downloading https://files.pythonhosted.org/packages/d0/fc/4935c68be8d56acedfefcc1cf15d881e40361a7078a8616a463c92d1b7/qtconsole-4.5.3-py2.py3-none-any.whl (121kB)
  100% |████████████████████████████████| 122kB 657kB/s
Collecting ipykernel (from jupyter)
  Downloading https://files.pythonhosted.org/packages/d4/16/43f51f65a8a88addf8af989a093b080a1ee77089398a928274531bd893/ipykernel-5.1.2-py3-none-any.whl (116kB)
  100% |████████████████████████████████| 122kB 876kB/s
```

Figura 2.1. Instalación de Jupyter Notebook

- Escribe **jupyter notebook** en la consola de comandos para abrir una sesión. No cierres la consola mientras estés utilizando tu **Jupyter notebook**.
- Se abrirá una pestaña en tu navegador predeterminado con la *interface de Jupyter*, en la cual podrás navegar entre los directorios de tu computadora, como se muestra en la figura 2.2.



Figura 2.2. Ejemplo de interface Jupyter Notebook

- Para organizar tus ejercicios, crea un directorio para todos tus trabajos.
- Crea una **Jupyter Notebook**, en el directorio antes creado, presionando el botón «**new**» seguido de **Python 3**.
- Ahora ya tienes tu primera **Jupyter Notebook** abierta y completamente funcional, como se muestra en la figura 2.3.

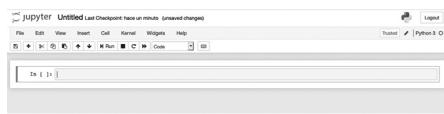


Figura 2.3. Ejemplo de interface Jupyter Notebook

Si no tuviste problemas con los pasos anteriores, ya estás listo para utilizar **Python y Jupyter Notebooks**.

Nota. En caso de que surja algún problema, revisa en internet tu caso específico. Existe mucha documentación para resolverlos, ya que aquí no podemos abordar todos los inconvenientes que pueden presentarse durante la instalación.

Dentro de los problemas más comunes se encuentran:

- No se ha descargado la versión de **Python** correcta.
- No se ha añadido la dirección de instalación de **Python** al «**PATH**» de las variables de entorno.
- **Pip** no se ha instalado.

2.3. Funciones básicas

Como introducción, se desarrollará el tradicional ejercicio de «**Hello World!**» para iniciar con este capítulo. A partir de aquí, podrás ver secciones de texto explicando el procedimiento a seguir, seguido de secciones de código que te permitirán observar cómo se realiza de manera práctica el tema revisado y la salida que genera el código.

Lo primero que haremos será imprimir en pantalla el texto «**Hello World!**». Utiliza la libreta que abriste previamente, en la parte superior puedes cambiar el título de tu **Jupyter Notebook** para mayor organización.

Ahora tienes una celda de código, en ella escribe **print("Hello World!")** y presiona el botón «**Run**» o usa la combinación de teclas **ctrl+Enter** (Windows/Linux) o **Cmd+Enter** (Mac). Esto ejecutará la celda actual, como se muestra en la figura 2.4.

Figura 2.4. Ejemplo de «Hello World»

Nota. Debajo de la barra de secciones de la **Jupyter Notebook** se encuentran botones de funciones rápidas, como lo son guardar, insertar celda, cortar, etc. El último botón de la derecha –ícono de teclado– tiene algunos de los atajos de teclado que te podrían ser útiles para interactuar de forma más rápida con **Jupyter Notebook**, revisalos.

De manera sencilla, creamos nuestro «**Hello World!**» con una línea de código. La función «**print**» te será de utilidad, ya que te permite imprimir en pantalla cualquier texto, número o variable que tengas.

Veamos cómo se comportan diferentes tipos de dato y operadores al desplegarlos con «**print**», como se muestra en la figura 2.5.

```
In [3]: entero=5
decimal=2.5
texto="Hello"
print(type(entero))
print(type(decimal))
print(type(texto))
print(type(texto))
print(type(texto))
print(type(texto))
print(type(entero), type(decimal))
print(type(entero), type(decimal))
print(type(texto))
```

Figura 2.5. Tipos de variables

En el código anterior (Figura 2.5), guardamos tres tipos de datos distintos en tres variables **entero**, **decimal** y **texto**. Imprimimos las variables en pantalla y realizamos algunas operaciones para revisar cómo se comportan los distintos tipos de dato. La función «**type**» permite conocer el tipo de dato de una variable. Observa que en **Python** el tipo de texto «**string**» está definido, indistintamente, por comillas simples « ` o comillas dobles « ` ».

Explora con los tipos de datos y operaciones para que te familiarices con ellos. Recuerda usar la función «**print**» para desplegar en pantalla los resultados —si no usas« **print**» solo se muestra la última línea de código— y la función «**type**» para conocer el tipo de dato.

El código se muestra en esta sección y puede ser descargado en: http://www.amese.net/libro_ia/Prog/

2.4. Tipos de datos

Se tienen los mismos tipos de datos que en la mayoría de los lenguajes de programación:

- Enteros - int().
- Flotantes - float().
- Complejos - complex().
- Cadena de caracteres - str().
- Booleano - bool().

2.5. Estructuras de datos

De igual manera, se tienen estructuras de datos para alojar conjuntos de datos de la manera más adecuada para la aplicación que se requiera. Estas estructuras de datos son:

- Listas - list()
- Tuplas - tuple()
- Diccionarios - dict()
- Cadenas de caracteres - str()
- Conjuntos - set()
- Conjuntos congelados - frozensets()

2.6. Ciclos y condicionales

Para realizar iteraciones o tomar decisiones dentro de un programa, se necesitan los ciclos, los cuales evalúan una variable u operación booleana para así iterar o elegir ciertas condiciones:

- while
- for
- if.

En **Python** no tienes que definir el tipo de dato desde un inicio, puedes asignar a una variable cualquier tipo de dato en cualquier momento, aunque es recomendable tener un orden que te permita tener más control sobre tu código, esto lo hace muy versátil y deja que te adaptes rápidamente a su uso.

El código se muestra en esta sección y puede ser descargado en: http://www.amese.net/libro_ia/Prog/

Para terminar, revisemos un pequeño código que nos permite revisar algunos de los tipos de datos, estructuras de datos y ciclos previamente revisados.

```
In [1]: nombres=[]
for x in range(5):
    nombre=input("Ingresa tu nombre: ")
    apellido=input("Ingresa tu apellido: ")
    nombres.append(nombre)
    print("Tu tipo de estructura de datos es:", type(nombres), end='\n\n')
x=0
while x<5:
    print("Nombre", x, ":", nombres[x])
    x+=1
if len(nombres)>0:
    print("La cantidad de nombres en la lista", len(nombres))
```

Figura 2.6. Ciclos y condicionales

El código previo nos permite ver en funcionamiento algunos conceptos revisados previamente, solo asignamos un texto mediante «**input**» que nos permite ingresar entradas de teclado en la forma de «**string**» y vamos sumando el valor previo de la variable más un nuevo valor. Esto lo hacemos de dos maneras para conocer ambos métodos:

- `variable = variable + aumento`
- `variable += aumento`

```

Ingresá tu nombre: Nombre1
Ingresá tu apellido:Apellido1
Ingresá tu nombre: Nombre2
Ingresá tu apellido:Apellido2
Ingresá tu nombre: Nombre3
Ingresá tu apellido:Apellido3

Tipo de estructura de dato: <class 'list'>
Nombre 0 : Nombre1 Apellido1
Nombre 1 : Nombre2 Apellido2
Nombre 2 : Nombre3 Apellido3

Cantidad de nombres en la lista: 3

```

Figura 2.7. Resultado de la ejecución del código de la figura 2.6

Hasta ahora hemos visto de manera simple algunas de las funciones más básicas para trabajar con Python, como lo son:

- Impresión en pantalla.
- Tipos de dato.
- Operadores.
- Entradas de teclado.
- Asignación de variables.
- Incrementos.
- Estructuras de datos.
- Ciclos.

Con esto cerramos esta introducción a **Python** y **Jupyter Notebook**. Para los ejercicios que se realizarán posteriormente, se requerirá un mayor entendimiento de los tipos de datos, operadores, funciones y clases. Pero todo se explicará en cada paso para que puedas seguirlo, en caso de que sea la primera vez que te encuentres con esos conceptos.

CAPÍTULO 3

Conceptos básicos de Inteligencia Artificial

En este capítulo, se abordan conceptos fundamentales para entender de manera más amplia la inteligencia artificial.

Conceptos como reglas, entropía, determinismo, técnicas heurísticas, espacio de soluciones —llamado también espacio de búsqueda—, instancias, atributos, características, observaciones e hiperparámetros, se explican de manera simple.

También, se abordan conceptos como distancias entre instancias de un mismo atributo. En este sentido, es importante mencionar algunas de las más importantes medidas de distancia que servirán de base para algunos algoritmos. Estas medidas son: distancia euclíadiana, Manhattan y Chebyshev. También se incluyen prácticas para ayudar con estos conceptos en el capítulo 8.

Por último, se abordan algunos tipos de algoritmos de inteligencia artificial y las diferencias que existen entre ellos.

3.1. Inteligencia artificial

La inteligencia artificial (IA) es relativamente un nuevo campo del conocimiento, el cual ha adquirido mucho interés por especialistas de muchas áreas. Cómo se define la IA puede variar enormemente. Una definición optimista puede ser: «el área de las ciencias de la computación que estudia cómo las máquinas pueden realizar actividades que usualmente requieren de producir una salida lógica». Si uno se queda con esta definición, se puede argumentar que algo tan simple como una operación aritmética realizada por una máquina —por ejemplo: una calculadora o una computadora— puede ser considerada como inteligente.

Debido a esto, es necesario manejar una versión diferente de lo que es la inteligencia artificial, por ejemplo: «la disciplina de las ciencias computacionales que estudian cómo las máquinas pueden imitar la inteligencia humana».

Sin embargo, algunas personas podrían decir que, bajo esa definición, las aplicaciones que tenemos de IA en este momento no necesariamente aplican bajo este concepto.

En este libro se aborda una versión más optimista de la inteligencia artificial, pues, aunque aún no se tienen aplicaciones en las cuales la IA pueda «pensar» o exhibir verdadera inteligencia o aún no puede realizar algunas funciones, el autor prefiere hacer notar que, en los últimos años, las aplicaciones que la IA ha podido realizar en muchas ocasiones mejor que una persona o con una mayor precisión o rapidez dan a este campo de conocimiento una enorme variedad de oportunidades para poder mejorar su rendimiento.

Como ejemplos en los que la inteligencia humana puede aún ser mejor que la inteligencia artificial se encuentran reconocer y clasificar objetos con solo una pequeña cantidad de muestras, mientras que la IA lo podría hacer, pero requiere una enorme cantidad de objetos como muestras de entrenamiento para poder reconocer una pequeña cantidad de objetos. Así mismo, la plasticidad cerebral de una persona podría reconocer otros objetos o discernir entre objetos muy parecidos y aprender sobre ellos, mientras que la IA actual requiere que se le indique qué objetos son y cuáles son las diferencias entre ellos. También ha avanzado mucho algunos métodos como el aprendizaje por transferencia, en el que aprende ciertos patrones y luego se expone al algoritmo a un problema similar para que pueda ajustarse y modificar su aprendizaje. En este ejemplo, se puede argumentar que, para un niño pequeño, bastaría ver una silueta de un

animal o un dibujo básico para reconocerlo en, digamos, un zoológico. Un algoritmo de inteligencia artificial, usualmente, ocuparía cientos o miles de imágenes para reconocer el mismo animal, por lo que, para esta y muchas aplicaciones más, no podríamos hablar de verdadera inteligencia.

Algunas de las razones por las cuales la IA está siendo tan popular para automatizar algunas cosas es porque el mundo actual exhibe las siguientes características:

- Vivimos en un mundo donde tenemos una enorme cantidad de datos y se requiere manejarlos de manera eficiente. Además, el cerebro humano no puede hacer un seguimiento de la gran cantidad de información que se sigue generando.
- Esa enorme cantidad de datos que se genera es de múltiples fuentes y está desorganizada además de ser caótica.
- Debido a que los datos siguen cambiando, se requiere de una manera automatizada «inteligente» de ser actualizados y organizados.

3.1.1. Ramas de la inteligencia artificial

Existen diferentes maneras de clasificar a las diferentes ramas de la inteligencia artificial, por ejemplo:

- Tipos de supervisión: se puede clasificar en **aprendizaje supervisado, semisupervisado o automático**.
- Por función de inteligencia se puede separar en **inteligencia general** o, si es de propósito específico, se llama **inteligencia estrecha** o específica.
- Por función humana: **visión máquina** –o visión por computadora–, **procesamiento de lenguaje natural, aprendizaje máquina** –que será explicado más adelante– y **reconocimiento de patrones**.
- Por técnicas de búsqueda: estas técnicas se utilizan mucho en IA. Estos programas examinan todas las posibles soluciones y eligen la mejor solución. Este tipo de técnicas se han utilizado mucho en juegos de estrategia, como ajedrez, Go, así como también los problemas de logística, redes, entre otros.
- Basados en lógica: los algoritmos de IA basados en lógica básicamente son un set de declaraciones que expresan hechos y reglas en relación con el dominio de un problema.
- Planeación: esta rama tiene que ver con la planeación óptima de un objetivo. Cuando se conoce el objetivo y el entorno, se puede llegar a la optimización de los recursos de tal forma que se maximice la ganancia con el costo mínimo.
- Heurística: son técnicas usadas para resolver un problema dado, lo cual es práctico y útil para resolver en el menor tiempo. Esto es debido a que se pueden buscar estrategias para resolver un problema sin que se asegure un óptimo. Se aborda las técnicas heurísticas más adelante en el presente libro.
- Programación genética: la programación genética es una manera de hacer que los programas puedan resolver una tarea imitando la genética –y evolución– de las especies. Los programas se codifican a manera de cromosomas –los cuales se componen de genes– y usan un algoritmo que puedan realizar una tarea específica de tal forma que cada generación de individuos codificados en cromosomas ofrezcan mejores individuos, es decir, mejores soluciones.
- Inteligencia basada en enjambre: esta rama de la IA toma de ejemplo cómo se comportan algunas especies de manera colectiva para resolver un problema en específico. Por ejemplo, la ya conocida aplicación del problema del viajero frecuente (TSP) puede ser resuelto mediante el comportamiento de las hormigas –conocido como *ant colony* o ACO– para encontrar la ruta más corta de su nido a una fuente de comida.

En términos más generales, se puede separar las ramas de la inteligencia artificial en IA, ML (aprendizaje máquina o *machine learning*) y DL (aprendizaje profundo o *deep learning*) como se muestra en la figura 3.1

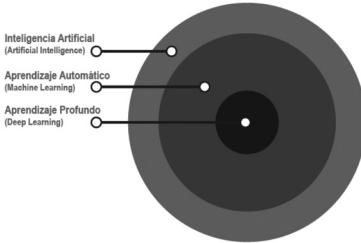


Figura 3.1. Taxonomía general de la inteligencia artificial

3.2. Reglas

Las reglas son una generalización de un conjunto de hechos, es el resultado de un tránsito desde lo particular y específico a un nivel mayor de abstracción y generalización.

Las reglas utilizan un formato IF - THEN para representar el conocimiento, la parte IF de una regla es una condición —también llamada premisa o antecedente— y la parte THEN de la regla —también llamada acción, conclusión o consecuente— permite inferir un conjunto de hechos nuevos si se verifican las condiciones establecidas en la parte IF.

Las reglas pueden expresar un amplio rango de asociaciones:

- Situación-acción: si está lloviendo y va a salir, entonces debe buscar un paraguas.
- Premisa-conclusión: si su temperatura es de 39 grados, entonces usted tiene fiebre.
- Antecedente-consecuente: si x es un gato, entonces x es un animal.

3.3. Entropía

La entropía en inteligencia artificial es una medida de aleatoriedad o desorden existente en un sistema. La medida de entropía o incertidumbre creada primero hace muchos años por Claude Shannon (en 1948) se sigue utilizando en muchos ámbitos como la estadística, comunicaciones y, en los últimos años, en inteligencia artificial.

La idea de entropía es relativamente confusa, principalmente, porque se liga a palabras como desorden, incertidumbre, aleatoriedad, entre otros. En términos simples, se puede explicar la entropía con el siguiente ejemplo:

	<p>Ejemplo</p> <p>Asumimos que tenemos 3 pacientes en la sala de espera de un hospital. Todos los pacientes se sometieron a los respectivos estudios donde, después del diagnóstico, se puede determinar únicamente si se tienen una enfermedad presente o ausente con los siguientes resultados:</p> <p>El paciente A que tiene el 95 % de probabilidades de que tenga la enfermedad.</p> <p>El paciente B tiene el 30 % de probabilidad de que tenga la enfermedad.</p> <p>El paciente C tiene el 50 % de probabilidad que la tenga y el 50 % que no la tenga.</p>
--	---

En el ejemplo anterior, se podría cuestionar quién de los tres pacientes se enfrenta con un mayor grado de incertidumbre. En este caso, no podríamos determinar eso con la menor probabilidad de que tenga la enfermedad, puesto que la respuesta sería entonces el paciente B. Si bien es cierto que el paciente B tiene una menor probabilidad de tener la enfermedad, también es cierto que tiene una mayor probabilidad de que no la tenga —es decir, 70 %—, puesto que su incertidumbre no es tan alta.

De la misma forma, el paciente A, a pesar de que su situación es más complicada, su grado de incertidumbre es más bajo, pues tiene una alta probabilidad de tener la enfermedad en cuestión.

Con relación a la inteligencia artificial y los algoritmos de aprendizaje máquina, tener una entropía alta significaría que los datos tienen una alta variabilidad y una baja predictibilidad, lo que dificultaría encontrar los modelos para poder entrenar el algoritmo. En la sección 5.7 se aborda con una mayor profundidad los cálculos necesarios para conocer la entropía, en este caso, aplicada a árboles de decisión.

En recolección de datos, se pretendería llegar a una entropía tendiente a 0, una entropía alta podría significar una gran cantidad de valores atípicos no válidos, problemas con la cardinalidad de los datos o que estén almacenados en distinto lugar y por diferentes personas que no homogeneizaron los datos de entrada.

3.4. Determinismo

Por determinismo definimos un algoritmo que es 100 % predecible, sin ninguna posibilidad de fallo. Por ejemplo, si en un juego implementamos una IA, si conocemos el mapa, el punto de salida y el de llegada siempre podremos saber cuál será el camino a seguir. Se puede tratar de añadir más variedad de posibilidades, como, por ejemplo, usando algo de lógica difusa en una FSM (*finite state machine*, o en español máquina de estados finitos), pero, aun así, seguiría siendo predecible. Al contrario, un algoritmo no determinista tiene la cualidad de poder hacer cosas para las que no está programada.

También tienen la cualidad de adaptarse al entorno y aprender. Es decir, es en cierto sentido algo inteligente, capaz de pensar por sí mismo. Para seguir con el ejemplo de un sistema para encontrar un camino de un punto a otro, podemos crear un AG (algoritmo genético) capaz de encontrar un camino, pero cada vez el resultado puede ser diferente, unas veces puede ir por unos sitios y otras por otros, sin que de antemano podamos saber con certeza por qué camino nos conducirá.

Si, de manera simple, podemos afirmar que es mucho mejor un algoritmo no determinista, la pregunta sería: ¿por qué seguir usando algoritmos deterministas? La respuesta es que, al ser una sucesión de instrucciones, son fáciles de programar y depurar, mientras que la inteligencia artificial no determinista es mucho más compleja.

3.5. Heurística y metaheurística

La heurística es un término que viene del griego (*εύπισκειν*) que significa «hallar, inventar». Se utiliza como un procedimiento para el que se tiene un alto grado de confianza en que encuentra soluciones de alta calidad con un costo computacional razonable, aunque no se garantice su valor óptimo o su factibilidad. Es decir, encuentra soluciones buenas y se ejecuta razonablemente rápido.

Las soluciones heurísticas, generalmente, son usadas cuando no existe una solución óptima bajo las restricciones dadas en tiempo, espacio o cuando no existe del todo. De esta manera, se puede utilizar la flexibilidad de un algoritmo heurístico o con reglas heurísticas, sin sacrificar buscar todas las posibles soluciones –por cuestiones del tamaño del espacio de búsqueda– en un tiempo determinado. De hecho, muchos algoritmos de inteligencia artificial son basados en reglas heurísticas.

Por otro lado, el término metaheurística es la combinación del prefijo griego «meta» («más allá», aquí con el sentido de «nivel superior») y «heurístico» (de *ευπισκειν*, *heuriskein*, «encontrar»).

En inteligencia artificial se utiliza también el término metaheurística, a veces como sinónimo del término heurística, aunque, en realidad, son conceptos algo diferentes:

- La heurística son reglas empíricas, las cuales dependen del problema a resolver.

- La metaheurística son estrategias genéricas que pueden ser usadas para resolver cualquier problema combinatorio de optimización.

A pesar de ser muy usados los algoritmos (meta) heurísticos, tienen algunos inconvenientes, algunos de ellos son:

- Muchas veces no se tiene la seguridad que se llegó a una solución óptima ni cuál es su solución óptima.
- Por sus características, no explora todas las posibles soluciones, por lo que es posible que no explore todo el espacio de búsqueda.

Sin embargo, este tipo de algoritmos (meta) heurísticos son muy efectivos para muchos problemas combinatorios, pues son relativamente sencillos de implementar y ofrecen soluciones muchas veces óptimas de problemas que mediante otro tipo de algoritmos serían muy complejos, muy tardados o inviables.

Algunos ejemplos de técnicas heurísticas y metaheurísticas son los siguientes:

- Búsqueda tabú.
- Recocido simulado.
- Escalando la colina.
- Algoritmos genéticos.
- Evolución diferencial.
- Optimización por colonia de hormigas (ACO).
- Colonia de abejas artificial (ABC).
- Optimización por enjambre de partículas (PSO).
- Sistema inmune artificial.
- ...Entre otros.

Las metaheurísticas, generalmente, se aplican a problemas que no tienen un algoritmo o heurística específica que dé una solución satisfactoria o bien cuando no es posible implementar ese método óptimo.

La diferencia entre ambas radica en que las heurísticas a menudo dependen del problema, es decir, define una heurística para un problema determinado. Las metaheurísticas son técnicas independientes de los problemas que se pueden aplicar a una amplia gama de problemas como, por ejemplo, TSP o del viajero frecuente con técnicas de colonia de hormigas (*ant colony*).

3.6. Espacio de búsqueda

El concepto de espacio de búsqueda es muy utilizado en inteligencia artificial. En términos generales, el espacio de búsqueda se refiere al dominio en el que la función existe o el algoritmo busca. En términos más simples, podemos definir el espacio de búsqueda como el espacio con una estructura finita donde existen todas las posibles soluciones del sistema a tratar.

Este concepto, en algunas ocasiones, es algo confuso, por lo que el siguiente ejemplo simple puede ser de ayuda:

Ejemplo
<p>Para exemplificar el concepto de espacio de búsqueda, imaginemos que estamos jugando el popular juego de «Adivina Quién»(r). El juego consiste en que cada uno de los jugadores tiene una especie de tablero con personajes cada uno con características diferentes de color de piel, género, color de cabello y otras características como gafas, corbata, etcétera.</p>
<p>Cada jugador elige un personaje para ser adivinado por el contrincante. Por turno se harán preguntas sobre el personaje que el contrincante eligió, como: ¿Tu personaje tiene gafas? De acuerdo a la respuesta, el jugador que preguntó bajará los personajes que no correspondan con la descripción hasta que uno de los jugadores</p>



En el ejemplo anterior del conocido juego, el espacio de búsqueda correspondería a todos los personajes, puesto que no puede ganar un jugador cuyo personaje no exista en las tarjetas. De la misma manera, la solución sería un elemento de mi conjunto de elementos de mi espacio de búsqueda que en ese juego se haya elegido. En algoritmos de inteligencia artificial para optimización, el espacio de búsqueda puede verse como en la figura 3.2.

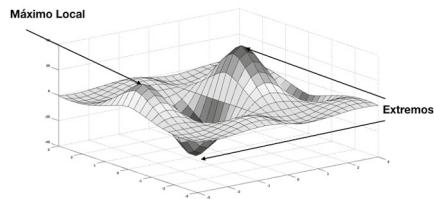


Figura 3.2. Ejemplo de un espacio de búsqueda para algoritmos de optimización

Muchos de los algoritmos de optimización están basados en búsquedas metaheurísticas, por lo que el espacio de búsqueda mostrado en la figura 3.2 puede contener una gran cantidad de soluciones, por lo que es importante encontrar el óptimo o el extremo en caso de estar buscando un máximo o un mínimo global. Supongamos que en este ejemplo se está buscando una solución óptima que pueda representarse como el máximo global. Si se busca de manera metaheurística, se puede encontrar una solución que represente un máximo local, pero no sería la mejor solución y no sería viable buscar todas y cada una de las soluciones del espacio de búsqueda —si fuera así, no sería un algoritmo metaheurístico—. Para ello, se introduce el concepto de exploración y explotación.

La exploración hace referencia a la capacidad de la técnica metaheurística para promover una rápida exploración del espacio de soluciones. Esto previene estancarse en una búsqueda local y en soluciones locales que no son óptimas.

La explotación emplea la previa exploración para crear más individuos alrededor de los puntos con mayor incidencia a ser el extremo global. La figura 3.3 muestra la diferencia entre exploración y explotación en un ejemplo de algoritmos genéticos.

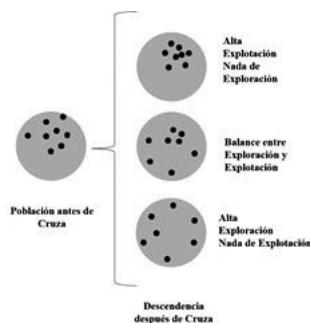


Figura 3.3. ejemplo de exploración vs explotación

Como se observa en la figura 3.3, es necesario un equilibrio entre ambas estrategias para poder alcanzar el extremo global en el espacio de soluciones, de otra forma, se puede terminar en convergencias prematuras o en que nunca converja el algoritmo.

3.7. Instancias

Antes que se aborden los temas de cómo los algoritmos de aprendizaje máquina operan, se explicarán las diferentes formas que las entradas y los datos pueden tener. En cualquier sistema, el entender qué son las entradas y las salidas es de crucial importancia. En el caso de las entradas, pueden tomar la forma de instancias, atributos y características. Cada instancia se caracteriza por los valores de los atributos que miden diferentes aspectos por cada medición. Si lo vemos en el sentido de una base de datos, las instancias serían los registros de esta, es decir, las filas. De esta forma, cada set de datos se representa como una matriz de atributos contra instancias (x, y).

Expresar los datos de entrada como un set de instancias independientes es la manera más común de representación en ejemplos de minería de datos y algoritmos de inteligencia artificial. Sin embargo, formular los problemas de inteligencia artificial únicamente de acuerdo con sets de instancias independientes es restrictivo.

3.8. Atributos

Cada instancia individual que provea una entrada —o salida— de un algoritmo de aprendizaje máquina es caracterizada por un set definido de atributos. Si lo vemos como una base de datos, los atributos serían el equivalente a las columnas —campos—, mientras que las instancias serían las filas, también llamados registros.

El valor de un atributo para una particular instancia es la medición de la cantidad a la cual el atributo se refiere. Existen diferentes distinciones de los tipos de atributos como numéricos, nominales, categóricos, ordinales, de intervalo, de proporción, etcétera, mismos que se abordarán en el capítulo 4, «Datos».

	Ejemplo
	<p>Para exemplificar las relaciones que se tienen entre objetos de diferentes instancias entre atributos tomemos el ejemplo de un árbol genealógico. En dicho ejemplo, se requiere modelar el concepto de hermana. Si tomamos nuestro propio árbol genealógico, en cada una de las ramas del árbol, sin importar cuántas generaciones hayan pasado, el concepto de «hermana» puede existir y está ligado a si tiene hermanos o no, y podría tener diferente nombre para cada instancia «hermana», por lo que depende de la relación hermano-hermana en cada una de las ramas.</p>

Los algoritmos de aprendizaje máquina usan también otro tipo de información de los atributos además de sus tipos. Por ejemplo, los atributos tienen consideraciones dimensionales que pueden ser usadas para restringir la búsqueda de expresiones o comparaciones de si su dimensionalidad es correcta. También, algunos atributos guardan lo que se conoce como metadatos, que se puede explicar de manera muy simple como datos de los datos. Por ejemplo, algunos datos pueden guardar un orden de temporalidad, que se requiere para poder implementar algoritmos de aprendizaje máquina como obtener el dato de la siguiente hora, el siguiente día, día previo, etcétera.

Otros de los principales factores a tomar en cuenta con respecto a los atributos es la selección de los atributos, discretización, escalamiento, transformación, filtrado, entre otros, mismos que se abordarán en el capítulo de datos.

En cuestión de la selección de los atributos, existen ocasiones en las cuales se tienen demasiados atributos y es inviable trabajar con todos. Esto es principalmente porque muchos son claramente redundantes o irrelevantes. Debido a eso, los datos tienen que ser preprocesados para seleccionar solo algunos de los atributos para usar en el entrenamiento. La selección de los atributos para implementar los algoritmos de aprendizaje máquina es muy importante, ya que algoritmos como los árboles de decisión, aprendizaje basado en similitud, agrupamiento (*clustering*), algoritmos basados en reglas, aprendizaje supervisado –principalmente regresión–, entre otros, son muy susceptibles a deteriorar su rendimiento si los atributos no son seleccionados correctamente.

Debido a esto, es muy común que muchos algoritmos de aprendizaje comiencen con una etapa de selección de atributos de tal forma que se eliminen los que no son completamente relevantes para el algoritmo. Una de las mejores maneras de seleccionar los atributos relevantes se basa en un entendimiento de los datos y del alcance del algoritmo. Sin embargo, existen métodos automáticos para poder eliminar los atributos que, para el algoritmo de aprendizaje máquina, son irrelevantes o redundantes. Por ejemplo, los algoritmos de reducción de dimensionalidad pueden ayudar a representar correctamente el problema, enfocándose principalmente en los atributos más representativos.

En general, existen dos estrategias diferentes para poder seleccionar los mejores atributos que representarán el set de datos para entrenar el algoritmo. La primera es realizar un análisis independiente basado en las características generales de los datos. La segunda es evaluar subconjuntos de atributos con el algoritmo de aprendizaje pretendido y evaluar su comportamiento y su precisión con respecto a los atributos elegidos.

Así mismo, algunos algoritmos de clasificación, modelado, predicción, entre otros, tienen que lidiar solamente con atributos nominales, por lo que no pueden manejar los que se encuentran en escalas numéricas. Para usar este tipo de atributos, usualmente, se realiza lo que se conoce como «discretización», es decir, cambiar los atributos numéricos a un rango pequeño de ellos.



Ejemplo

Para exemplificar la discretización de los atributos asumamos que tenemos que clasificar el atributo de temperatura corporal en una base de datos donde se requiere saber si se tiene fiebre.

En este caso, el atributo puede tener valores como: 37.1, 36.4, 36.8, 37.5, 38.5, 38.2, etc. Si no se discretiza, cada valor que no se repita será una observación diferente y se complicará hacer la clasificación.

Por ello, se puede discretizar en este caso en dos. Todos los valores (observaciones) pueden ser cambiados por un atributo nominal y además dicotómico (binario). Es decir, menor de 37,5 grados se cambia por «normal» o «0» y mayor o igual a 37,5 sería: «fiebre», «alta», o «1».

En otros casos, la discretización no ocurre asignando muchos diferentes valores. Otro ejemplo muy común es el de los árboles de decisión, en donde los atributos deben de ser discretizados en un número muy pequeño de valores –usualmente 2, pero depende del árbol– y no se pueden tener tantos valores diferentes en los atributos.

En este sentido, los atributos pueden ser cuantitativos o cualitativos, aunque pueden incluir otros objetos, como una imagen. Su significado es a menudo intercambiable con el término estadístico «variable». El valor de un atributo también se denomina característica. Los atributos valorados numéricamente se clasifican a menudo como categóricos y numéricos, así como discretos o continuos. En cuestión de características, no es un sinónimo de atributo. La característica se refiere a qué información se puede observar de un atributo.

Atributo categórico:

- Categóricos (cualitativos): representan categorías más que números. Operaciones como la suma o la resta no tienen sentido. Se dividen a su vez en:
- Nominales: no tienen orden significativo. Podemos realizar operaciones de igualdad o desigualdad.

- Ordinales: tienen orden definido. Se puede realizar igualdades, desigualdades, mayor y menor que.

Atributo numérico:

- Numéricos (cuantitativos): son atributos que son números y pueden ser tratados como tal. Se dividen a su vez en:
 - Intervalo: no existe un «cero», la división no tiene sentido. Se pueden hacer operaciones de igualdad, desigualdad, de orden, sumas y restas.
 - Tasa: el cero existe, la división tiene sentido. Podemos realizar operaciones que tiene intervalo y, además, multiplicación y división.

Atributo discreto:

- Tiene un número finito o contable de valores. En general, se representa como números enteros.
- Atributos binarios son un caso especial de ellos.

Atributo continuo:

- Tiene un número infinito de valores posibles. Es representado por números reales o de punto flotante. Se pueden obtener tan precisos como sea el instrumento de medición.

Ejemplos:

- Temperatura en grados centígrados: numérico (intervalo), continuo.
- Propiedad de un teléfono celular: categórico (nominal), binario.
- Tamaño de unas papas (pequeñas, medianas, grandes): categórico (ordinal), discreto.

3.9. Observaciones

Las observaciones son toda medición obtenida de nuestro objeto de estudio. Dichas observaciones pueden o no ser significativas y representativas, aunque la medición otorgue un valor erróneo, con ruido o alterado de cualquier manera, este registro es aún considerado como una observación. En la figura 3.4 se muestra la diferencia entre instancia, característica, observación y atributo.

Ejemplo de Instancias, Observaciones y Atributos				
Tiempo (seg)	Sensor 1 (mV)	Sensor 2 (mV)	Sensor 3 (mV)	
1	5	5	5	0
2	10	10	5	0
3	15	15	5	5
4	20	20	0	10

Figura 3.4. Diferencia entre instancia, característica, observación y atributo

Como se comentó anteriormente y se muestra en la figura 3.4, los conceptos de instancia, características, observación y atributo son muy diferentes. En dicho ejemplo, se muestran las instancias que son 4. En base de datos digamos que es el equivalente a registros —de manera horizontal o, dicho de otro modo, las filas—. Con respecto a las características, son la información como los valores que contienen las instancias de un atributo en específico. En este caso, serían los valores numéricos 5, 10, 15 y 20. Con respecto a las observaciones, serían los valores diferentes que tienen un atributo. En el caso del ejemplo, son 3 observaciones en lugar de 4, aunque sean 4 instancias. La razón para esto es porque la característica numérica 5 se repite en dos instancias por lo cual es valor ya fue «observado» en la primera instancia. Por último, toda la columna del sensor 3 es un atributo. Esto funciona para cada una de las columnas.



Ejercicio

De la siguiente tabla que se explica en el Apéndice A.5, mencione las instancias, atributos, características y observaciones.

Día	Clima	Temp.	Humedad	Viento	¿Salir?
1	Soleado	Cálido	Alta	Débil	No
2	Soleado	Cálido	Alta	Fuerte	No
3	Nublado	Cálido	Alta	Débil	Sí
4	Lluvioso	Templado	Alta	Débil	Sí
5	Lluvioso	Frión	Normal	Débil	Sí
6	Lluvioso	Frión	Normal	Fuerte	No
7	Nublado	Frión	Normal	Fuerte	Sí
8	Soleado	Templado	Alta	Débil	No
9	Soleado	Frión	Normal	Débil	Sí
10	Lluvioso	Templado	Normal	Débil	Sí
11	Soleado	Templado	Normal	Fuerte	Sí
12	Nublado	Templado	Alta	Fuerte	Sí
13	Nublado	Cálido	Normal	Débil	Sí
14	Lluvioso	Templado	Alta	Fuerte	No

Tabla 3.1. Ejemplo de instancias, atributos y observaciones

3.10. Hiperparámetros

En aprendizaje máquina, los hiperparámetros son las propiedades que se pueden modificar del proceso de entrenamiento. Por ejemplo, pesos, número de épocas, tasa de aprendizaje, función de activación, entre otros.

Los hiperparámetros son importantes debido a que controlan directamente el comportamiento de los algoritmos de entrenamiento y tienen un gran impacto en el modelo durante la fase de entrenamiento y, por consecuencia, impactan directamente en el rendimiento del algoritmo y sus resultados. Algunos de los ejemplos de hiperparámetros en un algoritmo –en este caso, redes neuronales artificiales– son:

- Sesgo de entrada («bias»).
- Función de activación.
- Inicialización de pesos de la red.
- Tasa de aprendizaje.
- Tamaño del lote (comúnmente llamado «batch»).
- Número de épocas.
- Número de capas.
- Número de neuronas por capa.
- Tipo de capas.
- Función de poda de neuronas (comúnmente llamado «pruning»).
- Tamaño del Kernel.
- Tamaño de zancada (comúnmente llamado «stride»).
- Tamaño de relleno (comúnmente llamado «padding»).

3.11. Medidas de distancia

En muchos algoritmos de IA, el cálculo de la distancia es un elemento crucial para el correcto desempeño del algoritmo. En este sentido, la distancia se utiliza para calcular la similitud entre un conjunto de vectores –usualmente dos–. En inteligencia artificial, se entiende como un vector a un arreglo unidimensional, por lo que la distancia sería la similitud entre dos vectores unidimensionales.

No se debe confundir la dimensionalidad de un problema con la dimensionalidad de un vector. Incluso si el problema tiene veinte parámetros, de cualquier manera, se mantendrá como un vector; los vectores siempre son arreglos unidimensionales, por lo que esa instancia con esos veinte parámetros se guardarán en un vector de longitud veinte.

Nuestro universo está hecho de tres dimensiones perceptibles, aunque algunas veces el «tiempo» es tratado como una cuarta dimensión. Sin embargo, esto no implica que el tiempo sea una cuarta dimensión, por lo menos, no en el sentido en el que las otras tres son. Por ello, debido a que dimensiones superiores a tres son imperceptibles para los humanos, es algo complicado hablar de espacios dimensionales más altos que tres, sin embargo, es algo muy utilizado en algoritmos de inteligencia artificial.

Por ejemplo, si retomamos el ejemplo de clasificación de lirios que se ha usado previamente (apéndice A.1), el set de lirios tiene cinco atributos, los cuales son:

- Longitud del sépalo.
- Ancho del sépalo.
- Longitud del pétalo.
- Ancho del pétalo.
- Especie de lirio.

Los primeros cuatro pueden ser considerados como parte de un vector unidimensional de longitud cuatro, mientras que el quinto atributo es el de decisión, el cual no es numérico, pero puede ser codificado, por lo que daría tres dimensiones más al vector –utilizando la codificación uno-a-n, como se explica en la sección 4.9–.

Para poder conceptualizar el concepto de distancia, imaginemos una hoja de papel donde uno de los vectores de distancia se posiciona en un punto x , y de la hoja (x_1, y_1) , mientras que el otro punto se posiciona en otro punto x , y de la hoja (x_2, y_2) , en este caso). La distancia sería entonces la cercanía entre (x_1, y_1) y (x_2, y_2) . En la siguiente sección, se explican los tres métodos más utilizados para calcular la distancia entre dos vectores: distancia euclídea, distancia Manhattan y distancia Chebyshev.

3.11.1. Distancia euclídea

La distancia euclídea está basada en la distancia bidimensional entre dos vectores. Esto es, la distancia entre dos puntos como si se dibujara una línea con una regla de un punto al otro, como se muestra en la figura 3.5. Esta es una de las distancias que más se usan en algoritmos de aprendizaje máquina.

Especificamente, si se tienen dos puntos (x_1, y_1) y (x_2, y_2) , la distancia euclídea puede ser calculada mediante la ecuación 3.1

$$d = \sqrt{(X_2 - X_1)^2 + (Y_2 - Y_1)^2} \quad (3.1)$$

La representación gráfica de la distancia euclídea, como si se tratara de dos puntos en una hoja de papel, puede observarse en la figura 3.5.

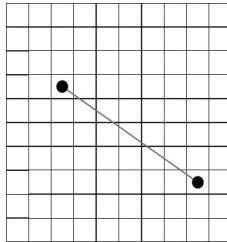


Figura 3.5. Representación de la distancia euclidiana

La figura 3.6 muestra un ejemplo de cálculo de la distancia euclidiana de dos puntos (2,1) y (-2,2).

$$\begin{aligned} \text{dist}((2,1), (-2,2)) &= \sqrt{(2 - (-2))^2 + (1 - 2)^2} \\ &= \sqrt{(2 + 2)^2 + (-1 - 2)^2} \\ &= \sqrt{(4)^2 + (-3)^2} \\ &= \sqrt{16 + 9} \\ &= \sqrt{25} \\ &= 5 \end{aligned}$$

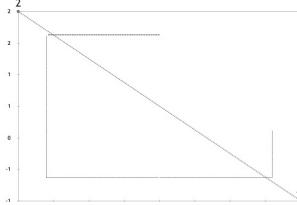


Figura 3.6. Ejemplo de cálculo de la distancia euclidiana

3.11.2. Distancia Manhattan

La distancia Manhattan es también llamada distancia del taxi (o «Taxicab»). Se puede entender mejor la distancia como si alguien estuviera manejando por una ciudad. En este ejemplo, un conductor no podría manejar en una ciudad como si se tratara de llegar de un punto a otro utilizando la distancia euclidiana. Se lo impiden los edificios, parques, forma de las calles, etcétera, por lo que, para calcular este tipo de distancias, Manhattan utiliza un método basado en el desplazamiento hacia un solo eje a un tiempo, como se puede ver en la figura 3.7.

Para calcular la distancia Manhattan entre dos puntos, se suman las distancias absolutas como se muestra en la siguiente ecuación:

$$D = \sum_{i=1}^n |x_i - y_i| \quad (3.2)$$

La distancia Manhattan tiene muchas aplicaciones. Se utiliza muchas veces en tratamiento de imágenes y para determinar tiempo en mapas, por ejemplo, aplicaciones de ejercicio y transporte. La principal diferencia entre la distancia euclidiana y la distancia Manhattan es que, cuando se tratan de distancias largas, la distancia Manhattan es penalizada desproporcionadamente en comparación con el método de distancia euclidiana.

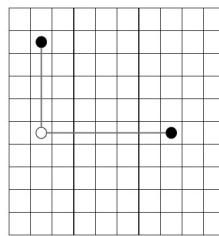


Figura 3.7. Representación de la distancia Manhattan

3.11.3. Distancia Chebyshev

La distancia Chebyshev también es muy utilizada en algoritmos de aprendizaje máquina. La distancia Chebyshev se conoce comúnmente como distancia ajedrez. En el ajedrez, cada pieza se mueve de manera diferente en el tablero. En este sentido, la distancia Chebyshev se puede interpretar como el número de movimientos en los que el rey puede llegar de su posición inicial a cierta posición en el tablero, como se muestra en la figura 3.8.

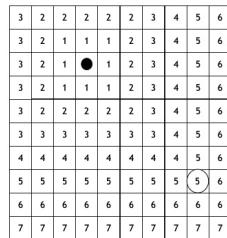


Figura 3.8. Representación de la distancia Chebyshev

3.12. Tipos de algoritmos de inteligencia artificial

Es útil tener en consideración los más populares tipos de algoritmos de inteligencia artificial para poder saber cuáles están disponibles y cuándo se pueden utilizar. Hoy en día, existen tantos algoritmos diferentes que uno se puede sentir abrumado por tantos métodos diferentes para resolver algún problema en particular.

En general, se pueden agrupar de acuerdo al tipo de aprendizaje o la similitud.

3.12.1. Algoritmos de acuerdo al estilo de aprendizaje

Con respecto al estilo de aprendizaje, existen diferentes maneras en las cuales se puede modelar un problema de acuerdo al medio ambiente, la experiencia o la interacción de los datos.

De una manera más amplia, se puede clasificar el aprendizaje en tres grandes rubros: supervisado, semisupervisado y automático.

3.12.2. Aprendizaje supervisado

Los datos de entrada se conocen como datos de entrenamiento. En este tipo de aprendizaje, se conoce el resultado —llamado etiqueta—, como se muestra en la figura 3.9. El modelo se prepara a través de un proceso de entrenamiento para poder realizar predicciones al respecto. El entrenamiento continúa hasta que se logran los resultados de exactitud requeridos. Problemas de clasificación y regresión son ejemplo de este tipo de aprendizaje.

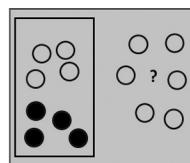


Figura 3.9. Ejemplo de algoritmos de aprendizaje supervisado

3.12.3. Aprendizaje automático

En este tipo de aprendizaje, los datos de entrenamiento no están etiquetados, es decir, no se conoce el resultado.

En este tipo de aprendizaje, el modelo se prepara deduciendo estructuras presentes en los datos de entrenamiento. Esto puede ser en general mediante reglas generales.

También, se puede realizar mediante un proceso matemático que reduce redundancias o puede ser organizado por similitud, como veremos más adelante (práctica 8.14). En este tipo de aprendizaje, se encuentran algoritmos como k-medias («k-means»).

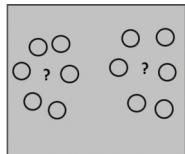


Figura 3.10. Ejemplo de algoritmos de aprendizaje automático

3.12.4. Aprendizaje semisupervisado

Este tipo de aprendizaje en ocasiones no se toma en cuenta, pues muchas veces se asume que o no se tienen etiquetas o se tienen todas las etiquetas. El aprendizaje semisupervisado funciona como una especie de mezcla entre ambos. En este tipo de aprendizaje (figura 3.11), se cuenta con una predicción deseada o un problema definido, pero el modelo tiene primero que aprender los patrones y estructuras para organizar los datos a la vez que realiza las predicciones. En este tipo de problemas, también se encuentran clasificación y regresión.

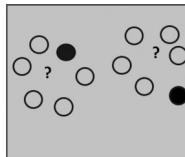


Figura 3.11. Ejemplo de algoritmos de aprendizaje semisupervisado

Este tipo de taxonomía de separación de algoritmos es adecuado, pero más allá de las etiquetas de los datos, se pueden clasificar los algoritmos de diferentes maneras. Por ejemplo, se pueden clasificar de acuerdo a su similitud o funcionalidad.

Es importante conocer la taxonomía de los algoritmos —es decir, manera de organización—, pues ayuda a pensar en las diferentes maneras en la que los datos pueden ser presentados y el proceso de entrenar los modelos y, así, elegir el que mejor pueda dar resultados.

3.12.5. Algoritmos de acuerdo a su similitud

Los algoritmos de inteligencia artificial se agrupan muchas veces de acuerdo a su similitud en términos de su funcionamiento interno. Por ejemplo, por árboles de decisión o redes neuronales. La mayoría de los algoritmos se pueden clasificar de esta manera, aunque existen algunos que pertenezcan a más de un grupo a la vez.

3.12.6. Algoritmos de regresión

Los algoritmos de regresión son métodos estadísticos que tratan de determinar la relación entre una variable dependiente —usualmente denominada Y— y una serie de otras variables conocidas como variables independientes.

Los algoritmos de regresión modelan la relación entre las variables y es iterativamente mejorada utilizando métricas de error en la predicción que se realiza del modelo.

Los algoritmos de regresión más populares son:

- Regresión lineal.
- Regresión logística.
- Regresión por mínimos cuadrados (OLSR, por sus siglas en inglés).
- Regresión polinomial.
- Regresión por escalón.
- Regresión LASSO.
- Regresión multivariable adaptativa (MARS, por sus siglas en inglés).
- Regresión por Kernel.
- ...Entre otras.

La figura 3.12 muestra una representación de la estimación de datos mediante algoritmos de regresión.

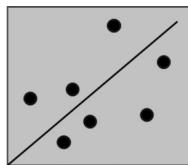


Figura 3.12. Ejemplo de algoritmos de regresión

3.12.7. Algoritmos basados en árboles de decisión

Los algoritmos basados en árboles de decisión son aquellos que construyen un modelo de decisiones basado en los valores actuales de los atributos de la base de datos.



Tip

Los algoritmos basados en árboles de decisión se recomiendan cuando no se tienen muchos atributos ni muchas observaciones por atributo. Cuando esto no es posible, se pueden discretizar el número de observaciones por atributo. Por ejemplo, en lugar de unatributo como humedad donde se tienen valores enteros, se puede discretizar en "normal" y "alta".

Las decisiones se construyen en forma de árboles cuya estructura es gráfica y es fácil de seguir su predicción siguiendo las ramas del árbol. En general, los árboles de decisión son sencillos de realizar a nivel algorítmico, ya que se basan en decisiones anidadas de tipo IF-THEN y por su naturaleza gráfica, son muy utilizadas en aprendizaje máquina.

Los algoritmos basados en árboles de decisión más populares son:

- Árboles de clasificación y regresión (CART, por sus siglas en inglés).
- ID3.
- CHAID.
- C4.5 / C5.0 (diferentes versiones de este algoritmo).
- M5.
- Árboles de decisión condicional.

La figura 3.13 muestra una representación gráfica de los árboles de decisión.

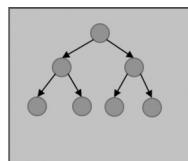


Figura 3.13. Ejemplo de algoritmos basados en árboles de decisión

3.12.8. Algoritmos de agrupamiento (clustering)

Clustering describe tanto la clase del problema a tratar como la clase de los métodos a utilizar.

Los métodos de clustering son típicamente organizados por cómo se modelan como, por ejemplo, por centroides o jerárquicos.

Los algoritmos más populares de agrupamiento son:

- Basados en centroides (familia K-means):
 - K-means.
 - K-means++.
 - K-means esférico.
 - K-means basado en Kernel.
 - K-means difuso (FCM y FCS).
- Agrupamiento jerárquico:
 - Clustering jerárquico.
 - CURE.
 - HC.
 - BIRCH.
- Agrupamiento basado en densidad:
 - DBScan.
 - HDBScan.
- Agrupamiento basado en matrices de Markov:
 - MCL.

La figura 3.14 muestra un ejemplo de los algoritmos de agrupamiento (clustering).

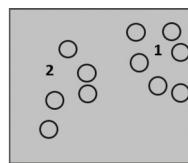


Figura 3.14. Ejemplo de algoritmos de agrupamiento

3.12.9. Algoritmos basados en instancias

Los algoritmos basados en instancias son un tipo de algoritmo en el cual se requiere de instancias entrenadas que se requieren para construir el modelo.

Estos métodos usualmente se basan en la construcción del modelo con instancias de ejemplo y se compara nuevas instancias a la base de datos con la que se construyó el modelo utilizando medidas de similitud –generalmente, calculando distancia entre ambas–. Las medidas de distancia son abordadas en la sección 3.11.

Los algoritmos basados en instancias más populares son:

- K-vecinos cercanos (KNN, por sus siglas en inglés).
- LVQ.
- Mapas autoorganizados (SOM).
- LWL.

La figura 3.15 muestra un ejemplo de los algoritmos basados en instancias.

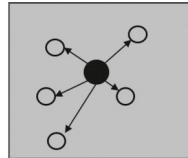


Figura 3.15. Ejemplo de algoritmos basados en instancias

3.12.10. Algoritmos basados en reducción de dimensionalidad

En aprendizaje máquina (ML), la «dimensionalidad» se refiere al número de características –variables de entrada– que tiene una base de datos. Cuando el número de características es mucho más grande que el número de observaciones que se requieren, puede causar problemas para construir un modelo eficiente. Por esta razón, se requiere de algoritmos de reducción de dimensionalidad para poder simplificar las características más relevantes de la base de datos.

Los algoritmos más populares de reducción de dimensiones son:

- Análisis de componentes principales (PCA).
- Regresión de componentes principales (PCR).
- Análisis de discriminantes lineales (LDA).
- Análisis de discriminantes cuadráticos (QDA).
- Análisis de discriminantes flexibles (FDA).

La figura 3.16 muestra un ejemplo de los algoritmos basados en reducción de dimensionalidad.

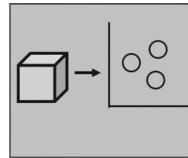


Figura 3.16. Ejemplo de algoritmos basados en reducción de dimensionalidad

3.12.11. Algoritmos de ensamble

Los algoritmos de ensamble son muy usados para mejorar la predicción de modelos de aprendizaje máquina. Consisten en entrenar modelos de manera independiente y combinar las predicciones para poder realizar una predicción global. Este tipo de algoritmos se utilizan en ocasiones para aprendizaje por transferencia, donde se realiza un aprendizaje para algo en concreto y ese mismo entrenamiento se utiliza para que el algoritmo aprenda algo similar.

Estos modelos independientes se conocen como metamodelos y la salida de la combinación de estos metamodelos es la salida de metaaprendizaje.

Los algoritmos más populares de algoritmos de ensamble son:

- Boosting.
- Bagging.
- Stacking (también llamada «stacked generalization»).
- AdaBoost.
- Bosques aleatorios.

La figura 3.17 muestra un ejemplo de los algoritmos de ensamble.

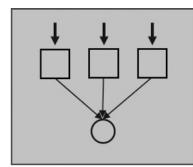


Figura 3.17. Ejemplo de algoritmos de ensamble

3.12.12. Algoritmos basados en redes neuronales artificiales

Este tipo de algoritmos serán cubiertos más a detalle en el capítulo 6. Los algoritmos basados en redes artificiales también son conocidos como redes neuronales artificiales. Este tipo de algoritmos están inspirados en la estructura y funcionalidad de las redes neuronales biológicas.

Actualmente, existen cientos de tipos de redes neuronales artificiales y variaciones que se utilizan en muchas aplicaciones.

Los algoritmos más populares de redes neuronales artificiales son:

- Perceptrón.
- Perceptrón multicapa (MLP, por sus siglas en inglés).
- De retropropagación.
- Red de Hopfield.
- Red de Hebb.
- Red de función de base radial (RBFN, por sus siglas en inglés).

3.12.13. Algoritmos basados en redes neuronales profundas

Los algoritmos basados en redes neuronales profundas se pueden considerar una subcategoría de las redes neuronales artificiales tradicionales. Usualmente, su arquitectura –se conoce como topología– son redes neuronales más complejas, con más neuronas por capa y más capas por cada red, además de una mayor cantidad de hiperparámetros, los cuales regulan el comportamiento y rendimiento de la red. Usualmente, son buenas para la clasificación, modelado y predicción de bases de datos muy grandes como video, audio, datos de tiempo continuo, imágenes, entre otros.

Los algoritmos más populares de redes neuronales profundas son:

- Redes neuronales convolutivas (también se le llaman convolucionales o CNN):
 - VGG.
 - Inception.
 - Xception.
 - ResNet.
- Redes neuronales recurrentes:
 - Redes de memoria de corto/largo plazo (LSTM, por sus siglas en inglés).
 - GRU.
 - Redes Elman.
- Autocodificadores apilados (*stacked auto-encoders*):
 - Autocodificadores SAE.
 - Autocodificadores DAE.
 - Autocodificadores CAE.
- Máquina profunda de Boltzmann (DBM por sus siglas en inglés).
- Redes de creencia profunda (DBN, por sus siglas en inglés).
- Transformadores.
- Redes de celda de memoria.
- Redes basadas en mecanismos de atención.

La figura 3.18 muestra un ejemplo de los algoritmos basados en redes neuronales profundas.

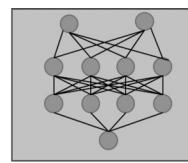


Figura 3.18. Ejemplo de redes neuronales profundas

En este texto no se cubrirán algoritmos de otros tipos, como los de selección de características, optimización, métodos bayesianos, entre otros.

CAPÍTULO 4

Manejo de Datos

Cada día estamos abrumados por datos. Datos de nuestras computadoras, nuestros teléfonos móviles, dispositivos vestibles como relojes inteligentes, de los autos que manejamos, datos que se guardan en la nube, de imágenes, ligas, videos que nos mandan, entre muchos otros. La cantidad de datos en el mundo y en nuestras vidas incrementan cada día, lo cual aumenta la complejidad de encontrar patrones en los mismos.

Que la gente encuentre patrones no es nada nuevo. Desde el principio de la civilización, los humanos han tratado de encontrar patrones a partir de datos. Por ejemplo, los granjeros buscaban patrones en sus cosechas, los antiguos astrónomos también buscaban patrones para el movimiento de los planetas, eclipses e, incluso, cometas, mismo que está documentado. También los cazadores buscaban patrones de comportamiento en migración animal. Hoy en día, los políticos y medios de comunicación buscan patrones en opiniones, tendencias de votantes, etcétera.

El trabajo de un experto en minería de datos es un poco como el de un bebé, en el sentido que el experto busca hacer sentido de los datos y encontrar patrones que gobiernan como su sistema está definido con respecto a sus datos, de la misma manera en la que un bebé intenta hacer sentido del mundo que le rodea con la información que tiene y su corta experiencia que ha acumulado en su vida.

Conforme nos sentimos cada vez más abrumados de los datos que genera el mundo que nos rodea, las oportunidades para poder modelar los datos, se vuelve más interesante. Este trabajo es el que conocemos como minería de datos. No solo se trata de revisarlos, se trata de discernir entre los datos que sirven de los que no, limpiarlos de ruido, escalarlos en un rango que convenga al algoritmo, encontrar patrones estructurales tanto estáticos —que siempre están ahí— como dinámicos —que cambian con respecto a sus entradas o a otros parámetros conforme pasa el tiempo—.

Existen diversas estrategias para poder conocer los datos. Muchas veces, no estamos familiarizados con los mismos y eso conlleva a que no se elijan las mejores herramientas o algoritmos para poder tratarlos o extraer información o patrones de ellos.

No es suficiente con tener datos, estos tienen que ser útiles. Para ello, se requiere analizar los datos y existen diversas técnicas que dependen del tipo de información que se esté recopilando para poder analizarla.

En este sentido, se recomienda tener definida la técnica a utilizar antes de implementarla. A grandes rasgos, podemos agrupar los datos por el tipo de información que se esté recopilando, es decir en datos cualitativos y cuantitativos.

- Datos cualitativos: estos datos se representan de manera verbal o gráfica.
- Datos cuantitativos: estos datos se representan de forma numérica y se basa en resultados tangibles.

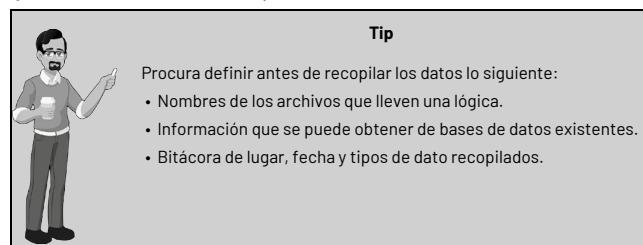
Para poder realizar un correcto análisis de los datos, es importante tomar en cuenta las siguientes consideraciones:

1. **Definir las preguntas.** En esta consideración se tiene que realizar preguntas como ¿qué datos quiero obtener? ¿Qué preprocesamiento voy a utilizar? ¿Cuántos datos requiero como mínimo? Entre otras.

2. Establecer prioridades en el análisis. En esta consideración se tiene que hacer las siguientes preguntas: ¿qué se va a medir? ¿Qué tipo de datos necesitas? ¿Cómo lo vas a medir? Esto es importante definirlo, especialmente, porque en la fase de recopilación de datos el proceso de medición puede respaldar o desacreditar el análisis.

3. Recopilar datos. Conforme se recopilen y organicen los datos es necesario tomar en cuenta lo siguiente: primeramente, ¿qué información que se requiere se puede recopilar de bases de datos ya existentes? Y dicha información, ¿qué tan fiable es? Es importante tener protocolos para asignar nombres a los archivos y documentar los tipos de datos y cuáles son las posibles observaciones que se puede tener. Es importante para evitar errores de cardinalidad o de adquisición.

4. Analizar los datos. El análisis de los datos es crucial para el óptimo funcionamiento del algoritmo. El análisis incluye: ver distribución de los datos, relaciones, tendencia, observar si tiene datos atípicos o muchos datos faltantes y determinar la razón por la cual esto ocurre con frecuencia.

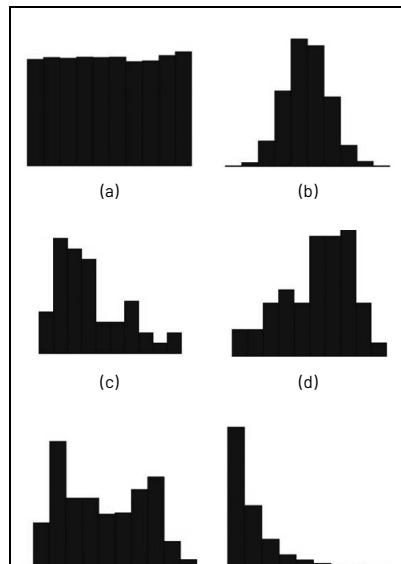


4.1. Análisis de distribución de los datos

El análisis de los datos es fundamental cuando se trabaja con algoritmos de inteligencia artificial y aprendizaje máquina. Por ejemplo, para características que son continuas, se tiene que examinar primero su media y desviación estándar. Esto se realiza para poder tener una idea de la tendencia central de los datos y la variación de los valores dentro del set de datos.

También se tienen que evaluar los valores mínimos y máximos para entender el rango en los que se encuentra cada característica de los datos. En este sentido, los histogramas de cada característica son útiles para entender los valores de cada característica y cómo se distribuyen en un rango de datos.

Cuando se generan histogramas, existen un número de formas que pueden tomar de acuerdo a cómo están distribuidos los datos. Estas formas se conocen comúnmente como distribución probabilística estándar.



(e) (f)

Figura 4.1. Histogramas ejemplo de seis diferentes tipos de distribución de los datos. a) Uniforme, b) normal(unimodal), c) unimodal sesgado izquierda, d) unimodal sesgado derecha, e) multimodal, f) exponencial

La figura 4.1 muestra los diferentes tipos de distribución de los datos. En una distribución uniforme, no se ve una clara tendencia de los datos. Es decir, están distribuidos en todo el rango con prácticamente los mismos valores de probabilidad. En la figura 4.1(b), se puede observar una forma común de una distribución normal. Dicha distribución se caracteriza por una fuerte tendencia a un valor central y una cierta simetría a ambos lados de la tendencia central.

Es importante mencionar que la simetría no tiene que ser perfecta para ser una distribución normal, pero sí tener una medida de tendencia central. Como ejemplo, si se toman una cierta cantidad de datos al azar, digamos, la altura de niños de cuarto de primaria en alguna escuela. En este ejemplo, será muy probable que se tenga una distribución normal, con una cierta tendencia hacia una cierta estatura y a sus lados, niñas y niños más altos y menos altos.

Histogramas que siguen una distribución normal también pueden ser descritos como unimodales, puesto que tienen un valor de tendencia central. Las figuras 4.1(c) y 4.1(d) muestran un ejemplo de un histograma unimodal que presentan sesgo. Esto es, una tendencia hacia un valor muy alto o muy bajo, respectivamente.

La figura 4.1(e) muestra una distribución multimodal donde se pueden ver dos o más valores de tendencia en la misma distribución. Eso ocurre con frecuencia cuando se tienen mediciones a través de un cierto número de diferentes grupos. Si volvemos al ejemplo de la medición de los niños de cuarto de primaria, una tendencia multimodal ocurriría si, además de medir a los niños y niñas de cuarto de primaria, midiéramos a los niños y niñas de primero de secundaria y los pusiéramos en la misma gráfica de tendencia. Se notaría –probablemente dos– tendencias centrales muy diferentes. La figura 4.1 (f), por su parte, sigue una distribución exponencial, en la que la probabilidad de pequeños valores es muy alta, pero la diferencia entre los valores más altos y los menos altos es muy grande –es decir, decrementa o incrementa de manera exponencial–.

4.2. Distribución normal

La distribución normal –también llamada distribución gaussiana o de Gauss– es la que más se utiliza en estadística y la que probablemente más se vea en los datos que normalmente se utilizan en algoritmos de inteligencia artificial. La distribución normal tiene forma de campana y, por lo tanto, es simétrica. La distribución está asociada por las funciones de densidad de probabilidad, las que, para una distribución normal, se calcula de la siguiente manera:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2} \quad (4.1)$$

Donde:

$f(x)$ = función de densidad de probabilidad.

σ = desviación estándar.

μ = media.

Donde x es cualquier valor, y μ y σ son parámetros que definen la forma de la distribución gaussiana. Es importante definir el tipo de la distribución, ya que la misma media μ pero diferente desviación estándar σ o viceversa puede generar una distribución muy diferente como se muestra en la figura 4.2.



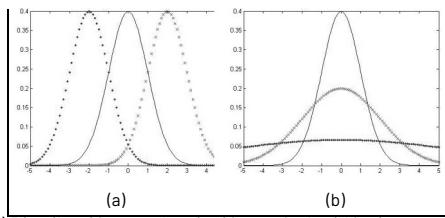


Figura 4.2. Tres distribuciones gaussianas. (a) Distribución con desviación estándar idéntica pero diferentes medias. (b) Distribución con media idéntica, pero diferente desviación estándar

4.3. Identificación de problemas de calidad en los datos

Una vez que se conocen los datos, se tiene que explorar cualquier problema de calidad en los mismos. Podemos definir un problema de calidad en los datos como cualquier dato inusual que se encuentre en la base de datos. Los problemas más comunes de los datos son: valores faltantes, problemas de cardinalidad irregular y valores atípicos –también llamados «outliers»–.

Es muy común que algunas instancias de los datos tengan valores faltantes. Lo primero que se tiene que realizar es revisar si se puede determinar la razón por la cual estos valores faltan. Es recomendable, si faltan muchos valores, revisar cada atributo y hacer una tabla de cuántas instancias tienen valores faltantes. Eso dará una idea de cuáles atributos tienen mayor incidencia de faltar y si el porcentaje es muy alto en cada atributo.

En muchas ocasiones, los problemas de valores faltantes son errores en la integración de los datos o en cómo se generaron valores a partir de otros atributos u observaciones. Si los valores faltantes fueron generados por errores en la integración de los datos, los valores faltantes pueden ser regenerados en algunos casos.

Algunas veces, se pueden tener valores faltantes por otras razones. Por ejemplo, algunos datos solo pueden estar disponibles o pueden recopilarse a partir de cierta fecha. En otras ocasiones, algunos datos deben de ser ingresados manualmente, lo que complica conocer si son correctos –aunque no faltantes– y se pueden cometer errores de captura.

Los problemas de calidad en los datos debido a cuestiones de cardinalidad irregular se dan cuando una característica tiene un dato que no es lo que se esperaría de una característica. Este valor inesperado o no correspondiente se conoce como cardinalidad irregular. No necesariamente es incorrecto o irregular siempre que se da un valor diferente.

Por ejemplo, si se está haciendo un censo y uno de los atributos es número de hijos, uno esperaría que fuera un valor pequeño –cero, uno, dos o tres–, aunque puede haber instancias donde una persona contestó doce o quince. Esto es un error de cardinalidad, pero no es incorrecto, pues es el valor real que la persona contestó y no un error de captura. Otro problema de calidad de los datos debido a una cardinalidad irregular se puede presentar cuando una característica presenta un valor mucho mayor de lo que se esperaría. No confundir esto con los valores atípicos (*outliers*), que son valores que salen de la norma de distribución y pueden presentar un error. En este ejemplo, supongamos una característica categórica de género. Supongamos que se asigna un cero cuando el género es masculino y un uno cuando es femenino. Si se tiene un valor de seis, es un valor que no puede estar correcto en cuestión de género y sale de la norma de cardinalidad de esta característica. La práctica de la sección 8.6 muestra un ejemplo de detección de errores de cardinalidad.

También en este ejemplo pueden ser registrados de diferente manera, aunque solo se esperen dos tipos de género. Cuando se integran los datos, puede ser que se encuentren con valores de masculino, femenino, F, M, hombre, mujer, H, M, 0, 1, etcétera, por lo que se requiere homogeneizar para no tener muchas categorías

cuando solo se deben de tener dos. También en español sería un problema de calidad en los datos que masculino y mujer comienzan con la misma letra y son obviamente dos categorías diferentes, por lo que no homogeneizar los criterios para capturar esta información y tener M de masculino y M de mujer conllevaría problemas en la calidad de los datos.

Por último, los llamados «outliers» son valores atípicos que se localizan muy alejados de la tendencia central de una característica. A grandes rasgos, se puede categorizar los valores atípicos como válidos y no-válidos. Una práctica ejemplo de detección de valores atípicos se muestra en la sección 8.7.

Los valores atípicos no-válidos son los que se observan en una instancia y son claramente un error en una instancia. Este tipo de valores atípicos no-válidos se consideran ruido en los datos. Este tipo de valores atípicos pueden presentarse por diferentes razones. Por ejemplo, por error humano, en donde se registran instancias de manera manual y, por ejemplo, al teclear edad, en lugar de 15, se escribe 150.

Los atípicos válidos son datos que, a pesar de ser estadísticamente muy diferente a la tendencia central de una característica, es correcto el valor. Un ejemplo simple puede ser salario. Cuando se capture salario, se puede tener una tendencia salarial en cierto rango, pero se pueden tener individuos que ganen muchas veces más que la inmensa mayoría de la muestra en cuestión. En este caso, estadísticamente, es un valor muy alejado de la media, pero válido para la muestra.

Hay varias maneras para determinar los valores atípicos en los datos. Una de ellas es examinar los valores mínimos y máximos de cada característica y usar el conocimiento que se tiene de los datos para determinar si es un valor factible o no. Por ejemplo, se tiene un valor de -10 en un atributo de edad, esto es claramente un error. En estos casos, se tiene dos opciones, poder corregir si se tiene el conocimiento de qué valor podría tener la instancia o remover la instancia completa de acuerdo a las circunstancias. También se pueden realizar métodos de imputación para determinar el valor faltante. La importancia de la imputación de los datos se muestra en la sección 4.5, mientras que la práctica de la sección 8.10 utiliza un método de imputación múltiple llamado MICE.

Para mostrar la importancia de saber manejar los valores atípicos y cómo estos pueden modificar la tendencia de los datos, se muestra la figura 4.3.

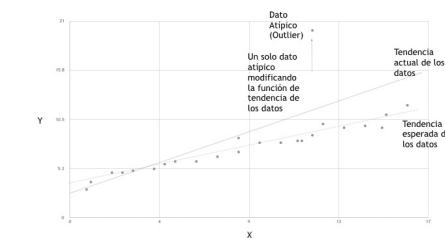


Figura 4.3. Tendencia de los datos cuando es modificada por un dato atípico

 **Tip**

Es importante dedicar tiempo a determinar si los «outliers» son válidos o no-válidos, ya que muchos algoritmos de aprendizaje son susceptibles a estos valores atípicos y los resultados podrían variar drásticamente si estos valores no son tratados adecuadamente.

Otra de las maneras para identificar los valores atípicos es comparar la brecha entre la mediana, los valores mínimos, máximos y los cuartiles 1 y 3. Si la brecha entre el valor del 3.^{er} cuartil y el valor máximo es considerablemente mayor, es muy probable que se trate de un valor atípico que tiene que ser revisado. Así mismo, la brecha entre el valor del 1.^{er} cuartil y el valor mínimo es considerablemente menor, se trata de un

outlier. Los diagramas de caja (*boxplot*) y los histogramas son representaciones visuales de datos que pueden ayudar a identificar este tipo de valores.

4.4. Manejo de valores faltantes

La manera más fácil de manejar, aunque normalmente no la mejor, es la de deshacerse de la instancia completa que tenga los valores faltantes. Sin embargo, esto representa pérdida de los datos, pues las demás características de esa instancia sí tienen datos válidos. Tampoco remover una característica completa es una buena idea. Como regla general, solo las características que carezcan por lo menos de 60 % de los datos de esa característica sería conveniente removerla.



Tip

El nombre del juego en cuanto a selección de algoritmos se llama: Conoce tus datos. Mientras más información se tenga sobre los datos y se conozca mejor, como se obtuvieron, que tipo de datos son, si tiene valores atípicos (llamados también Outliers), la frecuencia y el grado con el que pueden cambiar, etcétera, mayor será el éxito del algoritmo a escoger.

Existen algunas técnicas para compensar esos valores cuando el número de datos faltantes es menor. Uno de ellos es la llamada imputación. La técnica de imputación reemplaza valores de datos faltantes con valores factibles basados en un estimado de los valores de la característica que sí está presente. Para asegurar que la imputación funcione de manera correcta, los valores faltantes deben de ser acompañados con un análisis de la tendencia central de esa característica. Para valores continuos, la media o mediana son las características más comunes para realizar la imputación, para valores categóricos, la moda es lo más común, aunque existen muchas maneras de imputar datos.

Las técnicas de imputación suelen dar buenos resultados y evitar las pérdidas asociadas de datos por eliminar completamente instancias o características de un set de datos.

Por último, es importante conocer si los valores faltantes pueden ser reemplazados por algún número que represente la tendencia central de la distribución, si se puede eliminar la instancia completa o no se puede hacer ninguna de las anteriores. En este caso, si se identifica el valor faltante, se puede llenar los atributos faltantes mediante valores improbables. Por ejemplo, si se tiene un atributo de edad y no se conoce, pero tampoco se puede imputar el valor ni borrar la instancia, se puede poner un -1, que indica que ese valor no existía y realizar anotaciones en la base de datos de que dicho cambio se realizó.

4.5. Imputación de datos

Muchos de los conjuntos de datos que se obtienen de diferentes fuentes contienen distintas pérdidas y anomalías asociadas a diferentes razones. Por ejemplo, puede ser que no se tienen datos válidos o no existen debido a protocolos de adquisición deficientes o erróneos, errores en los instrumentos de medición, inconsistencias con entradas de manera manual, entre otros.

La pérdida de estos valores puede representar un problema y afecta cualquier conjunto de datos sobre el cual se busque realizar algún algoritmo de IA. La presencia de esas pérdidas de datos usualmente requiere de una etapa de preprocesamiento, en la cual se preparen los datos para que resulten útiles.

En ciencia de datos, usualmente, se tienen dos enfoques para lidiar con estos valores faltantes: omitir las instancias con valores faltantes o realizar técnicas de imputación y estimar los datos faltantes utilizando los valores existentes.

La imputación es una técnica para reemplazar valores perdidos con valores que se encuentran. El objetivo de las técnicas de imputación es la de tener una base de datos con la menor cantidad de instancias faltantes que contengan la misma distribución que los datos existentes para que puedan ser analizados.

En general, las técnicas de imputación se pueden clasificar de la siguiente manera:

- Técnicas por información externa o deductiva.
- Técnicas deterministas.
- Técnicas aleatorias.

4.5.1. Técnicas por información externa o deductiva

Este tipo de técnicas de imputación se realizan cuando los datos faltantes se pueden deducir de otras instancias completas, siguiendo algunas reglas predefinidas.

También, se utilizan las tablas llamadas LUT (o *look-up table*, por sus siglas en inglés) que sirve como base para realizar la imputación debido a que tiene información relacionada para imputar los datos faltantes.

4.5.2. Técnicas deterministas de imputación

Las técnicas deterministas de imputación funcionan cuando, de acuerdo a las mismas condiciones de los datos, producen las mismas respuestas. Entre estas técnicas se encuentran:

- **Imputación por regresión:** se ajusta un modelo usualmente lineal, aunque puede ser polinomial. Los datos faltantes se toman del ajuste que se realizó por medio de la regresión.
- **Imputación de la media (o moda):** en este método de imputación se llena el dato faltante de cada atributo con la media de las instancias no faltantes en caso de atributos de características cuantitativas o con la moda en caso de atributos con características cualitativas. Este tipo de imputación puede tener la desventaja que reduce su varianza, por lo que se pierde la distribución de su atributo.
- **Imputación por media de clases:** la imputación se realiza por cada clase dentro del mismo atributo. Se calcula la media de las instancias que tienen valor por cada clase y se llena con el valor faltante para cada una de las clases.
- **Imputación por vecino más cercano:** se calcula la distancia entre la instancia a imputar, usualmente, por medio de la distancia euclíadiana y los datos que tienen valor establecido, como se mostró en la sección 3.11. Una vez que se calcula el dato más cercano, se utiliza este para imputar la instancia faltante.
- **Imputación por algoritmo EM:** el algoritmo EM (*expectation maximization*, por sus siglas en inglés) es un método iterativo muy utilizado en estadística, el cual consiste en estimar los parámetros para máxima semejanza donde el modelo depende de variables no observadas. Itera entre la expectativa —que crea una función de la estimación de los parámetros— y la maximización —que calcula los parámetros derivados de la estimación—.

4.5.3. Técnicas estocásticas de imputación

Las técnicas estocásticas de imputación son aquellas que, cuando se repite el método bajo las mismas condiciones, producen resultados diferentes. No son tan utilizadas, ya que no se producen resultados concretos y es difícil estimar qué tan buena es la imputación. Entre estas técnicas se encuentran:

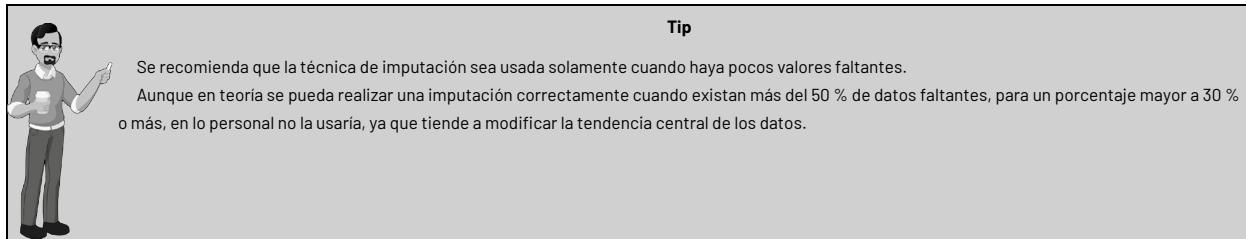
- **Imputación aleatoria:** en este caso, se toman las posibles observaciones desde el menor hasta el mayor valor que se encuentra en ese atributo. Posteriormente, se selecciona un número que esté en el intervalo de ese atributo de manera aleatoria.

- **Imputación secuencial:** «hot-deck»: este método funciona tomando de manera secuencial datos que ya se tienen para reemplazar los datos faltantes. Se comienza desde un dato inicial de manera aleatoria y se reemplaza el dato faltante con ese dato inicial, después se toma el siguiente dato que se tiene y se reemplaza con el siguiente dato faltante y así sucesivamente.

4.5.4. Métodos de imputación múltiple

Los métodos de imputación múltiple consisten en realizar varias imputaciones de las observaciones faltantes para luego analizar los conjuntos de datos completados y combinar los resultados obtenidos, con el fin de obtener una estimación final. Entre estas técnicas se encuentran:

- **Imputación múltiple por cadenas de Markov (MCMC):** este método de imputación es un proceso de selección aleatoria generado por medio de cadenas de Markov. Utiliza simulación paramétrica generando muestras aleatorias a partir de métodos bayesianos.
- **Imputación múltiple por ecuaciones encadenadas (MICE):** este método consta de hacer copias de los datos, en donde se realiza una regresión en uno de los atributos y con ello se hacen imputaciones a los otros atributos que tienen datos faltantes. Con esos datos, se ajusta el primer atributo del que se hizo la regresión y se revisan los otros datos que originalmente tenían datos faltantes. Este proceso es iterativo mientras siga habiendo datos faltantes, como lo muestra el ejercicio de la sección 8.10.



4.6. Determinar relaciones entre atributos de los datos

Cuando se tienen muchos atributos o muchas instancias de cada atributo, es indispensable determinar qué relaciones se pueden realizar entre cada atributo. Para mostrar las relaciones que tienen entre sí los atributos, utilizaremos el ejemplo de los gorriones, mismo que se muestra en el apéndice A.3. Un ejemplo del código que se utiliza para determinar las relaciones entre los atributos se muestra en la sección 8.5.

En este ejemplo, la tabla nos muestra la diferencia entre gorriones supervivientes y no supervivientes después de una tormenta mediante cinco atributos, todos ellos de longitud (en mm): X1 - longitud total del gorrión, X2 - extensión del ala, X3 - longitud del pico, X4 - longitud del húmero y X5 - longitud del esternón. Al ver la tabla de las medidas de los gorriones (apéndice A.3), no está claro si existe una relación entre los supervivientes y los no supervivientes. En este ejemplo, se muestran algunas herramientas visuales para poder determinar las relaciones entre los diferentes atributos.

4.6.1. Gráfica de dispersión (scatter)

La gráfica de dispersión –comúnmente llamada scatter– muestra la relación que existe entre un atributo y otro –también puede representarse en una gráfica 3D–. En el caso de estudio de los gorriones, se pueden tener gráficas de dispersión de las siguientes características: X1 - X2, X1 - X3, X1 - X4, X1 - X5, X2 - X3, X2 - X4, X2 - X5, X3 - X4, X3 - X5 y X4 - X5. La primera característica elegida como argumento se mostrará en el eje de la x, mientras que la segunda en el eje de la y, como se muestra en la figura 4.4.

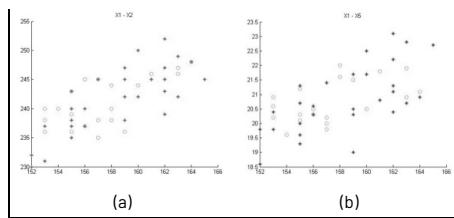


Figura 4.4. Gráfica de dispersión 2D. a) Dispersion X1 (longitud total) vs X2 (extensión del ala). b) X1 vs X5 (longitud del esternón)

Como puede observarse en la figura 4.4, la dispersión de los datos muestra que no existe una tendencia clara de los gorriones supervivientes –círculos– con respecto a los no supervivientes –rombos– con respecto a las características de la longitud total con respecto a la extensión del ala, de la misma manera, la longitud total y la longitud del esternón no muestra una clara distribución, lo que lo convierte en un problema de clasificación un poco más complejo.

En general, recomiendo ver cada una de las características con las demás. Hay un método gráfico que puede mostrar la matriz de todas las características llamada «plotmatrix», como se muestra en la figura 4.5.

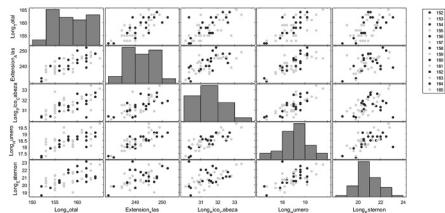


Figura 4.5. Gráfica de dispersión tipo matriz

La gráfica de la figura 4.5 muestra toda la colección de características acomodadas en una matriz. Esto es útil para poder explorar las relaciones que existen entre grupos de características. Cada fila representa la característica en cuestión y su relación con la característica que se encuentra en la fila, mientras que la diagonal muestra la distribución de dicha característica.

4.6.2. Gráfica de barras apilada (stacked bar)

En ocasiones, es útil mostrar las características en términos de porcentajes en el sentido que el conjunto de las características sean un todo –es decir, 100 %–. Si continuamos con el ejemplo de los gorriones, podría ser útil saber el número de supervivientes vs no supervivientes para determinar si las relaciones encontradas puedan ser estadísticamente válidas. Es decir, cuando tenemos que una característica hace un gran porcentaje –90 %, por ejemplo– del todo, no podríamos determinar si las relaciones entre características son válidas, pues hay muchas más instancias de una clase que de otra, esto se conoce usualmente como desbalance de clases. En el caso de estudio, la figura 4.6 muestra que la relación entre supervivientes y no supervivientes es de 42 % - 58 %, lo cual, si tiene suficientes instancias, podríamos decir que estadísticamente es válido, aunque no sean exactamente el mismo número.

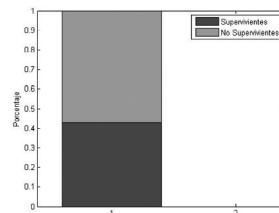


Figura 4.6. Gráfica de barras apilada

4.6.3. Gráfica de caja (boxplot)

Este tipo de gráficas son muy útiles para poder ver visualmente cómo están distribuidos los datos. En las figuras 4.7 y 4.8, se muestran un par de ejemplos de este tipo de gráficas. La gráfica de caja muestra la distribución de las características que se deseen en el eje de la x, mientras que el eje de la y muestra su distribución como si fuera una caja.

La línea horizontal inferior de cada característica y la línea punteada conectada a esta muestra la distribución del primer cuartil de los datos –es decir, de 0 % a 25 %–, posteriormente, se encuentra la «caja» que se posiciona del 1.^{er} al 3.^{er} cuartil –del 25 % al 75 %–. La línea roja que atraviesa cada caja es la media de los datos, posteriormente, se visualiza una línea vertical que se extiende hasta el cuarto cuartil para representar toda la distribución de los datos. Estas líneas que culminan en otra línea horizontal generalmente se llaman «whiskers».

Si existiese un dato atípico (*outlier*), se representa fuera de la distribución de la gráfica de caja mediante un círculo o un asterisco, misma que en las gráficas 4.7 y 4.8 no se encuentra ninguno.

En el caso de estudio de la figura 4.7, se muestra la distribución de la variable X1 - longitud total del gorrión tanto de los supervivientes como de los no supervivientes. En dicha gráfica se puede observar la relación entre los supervivientes y no supervivientes, mostrando que la media de los supervivientes es menor que los no supervivientes, mientras que el inicio de la caja –25 %– no muestra ninguna diferencia estadísticamente significativa. Así mismo, los datos que están distribuidos del inicio al primer cuartil y del tercer cuartil al final de los datos indican que la distribución longitud de los no supervivientes es mucho mayor, por lo que no podríamos concluir que los supervivientes tenían una longitud menor, debido a esta distribución.

Este tipo de gráficas de distribución en forma de caja ayuda a comparar rápidamente cómo se distribuyen los datos y ver si estadísticamente se pueden determinar relaciones entre diferentes características. Este tipo de gráficas no se limita a dos características, por simplicidad, se realizaron supervivientes contra no supervivientes, pero no existe un límite de características que puedan ser visualizadas en este tipo de gráficas.

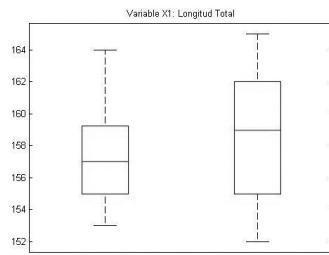


Figura 4.7. Gráfica de caja. Característica X1 - longitud total

La figura 4.8 muestra una comparación interesante de la distribución de la característica X2 –extensión del ala– entre los supervivientes y los no supervivientes. La caja muestra que su distribución de 25 % a 75 % es prácticamente idéntica, es decir, la extensión de sus alas es muy parecida.

Sin embargo, la media de los supervivientes es menor, es decir, en promedio, los supervivientes son más pequeños. También se puede observar que la distribución de los supervivientes es más compacta, es decir, están distribuidos en un rango menor de milímetros que los no supervivientes. Este tipo de gráficas permite ver mejor la relación entre las diversas características.

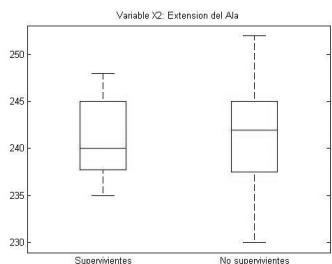


Figura 4.8. Gráfica de caja. Característica X2 - extensión del ala.

4.6.4. Consideraciones de las relaciones entre variables

Existen medidas formales para encontrar relaciones entre las variables como lo son la covarianza y la correlación. Sin embargo, se necesita tener cuidado de usarlas si no se ha realizado una inspección previa a los datos. Un ejemplo de cómo los datos pueden tener medidas idénticas de correlación pero diferente distribución y relación entre atributos es lo que se conoce como cuarteto de Anscombe, como se muestra en la figura 4.9.

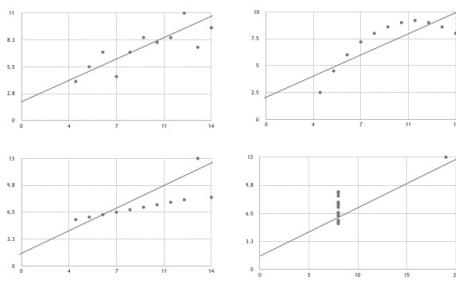


Figura 4.9. Cuarteto de Anscombe

4.7. Simplificación de datos escasos

En algunas ocasiones, se presentan datos que parecen incompletos o inexistentes. Como se comentó antes, si se requieren los datos y no están, se pueden realizar técnicas para incluir datos –como imputación – sin que la tendencia central cambie o se tenga que eliminar un gran número de instancias.

Sin embargo, en algunas ocasiones, algunos atributos tienen un valor de cero para muchas instancias. Esto se ve muy frecuentemente cuando son muchos atributos y no necesariamente deben de estar todos. En este caso, se puede transformar los datos en un formato que ocupe menos espacio. Esto se conoce como «sparse data» o simplificación de datos escasos.

Un ejemplo muy frecuente de este tipo de «escasez de datos» sería el de artículos de un supermercado o tienda. Sería inviable e impráctico tener muchos «0» por cada artículo que cada cliente no compró y solo datos diferentes de 0 en artículos que sí compró cada cliente. Este ejemplo se vería de la siguiente manera:

0, 0, 31, 0, 0, 0, 55, 0, 0, 0, «Cliente tipo A».

0, 0, 0, 0, 49, 0, 0, 0, 0, 0, «Cliente tipo B».

En lugar de eso, mediante esta técnica, los atributos que no contienen un cero pueden ser explícitamente identificados de acuerdo a su número de atributo y el valor que le corresponde de la siguiente manera:

{ 2 31, 4 55, 12 «Cliente tipo A»}

{ 4 49, 12 «Cliente tipo B»}

En este caso, los datos tipo «sparse» significan lo siguiente. En la primera instancia tenemos que hay dos atributos con 0, seguido de un treinta y uno, posteriormente, cuatro atributos seguidos con un 0, después un

55 para un total de doce atributos para el «cliente tipo A». En el caso del cliente tipo B, se tienen cuatro veces atributos con 0, posteriormente, un 49. Nótese que las comas solo se colocan después del atributo que no tiene 0 y va seguido del número de 0 anteriores y posteriores. El último número antes del nombre de la clase –en este caso cliente tipo A y cliente tipo B– es el número total de atributos que se tiene, en este caso, doce.

En este caso, se necesita especificar que los datos son de tipo «sparse». Este tipo de «simplificación» de datos es muy útil para un conjunto de datos con muchas instancias y una gran cantidad de atributos, muchos de los cuales no tienen un valor o es 0.

4.8. Cantidades y descripciones de los datos

En estadística, los datos son típicamente categorizados en cuantitativos y cualitativos. Por ejemplo, consideremos una taza de té. Se puede describir esa taza de manera cuantitativa y cualitativa.

Si se quiere describir la taza de té de manera cualitativa con la lista de los siguientes atributos:

- Aromático.
- Taza blanca.
- Contenido verde claro.
- Caliente.

Se puede cuantificar los atributos de la taza de té de manera no numérica, es decir, cualitativa, pero también de manera cuantitativa de la siguiente manera:

- 230 ml.
- 49 calorías.
- 90 °C.
- \$53 pesos.

Todos estos son atributos cuantitativos que describen la taza de té. Se pueden describir los atributos cuantitativos y cualitativos en mayor detalle de la siguiente manera:

- Datos nominales.
- Datos ordinales.
- Datos de intervalo.
- Datos de proporción.

Esa descripción, generalmente, se puede determinar con respecto a los operadores que pueden ser válidos para cada tipo de datos. Por ejemplo, si tenemos dos colores, podemos determinar si los dos colores son iguales –es decir, que dos instancias tengan el mismo color–. Sin embargo, no se pueden utilizar operadores de suma –no se pueden sumar dos colores–, operadores relacionales –no puedo, por ejemplo, decir: el verde es mayor o igual al marrón–. De acuerdo a estas propiedades, los datos como colores son nominales.

En cuestión de los ordinales, las observaciones son aquellas que requieren un tipo de orden, ya sea cualitativo o cuantitativo. Por ejemplo, si regresamos a la taza de té, su observación ordinal sería la medida de caliente. Si yo lo manejo en grados centígrados o en qué tan caliente está, las observaciones hechas pueden estar en función de un orden. De esa manera, muy caliente tiene una mayor temperatura que caliente, tibia, fría y muy fría. En los datos ordinales no solo se pueden ordenar, sino que sus niveles para poder ordenarlo deben de estar claramente definidos.

Existen otros ejemplos en donde, a pesar de ser datos numéricos, no pueden ser ordenados. Un ejemplo es el código postal. En algunos países, los códigos postales son datos numéricos de cinco dígitos. El número indica la zona, pero no está ligada ni a altitud o latitud, por lo que no podríamos ordenarla de menor a mayor esperando una ordinalidad de los datos.

Tanto los datos nominales como ordinales pueden ser tanto cualitativos como cuantitativos, mientras que los de intervalo y proporción son únicamente cuantitativos. La siguiente tabla muestra los atributos de datos y las operaciones que se pueden realizar con ellos.

Operación	Nominal	Ordinal	Intervalo	Proporción
* o /	No	No	No	Sí
+ o -	No	No	Sí	Sí
< o >	No	Sí	Sí	Sí
= o i=	Sí	Sí	Sí	Sí
Ejemplo	Género	Caliente/Frío	Año	Edad

Tabla 4.1. Atributos de datos y las operaciones que se pueden realizar con cada uno

La diferencia entre intervalo y proporción suele ser un poco más compleja que entre ordinal y nominal. En general, los datos de tipo intervalo tienen un inicio –generalmente cero–, mientras que los de proporción su inicio no necesariamente lo es.

Por ejemplo, en la tabla 4.1. se observa el ejemplo de intervalo como año y proporción como edad. En el ejemplo de la tabla se muestra qué edad comienza en cero, pues no hay edades que comiencen en números negativos o menores a cero, mientras que año no tiene un origen definido y puede comenzar en algún otro número.

La temperatura puede ser tanto proporción como intervalo, depende de la escala que se use. Es decir, si se mide la temperatura en grados Kelvin, 0 K es cero absoluto, por lo que la temperatura en ese caso es proporción. Por otro lado, si se mide en grados Celsius o Fahrenheit, es un intervalo, pues el cero no es el origen para ninguno de los dos.

4.9. Normalizar y escalar los datos

Tanto las entradas como las salidas de un algoritmo de aprendizaje máquina son típicamente vectores, números de punto flotante, enteros, etcétera. En el mundo real, las observaciones que se realizan pueden ser de tipo nominal, ordinal, intervalo o proporción. Los datos tanto nominales como ordinales no son necesariamente cuantitativos como se describió en el apartado anterior, por lo que es importante en muchas ocasiones convertir estos datos en números que puedan ser útiles para el algoritmo.

Algunos algoritmos de inteligencia artificial requieren que todos los datos se centren en un rango específico de valores, normalmente de -1 a +1 o de 0 a +1. Incluso si no se requiere que los datos se encuentren dentro de valores como entre -1 a +1, es una buena idea generalmente asegurarse de que los valores se encuentren dentro de un rango específico.

La normalización de los datos es algo que vemos mucho más seguido de lo que pensamos. Uno de los ejemplos más comunes de normalización es el de porcentajes. Si se ve en una tienda o centro comercial algún letrero que diga, por ejemplo, 20 % de descuento, se puede realizar rápidamente el cálculo del 80 % del valor de la etiqueta que costaría dicho artículo. Ese 20 % puede representar un valor muy pequeño –si hablamos, por ejemplo, de dulces– o muy grande –por ejemplo, una televisión o un ordenador–, pero el porcentaje es el mismo y puede ser fácilmente calculado.

Existen diversas maneras de escalar los atributos. En este apartado se abordarán la codificación uno-a-n y varios tipos de normalización tanto para atributos ordinales como cuantitativos.

4.9.1. Codificación uno-a-n

Existen diversos métodos para normalizar valores ordinales, uno de ellos es el de codificación uno-a-n. Uno-a-n es una manera simple de normalización. Por ejemplo, consideremos como caso de estudio los datos de los lirios (apéndice A.1).

Como se ha comentado, en este ejemplo se busca clasificar las clases de lirios, y para eso se tiene el largo y ancho del pétalo y el largo y ancho del sépalo, además de la clase a la que pertenece. Las clases para este ejemplo son: «*Iris setosa*», «*Iris versicolor*» e «*Iris virginica*». Para este ejemplo, podemos tomar tres instancias, es decir, una instancia de cada clase de manera aleatoria:

5.7, 3.8, 1.7, 0.3, *Iris setosa*.

5.0, 2.0, 3.5, 1.0, *Iris versicolor*.

6.3, 2.9, 5.6, 1.8, *Iris virginica*.

Como se puede observar, la especie es no numérica, por lo que sería útil normalizarla, en este caso, con la codificación uno-a-n.

Los primeros cuatro atributos son observaciones de proporción, pues su origen es cero para cada uno. Sin embargo, el quinto atributo es nominal, también llamado categórico, pues describe una categoría o una clase, que, en este caso, es uno de los tres tipos de lirio.

En la normalización uno-a-n, el algoritmo tendrá para el caso de estudio que nos ocupa, tres atributos de salida o categóricos, es decir, uno para cada clase de lirio. En este caso, el algoritmo de aprendizaje máquina tendrá que ser entrenado para aceptar cuatro atributos de entrada y arrojar tres atributos de salida, dependiendo de la clase de lirio de la que se trata.

Para generar la normalización uno-a-n, se agrega la salida ideal para cada una de estas salidas, en este caso, podría ser +1 y una salida negativa para definir las clases que no son, por ejemplo, -1. Esto es particularmente útil cuando se tratan de algoritmos como redes neuronales, en donde se asignan los pesos de acuerdo al valor que se quiere para el atributo de salida. En el caso de estudio que pusimos quedaría para *Lirio setosa* de la siguiente manera:

5.7, 3.8, 1.7, 0.3, **1, -1, -1**

De la misma forma, *Lirio versicolor* quedaría de la siguiente manera:

5.0, 2.0, 3.5, 1.0, **-1, 1, -1**

Finalmente, *Lirio virginica* quedaría de la siguiente manera:

6.3, 2.9, 5.6, 1.8, **-1, -1, 1**

Si el rango que se le quiere dar es de 0 a 1, solo se podría reemplazar el -1 por 0, para poder realizar este tipo de normalización.

4.9.2. Normalización de valores ordinales

Los valores ordinales, como se comentó anteriormente, no son necesariamente numéricos. Sin embargo, tienen un ordenamiento implícito. Por ejemplo, consideremos los niveles de educación por los que un estudiante progresá en algunos países. Los niveles están numerados, pero no ocurren en un orden y parece que el orden se repite.

Para poder normalizar un set ordinal, se tiene que preservar el orden y codificaciones como el uno-a-n pierde ese orden. En este sentido, si tenemos, por ejemplo, que los atributos de salida son *Lirio versicolor* y su codificación es -1, 1, -1, no significaría necesariamente que sea el segundo, pues no tienen un orden específico.

En este ejemplo específico de los grados de los estudiantes, podríamos ordenarlos de la siguiente manera:

- Preescolar 1(1).
- Preescolar 2 (2).
- Preescolar 3(3).
- Preprimaria(4).
- Primaria 1(5).
- Primaria 2(6).
- Primaria 3(7).
- Primaria 4 (8).
- Primaria 5(9).
- Primaria 6(10).
- Secundaria 1(11).
- Secundaria 2 (12).
- Secundaria 3(13).
- Bachillerato 1(14).
- Bachillerato 2 (15).
- Bachillerato 3(16).

En este ejemplo, asumimos que se incluyeron los primeros tres grados de preescolar y preprimaria, el cual en muchos colegios es opcional y no está considerado en ciertas escuelas. También se asume que primaria 1 es 1.^{ero} de primaria, lo mismo que preescolar 1 es 1.^{ero} de preescolar, etc., mientras que bachillerato en muchas escuelas es semestral y no anual. Para ejemplificar la normalización de categorías nominales, hacemos dichas menciones.

En este caso, la normalización de los datos —valores— nominales sería la numeración que está en paréntesis.

4.9.3. Normalización de valores cuantitativos

Tanto las observaciones de intervalo como las de proporción se pueden normalizar de la misma manera. Lo primero que se tiene que hacer con los valores cuantitativos es observar el rango en el cual se encuentran dichos valores y el intervalo al que se quiere normalizar. En este sentido, no todos los valores de tipo cuantitativo requieren ser normalizados. Esto depende también del tipo de algoritmo a implementar y el rango en el cual se encuentran los valores.

En este caso, se ejemplificará la normalización con la base de datos de enfermedad coronaria, llamado HeartDisease.xls, misma que se explica a mayor detalle en el apéndice A.2. En esta base de datos se tienen cinco atributos, los cuales son: *sbp* (atributo tipo intervalo, presión sistólica), *famhist* (atributo binario, historial familiar), *obesity* (atributo tipo intervalo, índice de masa corporal), *age* (atributo tipo proporción, edad) y, finalmente, *chd*(atributo binario, enfermedad coronaria).

Tomemos en este ejemplo el tercer atributo, *obesity*, para poder explicar de manera sencilla este tipo de normalización.

Como se explicó anteriormente, algunos algoritmos de aprendizaje máquina requieren que los datos numéricos estén normalizados a un rango específico, pero otros no. Por ello, es importante saber los rangos en los que trabajan los datos para generar las salidas relevantes. En este ejemplo, todos los datos serán normalizados de -1 a +1, por lo que se necesitará tomar en cuenta los siguientes máximos y mínimos:

- **Máximo de los Datos:** el valor más alto de la observación sin normalizar.
- **Mínimo de los Datos:** el menor valor de la observación sin normalizar.

- **MáximoNormalizado:** el valor más alto límitrofe al que el máximo de los datos será normalizado.
- **MínimoNormalizado:** el valor más bajo límitrofe al que el mínimo de los datos será normalizado.

De esta forma, los valores del presente ejemplo corresponderán de la siguiente manera:

- **MáximoModelosDatos:** 46.58
- **MínimoModelosDatos:** 14.7
- **MáximoNormalizado:** 1
- **Mínimo Normalizado:** -1

Para normalizar los datos, primeramente, se calculan los rangos, de máximo de los datos al mínimo de los datos y de máximo normalizado al mínimo normalizado, de la siguiente manera:

$$\text{Rango de Datos} = \text{MáximoModelosDatos} - \text{MínimoModelosDatos} = 46.58 - 14.7 = 31.88.$$

$$\text{Rango Normalizado} = \text{MáximoNormalizado} - \text{MínimoNormalizado} = 1 - (-1) = 2.$$

En el caso de la normalización, pongamos como ejemplo una observación de índice de masa corporal de este ejemplo de $\text{imc} = 23$. Para esta normalización, primeramente, se necesita determinar en qué punto el número se encuentra con respecto al **rango de datos**, lo cual se realiza de la siguiente manera:

$$D = \text{ejemplo(imc)} - \text{Mínimo de los datos} = 23 - 14.7 = 8.3.$$

Se convierte de esta manera a porcentaje:

$$DPct = D / \text{Rango de datos} = 8.3 / 31.88 = 0.2603 (\text{o } 26.03\%).$$

De la misma manera, se calcula el rango normalizado:

$$dNorm = \text{Rango normalizado} * DPct = 2 * 0.2603 = 0.5207.$$

Por último, se agrega dNorm al mínimo normalizado, quedando de la siguiente manera:

$$\text{Normalizado} = \text{Mínimo normalizado} + dNorm = \mathbf{-0.4792}.$$

4.10. Representación de los datos

Existen diferentes maneras para poder representar los datos en algoritmos de inteligencia artificial, algunos de ellos son abordados en la presente sección.

4.10.1. Tablas de decisión

Una de las más simples y más rudimentarias maneras de representar la salida de un algoritmo de inteligencia artificial es hacer lo mismo que con la entrada, es decir, por medio de una tabla de decisión.

Por ejemplo, la figura 4.10 muestra las primeras diez instancias de los datos correspondientes a enfermedad coronaria (apéndice A.2) donde el atributo de la columna E —llamado chd— es el atributo de decisión, esto es, la salida del sistema que, en este caso, es de tipo nominal binario, donde cero significa que no tiene una enfermedad coronaria, mientras que uno significa que sí lo tiene.

	A	B	C	D	E
1	sbp	famhist	obesity	age	chd
2	160	1	25.3	52	1
3	144	0	28.87	63	1
4	118	1	29.14	46	0
5	170	1	31.99	58	1
6	134	1	25.99	49	1
7	132	1	30.77	45	0
8	142	0	20.81	38	0
9	114	1	23.11	58	1
10	114	1	24.86	29	0

Figura 4.10. Ejemplo de una tabla de decisión

4.10.2. Árboles de decisión

Los árboles de decisión son representaciones gráficas de posibles soluciones a una decisión basada en ciertas condiciones. Se llama árbol de decisión porque comienza con un rectángulo —llamado nodo raíz— en el cual se bifurca en diferentes raíces, como un árbol. Una representación de un árbol de decisión con los datos tomados del apéndice A.5: «Condiciones climáticas», se muestra en la figura 4.11.

Los árboles de decisión son útiles, no solo debido a que, como son métodos visuales, proveen exactitud y facilidad de interpretación, sino debido a que requieren un proceso documentado y sistemático, por lo que son un buen método de representar los datos.

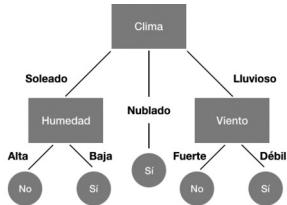


Figura 4.11. Ejemplo de un árbol de decisión

4.10.3. Reglas de clasificación

Las reglas de clasificación son una alternativa popular a los árboles de decisión. El antecedente —también llamado precondición— de una regla es una serie de pruebas que se parecen a los nodos en los árboles de decisión, mientras que el consecuente —también llamado conclusión o decisor— muestra la clase que aplican a las instancias de dicha regla.

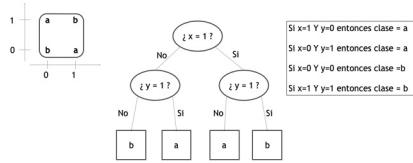


Figura 4.12. Regla de clasificación

En el ejemplo de la figura 4.12, se muestra la regla de clasificación que se comporta como una compuerta XOR, este es un ejemplo muy común cuando se abordan temas de lógica digital, en este caso, para dos atributos —en lógica digital, entradas—. En este ejemplo, cuando x e y son idénticos —ya ambos sean cero o ambos sean uno—, la clase pertenecerá a b , mientras que, si son diferentes, la clase será a .

Esta manera de representar los datos, generalmente, es muy simple y práctico. Sin embargo, es generalmente usado cuando se tienen atributos de decisión booleanos —binarios— y un set de atributos de entrada pequeño, lo que no es muy común en casos de estudio reales.

4.10.4. Agrupaciones

Cuando se trata de aprendizaje en el cual se tiene que separar por clases o grupos, la salida, es decir, el atributo de decisión toma la forma de un diagrama en el cual visualmente se puede observar que las instancias se separan en grupos como se muestra en la figura 4.13.

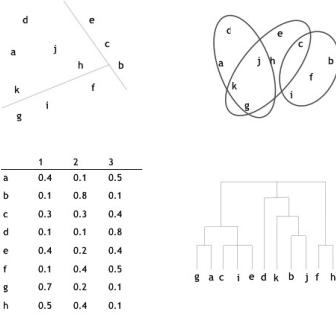


Figura 4.13. Diferentes maneras de visualizar las agrupaciones

Diagramas como el de la figura 4.13d (inferior derecha) también se les conocen como dendrogramas. Este término se aplica de la misma manera que los diagramas de árbol (del griego *dendron* que significa «árbol»). Los algoritmos de agrupación, en general, tienen etapas en las cuales un set de reglas o un árbol de decisión es creado para ubicar cada instancia en el grupo que le corresponde.

4.11. Metadatos

El concepto metadatos es otro que se confunde, se menciona datos y metadatos como dos conceptos sinónimos, aunque tienen diferencias fundamentales. Primeramente, ya se comentó el término datos. Pongamos como ejemplo el siguiente: asumamos por un momento que tienes veinte años. Si queremos guardar dicha información, el número veinte serán los datos y serán representados en un atributo que se llame, por ejemplo, edad. Hasta aquí todo bien.

El término metadatos son datos sobre los datos en sí. Si mandas un mensaje vía Twitter –es decir, un tweet – donde digas: «Tengo 20 años», los datos sería la cadena que se envió, es decir: «Tengo 20 años». En este caso, los metadatos son datos acerca de los datos –del tweet– como la hora en la que fue enviado, desde dónde se envió el tweet, qué teléfono se usó para enviar el tweet, entre otros.

En términos simples, los metadatos son información estructurada que describe y explica los datos para hacer más simple de localizar, almacenar, usar y manejar un recurso de información. Da a los datos su contexto y significado más allá de los datos en sí.

CAPÍTULO 5

Aprendizaje máquina

El diccionario define aprender de la siguiente manera:

- Obtener conocimiento ya sea por estudio, experiencia o por enseñanza.
- Darse cuenta / ser consciente de la información por medio de la observación.
- Recibir instrucción.
- Memorizar.
- Ser informado.

Cuando se trata de aprendizaje de las computadoras, es decir, el llamado aprendizaje máquina, algunas de estas definiciones son muy difíciles de validar. Por ejemplo: ¿cómo comprobar que la máquina adquirió conocimiento o puede ser consciente de la información que puede «observar»?

El término conciencia en una máquina también llama a un profundo debate. Sin embargo, podemos identificar diferentes tipos de aprendizaje en aplicaciones de inteligencia artificial. En este sentido, los podemos separar en tipos de aprendizaje basados en cómo se crean los modelos de aprendizaje máquina y aprendizaje basado en el estilo de aprendizaje.

En relación a los diferentes estilos de aprendizaje se pueden definir los siguientes:

- Aprendizaje inductivo.
- Aprendizaje analítico o deductivo.
- Aprendizaje evolutivo.
- Aprendizaje conexionista.

En relación a cómo se crean los modelos, podemos definir los tipos de aprendizaje de la siguiente manera:

- Aprendizaje basado en similitud.
- Aprendizaje basado en probabilidad.
- Aprendizaje basado en error.

También, en relación a la información que se conoce y los modelos de entrenamiento de aprendizaje se pueden dividir los tipos de aprendizaje en:

- Aprendizaje supervisado.
- Aprendizaje semiautomático.
- Aprendizaje automático.

En este sentido, se puede separar el aprendizaje en automático –no supervisado– y supervisado, teniendo en cuenta que el semiautomático sería una combinación de ambas, como se muestra en la figura 5.1.

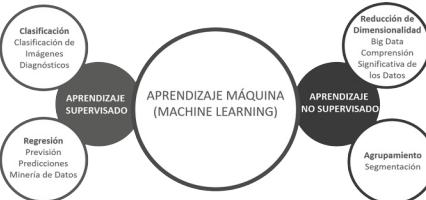


Figura 5.1. Tipos de aprendizaje automático

Además de los conceptos mencionados en el capítulo 3, es importante mencionar algunos términos estándar y parte de la nomenclatura que se utiliza normalmente en aprendizaje máquina –comúnmente llamado *machine learning*–.

En este sentido, podemos revisar los datos. Como se mencionó, los datos están distribuidos usualmente en una base de datos a manera de filas y columnas. Las filas se pueden ver como las instancias (sección 3.7) y las columnas como atributos –usualmente, en aprendizaje máquina, se le conoce también como características, aunque tiene algunas diferencias–. Esto se muestra en la sección 3.8.

Existen otro tipo de datos que de momento no se considerarán como, por ejemplo, imágenes, texto, videos, entre otros.

Cuando se trabaja con algoritmos de aprendizaje máquina, se tienen que identificar también el tipo de dato. Los datos pueden ser categóricos, ordinales, de intervalo o de proporción, y pueden contener cadenas de caracteres, fechas, horas, matrices dentro de cada instancia, videos, imágenes, entre otros. Aunque en la práctica se reducen a real o valores categóricos para trabajar con aprendizaje máquina, es importante conocer qué tipo de datos contiene la base de datos.

Otro punto importante, mismo que será abordado en las secciones de entrenamiento y evaluación del modelo, es elegir tanto el set de entrenamiento como el set de prueba de nuestra base de datos. El set de entrenamiento contiene los datos que van a ser alimentados para definir –entrenar– el modelo, mientras que el set de pruebas contiene los datos con los que se validará la exactitud del modelo y las métricas de error.

Otros conceptos que se deben tener muy en cuenta y que se abordarán en la sección 5.10 es el sobreajuste o sobreaprendizaje y el subajuste –llamado también subaprendizaje–. El sobreajuste se presenta cuando el modelo aprende los datos de entrenamiento de manera muy exacta, es decir, no generaliza bien. Esto resulta en un pobre rendimiento con datos de prueba o de validación, pues, usualmente, el modelo no es capaz de ajustarse a los nuevos datos.

Por otro lado, en el subajuste ocurre lo contrario: el modelo no ha aprendido lo suficiente de los patrones de los datos, ya sea porque el entrenamiento terminó prematuramente, no se utilizó el algoritmo o las configuraciones –parámetros– adecuadas. La consecuencia es que tendrá un bajo rendimiento en todos los datos, incluyendo los de entrenamiento.

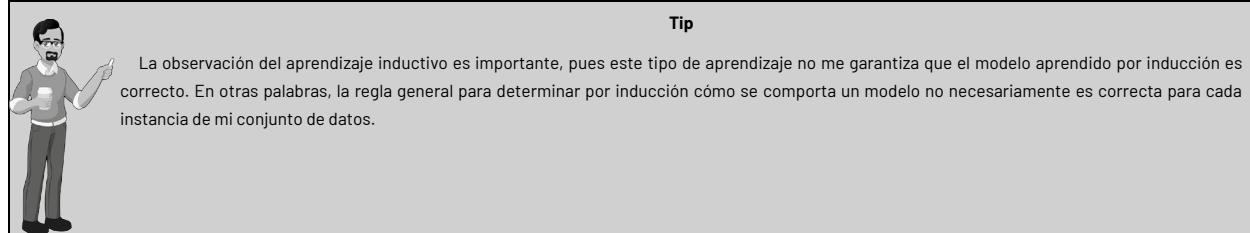
5.1. Aprendizaje inductivo

Se crean modelos de conceptos a partir de generalizar ejemplos simples. Buscamos patrones comunes que expliquen los ejemplos. Se basa en el razonamiento inductivo:

Obtiene conclusiones generales de información específica. Todo el conocimiento obtenido es nuevo, por lo que no preserva el conocimiento –nuevo conocimiento puede invalidar lo obtenido–.

El aprendizaje inductivo es la capacidad de obtener nuevos conceptos, más generales, a partir de ejemplos. Este tipo de aprendizaje conlleva un proceso de generalización/especialización sobre el conjunto de ejemplos de entrada. Los algoritmos implementados son, además, incrementales.

Esta característica permite visualizar el efecto causado por cada uno de los ejemplos de entrada en el proceso de obtención del concepto final. Además de la generalización de conceptos, el programa permite clasificar conjuntos de ejemplos a partir de los conceptos obtenidos anteriormente. De este modo, se puede comprobar, para cada ejemplo de un conjunto dado, a qué clase pertenece dicho ejemplo.



De forma más simple podemos ver el aprendizaje inductivo como el proceso de aprender una función. Por ejemplo: en un par $(x; f(x))$, donde x es la entrada y $f(x)$ la salida. El proceso de inferencia inductiva pura –o inducción– es:

Dada una colección de ejemplos de f , regresar una función h tal que se aproxime a f . A la función h se le llama la hipótesis o modelo.

5.2. Aprendizaje analítico o deductivo

El aprendizaje analítico aplica la deducción para obtener descripciones generales a partir de un ejemplo de concepto y su explicación. Obtiene conocimiento mediante el uso de mecanismos bien establecidos. Nuevo conocimiento no invalida el ya obtenido. Se fundamenta en la lógica matemática.

5.3. Aprendizaje evolutivo

El aprendizaje evolutivo, aunque también se le llama erróneamente genético, aplica algoritmos inspirados en la teoría de la evolución para encontrar descripciones generales a conjuntos de ejemplos. En este sentido, los algoritmos genéticos son evolutivos, pero no todos los algoritmos de aprendizaje evolutivo son algoritmos genéticos. En este texto, no se abordará a fondo acerca del aprendizaje evolutivo.

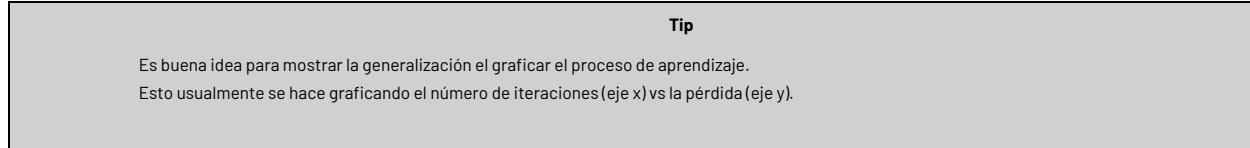
5.4. Generalización

En algunas ocasiones, no se tiene la certeza de que el modelo está aprendiendo de manera correcta. Por ello, es importante revisar cómo está aprendiendo el algoritmo.

Incluso si el algoritmo está inspirado en cómo «aprende» el cerebro humano, en la práctica, funcionan de manera muy diferente. Hacer énfasis en estas diferencias ayuda a entender un poco cómo funcionan internamente los algoritmos de aprendizaje máquina, pues la palabra aprender se puede confundir con que el modelo «entienda», lo cual no es lo mismo.

En este sentido, el objetivo final de un algoritmo de aprendizaje máquina es el de encontrar patrones en los datos que se le dan como modelo de entrenamiento. De ahí que es importante la generalización.

La generalización se observa cuando el algoritmo puede encontrar patrones y hacer predicciones basadas en instancias que el modelo no ha visto –es decir, los datos de prueba–.





Se puede graficar la pérdida de entrenamiento y la pérdida de validación en la misma gráfica para evaluar si el modelo está generalizando de manera correcta.

5.5. Aprendizaje máquina

Para poder explicar algunos tipos de aprendizaje máquina, es conveniente explicar el concepto de aprendizaje máquina y su diferencia con el concepto de inteligencia artificial. Existen muchas definiciones de lo que es la inteligencia artificial, mismas que ya fueron abordadas. El aprendizaje máquina se refiere a la manera en la que un sistema puede generar el conocimiento necesario para que aprenda sin que ello esté explícitamente programado. En otras palabras, no todos los algoritmos de inteligencia artificial son de aprendizaje máquina, pero los algoritmos de aprendizaje máquina están contenidos dentro de la inteligencia artificial.

Se puede definir aprendizaje máquina de muchas formas y hay muchos textos que definen aprendizaje máquina de maneras muy diversas. Una de las definiciones que más me gusta es la siguiente: «Aprendizaje máquina es el campo que trata de cómo se construyen los programas computacionales que automáticamente mejoren con la experiencia».

Esta definición, me gusta por varias razones: en primer lugar, hace referencia a «mejorar automáticamente» y «experiencia». Es una excelente definición que nos indica que no todos los programas pueden mejorar su rendimiento de acuerdo a la cantidad de datos que hayan «observado» y separa esta definición de otros paradigmas informáticos como lo son la programación dinámica, por ejemplo.

A pesar de que el aprendizaje máquina se ligue como parte de las ciencias computacionales, es importante mencionar que, para el diseño, el entrenamiento, las pruebas, la mejora del modelo, cálculo del error, etcétera, se requiere un cierto grado de conocimiento estadístico y matemático. En este libro, no se trabajarán conceptos matemáticos muy complejos para facilitar el diseño y la implementación de los algoritmos que se plantean.

5.5.1. Aprendizaje por refuerzo

El aprendizaje por refuerzo tiene su origen en la psicología del aprendizaje animal. Se basa principalmente en la idea de que se obtendrá una recompensa o «premio» por las acciones correctas, mientras que se obtendrá un «castigo» por las acciones incorrectas. Informalmente, el aprendizaje por refuerzo es definido como un proceso a prueba y error del rendimiento del algoritmo mediante una retroalimentación del evaluador —comúnmente llamado *learner* o evaluador—.

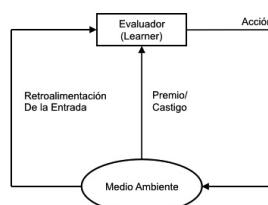


Figura 5.2. Ejemplo de un algoritmo de aprendizaje por refuerzo

Un problema típico de aprendizaje por refuerzo se puede observar en la figura 5.2. El evaluador recibe información del medio ambiente, como una descripción del estado actual del ambiente percibido. Una

acción es ejecutada, después de la cual el evaluador recibe una señal de refuerzo —el «premio» o el «castigo»—. Este refuerzo puede ser una señal positiva o negativa dependiendo de qué tan correcta fue la acción, a un refuerzo negativo corresponde un castigo por una acción incorrecta del evaluador.

5.5.2. Aprendizaje basado en similitud

El aprendizaje basado en similitud en lo que refiere a aprendizaje máquina tiene su origen en la idea de que la mejor manera de realizar una predicción o clasificación es simplemente observar lo que ha funcionado bien anteriormente y predecir el mismo comportamiento de nuevo.

Uno de los conceptos fundamentales de este tipo de aprendizaje es la definición de medida de similitud entre instancias. Muchas veces, esta medida de similitud es realmente alguna medida de distancia. Algunos diferentes ejemplos de las medidas de distancia se explican en la sección 3.11.

Como ejemplo de aprendizaje basado en similitud, uno de los algoritmos más utilizado es el de los vecinos más cercanos —también llamado KNN o «K-Nearest Neighbor»—. La fase de entrenamiento necesaria para construir un algoritmo KNN es relativamente simple y solo se requiere de guardar todas las instancias en memoria. En términos simples, KNN requiere de conocer las distancias más cortas de la instancia —dato de prueba— a predecir, y funciona de la siguiente manera:

Asumamos que tenemos un espacio de características R^2 y que tenemos dos clases como se muestra en la figura 5.3. En dicha figura, se muestran las dos clases, una representada por rombos naranjas (clase A) y la otra representada por triángulos azules (clase B), mientras que la instancia que se quiere clasificar —es decir, si pertenece a la clase A o a la clase B— es representada por el círculo amarillo.

En este ejemplo, si tomamos los 3K vecinos más cercanos, podemos trazar un círculo imaginario a los tres puntos o instancias que están más cerca del círculo. Esto nos lleva a determinar que, por medio de la distancia más cercana, es decir, en este caso la distancia euclidiana, dos instancias corresponden a la clase A, mientras que una pertenece a la clase B. En términos más simples, tengo 2/3 o 66.66 % de acertar que el círculo amarillo pertenece a la clase A, mientras que solo 1/3 o 33.33 % de que el círculo amarillo pertenezca a los triángulos, es decir a la clase B.

Sin embargo, si se aumenta el número de K, por ejemplo, a 7K vecinos, como se muestra también en la figura 5.3, el círculo que encerraría las instancias más cercanas al círculo abarcaría cuatro triángulos y tres rombos, por lo que mi clasificación cambiaría de clase A a clase B. A pesar de que KNN es muy utilizado y funciona bien en muchos casos para problemas de clasificación, es importante determinar el número óptimo de K vecinos.

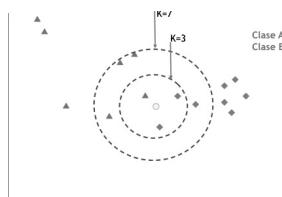


Figura 5.3. Ejemplo de algoritmo de vecinos más cercanos K-NN

Este tipo de algoritmos basados en similitud son muy útiles para clasificación. K-NN da buenos resultados principalmente cuando se trata de clasificar un número pequeño de clases con pocos atributos. Cuando se trabaja con muchos atributos, es importante determinar el peso de cada atributo en el resultado para poder determinar cuáles usar o, en su defecto, reducir la dimensionalidad de los datos.

Para poder realizar el entrenamiento se requiere tener claro cuál es el valor óptimo de K —número de vecinos óptimo—. Esto se puede hacer mediante prueba y error o mediante algunos métodos como el

método del codo —comúnmente conocido como *elbow method*—. Cuando se sabe el K óptimo, el entrenamiento es relativamente simple, pues deberá de guardar en memoria todas las distancias del punto que se quiera conocer su clase a todas las demás instancias con las que se construyó el modelo, después ordenarlas y tomar las menores distancias euclidianas con respecto al número K óptimo. Esto se repite para cada dato de prueba con el que se quiera conocer su clase.

Pseudocódigo Algoritmo KNN



```

Función KNN (X,Y,x)
//X:datos_entrenamiento, Y: Clase_de_x, x:dato_a_clasificar
{
    for i desde 1 a m
        Calcular la distancia d(Xi, x)
    end for
    Calcular set I contenido índices para las distancias más cortas d(Xi, x)
    Return la etiqueta más clases haya contenido para Yi [donde i pertence a I]
}

```

5.5.5. Aprendizaje basado en probabilidad

Existen algoritmos de aprendizaje basados en predicciones probabilísticas, las cuales se basan en la teoría de probabilidad de las ciencias computacionales.

En esta sección se asume que el lector tiene conocimientos básicos de teoría de probabilidad, incluyendo los fundamentos para calcular probabilidades basados en frecuencias relativas, calcular probabilidades condicionales, regla del producto y de la cadena, entre otras.

Para exemplificar este tipo de aprendizaje, se tomará el ejemplo del apéndice A.6 «Síntomas de meningitis». En este ejemplo, se tienen los siguientes atributos: ID, dolor de cabeza, fiebre, vómito, meningitis, los cuales, salvo ID, son atributos categóricos dicotómicos —es decir, que solo toman dos valores —y cuyas observaciones son solo sí y no. El último atributo es el atributo de decisión e indica si la persona de acuerdo a los síntomas tiene meningitis o no.

ID	Dolor de Cabeza	Fiebre	Vómito	Meningitis
1	Sí	Sí	No	No
2	No	Sí	No	No
3	Sí	No	Sí	No
4	Sí	No	Sí	No
5	No	Sí	No	Sí
6	Sí	No	Sí	No
7	Sí	No	Sí	No
8	Sí	No	Sí	Sí
9	No	Sí	No	No
10	Sí	No	Sí	Sí

En este ejemplo, desde el punto de vista probabilístico, cada característica de los datos sería una variable aleatoria, mientras que el espacio de muestreo del dominio asociado con el problema de predicción es el conjunto de todas las posibles combinaciones de valores. Cada fila —instancia— representa un experimento, el cual es asociado con una característica descriptiva. El conjunto de los posibles valores que puede tomar ese experimento sería un evento.

De esta forma, una función de probabilidad $P()$ es la probabilidad de un evento. Por ejemplo, $P(\text{Fiebre} = \text{Sí})$ toma la probabilidad de que el atributo fiebre tenga la observación sí, que en este caso es de 0.4 o 40 % si se calcula manualmente de la base de datos. Las funciones de probabilidad para atributos categóricos se conoce como **funciones de probabilidad de masa**, mientras que las funciones de probabilidad para atributos continuos se conocen como **funciones de probabilidad de densidad**.

Por otro lado, una **probabilidad conjunta** se refiere a la probabilidad que adquiere un valor específico para varias características al mismo tiempo. Por ejemplo, $P(\text{Meningitis} = \text{Sí}, \text{dolor de cabeza} = \text{Sí}) = 0.2$.

Por último, una **probabilidad condicional** se refiere a la probabilidad de una característica tomando un valor específico siendo que ya se sabe la probabilidad de una característica diferente. Por ejemplo, $P(\text{Meningitis} = \text{Sí}, \text{dolor de cabeza} = \text{Sí}) = 0.2857$.

En general, es útil conocer las probabilidades para todas las posibles observaciones de una característica. Para esto, se utiliza la **distribución de probabilidad**. La distribución de probabilidad es una estructura que describe la probabilidad de cada posible valor que una característica puede tener. Por ejemplo, para la base de datos de meningitis se escribiría $P(\text{Meningitis}) = <0.3, 0.7>$ siendo 0.3 la probabilidad verdadera y 0.7 la probabilidad falsa. Es importante mencionar también que la distribución de probabilidad debe ser igual a 1.

Teniendo la probabilidad de distribución conjunta, se puede calcular la probabilidad de cada evento sumando todas las probabilidades de distribución individuales.

5.5.6. Aprendizaje basado en error

El aprendizaje basado en error consiste en minimizar el error total con respecto al entrenamiento del modelo. Los métodos más utilizados de aprendizaje basado en error son los métodos de regresión. Para ejemplificar este tipo de aprendizaje, se tomará en cuenta la regresión lineal y polinomial y un extracto de los datos de partículas contaminantes PM10, mencionados en el apéndice A.7.

La regresión lineal se puede expresar de la siguiente forma:

$$y = f(x) = \beta_0 + \beta_1 x \quad (5.1)$$

Donde β_0, β_1 son los coeficientes del modelo lineal.

También se puede establecer una relación no lineal con el fin de tener un modelo que asemeje mejor el comportamiento de los datos, a esto lo conocemos como regresión polinomial, y se expresa de la siguiente forma:

$$f(x) = \beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3 + \dots + \beta_n x^n \quad (5.2)$$

¿Qué grado de polinomio desea?

1

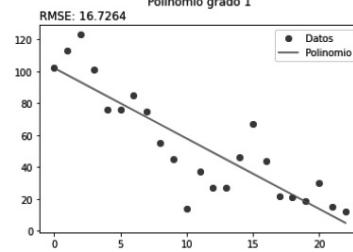


Figura 5.4. Ejemplo de regresión lineal

La figura 5.4 muestra un ejemplo de regresión lineal mostrando la raíz del error mínimo cuadrático (RMSE), el cual se calcula mediante la ecuación:

$$RMSE = \sqrt{\frac{\sum_{n=1}^N (\hat{r}_n - r_n)^2}{N}} \quad (5.3)$$

Donde \hat{r}_n es el valor predicho por el modelo, r_n es el valor observado y N es el número de observaciones que se tienen.

En la figura 5.4, se muestra la relación lineal entre los datos y la contaminación. Esta relación no es perfecta, aunque no requiere serlo. Si el modelo se ajusta de manera muy cercana a los datos, podría significar una mala generalización, ya que es muy probable que el modelo no se pueda ajustar a datos nuevos, como se ha comentado anteriormente. En este caso, la raíz de la suma de los errores cuadráticos es de 16.7264.

En este caso, también se podría utilizar la simple ecuación de la pendiente, la cual puede ser escrita como se muestra en la ecuación 5.4:

$$y = mx + b \quad (5.4)$$

Donde m es la inclinación de la línea. De esta manera simple, se puede explicar el aprendizaje basado en error, donde la línea predice un y valor por cada valor x de entrada y donde cada valor x intercepta la predicción y , el error sería 0.

Si el grado del polinomio en el mismo caso fuera tres, quedaría el modelo como se muestra en la figura 5.5.

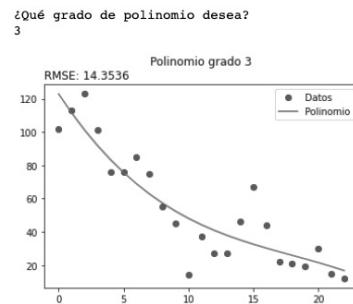


Figura 5.5. Ejemplo de regresión polinomial de grado 3

En la figura 5.5, se muestra un polinomio de grado 3. Si se calcula el error utilizando la ecuación de RMSE (ecuación 5.3) sería de 14.3536, el cual es menor que el error con la regresión lineal.



Se podría pensar que a mayor grado el polinomio, menor el error; sin embargo, esto tiene un límite. Si se intenta un polinomio de un grado mucho mayor, se podría obtener un menor error, pero se estaría sobre aprendiendo. Se recomienda un buen balance entre el grado del polinomio y el error.

5.5.7. Aprendizaje Supervisado

El aprendizaje supervisado es una técnica en la que, teniendo la etiqueta o atributo de salida de cada instancia, se sabe la clase a la que pertenece. El éxito de la implementación en este tipo de aprendizaje depende en gran medida de la cantidad de instancias independientes —que no participaron en la clasificación inicial o entrenamiento— con respecto al modelo creado en el entrenamiento.

La tasa de éxito en datos de ejemplo da una medida objetiva de qué tan bien se ha entrenado el modelo. Para muchas aplicaciones de inteligencia artificial, el rendimiento del algoritmo es medido más subjetivamente en términos de qué tan aceptable es la descripción del aprendizaje con respecto a errores humanos, como, por ejemplo, en las reglas o los árboles de decisión.

La figura 5.6 muestra la diferencia entre los tipos de aprendizaje de acuerdo al etiquetado de los datos.

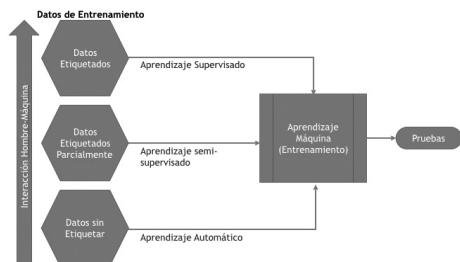


Figura 5.6. Tipos de aprendizaje por interacción hombre-máquina

Especificamente, el objetivo del aprendizaje supervisado es procesar un conjunto de datos cuyos atributos se han dividido en dos grupos y derivan de una relación entre los valores de uno y los valores del otro. Estos dos grupos se denominan a veces predictores y objetivos, respectivamente. En terminología estadística, se denominan variables independientes y dependientes, respectivamente. El aprendizaje es «supervisado» porque la distinción entre los predictores y las variables objetivo es conocida y escogida por un proceso de minería de datos o de manera manual.

5.5.8. Aprendizaje automático

El objetivo del aprendizaje no supervisado –automático– es que una computadora procese un conjunto de datos y extraiga una estructura que sea relevante para el problema en cuestión. Esto puede tomar diferentes formas. En el análisis de conglomerados y algunas formas de estimación de la densidad, el objetivo es encontrar un pequeño número de «grupos» de observaciones que tengan una estructura simple. Los modelos gráficos y otras formas de dependencia reducen las interrelaciones entre las variables a un número más pequeño. En las redes neuronales, el aprendizaje no supervisado es una forma de análisis de conglomerados o estimación de la densidad en la que la red está siendo entrenada para reproducir la información.

5.6. Metodología propuesta para el desarrollo de algoritmos de IA

Existen muchas metodologías para el desarrollo e implementación de algoritmos de inteligencia artificial, en este caso de aprendizaje máquina. Durante el desarrollo de los códigos del presente libro, utilizaremos la siguiente en orden secuencial: 1. Recolectar. 2. Preparar. 3. Analizar. 4. Entrenar. 5. Probar. 6. Usar. Es importante mencionar que algunos pasos como el 3 y el 4 son iterativos, es decir, si las pruebas no son satisfactorias, se volverá a entrenar el algoritmo para posteriormente probarlo.

5.6.1. Recolectar

En este paso, se debe tener claro qué datos recolectar y cómo. Esto lo podemos poner en esta etapa, ya que algunas bases de datos ya se encuentran disponibles y, en ese caso, la primera etapa sería el preprocesamiento de estos.

En caso contrario, se tendrá que comenzar por definir los datos con los cuales se va a trabajar el algoritmo de aprendizaje máquina.

Tip



Hay que tomar en cuenta varias cosas cuando se requiere capturar datos de la base de datos, por ejemplo:

- ¿Qué tipo de datos se van a capturar?
- ¿Cómo se van a guardar los datos? Es decir, en un archivo de texto, como una base de datos, un csv o algún otro formato.
- Si se van a guardar como una base de datos, ¿cómo se van a procesar? Por RDBMS, NoSQL.
- Se requiere guardar o se puede procesar en línea?

5.6.2. Preparar

La preparación se refiere principalmente a cómo se van a manejar o preprocesar los datos que se tienen. En cuestión de la preparación de los datos, se deben tener en cuenta varios factores. Los más importantes son:

- Limpieza de datos e imputación.
- Filtrado.
- Escalamiento.
- Valores inconsistentes.

En cuestión de la limpieza de los datos, uno de los mayores problemas, que ya se han abordado en este libro, es el de los valores faltantes. Existen varias maneras de lidar con los valores faltantes, entre ellas se pueden mencionar las siguientes:

- No hacer nada. Esto no es del todo recomendable, aunque existen algunos algoritmos de aprendizaje máquina que pueden lidar correctamente si un atributo tiene valores faltantes.
- Técnicas de imputación: como se mencionó anteriormente, una de las formas más utilizadas para lidar con valores faltantes son los métodos de imputación.
- Eliminar la instancia completa. Esto solo se recomienda cuando existan muchos datos de esa instancia que no están presentes. Como regla general, más del 50 % de datos faltantes de esa instancia.
- Eliminar el atributo completo. Esto tampoco es muy recomendable. Existen muchos métodos de imputación que se pueden utilizar cuando hay algunos datos faltantes en un atributo. Sin embargo, si faltan el 60 % o más de datos de ese atributo, es muy probable que la imputación no consiga ser útil y es más recomendable eliminar el atributo completo o no utilizarlo para el entrenamiento posterior.
- En la medida de lo posible, se puede analizar si existen muchos datos faltantes al volver a capturar o realizar la adquisición de los datos que no están presentes. Esto no siempre es posible, pero vale la pena determinar si se pueden llenar las instancias que no contienen datos mediante una readquisición de estos.



Tip

No existe una única regla para saber qué método de imputación utilizar.

Lo que se recomienda es evaluar los datos e implementar el que se observe más adecuado.

En ocasiones una imputación por media o media por clases es suficiente, mientras que otras veces se requieren métodos más avanzados como por cadenas múltiples (MICE) o por cadenas de Markov.

Además, la limpieza de datos no solo incluye los datos faltantes. En muchas ocasiones, se requiere hacer ingeniería de características. La ingeniería de características incluye algunas de las técnicas ya mencionadas, aunado a buscar instancias duplicadas, buscar altas correlaciones entre atributos, entre otras. Esto es útil para evitar duplicidades de atributos o instancias que pueden afectar la distribución de los datos y sesgar la capacidad predictiva del algoritmo en etapas posteriores.

En cuestión del escalamiento –también llamado normalización–, es importante mencionar que la mayoría de las bases de datos contienen características muy variadas en cuestión de magnitudes. Esta es una de las razones por las cuales se recomienda revisar con anterioridad el tipo de datos que maneja, mínimo, máximo, media –o moda, según el tipo de dato–, desviación estándar y varianza para cada atributo a la vez que revisar si se tienen datos atípicos, realizar un histograma de la distribución por cada atributo de la base de datos. Cuando se tienen diferentes magnitudes en cada atributo, es útil realizar un escalamiento, pues la falta de normalización puede afectar en muchas ocasiones el rendimiento de un algoritmo.

Los más comunes métodos de escalamiento son:

- Reescalamiento –normalización min-max–.
- Normalización por media.
- Escalamiento U (o U-length).
- Z-score –estandarización–.

En cuestión de los valores inconsistentes, más comunes en las bases de datos de los que se cree, las inconsistencias se pueden dar por numerosas razones, por ejemplo:

Los datos pueden estar distribuidos en diferentes formatos. Por ejemplo, para poder poner sexo, se puede poner como M, H, masculino, femenino, etcétera. Esta inconsistencia genera más observaciones de las que deberían de ser –dos en este caso–. Como humanos, podemos llegar fácilmente a una clasificación correcta de las posibles respuestas que se puedan dar, sin embargo, los algoritmos de aprendizaje máquina tienen que ser entrenados para poder obtener estas conclusiones. Lo mejor en todo caso es distribuir de manera homogénea y reducir la redundancia en las observaciones para cada atributo.

5.6.3. Analizar

Además de las inconsistencias de los formatos y la preparación de estos, es recomendable realizar cierto análisis con la base de datos. Por ejemplo, es recomendable analizar los datos con los que se va a comenzar el entrenamiento (secciones 5.6.4 «Entrenar» y 5.9 «Entrenamiento del modelo») y con los que se va a realizar las pruebas (sección 5.6.5 «Probar» y 5.10 «Evaluación del modelo»). Existen diferentes métodos para realizar esto, pero es importante tenerlo en cuenta. En ocasiones, se separan los datos en dos: entrenamiento y pruebas, mientras que a veces se separa en tres: entrenamiento, validación y pruebas.

De acuerdo al número de instancias, es recomendable determinar si se puede hacer un simple 80-20 para entrenamiento –80 % de los datos para realizar el entrenamiento del modelo– y el 20 % restante para pruebas o se puede implementar 70-30. Si se requiere de validar el modelo se puede optar por una distribución 80-10-10.

Otro método para realizar las pruebas es el método conocido como validación cruzada o *k-fold*. A pesar de que en la etapa de probar es donde se implementa, es necesario que en la etapa de análisis se defina cómo se va a probar el modelo.

La validación cruzada (*k-fold*) es un procedimiento muy utilizado en aprendizaje máquina en donde se realiza una agrupación de acuerdo al número de *k* en el que se van a separar para poder realizar las pruebas. Por ejemplo, si *k* = 5, se hace una partición de cinco y se toma cada 20 % de los datos para prueba de manera secuencial. Este procedimiento en este caso se haría cinco veces.

Tip

No se recomienda cuando se realiza «*k-fold*» ordenar los datos de acuerdo a ningún atributo. Ya que se toman para hacer los particiones de manera ordenada y se segmenta en *k* grupos, algunas pruebas pueden determinar que el algoritmo es muy bueno, y otras mostrar una exactitud muy baja.

Los valores más usados para *k* son 3, 5 y 10, se recomienda usar alguno de esos valores cuando se utilice este método.



También es recomendable analizar con qué tipos de datos se cuenta para poder elegir el tipo de algoritmo más adecuado y su implementación. Para esta etapa, parte del análisis de la distribución de los datos ya estará hecha, pues la preparación incluye determinar valores faltantes y, si se realiza imputación, ya se tendrán datos sobre distribución, desviación estándar, varianza, entre otros, lo que es muy útil en esta etapa de la metodología propuesta.

5.6.4. Entrenar

Una vez que los datos de entrenamiento y de prueba estén separados, es momento de entrenar el modelo. Para esto, como su nombre lo indica, solamente se van a utilizar los datos de entrenamiento, los de prueba no se van a tocar hasta la siguiente etapa.

El entrenamiento consiste en alimentar al modelo con los datos de entrada y evaluar la salida, correlacionando la salida con la salida esperada. Este proceso es iterativo hasta que la diferencia entre la salida obtenida y la esperada es igual —o lo más cercana a ser igual—. El resultado de esta correlación, usualmente, se utiliza para modificar el modelo.

La evaluación del modelo se abordará más a fondo en la sección 5.10.

5.6.5. Probar

Existen muchas maneras de poder medir el rendimiento de los algoritmos de aprendizaje máquina. Al llegar a esta etapa, ya se debe tener una idea clara de cómo se harán las pruebas y los datos ya se debieron de haber segregado, es decir, separado. También es importante tomar en cuenta que, hasta esta etapa, los datos que se separaron para pruebas no se han usado, de lo contrario, los valores de rendimiento del modelo no serían válidos.

Existen diferentes métricas para evaluar el rendimiento del algoritmo, algunas de ellas son:

- Tasa de clasificación.
- Accuracy (exactitud).
- Precisión.
- Sensibilidad (*recall*).
- Especificidad.
- Puntaje F-beta.

Las pruebas del modelo se abordarán más a fondo en la sección 5.10.

5.6.6. Usar

En esta sección se deben de evaluar diversos factores. Primeramente, llegados a esta etapa, el algoritmo está listo para ser utilizado y realizar predicciones. Es importante mencionar en esta etapa si es un API, parte de una plataforma, si se tiene que correr desde el framework con que se creó, con qué parámetros de entrada —argumentos—, etcétera. Algunos de los puntos a considerar son:

- ¿El sistema es capaz de realizar predicciones en tiempo real o fuera de línea?
- Si el sistema hace predicciones en línea, ¿cuánto tiempo tarda en realizar la predicción? Esta respuesta puede variar desde milisegundos hasta horas.
- ¿Es adecuado el tiempo que toma el algoritmo para generar una respuesta de salida?
- ¿Qué tan frecuentemente se requiere que los modelos se actualicen?
- ¿Cuál es el tamaño máximo de instancias y atributos con los que puede trabajar el modelo?
- ¿Existen algunas restricciones que se tienen que tomar en cuenta para ejecutar el modelo?
- ¿Cuáles son los argumentos que se deben de definir para ejecutar el modelo?
- Se deben mencionar —y en su caso guardar— los metadatos relevantes como lo son: configuración, dependencias, fechas y todo lo que se necesita. Esto es especialmente importante cuando se trabaja con algoritmos en ambientes altamente regulados —como sistemas bancarios o bursátiles, por ejemplo—.
- Si el modelo o el algoritmo se tiene que actualizar constantemente, ¿cuál es el plan de actualización que se seguirá?
- ¿Cuál es el plan de reentrenamiento en caso de que los resultados ya no sean los esperados?

5.7. Aprendizaje por árboles de decisión

Los árboles de decisión son de los algoritmos de aprendizaje máquina más usados en la historia. Son usados tanto para problemas de clasificación como para regresión. Una de las preguntas primordiales es la razón por la cual usar árboles de decisión. La respuesta es simple: tiene varias ventajas con respecto a otros métodos de aprendizaje máquina, como son:

- La manera de interpretar la información es simple, y hacer sentido y entender los resultados es relativamente fácil.
- Los árboles de decisión se entrena con las instancias de ejemplos ya vistos y son proclives a predecir circunstancias no previstas anteriormente.
- La lógica de interpretación de los datos es clara, al contrario de algunos algoritmos, que ven los parámetros como cajas negras —como redes neuronales, máquinas de soporte de vectores, etcétera—.

La construcción de un árbol de decisión es relativamente simple: cada nodo representa una característica —atributo—, cada rama representa una decisión —regla— y cada hoja representa una salida —categórica o un valor continuo—.

Existen varios algoritmos para poder construir un árbol de decisión. Dos de los más utilizados son CART (árboles de clasificación y regresión) e ID3 (dicotomizador iterativo 3). En este libro, se abordará el algoritmo ID3 con un ejemplo práctico paso a paso.



Ejemplo

Uno de los ejemplos más usados para explicar la creación de árboles de decisión es el de la condición climática. El árbol se construye con respecto al clima, la temperatura, la humedad y el viento, con el atributo de decisión si de acuerdo a las condiciones es conveniente salir a la intemperie o no. La explicación de la base de datos de las 14 instancias se encuentra en el apéndice A.5.

En el ejemplo, se tienen catorce días con diferentes condiciones climáticas y un atributo de decisión de salir binario —dicotómico— cuya única respuesta es ya sea «No» o «Sí», como se muestra en la tabla 5.1.

Día	Clima	Temperatura	Humedad	Viento	¿Salir?
1	Soleado	Cálido	Alta	Débil	No
2	Soleado	Cálido	Alta	Fuerte	No
3	Nublado	Cálido	Alta	Débil	Sí
4	Lluvioso	Templado	Alta	Débil	Sí
5	Lluvioso	Frío	Normal	Débil	Sí
6	Lluvioso	Frío	Normal	Fuerte	No
7	Nublado	Frío	Normal	Fuerte	Sí
8	Soleado	Templado	Alta	Débil	No
9	Soleado	Frío	Normal	Débil	Sí
10	Lluvioso	Templado	Normal	Débil	Sí
11	Soleado	Templado	Normal	Fuerte	Sí
12	Nublado	Templado	Alta	Fuerte	Sí
13	Nublado	Cálido	Normal	Débil	Sí
14	Lluvioso	Templado	Alta	Fuerte	No

Tabla 5.1. Base de datos para caso de estudio de condiciones climáticas

Los datos presentados en la tabla 5.1. presentan los siguientes atributos:

- **Día.** Es un atributo ordinal, representa los días del 1 al 14.
- **Clima.** Es un atributo nominal con tres diferentes observaciones, las cuales son: soleado, nublado y lluvioso.
- **Temperatura.** También es un atributo nominal con tres diferentes observaciones, las cuales son: frío, templado y cálido.
- **Humedad.** Atributo nominal con dos observaciones: alta o normal.
- **Viento.** Atributo nominal binario. Sus dos observaciones son: débil y fuerte.
- **Salir.** Atributo nominal de decisión. Determina si se puede salir o no y sus observaciones son: sí y no.

Para poder implementar el algoritmo ID3, se tienen que encontrar primero la entropía del sistema –en este caso, la base de datos de catorce instancias– y la ganancia de información para cada atributo.

Primeramente, para crear el árbol con ID3, se necesita determinar la entropía total, misma que se calcula mediante la siguiente ecuación:

$$\text{Entropía}(S) = - \sum p(I) \cdot \log_2 p(I) \quad (5.5)$$

En este caso, el atributo de decisión es el atributo salir que tiene dos respuestas: sí y no. De catorce instancias totales en este ejemplo se tienen nueve (9/14) cuya observación es sí y cinco (5/14) cuya observación es no.

Por lo que la ecuación 5.5 quedaría para determinar la entropía de la siguiente manera:

$$\text{Entropía}(\text{Decisión}) = - p(\text{Sí}) \cdot \log_2 p(\text{Sí}) - p(\text{No}) \cdot \log_2 p(\text{No}) \quad (5.6)$$

$$\text{Entropía}(\text{Decisión}) = -(9/14) \cdot \log_2(9/14) - (5/14) \cdot \log_2(5/14) = 0 \quad (5.7)$$

Una vez que se tiene la entropía del sistema, se tiene que definir el nodo raíz de los atributos que se tienen. En este caso, se tiene que escoger si clima, temperatura, humedad o viento podría ser escogido como su nodo principal. Para poder realizar esto, se tienen que encontrar el atributo que mejor clasifica los datos de entrenamiento, es decir, el atributo que tenga un mayor valor de ganancia de información.

Se calcula la ganancia de cada atributo mediante la siguiente ecuación:

$$\text{Ganancia}(S, A) = \text{Entropía}(S) - \sum [p(S|A) \cdot \text{Entropía}(S|A)] \quad (5.8)$$

Si comenzamos, por ejemplo, con el atributo viento, tenemos dos observaciones posibles: débil y fuerte, por lo que la ganancia para el atributo de viento se calcularía de la siguiente manera:

$$\begin{aligned} \text{Ganancia}(\text{Decisión}, \text{Viento}) &= \text{Entropía}(\text{Decisión}) - [p(\text{Decisión}|Viento = \text{Débil}) \cdot \text{Entropía}(\text{Decisión}|Viento = \text{Débil}) + p(\text{Decisión}|Viento = \text{Fuerte}) \cdot \text{Entropía}(\text{Decisión}|Viento = \text{Fuerte})] \\ &= (5.9) \end{aligned}$$

Existen en el ejemplo ocho instancias en donde el atributo viento es débil –instancias 1, 3, 4, 5, 8, 9, 10, 13–, de las cuales, en dos –instancias 1 y 8–, el atributo salir da como una observación «no» y las restantes seis dan como atributo de decisión una observación «sí». Por lo que la entropía del tributo viento débil se calcularía de la siguiente manera:

$$\text{Entropía}(\text{Decisión}|\text{Viento} = \text{Débil}) = - p(\text{No}) \cdot \log_2 p(\text{No}) - p(\text{Sí}) \cdot \log_2 p(\text{Sí}) \quad (5.10)$$

$$\text{Entropía}(\text{Decisión}|\text{Viento} = \text{Débil}) = -(2/8) \cdot \log_2(2/8) - (6/8) \cdot \log_2(6/8) = 0.811 \quad (5.11)$$

En cuestión de viento = fuerte se tienen seis instancias, donde se divide tres con la observación del atributo de decisión no y tres con sí.

Si hacemos el mismo procedimiento como se mostró en la ecuación 5.10 para viento = fuerte, el resultado sería el siguiente:

$$\text{Entropía}(\text{Decisión}|\text{Viento} = \text{Fuerte}) = -(3/6) \cdot \log_2(3/6) - (3/6) \cdot \log_2(3/6) = 1 \quad (5.12)$$

Con ambas entropías –una para cada observación– se calcula la ganancia de información de la siguiente manera:

$$\begin{aligned} \text{Ganancia}(\text{Decisión}, \text{Viento}) &= \text{Entropía}(\text{Decisión}) - [p(\text{Decisión}|\text{Viento} = \text{Débil}) \cdot \text{Entropía}(\text{Decisión}|\text{Viento} = \text{Débil}) + p(\text{Decisión}|\text{Viento} = \text{Fuerte}) \cdot \text{Entropía}(\text{Decisión}|\text{Viento} = \text{Fuerte})] \\ &= 0.940 - [(8/14) \cdot 0.811] - [(6/14) \cdot 1] = 0.048 \end{aligned} \quad (5.13)$$

Se realiza la misma operación con los otros atributos quedando de la siguiente manera:

- Ganancia(Decisión, Viento) = 0.048
- **Ganancia(Decisión, Clima) = 0.246**
- Ganancia(Decisión, Temperatura) = 0.029
- Ganancia(Decisión, Humedad) = 0.151

Como puede observarse, el atributo clima es el que tienen una ganancia de información mayor, por lo que queda como nodo raíz el atributo clima con sus tres observaciones como se muestra en la figura 5.7.

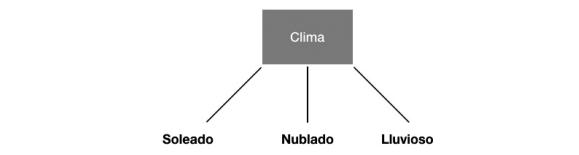


Figura 5.7. Nodo raíz para un árbol de decisión con el algoritmo ID3

El siguiente paso es revisar el atributo de decisión del nodo raíz con respecto a las tres observaciones de clima –soleado, nublado y lluvioso–.

Con lo que respecta a nublado, se tienen cuatro instancias y en toda la observación del atributo salir es sí, por lo que, bajo esas condiciones, según la tabla 5.2, se podría salir a la intemperie.

Día	Clima	Temperatura	Humedad	Viento	¿Salir?
3	Nublado	Cálido	Alta	Débil	Sí
7	Nublado	Friό	Normal	Fuerte	Sí
12	Nublado	Templado	Alta	Fuerte	Sí
13	Nublado	Cálido	Normal	Débil	Sí

Tabla 5.2. Atributos para observación de clima nublado

Con lo que respecta al clima soleado, se puede ver en la tabla 5.2 que se puede salir 2/5 veces, pero 3/5 no, por lo que se tendría que realizar otro procedimiento de cálculo de ganancia para saber qué otro atributo determina el salir con el clima soleado.

Día	Clima	Temperatura	Humedad	Viento	¿Salir?
1	Soleado	Cálido	Alta	Débil	No
2	Soleado	Cálido	Alta	Fuerte	No
8	Soleado	Templado	Alta	Débil	No
9	Soleado	Frió	Normal	Débil	Sí
11	Soleado	Templado	Normal	Fuerte	Sí

Tabla 5.3. Atributos para observación de clima soleado

Realizando los cálculos de la ganancia de información como se mostraron anteriormente, se necesita saber qué factor pesa más teniendo un clima soleado –la temperatura, la humedad o el viento–, obteniendo lo siguiente:

- Ganacia (Clima = Soleado|Temperatura) = 0.570
- **Ganacia (Clima = Soleado|Humedad) = 0.970**
- Ganacia (Clima = Soleado|Viento) = 0.019

Por lo que el factor humedad es el que más contribuye en la decisión del atributo salir. Si se ve la tabla 5.3, si la humedad es alta, aunque el clima sea soleado, la decisión de salir siempre será no, mientras que, si la humedad es normal, la decisión de salir siempre será sí. Hasta este momento, el árbol de decisión con ID3 quedaría como se muestra en la figura 5.8.

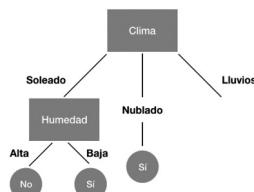


Figura 5.8. Árbol de decisión con el algoritmo ID3, decisión clima soleado y nublado.

Por último, se requiere realizar la decisión de salir o no con el clima lluvioso, como se muestra en la tabla 5.4.

Día	Clima	Temperatura	Humedad	Viento	¿Salir?
4	Lluvioso	Templado	Alta	Débil	Sí
5	Lluvioso	Frió	Normal	Débil	Sí
6	Lluvioso	Frió	Normal	Fuerte	No
10	Lluvioso	Templado	Normal	Débil	Sí
14	Lluvioso	Templado	Alta	Fuerte	No

Tabla 5.4. Atributos para observación de clima lluvioso

Se calcula de la misma forma la ganancia de los atributos de temperatura, humedad y viento tomando en cuenta el clima lluvioso, como se mostró anteriormente, teniendo una mayor ganancia el atributo viento. Es decir, con clima lluvioso y viento fuerte, la decisión de salir es no, mientras que, con viento débil, la decisión es sí, por lo que el árbol de decisión usando ID3 quedaría terminado como se muestra en la figura 5.9.

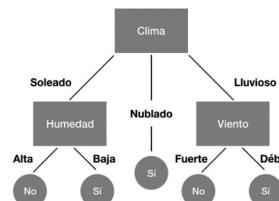


Figura 5.9. Árbol de decisión con el algoritmo ID3 terminado

A pesar de que los árboles de decisión son útiles para diversos problemas de clasificación, muestran las siguientes debilidades:

- Los árboles de decisión son menos apropiados para tareas de estimación donde el atributo de decisión es de tipo continuo.

- Son proclives a cometer errores cuando se tienen pocas instancias y muchos atributos con muchas observaciones cada uno.

5.8. Bosques aleatorios

Los bosques aleatorios son un tipo de algoritmo de ensamble muy popular y efectivo.

Es ampliamente usado para problemas de clasificación y regresión tanto para datos categóricos como para datos continuos –en serie de tiempo– aunque para estos últimos requiere ser transformado en un problema de aprendizaje supervisado.

En palabras más simples, los bosques aleatorios construyen múltiples árboles de decisión y los mezclan para obtener resultados más precisos y estables, como se muestra en la figura 5.10. Se muestra un ejercicio de bosques aleatorios en la sección 8.17.

Como se puede observar en la figura 5.10, los árboles de decisión se construyen a partir de un subset aleatorio de características. Es esta diversidad la que generalmente resulta en un mejor modelo.

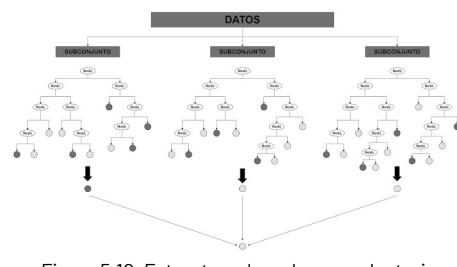


Figura 5.10. Estructura de un bosque aleatorio

Ejemplo

Para exemplificar los bosques aleatorios, tomemos como ejemplo la salida a algún restaurante. En este caso, puedo preguntar a conocidos por recomendaciones de restaurantes. Cada persona me dará sus recomendaciones basadas en su propia experiencia. Con esa información, y mis preferencias personales, puedo tomar la decisión de en qué restaurante comer.



Otra de las ventajas de los bosques aleatorios es que con esta técnica es fácil medir la importancia relativa que cada característica tiene en la predicción. También, los bosques aleatorios no tienden a sobreaprender, que es uno de los principales problemas que tienen los algoritmos de aprendizaje máquina. La razón de esto es debido a que, al construir un set diverso de árboles de decisión de manera aleatoria, los nodos se separan de manera sucesiva y se eligen los mejores, por lo que reduce la entropía del modelo y, por lo tanto, la varianza entre modelos. Con esto, se puede obtener un modelo robusto sin sobreentrenarlo.

A pesar de que se puede pensar en los bosques aleatorios como una colección de árboles de decisión, existen diferencias fundamentales entre ambos.

Por ejemplo, si se tiene un set de datos etiquetados de entrenamiento en un árbol de decisión, el árbol formulará un set de reglas, las cuales pueden ser usadas para realizar las predicciones, mientras que, en este caso, un bosque aleatorio seleccionará de manera aleatoria las observaciones y las características para construir los árboles de decisión y después promediará los resultados.

Los hiperparámetros a tomar en cuenta en un bosque aleatorio son los siguientes:

- **Número de estimadores.** Este hiperparámetro es el número de árboles de decisión que puede construir el algoritmo antes de realizar la predicción final. En general, al incrementar el número de estimadores,

puede mejorar hasta un cierto punto la predicción, aunque el costo computacional se eleva gradualmente también.

- **Número máximo de características.** Este hiperparámetro es el número máximo de características que el bosque aleatorio toma en cuenta para separar un nodo.
- **Número mínimo de hojas.** Este hiperparámetro también es importante. Determina el número mínimo de hojas requerido para separar un nodo interno. Un número muy pequeño puede resultar en un modelo inestable, mientras que un número muy grande repercutirá en el tiempo de cómputo y no necesariamente en un mejor modelo.
- **Número de trabajos.** Define cuántos procesadores puede usar a un tiempo. Usualmente, el valor -1 indica que no hay límite y el algoritmo tratará de optimizar el número de trabajos que puede realizar en los procesadores.
- **Estado aleatorio.** Es la medida de aleatoriedad del bosque. Si se define un valor, el modelo producirá un mismo resultado cada vez, por lo que el algoritmo se vuelve robusto en la medida que los resultados son repetibles. Obviamente, esta repetibilidad depende de que los datos de entrenamiento sean los mismos, puesto que, en la realidad, las pruebas pueden dar resultados diferentes, pero el algoritmo seguirá siendo robusto.

De las desventajas de los bosques aleatorios es que el número de árboles para generar el modelo influye en el tiempo de cómputo y suele ser un problema para el entrenamiento en línea.

5.9. Entrenamiento

Uno de los aspectos fundamentales para poder realizar algoritmos de aprendizaje máquina es el proceso de entrenamiento. Podemos definir el entrenamiento en IA como el proceso por el cual un algoritmo se adapta a los datos. Es decir, es el proceso en el que se detectan los patrones de un conjunto de datos: el corazón del aprendizaje máquina. Una vez identificados los patrones, se pueden hacer predicciones con nuevos datos que se incorporen al sistema.

Por ejemplo, los datos históricos de las compras de libros en una web *online* se pueden usar para analizar el comportamiento de los clientes en sus procesos de compra –títulos visitados, categorías, historial de compras, etcétera–, agruparlos en patrones de comportamiento y hacer recomendaciones de compra a los clientes nuevos que siguen los patrones ya conocidos o aprendidos.

Sin embargo, existen ocasiones en las que el algoritmo se adapta demasiado bien o demasiado mal a los datos. En general, se conoce como subentrenamiento –subaprendizaje o subajuste– cuando el modelo se adapta muy mal y sobreentrenamiento –sobreaprendizaje o sobreajuste– cuando se adapta demasiado bien. Esto sucede debido a que, en ocasiones, cuando se trabaja con un set de datos para predecir o clasificar, se busca que la precisión del modelo sea alta, mientras más se modifican los parámetros para que la precisión sea alta, se agregan o quitan características sin darse cuenta de que, para los datos de prueba o para otros datos que puedan ser agregados posteriormente, este modelo no es útil.

El sobreentrenamiento se refiere a un algoritmo que modela los datos demasiado bien. Este «demasiado bien» se refiere a que el entrenamiento también incluye el ruido del sistema y el modelo trata de que su predicción sea idéntica en todos los datos. El problema de esto es que el modelo no es real y cualquier dato nuevo impactará negativamente en los resultados y la habilidad del modelo para poder predecir correctamente.

Lo contrario es el subentrenamiento. Este se refiere a los modelos que son tan simples que no logran explicar su varianza y los cambios en el modelo propuesto y por obvias razones, la exactitud del modelo

tendrá un rendimiento pobre. La explicación de un entrenamiento correcto con respecto a los datos, los mismos datos subentrenados y sobreentrenados, se muestran en la figura 5.11.

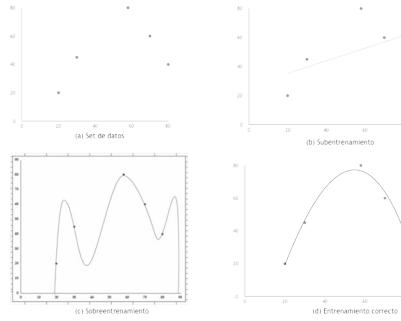
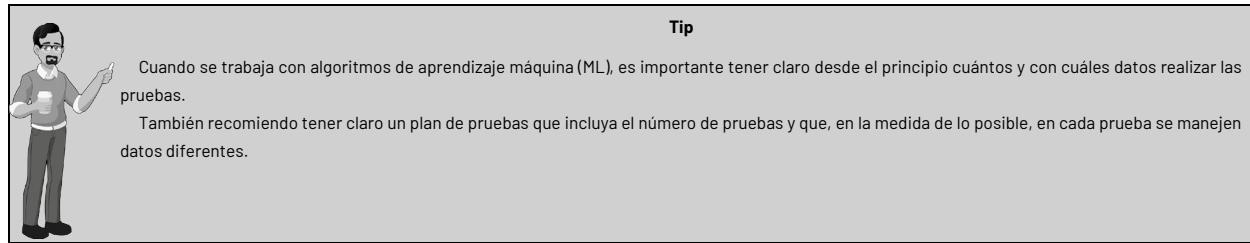


Figura 5.11. Diferentes tipos de entrenamiento

5.9.1. Partitionar los datos de entrenamiento

No es recomendable usar todos los datos disponibles para el entrenamiento por diferentes razones. En primera, puede haber ruido incrustado en los datos y un buen algoritmo de aprendizaje tiene que ser capaz de, a pesar del ruido, realizar predicciones o clasificaciones de manera exitosa. También es importante no utilizar todos los datos disponibles, pues al utilizar todos los datos, el algoritmo sobreaprende, es decir, «memoriza» el comportamiento de todos los datos y no le será posible ver los patrones en los datos como realmente son. Si esto ocurre, entrenaríamos un algoritmo de manera «perfecta», pero, en realidad, estaría sesgada su capacidad predictiva.



Supongamos que escogemos entrenar el algoritmo con el 80 % de los datos y dejamos, es decir, apartamos el 20 % para poder realizar las pruebas, esto aún genera muchas veces dudas de qué 20 % elegir o cómo tomarlo.

La figura 5.12 muestra la partición que se recomienda hacer para entrenar el modelo y posteriormente probarlo. Es importante mencionar que no muestra el cómo se debe de hacer la partición en cuestión del conjunto de datos, sino que es importante separar el conjunto de datos para validar y poder probar el modelo entrenado, que en este caso se separaría como 80-10-10.

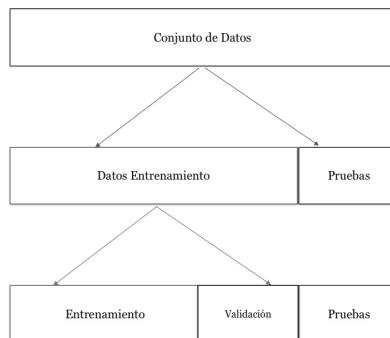
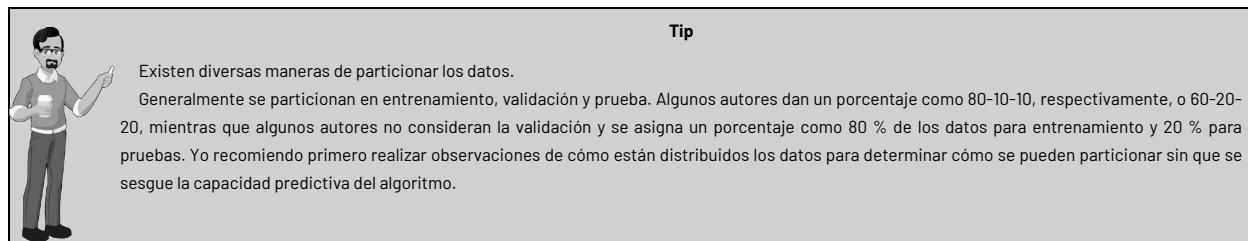


Figura 5.12. Partición recomendada del conjunto de datos en datos de entrenamiento, validación y pruebas

Algunos modelos no toman en cuenta la validación, por lo que se tiene que particionar de acuerdo a la distribución de los datos, así como también el algoritmo que se considerará para poder entrenar los datos. Así mismo, el modelo estaría sesgado si se utilizan todos los datos disponibles para realizar un modelo, pues se pretendería que tiene un 100 % de precisión el modelo de ser correcto, lo que normalmente no es una predicción real.

Un entrenamiento exitoso en cuestión de algoritmos de inteligencia artificial es el de poder crear un modelo que pueda detectar el ruido y, a pesar de este, predecir de manera adecuada. En este sentido, el ruido de los datos lo podemos definir como distorsiones en los datos que no pueden ser reproducidas consistentemente.



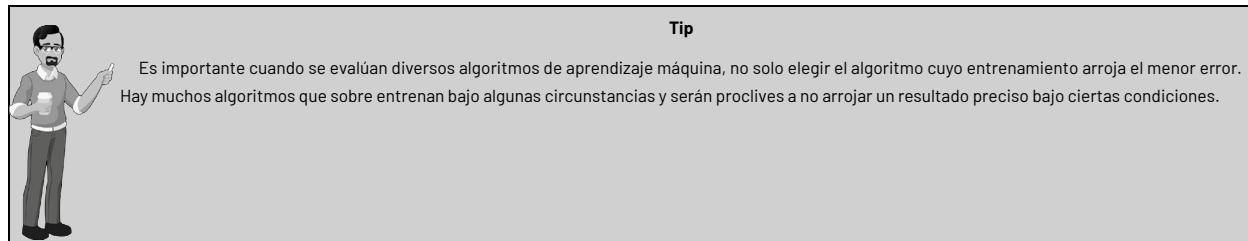
El hecho de no realizar un entrenamiento correcto de los datos tiene que ver, entre otras cosas, con que el algoritmo memoriza el ruido. Debido a que el ruido no es consistente, esto impedirá en gran medida la habilidad del algoritmo de reconocer patrones de los datos más allá del set de datos y ver falsos patrones. Esto se conoce comúnmente como sesgo del set de entrenamiento.

El sesgo en la selección de los datos es otro problema. Si se toma el conjunto de datos como en la figura 5.12, el entrenamiento se podría seleccionar entre 80 % y el 20 % para pruebas, o ese 20 % dividirlo entre validación y pruebas.

5.10. Evaluación del modelo

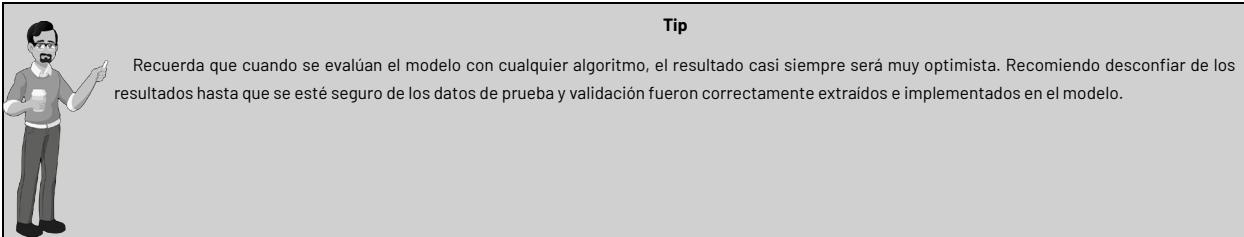
La evaluación del modelo es parecida a cómo se evalúan a los estudiantes en las escuelas. El hecho de dar calificaciones tiene ciertos propósitos. Uno de los propósitos más básicos es el de proveer una retroalimentación al alumno conforme avanza en el camino del aprendizaje. De la misma manera, se tiene que evaluar el algoritmo para determinar el grado en el cual dicho algoritmo pudo aprender y así determinar el grado de éxito del algoritmo con respecto al entrenamiento y poder retroalimentar al algoritmo para que cada vez pueda tener una mayor capacidad predictiva.

En el presente apartado, se abordarán diversos métodos para poder evaluar la calidad del modelo. Uno de ellos tiene que ver con medidas de rendimiento del algoritmo. Podemos definirlo de manera simple como la comparación entre el resultado del modelo y los resultados que esperamos.



Una de las maneras para poder medir el rendimiento de los algoritmos de aprendizaje máquina es el de segregación. Este consiste principalmente en apartar –es decir, segregar– una porción del conjunto de datos de los datos de entrenamiento para, posteriormente, evaluar el modelo. Como se comentó anteriormente, es importante que se segregue los datos de prueba antes de la etapa de entrenamiento. El

hacerlo posteriormente es un error, pues se prestará a sobreentrenamiento y a una sobrevaloración de la capacidad predictiva del algoritmo en cuestión, debido a que el algoritmo ya «observó» los datos, por lo tanto, sabe cómo se comportan y los patrones que pudieran tener, influyendo falsamente en el número de errores del algoritmo.



Recuerda que cuando se evalúan el modelo con cualquier algoritmo, el resultado casi siempre será muy optimista. Recomiendo desconfiar de los resultados hasta que se esté seguro de los datos de prueba y validación fueron correctamente extraídos e implementados en el modelo.

La manera más simple de evaluar un modelo con características nominales y discretas es por medio de la tasa de clasificación. Podemos definir la tasa de clasificación incorrecta como uno —es decir, el 100 %— menos el número de predicciones incorrectas dividido entre el número total de predicciones, como se muestra en la ecuación 5.14:

$$\text{Tasa de clasificación} = 1 - \frac{\text{clasificación incorrecta}}{\text{total de predicciones realizadas}} \quad (5.14)$$

Si, por ejemplo, tenemos que se realizaron veinte predicciones, de las cuales cuatro se clasificaron de manera incorrecta —por ejemplo, las instancias i2, i5, i8 e i15—, significa que el 20 % fue clasificado incorrectamente, es decir, mi tasa de clasificación sería de $1 - 0.2 = 0.8$ u 80 % de clasificaciones correctas.

Sin embargo, en muchas ocasiones, no es suficiente solamente saber el número de clasificaciones incorrectas del total, sino tener un panorama más amplio del rendimiento del algoritmo. Una de estas medidas de rendimiento de un algoritmo es la creación de una matriz de confusión.

La matriz de confusión provee a mayor detalle con respecto a un set de prueba qué tan bien está prediciendo —o clasificando, depende del problema en cuestión— el algoritmo. La matriz de confusión funciona de la siguiente manera: para un problema con una característica que funcione como binaria —para cuestiones de explicarlo, nos referiríamos a esos niveles binarios como positivos o negativos—. En este sentido, solo puede haber cuatro diferentes tipos de resultados que arrojen la matriz de confusión:

- Verdadero positivo (TP): se refiere TP a sus siglas en inglés (*true positive*). Se dice de una instancia en el set de pruebas en la que se espera que tenga un valor positivo de su característica y se predice un valor positivo también.
- Verdadero negativo (TN): se refiere TN a sus siglas en inglés (*true negative*). Es una instancia en el set de pruebas en la que se espera un valor negativo de su característica y se predice un valor negativo también.
- Falso positivo (FP): es una instancia en la cual se tiene un valor negativo a pesar de haber predicho un valor positivo.
- Falso negativo (FN): es una instancia en la cual se tienen un valor positivo a pesar de haber predicho un valor negativo.

Cada espacio en la matriz de confusión representa uno de estos posibles resultados (TP, TN, FP, FN) y se cuenta el número de veces que este resultado ocurrió en la prueba del algoritmo una vez que fue entrenado. La tabla 5.5 muestra la estructura de la matriz de confusión. Las columnas en la tabla muestran la predicción —positiva y negativa—, mientras que las filas muestran lo que se esperaba.

		Predicción	
		Positiva	Negativa
Valor	Positiva	TP	FN
	Negativa		

Esperado	Negativa	FP	TN
1	90.00%	2.00%	2.00%
2	1.00%	95.00%	0.00%
3	0.00%	20.00%	75.00%
4	0.00%	0.00%	98.00%
5	1.00%	1.00%	96.00%

Tabla 5.5. Matriz de confusión para cálculo del error

Es incluso obvio que los valores que representan la diagonal, es decir, TP y TN, son los que interesan entre la predicción.

Se puede calcular la tasa de clasificación de una manera más precisa con la matriz de confusión de acuerdo a la ecuación 5.15. Utilizando la matriz, se puede calcular como una métrica de exactitud del modelo.

$$\text{Exactitud} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \quad (5.15)$$

La precisión se puede definir como la tasa de las muestras predichas que son relevantes y se calcula de acuerdo a la ecuación 5.16.

$$\text{Precisión} = \frac{\text{TP}}{\text{TP} + \text{FP}} \quad (5.16)$$

La sensitividad –también llamado *recall*– se puede definir como la tasa de las muestras seleccionadas que son relevantes a la prueba. Se calcula de acuerdo a la ecuación 5.17.

$$\text{Sensitividad} = \frac{\text{TP}}{\text{TP} + \text{FN}} \quad (5.17)$$



Ejemplo

Las pruebas para los algoritmos de árboles de decisión que se recomiendan son diferentes a la mayoría de las pruebas para los demás algoritmos. Se recomienda realizar pruebas de cobertura o MC/DC. En dichas pruebas, se busca que las instancias con las que se prueba el algoritmo pasen por lo menos una vez por cada nodo final del árbol.

Por otro lado, el puntaje F1 –puntaje F-beta con una beta de 1– se puede definir como la media harmónica entre *recall* y precisión y se calcula como se muestra en la ecuación 5.18. Usualmente, se utiliza un puntaje F-beta de 1, pues, en general, se le da el mismo peso tanto al *recall* como a la precisión, aunque no siempre es el caso, como se mostrará más adelante.

$$\text{Puntaje F1} = \frac{2 * \text{TP}}{(2 * \text{TP} + \text{FP} + \text{FN})} \quad (5.18)$$

Una vez que se tienen las métricas de evaluación del modelo, se recomienda hacer una matriz de confusión entre todas las clases que tiene el modelo. Este tipo de matriz tiene varias funciones: por un lado, se puede observar la exactitud del modelo por clase de una manera gráfica, pero también es útil para saber qué clases está confundiendo con mayor frecuencia y, de esta manera, ajustar el modelo, como se muestra en la figura 5.13.

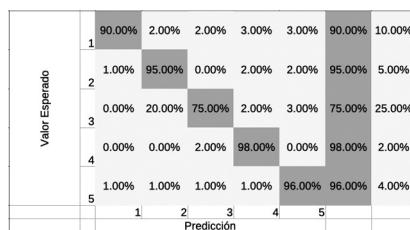


Figura 5.13. Matriz de confusión para un ejemplo de cinco clases

Como se puede observar en la figura 5.13, se tiene un ejemplo de matriz de confusión con cinco clases, en el eje de las abscisas se muestra la predicción de cada una de las clases, usualmente, solo se numeran, no se

pone la etiqueta de la clase. En el eje de las ordenadas, se tiene el valor real de la clase –valor esperado–, también llamado en inglés *ground truth*, con el mismo número de clases, en este caso, cinco.

En la diagonal principal, se tiene la exactitud de los verdaderos positivos y verdaderos negativos, usualmente, de un color diferente, mientras que en la siguiente columna de las clases se muestra la suma de la exactitud de cada clase.

Hay que notar que en las demás celdas también hay porcentajes, estas significan los falsos positivos que se obtuvieron en la predicción. La suma de la sexta columna y la séptima, en este caso, suman el 100 % de las pruebas. Por ejemplo, si tomamos la figura 5.13, la fila tres correspondería a la clase tres. En este caso, que en la primera columna de la fila tres se tiene un 0 % y en la segunda un 20 % significa que, en las pruebas realizadas, la clase tres no la confundió con la uno –0 % de error–, pero en el 20 % de los errores al clasificar la clase tres, la confundió con la dos. Así mismo, la columna cuatro de esa fila solo tiene un 2 % y la columna cinco un 3 %. La columna seis de los verdaderos positivos y la columna siete de los errores debe sumar 100 %. Con este ejemplo, se puede notar que, para la clase dos, el clasificador suele pensar en un 20 % que se tienen una clase tres, por lo que es en ese punto donde hay que ajustar el modelo.

5.10.1. Sesgo

El sesgo (*bias*) se define como la raíz cuadrada de la media de la diferencia entre las predicciones y los valores reales. Es una medida de qué tan bien el modelo se ajusta a los datos.

Un sesgo de cero indica que tanto el error de entrenamiento como el error de validación tienden a cero, y que el modelo representa a los datos reales a la perfección. Evidentemente, esto es poco realista por la distribución de los datos, el ruido, el algoritmo utilizado, entre muchos otros, así que el sesgo es prácticamente inevitable. Esto, usualmente, se conoce como **error irreductible**.

Lo que se espera del entrenamiento del algoritmo es que la pérdida decrezca conforme se avanzan las iteraciones. Si esto no sucede, se asume que el modelo no se ajusta apropiadamente a los datos de entrenamiento. En este caso, una de las estrategias es modificar el algoritmo o cambiar los hiperparámetros y/o configuraciones de este.

En este caso, se conoce como subaprendizaje –o infraajuste – *underfitting*–, aunque con una diferencia: en el caso del sesgo, puede llegar un punto en el que ya no decrezca el error, en el caso de sub-aprendizaje puede, que el modelo aún aprenda.

En parte, el sesgo puede estar fácilmente escondido en los datos de entrenamiento sin que lo note el algoritmo y pueda parecer que el error baja de manera normal. La manera en que este sesgo pueda ser descubierto es mediante un plan de pruebas.

Las redes profundas también aplican el término de sesgo. Indica la tendencia del comienzo del valor de la función de activación. En ocasiones, tienen un valor mayor a cero para poder tener una salida de la neurona de manera prematura. Si el valor es menor a cero, la función de activación necesita un mayor valor para poder ser activada, como se puede ver en la figura 5.14 para una función de rectificación lineal (ReLU). Para el caso de las redes convolucionales (CNN), el sesgo que se aplica después de la salida de la capa convolutiva usualmente se utiliza como una especie de umbral para determinar las partes de la entrada de la imagen que se muestren y cuáles no.

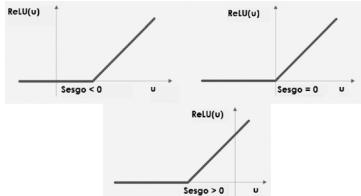


Figura 5.14. Sesgo de una función de activación

5.10.2. Sobreaprendizaje

También llamado sobreajuste. En algún punto del entrenamiento de un modelo, la pérdida de validación se estabiliza –o comienza a incrementar de nuevo, lo que se debe a una mala generalización–, mientras que la pérdida del entrenamiento continúa decreciendo. Esto, usualmente, se debe a que el modelo está sobreaprendiendo, muchas veces debido a la cantidad de parámetros que tiene o a los valores que se le dan a los mismos, que trata de ajustarse exactamente a los datos que se tienen.

Cuando sucede el sobreaprendizaje, el modelo es capaz de describir perfectamente los datos de entrenamiento, pero pierde mucha precisión cuando se intenta describir los datos de cada prueba. Esto, usualmente, es un problema, pues el modelo tiene que describir con cierta exactitud datos que no haya visto antes.

5.10.3. Subaprendizaje

Bajo-aprendizaje, subaprendizaje o infraajuste –llamado comúnmente por su anglicismo *underfitting*– se refiere a la incapacidad del modelo de ni ajustarse a los datos de entrenamiento ni generalizar con datos que el algoritmo no haya observado previamente.

Cuando se tiene un algoritmo de estas características, no es un modelo apropiado y tendrá un bajo rendimiento. La estrategia cuando se tiene un algoritmo que tenga un bajo-aprendizaje es el de cambiar de algoritmo o probar con nuevos parámetros, aunque, usualmente, cambiando de algoritmo por uno que puede hacer una mejor representación de los datos puede mejorar el aprendizaje.

5.10.4. Regularización

El sobreaprendizaje puede suceder después de un cierto número de iteraciones de entrenamiento. Para poder determinar si está sobreaprendiendo o subaprendiendo, se recomienda graficar la pérdida del entrenamiento con respecto al número de iteraciones –épocas, en el caso de las redes neuronales–. Una gráfica típica de la pérdida de entrenamiento se puede observar en la figura 5.15.

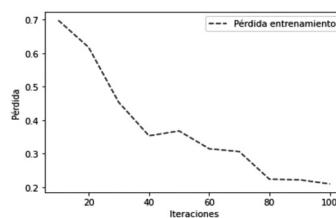


Figura 5.15. Comportamiento típico de la pérdida del entrenamiento con respecto al número de iteraciones

Una gran diferencia entre la pérdida de validación y la pérdida de entrenamiento es una pista que el modelo no generaliza bien. Esto se muestra en la figura 5.16.

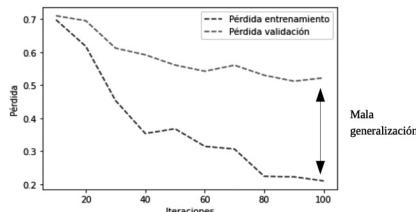


Figura 5.16. Diferencia entre pérdida del entrenamiento y pérdida de la validación, representando una mala generalización

La solución más simple para evitar este sobreentrenamiento es detener el algoritmo tan pronto como la diferencia entre la pérdida de entrenamiento y validación comience a incrementar, esto se conoce como terminación prematura. La figura 5.17 muestra el punto en la que ocurre el sobreajuste. Alternativamente, la regularización puede ayudar para que la diferencia no incremente.

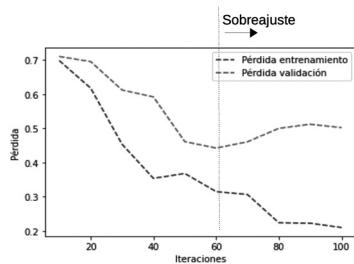


Figura 5.17. Pérdida del entrenamiento y pérdida de la validación mostrando el momento de sobreaprendizaje

La regularización es un método para evitar una varianza alta y un sobreajuste e incrementar la regularización. El objetivo fundamental de la regularización es mantener las predicciones en un estado estable y la pérdida de la validación en un valor cercano a la pérdida del entrenamiento. Sin entrar en muchos detalles, usualmente, se implementan las funciones de regularización L1 y L2 para reducir los coeficientes y penalizar los coeficientes grandes. Hay diferentes formas de estos algoritmos de regularización. Por ejemplo, en redes neuronales se conoce usualmente con el anglicismo *dropout*. La diferencia entre previo a la regularización y después de una regularización se muestra en la figura 5.18.

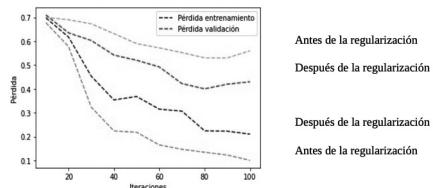


Figura 5.18. Diferencia de la pérdida del entrenamiento y pérdida de la validación antes y después de la regularización

5.10.5. Generalización

El que un modelo aprenda los datos es bueno, pero es más importante, cuando se desarrollan algoritmos de aprendizaje máquina, que los modelos tengan buena generalización. Los resultados de cualquier buen modelo se suponen que deben de tener tanto un bajo sesgo como una baja varianza, por lo que la generalización puede ser necesaria para evitar un sobreaprendizaje o subaprendizaje.

La generalización se presenta cuando el modelo puede aprender correctamente, incluso con datos que el modelo nunca haya visto. Es por eso que es importante graficar no solo la pérdida del entrenamiento y observar que decrece, sino también graficar la pérdida de la validación. Típicamente, la pérdida de la validación será un poco más alta, pero, para observar una buena generalización, deberán de decrecer relativamente a la par.

5.11. Predecir el rendimiento de un modelo

Para problemas de clasificación, es común medir el rendimiento del algoritmo en términos de la tasa de error. El algoritmo predice la clase de cada instancia, si es correcta, se cuenta como éxito, de lo contrario, se cuenta como error. La tasa de error se mide como la proporción de errores con respecto a todas las instancias de prueba, la cual mide el rendimiento en general del algoritmo.

Sin embargo, en muchas ocasiones, es importante poder tener métricas de predicción del comportamiento del algoritmo, es decir, cómo se comportará en futuras predicciones. Para poder predecir el rendimiento del algoritmo con datos nuevos, se requiere evaluar la tasa de error por medio del set de pruebas. Para poder realizar esta evaluación, se tiene que asegurar de que tanto el set de entrenamiento como el set de pruebas son representativos de los datos a evaluar, de otra manera, se estará sesgando la capacidad predictiva del algoritmo.

Por ejemplo: consideremos que se tiene la métrica de tasa de error, se puede tener en un algoritmo, por ejemplo: 75 % de exactitud del modelo –o, dicho de otra manera, una tasa de error del 25 %–. Sin embargo, se tendrá un rango mayor de confianza si ese 25 % de tasa de error está basado en un set de prueba de diez mil instancias en lugar de uno de diez instancias.

La pregunta entonces sería: ¿cuál sería la medida ideal de confianza en la tasa de error? Si se tiene un error del 25 %, es un error esperado, pero no se puede asegurar con el 100 % de confianza en todos los casos. Al ser un estimado, se esperaría que el rendimiento del algoritmo sea del 75 % o alrededor de esa cifra.

El problema radica en que alrededor del 75 % no da una medida exacta, puede ser con una desviación del 5 %, del 10 % o del 20 %. Si fuera del 20 % la desviación, significaría que podría caer la medida del rendimiento a 65 % –es decir, 10 % abajo del estimado o hasta 10 % arriba del estimado–.

Para contestar estas preguntas, se requiere de un poco de razonamiento estadístico. En estadística, una sucesión de eventos independientes que aciertan o fallan es llamado un proceso Bernoulli.

El ejemplo clásico es tirar una moneda al aire. Puede salir cara o puede salir cruz. Asumamos que la moneda está manipulada, por lo que saldrá cara setenta y cinco veces de cada cien que se tire la moneda. Asumamos, también, que siempre predecimos que saldrá cara, por lo que tendríamos el 75 % de tasa de éxito, como se comentó con anterioridad.

En este ejemplo, la tasa de éxito de que la moneda sea cara se determina como p . Supongamos que de N veces que se tira la moneda, S son exitosos –es decir, salió cara–, entonces la tasa de éxito sería:

$$f = S/N \quad (5.19)$$

La media y la varianza de cada proceso Bernoulli con una tasa de éxito p sería p y $p(1-p)$, respectivamente.

La probabilidad de que una variable aleatoria X con media 0 caiga dentro de un rango de confianza $2x$ es:

$$\Pr[-z \leq X \leq z] = c \quad (5.20)$$

Es decir, para $\Pr[X > z] = 5\%$, implica que hay un 5 % de probabilidades de que X caiga más de 1.65 fuera de su desviación estándar de su media, como se muestra en la tabla 5.6.

Pr[X > z]	z
0,10 %	3,09
0,50 %	2,58
1,00 %	2,33
5,00 %	1,65
10,00 %	1,28
20,00 %	0,84
40,00 %	0,25

Tabla 5.6. Probabilidad que una variable aleatoria X se encuentre dentro de un rango de confianza

O:

$$\Pr[-1.65 \leq X \leq 1.65] = 90\%.$$

Por último, se tiene la fórmula para determinar los límites de confianza con la variable aleatoria f .

$$p = \left(f + \frac{z^2}{2N} \pm z \sqrt{\frac{f}{N} - \frac{f^2}{N^2} + \frac{z^2}{4N^2}} \right) / \left(1 + \frac{z^2}{N} \right) \quad (5.21)$$

Realizando los cálculos de p , los resultados serían los límites inferior y superior del intervalo de confianza.

Ejemplo: con $f = 75\%$ y $c = 80\%$ —es decir, con $z = 1.28$ —, con $N = 1000$, el límite de confianza cambia de 0.732 a 0.767, con $N = 100$, el límite de confianza va desde 0.691 a 0.801, mientras que con $N = 10$, el límite de confianza se posiciona desde 0.549 a 0.881. Como se puede observar, la diferencia en el límite de confianza es menor conforme el número de instancias aumenta.

5.12. Validación cruzada

En algunas ocasiones, se requiere de técnicas que permitan asegurarse de que los datos puedan ser de entrenamiento y de prueba —pero no al mismo tiempo—. Es decir, si se toma las pruebas 80-20, se debe de tomar el 20 % de los datos para pruebas y separarlos antes de entrenar el modelo. En este caso, se toman los datos de manera aleatoria y, a menos que se hagan suficientes pruebas, no se tiene la certeza de que las pruebas puedan ser bien representadas por los datos que se separaron. Otro problema es que en algunas bases de datos se tiene solo un número limitado de instancias y no es tan conveniente realizar este tipo de validación 80-20.

Si se tiene una base de datos en donde las clases no tengan un balance —es decir, que se tengan muchas más instancias de una clase que de otras—, se tiene otro problema cuando se realizan pruebas por el método 80-20, y es que no se tiene la certeza de que en algunas pruebas se tomen instancias en las que se puedan probar todas las clases, estas son algunas de las razones de por qué la validación cruzada es muy utilizada.

La validación cruzada es un procedimiento que consiste en partir en k veces los datos y realizar pruebas con cada k -partición. También se le conoce a este método como k -fold.

El parámetro k indica cuántas veces se tiene que particionar los datos. Los k más utilizados son: 3, 5 y 10 particiones, cuando se utiliza el término k -fold, la k se suele reemplazar por el número de k -particiones, por ejemplo, se puede decir que se va a realizar una validación «10-fold».

Otra de las razones por la cual el método es muy popular es porque los resultados tienen un menor sesgo y se realiza una estimación menos optimista del rendimiento y la exactitud de un algoritmo.

El procedimiento general de la validación cruzada es el siguiente:

1. Se desordenan los datos de manera aleatoria.
2. Se separan los datos en k grupos. En este caso, se recomienda crear copias de los datos, cada una con la partición que le corresponde. Por ejemplo, si se quiere hacer un «5-fold», se hacen cinco copias con los datos. En la primera copia se quita la primera/quinta parte, en la segunda copia, la segunda/quinta parte y así sucesivamente hasta completar las cinco copias.
3. Las partes que se quitaron serán los bancos de pruebas. La primera prueba se realiza con el modelo de la primera copia y los datos de prueba que se quitaron del mismo y así sucesivamente.
4. Se guardan las métricas de calidad —error, tasa de clasificación, exactitud, precisión, sensitividad y puntaje F-beta—.
5. Se desecha el modelo y el banco de pruebas con el que se realizó.
6. Se cambia al siguiente modelo con el siguiente banco de pruebas, se repite el procedimiento desde el paso cuatro hasta que se terminen las pruebas con todas las particiones.

La manera estándar de conocer la exactitud del modelo si se tiene un set fijo de instancias es utilizar una validación cruzada «10-fold». En este caso, los datos se dividen en diez partes lo más homogéneamente posible –es decir, que cada partición contenga aproximadamente la misma cantidad de datos–, se realiza diez copias de los datos y se hace el entrenamiento y prueba diez veces, cada una con los datos de cada copia, como se muestra en la figura 5.19. Posteriormente, se estima el error mediante una media o un gráfico de caja.

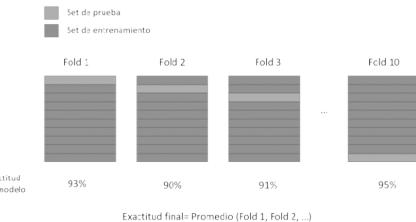


Figura 5.19. Ejemplo de la distribución de las pruebas para una validación cruzada «10-fold»

5.13. Validación en línea

La validación en línea también ocurre después de que un modelo ha sido generado. La diferencia es que, cada vez que recibe una instancia nueva, el modelo se puede reentrenar para poder ajustarse a los datos que se han validado.

La validación en línea ocurre cuando el entrenamiento acumula aprendizaje de un cierto número de instancias y se va actualizando conforme se van obteniendo más datos. En la validación en línea, se van haciendo pruebas con un número designado de datos llamado **tamaño del lote**.

Este tipo de pruebas y validación son útiles cuando el algoritmo requiere aprender y entrenarse a la vez o cuando no se tienen todos los datos de prueba o se siguen recibiendo datos en tiempo real o con relativa frecuencia. Es así como, usualmente, aprende el cerebro humano. No tenemos toda la información que vamos a recibir de antemano, ajustamos nuestras decisiones con respecto a la información que tenemos en ese momento, y podemos ajustar conforme se tiene más información.

Sin embargo, la validación en línea no es muy común, pues muchos algoritmos no pueden ser entrenados en línea.

5.14. Validación a una instancia

La validación a una instancia está considerada como una forma extrema de la validación cruzada, también llamada *f-fold*. Consiste en realizar igual número de pruebas como instancias se tengan en la base de datos utilizando una única instancia de prueba.

Esto significa que cada *fold* de la prueba contiene una sola instancia y el set de entrenamiento contienen todas las demás instancias. Este tipo de pruebas funcionan muy bien cuando se tiene un número reducido de instancias y no se tienen buenos resultados como para hacer una validación cruzada tipo «10-fold» a los datos. Al final de la etapa de pruebas, se obtiene la media de todos los resultados con validación a una instancia.

5.15. Pruebas de cobertura

Las pruebas de cobertura son muy utilizadas principalmente para algoritmos como árboles de decisión. En dichas pruebas, se pretende que se prueben casos por cada nodo final de tal forma que se cubran todos –o

prácticamente todos— los nodos en el plan de pruebas.

Se pretende en estos casos que la cobertura sea igual o cercana al 100 %. Por ejemplo, si se tiene el árbol de decisión de las condiciones climáticas que se explicó anteriormente y se muestra en el apéndice A.5, en la figura 5.20 se realizarían las siguientes pruebas para asegurar una buena cobertura.

En la figura 5.20 se muestran los tres tipos de nodos que se pueden encontrar en un árbol de decisión: nodo inicial o raíz, nodos intermedios y nodos finales. También se muestran las posibles transiciones que se pueden tomar de acuerdo a la instancia de prueba por medio de flechas. Las pruebas de cobertura se enfocan en llegar por lo menos una vez a cada nodo final.

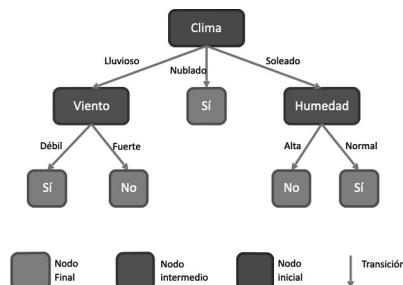


Figura 5.20. Ejemplo de un árbol de decisión para pruebas de cobertura

El plan de pruebas para este tipo de algoritmos se realiza numerando cada nodo final. El objetivo de esto es que se pueda tomar por lo menos una instancia cuya transición sea cada nodo final.

Si tomamos como ejemplo la figura 5.21, se observa que se deben tener por lo menos cinco instancias de prueba de acuerdo a la numeración de los nodos finales. No importa si se numeran los nodos finales de izquierda a derecha o de derecha a izquierda, siempre y cuando se sea consistente. En dicho ejemplo, se está probando el nodo final cuatro —marcado con otro color—. En esta prueba de cobertura, se tiene que probar que una instancia cuyo clima sea soleado y la humedad sea alta —por ejemplo, las instancias de la base de datos del apéndice A.5: i1, i2 e i8—.

De esta forma, se buscan las instancias que podrían cumplir con cada nodo final y realizar por lo menos una prueba con cada uno. Se termina la fase de pruebas cuando se tenga la cobertura deseada.

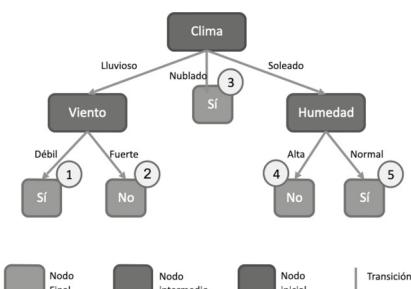


Figura 5.21. Ejemplo de una prueba de cobertura



Ejemplo

En ocasiones, no es posible tener una cobertura del 100 % en las pruebas. Esto se debe a múltiples razones: una de ellas es el gran número de posibles nodos que el modelo puede tener o la profundidad del árbol. Otra puede ser que provengan de otra función o módulo y no se pueda llegar a ese nodo final con los datos de entrada. Esto último se conoce como falta de visibilidad en las pruebas. Se recomienda que se realice la mayor cobertura posible aunque no alcance el 100 %.

5.16. Métricas de exactitud de un modelo

Existen varias maneras de determinar qué tan bueno es un modelo para representar los datos. Algunas de ellas se han abordado en este libro. Sin embargo, de manera general, se pueden agrupar en métricas para datos categóricos y métricas para datos continuos.

5.16.1. Métricas para datos categóricos

Esta sección describe las más importantes métricas para evaluar el rendimiento de datos categóricos.

En este capítulo se ha hablado de algunas métricas para evaluar el rendimiento de los algoritmos cuando se tienen datos categóricos. Por ejemplo, la tasa de clasificación, misma que puede ser calculada de la siguiente manera:

$$\text{Tasa de clasificación} = 1 - (\text{clasificación incorrecta} / \text{total de predicciones realizadas}) \quad (5.22)$$

Además de los verdaderos positivos (TP), verdaderos negativos (TN), falsos positivos (FP) y falsos negativos (FN), se pueden generar métricas con las tasas de incidencias de cada uno de ellos de la siguiente manera:

$$\text{TPR (tasa de verdaderos positivos)} = \text{TP} / (\text{TP} + \text{FN}) \quad (5.23)$$

$$\text{TNR (tasa de verdaderos negativos)} = \text{TN} / (\text{TN} + \text{FP}) \quad (5.24)$$

$$\text{FPR (tasa de falsos positivos)} = \text{FP} / (\text{TN} + \text{FP}) \quad (5.25)$$

$$\text{FNR (tasa de falsos negativos)} = \text{FN} / (\text{TP} + \text{FN}) \quad (5.26)$$

Existe una gran relación entre dichas métricas. Por ejemplo:

$$\text{FNR} = 1 - \text{TPR} \quad \text{y} \quad \text{FPR} = 1 - \text{TNR} \quad (5.27)$$

La exactitud con la que el modelo mide el rendimiento de un algoritmo puede ser calculada mediante la métrica exactitud (*accuracy*) de la siguiente manera:

$$\text{Exactitud} = (\text{TP} + \text{TN}) / (\text{TP} + \text{TN} + \text{FP} + \text{FN}) \quad (5.28)$$

Para mostrar la tasa de las muestras predichas que son relevantes se tiene la métrica de precisión. En términos más simples, la precisión es la métrica que nos permite cuantificar el número de predicciones correctas realizadas. Puede ser calculada de la siguiente manera:

$$\text{Precisión} = \text{TP} / (\text{TP} + \text{FP}) \quad (5.29)$$

Así mismo, el *recall* (*sensitividad*) muestra la tasa de las muestras que se seleccionaron para las pruebas y que son relevantes para la misma. En otras palabras, el *recall* se puede definir como el porcentaje de predicciones correctas tomadas de la clase de predicciones correctas sin tomar en cuenta los falsos positivos. Se calcula de la siguiente manera:

$$\text{Sensitividad} = \text{TP} / (\text{TP} + \text{FN}) \quad (5.30)$$

Actualmente, se utilizan, además de las métricas anteriores para datos categóricos, tanto puntaje F1 como puntaje F2. El puntaje F1 calcula la media armónica de la precisión y *recall* de forma que se enfatiza al valor

más bajo entre ambas. Se calcula de la siguiente manera:

$$\text{Puntaje F1} = (\frac{2 * \text{TP}}{2 * \text{TP} + \text{FP} + \text{FN}}) \quad (5.31)$$



Ejemplo

El puntaje F-2 se utiliza mucho en aplicaciones de IA médicas. Por ejemplo, un clasificador de mastografías que detecta posible tumores. En este caso, se considera mucho más grave que no detecte un posible tumor (falso negativo) que una detección de un tumor inexistente (falso positivo). En este caso, se ajusta el factor beta como 2, pues recall es el doble de importante que la precisión.

Para algunas aplicaciones, se tiene que considerar también la métrica de puntaje F2 –también llamado $F\beta$ -score o puntaje $F\beta$ con un β de 2–. Como se mostró en el ejemplo anterior, en algunas ocasiones, el *recall* es mucho más importante que la precisión, por lo que el factor beta (β) entra en juego.

$$F\beta = \frac{(1 + \beta^2) * \text{precision} * \text{recall}}{(\beta^2 * \text{precision}) + \text{recall}} \quad (5.32)$$

Si se usa el ejemplo de las mastografías que se vio anteriormente, se considera mucho peor un falso negativo que un falso positivo. En este caso, podemos asumir que el modelo se prueba diez veces con diferentes mastografías. En este ejemplo hipotético, el modelo detecta un tumor en seis mastografías y ningún tumor en cuatro. Después se descubre que el modelo clasificó erróneamente uno que no era tumor en el grupo de los que sí detectó tumor y dos tumores en el grupo de los cuatro en los que no se había detectado un tumor.

En este caso, las métricas para datos categóricos quedarían de la siguiente manera:

TP = 5

TN = 2

FP = 1

FN = 2

Tasa de clasificación = $1 - (3/10) = 0.7$, es decir, la tasa de clasificación es 0.7 o 70 %. Evidentemente, son muy pocas pruebas para dar métricas más exactas, pero sirve como un buen ejemplo de cómo interpretar las métricas de rendimiento para datos categóricos.

A pesar de que no es mala, no se puede determinar qué está sucediendo con el modelo ni con las falsas clasificaciones, por lo que se determina la exactitud tomando en cuenta todos los falsos entre todas las pruebas realizadas.

Exactitud = $(\text{TP} + \text{TN}) / (\text{TP} + \text{TN} + \text{FP} + \text{FN}) = 7 / 10 = 0.7$.

Mientras que la precisión:

Precisión = $\text{TP} / (\text{TP} + \text{FP}) = 5 / (5 + 1) = 0.833$ o 83.33 periódico.

Como se puede observar, la exactitud muestra un resultado más bajo que la precisión. Esto se debe a que las pruebas mostraron que se tiene tanto falsos positivos como falsos negativos, por lo que siempre se recomienda que ambas métricas se obtengan a la par. Una diferencia muy grande entre precisión y exactitud puede evidenciar que las pruebas están detectando una gran cantidad de falsos negativos. En el caso del *recall* (sensitividad), quedaría de la siguiente manera:

Sensitividad = $\text{TP} / (\text{TP} + \text{FN}) = 5 / (5 + 2) = 0.7142$ o 71.42 %.

Como el valor de la sensitividad es similar al de la exactitud, indica en este caso que las pruebas no son proclives estadísticamente a dar una gran cantidad de falsos positivos.

Por último, usualmente en estos casos, los puntajes F1 y $F\beta$ se analizan de manera conjunta de la siguiente manera, teniendo en cuenta que el factor beta es dos, debido a que se considera en este ejemplo el doble de

importante un falso negativo que un falso positivo:

$$\text{Puntaje F1} = (2 * \text{TP}) / (2 * \text{TP} + \text{FP} + \text{FN}) = (2 * 5) / (2 * 5 + 1 + 2 = 10 / 10 + 3 = 0.7692 \text{ o } 76.92\%).$$

Mientras que el puntaje $F\beta$ quedaría de la siguiente manera:

$$F\beta = (1+\beta^2) * \text{precisión} * \text{recall} / (\beta^2 * \text{precisión} + \text{recall}) = ((1+2^2) * \text{precisión} * \text{recall}) / (2^2 * \text{precisión} + \text{recall}) = (1+4) * 0.833 * 0.7142 / (4 * 0.833) + 0.7142 = (5) * 0.833 * 0.7142 / (3.332) + 0.7142 = 2.9746 / 4.0462 = 0.7351.$$

De esta forma, se puede ver que el puntaje F_2 en este caso tiene un menor balance, pues es más importante no tener verdaderos negativos que falsos negativos.

Existe también el balance contrario llamado puntaje $F_{0.5}$. Los F-score se pueden analizar de la siguiente manera:

Puntaje $F_{0.5}$ (beta de 0.5): más peso a la precisión, menos peso al *recall*. Es decir, pone más atención en minimizar falsos positivos que en falsos negativos.

Puntaje F1(beta de 1): un balance entre precisión y *recall*.

Puntaje F2(beta de 2): menos peso a la precisión, pero más peso al *recall*. Es decir, pone más atención en minimizar los falsos negativos que los falsos positivos.

Debido a esto, se debe tener una idea de si se da el mismo peso a los falsos positivos que a los falsos negativos o se requiere un balance entre FN y FP. El factor beta controla ese balance como se mostró.

5.16.2. Métricas para datos continuos

Hasta ahora, se han visto únicamente las métricas de error que se enfocan en datos categóricos, es decir, si existe error o no. Se enfocan principalmente en los falsos positivos y falsos negativos. Sin embargo, estas métricas no resultan útiles cuando se trata de datos continuos, por lo que se debe tener métricas de qué tan alejada está la predicción de los valores reales, no solo si es el valor exacto o no.

Para datos continuos, algunas de las métricas que pueden utilizarse son las siguientes:

- Error medio cuadrático (MSE, por sus siglas en inglés).
- Error medio absoluto (MAE, por sus siglas en inglés).
- Error relativo cuadrático (RSE, por sus siglas en inglés).
- Error relativo absoluto (RAE, por sus siglas en inglés).
- Coeficiente de correlación (CC).

MSE es probablemente el método más comúnmente utilizado para poder predecir el error. También se utiliza su raíz cuadrada, llamada *root mean-squared error* (RMSE). Si consideramos el valor real —actual— como a y el valor predicho como p , MSE y RMSE pueden calcularse como la diferencia entre el valor actual y el predicho al cuadrado entre el número de valores (n) de la siguiente manera:

$$\text{MSE} = ((p_1 - a_1)^2 + \dots + (p_n - a_n)^2)/n \quad (5.33)$$

$$\text{RMSE} = \text{SQRT} ((p_1 - a_1)^2 + \dots + (p_n - a_n)^2)/n \quad (5.34)$$

El error medio cuadrático permite evaluar el rendimiento de múltiples modelos en un problema de predicción cuando se trata de datos continuos. MSE varía en un rango de $[0, \infty]$, siendo un menor valor de MSE, un mejor rendimiento del modelo.

El error medio absoluto (MAE) es una medida que solo promedia las magnitudes de cada error de manera individual sin tener en cuenta su signo. Tiene una ventaja con respecto a MSE, con MAE, los valores atípicos

(outliers) no son exagerados y se trata a todos los errores de la misma manera de acuerdo a su magnitud. MAE se calcula de la siguiente manera:

$$MAE = (|p_1 - a_1| + \dots + |p_n - a_n|)/n \quad (5.35)$$

Con el error relativo cuadrático (RSE) se promedian todos los valores obtenidos con respecto a los valores de entrenamiento (p), por lo que RSE toma todos los errores al cuadrado y los normaliza dividiendo todos los errores del predictor, por medio de la siguiente ecuación:

$$RSE = ((p_1 - a_1)^2 + \dots + (p_n - a_n)^2) / ((a_1 - \bar{a})^2 + \dots + (a_n - \bar{a})^2) \quad (5.36)$$

Donde \bar{a} es la media del valor real.

El error relativo absoluto (RAE) es el valor del error absoluto, normalizado de la misma manera que con MAE y se calcula de la siguiente manera:

$$RAE = (|p_1 - a_1| + \dots + |p_n - a_n|) / (|a_1 - \bar{a}| + \dots + |a_n - \bar{a}|) \quad (5.37)$$

Por último, el coeficiente de correlación (CC) mide la correlación estadística entre los valores actuales (a) y los valores predichos (p). El coeficiente de correlación varía de 1 para un resultado de correlación perfecto a un 0, donde no existe correlación o -1 cuando la correlación es perfecta, pero negativa. Se calcula de la siguiente manera:

$$CC = S_{PA} / \sqrt{S_P S_A} \quad (5.38)$$

Donde:

$$S_{PA} = \text{SUM}((p_i - \bar{p})(a_i - \bar{a})) / (n - 1)$$

$$S_P = \text{SUM}((p_i - \bar{p})^2) / (n - 1)$$

$$S_A = \text{SUM}((a_i - \bar{a})^2) / (n - 1)$$

En general, se utiliza de manera indistinta la medición del error o, por defecto, se utiliza RMSE. En realidad, no existe una receta para siempre utilizar un método en particular, sino que dependerá de los datos, la aplicación, entre otros.

Se tiene que preguntar:

- ¿Qué se está tratando de minimizar?
- ¿Cuál es el costo que representa el resultado —y los cálculos— de cada uno de los métodos?

Para ejemplificar las métricas de error para datos continuos, tomemos como ejemplo los datos de partículas contaminantes PM10 del apéndice A.7. En este caso, solo se tomaron cien datos para ejemplificar dichas métricas, como puede observarse en la tabla 5.7.

Datos			
Reales	Predichos	Reales	Predichos
113	66	80	72
141	90	49	78
118	111	36	61
154	100	55	50
227	120	45	58
216	157	63	55
211	151	95	66
183	147	98	87
145	133	117	92
193	114	125	101
144	139	95	104
112	121	97	85
74	102	74	82
66	79	64	69
70	73	45	61
60	76	27	50
40	72	70	37
52	58	52	61
51	61	65	58
73	60	81	66
57	72	83	76
62	64	101	79
74	64	77	89
57	69	80	75
81	60	73	76
92	71	68	73
71	79	61	71
55	68	38	68
86	56	51	54
103	73	68	59

Tabla 5.7. Ejemplo de datos utilizados para calcular las métricas para datos continuos

En el caso de la tabla 5.7, se realizó una predicción para exemplificar la métrica de error utilizando una red profunda tipo recurrente (LSTM, en este caso). La figura 5.22 muestra solo cien datos –los datos reales vs lo que predijo la red–.

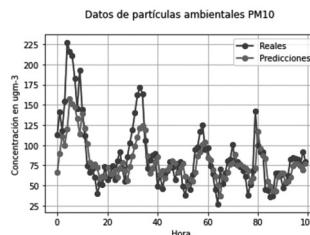


Figura 5.22. Ejemplo de datos reales vs datos predichos por una red recurrente tipo LSTM

Comenzamos por MSE calculando el error cuadrático medio para cada punto predicho menos cada punto real al cuadrado, entre cien, pues es el número de puntos que se predijeron, quedando de la siguiente manera:

$$\text{MSE} = ((p_1 - a_1)^2 + \dots + (p_n - a_n)^2) / n = 70760 / 100 = \mathbf{707.6}$$

Para calcular RMSE, solo se tiene que sacar su raíz cuadrada, por lo que RMSE quedaría de la siguiente manera:

$$\text{RMSE} = \text{SQRT}(\text{MSE}) = \mathbf{26.6007}$$

El error medio absoluto se obtiene calculando el valor absoluto de cada valor predicho menos el valor actual entre el número de puntos –cien en este caso–, quedando de la siguiente manera:

$$\text{MAE} = (|p_1 - a_1| + \dots + |p_n - a_n|) / n = 1916 / 100 = \mathbf{19.16}$$

El error relativo cuadrático (RSE) se calcularía de la siguiente manera:

$$\text{RSE} = ((p_1 - a_1)^2 + \dots + (p_n - a_n)^2) / ((a_1 - a)^2 + \dots + (a_n - a)^2) = \mathbf{0.4184}$$

Por otro lado, el error relativo absoluto quedaría de la siguiente manera:

$$\text{RAE} = (|p_1 - a_1| + \dots + |p_n - a_n|) / (|a_1 - a| + \dots + |a_n - a|) = \mathbf{0.6333}$$

Por último, el coeficiente de correlación (CC) se calcula de la siguiente manera:

$$\text{CC} = S_{PA} / \text{SQRT}(S_P S_A) = \mathbf{0.8158}$$

Existen algunas otras métricas de error como el coeficiente R^2 que son independientes del dominio a la suma de errores cuadrados entre la suma de todos los errores.

5.17. Precisión y exactitud

En términos de aprendizaje, los resultados se pueden confiar solo en la medida en la que se puede confiar los datos de entrada que la producen. Por ello, es esencial que nuestro sistema de recolección de información no tenga errores o se puedan detectar a tiempo como tales.

En este sentido, los términos exactitud y precisión entran en el juego. Exactitud (*accuracy*) se define como qué tan cerca las mediciones se encuentran de su valor «verdadero», mientras que la precisión se refiere a qué tan cerca las mediciones están de cada una. En otras palabras, la exactitud describe la diferencia entre las mediciones y el valor que deben de tener, mientras que la precisión describe la variación que existe entre cada dato con el mismo dispositivo.

La figura 5.23 muestra gráficamente la diferencia entre exactitud y precisión por medio de un tablero como si se jugara a los dardos. Exacto y preciso sería que todos los dados acertaran a la diana; no exacto pero preciso darían lejos del centro de la diana, pero todas en un lugar similar; de la misma manera, exacto pero no preciso darían muy cerca de la diana, pero separadas entre sí.



5.18. Metaaprendizaje

El metaaprendizaje en aprendizaje máquina se refiere a la capacidad del algoritmo para aprender de otros algoritmos de aprendizaje.

Lo más común es que se usen los algoritmos de aprendizaje máquina para combinar las predicciones de otros algoritmos de aprendizaje máquina mediante lo que se conoce como algoritmos de ensamble (ver sección 3.11).

Sin embargo, el metaaprendizaje también se refiere al proceso automático para ajustar un algoritmo y optimizar su rendimiento, es decir, cuando el algoritmo aprende a aprender.

En aprendizaje máquina, los algoritmos aprenden de los datos históricos y de ahí obtienen patrones para poder aprender con el paso del tiempo –iteraciones, generaciones, épocas– y generar modelos que pueden ser usados para datos futuros, en la mayoría de los casos, son los datos de prueba y/o validación.

En el caso de los algoritmos de metaaprendizaje, se requiere de la presencia de otros algoritmos de aprendizaje que ya hayan sido entrenados con datos para poder aprender de esto. Es decir, este tipo de algoritmos realizan una predicción final a partir de predicciones hechas por otros modelos.

Uno de los ejemplos de algoritmos de metaaprendizaje es la generalización apilada –usualmente, se conoce como *stacking*–, es probablemente el método de metaaprendizaje más popular. Este es un tipo de método de ensamble que combina dos o más algoritmos y utiliza lo que se conoce como un metamodelo para aprender cómo combinar mejor las predicciones para dar un mejor resultado. Este algoritmo funciona en dos etapas: en la primera, los clasificadores bases son entrenados. Posteriormente, los clasificadores base darán sus predicciones y el metaclásificador hará la predicción final en función de los clasificadores base.

Existen también otras maneras de realizar metaaprendizaje. Una de las más comúnmente utilizada es lo que se conoce como los metamodelos.

El metamodelo, básicamente, se basa en modelos con diferentes parámetros que tienen diferentes predicciones cada vez que tienen datos de prueba. El metamodelo elige el modelo cuyos parámetros está teniendo un mejor rendimiento mediante pesos o reglas, de tal manera que asigna un mayor peso si un

modelo está teniendo consistentemente un mejor resultado. A los metamodelos también se les conoce como optimización de modelos u optimización de parámetros de modelos.

Entre los metamodelos que más se utilizan para optimización de modelos se encuentran descendiente en gradiente, mínimos cuadrados, entre otros. Es importante mencionar que estos algoritmos no solo se utilizan para optimizar otros modelos, también para predecir y clasificar.

Existe otro concepto en el área de metaaprendizaje que vale la pena mencionar. En algunas ocasiones, se puede alcanzar el metaaprendizaje mediante el aprendizaje multitareas.

En este caso, el aprendizaje se ve como un algoritmo que mejora con la experiencia de realizar una tarea, por lo que el metaaprendizaje se puede definir como un algoritmo multitareas que mejora su rendimiento conforme va acumulando experiencias y realizando tareas.

Esto funciona bien cuando, en lugar de desarrollar un algoritmo para cada tarea que se desee realizar o seleccionar y configurar los parámetros de cada algoritmo para cada tarea, se buscan mecanismos para que, por medio de metaaprendizaje, se puedan agrupar tareas basadas en su similitud.

Entre estas técnicas se encuentran el aprendizaje por transferencia y la implementación de técnicas de redes profundas en aplicaciones, por ejemplo, de visión computacional.

El aprendizaje por transferencia se refiere en general al proceso donde el modelo se entrena con un problema similar en algunos aspectos a un segundo problema.

Esto es común en algunos modelos de redes neuronales profundas, donde el modelo se entrena primeramente para un problema similar al que se está tratando de resolver. En este caso, una o más capas del modelo son usadas para entrenar el nuevo problema.

Esto funciona usualmente con las capas del entrenamiento anterior ajustándolas con los nuevos datos hasta que el error de validación se estabilice y el error de entrenamiento vuelva a ser bajo. Con esto, se puede reentrenar una red para que aprenda cosas —por ejemplo, objetos— que no haya visto antes y alcanzar un modelo de metaaprendizaje por medio de aprendizaje por transferencia.

CAPÍTULO 6

Redes Neuronales

En este capítulo, se abordarán brevemente uno de los algoritmos más utilizados en inteligencia artificial: redes neuronales artificiales.

Las redes neuronales utilizan esa premisa, una especie de arquitectura –en redes neuronales se llama topología– que utiliza el funcionamiento de las redes neuronales biológicas para aprender como lo hace el cerebro humano.

Las redes neuronales están diseñadas para identificar patrones y aprender de ellos. Han sido muy utilizadas para diferentes tareas como la predicción, clasificación, regresión, entre otras. Una de las desventajas de las redes neuronales es que se requiere convertir todos los datos a un formato numérico, y en el caso de la predicción de señales, audio, imágenes, video, texto, entre otros, no es tan simple. Otra de las desventajas, principalmente, lo que se refiere a las redes neuronales profundas, usualmente, se requieren muchos datos para poder obtener buenas predicciones, con los cuales, a veces, no se cuentan, aunque existen procesos de creación de datos sintéticos que ayudan a esto –se conoce usualmente en redes neuronales como *data augmentation*–.

Las redes neuronales artificiales funcionan por capas, es decir, de manera jerárquica. En muchas ocasiones, nuestro cerebro funciona de la misma manera: supongamos, por ejemplo, que vemos una mesa, nuestro cerebro identifica la forma, después identifica qué objeto es, pues ya se tiene aprendido o nuestra memoria a largo plazo ya sabe qué es. Así mismo, podemos identificar colores, texturas, objetos posicionados encima, debajo o alrededor de la mesa, etcétera. Obviamente, no todo lo que hace nuestro cerebro funciona así, pero muchas funciones superiores se realizan de manera jerárquica por nuestro cerebro.

De la misma manera, las redes neuronales artificiales –en adelante las llamaremos solo redes neuronales– utilizan la jerarquía de neuronas para aprender. A esta jerarquía se le conoce como capas. Cada capa de una red neuronal es un set de neuronas que funcionan de manera independiente y las neuronas de esa capa se conectan a las neuronas de la siguiente capa. No hay un límite en el número de capas o de neuronas por capa, aunque hay que tomar en cuenta que, a mayor cantidad de capas y de neuronas, la complejidad de la red aumenta y no necesariamente aumenta la capacidad predictiva de nuestra red.

Una de las preguntas que más recibo es: ¿cómo sé cuántas neuronas y cuántas capas necesito? La respuesta es diferente para cada aplicación o problema a la mano, pero intentaré explicar de qué manera se puede construir y entrenar una red neuronal.

Si tenemos datos de entrada de N dimensiones, la capa de entrada consistirá entonces de N neuronas. Si se sabe que se tiene M clases en los datos de entrenamiento, la capa de salida consistirá en M neuronas. Las capas intermedias se llaman capas ocultas. El número de capas depende de la topología de la red. Una red neuronal simple consiste en un par de capas ocultas, mientras una red profunda consiste en muchas más.

Para poder realizar una clasificación, primeramente, se tiene que recolectar y preparar los datos de entrenamiento y etiquetarlos, posteriormente, la red neuronal se entrena a sí misma hasta que el error baja a cierto umbral.

El error es la diferencia entre la salida de la red y la salida predicha. Basada en la cantidad de error, la red neuronal se ajusta a sí misma hasta que el resultado se acerca lo más posible a la solución. El perceptrón es

la estructura básica de una red neuronal como se muestra en la figura 6.1.

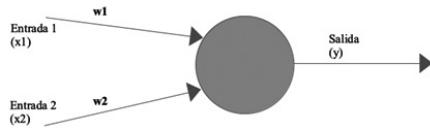


Figura 6.1. Topología de una red neuronal simple

La evolución de las redes neuronales artificiales ha cambiado mucho con los años, pero, básicamente, se sigue basando en el trabajo de Santiago Ramón y Cajal de las redes neuronales biológicas. En dicho trabajo, se pudo demostrar que:

- Las neuronas se comunican entre sí.
- Forman redes complejas.
- Se encuentran físicamente separadas de otras neuronas.

De esta manera, el perceptrón funciona de manera similar. Se tienen neuronas de entrada, de salida, interconexiones entre sí y una función de activación –en este caso, binaria– para decidir la salida.

A pesar de que el perceptrón es una red neuronal simple, la mayoría de las redes mucho más complejas –convolutivas, recurrentes, generativas, de celda de memoria, de creencia profunda, transformadores, codificadores, etcétera– se basan en versiones con más capas y más neuronas por capa del perceptrón, por lo que es importante aprender su funcionamiento.



Ejemplo

Si se tienen datos de entrada de N dimensiones, un perceptrón calcula la sumatoria de los pesos de esos N datos, les agrega una constante y produce una salida.

Esa constante se conoce como el sesgo (bias) de la neurona.

6.1. Redes neuronales convolucionales

Una de las topologías de redes neuronales profundas más usadas en este momento son las redes convolucionales –también llamadas convolutivas o CNN por sus siglas en inglés–, especialmente, para reconocimiento de imágenes.

Cuando se trabaja con redes neuronales tradicionales –por ejemplo, perceptrón–, los datos de entrada se necesitan convertir en un vector de entrada. Este vector es utilizado como las entradas a la red neuronal, las cuales pasan la información a través de las capas ocultas hasta llegar a la salida. En todas las redes neuronales, incluidas las CNN, cada neurona se conecta a las neuronas de la siguiente capa, no a neuronas dentro de la misma capa. La última capa representa la capa de salida y representa la salida de la red.

Consideremos, por ejemplo, la estructura de una imagen pequeña, de 640x480 píxeles a color. Típicamente, el que sea a color implica que se tienen tres canales para representarla –rojo, verde y azul o RGB en ese orden–, es decir, una matriz de $640 \times 480 \times 3$. Para poder representarla adecuadamente, requeriríamos 921 600 pesos por cada neurona. Si a eso agregamos varias neuronas por capa y varias capas, se volvería computacionalmente muy complejo y poco manejable.

Sin embargo, con las redes CNN, se pueden trabajar imágenes de manera más simple, pues consideran la estructura de las imágenes cuando se procesan los datos, de ahí que sea la topología de red más utilizada en procesamiento de imágenes.

Las neuronas de las CNN están distribuidas en tres dimensiones, largo, ancho y profundidad, exactamente el número y dimensionalidad que se requiere para cada imagen, es decir, largo, ancho y número de canales. Cada neurona en una CNN en una capa está conectada a algunas neuronas de la salida de la capa anterior, en contraste con una neurona con miles de conexiones, como se explicó anteriormente.

Esto funciona como una especie de filtro donde la red CNN tiene que capturar todas las características relevantes de mis datos –en este caso, imágenes–. Esto se conoce como extraer características y depende de lo que se quiera entrenar –gestos de personas, bordes, siluetas, detección de objetos, animales, peatones, etcétera–. En las capas posteriores, se extraen características superiores y se va ajustando por medio de pesos y sesgos –se conoce como *bias* en redes neuronales– hasta que pueda diferenciar estos objetos.

Existen diferencias fundamentales entre una red CNN y un perceptrón. Si regresamos al ejemplo de las imágenes, podemos asumir que, si tomamos como entrada una matriz en una CNN, pero en un perceptrón ingresamos la misma información, solo que no en forma de matriz, la pregunta sería: ¿por qué no podemos simplemente ingresar la misma información en un perceptrón pero en forma de vector?

Para responder esto, consideremos una imagen en un solo canal, en este caso, el canal R (rojo) de 7x7 píxeles. Podríamos argumentar que, en lugar de tener la matriz de 7x7, podemos usar un perceptrón y tener una entrada de 49x1, es decir, un vector de cuarenta y nueve elementos. En este caso, el cerebro humano no tardará mucho en procesar una imagen si ya la ha visto anteriormente. Si, por ejemplo, la imagen contiene una letra («A»), el cerebro la identificará de manera eficiente. Un ejemplo de esto se muestra en la figura 6.2.



Figura 6.2. Imagen vista como una matriz

Si, en cambio, la visualizamos como un vector quedaría como en la figura 6.3.



Figura 6.3. Imagen vista como un vector

A pesar de contener la misma información, el cerebro humano no reconocería tan fácilmente la «A» si se muestra como un vector a si se muestra como una matriz. Sucede algo similar con las redes CNN y el perceptrón.

Una CNN puede reconocer estos patrones porque reconoce las dependencias espaciales –en tres dimensiones, en este caso– y se reducen considerablemente los parámetros para hacerlo, en parte porque los pesos son reutilizados y el número de neuronas que se requieren para poder hacer este reconocimiento son mucho menores.

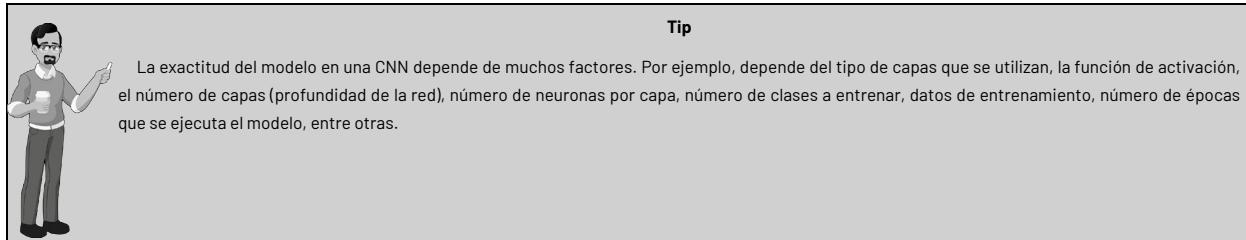
Las redes CNN típicamente tienen los siguientes tipos de capas:

- **Capa de entrada.** En el caso de imágenes, esta capa acepta como entrada todos los píxeles de la imagen como son.
- **Capa convolutiva.** Esta capa calcula las convoluciones entre las neuronas y los diferentes segmentos de la imagen. Básicamente, esta capa calcula el punto entre los pesos de la neurona y los diferentes segmentos de imagen de la salida de la capa previa.
- **Capa RLU.** Esta capa aplica la función de activación a las salidas de la capa previa. RLU significa unidad de rectificación lineal (por sus siglas en inglés). Esta capa es necesaria para agregar elementos no lineales a

la red y que esta pueda generalizar de manera correcta. El concepto de generalización se explica en el capítulo 5.4.

- **Capa de pooling.** Esta capa, usualmente, se maneja con el anglicismo *pooling*. Esta capa sirve para muestrear la salida de la anterior capa de tal forma que la estructura resultante sea de menor dimensión. Esta capa ayuda a mantener únicamente las partes relevantes de la imagen en la red. Usualmente, se utiliza al algoritmo Max-Pooling, aunque hay diferentes tipos de algoritmos en esta capa que pueden servir.
- **Capa de conexión completa.** Se conoce usualmente como *fully connected*. Esta capa calcula los resultados del *pooling*. El resultado es de tamaño $1 \times 1 \times NC$, donde NC es el número de clases que se ingresaron a la red.
- **Capa de salida.** Esta funciona de la misma manera que las capas de salida de las otras topologías de red.

De esta forma, al paso de las diferentes capas, la imagen se transforma de una matriz de píxeles a valores por cada clase.



6.2. Redes neuronales recurrentes

Otra de las topologías de redes profundas que se han estudiado y trabajado de manera extensa en los últimos años son las redes recurrentes.

Las redes recurrentes (RNN, por sus siglas en inglés) tienen muchas aplicaciones, entre ellas las que tratan de señales de tiempo. Han sido altamente usadas en aplicaciones como reconocimiento de habla, predicción de variables meteorológicas, predecir precios de mercados financieros, criptomonedas, entre otros.

Las redes recurrentes profundas, en general, también funcionan como las redes neuronales biológicas en el sentido que reciben entradas y, cuando un cierto umbral se alcanza, dispara una salida. También se parecen en el sentido que forman redes de neuronas que tienen diferentes funciones.

El concepto principal de las topologías de redes profundas basadas en recurrencias –en este caso, RNN– es tomar la información relevante de la capa anterior a manera de secuencia. Esto se hace asumiendo que todas las entradas y salidas de la red RNN son interdependientes entre sí. Esto no siempre se puede asumir, pero casi siempre se puede tomar ventaja de esa interconectividad entre capas.

La razón por la cual se llaman redes recurrentes es debido a que esta topología de red realiza la misma tarea para cada neurona de una secuencia y la salida depende del cálculo anterior. Es por eso que algunos tipos de redes recurrentes se les conoce como de memoria –LSTM, por ejemplo–, pues guarda la información de lo que calculó hasta ese punto y la envía a la siguiente capa. Un ejemplo de este tipo de redes es la red Elman, cuyo diagrama genérico se muestra en la figura 6.4.

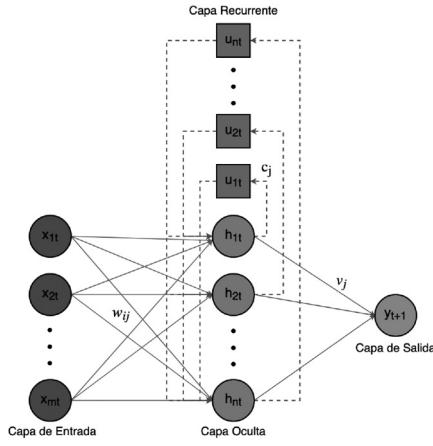


Figura 6.4. Ejemplo de una red recurrente

En el ejemplo de la figura 6.4, se observa que, además de las típicas capas de entrada, oculta y de salida, se encuentran las capas recurrentes. Esta capa recurrente tiene la función de «recordar» la salida de cada neurona de las capas ocultas, lo que permite a la red adaptarse a los datos cambiantes de entrada. Es por esta razón que las redes recurrentes son muy utilizadas para datos dependientes del tiempo o continuos.

6.3. Funciones de activación

Las funciones que se utilizan para que, cuando llega la entrada a un cierto umbral, se dispare, se conocen en redes neuronales artificiales como funciones de activación.

En otras palabras, una función de activación en una red neuronal artificial se define como la suma ponderada de las entradas, que se transforma en una salida de un nodo o nodos de una capa de la red.

En algunas ocasiones, la función de activación es llamada función de transferencia. Algunas funciones de activación no son lineales y el elegir bien el tipo de función de activación tiene un gran impacto en el rendimiento de una red neuronal.

Como regla general, todas las capas ocultas utilizan la misma función de activación, mientras que algunas capas de salida utilizan una diferente dependiendo del tipo de predicción que requiere el modelo.

Existen un gran número de funciones de activación, las más comunes son las siguientes:

- **Función de escalón** (stepwise, también llamada heaviside).
- **Función sigmoide**.
- **Función tanh** (tangente hiperbólica).
- **Función de rectificación lineal** (llamada usualmente ReLU).
- **SeLU**.
- **Softmax**.
- Entre otras...

6.3.1. Función de escalón

La función de escalón —también llamada *heaviside*— es una función de activación simple. Si la salida se encuentra por encima del umbral, la función se dispara, de lo contrario, no. Gráficamente, se puede ver en la figura 6.5.

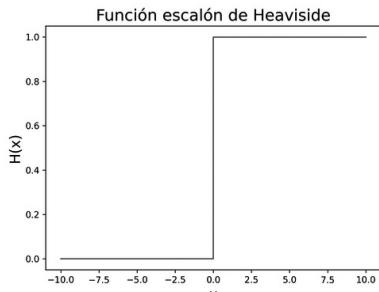


Figura 6.5. Función de activación de escalón

Como se puede ver en la figura 6.5, la salida es 1 si el valor de x es más grande o igual que 0.0 y 0 si el valor es menor a 0. Este tipo de funciones tienen el problema de que, en el punto 0.0, no es diferenciable y el cambio es muy abrupto, aunque es muy fácil de implementar.

Para algunas aplicaciones en donde la actualización de los pesos (w) es gradual, este tipo de funciones de activación no se recomiendan.

6.3.2. Función sigmoide

Para evitar el problema anteriormente mencionado de la función escalón, se puede utilizar la función sigmoide. La función sigmoide —también llamada función logística— se define como se muestra en la figura 6.6.

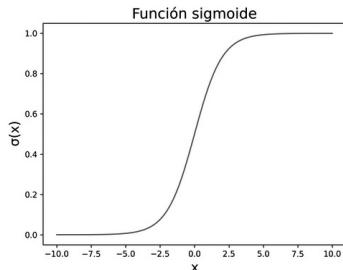


Figura 6.6. Función de activación sigmoide

Como se muestra en la figura 6.6, el valor de la función tiende a 0 cuando x tiene valores negativos y tiende a 1 cuando el valor de x es positivo. Este tipo de funciones de activación son, en general, muy útiles cuando se tiene un problema que requiere una respuesta probabilística, pues varía la activación de 0 a 1.

A pesar de que es muy usada la función de activación sigmoide, tiene una gran desventaja: para algunas aplicaciones donde se requiere un rango mayor de respuestas de salida, la función sigmoide puede tener un error grande. Es decir, en este tipo de función de activación, a cambios muy grandes de las entradas, le corresponde cambios pequeños en la salida. Este problema se conoce como *gradiente excesivo* o *escarpado* (del inglés *steep gradient*). Este problema incrementa de manera exponencial conforme se incrementan las capas de una red, por lo que se complica poder escalar los valores de salida de las redes que usan este tipo de función de activación.

6.3.3. Función tanh (tangente hiperbólica)

La función de activación tanh es una versión reescalada de la función sigmoide. Su salida varía de -1 a 1 en lugar de 0 a 1 como en la función sigmoide, como se muestra en la figura 6.7.

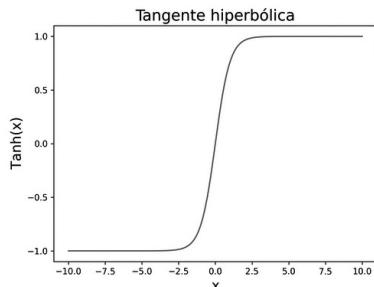


Figura 6.7. Función de activación tanh

Esta función de activación se utiliza principalmente en redes recurrentes. La razón por la cual tiene una gran utilidad se debe a que, como los valores están centrados alrededor del 0, los gradientes pueden ser mayores. Esto ayuda a que la tasa de aprendizaje del modelo pueda ser optimizada, lo cual sirve para que el modelo se pueda entrenar más rápido.

6.3.4. Función de rectificación lineal (ReLU)

La función de activación por unidad de rectificación lineal —comúnmente llamada ReLU— es probablemente la función de activación más popular para redes convolucionales y recurrentes. La función devuelve 0 cuando se obtiene una entrada negativa y, cuando se obtiene una entrada positiva, devuelve el valor de entrada. La figura 6.8 muestra gráficamente el comportamiento de la función ReLU.

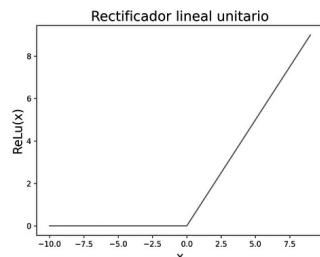


Figura 6.8. Función de activación ReLU

Con respecto a su implementación, se puede realizar de manera muy simple mediante sentencias *if*:

```
if input > 0:  
    return input  
else:  
    return 0
```

También se puede describir la función $\text{ReLU}()$ matemáticamente usando la función $\max()$ con el umbral definido en 0.0 y la entrada de la neurona z de la siguiente manera:

```
Relu = max {0, z}
```

Algunas de las ventajas de la función de activación ReLU son:

- **Simplicidad.** Esta función de activación es muy simple de implementar, pues no requiere de ajustar de manera exponencial la entrada, solo requiere de una función tipo `max()`.
- **Adecuada aproximación.** Tanto la función sigmoide como la función tanh tienen que ser aproximadas a 0, y, en ocasiones, no da 0 exacto debido al rango que está manejando la entrada. Sin embargo, la función ReLU sí da un 0 exacto cuando sus valores de entrada son 0 o negativos.
- **Comportamiento lineal.** En general, el entrenamiento se simplifica cuando sus funciones de activación son de tipo lineal, lo que es el caso de la función ReLU. También esto facilita la optimización de la conexión entre neuronas de diferentes capas, lo que lo hace una buena opción.

CAPÍTULO 7

Factores que afectan el rendimiento en un algoritmo de IA (factores a considerar)

Realizar un algoritmo de inteligencia artificial no es una tarea simple. Existen muchos algoritmos, cada uno con diversos parámetros que pueden afectar seriamente el rendimiento del algoritmo. A pesar de que actualmente existen muchas maneras de determinar si los algoritmos empleados son los adecuados, muchas veces se requieren validar de manera empírica, lo que puede resultar en errores que pueden afectar el rendimiento del algoritmo.

En el presente capítulo, se abordan una serie de factores que pueden afectar el rendimiento de un algoritmo de inteligencia artificial, tales como la manipulación y preparación de los datos, la selección de la arquitectura, los métodos y parámetros de optimización, entre otros. Estos factores no representan necesariamente una lista exhaustiva de todos los puntos a considerar ni tampoco son una regla que sigan todos los tipos de algoritmos, pues, actualmente, se cuentan por cientos.

7.1. Preparación de los datos

Uno de los factores más importantes que se tienen que tomar en cuenta al implementar un algoritmo de aprendizaje máquina es el de la preparación de los datos. En el capítulo 4, se abordan algunos temas con respecto a la simplificación, distribución, normalización, entre otros de los datos principalmente de entrada. Puede parecer algo obvio, pero, cuando se trata de datos, el primer paso es decidir qué atributos son de entrada, cuáles de salida y cuáles –de ser necesario– son atributos que no se van a utilizar para el algoritmo.

El segundo paso en cuanto a la preparación de datos tiene que ser el preparar los datos. Este proceso incluye manejar los datos atípicos (*outliers*) y problemas de cardinalidad en los datos (sección 4.3), manejar datos faltantes (sección 4.4), transformar los datos no numéricos a datos numéricos, escalar los datos, discretizar atributos, entre otros (sección 4.8).

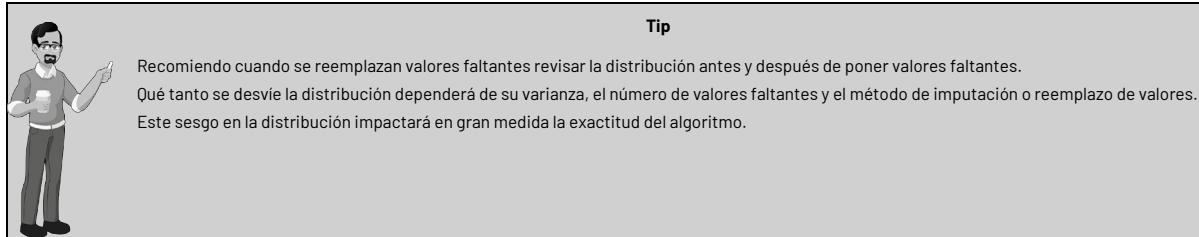
7.1.1. Datos faltantes

Es muy común en datos del mundo real que se tengan valores faltantes como se comentó en la sección 4.4. Cuando se tienen este tipo de valores, se tienen las siguientes opciones para lidiar con ellas:

- Remover la instancia o el atributo completo si hay valores faltantes. Esto puede parecer la solución más simple. Es decir, si faltan valores, se elimina completamente la instancia o incluso un atributo. Sin embargo, se introducen problemas mayores: 1) la información disponible para el entrenamiento se reduce drásticamente y puede complicar mucho el entrenamiento del algoritmo, especialmente, si los datos están limitados, y 2) información importante se puede perder por la eliminación completa, principalmente, si se trata de un atributo completo.
- Reemplazar cada valor faltante. Esto puede hacerse de diferentes maneras. Por ejemplo, para el caso de valores continuos, se utiliza generalmente la media, mientras que, para valores nominales o discretos, el valor que más se utiliza es la moda. Esto, generalmente, no introduce un sesgo en los datos cuando faltan

pocos datos. También se pueden utilizar métodos de imputación. La regla general en este caso de reemplazar los valores faltantes es la de no reemplazarlos si faltan una gran cantidad de datos.

- Codificar los valores faltantes. En este caso, se recomienda utilizarlo únicamente para ciertos casos. Estos casos son para datos con valores positivos con características numéricas. Esta codificación consiste en reemplazar cada valor faltante con valores negativos –por ejemplo: -1 o -9999–.



También, como recomendación general, para cada instancia que tenga un valor faltante, se recomienda agregar un atributo extra para indicar en cuáles instancias faltan datos. Si se entrena un algoritmo de inteligencia artificial con y sin esos valores que se agregaron y los patrones de los datos no cambian, entonces los valores que se agregaron no sesgaron el algoritmo ni tuvieron una influencia negativa en el rendimiento del algoritmo.

7.1.2. Valores atípicos

Como se mencionó en el capítulo 4, los valores atípicos pueden seriamente afectar el rendimiento de un algoritmo. Los valores atípicos pueden ser abordados de la siguiente manera:

- Remover los valores atípicos antes de comenzar la etapa de entrenamiento del algoritmo. Esto se realiza, generalmente, usando técnicas estadísticas. Aunque esto resuelve el problema de tener los valores atípicos, puede introducir un efecto negativo importante, es decir, también se puede perder información importante y tendencia de los datos conforme se elimina un valor atípico, esto, principalmente, si es un valor atípico válido, como se comentó en la sección 4.3.
- Remover los valores atípicos bajo ciertas circunstancias. Es importante conocer las razones por las cuales se pueden remover los valores atípicos: entre ellas se encuentra; si se sabe que es incorrecto. Esta es una medida para poder eliminarlo, si los valores de ese atributo se deberían de encontrar en un rango y se encuentran muy lejanos a este. La inspección visual de los datos, los diagramas de caja y los histogramas sirven de ayuda para saber si su valor es incorrecto. Así mismo, si el valor atípico en cuestión es solo uno y se tienen muchos datos, aunque no se sepa si está correcto o no, puede ser removido sin afectar el entrenamiento de los datos.
- No remover los valores atípicos bajo ciertas circunstancias. También existen razones por las cuales no se recomienda eliminar los valores atípicos. Una de ellas es: cuando los datos son críticos. Esto significa que eliminar o modificar una instancia con *outlier* cambia mucho los valores y la tendencia de los datos, por lo cual se tienen que dejar. Existen algunas recomendaciones para lidiar con estos casos donde no es conveniente eliminarlos. También se recomienda no remover los valores atípicos cuando son muchos. Si se pueden graficar y se muestra una tendencia diferente debido a los *outliers*, es un error eliminarlos si son muchos.

En general, si son muchos datos atípicos, una recomendación que genera resultados positivos es hacer dos análisis. De otra forma, los datos atípicos, a pesar de ser válidos, modificarán la tendencia de los datos y, por lo tanto, los modelos de aprendizaje máquina estarán sesgados. Un ejemplo del tipo de segmentación de datos cuando se tienen más de un *outlier* en estos se muestra en la figura 7.1.

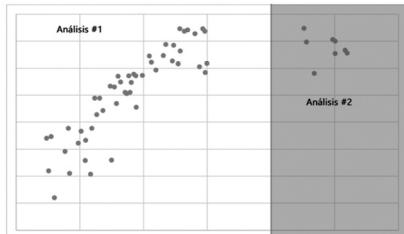


Figura 7.1. Consideración de análisis para datos con outliers

7.1.3. Escalar y normalizar

Existen muchas razones por las cuales se tiene que escalar/normalizar los datos previo a entrenarlos para generar un algoritmo de aprendizaje máquina. Sin embargo, como regla general, si el algoritmo que se quiere implementar calcula algún tipo de distancia o asume que los datos son normales, es conveniente escalar las características.

Algunos ejemplos de algoritmos donde conviene realizar un escalamiento previo son:

- **Algoritmos de clasificación como K-NN** –conocido como vecinos más cercanos–. En este tipo de algoritmos, se utiliza una medida de distancia para calcular los vecinos –generalmente, distancia euclídea, como se mostró en la sección 3.11.1.–, la cual es sensible a las magnitudes, por lo que es importante escalar sus características.
- **Algoritmos de reducción de dimensionalidad.** El escalamiento es importante en algoritmos como el análisis de componentes principales (PCA, por sus siglas en inglés). Esto es debido a que PCA se enfoca en obtener características con una varianza máxima y la varianza sería alta con características de una magnitud alta, lo que puede sesgar PCA hacia una magnitud más alta de lo que debería.
- **Algoritmos basados en gradiente descendente.** En general, se puede optimizar el proceso de gradiente descendente realizando un escalamiento. Esto se debe a que θ descenderá más rápidamente para rangos pequeños que en rangos grandes, por lo que ayuda el escalamiento. Cuando en este tipo de algoritmos se tienen rangos grandes θ , oscila ineficientemente y tomará más iteraciones llegar al óptimo.
- Algunos **métodos de regresión** que dependen de la escala de las características. Para algunos casos de regresión –como lo son regresión LASSO y «Kernel Ridge», entre otros–, los parámetros son influenciados por su magnitud, por lo que el escalamiento es importante en estos casos.

Por otro lado, algunos algoritmos no tendrán mucho efecto al ser escalados los datos de los parámetros de entrada. Por ejemplo, los modelos basados en árboles no son algoritmos donde se requieren distancias y pueden manejar varios rangos sus atributos. Por lo tanto, este tipo de algoritmos no requieren necesariamente escalamiento. Así mismo, algoritmos como Bayes ingenuo, LDA, entre otros, asigna pesos a las características, por lo que el escalamiento no ayudará mucho y no tendrá el efecto deseado, por lo que no se recomienda.

7.1.4. Lidiar con el ruido

Principalmente, para problemas con un número limitado de instancias de entrenamiento, tener un control del ruido que entra al sistema es crucial para el correcto desempeño del algoritmo.

Tener un sistema sin ruido es prácticamente imposible, pues el ruido del sistema es un problema ineludible. De manera general, el ruido puede tener dos orígenes: los errores implícitos introducidos por las herramientas de medición, como los sensores, sistemas de medición, entre otros, y los errores aleatorios introducidos por los procesos o personas que participaron en la obtención de datos.

Por lo tanto, la presencia de ruido en los datos puede afectar las características internas del problema a clasificar. Por ejemplo, el ruido puede derivar en la creación de pequeños *clusters* o agrupaciones de una clase en particular en una región de una clase diferente o sesgar el área de frontera entre un *cluster* y otro.

La frontera entre las clases y que se superpongan entre sí también son factores por considerar y que pueden ser afectados por la presencia de ruido. Todas estas alteraciones dificultan la extracción de conocimiento de los datos y pueden modificar los resultados cuando se entrena nodelos en los que se encuentra una gran cantidad de ruido.

Existen algunos otros tipos de ruido que pueden existir en los datos. Algunos de ellos son:

- **Ruido de etiquetado o ruido de clase.** En este caso, una instancia se etiqueta en una categoría diferente por error. El ruido de clase puede ser atribuido a diferentes causas, como la subjetividad cuando se realizó la clasificación, los errores de entrada de los datos, entre otros.
- **Errores de atributo.** Estos errores ocurren cuando no se sabe el valor del atributo y se trata de compensar de alguna manera –por ejemplo, métodos de imputación–. También hay ocasiones en que se tienen valores en los atributos de tipo «X», es decir, no importa. Estos valores fueron capturados así, pero un algoritmo de clasificación tiene problemas para ubicar estos valores en sus clases.

Por otro lado, existen ocasiones donde no se tiene ruido, pero, para efectos del entrenamiento y las pruebas del algoritmo, se tiene que simular el ruido. Esto, usualmente, se usa para poder mejorar la confiabilidad del algoritmo y que los datos no estén hechos «a modo». La generación del ruido posee tres características principales:

1. **El sitio donde el ruido es introducido.** El ruido puede afectar los atributos de entrada o la clase de salida, impidiendo el proceso correcto de aprendizaje y el modelo resultante.
2. **La distribución del ruido.** La manera en la que el ruido inducido o simulado requiere ser introducido debe de ser, por ejemplo, uniforme o gaussiana.
3. **La magnitud de los valores introducidos.** Los valores que se introduzcan de ruido simulado deben de ser acordes a los máximos, mínimos y desviación estándar de cada atributo.

Como se comentó anteriormente, en las bases de datos de la vida real la cantidad y tipo de ruido son generalmente desconocidos. Por lo tanto, no se puede asumir el tipo y cantidad de ruido cuando se tienen datos. Debido a esto, las bases de datos se considera que no tienen ruido, en el sentido de que no se puede saber dónde está el ruido.

Para poder crear un set de datos con ruido, se comienza con el set original y se consideran los factores que se mencionaron anteriormente. El esquema general es el siguiente:

- Un nivel de ruido (en %), ya sea tipo clase o atributo se introduce a la base de datos original.
- Ambos sets de los datos se partitionan en N diferentes partes, con el mismo número de instancias cada uno –parecido a lo que se hace para pruebas de validación cruzada (sección 5.12)–.
- Los sets se dividen en el número de particiones cada uno con una parte del ruido simulado, como se muestra en la figura 7.2.

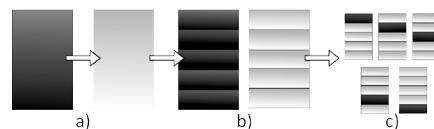


Figura 7.2. Simulación de ruido en bases de datos basado en k-fold

En la figura 7.2, se puede observar el proceso para la simulación de ruido en una base de datos siguiendo

los lineamientos mencionados previamente. En la figura 7.2 a), se observa que se crea una copia con ruido simulado de la base de datos. Posteriormente, se realizan las particiones homogéneas en ambas bases de datos –llamadas *folds*–. En este ejemplo, el número *k* de *folds* es cinco, como se muestran en la figura 7.2 b).

Por último, en la figura 7.2 c), se muestra que se hacen cinco copias de la base de datos de tal forma que cada partición de la base de datos se integre con la base de datos original. Así se tienen cinco bases de datos, cada una con ruido simulado que permite hacer las pruebas del modelo del algoritmo.

7.2. Velocidad de la predicción

Cuando se evalúan modelos en cuestión a su rendimiento, generalmente, solo se toma en cuenta la exactitud del modelo, aunque existen otros requerimientos que también deben de ser tomados en cuenta al evaluar el modelo.

Uno de estos requerimientos es la velocidad de la predicción. En este sentido, no hay una receta de cocina o un algoritmo que siempre sea más rápido en todos los sentidos que cualquier otro. Depende mucho de la aplicación, el número de instancias, el número de atributos, los parámetros con los que se entrene el algoritmo y la manera en la que el algoritmo se codificó.

En cuestión a la velocidad de predicción se debe revisar qué tan rápido puede un modelo realizar predicciones y qué tan rápido es crucial que las realice. Muchas veces, se puede sacrificar un poco la rapidez si no es tan crucial a cambio de una exactitud mayor. Por ejemplo, algunos tipos de regresión como la logística funcionan y realizan predicciones de manera rápida debido a que, teniendo la ecuación correspondiente y fijando un umbral, el algoritmo realiza el cálculo. Por otro lado, algoritmos como el de *k*-vecinos cercanos (KNN) es mucho más lento principalmente cuando se tienen muchas instancias o el número de *K*-vecinos es muy alto, pero puede compensar su exactitud a un número de *K*-vecinos alto con respecto al tiempo que tarda en calcular la distancia euclídea del punto a predecir con respecto a cada una de las instancias.

Si, por ejemplo, se requiere un modelo que detecte en tiempo real fraudes en un sistema de tarjetas de crédito, se requiere un sistema que realice probablemente miles de predicciones por segundo para realizar un modelo y determine desviaciones de este que puedan considerarse como fraude. En este ejemplo, un sistema que tenga una precisión muy alta, pero requiera mucho tiempo de modelado, no será viable comparado con un modelo fácil de usar y rápido, aunque no tan preciso.

7.3. Capacidad de reentrenamiento

En secciones previas, se comentó las diversas estrategias de un modelo para calcular su rendimiento. Sin embargo, en algunas ocasiones, su rendimiento es bajo o no ha alcanzado su punto de mayor exactitud, se tiene que cambiar alguno de sus hiperparámetros. Algunos modelos tienen la capacidad de reentrenamiento y se puede reentrenar un modelo relativamente fácil, mientras que otros conviene cambiar el algoritmo y volver a entrenar que tratar de reentrenar el nuevo modelo.

Por ejemplo, los algoritmos basados en probabilidad de Bayes y KNN son buenos ejemplos de algoritmos que pueden ser reentrenados relativamente fácil, mientras que árboles de decisión y algunos modelos de regresión son ejemplos de algoritmos que no es fácil reentrenar.

7.4. Configuración de hiperparámetros

Los hiperparámetros son importantes debido a que controlan directamente el comportamiento del entrenamiento y tienen un gran impacto en el rendimiento del modelo.

Seleccionar los hiperparámetros apropiados es fundamental para tener resultados satisfactorios cuando se trata de las topologías de redes neuronales artificiales. Por ejemplo, si la tasa de aprendizaje es muy baja, el modelo no detectará importantes patrones en los datos. De la misma manera, si la tasa de aprendizaje es muy alta, se tendrá muy probablemente un sobreaprendizaje.

Básicamente, los tipos de hiperparámetros pueden dividirse en dos:

- Hiperparámetros para optimizar el modelo.
- Hiperparámetros específicos del modelo.

Se recomienda ajustar los siguientes hiperparámetros para mejorar el rendimiento del modelo:

- **Ajustar el sesgo de entrada a un valor menor.** En general, el sesgo de entrada al nodo está fijo en un valor específico, ese valor es 1.0. El sesgo de entrada tiene el efecto de recorrer el valor de la función de activación y, en ocasiones, el valor es muy alto. Si se utiliza la función de activación ReLU, considera cambiar el sesgo a un valor más pequeño, como 0.1, después compara tus resultados con un sesgo predefinido como 1.0.
- **Inicialización de los pesos de la red.** Antes del entrenamiento de la red, los pesos de la red deben de ser inicializados. Esto, usualmente, se hace de manera aleatoria a valores pequeños. Se recomienda que la inicialización sea valores pequeños centrados a 0. De esta manera, la mitad de las neuronas en la primera época tendrán una salida de 0 y le darán oportunidad a la red de tener más épocas para poder ajustar los pesos necesarios. Hay otras maneras de inicializar los pesos. Una de ellas es utilizar valores aleatorios en un rango dado por $\pm 1/\sqrt{2/n}$ donde n es el número de nodos de la capa previa.
- **Escalamiento de datos de entrada.** En secciones previas, se mostró la utilidad de escalar –normalizar– los valores y la utilidad que esto generaba. En relación a las redes neuronales, es una buena práctica escalar los datos de entrar previo a iniciar el entrenamiento de la red. La normalización puede ser en el rango de 0 a 1 o de -1 a 1. Sin escalar los datos de entrada, el entrenamiento de la red puede dar muchos problemas. Por ejemplo, los pesos de la red pueden ser muy grandes, haciendo el entrenamiento muy inestable e incrementando el error de generalización.
- **Ajustar la tasa de aprendizaje.** Se recomienda que la tasa de aprendizaje se ajuste a 0.001. Si la tasa de aprendizaje es muy pequeña, el modelo tardaría miles –o cientos de miles– de épocas para llegar a un estado ideal. Por otro lado, si la tasa de aprendizaje es mucho más grande que el valor óptimo, el modelo sobreaprenderá o no convergerá.
- **Tamaño del lote.** También conocido como batch, tiene el efecto de tomar un número n de muestras para después calcular el error, retropropagar y ajustar los pesos. Es por esto que el tamaño del lote es importante. Si el batch es de 1, es muy posible que la muestra sea insuficiente para ajustar los pesos de manera correcta y tardará mucho el modelo en aprender o no aprenderá los patrones. Por el contrario, si el lote es muy grande, el modelo no aprenderá, pues tendrá muchos datos que tratar de memorizar por época, lo cual le será muy difícil a la red neuronal artificial. Se recomienda incrementar gradualmente el tamaño de lote en un factor de 2^n , partir de 2^1 , es decir: 2, 4, 8, 16, 32, 64, 128, 256, 512 y 1024. Un valor muy común para el lote es de 32, aunque no para todas las aplicaciones puede funcionar bien.
- **Número de épocas.** Para elegir de manera correcta el número de épocas –iteraciones– de un algoritmo basado en redes neuronales artificiales, se tiene que revisar el error de validación. La manera manual para saber el número de épocas correctas es seguir entrenando el modelo mientras que el error de validación siga decreciendo. La otra manera es mediante una técnica que se conoce como paro prematuro, en el que

se sigue entrenando hasta que el error de validación no ha mejorado por las últimas diez, quince o hasta veinte épocas.

Existen, además, métodos automáticos para encontrar la mejor configuración de hiperparámetros, comúnmente conocido como optimización de hiperparámetros. Los tipos algoritmos para optimización de hiperparámetros más comunes son:

- Búsqueda por cuadrícula o rejilla.
- Búsqueda aleatoria.
- Búsqueda heurística.
- Optimización bayesiana.

La búsqueda por cuadrícula (*grid*) es la técnica más común de optimizar hiperparámetros. Consiste en una especie de fuerza bruta que requiere crear dos sets de hiperparámetros:

1. La tasa de aprendizaje.
2. El número de capas.

Este tipo de búsqueda entrena al algoritmo para todas las combinaciones posibles con esos dos sets de hiperparámetros —la tasa de aprendizaje y el número de capas— y mide el rendimiento utilizando la técnica de validación cruzada (*k-fold*). Esta técnica de optimización de hiperparámetros es relativamente fácil de implementar, pero suele tardar mucho, pues busca cada combinación de hiperparámetros y compara los resultados de cada combinación.

Por otro lado, la búsqueda aleatoria evalúa la cantidad de sets de una distribución de probabilidad. En lugar de tratar todos los hiperparámetros y checar cientos de miles de combinaciones, se asigna un porcentaje y solo realiza la combinación de, por ejemplo, un 10 % del total de posibles combinaciones aleatorias.

Esto tiene un inconveniente, como la búsqueda es aleatoria, no utiliza la información y resultados de anteriores experimentos para seleccionar el siguiente set, por lo tanto, no siempre es la mejor estrategia para optimizar los parámetros.

La búsqueda heurística emplea técnicas como algoritmos genéticos, programación genética, entre otros. Consiste en determinar el espacio de búsqueda y buscar los parámetros que pudieran servir para optimizar los hiperparámetros. De esta manera, se puede en pocas iteraciones —se conoce usualmente como generaciones— llegar a buenos resultados sin agotar todo el espacio de búsqueda. Esto tiene un gran inconveniente: cuando se trabaja con algoritmos genéticos, se requiere de una función de aptitud para poder determinar si la búsqueda heurística está llegando a un óptimo. En el caso de modelar los hiperparámetros, no se puede hacer mediante una ecuación, lo que se conoce como ajuste mediante función en blanco.

El utilizar estas técnicas heurísticas contravienen uno de los principios fundamentales de los algoritmos genéticos, por lo que se navega a «ciegas» entre los parámetros para encontrar mejores soluciones y no se sabe si se llegó a un óptimo o se detuvo el algoritmo de manera prematura.

Para contrarrestar este problema de técnicas heurísticas de optimización, se utilizan usualmente lo que se conoce como optimización bayesiana. La optimización bayesiana utiliza una función de aproximación que, usualmente, es el proceso gaussiano, el cual consiste en mantener la distribución actual de la función y estimar la distribución posterior cuando los resultados se vayan conociendo.

El proceso gaussiano utiliza una matriz de covarianza para asegurarse de que los valores de los parámetros están cerca entre sí.

7.5. Seleccionar el algoritmo adecuado

Existen diferentes tipos de algoritmos de aprendizaje máquina, por lo que es importante saber qué se va a elegir de acuerdo a distintas consideraciones como ya se ha comentado, pues no todos los algoritmos son para todas las funciones que se quieran realizar.

Por ejemplo, en ocasiones, se requiere de un algoritmo que muestre visualmente resultados con pocas instancias y pocos atributos, o, en ocasiones, se requiere entrenar para encontrar soluciones, o reducir la dimensionalidad o encontrar un valor por lo que se requiere un algoritmo de regresión.

En este sentido, si se requiere entrenar un algoritmo para aprender, encontrar soluciones, reconocer patrones, clasificar datos y predecir eventos futuros, usualmente, se utilizan las redes neuronales. Existen muchos tipos de redes neuronales, cada una con su topología –estructura interna– y sus ventajas y desventajas. El comportamiento de una red neuronal está definido de acuerdo a cómo sus entradas, interconexiones, número de capas, número de neuronas por capa y números de salidas están diseñadas, así como también a la fuerza de sus conexiones –llamadas pesos– y cómo esos pesos se ajustan automáticamente. La forma genérica de una red neuronal se muestra en la figura 7.3.

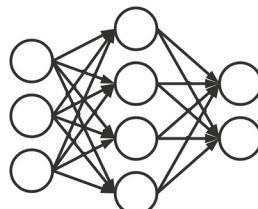


Figura 7.3. Forma genérica de una red neuronal

Para programadores más experimentados, las redes neuronales son muy buenas en modelar datos no-lineales con un gran número de entradas o de neuronas en las distintas capas. Las redes neuronales artificiales (ANN, por sus siglas en inglés), también son muy utilizadas para resolver problemas que son muy complejas para un algoritmo simple. Sin embargo, en general, las ANN son costosas a nivel computacional –poca velocidad de predicción– y es difícil entenderlas una vez que tienen una solución. Se les ve en general, como una gran caja negra que da un resultado y no se entiende del todo cómo se llegó a él. Otra de las desventajas de las redes neuronales es que los ajustes para maximizar el potencial y disminuir el error de las redes neuronales no siempre es tan fácil de realizar.

Por otro lado, el clasificador bayesiano ingenuo («Naive Bayes», por su término en inglés) se utiliza principalmente cuando los datos no son complejos y la tarea a clasificar es relativamente simple. Es un clasificador de baja varianza por lo que presenta ventajas con respecto a la regresión logística y los algoritmos basados en vecinos o criterios de distancia (como KNN) cuando se tienen datos disponibles limitados para entrenar un modelo.

El clasificador bayesiano ingenuo también es una buena opción cuando se tienen recursos computacionales, ya sea procesamiento o memoria bastante limitados. Debido a que Bayes ingenuo es muy simple, no tiende a sobreentrenar datos y puede ser entrenado relativamente rápido.

Sin embargo, si el número de datos crece en tamaño y varianza y se necesita un modelo más complejo, seguramente, este no será el mejor modelo por utilizar. En general, es la primera opción cuando se trabaja con texto –como filtros de correo no deseado, análisis de sentimiento, etcétera–. El funcionamiento de manera general se puede observar en la figura 7.4.

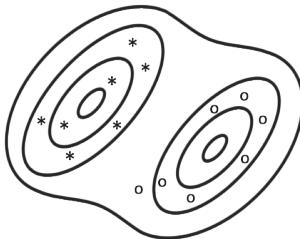


Figura 7.4. Forma de un clasificador tipo «Naive Bayes»

Los árboles de decisión son métodos muy sencillos, como se vio con anterioridad, además de ser métodos gráficos. Para seguir la decisión de un árbol, basta con seguir el árbol desde la raíz hacia abajo, hasta el nodo hoja que contiene la respuesta, por lo que es un método fácil de seguir. Los árboles de decisión para clasificación dan respuestas que son nominales, es decir, que son cierto o falso –sí o no, 0 o 1, etc.–, mientras que los árboles de decisión para regresión dan respuestas numéricas.

La principal desventaja de los árboles de decisión es que tienden a sobreentrenar, aunque hay métodos de ensamble para poder contrarrestar esto. La figura 7.5 muestra genéricamente un árbol de decisión.

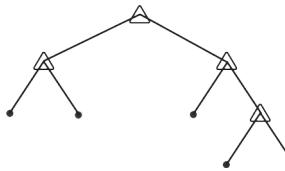


Figura 7.5. Forma genérica de un árbol de decisión

Existen otros métodos como los basados en similitud –o distancia–, como el k-vecinos cercanos (o KNN, por sus siglas en inglés). Esta manera de clasificar los datos por similitud o distancia es una manera simple, pero efectiva, y k-vecinos cercanos hace exactamente eso.

KNN es un algoritmo de aprendizaje en el cual realmente no se requiere de una fase intensiva de entrenamiento, por lo que es muy fácil de implementar. Solo se requiere de cargar los datos de entrenamiento y encontrar la k adecuada, es decir, el número óptimo de vecinos cercanos y buscar de cada punto de prueba el número k de vecinos y, con esa información, determinar de acuerdo a la cercanía qué clasificación tienen los puntos de prueba.

Aunque dicho entrenamiento es corto, las pruebas pueden ser largas principalmente si son muchas instancias, pues tiene que calcular la distancia del punto a todos y cada uno, ordenarlos por distancia y, posteriormente, tomar solo el número k de vecinos más cercanos, y, con esa información, determinar su clasificación, por lo que puede representar una serie de cálculos extensivos.

El principal problema de KNN es que no siempre los vecinos determinan la clase, sino otro tipo de atributos, y si no se eligen los atributos adecuados, esto puede llevar a que la clasificación sea errónea. En la figura 7.6 se muestra un ejemplo gráfico de KNN.

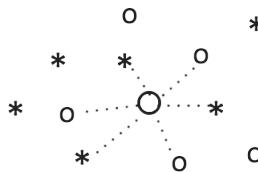


Figura 7.6. Forma del algoritmo K-vecinos cercanos

Otro de los algoritmos que están siendo muy usados es el de las máquinas de soporte de vectores (SVM, por sus siglas en inglés). Los SVM son útiles cuando los datos tienen dos clases, aunque se tengan muchas dimensiones. Los SVM clasifican los datos y encuentran el mejor hiperplano que separa todos los puntos de

una clase con respecto a la otra. El mejor hiperplano de un SVM es aquel con el más grande margen entre dos clases.

Para crear un SVM, entre más clases se tengan, el modelo necesita ser categorizado en subproblemas binarios, con un SVM para cada subproblema. Los SVM tienen varias ventajas, primeramente, el modelo puede tener una exactitud muy alta y no tiende a sobreentrenar datos.

Otra de las ventajas que tienen los SVM es que son relativamente fáciles de interpretar y tienden a ser modelos rápidos una vez que se entiende el funcionamiento de un SVM y son muy buenos cuando se tienen datos no-lineales que clasificar. Una de las desventajas de los SVM es que toma su tiempo crear e implementar el modelo. También si se tienen más de dos clases, la complejidad y, por lo tanto, la velocidad se afectan mucho. La figura 7.7 muestra una representación genérica de los SVM, mientras que la figura 7.8 muestra una representación de algunos algoritmos que existen de acuerdo al tipo de datos que se tienen o lo que se quiere realizar.

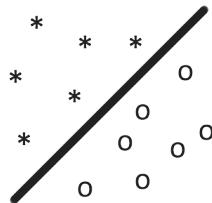


Figura 7.7. Forma del algoritmo máquina de soporte de vectores

Como se ha mostrado, existen muchos tipos de algoritmos, cada uno con características diferentes. En la figura 7.8, se muestran solo algunos de ellos agrupados de la siguiente manera: regresión, regularización, reducción de dimensionalidad, árboles de decisión, agrupación, ensamble, redes neuronales. (En esta figura, solo se incluyeron las redes profundas).

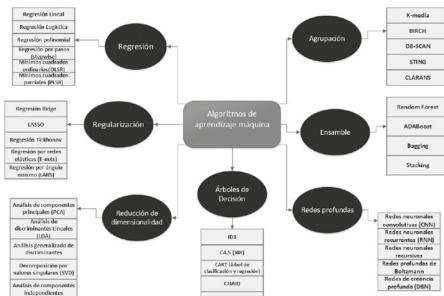


Figura 7.8. Tipos de algoritmos de aprendizaje máquina

7.6. Elección de la función de activación

La elección de la función de activación es una parte crucial para el correcto diseño de una red neuronal artificial.

La correcta elección de la función de activación en las capas ocultas determinará qué tan bien la red aprende los datos de entrenamiento, mientras que la elección de la función de activación en la capa de salida define el tipo de predicciones que hará el modelo.

Como se comentó anteriormente, una red neuronal artificial puede tener tres tipos de capas: la capa de entrada, las capas ocultas, que toman la entrada de otra capa y la pasan a la salida hacia la siguiente capa, y, por último, la capa de salida.

Aunque las redes neuronales fueron diseñadas para usar la misma función de activación, pues son usadas por cada neurona para un procesamiento interno, es muy común que, dentro de una red profunda, existan

diferentes funciones de activación.

Típicamente, todas las capas ocultas suelen usar la misma función de activación mientras que la capa de salida puede usar la misma o usar una diferente, dependiendo del tipo de predicción que el modelo requiera.

Aunque existen muchos tipos de funciones de activación, mayoritariamente, solo se utilizan algunas como de escalón, sigmoide, de tangente hiperbólica, softmax y de rectificación lineal –conocida comúnmente como ReLU–.

Una capa oculta en una red neuronal artificial es una capa que recibe entradas de otra capa –muchas veces, otra capa oculta– y provee una salida que va hacia otra capa –otra capa oculta o la capa de salida–. Las redes neuronales artificiales pueden tener cero, una o más capas ocultas.

La función más simple es la función de activación lineal. En esta función de activación, no se requiere ninguna transformación, por lo que la implementación de redes con este tipo de funciones de activación es bastante simple. Actualmente, las redes profundas, normalmente, no tienen este tipo de funciones de activación en sus capas intermedias, pero para realizar, por ejemplo, una regresión, siguen siendo muy usadas en la capa de salida.

Las funciones de tipo no lineales son preferibles y de mucho mejor estructura para tratar de mejor manera la compleja topología de redes de muchas capas. En este tipo de funciones de activación, se encuentran la sigmoide y la tangente hiperbólica.

La función de activación tipo sigmoide también se le conoce como función logística y es muy popular. En esta función, la entrada es transformada en un valor entre 0.0 y 1.0. De esta forma, valores de entrada muchos más grandes que 1.0 se transforman a 1.0, lo mismo que valores mucho menores a 0.0 se ajustan a 0. Esto puede representar un problema cuando se tienen escalas muy dispares en datos de entrada, esta es una de las razones por las cuales se recomienda escalar –normalizar– los datos de entrada cuando se trabaja con redes neuronales, aunque depende del tipo de aplicación. La forma de la función sigmoide se asemeja a una «S», como se puede ver en la figura de la sección 6.3.2.

Por otro lado, la función de tangente hiperbólica –usualmente conocida como tanh– es relativamente similar su función de activación con respecto a la sigmoide, con excepción que los valores de salida los escala entre -1.0 y 1.0. Hace años, esta función comenzó a reemplazar para casi todas las topologías de red a la sigmoide porque ofrecía un mejor valor predictivo y era fácil de entrenar. En la actualidad, la mayoría de las topologías de red tipo recurrente siguen usando esta función de activación.

Como se comentó, un problema que en general sucede tanto con la función sigmoide como la tanh es que se saturan. Esto significa que, si tiene valores o muy grandes o muy pequeños, las funciones son relativamente sensibles a cambios muy pequeños, pues todo lo escala a esos valores. Debido a esta sensibilidad, una vez que la función se satura, es muy difícil que el algoritmo continúe aprendiendo y adaptando los pesos de los nodos para mejorar el rendimiento del modelo.

Probablemente, la función de activación más común es la ReLU. Esta red es muy simple de implementar, pues depende de una función lineal que manda a la salida el valor máximo entre 0.0 y la salida de la anterior neurona. Es decir, $\max(0.0, x)$.

Es importante determinar en qué ocasiones se recomienda utilizar ReLU o alguna otra función de activación para asegurar un mayor rendimiento. En este sentido, ReLU funciona mejor en la mayoría de los casos y es más conveniente para realizar un entrenamiento.

Se recomienda utilizar la función de activación ReLU para una aplicación donde se utilice un perceptrón multicapas (MLP) y las redes convolutivas (CNN), pues, en general, muestran mejores resultados y son más rápidas para entrenar que las funciones sigmoide y tanh. En otros tipos de redes neuronales artificiales, se recomienda comenzar con ReLU y, si se requiere, cambiar de función de activación. Probablemente, la única

excepción para esto sea las redes tipo recurrente (Elman, LSTM, GRU). En este caso, debido a que propaga los valores de la anterior capa, lo más recomendable es utilizar una función de activación de tangente hiperbólica.

En suma, si se requiere hacer una clasificación, la función de activación que mejor va a funcionar es la siguiente: para múltiples etiquetas sigmoide, para multiclase: softmax, y para clasificación binaria: sigmoide. Si, por otro lado, se tiene un problema de regresión, la función de activación lineal funcionará bien y es simple de implementar. Para redes convolutivas, usualmente, va a funcionar mejor ReLU. En este tipo de redes, la capa de salida es usualmente softmax.

7.7. Mejorar la exactitud y el rendimiento del algoritmo

Además de las estrategias para lidiar con los factores que afectan el rendimiento de un algoritmo que se mencionaron con anterioridad, existen algunos puntos que se pueden revisar para mejorar la exactitud y el rendimiento de un algoritmo de inteligencia artificial.

Existen diferentes estrategias para mejorar el rendimiento de un algoritmo, algunas de ellas son:

- Mejorar el rendimiento con datos.
- Mejorar el rendimiento con otro algoritmo.
- Mejorar el rendimiento con ajustes y configuraciones.
- Mejorar el rendimiento con ensambles.

7.7.1. Mejorar el rendimiento mediante datos

En anteriores apartados ya se ha hablado de limpieza de datos, escalamiento, entre otros. Este tipo de estrategias, usualmente, sirve para obtener mejores resultados con el algoritmo seleccionado de aprendizaje máquina.



Tip

En ocasiones, es preferible crear diferentes perspectivas de los datos para poder entender la estructura de los mismos. Por ejemplo, saber si se requieren más datos, ya sea que se puedan recolectar o crear datos nuevos con una distribución igual o muy parecida (esto se conoce como datos sintéticos).

Existen diversas maneras por las cuales, cambiando la perspectiva de los datos, se puede obtener mejor calidad de los datos y, por ende, mejorar el rendimiento del algoritmo. Por ejemplo: el uso de datos sintéticos puede ayudar cuando se tiene pocas instancias de prueba o las clases no están balanceadas. Eso quiere decir que se tienen muchas menos instancias de una clase que de las demás.

Existen diversos métodos para generar datos sintéticos, uno de ellos es aumentar el número de instancias permutando los datos existentes utilizando modelos probabilísticos para generar nuevos datos sin sesgar la distribución del atributo.

Otra de las estrategias para mejorar el rendimiento del algoritmo mediante datos es la limpieza de estos. Si existen valores corruptos, valores faltantes o atípicos hay que identificarlos. Así mismo, los valores atípicos pueden ser valores válidos o inválidos y, de acuerdo a la distribución del dato y el conocimiento de las características del atributo, se puede determinar si los datos atípicos son válidos o inválidos.

También puede ser una buena estrategia reestructurar el problema y, con esto, reestructurar los datos que se tienen disponibles. Por ejemplo, hay que hacerse las preguntas: ¿se puede cambiar el tamaño o la

distribución de los datos? ¿Se puede cambiar el tipo de predicción que se quiere hacer?

En ocasiones, el volver a muestrear o intentar el algoritmo con una muestra representativa puede funcionar mejor que con toda la base de datos. Hay que tener cuidado con cuáles datos se trabaja si se hace un muestreo. En general, se toma una muestra de manera aleatoria por toda la base de datos de manera que la distribución de los datos no cambie –o cambie lo menos posible–.

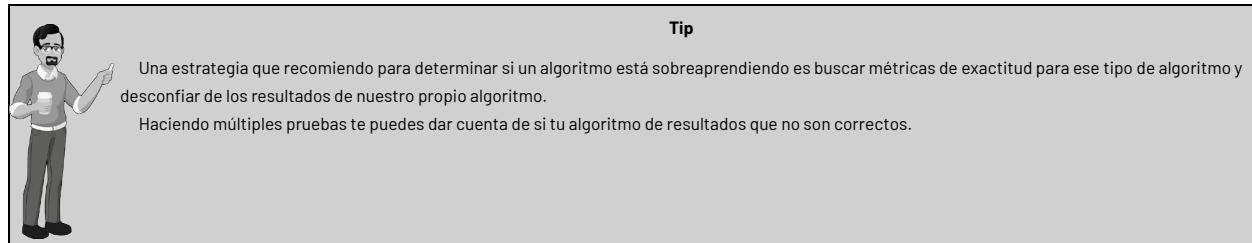
En cuestión del tipo de predicción que se puede hacer, se tiene que observar si es posible reestructurar los datos para tratar un problema como regresión, clasificación multiclase, binarizarlo o manejo de series de tiempo. Esto puede funcionar bien en algunas ocasiones para obtener mejores predicciones.

Otra de las estrategias para mejorar el rendimiento con respecto a los datos es revisar si se puede proyectar los datos. Esto es, cuando se trabaja con bases de datos que tienen muchos atributos, no se sabe a ciencia cierta cuáles atributos son más importantes para trabajar con ellos ni si se requiere de todos los atributos para la predicción que se está tratando de implementar.

En este caso, se pueden proyectar los datos para reducir la dimensionalidad. Los algoritmos PCA y LDA son buenos para esto. Otra estrategia es calcular la correlación que tienen los atributos mediante algoritmos, como, por ejemplo, el coeficiente de correlación de Pearson. También se puede calcular la entropía total del sistema y la ganancia de información como se maneja en árboles de decisión para saber qué atributos tienen más información que otros y, así, elegir únicamente los que sirven más al algoritmo.

7.7.2. Mejorar el rendimiento mediante algoritmos

El aprendizaje máquina se basa en algoritmos. Es por ello que, cuando se trabaje con algoritmos, se deben tener estrategias muy claras sobre entrenamiento, validación, pruebas.



Una de las estrategias para mejorar el rendimiento del algoritmo es evaluar los métodos de validación y prueba que se están usando. Es muy común que el algoritmo no muestre un desempeño aceptable y que sea debido al método de validación y pruebas. Se recomienda realizar varios métodos de prueba y varias pruebas por cada método para tener una idea más clara del rendimiento del algoritmo.

Cuando se utiliza un método *k-fold*, cada prueba de cinco particiones –en el caso de *5-fold*– debe ser repetida con un orden diferente de los datos. Esto asegura que el algoritmo no se «aprende» los datos de prueba en algunas ocasiones, sino que, en realidad, el algoritmo puede generalizar bien.

También es recomendable sacar provecho de falsos negativos y de verdaderos negativos. Si se usan estas métricas, se puede determinar no solo la exactitud del modelo, sino cuáles clases está mal clasificando mediante una matriz de confusión, como se mostró en la sección 5.10 «Evaluación del modelo».

Además de la evaluación cuando se trabaja con los algoritmos, es recomendable revisar el tipo de algoritmos con los que los datos pueden funcionar con una alta tasa de clasificación. ¿Se puede resolver con algoritmos lineales como algunos tipos de regresión, alguna red neuronal simple como un perceptrón? Usualmente, los métodos lineales o simples son más fáciles de entender y también de entrenar. Si se obtienen buenos resultados, es preferible usar estos métodos.

En otras ocasiones, los métodos lineales no son suficientes, o los datos son altamente no lineales o se tienen demasiadas instancias, por lo que se requieren otros métodos más complejos. Si este es el caso, se

evalúa primeramente qué algoritmos están disponibles en la literatura para implementarlos para este tipo de aplicaciones.

Otra estrategia que se puede seguir una vez que se ha elegido el algoritmo es el de evaluar las configuraciones estándares, es decir, los hiperparámetros que se van a utilizar. Para cada aplicación existe en la literatura las configuraciones que se utilizan y muchas de esas veces son las mejores que se pueden aplicar para el problema en cuestión.

El modificar las configuraciones hasta que se encuentra la ideal es una de las etapas que más pueden causar dolores de cabeza y problemas en aprendizaje máquina. Es muy útil revisar las curvas que se pueden generar con el algoritmo. Sugiero hacerse las siguientes preguntas con respecto a esto:

- ¿La tasa de aprendizaje va subiendo poco a poco o sube muy rápido?
- ¿El aprendizaje de validación se va dispersando de la curva de aprendizaje de prueba?
- ¿La exactitud del modelo varía mucho? Es decir, en algunas pruebas se tiene, por ejemplo 95 % y en otras 50 %. Si es así, es muy probable que la especificidad sea muy baja también y los resultados sean muy dependientes de los datos de prueba o no se estén haciendo las pruebas correctas.

Esto da una idea de lo que está pasando con el algoritmo. Por un lado, pueden ser muy dependientes los resultados de las pruebas o pueden tener una mala generalización, por otro lado, puede estar sobreaprendiendo y, cuando el algoritmo tiene datos de prueba diferentes, no reconoce sus patrones.

Otro de los puntos que puede ayudar a mejorar el rendimiento de un algoritmo es la optimización. En muchas ocasiones, se pueden optimizar los parámetros de entrada o se puede realizar una optimización estocástica —como algoritmos genéticos o algún algoritmo de inteligencia de enjambre como ACO, PSO, BFOA, entre otros—.

7.7.3. Mejorar el rendimiento mediante algoritmos de ensamble

Una de las mejores estrategias para mejorar el rendimiento de un algoritmo es la de añadir un algoritmo de ensamble, que es una de las estrategias del metaaprendizaje.

Los algoritmos de ensamble básicamente combinan las predicciones de múltiples modelos. De hecho, usualmente, es más recomendable combinar las predicciones de varios modelos que realicen una buena estimación que tener un solo algoritmo al que ya le cambié muchas veces su configuración interna hasta que me dio el mejor posible resultado.

En este sentido, hay que saber identificar si se pueden combinar las predicciones directamente del mismo algoritmo o tienen que ser de diferentes. En ocasiones, basta con tomar la media o la moda de los modelos que dieron buenas predicciones.

Otra estrategia que usualmente funciona bien es combinar modelos con diferentes configuraciones y elegir el que para esos datos de entrenamiento dio el mejor resultado. Otra estrategia que funciona bien y que es muy utilizada es crear diferentes muestras de los datos, entrenarlos por separado y combinar las predicciones. A este algoritmo se le conoce usualmente como *bagging*.

Otra estrategia que funciona bien es utilizar modelos para corregir errores de predicción. Este método se conoce como *boosting* y, usualmente, funciona bien cuando se realizan pruebas en las que, en ocasiones, el error de predicción es pequeño y, en ocasiones, incrementa considerablemente debido a los datos de entrenamiento.

Por último, un algoritmo que se utiliza para poder generalizar mejor las predicciones se conoce como *stacking*. Con este algoritmo, se pueden tener submodelos que funcionan bien de diferentes maneras y, simplemente, el modelo combina los mejores resultados de cada etapa o submodelo.

CAPÍTULO 8

Prácticas de IA en Python

En este capítulo, se abordarán una serie de prácticas para exemplificar los conceptos que se desarrollaron en el presente libro. Como en cualquier ambiente de programación, las librerías, funciones y hasta los algoritmos pueden ser implementados de manera diferente. Lo que se muestra aquí son ejemplos simples de implementar y entender utilizando bases de datos disponibles de manera gratuita en diferentes medios y con diferentes tipos de implementaciones para mostrar una mayor versatilidad con el código.

Las prácticas varían en complejidad y contenido. Se muestra, por ejemplo, los diferentes métodos de normalización, cálculos de métricas de distancia, obtener relaciones entre atributos y diversos errores que los datos pueden tener. Todos estos algoritmos son parte de la metodología en su etapa de preparación de datos.

Así mismo, cuando se realiza la preparación, como parte del análisis, se puede realizar diversos procesos como reducción de dimensionalidad –con análisis de componentes principales–, creación de datos sintéticos en caso de que no se tengan suficientes instancias para realizar la prueba, técnicas de imputación múltiple como MICE o imputación por regresión. Este algoritmo de imputación por regresión puede ser utilizado también como un algoritmo para determinar patrones por sí mismo.

En cuanto a los algoritmos, una vez que se preparan y analizan los datos, se presentan algunos algoritmos simples, pero que, a la vez, pudieran ser de utilidad para aprendizaje máquina. Entre ellos se encuentran: descendiente al gradiente, k-medias, KNN, árboles de decisión por ID3 y bosques aleatorios.

En lo que respecta a las pruebas que deben de ser realizadas por los algoritmos para conocer la certeza con la cual el algoritmo aprende, se incluye un ejercicio de validación cruzada, que es uno de los métodos más utilizados en aprendizaje máquina.

Por otro lado, se incluyeron algunas prácticas de una de las técnicas que más se utilizan en muchas aplicaciones: las redes neuronales artificiales. En este sentido, se incluyeron algoritmos de topologías variadas, que pueden ser utilizadas con distintos fines. Por ejemplo, un perceptrón simple y uno multicapa, conocido como MLP, para entender más a fondo sobre las topologías de redes neuronales, una red convolutiva que es muy utilizada para aplicaciones donde se requiere la predicción o clasificación de imágenes y, por último, una red profunda recurrente (RNN), la cual es muy utilizada para datos de dominio de tiempo –de tipo continuo–.

De esta forma, se muestra de una manera más clara la manera de implementar los conceptos mostrados en el presente libro.

Por último, me parece importante mencionar que se debe de mantener actualizadas las versiones de las librerías con las que se trabajan en el presente libro y revisar la documentación de estas. Pueden actualizar las librerías con el siguiente código de la ventana de comandos:

```
python -m pip install -U matplotlib
```

Se puede reemplazar la librería –matplotlib, en este caso– con la librería que se deseé actualizar o instalar.

8.1. Normalización

Dentro de los pasos más importantes en el preprocesamiento de los datos se encuentra el de normalización. En este proceso, se realiza una transformación a los datos que permite dimensionar la magnitud de los datos a un valor determinado.

Una de las ventajas que tiene la normalización de los datos es que, en ocasiones, su uso puede mejorar el resultado de modelos de aprendizaje automático.

El código se muestra en esta sección y puede ser descargado en: http://www.amese.net/libro_ia/Prog/

Para iniciar con esta sección, importaremos las librerías que utilizaremos.

Código:

```
import random
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

La librería **random** nos servirá para crear el conjunto de datos al que aplicaremos diversos tipos de normalización. La librería **numpy**, para obtener algunos datos estadísticos de manera más sencilla, por último, **seaborn** y **matplotlib.pyplot**, para graficar nuestros resultados y observar gráficamente las diferencias entre los métodos de normalización que abordaremos.

Iniciaremos por crear una función para cada uno de los métodos de normalización que abordaremos:

- Normalización min-max.
- Normalización z-score.
- Normalización por medias.
- Normalización for vector unitario.

8.1.1. Min-max

En el caso de **min-max**, utilizaremos la fórmula que permite obtener el vector de salida en el rango que el usuario prefiera, a diferencia de la fórmula simple, que entrega un vector con valores entre 0 y 1.

La normalización min-max se muestra en la ecuación 8.1

$$x' = a + \frac{x - \min(x)}{\max(x) - \min(x)} * (b - a) \quad (8.1)$$

Donde:

- **x** es el dato original.
- **x'** es el dato normalizado.
- **a** es el valor mínimo deseado.
- **b** es el valor máximo deseado.

Para implementarlo, se tiene el siguiente código:

Código:

```
def minmax_normalization(x, lowLimit=0, highLimit=1):
    minX=min(x)
    maxX=max(x)
    return [((xi-minX) / (maxX-minX)) * (highLimit-lowLimit) + lowLimit for xi in x]
```

Podemos observar que nuestra función **minmax_normalization** recibe como argumento el vector de datos **x**. Adicionalmente, tiene los argumentos **lowLimit** y **highLimit** que tienen, por defecto, el valor de 0 y 1 correspondientemente. Esto es debido a que, si no se especifica este intervalo, se asume que se quiere una normalización simple a 0 y 1. De igual manera, se da la opción a que se obtenga el vector normalizado en el intervalo que el usuario requiera.

8.1.2. Z-score

Ahora, la normalización **z-score** tiene la característica de que su vector normalizado de salida tiene una media de 0 y desviación estándar de 1. Este método de normalización ha demostrado tener un efecto positivo en los algoritmos de aprendizaje automático, por lo cual es ampliamente utilizada.

La ecuación para obtenerla se muestra en la ecuación 8.2:

$$x' = \frac{x - \mu}{\sigma} \quad (8.2)$$

Donde:

- **x** es el dato original.
- **x'** es el dato normalizado.
- μ es la media del vector de entrada.
- σ es la desviación estándar de x.

El código para implementar esta normalización es el siguiente:

Código:

```
def zscore_normalization(x):
    m=np.mean(x)
    s=np.std(x)
    return [(xi-m)/s for xi in x]
```

8.1.3. Normalización por medias

Este método de normalización es similar al de **z-score**, solo cambia en denominador de la ecuación, como puede mostrarse en la ecuación 8.3:

$$x' = \frac{x - \mu}{\max(x) - \min(x)} \quad (8.3)$$

Donde:

- **x** es el dato original.
- **x'** es el dato normalizado.
- μ es la media del vector de entrada.

Código:

```
def mean_normalization(x):
    m=np.mean(x)
    minX=np.min(x)
    maxX=np.max(x)
    return [(xi-m) / (maxX-minX) for xi in x]
```

Más adelante, revisaremos algunas otras características de este método comparado con los demás.

8.1.4. Normalización por vector unitario

El último método por revisar se conoce como **unit vector**, aquí se normaliza cada elemento del vector de entrada con base a la distancia euclidiana del vector.

Se usa la siguiente ecuación:

$$x' = \frac{x}{\|x\|} \quad (8.4)$$

Donde:

- x es el dato original.
- x' es el dato normalizado.
- $\|x\|$ es la distancia euclidiana del vector de entrada x .

Código:

```
def unit_vector(x):  
    unitVector=0  
    for xi in x:  
        unitVector+=xi**2  
    unitVector=unitVector**(1/2)  
    return [xi/unitVector for xi in x]
```

Aquí obtenemos un vector de salida con valores muy pequeños, su característica es que cada elemento es tomado como una componente para crear un vector unitario, es decir, un vector con magnitud de uno. Se puede comprobar al obtener la magnitud del vector normalizado, realizaremos este proceso más adelante.

8.1.5. Comparación entre métodos

Una vez que hemos revisado distintos métodos de normalización, procedamos a revisar de una forma más dinámica sus características.

Para iniciar, crearemos un conjunto de datos con valores en el intervalo $[0,10]$ con quinientos datos.

Código:

```
# random.random() genera un valor aleatorio entre 0 y 1.  
x=[random.random()*10 for _ in range(500)]
```

Con la línea de código anterior, generamos el conjunto de datos deseado, multiplicamos por diez debido a que queremos que nuestro rango sea $[0,10]$.

Ahora procederemos a obtener la normalización de nuestro conjunto de datos x para cada uno de los métodos revisados.

Código:

```
x_mm=minmax_normalization(x)  
x_mm2=minmax_normalization(x, lowLimit=-5, highLimit=6)  
x_z=zscore_normalization(x)  
x_m=mean_normalization(x)  
x_u=unit_vector(x)
```

Lo primero que haremos es revisar algunas estadísticas de nuestros datos normalizados. Para reducir el código y poder utilizar un ciclo **for**, guardaremos los conjuntos normalizados en una lista llamada **data** que

contendrá los datos y otra lista con los nombres correspondientes, llamada **names**.

También desplegaremos un **distplot** de la librería **seaborn**, este gráfico muestra un histograma de los datos junto con una gráfica de densidad de probabilidad. Y, finalmente, un **boxplot** con los cinco vectores normalizados.

Código:

```
names=['MinMax', 'MinMax2', 'Z-score', 'Mean Norm', 'Unit Vector']
data=[x_mm, x_mm2, x_z, x_m, x_u]

for index in range(len(names)):
    print(names[index])
    print('Min:', np.min(data[index]), 'Max:', np.max(data[index]),
          'Media:', np.mean(data[index]), 'Std:', np.std(data[index]))
    sns.distplot(data[index])
    plt.show()
print('Comparación de datos')
plt.boxplot(data, labels=names)
plt.show()
```

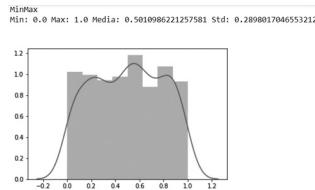


Figura 8.1. Distribución de datos con la normalización min-max

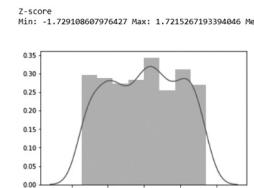


Figura 8.2. Distribución de datos con la normalización z-score

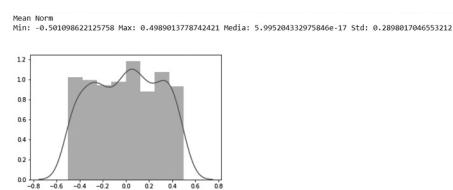


Figura 8.3. Distribución de datos con la normalización por medias

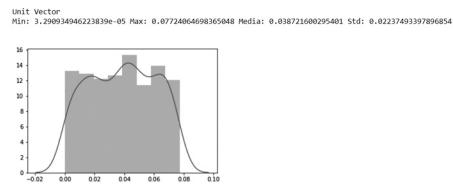


Figura 8.4. Distribución de datos con la normalización por vector unitario

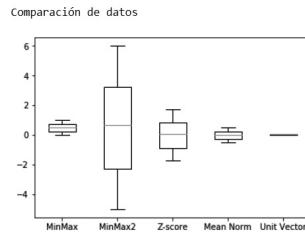


Figura 8.5. Comparación de distribución de datos utilizando normalización

Del resultado anterior, podemos observar que todas las gráficas muestran una misma distribución, simplemente, cambian los valores en el eje x en los que oscilan los datos.

En el último gráfico, se puede ver claramente la diferencia en la magnitud de los datos. Los datos de **min-max** oscilan entre 0 y 1, como es esperado. **Min-max2** oscila entre los valores ingresados -5 y 6. Y esto se repite sucesivamente con cada método.

Ahora, grafiquemos cada conjunto normalizado junto con tres líneas, la primera en -1, la segunda en 0 y la tercera en 1. Esto nos permitirá ver algunas características de cada función.

Código:

```
for index in range(5):
    print(names[index])
    plt.plot([0,500],[1,1], 'r')
    plt.plot([0,500],[-1,-1], 'r')
    plt.plot([0,500],[0,0], 'g')
    plt.plot(data[index], '*')
    plt.show()
```

Da como resultado las siguientes figuras:

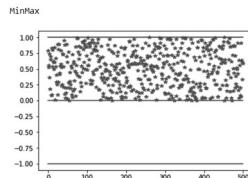


Figura 8.6. Conjunto de datos normalizado con min-max

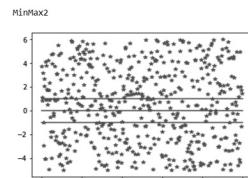


Figura 8.7. Conjunto de datos normalizado con min-max2

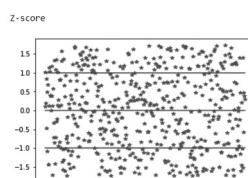


Figura 8.8. Conjunto de datos normalizado con z-score

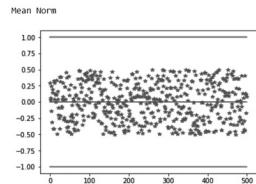


Figura 8.9. Conjunto de datos normalizado con normalización por medias

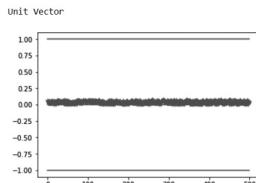


Figura 8.10. Conjunto de datos normalizado con normalización por vector unitario

En cuanto al rango, se puede validar lo siguiente:

- **Min-max.** Se encuentra en el rango esperado de 0 y no existen valores negativos.
- **Min-max2.** Tiene una misma distribución que el resto de los datos, pero en el intervalo deseado. Da la posibilidad a contar con valores negativos.
- **Z-score.** Los datos están distribuidos uniformemente alrededor del valor 0. Los valores de -1 y 1 también están en una zona donde la distribución de los datos es cercana a estos valores, por lo cual puede verse gráficamente un dato que ya conocíamos, la desviación estándar es 1.
- **Normalización por medias.** Aun cuando este tipo de normalización se calcula de manera similar a z-score, el vector resultante tiene magnitudes menores. Se observa que también tiene valores negativos.
- **Normalización por vector unitario.** Cuenta con valores muy pequeños, comparado con los demás métodos. No cuenta con valores negativos.

8.2. Cálculo de distancia euclíadiana

Dentro de los métodos de medición de distancia, la distancia euclíadiana es, posiblemente, la más conocida y utilizada de todas. Esta distancia es la más corta que existe entre dos puntos dentro de un espacio n-dimensional, esto se puede calcular mediante el teorema de Pitágoras. Recordemos que, cuando hablamos de dimensiones en inteligencia artificial, nos referimos a la cantidad de atributos que podemos representar en un conjunto de datos.

El código se muestra en esta sección y puede ser descargado en:

http://www.amese.net/libro_ia/Prog/8.2_Euclidian.ipynb

Para visualizarlo de una mejor manera, utilizaremos el conjunto de datos de **gorriones** —cuya explicación se encuentra en el apéndice A.3—. Iniciemos con un ejemplo en un plano bidimensional.

Código:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
df=pd.read_csv('database/gorriones.csv')
df.head()
```

	X1	X2	X3	X4	X5
0	166	245	31.6	18.5	20.5
1	154	240	30.4	17.9	19.6
2	153	240	31.0	18.4	20.6
3	153	238	30.9	17.7	20.2
4	155	243	31.5	18.6	20.3

Figura 8.11. Conjunto de datos de gorriones

En el código anterior, leemos y desplegamos una muestra del conjunto de datos, como se muestra en la figura 8.11. Podemos ver que tenemos cinco dimensiones disponibles para este ejercicio, tomaremos las primeras dos. Revisemos los puntos disponibles mediante una gráfica **scatter** para ver su distribución, usamos el método **annotate** de la librería **matplotlib** para indicar el índice de cada punto graficado.

Código:

```
ejeX=df['X1']
ejeY=df['X2']
index=np.arange(0,ejeX.shape[0])

plt.scatter(ejeX, ejeY)
```

```

for ind in index:
    plt.annotate(ind, (ejeX[ind], ejeY[ind]))

```

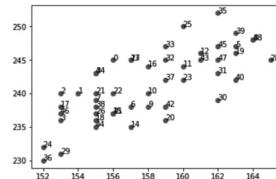


Figura 8.12. Datos e índices de la base de datos de gorriones X1 vs X2

Seleccionemos dos puntos que estén muy cercanos, tomaremos el punto 36 y el 29. Los guardaremos, respectivamente, en las variables **p1** y **p2**.

Código:

```

p1=[ejeX[36],ejeY[36]]
p2=[ejeX[29],ejeY[29]]

```

Ahora, crearemos dos funciones que nos permitan calcular la distancia euclíadiana y que sea desplegada de una forma visual, donde podamos ver sus componentes en el eje X y el eje Y. Creamos la primera función **dist_euc** que se encargará únicamente de entregar la distancia euclíadiana. Posteriormente, la función **distancia_euclidiana** usará la función **dist_euc** y también mostrará de forma visual las componentes.

Código:

```

def dist_euc(p1, p2, dec=4):
    acumulador=0
    for index in range(len(p1)):
        acumulador+= (p1[index]-p2[index])**2
    return np.round(acumulador** (1/2) ,dec)

```

Aplicaremos primero la función **dist_euc** a los puntos **p1** y **p2** para ver el resultado. En la siguiente línea, aplicaremos la función **distancia_euclidiana**. Observemos las diferencias, ventajas y desventajas. Observa que el punto de inicio es de color verde y el punto final es de color rojo.

Código:

```

def distancia_euclidiana(p1, p2, dec=4):
    if len(p1)>2:
        distancia=dist_euc(p1[:2], p2[:2], dec)
    else:
        distancia=dist_euc(p1, p2, dec)
    # Calculamos las diferencias en cada eje
    diferenciaX=p2[0]-p1[0]
    diferenciaY=p2[1]-p1[1]
    # Creamos 2 listas, una para cada eje, que contengan las componentes de los 3 puntos (inicio, avance en X, avance en Y)
    x=[p1[0], p1[0]+diferenciaX, p2[0]]
    y=[p1[1], p1[1], p1[1]+diferenciaY]
    # Graficamos los componentes
    plt.plot(x,y, marker='*')
    # Desplegamos las magnitudes de las componentes
    plt.annotate(diferenciaX, (p1[0]+(diferenciaX/2), p1[1]))

```

```

plt.annotate(diferenciaY, (p2[0], p1[1]+(diferenciaY/2)))
# Hipotenusa
x=[p1[0],p2[0]]
y=[p1[1],p2[1]]
# Graficamos la hipotenusa
plt.plot(x, y)
plt.annotate(distancia, (p1[0]+(diferenciaX/2), p1[1]+(diferenciaY/2)))
# Marcamos el origen y final
plt.plot(p1[0],p1[1], 'g.', markersize=20)
plt.plot(p2[0],p2[1], 'r.', markersize=20)
return distancia

```

Código:

```

d1=dist_euc(p1, p2)
print(d1)

```

1.4142

Código:

```

d1=distancia_euclidiana(p1, p2)
print(d1)

```

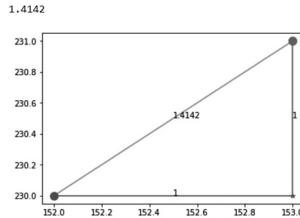


Figura 8.13. Resultado de distancia euclidiana

Como podemos ver, el resultado desplegado es idéntico, como es esperado. En realidad, puedes utilizar la primera función si solo quieres obtener el valor. Por otra parte, si prefieres tener un apoyo visual de la distancia, es preferible usar la segunda función.

Nota. La función **distancia_euclidiana** solo te mostrará una visualización 2D de los puntos, por su parte, la función **dist_euc** es más general y aplica para puntos n-dimensionales.

Elijamos otro par de puntos, esta vez lejanos, para observar el comportamiento de nuestras funciones. Por ejemplo, 24 y 35. Esta vez invertiremos los puntos en la función **distancia_euclidiana**. Observa el resultado.

Código:

```

p1=[ejeX[24], ejeY[24]]
p2=[ejeX[35], ejeY[35]]

d1=dist_euc(p1, p2)
print("Distancia 1:", d1)
d2=distancia_euclidiana(p2, p1)
print("Distancia 2:", d2)

```

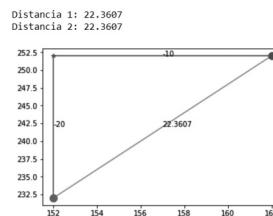


Figura 8.14. Resultado del segundo cálculo de la distancia euclídea

Como es de esperarse, la magnitud de la hipotenusa es idéntica, simplemente, cambiaron las componentes debido a que elegimos como punto de inicio el **p2** y como punto final el **p1**.

Por último, evaluemos puntos con dimensiones superiores, utilicemos las cinco dimensiones con las que cuenta nuestro conjunto de datos. Esta vez elegiremos el índice 0 y 1.

Código:

```
p1=df.values[0,:]
p2=df.values[1,:]
print("Punto 1:", p1)
print("Punto 2:", p2)
d1=dist_euc(p1, p2)
print("Distancia 1:", d1)
d2=distancia_euclidiana(p1, p2)
print("Distancia 2:", d2)
```

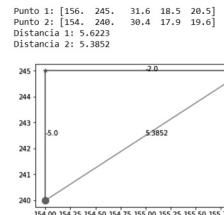


Figura 8.15. Resultado tres de la distancia euclídea

Podemos ver que ahora las distancias calculadas son distintas, esto es debido a que en la función **dist_euc** hacemos el cálculo en n-dimensiones de la distancia. Por su parte, en la función **distancia_euclidiana** solo utilizamos dos dimensiones para el cálculo, ya que tenemos que limitarnos a las dimensiones del plano para mostrar un resultado correcto.

Ahora tenemos dos funciones que nos permiten hacer el cálculo de la distancia euclídea entre dos puntos para darles la aplicación que nos convenga. Realicemos una última función que nos permite ver lo sencillo que es escalar un programa que se realiza de manera general. Crearemos una función que nos muestre gráficamente la distancia y componentes desde un origen a un conjunto de puntos.

Código:

```
def distancia_euclidiana_n_puntos(origen, puntos):
    for p in puntos:
        distancia_euclidiana(origen, p)
```

Elegimos como origen el índice 10 y para nuestro conjunto de puntos elegimos cuatro índices arbitrarios.

Código:

```
origen=[ejeX[10], ejeY[10]]
index=[0, 30, 35, 44]
puntos=df.values[index,:]
```

```
distanzia_euclidiana_n_puntos(origen, puntos)
```

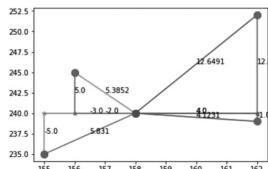


Figura 8.16. Distancia euclidiana para varios índices simultáneos

El código que aquí revisaste te sirve como base para que puedas adecuarlo a implementaciones propias que requieras. Probablemente, para aplicaciones de inteligencia artificial te sea de mayor utilidad la función **dist_euc**. Te comarto una función de la librería **scipy** que funciona de la misma manera y la compararemos la función que acabamos de implementar.

Código:

```
from scipy.spatial import distance

p1=[ejeX[36],ejeY[36]]
p2=[ejeX[29],ejeY[29]]
print("Ejemplo con puntos 2D")
print("Función Creada: ", dist_euc(p1, p2, dec=16))
print("Función de scipy:", distance.euclidean(p1, p2), end='\n\n')

p1=df.values[0,:]
p2=df.values[1,:]

print("Ejemplo con puntos 5D")
print("Función Creada:", dist_euc(p1, p2, dec=14))
print("Función de scipy:", distance.euclidean(p1, p2))
```

```
Ejemplo con puntos 2D
Función Creada: 1.4142135623730951
Función de scipy: 1.4142135623730951

Ejemplo con puntos 5D
Función Creada : 5.62227711874824
Función de scipy: 5.62227711874824
```

Se puede ver que los resultados son idénticos. Puedes utilizar la función que tú quieras en tus implementaciones, por medio de estos ejercicios has revisado una implementación desde cero para cálculo de distancia euclidiana.

8.3. Cálculo de distancia Manhattan

La distancia Manhattan es la distancia que hay en las ciudades, ya que hay que rodear los edificios para poder llegar a nuestro destino, no podemos recorrer la ciudad siempre en una línea recta, a menos que podamos atravesar edificios.

El código se muestra en esta sección y puede ser descargado en:
http://www.amese.net/libro_ia/Prog/8.

Para mostrar cómo funciona esta distancia, veamos un ejemplo. Para esto ocuparemos una imagen que representa las manzanas de una parte de una ciudad, en formato png.

Para abrir la imagen utilizaremos, el módulo **image** de la librería **matplotlib**. Para poder leer la imagen, el archivo «Manhattan.png» se encuentra en el mismo directorio. La función **imread** carga la imagen en la libreta como se muestra en el siguiente código:

Código:

```
import matplotlib.image as mpimg
import matplotlib.pyplot as plt

# Leemos la imagen
mapa = mpimg.imread(r'Manhattan.png')
# Mostramos la imagen
plt.imshow(mapa)
```

La imagen se muestra en la figura 8.17.

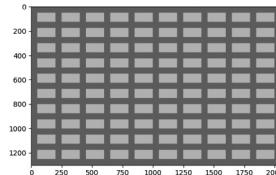


Figura 8.17. Cuadrícula para representar la distancia Manhattan

Ahora vamos a definir dos puntos: p1 y p2, punto inicial y punto final, y vamos a graficarlos:

Código:

```
import matplotlib.image as mpimg
import matplotlib.pyplot as plt

# Leemos la imagen
mapa = mpimg.imread(r'Manhattan.png')
# Mostramos la imagen
plt.imshow(mapa)

p1, p2 = [625, 150], [1825, 900]
# Graficamos el punto de origen (p1)
plt.plot(p1[0], p1[1], marker='o', color='k')
# Graficamos el punto de llegada (p2)
plt.plot(p2[0], p2[1], marker='o', color='k')
```

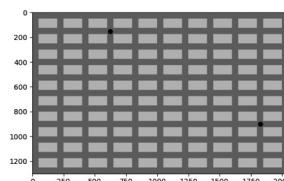


Figura 8.18. Cuadrícula para representar la distancia Manhattan mostrando puntos de inicio y de fin

Y dibujemos las trayectorias que se forman en el eje Y y en el eje X:

Código:

```
import matplotlib.image as mpimg
import matplotlib.pyplot as plt

# Leemos la imagen
mapa = mpimg.imread(r'Manhattan.png')
# Mostramos la imagen
plt.imshow(mapa)

p1, p2 = [625, 150], [1825, 900]
# Dibujamos la distancia vertical
```

```

plt.plot([p1[0], p2[0]], [p2[1], p2[1]], color='r')
# Dibujamos la distancia horizontal
plt.plot([p1[0], p1[0]], [p1[1], p2[1]], color='r')
# Graficamos el punto de origen (p1)
plt.plot(p1[0], p1[1], marker='o', color='k')
# Graficamos el punto de llegada (p2)
plt.plot(p2[0], p2[1], marker='o', color='k')

```

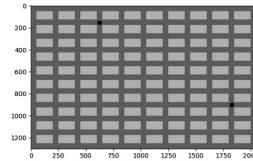


Figura 8.19. Cuadrícula para representar la distancia Manhattan con puntos de inicio y de fin

En el código anterior, vemos dos cosas: primero, el eje Y está invertido, y, segundo, tenemos valores que van de 0 a 2025 y de 0 a 1275 en el eje X y en el eje Y, respectivamente. Aunque estos valores estén bien porque son los píxeles que tiene la imagen que abrimos, no es una muy buena manera de visualizar la distancia Manhattan. Así que vamos a invertir el eje Y y a escalar los ejes para apreciar mejor la distancia Manhattan. Son diez manzanas a lo largo y ancho por lo que vamos a dividir los ejes en números del 0 al 10. Hacer estos cambios requerirá reescribir la mayor parte del código que habíamos hecho, como se muestra en la figura 8.20.

Código:

```

import matplotlib.image as mpimg
import matplotlib.pyplot as plt

p1, p2 = [3, 2], [9, 9]
# Escalamos las coordenadas en X
x1, x2 = 200*p1[0] + 25, 200*p2[0] + 25
# Escalamos las coordenadas en Y
y1, y2 = 125*p1[1] + 25, 125*p2[1] + 25
# Leemos la imagen
mapa = mpimg.imread(r'Manhattan.png')
# Mostramos la imagen
plt.imshow(mapa)
# Dibujamos la distancia vertical
plt.plot([x1, x2], [y2, y2], color='r')
# Dibujamos la distancia horizontal
plt.plot([x1, x1], [y1, y2], color='r')
# Graficamos el punto de origen (p1)
plt.plot(x1, y1, marker='o', color='k')
# Graficamos el punto de llegada (p2)
plt.plot(x2, y2, marker='o', color='k')
# Escalamos el eje X
plt.xticks([200 * x + 25 for x in range(11)],
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '10'])
# Escalamos el eje Y
plt.yticks([125 * y + 25 for y in range(11)],

```

```
[ '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '10' ])
# Invertimos el eje Y
plt.gca().invert_yaxis()
```

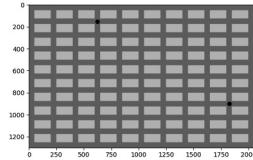


Figura 8.20. Cálculo de distancia Manhattan por número de píxeles

Vamos a empezar redefiniendo los puntos **p1** y **p2**, como pretendemos tener una escala de 0 a 10 en ambos ejes, no tiene sentido que tengamos un punto con coordenadas (625, 150), por ejemplo. Lo siguiente es sacar la escala, en esta imagen, los cuadros que representan las manzanas de la ciudad son de 150 px de largo por 75 px de ancho, y las «avenidas» son de 50 px de ancho, por lo que, para que nos podamos posicionar en el origen y que sea un punto medio de la avenida, deberíamos posicionarnos en 25 px, de esta forma, el origen (0, 0) corresponde a (25, 25).

Después, para avanzar avenidas en el eje Y, debemos avanzar los 75 px del bloque, más 50 px de una avenida, más los 25 px iniciales. Para el eje X, lo único que cambia es el tamaño del bloque, que es de 150 px, por lo que la ecuación queda de la siguiente forma: $x = (150+50)x_1 + 25$, simplificando, $x = 200x_1 + 25$, donde x_1 es el número de avenidas que se tienen que avanzar.

Con estas relaciones, se debe reescribir las coordenadas en el código –recordemos que la posición 0 de una lista corresponde a nuestra coordenada en X y la posición 1 a la coordenada en Y– como se muestra, después, abrimos la imagen, dibujamos la distancia vertical y horizontal, graficamos **p1** y **p2**, dividimos los ejes X e Y utilizando las funciones **xticks** y **yticks**, respectivamente, ambas funciones reciben como parámetros dos listas. En la primera, va la manera en que se va a dividir el eje y, en la segunda, las etiquetas que va a tener el eje, para las divisiones podemos utilizar las ecuaciones planteadas anteriormente, ambas funciones pertenecen a la librería **matplotlib**, y, finalmente, invertimos el eje Y, con la función **invert_yaxis()** de la clase **axes**, a la cual podemos acceder a través de la función **gca()** de **matplotlib**.

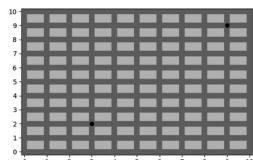


Figura 8.21. Cálculo de distancia Manhattan con dimensiones de cuadros

La figura 8.21 es más entendible, pero aún no sabemos la distancia entre esos dos puntos, creemos una función que nos calcule la distancia Manhattan, dados dos puntos.

Código:

```
def dist_manhattan(p1, p2):
    distancia = 0
    for i in range(len(p1)):
        distancia += abs(p1[i]-p2[i])
    return distancia
```

En esta función recibimos dos parámetros, que son **p1** y **p2**, declaramos **distancia = 0**, para ir sumando las distancias en este objeto y usando un *for*, que se ejecutará tantas veces como elementos tenga **p1**, vamos

sumando la distancia que hay entre la **x** y la **y**, esa distancia la sumamos y guardamos en **distancia**, y, finalmente, devolvemos **distancia**.

Hemos creado la función, pero tenemos que visualizar la distancia de algún modo. En este caso, vamos a visualizarla haciendo uso de la función **title**, para que la veamos como título de la imagen.

Código:

```
# Llamamos la función y la convertimos en string
dist = str(dist_manhattan(p1, p2))

# Título de la figura
plt.title('Distancia Manhattan\n')

# Ponemos el valor de la distancia como título
plt.title('Distancia: ' + dist, loc='right')
```

Como la función **title** solo recibe cadenas, es decir, *strings*, al momento de llamar a la función **dist_manhattan()** hay que convertir el valor que devuelve en *string* y guardarlo en otra variable. Después, ponemos el título de la gráfica, que es, «Distancia Manhattan», y para la distancia vamos a poner: «Distancia» y le vamos a añadir el valor en *string* de la distancia calculada, con el operador **+**, finalmente, con el parámetro **loc**, decidimos dónde colocar el título, en este caso, lo pusimos a la derecha, como se muestra en la figura 8.22.

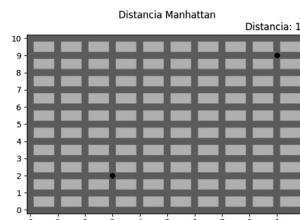


Figura 8.22. Cálculo de la distancia Manhattan con resultado de la distancia

8.4. Cálculo de distancia Chebyshev

La distancia Chebyshev es la distancia máxima, ya sea en el eje X o en el eje Y. Esta distancia coincide con el mínimo número de movimientos que un rey ocupa para moverse de una casilla a otra en un tablero de ajedrez.

El código se muestra en esta sección y puede ser descargado en:
http://www.amese.net/libro_ia/Prog/8.

Vamos a dar un ejemplo: creamos la función para calcular la distancia Chebyshev primero, que será la función **dist_chebyshev()**, aquí sacamos la diferencia entre la **x** y la **y** y devolvemos la distancia mayor.

Código:

```
# Función que calcula la distancia Chebyshev dados dos puntos.
# parámetro p1: punto inicial, vector de dos dimensiones.
# parámetro p2: punto final, vector de dos dimensiones.

import matplotlib.pyplot as plt
import matplotlib.image as mpimg

def dist_chebyshev(p1, p2):
    # Sacamos la diferencia para «x» y para «y»
    x, y = abs(p1[0] - p2[0]), abs(p1[1] - p2[1])
```

```

# Devolvemos la distancia mayor
if x > y:
    return x
else:
    return y
print(dist_chebyshev([0,1], [0,12]))
print(dist_chebyshev([3,6], [9,7]))

```

Como observamos, la distancia devuelta por la función es la distancia máxima.

Procedamos a abrir la imagen, utilizaremos las mismas funciones que utilizamos en la distancia Manhattan. La imagen se muestra en la figura 8.23.

Código

```

def con_escala(p1, p2):
    # Función que grafica la distancia Chebyshev en un tablero de ajedrez, con
    # coordenadas escaladas.

    # parametro p1: punto inicial, vector de dos dimensiones, con valores en [0,8].
    # parametro p2: punto final, vector de dos dimensiones, con valores en [0,8].

    # Leemos la imagen
    tablero = mpimg.imread(r'tableroAjedrez.png')
    # Mostramos la imagen
    plt.imshow(tablero)

```

La imagen se encuentra en: http://www.amese.net/libro_ia/Prog/tableroAjedrez.png

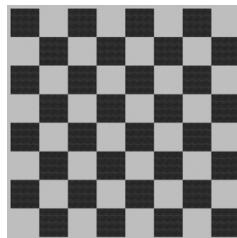


Figura 8.23. Tablero de ajedrez para la distancia Chebyshev

En este caso, los cuadros del tablero son de 200 px por 200 px y el margen es de 25 px. Usaremos el mismo procedimiento que utilizamos en la distancia Manhattan para escalar los ejes y para invertir el eje Y. La única diferencia es que, en vez de movernos por las «avenidas» como en la distancia Manhattan, nos vamos a mover por todos los cuadrados, así, debemos ver el tablero como una matriz de 8x8.

Los *for* anidados los usamos para obtener las coordenadas de cada cuadro y anotar la distancia con respecto de la posición inicial **p1**.

En el punto de llegada, utilizamos como marcador a un 'rey de ajedrez', podemos usar símbolos en código utf-8, para hacer esto hay que poner '\$' al inicio y al final del código utf-8. Para este caso, el código del símbolo del rey es 265A, para que Python lea código utf-8 hay que agregarle '\u' al inicio del código, entonces, nos quedaría así: '\u265A', y para poderlo utilizar como marcador: '\$\u265A\$'.

Código:

```

def con_escala(p1, p2):
    # Función que grafica la distancia Chebyshev en un tablero de ajedrez, con
    # coordenadas escaladas.

```

```

# parametro p1: punto inicial, vector de dos dimensiones, con valores en [0,8].
# parametro p2: punto final, vector de dos dimensiones, con valores en [0,8].

# Leemos la imagen
tablero = mpimg.imread(r'tableroAjedrez.png')

# Mostramos la imagen
plt.imshow(tablero)

# Escalamos las coordenadas en X
x1, x2 = 200 * p1[0] + 125, 200 * p2[0] + 125
# Escalamos las coordenadas en Y
y1, y2 = 200 * p1[1] + 125, 200 * p2[1] + 125

# Anotamos la distancia para cada cuadro con respecto a la posición del rey
# Posición en X
for i in range(8):
    # Posición en Y
    for j in range(8):
        # Llamamos a la función dist_chebyshev y la convertimos a string
        # para obtener la distancia con respecto de la posición inicial
        # del rey
        dist = str(dist_chebyshev(p1, [i, j]))
        # escalamos las coordenadas
        x, y = 200 * i + 125, 200 * j + 125
        # Anotamos las coordenadas
        plt.annotate(dist, (x, y), fontsize=11, color='k')

# Graficamos el punto de origen (p1)
plt.plot(x1, y1, marker='o', color='r')
# Graficamos el punto de llegada (p2)
plt.plot(x2, y2, marker='$\u265A$', color='k', markersize=25)
# Escalamos el eje X
plt.xticks([200 * x + 125 for x in range(8)],
['0', '1', '2', '3', '4', '5', '6', '7'])
# Escalamos el eje Y
plt.yticks([200 * y + 125 for y in range(8)],
['0', '1', '2', '3', '4', '5', '6', '7'])
# Invertimos el eje Y
plt.gca().invert_yaxis()

dist = str(dist_chebyshev(p1, p2))
# Escribimos el título de la gráfica
plt.title('Distancia Chebyshev\n')
# Ponemos la distancia entre p1 y p2 como un título
plt.title('Distancia: ' + dist, loc='right')

if __name__ == '__main__':
    p1, p2 = [1, 2], [7, 6]
    con_escala(p1, p2)
    plt.show()

```

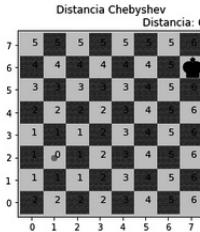


Figura 8.24. Distancia Chebyshev mostrando el tablero de ajedrez y el rey

Haciendo un poco más general nuestro código, se puede utilizar para abrir cualquier imagen y medir la distancia Chebyshev entre dos puntos.

Para realizar esto, podemos crear una función que reciba como parámetros la ubicación de la imagen a abrir (*imagen*), el punto inicial (*p1*), el punto final (*p2*) y la escala (*escala*), siendo este último parámetro opcional con un valor por defecto de 1.

Abrimos la imagen, verificamos que la escala sea un número positivo, invertimos el eje Y, obtenemos los límites del eje X con la función *xlim*, la cual nos devuelve el límite inferior y el límite superior y los guardamos en *xlim_inf* y *xlim_sup* respectivamente, hacemos lo mismo para el eje Y, pero con la función *ylim*, y guardamos los límites en *ylim_inf* y *ylim_sup*.

Los límites superiores *xlim_sup* y *ylim_sup* corresponden a las dimensiones de nuestra imagen, por lo que serán nuestros parámetros al momento de realizar las divisiones de los ejes. Para realizar las divisiones de los ejes, necesitamos escalar el límite superior primero, entonces dividimos *xlim_sup* entre *escala* y obtenemos nuestro límite escalado.

Haremos, como máximo, diez divisiones en cada eje, encontrando los valores óptimos de las divisiones que, además de ser los correctos de acuerdo a la escala proporcionada, serán de fácil visualización. Esto lo logramos creando un ciclo para el eje X –y otro para el eje Y–, donde se almacenará la magnitud de cada división, va aumentando de valor de acuerdo a la dimensión de nuestra imagen. Una vez encontrado un valor adecuado para las divisiones, procedemos a calcular las divisiones que se harán en el eje, utilizando la variable *rango* que almacenará el número de divisiones que caben en el eje.

Con los datos X y rango, podemos crear las divisiones del eje X, llamando a la función *xticks*, donde mandamos la magnitud de nuestras divisiones (X) multiplicado por cada valor entero entre 0 y rango multiplicado por la escala. Recordemos que *xticks* maneja un valor en píxeles, por lo que hay que regresar los valores escalados a los valores reales multiplicando por la escala.

Para las etiquetas de las divisiones es casi el mismo proceso, solo que esta vez redondeamos nuestro valor a dos decimales y no multiplicamos por la escala, y con esto queda nuestro eje X escalado. Para el eje Y hacemos lo mismo que hicimos para el eje X, solo que ahora utilizaremos a *ylim_sup* en vez de *xlim_sup*. Ya con los ejes escalados, lo siguiente será escalar las coordenadas de los puntos *p1* y *p2*, calcular la distancia Chebyshev y mostrarla en la imagen junto con el título y la escala de esta. Como paso final, mostraremos la cuadrícula de la imagen para darle más legibilidad.

Pensar en términos de píxeles es más difícil que en términos de centímetros, metros, pulgadas o cualquier unidad de longitud que estemos acostumbrados a usar, es por ello que escalar la imagen es útil, porque podemos convertir los píxeles a otra unidad si sabemos la equivalencia. Por ejemplo, en la imagen que utilizamos para este ejemplo, pusimos una escala de 1:100 porque cada 100 px son 5 cm en nuestra imagen, entonces tenemos una distancia de 16.5 cm y ya no una distancia de 330 px. Esto se muestra en el siguiente código:

```
Código:  
def aplicacion(imagen, p1, p2, escala=1):
```

```

# Leemos la imagen
img = mpimg.imread(imagen)
# Mostramos la imagen
plt.imshow(img)
# Comprobamos que la escala sea mayor a cero
if escala <= 0:
    escala = 1

# Invertimos el eje Y
plt.gca().invert_yaxis()
# Obtenemos los límites de los ejes
xlim_inf, xlim_sup = plt.xlim()
ylim_inf, ylim_sup = plt.ylim()
# Escalamos el eje X
# Ciclo para saber de cuánto será cada división
X = 0
xlim_escalado = xlim_sup / escala
while X < xlim_escalado / 10:
    if xlim_escalado > 0 and xlim_escalado < 100:
        X += 1
    elif xlim_escalado >= 100 and xlim_escalado < 500:
        X += 5
    elif xlim_escalado >= 500 and xlim_escalado < 1000:
        X += 10
    elif xlim_escalado >= 1000:
        X += 50
    rango = 0
    while (X * rango) < xlim_escalado:
        rango += 1
    plt.xticks(
        [X * escala * x for x in range(0, rango)],
        labels=[round(X * x, 2) for x in range(0, rango)],
    )
# Escalamos el eje Y
# Ciclo para saber de cuánto será cada división
Y = 0
ylim_escalado = ylim_sup / escala
while Y < ylim_escalado / 10:
    if ylim_escalado > 0 and ylim_escalado < 100:
        Y += 1
    elif ylim_escalado >= 100 and ylim_escalado < 500:
        Y += 5
    elif ylim_escalado >= 500 and ylim_escalado < 1000:
        Y += 10
    elif ylim_escalado >= 1000:
        Y += 50
    # Ciclo para saber cuántas divisiones habrá
    rango = 0

```

```

while (Y * rango) < ylim_escalado:
    rango += 1
plt.yticks(
    [Y * escala * y for y in range(0, rango)],
    labels=[round(Y * y, 2) for y in range(0, rango)],
)
# Escalamos las coordenadas de los puntos en X
x1, x2 = escala * p1[0], escala * p2[0]
# Escalamos las coordenadas de los puntos en Y
y1, y2 = escala * p1[1], escala * p2[1]

# Graficamos el punto de origen (p1)
plt.plot(x1, y1, marker="o", color="r")
# Graficamos el punto de llegada (p2)
plt.plot(x2, y2, marker="o", color="k")

dist = str(round(dist_chebyshev(p1, p2), 2))
# Escribimos el título de la gráfica
plt.title("Distancia Chebyshev\n")
# Ponemos la distancia entre p1 y p2 como un título
plt.title("Distancia: " + dist, loc="right")
# Ponemos la escala
esca = str(escala)
plt.title("Escala: 1:" + esca, loc="left")
plt.grid()

```

Ahora procedemos a abrir la imagen, dar los puntos y dar la escala para nuestra imagen. En este caso, abriremos la imagen de un perro, la cual se encuentra en: http://www.amese.net/libro_ia/Prog/ y se muestra en la figura 8.25.

Código:

```

# Seleccionamos la ubicación del archivo, si está en el
# mismo directorio, basta con poner el nombre del archivo
imagen = "perro.jpg"
# Mandamos llamar a la función
aplicacion(imagen, [4, 9.3], [10.5, 5.7], 100)

```



Figura 8.25. Cálculo de la distancia para una imagen ejemplo

8.5. Relaciones entre atributos

Como ya se ha mencionado, es importante poder comparar y visualizar datos entre sí. Para ello, Python cuenta con varias herramientas que nos son útiles para este fin. En este ejemplo, utilizaremos el conjunto de

datos gorriones y vamos a ver la relación que guardan los atributos entre los **supervivientes** de los **no supervivientes**, el cual se explica a mayor detalle en el apéndice A.3.

El código se muestra en esta sección y puede ser descargado en:
http://www.amese.net/libro_ia/Prog/8.

Lo primero que haremos será abrir la ubicación del archivo que vayamos a graficar.

8.5.1. Gráfica de scatter en 3D

Para realizar una gráfica de dispersión, Python cuenta con la función **scatter**, esta función gráfica en 2D por defecto, aunque se pueden modificar algunas propiedades de esta, utilizando funciones modificadoras como la función **gca**.

Código:

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

archivo = input("Ubicación del archivo: ")
try:
    df = pd.read_csv(archivo)
    nombre_columna = []
    for dato in df.columns:
        nombre_columna.append(dato)
    if len(nombre_columna) < 4:
        raise Exception("Tiene menos de 3 columnas.")
except Exception as e:
    print("Archivo no válido.")
    print(e.__class__.__name__, e)
```

Pedimos al usuario que ingrese la ubicación del archivo a abrir y comprobamos mediante un **try/except** que el archivo sea válido y que tenga al menos tres columnas porque, de otro modo, no sería posible hacer una gráfica en 3D.

Ubicación del archivo: Gorriones.csv

En seguida, desplegamos el nombre de las columnas para poder elegir por consola las tres columnas que queremos graficar.

Para esto, crearemos una lista vacía donde guardaremos los nombres de las columnas a graficar, pero antes de guardarlas verificaremos que las columnas se encuentren en el archivo y que, además, sean numéricas, así que primero filtraremos las columnas que son numéricas con la ayuda de la función **select_dtypes**, de la librería pandas, misma que recibirá como parámetro el objeto **np.number**, con esto estaremos seleccionando solo columnas numéricas, luego, mediante un ciclo *while* anidado dentro de un ciclo *for* se comprobará si existe la columna, si la columna no se encuentra en el archivo se notificará al usuario y no se podrá salir del ciclo hasta que se ingresen las columnas correctas.

Una vez ingresadas las columnas es importante comprobar que las tres tengan el mismo número de datos, esto lo haremos con un *if*, en caso de que no tengan el mismo número de datos, levantaremos una excepción.

Código:

```

# Desplegamos las columnas que hay en el archivo
print("Columnas:")
# Filtramos las columnas numéricas
columnas_numericas = df.select_dtypes(include=np.number).columns
print(columnas_numericas)
# Creamos una lista vacía que contendrá los ejes a graficar
x = []
# Pedimos al usuario que seleccione las 3 columnas a graficar
for i in range(3):
    # Evitamos que ingrese una columna que no exista
    while True:
        eje = input("Ingrese la columna " + str(i + 1) + ": ")
        if eje in columnas_numericas:
            break
        else:
            print("Esa columna no está en el archivo.")
            x.append(eje)
    # Verificamos que las tres columnas tengan la misma cantidad de elementos
    if len(x[0]) == len(x[1]) and len(x[1]) == len(x[2]):
        pass
    else:
        raise Exception("Las columnas no tienen el mismo número de datos.")

```

```

Columnas:
Index(['X1', 'X2', 'X3', 'X4', 'X5'], dtype='object')
Ingrese la columna 1: X1
Ingrese la columna 2: X2
Ingrese la columna 3: X3

```

Después, para poder visualizar atributos diferentes, necesitamos marcadores diferentes, una forma de hacer esto es crear una lista de marcadores diferentes para poder acceder a ellos más fácilmente.

Código:

```
marcadores = ["o", "x", "*", "v", "^", "l", "s", "p", "P"]
```

Una vez creada la lista de marcadores que nos ayudará a distinguir las diferentes instancias de un mismo atributo, vamos a preguntar si hay algún atributo a visualizar.

Código:

```

# Preguntamos si hay algún atributo a visualizar
atrib = input("Atributo a visualizar, máximo 9 ('s' para sí, 'n' para no): ")
# Si lo hay entra aquí
if atrib.lower() == "s":
    # Evitamos que ingrese una columna que no exista
    while True:
        atrib = input("Nombre de columna: ")
        if atrib in nombre_columna:
            break
        else:
            print("Esa columna no está en el archivo.")
    # Guardamos todas las instancias distintas
    instancias = set(df[atrib])
    # Verificamos que sean menos de 9

```

```

if len(instancias) < 10:
    i = 0
    # Ciclo para graficar
    for instancia in instancias:
        # Filtramos los datos de acuerdo al atributo
        subx = df.loc[df['atrib'] == instancia]
        # Lo graficamos con marcadores distintos para cada atributo
        plt.gca(projection="3d").scatter(
            subx[x[0]], subx[x[1]], subx[x[2]], marker=marcadores[i]
        )
        i = i + 1
    # Le ponemos título a la gráfica
    plt.title("{} - {} - {}".format(x[0], x[1], x[2]))
    # Le ponemos leyendas con los atributos
    plt.legend(instancias)
else:
    print("Número de instancias excedido, máximo 9.")

# Si no hay atributo a visualizar, se grafica directamente
else:
    plt.gca(projection="3d").scatter(df[x[0]], df[x[1]], df[x[2]])
    # Le ponemos título a la gráfica
    plt.title("{} - {} - {}".format(x[0], x[1], x[2]))

```

Si lo hay, necesitamos definir las condiciones para graficar. En caso de que haya un atributo a graficar, pedimos el nombre de la columna y ponemos una restricción de máximo nueve instancias distintas porque son nueve los marcadores que hay en la lista. Si queremos más de nueve, basta con agregar más marcadores a la lista. De igual modo que como hicimos anteriormente, utilizamos un **try/except** para evitar que se ingrese una columna que no exista en el archivo, guardamos las instancias —que son los elementos de la columna seleccionada— en un **set**, gracias a las propiedades del **set**, solo tendremos las instancias distintas una sola vez, comprobamos que sean máximo nueve y procedemos a graficar.

Para graficar, utilizaremos un ciclo **for** debido a que, para cada instancia distinta, necesitaremos crear un subconjunto de datos. Esto lo hacemos con la función **loc** de la librería pandas, que puede funcionar con valores booleanos como parámetro, de ahí que comparemos los datos de la columna con cada instancia, si la instancia es la misma, nos arrojará un valor *True* y se agregará al subconjunto, si no son iguales, nos arrojará un valor *False* y no se agregará. Ahora, para que la gráfica sea en 3D, tenemos que pasarle el parámetro «**projection=3d**» a la función **gca** y llamar a la función **scatter**, con las columnas del subconjunto creado y el marcador —nótese que a *marker* le pasamos posiciones de la lista de marcadores que creamos previamente, por lo que depende de *i*, por cada vez que se ejecute el ciclo, incrementamos *i* en una unidad, así que la posición del marcador avanza y nos proporciona un marcador diferente cada vez—.

Por último, le ponemos título y leyendas a nuestra gráfica con las funciones **title** y **legend**, respectivamente.

```

Columnas:
Index(['X1', 'X2', 'X3', 'X4', 'X5'], dtype='object')
Ingrese la columna 1: X1
Ingrese la columna 2: X2
Ingrese la columna 3: X3
Atributo a visualizar, máximo 9 ('s' para sí, 'n' para no): s
Columnas:
['X1', 'X2', 'X3', 'X4', 'X5', 'Supervivencia']
Nombre de columna: Supervivencia

```

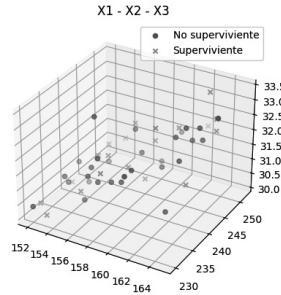


Figura 8.26. Gráfica que muestra la relación entre tres atributos

En el caso de que no haya ningún atributo que se quiera visualizar, graficamos directamente las columnas proporcionadas, como se muestra en la figura 8.27.

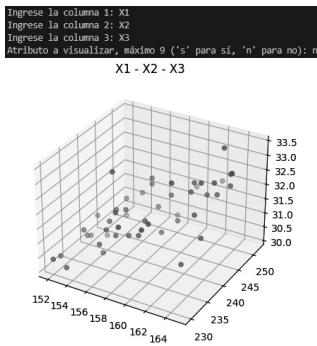


Figura 8.27. Gráfica que muestra la relación entre los tres atributos seleccionados

8.5.2. Gráfica de dispersión tipo matriz

Si lo que queremos es visualizar cada una de las características con las demás utilizando Python, una de nuestras opciones es la función **scatter_matrix** de la librería pandas. Esta función nos muestra cada una de las características en una matriz, lo que nos permite ver cómo se relaciona cada atributo con los demás.

Scatter_matrix recibe un objeto de tipo *dataframe*, un *string* para *marker*, un *string* o una secuencia de colores para *color*, entre otros parámetros. A diferencia de **scatter** de **pyplot**, no nos permite generar más de una gráfica dentro de la misma figura, por lo que no es viable generar subconjuntos de datos como lo hicimos previamente. Aunque esta función recibe un solo marcador para todos los datos, puede recibir varios colores, puesto que, para el color, recibe un *array*. Luego, nuestra mejor opción para poder visualizar diferentes instancias de un mismo atributo es usar un color diferente por cada instancia. Además, una de las ventajas de esta función es que filtra automáticamente las columnas que no son numéricas.

Crearemos una lista de colores para poderlos manipular más fácilmente, tal y como hicimos anteriormente con los marcadores.

Código:

```
colores = ["red", "blue", "black", "green", "gray", "yellow", "cyan"]
```

Abrimos el archivo donde están los datos que queremos visualizar.

Código:

```
archivo = input("Ubicación del archivo: ")
try:
    df = pd.read_csv(archivo)
    nombre_columna = []
```

```

for dato in df.columns:
    nombre_columna.append(dato)
except Exception as e:
    print("Archivo no válido.")
    print(e.__class__.__name__, e)
atrib = input("Atributo a visualizar, máximo 7 ('s' para sí, 'n' para no): ")
# Si hay atributo a visualizar entra aquí
if atrib.lower() == "s":
    # Evitamos que ingrese una columna que no exista
while True:
    atrib = input("Nombre de columna: ")
    if atrib in nombre_columna:
        break
    else:
        print("Esa columna no está en el archivo.")
    # Guardamos todos los atributos distintos
    atributo = set(df[atrib])
    # Creamos una lista vacía para agregar los colores
    color = []
    # Agregamos colores de acuerdo a la cantidad de atributos distintos
    for i in range(len(atributo)):
        color.append(colores[i])
    # Creamos un diccionario para ligar los colores y el atributo
    color_at = dict(zip(atributo, color))
    # Llamamos a la función scatter_matrix
    pd.plotting.scatter_matrix(
        df, c=[color_at.get(C) for C in df[atrib]], diagonal="hist"
    )
    print(color_at)

    # Si no hay atributo a visualizar entra aquí
else:
    pd.plotting.scatter_matrix(df)

```

En este código, utilizamos un **try/except** para verificar que se utilice un archivo válido y guardamos el nombre de las columnas existentes en el archivo.

Preguntamos si se quiere visualizar algún atributo, hacemos la observación de que sean máximo siete atributos distintos, debido a que solo tenemos siete colores en nuestra lista. En caso de que sí se quiera visualizar un atributo, pedimos que se ingrese el nombre de la columna y nos aseguramos de que la columna exista en el archivo mediante un **try/except**. Una vez más, guardamos las instancias en un **set** para tener solo una vez las instancias que son distintas. Ahora, de alguna forma, tenemos que vincular los colores con las instancias, esto lo haremos creando un diccionario para que, de esta manera, a cada instancia le corresponda un color.

Vamos a crear una lista vacía llamada **color** donde guardaremos tantos colores como instancias distintas tengamos, con ayuda de un ciclo **for** podemos guardar en la lista vacía el mismo número de colores que tengamos de instancias y utilizaremos las funciones **dict** y **zip** para crear nuestro diccionario a partir de la lista de colores **color** y el conjunto de instancias **atributo**. La función **zip** crea un objeto iterador entre dos o

más objetos iterables, mientras que la función **dict** es la que crea el diccionario. Es importante el orden en que ponemos nuestras listas porque la que vaya primero corresponderá a las llaves (**key**), y la segunda a los objetos asociados a dichas llaves (**values**).

En nuestro caso, las **keys** serán las instancias y los **values** serán los colores. Con esto, a cada instancia le corresponde un color, se hizo de esta forma porque le vamos a mandar a **scatter_matrix** una secuencia de colores para el parámetro **color**, así cada dato tendrá un color específico dependiendo de cuál sea su instancia del atributo que nos interesa ver. Finalmente, imprimimos nuestro diccionario para saber a qué dato le corresponde cada color –en nuestro caso, a la instancia **no superviviente** le corresponde el color rojo y a la instancia **superviviente** le corresponde el color azul–.

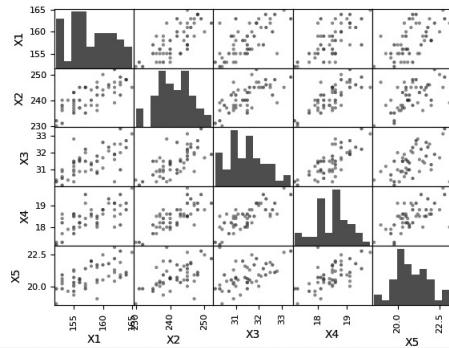


Figura 8.28. Matriz de dispersión que muestra las clases en diferentes colores

```
Atributo a visualizar, máximo 7 ('s' para sí, 'n' para no): s
Nombre de columna: Supervivencia
{'No superviviente': 'red', 'Superviviente': 'blue'}
```

En caso de que no queramos visualizar ningún atributo, simplemente, llamamos a **scatter_matrix** con nuestro *dataframe* como parámetro y en atributo a visualizar ponemos «**n**».

8.5.3. Gráfica de barra apilada (stacked bar)

Cuando se quiere ver el porcentaje de cada instancia de un atributo, nos resulta especialmente útil ver los datos en una gráfica de *barra apilada* (*stacked bar*).

En Python, contamos con la función **bar** de **pyplot**, la cual nos genera una gráfica de barra apilada. Esta función recibe como parámetros la posición de la barra, el número de datos y el fondo (**bottom**), es decir, donde empiezan los datos.

Abrimos el archivo y verificamos que sea válido a través de un **try/except**, además, pedimos dentro del mismo **try/except** el atributo a visualizar porque, de otro modo, no tiene sentido realizar una gráfica de barra apilada. Una vez más, aprovechamos las propiedades de **set** para guardar todas las instancias diferentes una sola vez.

Para poder apilar los datos, el parámetro que necesitamos modificar es el fondo, de tal modo que, cada vez que grafiquemos una instancia, la siguiente comience donde terminó la última. De esta forma, vamos a inicializar una variable llamada **fondo** en cero, que es donde debe empezar la primera instancia. Posteriormente, para poder ver los datos en porcentajes, debemos conocer el total de datos, para ello usaremos la función **count** de pandas en la columna del atributo y guardaremos ese valor en la variable **total**.

A través de un ciclo **for** graficaremos cada instancia, tantas como tengamos, el **for** se ejecutará una vez por cada instancia. Dentro del ciclo **for** vamos a ir filtrando los datos de acuerdo a su instancia, esto lo hacemos con la función **loc** de pandas. Una vez filtrados los datos, haremos el conteo de ellos con **count** para saber cuántas veces aparece cada instancia en el conjunto y los dividimos entre el total de datos total, este porcentaje lo guardaremos en **sub**.

Después, procedemos a llamar a la función bar para graficarlos. Como en este caso solo crearemos una gráfica, es indiferente la posición que le mandemos para la gráfica, podemos utilizar cualquier número, por ejemplo, el 1; el segundo parámetro que le mandaremos es el porcentaje de los datos y el tercer parámetro será *bottom*, al cual le asignaremos el valor de fondo. En seguida, a fondo le sumamos el porcentaje de los datos almacenado durante esta iteración en *sub* para que el siguiente porcentaje empiece ahí.

El proceso se repite tantas veces como instancias tengamos. Cuando se hayan pasado todas las instancias, el programa sale del ciclo *for* y nos disponemos a ponerle los límites al eje X con la función **xlim** de **pyplot** para que nuestra barra sea visible, como utilizamos el 1 como posición de la barra, unos buenos límites serían (0, 2), ponemos las leyendas que serán las instancias distintas y, por último, colocamos el título, al cual le agregaremos el nombre del atributo que estamos viendo, como se muestra en la figura 8.29.

Código:

```
try:
    df = pd.read_csv(archivo)
    nombre_columna = []
    for dato in df.columns:
        nombre_columna.append(dato)
    # Imprimimos las columnas que hay
    print(nombre_columna)
    # Pedimos un atributo (columna) a visualizar y verificamos que exista
    while True:
        atributo = input("Seleccione el atributo a visualizar: ")
        if atributo in nombre_columna:
            break
        else:
            print("El atributo no existe")
    # Guardamos todos las estancias distintas de ese atributo
    atrib = set(df[atributo])
except Exception as e:
    print("Archivo no válido.")
    print(e.__class__.__name__, e)
# Inicializamos el fondo
fondo = 0
# Sacamos el total para sacar el porcentaje
total = df.count()
# Graficamos cada una de las diferentes instancias
for at in atrib:
    # Filtramos los datos
    sub = df.loc[df[atributo] == at]
    # Hacemos el conteo y dividimos para sacar porcentaje
    sub = sub[atributo].count() / total
    # Llamamos a la función bar()
    plt.bar(1, sub, bottom=fondo)
    # Vamos sumando el fondo para que sea una barra apilada
    fondo += sub
# Establecemos el límite para el eje x
plt.xlim(0, 2)
```

```
# Ponemos las leyendas
plt.legend(atrib)
# Ponemos el título
plt.title("Porcentaje de datos de acuerdo a " + atributo)
```

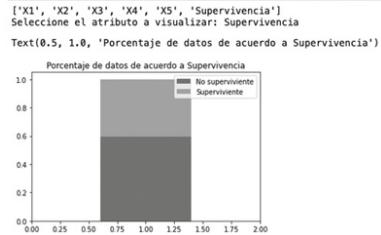


Figura 8.29. Gráfica de barra apilada

8.5.4. Gráfica de cajas (boxplot)

La gráfica *boxplot* nos ayuda a ver la distribución de los datos. Esta gráfica la podemos generar con la función **boxplot** de **pyplot**.

Para este caso, veremos cómo se distribuyen los datos de cada columna de acuerdo a un atributo en específico.

Para realizar esta figura, abrimos el archivo donde están nuestros datos, imprimimos los nombres de las columnas que existen en nuestro archivo y, mediante un *while*, pedimos el nombre del atributo a visualizar y verificamos que se encuentre en el archivo. Una vez que comprobamos que el atributo existe en el archivo, guardamos todas sus instancias distintas en un set, posteriormente, se filtran las columnas no numéricas para no tener errores al momento de graficar. Esto lo hacemos con la función **select_dtypes** de la librería **pandas**, la cual nos permite filtrar datos, gracias a su parámetro **include**; para conservar solo las columnas numéricas, le mandamos el parámetro **np.number** en **include** (`include=np.number`).

Como vamos a realizar una gráfica por cada atributo o columna del archivo, necesitamos saber cuántos son con el fin de implementar adecuadamente la función **subplot**, la cual nos permite realizar varias gráficas en una misma figura. Posteriormente, se obtiene el número de atributos –de los datos ya filtrados– con la función **len** y sacamos raíz cuadrada para poder distribuir de manera óptima las columnas y filas de gráficas que tendrá nuestra figura.

Por ejemplo, si tenemos cuatro atributos, la mejor manera de visualizar las gráficas sería estableciendo dos columnas y dos filas, si tenemos cinco atributos, lo mejor sería tener dos columnas y tres filas o tres columnas y dos filas.

En la comparación, lo que hacemos es hacer un *casting* a la raíz del total de columnas para quedarnos solamente con la parte entera del número, luego lo comparamos con la raíz original y, si son iguales, significa que nuestra raíz es exacta.

Una vez que se tiene el número de columnas y de filas a implementar, es hora de crear el ciclo donde graficaremos. Este ciclo creará una gráfica de *boxplot* por cada atributo distinto al que queramos visualizar y la posicionará dentro de nuestra figura, por lo que dentro de este ciclo se llamará a la función **subplot** con la posición correspondiente para cada gráfica –posición que va de 1 hasta *n*, mismo que controlaremos con una variable previamente inicializada–. Se recorrerá cada columna –creando un objeto iterativo para poderlo ejecutar tantas veces como columnas haya– y creará un *boxplot* por cada atributo distinto del atributo a visualizar. Es decir, si tenemos dos atributos distintos, se creará una gráfica con dos *boxplots*, si fueran tres atributos, sería una gráfica con tres *boxplots* y así sucesivamente.

Dicho lo anterior, procedemos a crear el objeto iterativo **iter_columnas** que contendrá el nombre de las columnas a graficar. No podemos acceder directamente a los nombres de las columnas desde el objeto en sí, sino que tenemos que llamar a la función **next**, que recibe al objeto iterativo y nos devuelve otro objeto.

En este caso, el nombre de una columna, por lo que después de crear **iter_columnas** llamamos a **next** para obtener el nombre de la primera columna que graficaremos, el cual lo guardaremos en **columna**. También inicializamos la variable que llevará el control de las posiciones del **subplot** llamada **posicion**, la inicializamos en uno, porque esa es la primera posición para **subplot**. Hecho esto, creamos un *for* que será para movernos por las filas –variable **n**– y dentro de este creamos otro para movernos por las columnas –variable **m**–, ahora ya podemos llamar a **subplot**, mandándole como parámetros: **n**, **m** y **posicion** (**subplot** recibe tres parámetros: *nrows*, *ncols* e *index*).

Lo siguiente es inicializar una variable **pos_box** para controlar la posición de los *boxplots* que se tengan y crear un *for* para recorrer todas las instancias distintas. Dentro de este tercer *for* creamos los subconjuntos

para graficar de acuerdo a las instancias diferentes que haya. Filtramos los datos de cada columna con la siguiente sintaxis:

`df[columna].loc[df[atrib] == at]`, donde **df** es el nombre de nuestro objeto tipo *dataset*, **columna** es la variable que contiene el nombre de una columna –dato obtenido de nuestro objeto iterador **iter_columnas**–, **atrib** es el nombre del atributo que estamos visualizando y **at** es una instancia del atributo, este subconjunto lo guardamos en **subx**.

A continuación, llamamos a la función *boxplot* a la que le mandamos como parámetros: los datos a graficar **subx**, la etiqueta de los datos **at** y la posición **pos_box**, luego incrementamos en una unidad la posición del *boxplot* **pos_box**, y el ciclo se repite tantas veces como instancias distintas tengamos. Una vez terminado este ciclo, se llama a la función **title** para ponerle título a la gráfica que acabamos de hacer, aplicamos un *try/except* –fuera del tercer *for* y dentro del segundo–, donde ejecutaremos la función **next** para obtener otra columna y aumentamos en una unidad **posición** para que la posición del subplot cambie. Se hace dentro de un *try/except* porque, cuando se utiliza la función **next** en el objeto iterable y ya no hay elementos en él, se levanta una excepción que hay que manejar –en nuestro código esto ocurre cuando la raíz cuadrada del número de columnas no es exacta–. El código de lo que se acaba de explicar se muestra a continuación, mientras que la figura resultante se muestra en la figura 8.30:

```
Código:  
try:  
    df = pd.read_csv(archivo)  
    nombre_columna = []  
    for dato in df.columns:  
        nombre_columna.append(dato)  
    except Exception as e:  
        print("Archivo no válido.")  
        print(e.__class__.__name__, e)  
    # Desplegamos las columnas que hay en el archivo  
    print("Columnas:\n", nombre_columna)  
    # Evitamos que ingrese una columna que no exista  
    while True:  
        atrib = input("Atributo a visualizar: ")  
        if atrib in nombre_columna:  
            break  
        else:  
            print("Esa columna no está en el archivo.")  
    # Guardamos todos las estancias distintas  
    atributo = set(df[atrib])  
    # Filtramos las columnas no numéricas  
    columnas = df.select_dtypes(include=np.number).columns  
    # Contamos el número de columnas a graficar  
    n_columnas = len(columnas)  
    # Número de columnas y filas que tendrá nuestro subplot  
    if int(np.sqrt(n_columnas)) == np.sqrt(n_columnas):  
        n = m = int(np.sqrt(n_columnas))  
    else:  
        n, m = int(np.sqrt(n_columnas)), int(np.sqrt(n_columnas)) + 1  
    # Creamos un objeto iterativo
```

```

iter_columnas = iter(columnas)
# Llamamos a la primera columna
columna = next(iter_columnas)
# Variable para llevar el control de subplot
posicion = 1
# Le ponemos título a la figura
plt.figure().suptitle(atrib, fontsize=20)
for i in range(n):
    for j in range(m):
        # Habilitamos el subplot
        plt.subplot(n, m, posicion)
        # Variable para llevar el control del boxplot
        pos_box = 1
        for at in atributo:
            # Filtramos los datos de acuerdo al atributo
            subx = df[columna].loc[df[atrib] == at]
            # Llamamos a la función boxplot
            plt.boxplot(subx, labels=[at], positions=[pos_box])
            pos_box = pos_box + 1
        # Agregamos el título de la columna
        plt.title(columna)
        # Manejamos la excepción que se genera cuando
        # ya no hay elementos en iter_columnas
        try:
            columna = next(iter_columnas)
            posicion = posicion + 1
        except Exception:
            break

```

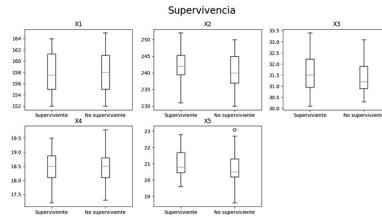


Figura 8.30. Gráfica de cajas de ejercicio de gorriones

8.6. Detección de errores de cardinalidad

Con los datos ya obtenidos, debemos revisarlos para identificar errores de cardinalidad, si es que los hay. Una forma de identificar los errores de cardinalidad es dando los parámetros dentro de los cuales nosotros consideramos que los datos deben de estar.

Para exemplificar esto, utilizaremos el conjunto de datos **edades**, el cual contiene dos columnas, una con datos numéricos –edades– y la otra con datos alfanuméricos –género– mismo que se explica en la sección A.8.2.

El código se muestra en esta sección y puede ser descargado en:
http://www.amese.net/libro_ia/Prog/8.

Primeramente, se importan las librerías y se abre el archivo con el que se estará trabajando ('edades.csv').

Código:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
df = pd.read_csv('Edades.csv')
edades = df['Edad']
genero = df['Genero']
df.head()
```

	Edad	Genero
0	23	h
1	22	m
2	21	m
3	25	m
4	30	h

Figura 8.31. Listado de base de datos de edad.csv

En el código anterior, leemos y desplegamos una muestra del conjunto de datos, como se muestra en la figura 8.31. Vamos a revisar primero la columna de edad para ver si encontramos algún error de cardinalidad. Para estos datos, necesitamos fijar los parámetros superior e inferior, esto lo hacemos pidiendo al usuario que los ingrese por consola.

Para esto definimos una **función** llamada **error_cardinalidad** que recibe como parámetro un objeto de tipo **data frame**.

Nuestra función debe servir tanto para datos numéricos como para datos alfanuméricos. Evidentemente, estos datos se tratan de distinta forma, por lo que debemos crear «dos casos». Puesto que nuestra primera línea de código es sacar la desviación estándar, para evitar errores en nuestro código, vamos a implementar un *try-except*. Si el tipo de dato es alfanumérico, levantaremos una excepción y estaríamos entrando al «segundo caso».

En el primer caso, haremos el código para tratar los tipos de datos numéricos, lo primero que hacemos es pedir al usuario que ingrese los parámetros (**límite_inf** y **límite_sup**), con los que se evaluarán los datos. Estos parámetros también tienen que ser numéricos, de lo contrario, tendríamos un error. Para esto vamos a utilizar un ciclo *while* y dentro de este pondremos otro *try-except* para limitar la entrada de datos a números solamente, adicional a esto debemos verificar que el límite superior (**límite_sup**) sea mayor al límite inferior (**límite_inf**).

Una vez que los límites son validados hay que graficar, para este programa crearemos una función llamada **graficar**, que explicaremos más adelante.

Al segundo caso se accede cuando a los datos no se les puede sacar la desviación estándar, lo que quiere decir que no son datos numéricos.

En este segundo caso, necesitamos los parámetros válidos, de modo que, los pediremos por teclado y los iremos agregando a una lista que llamaremos **parámetros**. Dado que no sabemos cuántos parámetros tiene el conjunto de datos, implementaremos un ciclo *while* para que el usuario agregue todos los parámetros necesarios. Después, creando el índice **index** de los datos y una lista vacía **errores**, utilizamos un ciclo *for* donde todos comparamos los datos del conjunto con los parámetros proporcionados y, si no coinciden, los agregamos a la lista de errores que imprimiremos al final del ciclo *for* como se muestra en el siguiente código:

Código:

```
def error_cardinalidad(A):
    try:
```

```

# Calculamos la desviación estándar con el fin de comprobar que el tipo de dato no
# sea string. En caso de que lo sea, se levanta la excepción y ejecuta el código
# para el dato tipo string.
A.std()

# Pedimos al usuario que ingrese el valor de los límites, implementamos un ciclo while para
# asegurar que el usuario ingrese un tipo de dato numérico y que además el límite superior sea
# mayor al inferior
while True:
    try:
        # Leemos el límite inferior
        limite_inf = int(input("Límite inferior: "))
        # Leemos el límite superior
        limite_sup = int(input("Límite superior: "))
        # Checamos si el límite superior es mayor al inferior
        # si lo es sale del ciclo while
        # sino levanta una excepción y vuelve a pedir los datos
        if limite_sup > limite_inf:
            break
        else: raise Exception('Límite superior no es mayor a límite inferior')

    except Exception as e:
        print('Los límites tienen que ser un numeros y el límite superior debe ser mayor al inferior')
        print(e.__class__.__name__,e)
        # Llama a la función graficar para mostrar los datos en una gráfica de dispersión y mostrar
        # los errores de cardinalidad
        graficar(A,limite_inf,limite_sup)
    except TypeError:
        # Aquí entra si el dato es de tipo carácter
        # creamos una lista vacía donde el usuario dará los parámetros válidos
        parametros = []
        # Creamos un ciclo para que el usuario agregue tantos parámetros como desee
        while True:
            parametros.append(input('Parámetro: '))
            agregar = input('Agregar otro parámetro?\n s n\n').lower()
            if agregar == 's' or agregar == 'si':
                pass
            else:
                break

```

Veamos ahora la función **graficar**, esta función recibe tres parámetros, el conjunto de datos y los dos límites definidos. Aquí también crearemos una lista vacía para los errores y/o *outliers*.

Haremos dos gráficas, una scatter y un boxplot, con las funciones de la librería **matplotlib**, **scatter** y **boxplot**, respectivamente. Haremos uso también de la función **subplot** de la misma librería para visualizar las dos gráficas en la misma ventana. Por último, usaremos un ciclo *for* para detectar los errores de cardinalidad y/o *outliers* que tenga el conjunto de datos de acuerdo a los límites proporcionados. En este ciclo, los errores identificados por el programa se mostrarán de color rojo y se agregarán a la lista **errores**, la cual se imprimirá al final.

Código:

```

# Creamos el índice de los datos
index = np.arange(0,A.shape[0])

# Creamos una lista vacía para agregar los errores de cardinalidad
errores = []

# Verificamos si los datos son válidos con base en los parámetros dados
for ind in index:
    if A[ind] in parametros:
        pass
    else:
        # Se agregan los errores a la lista de errores
        errores.append(str(ind)+':'+str(A[ind]))

# Imprimimos los errores de cardinalidad
print(errores)

```

Caso 1:

Código:

```

df = pd.read_csv('Edades.csv')
edades = df['Edad']
genero = df['Genero']

error_cardinalidad(edades)

```

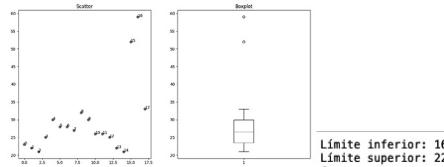


Figura 8.32. Gráficas de dispersión y de caja para detección de errores de cardinalidad

Una vez introducidos los límites del segundo programa, muestra dos gráficas. En la primera (Scatter), podemos observar todos los datos de los cuales los azules son los que están dentro de los límites y los rojos son aquellos definidos por nosotros como errores de cardinalidad, en la segunda (Boxplot), vemos los datos en un boxplot que nos muestra los outliers en nuestros datos.

Además, el programa nos devuelve una lista con los errores de cardinalidad, indicándonos el índice y el valor, de la forma: **[índice : valor]**.

```
[0:23, 3:25, 4:30, 5:28, 6:28, 7:27, 8:32, 9:30, 10:26, 11:26,
12:25, 15:52, 16:59, 17:39]
```

Figura 8.33. Listado de errores de cardinalidad

Veamos el caso en que los datos no son numéricos.

Caso 2:

Código:

```

df = pd.read_csv('Edades.csv')
edades = df['Edad']
genero = df['Genero']

error_cardinalidad(genero)

```

Esta vez, el programa nos pide los parámetros válidos, hasta que le digamos que ya no deseamos agregar otro parámetro para este conjunto de datos. Para este ejemplo solo se consideraron dos, **h** y **m**.

A la salida nos devuelve una lista con los errores de cardinalidad, tal como en el caso 1, indicándonos el índice y el valor, de la forma: **[índice : valor]**, como se muestra en la figura 8.34.

```
Parámetro: m
Aregar otro parámetro?
s n
s
Parámetro: h
Aregar otro parámetro?
s n
[9:j', '17:n']
```

Figura 8.34. Listado de errores de cardinalidad para datos no numéricos

8.7. Detección de valores atípicos (outliers)

El código para la detección de *outliers* se muestra en esta sección y puede ser descargado en: http://www.amese.net/libro_ia/Prog/8.

En este ejercicio, se usará la base de datos de gorriones, que se explica en el apéndice A.3.

Como ya se ha mencionado, una forma para detectar los *outliers* es comparando la brecha entre la mediana, los valores mínimos, máximos y los cuartiles uno y tres, respectivamente.

El primer cuartil es el valor que tiene por debajo al 25 % de los datos, el segundo cuartil es el que tiene por debajo al 50 % de los datos, el tercer cuartil es el que tiene por debajo al 75 % de los datos y el cuarto cuartil es el que tiene por debajo al 100 % de los datos.

Este método consiste en calcular el primer y tercer cuartil, calcular el rango intercuartil, IQR por sus siglas en inglés (*inter quantile range*) y los límites inferior y superior.

Para calcular los cuartiles, utilizamos la función **cuartil** de la librería **pandas**, el IQR es la diferencia entre el cuartil 3 (Q3) y el cuartil 1 (Q1): Q3-Q1, el límite inferior se calcula como $L_{inf} = Q1 - (1.5 * IQR)$ mientras que el superior se calcula como $L_{sup} = Q3 + (1.5 * IQR)$.

Ahora, pasemos esto a código. Primero las librerías y abrimos la BD de gorriones:

Código:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
df = pd.read_csv('Gorriones.csv')
df.head()
```

Utilizamos la misma función graficar que en el ejercicio 8.6 para detectar errores de cardinalidad, como se muestra en el siguiente código:

Código:

```
def graficar(A, limite_inf, limite_sup):
    # Creamos el índice
    index = np.arange(0, A.shape[0])
    # Creamos una lista vacía para agregar los errores de cardinalidad y/o los outliers
    errores = []
    # Graficamos los datos en una gráfica de dispersión
    plt.subplot(1, 2, 1)
    plt.scatter(index, A)
    plt.title('Scatter')
```

```

for ind in index:
    plt.annotate(ind, (ind, A[ind]))

if A[ind] < limite_inf or A[ind] > limite_sup:
    # Graficamos los outliers de color rojo
    plt.plot(ind, A[ind], color='red', marker='o')
    # Agregamos los outliers y/o errores de cardinalidad a la lista de errores
    errores.append(str(ind)+':'+str(A[ind]))

# Imprimimos los outliers
print(errores)
plt.subplot(1, 2, 2)
plt.boxplot(A)
plt.title('Boxplot')

```

Posteriormente, creamos una función llamada **det_outliers_quantiles**, luego, lo primero que hacemos es usar un *try-except* con el fin de evitar datos no numéricos, ya que, por obvias razones, este método no funciona con datos no numéricos, después hacemos el cálculo de los cuartiles, el IQR y los límites inferior y superior.

Código:

```

def det_outliers_quantiles(A):
    # try/except para evitar tipos de datos no numericos
    try:
        # Definimos el primer quantile (25%)
        Q1 = A.quantile(0.25)
        # Definimos el tercer quantile (75%)
        Q3 = A.quantile(0.75)
        # Calculamos el IQR
        IQR = Q3 - Q1
        # Calculamos los límites
        limite_inf = Q1 - (1.5*IQR)
        limite_sup = Q3 + (1.5*IQR)
        # Llamada a la función graficar
        graficar(A, limite_inf, limite_sup)
    except Exception as e:
        print('Tipo de dato no valido.\nTiene que ser numerico')
        print(e.__class__.__name__, e)

```

Para este ejemplo, utilizaremos la columna **X2**, guardada en **ejeY**, del conjunto de datos de gorriones, mismo que se explica a detalle en el apéndice A.3.

Código:

```

ejeY = df['X2']
det_outliers_quantiles(X1)

```

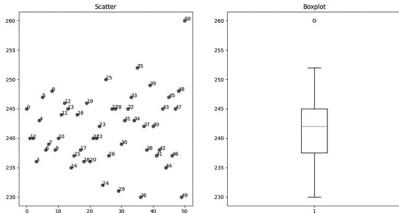


Figura 8.35. Gráficas de dispersión y de caja para detección de valores atípicos

Nuevamente, obtenemos como salida una lista que nos muestra, en este caso, el *outlier* de este conjunto de datos y las dos gráficas donde podemos visualizar el *outlier*.

8.8. Análisis de componentes principales (PCA)

El disponer de grandes cantidades de datos es, en la mayoría de las ocasiones, necesario para poder entender mejor el fenómeno, ya sea para poder hacer predicciones, modelarlo o clasificarlo mediante algoritmos de aprendizaje automático.

Sin embargo, para poder realizar este modelado, en ocasiones, se tienen demasiados atributos –es decir, dimensiones–, y no se sabe a ciencia cierta cuáles aportan una mayor cantidad de información y con cuáles dimensiones se incrementa la interpretabilidad de los datos. Para esto, se realizan las técnicas llamadas de reducción de dimensionalidad.

Una de las técnicas de reducción de dimensionalidad más utilizada es la de análisis de componentes principales (PCA, por sus siglas en inglés). PCA reduce la dimensionalidad de los datos y, al mismo tiempo, minimiza la pérdida de información.

Para poder realizar esta reducción de dimensionalidad con PCA, se tienen que realizar los siguientes pasos:

- Obtener los datos –en todas sus dimensiones–.
- Restar la media. Para que PCA funcione de manera correcta, se requiere restar la media de cada dimensión.
- Calcular la matriz de covarianza.
- Calcular los eigenvalores y los eigenvectores de la matriz de covarianza.
- Escoger los componentes y formar el vector de características. De la columna de los eigenvalores, el elemento con el número mayor es el componente principal del set de datos.
- En general, una vez que los eigenvectores se encuentran en la matriz de covarianza, se ordenan de mayor a menor dando estos los componentes principales en orden.

Usualmente, no es sencillo imaginar cómo interpretar gráficamente esta dimensionalidad. Imaginemos en una nube de puntos que representa los datos en tres dimensiones dadas por los atributos: var1, var2 y var3. A través de estos puntos, podemos trazar una línea recta, la cual es un componente principal (P_1). Esta línea contendrá la información de las distancias entre los puntos y la línea, donde la línea pasa por el centroide t de estos puntos. La representación gráfica de los componentes principales se muestra en la figura 8.36.

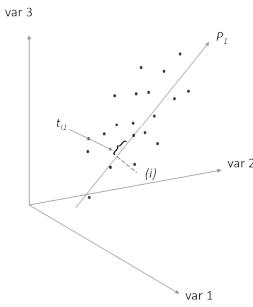


Figura 8.36. Representación gráfica de un componente principal

En este ejercicio, se hará una comparación de clasificación por KNN —solo la clasificación, se puede observar en el ejercicio de la sección 8.15— con y sin utilizar PCA. El código que utilizaremos en este ejercicio es el de datos de pasajeros del Titanic, como se muestra en la sección A.8.1.

El código para determinar los componentes principales por medio de PCA se muestra en esta sección y puede ser descargado en: http://www.amese.net/libro_ia8PCA

Comencemos por importar las librerías, como se muestra a continuación:

Código:

```
import numpy as np
import pandas as pd
import math
import matplotlib.pyplot as plt
import time
from collections import Counter
```

Posteriormente, se carga el archivo «Titanic.csv», que es un archivo separado por comas —o extensión .csv—, y se eliminan las columnas que sabemos que no aportarán nada para este ejercicio, las cuales son: «Nombre del pasajero (Name)», «Identificación del pasajero (PassengerId)», «Número de billete (Ticket)», «Número de cabina (Cabin)» y «Puerto de Embarcación (Embarked)». El código se muestra a continuación:

Código:

```
# Se lee el archivo
# Tiene que estar importado en el entorno de Jupyter
df = pd.read_csv("Titanic.csv", sep=',');
df = df.drop(['Name','PassengerId','Ticket','Cabin','Embarked'], axis = 1)
df.head()
```

Algunos atributos restantes tienen características lingüísticas, por lo que se tendrán que convertir a valores categóricos —binarios en este caso—. Este es el caso del atributo «género» (Gender). En este caso, se codificó como «0» - male (hombre) y como «1» - female (mujer), reemplazando cada valor por su correspondiente valor categórico, como se muestra en el siguiente código:

Código:

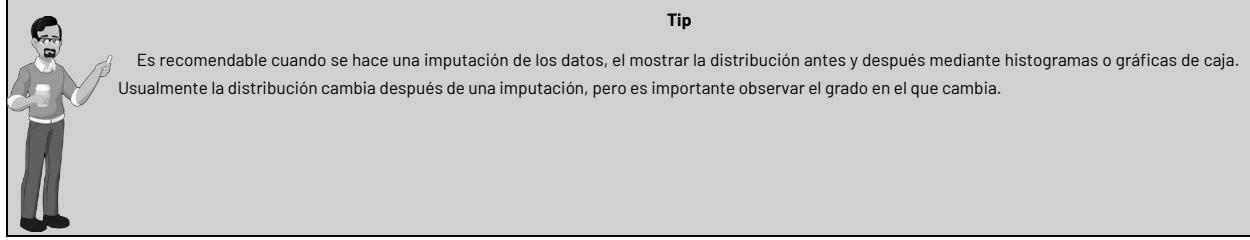
```
# Preprocesamiento de los datos
# Primeramente, se codifican los datos categóricos
# para poder trabajar con ellos de manera numérica
datos = df.where(~df.isin(['male']),0)
datos = datos.where(~datos.isin(['female']),1)
datos
```

La figura 8.37 muestra la codificación de este atributo en un listado de la base de datos.

	Survived	Pclass	Gender	Age	SibSp	Parch	Fare
0	0	3	0	22.0	1	0	7.2500
1	1	1	1	38.0	1	0	71.2833
2	1	3	1	26.0	0	0	7.9250
3	1	1	1	35.0	1	0	53.1000
4	0	3	0	35.0	0	0	8.0500
—	—	—	—	—	—	—	—
886	0	2	0	27.0	0	0	13.0000
887	1	1	1	19.0	0	0	30.0000
888	0	3	1	NaN	1	2	23.4500
889	1	1	0	26.0	0	0	30.0000
890	0	3	0	32.0	0	0	7.7500

Figura 8.37. Listado de base de datos codificado para análisis de componentes principales

Debido a que no es recomendable utilizar PCA con valores faltantes, se realiza una técnica llamada imputación por medias. Básicamente, se reemplazan los valores por la media de estos. En este caso, la imputación se realizó al atributo «Age» (Edad). Existen muchos otros métodos de imputación, como se mencionó en la sección 4.5. También se muestra un ejemplo de una imputación más avanzada llamada imputación múltiple por ecuaciones encadenadas (o MICE) en la sección 8.10. Para este ejercicio, también se recomienda una imputación tipo hot-deck o por tabla (LUT).



Código:

```
# Revisar datos nulos
datos.isnull().sum()

# Graficar antes de la imputación por media
fig, axes = plt.subplots(1,2, figsize=(12,4))
plt.ylabel("Frecuencia")
plt.xlabel("Age")
datosAI = datos['Age'].dropna()
axes[0].hist(datosAI, bins=20)
axes[1].boxplot(datosAI)
axes[1].grid(True)
plt.show()

# Imputación simple por media
x = datos['Age'].mean()
datos = datos.fillna(x)

# Graficar después de la imputación por media
fig, axes = plt.subplots(1,2, figsize=(12,4))
plt.ylabel("Frecuencia")
plt.xlabel("Age")
axes[0].hist(datos['Age'], bins=20)
axes[1].boxplot(datos['Age'])
axes[1].grid(True)
plt.show()

# Se observa si aún hay datos nulos
```

```
datos.isnull().sum()
```

Usualmente, cuando se trabaja con algunos algoritmos, como en este caso para reducción de dimensionalidad, es recomendable normalizar los datos. En este caso, se utilizó la normalización **min-max**. El código de la normalización se muestra a continuación, mientras que el resultado de los atributos normalizados se muestra en la figura 8.38.

Código:

```
# Guardar los valores máximo y mínimo y rango de cada atributo
maxDatos = []
minDatos = []
rangoDatos = []
maxNorm = 1
minNorm = 0
rangoNorm = maxNorm - minNorm

for i in range (0,datos.columns.size):
    maxDatos.append(datos.iloc[:,i].max())
    minDatos.append(datos.iloc[:,i].min())
    rangoDatos.append(maxDatos[i] - minDatos[i])

nombres = datos.columns.values.tolist()
datosNorm = pd.DataFrame(columns=nombres)
datosNorm

for j in range (len(datos.columns)):
    varNorm = []
    # Se selecciona uno de los atributos
    var = datos.iloc[:,j]
    for i in range (len(datos)):
        # Se obtiene el valor normalizado
        D = var[i] - minDatos[j]
        Dpet = D/rangoDatos[j]
        dNorm = rangoNorm*Dpet
        varNorm.append(minNorm+dNorm)
    datosNorm.iloc[:,j] = varNorm

datos = datosNorm
datos
```

	Survived	Pclass	Gender	Age	SibSp	Parch	Fare
0	0.0	1.0	0.0	0.271174	0.125	0.000000	0.014191
1	1.0	0.0	1.0	0.472229	0.125	0.000000	0.139196
2	1.0	1.0	1.0	0.321438	0.000	0.000000	0.015469
3	1.0	0.0	1.0	0.434531	0.125	0.000000	0.103644
4	0.0	1.0	0.0	0.434531	0.000	0.000000	0.015713
...
886	0.0	0.5	0.0	0.334404	0.000	0.000000	0.025374
887	1.0	0.0	1.0	0.233476	0.000	0.000000	0.058556
888	0.0	1.0	1.0	0.367921	0.125	0.353333	0.045771
889	1.0	0.0	0.0	0.321438	0.000	0.000000	0.058556
890	0.0	1.0	0.0	0.396833	0.000	0.000000	0.015127

891 rows x 7 columns

Figura 8.38. Normalización de datos para preprocessamiento de análisis de componentes principales

Para comenzar con el algoritmo PCA, es importante definir la covarianza y la matriz de covarianza. Así mismo, se tiene que restar la media de cada atributo para poder realizar los cálculos de manera correcta.

Esto se muestra en el siguiente código:

```
Código:
# Se define la covarianza y matriz de covarianza
def covarianza(X,Y):
    x_mean = X.mean()
    y_mean = Y.mean()
    n = len(X)
    cov = (((X-x_mean)*(Y-y_mean)).sum())/(n-1)
    return cov

def matriz_cov(data):
    atributos = data.columns
    n = len(atributos)
    m = np.zeros((n,n))
    for i in range(n):
        for j in range(n):
            X = data[atributos[i]]
            Y = data[atributos[j]]
            m[i][j] = covarianza(X,Y)
    return m

# Eliminando el atributo de decisión
datos1 = datos.drop(['Survived'],axis=1)

# Ajustar los datos restando la media a cada atributo
datosAjustados = pd.DataFrame(columns=datos1.columns, index=range(len(datos1)))
for i in datosAjustados.columns:
    datosAjustados[i] = datos1[i] - datos1[i].mean()
datosAjustados
```

Se tiene que quitar para este cálculo el atributo de decisión. Esto es debido a que, si se dejara, se tomaría en cuenta para los cálculos posteriores y, probablemente, este atributo sería uno de los componentes principales. Esto no es recomendable cuando se trabaja con algoritmos de aprendizaje máquina. La figura 8.39 muestra el cálculo de la covarianza a nuestros atributos restantes.

	Pclass	Gender	Age	SibSp	Parch	Fare
0	0.345679	-0.352413	-9.674689e-02	0.059624	-0.063599	-0.048707
1	-0.654321	0.647587	1.043086e-01	0.059624	-0.063599	0.076277
2	0.345679	0.647587	-4.648301e-02	-0.065376	-0.063599	-0.047390
3	-0.654321	0.647587	6.661074e-02	0.059624	-0.063599	0.040786
4	0.345679	-0.352413	6.661074e-02	-0.065376	-0.063599	-0.047146
...
886	-0.154321	-0.352413	-3.391704e-02	-0.065376	-0.063599	-0.037484
887	-0.654321	0.647587	-1.344448e-01	-0.065376	-0.063599	-0.004302
888	0.345679	0.647587	1.102236e-16	0.059624	0.269734	-0.017087
889	-0.654321	-0.352413	-4.648301e-02	-0.065376	-0.063599	-0.004302
890	0.345679	-0.352413	2.891282e-02	-0.065376	-0.063599	-0.047731

Figura 8.39. Cálculo de covarianza para PCA

Para obtener la matriz de covarianza, se realiza el cálculo de acuerdo a la ecuación 8.5.

$$\Sigma = \begin{bmatrix} \text{cov}(x_0, x_0) & \text{cov}(x_0, x_1) & \text{cov}(x_0, x_2) & \cdots & \text{cov}(x_0, x_n) \\ \text{cov}(x_1, x_0) & \text{cov}(x_1, x_1) & \text{cov}(x_1, x_2) & \cdots & \text{cov}(x_1, x_n) \\ \text{cov}(x_2, x_0) & \text{cov}(x_2, x_1) & \text{cov}(x_2, x_2) & \cdots & \text{cov}(x_2, x_n) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \text{cov}(x_n, x_0) & \text{cov}(x_n, x_1) & \text{cov}(x_n, x_2) & \cdots & \text{cov}(x_n, x_n) \end{bmatrix} \quad (8.5)$$

Con ayuda de la librería **numpy** de Python, se obtuvieron los eigenvectores y eigenvalores de la matriz. La función utilizada es **np.linalg.eig**.

Se ordenan los eigenvalores de mayor a menor. Posteriormente, se seleccionan los eigenvalores más grandes, esto es, lo que aportan más información, y se obtienen los componentes principales multiplicando los eigenvectores que corresponden a cada eigenvalor con la matriz de la base de datos de la siguiente manera , donde B es la matriz de la base de datos, son los eigenvectores de la matriz de covarianza y el componente principal resultante.

Código:

```
m = matriz_cov(datosAjustados)
m

L,V = np.linalg.eig(m)
# Eigenvalores
L

# Obtener el porcentaje de covarianza de cada uno de los atributos
total = L.sum()
p = (L/total)*100
p
```

El algoritmo PCA demuestra que los únicos atributos que aportan más información son «Pclass», «Gender» y «Age», por lo que son los que serán usados para la clasificación de los datos, dando una ganancia mayor al 90 %.

```
Pclass = 0.5068884066917456
Gender = 0.3582257646393924
Age = 0.06133823814795482
SibSp = 0.0394749234065113
Parch = 0.022221035379827194
Fare = 0.011851631734568829
```

Se puede demostrar que, con estos atributos, se puede tener una mayor exactitud que utilizando todos, además de la mejora que es observable con las métricas de error, sirve para determinar los atributos que aportan más información y reduce el tiempo de cómputo.

Para demostrar esto, nos adelantaremos un poco utilizando un método que se explica con otro ejemplo en la sección 8.15. Este método es el de K-vecinos cercanos (KNN).

Para este ejercicio, vamos a utilizar KNN con todos los atributos y con los atributos que me generaron los componentes principales. Las pruebas se realizarán con el método de 80-20 y las métricas de precisión, exactitud, sensitividad y f1 score (puntaje f1), en este caso. La separación de datos de prueba y entrenamiento y la función para el cálculo de las métricas mencionadas se muestran en el siguiente código:

Código:

```
# Separación de los datos 80 / 20 para entrenamiento y validación
o = len(datos.columns)
n = len(datos)
p = int((80 * n)/100)
# Obtener una muestra del 80 % de los datos de manera aleatoria
m = np.arange(n)
train = np.random.choice(m,p,replace = False)
```

```

test = []
# Todos aquellos indices que no están en train se utilizarán para testing
for i in m:
    if i not in train:
        test.append(i)

# Se seleccionan los datos de entrenamiento
X_train = datos.iloc[train,[1,2,3,4,5,6]]
true_train = datos.iloc[train,0]

# Se seleccionan los datos de prueba
X_test = datos.iloc[test,[1,2,3,4,5,6]]
true_test = datos.iloc[test,0]
#Función que obtiene las métricas del algoritmo para su validación
def metricas(claseP,true_train):
    TP = 0
    FP = 0
    TN = 0
    FN = 0
    # Determinar los TP, TN, FP y FN para la matriz de confusión del entrenamiento
    for i in range(len(claseP)):
        if (true_train[i] == claseP[i]) and true_train[i] == 1:
            TP += 1
        elif (true_train[i] == claseP[i]) and true_train[i] == 0:
            TN += 1
        elif (true_train[i] != claseP[i]) and true_train[i] == 0:
            FP += 1
        else:
            FN += 1
    accuracy = ((TP+TN) / (TP+TN+FP+FN))

    if TP + FP != 0:
        precision = TP/(TP+FP)
    else:
        precision = 0
    if TP + FN != 0:
        sensitividad = TP/(TP+FN)
    else:
        sensitividad = 0
    if precision != 0 and sensitividad != 0:
        f1 = (2*TP)/(2*TP+FP+FN)
    else:
        f1 = 0
    return [accuracy, precision, sensitividad, f1]

```

Se realiza el entrenamiento y prueba de KNN utilizando todos los atributos como se muestra en el siguiente código:

Código:

```
def knn(data,labels, k):
predict = []
for i in range(0,len(data)):
# Inicialización de las distancias
colección_distancias = []
data = np.array(data)
for index ,example in enumerate(data):
distance = np.linalg.norm(example - data[i]) #Obtener distancia euclíadiana entre dos puntos de n
dimensiones
colección_distancias.append((distance, index))
colección_distancias.pop(i)
colección_ordenada = sorted(colección_distancias)
# Se toman las primeras k muestras del arreglo ordenado
k_muestras = colección_ordenada[:k] #para k = 1
# Se obtiene el valor de los k vecinos más cercanos
índices = []
for i in (k_muestras):
índices.append(i[1])
k_muestras_labels = labels.iloc[índices]
predict.append(Counter(k_muestras_labels).most_common(1)[0][0])
return predict

Clasificación de Survived utilizando todos los atributos
error_rate = []

n = int(len(X_train)/20)
for k in range(2,n):
pred_i = knn(X_train.values.tolist(),true_train,k)
error_rate.append(np.mean(pred_i != true_train))
plt.figure(figsize=(10,6))
plt.plot(range(2,n),error_rate,color='blue', linestyle='dashed', marker='o',
markerfacecolor='red', markersize=10)
plt.title('Error Rate vs. K Value')
plt.xlabel('K')
plt.ylabel('Error Rate')

# Se toma el tiempo de entrenamiento de KNN
inicio = time.time()
train_pred = knn(X_train.values.tolist(),true_train,9)
final = time.time()
print('Tiempo de entrenamiento:' + str(final-inicio))

# Se calculan las métricas de error
accuracy, precision, sensitividad, f1 = metricas(train_pred,np.array(true_train))

print('Exactitud: ' + str(accuracy))
print('Precisión: ' + str(precision))
print('Sensitividad: ' + str(sensitividad))
print('f1 Score: ' + str(f1))
```

```

y_pred= knn(X_test.values.tolist(),true_test,9)
accuracy, precision, sensitividad, f1 = metricas(y_pred,np.array(true_test))

print('Exactitud: ' + str(accuracy))
print('Precisión: ' + str(precision))
print('Sensitividad: ' + str(sensitividad))
print('f1 Score: ' + str(f1))

```

Las métricas de error con todos los atributos quedarían como en la tabla 8.1. Es importante mencionar que el error no necesariamente es exactamente igual. Esto es debido al proceso de separación de datos, ya que esto es aleatorio.

Clasificación de sobrevivientes utilizando todos los atributos		
Métrica	Entrenamiento	Pruebas
Tiempo		8.0924
Exactitud	0.7837	0.8268
Precisión	0.6423	0.6911
Sensitividad	0.7586	0.8245
Puntaje F1	0.6956	0.752

Tabla 8.1. Error con todos los atributos para comparación con PCA

Por otro lado, utilizando los componentes principales que se obtuvieron previamente, hacemos el entrenamiento y prueba como se muestra en el siguiente código:

Código:

```

error_rate = []

n = int(len(X_train)/20)
for k in range(2,n):
    pred_i = knn(X_train.iloc[:,[0,1,2]].values.tolist(),true_train,k) #chechar función
    error_rate.append(np.mean(pred_i != true_train))
plt.figure(figsize=(10,6))
plt.plot(range(2,n),error_rate,color='blue', linestyle='dashed', marker='o',
markerfacecolor='red', markersize=10)
plt.title('Error Rate vs. K Value')
plt.xlabel('K')
plt.ylabel('Error Rate')
x = X_train.iloc[:,[0,1,2]].values.tolist()
inicio = time.time()
train_pred = knn(x,true_train,8)
final = time.time()
print('Tiempo de entrenamiento:' + str(final-inicio))
accuracy, precision, sensitividad, f1 = metricas(train_pred,np.array(true_train))

print('Exactitud: ' + str(accuracy))
print('Precisión: ' + str(precision))
print('Sensitividad: ' + str(sensitividad))
print('f1 Score: ' + str(f1))
y_pred2= knn(X_test.iloc[:,[0,1,2]],true_test,8)
accuracy, precision, sensitividad, f1 = metricas(y_pred2,np.array(true_test))

print('Exactitud: ' + str(accuracy))

```

```

print('Precisión: ' + str(precision))
print('Sensitividad: ' + str(sensibilidad))
print('f1 Score: ' + str(f1))

```

La tabla 8.2 muestra la clasificación de sobrevivientes utilizando solo los componentes principales.

Clasificación de sobrevivientes utilizando los componentes principales		
Métrica	Entrenamiento	Pruebas
Tiempo		5.9173
Exactitud	0.7893	0.8324
Precisión	0.6094	0.7058
Sensitividad	0.7952	0.8275
Puntaje F1	0.6900	0.7619

Tabla 8.2. Error con los componentes principales usando PCA

Como se puede observar en las tablas 8.1 y 8.2, usar PCA como un preprocesamiento previo a la clasificación puede ayudar a mejorar algunas métricas. Por ejemplo, el tiempo se mejoró en una tercera parte, mientras que hay algunas métricas que no exhiben mejoras sustanciales –por ejemplo, exactitud de entrenamiento–, hay algunas métricas que solo con utilizar PCA mejoraron.

Se recomienda, además del uso de componentes principales, utilizar otras técnicas de optimización de hiperparámetros, normalización, otras técnicas de imputación para mejorar el rendimiento del algoritmo como se explicó en el capítulo 7.

8.9. Creación de datos sintéticos

Los datos sintéticos, como su nombre lo indica, son datos que son creados artificialmente en lugar de ser obtenidos mediante mediciones o eventos. Generalmente, se utilizan algoritmos para aumentar el número de datos, ya sea porque las pruebas lo requieren o no existen suficientes instancias para entrenar un algoritmo. Otro nombre que adquieren la creación de datos sintéticos es la de *data augmentation*.

Existen varias razones por las cuales los datos sintéticos son importantes, algunas de ellas son:

- Cuando se requieren datos para prueba, pero no están disponibles en ese momento o no existen.
- Cuando se tiene una cantidad pequeña de instancias y no alcanzan para hacer tanto el entrenamiento como las pruebas.
- Cuando se requieren datos de entrenamiento, sin embargo, es costoso o inviable realizarlos. Por ejemplo, datos de choques de vehículos autónomos, entre otros.

Existen muchos beneficios de crear y usar datos sintéticos. En muchas ocasiones, los desarrolladores necesitan la flexibilidad para poder probar un algoritmo y no se tienen todos los datos disponibles o se requiere de mucho tiempo para poder adquirir y preparar suficientes datos.

Otra de las ventajas es que se tiene completa libertad sobre la cantidad y la distribución de los datos de un modelo. También se pueden hacer pruebas cuando las clases están desbalanceadas. Esto significa que hay muchas más instancias que pertenecen a una clase que a las demás. Esto, para algunos algoritmos, puede ser un problema y afecta a su rendimiento, mientras que teniendo datos sintéticos se puede observar el rendimiento de los algoritmos mucho antes de hacer pruebas con datos reales.

Otro de los beneficios de crear datos sintéticos es el de proteger la privacidad de los datos reales mientras se trabaja con el modelo utilizando datos sintéticos. Para muchas aplicaciones, los datos reales contienen información privada y sensible como datos biométricos, teléfonos, direcciones, coordenadas de GPS en

tiempo real, entre otros. En estos casos, se crean datos sintéticos para salvaguardar la información sensible y confidencial.

Existen diferentes tipos de datos sintéticos, los cuales tienen diversos propósitos. En general, se pueden dividir de la siguiente manera:

- Texto sintético.
- Datos sintéticos multimedia –como videos, imágenes, sonidos, etcétera–.
- Datos tabulares sintéticos.

Los textos sintéticos han sido un tema complejo hasta hace unos años. Crear texto en un idioma que haga sentido, con las complejas reglas gramaticales que existen en prácticamente cualquier idioma, no es una tarea simple, aunque hoy en día, existen nuevos modelos de aprendizaje máquina que pueden generar texto convincentemente bueno utilizando algoritmos de generación de lenguaje natural.

Los datos sintéticos también pueden ser multimedia, como imágenes, video o sonidos. Usualmente, los datos sintéticos multimedia se crean utilizando características similares a los datos que ya se tienen, ya sea para agregar datos al entrenamiento –las redes profundas, por ejemplo, suelen utilizar un gran número de datos y, si se está utilizando un algoritmo para, por ejemplo, clasificar imágenes, en ocasiones, se requieren de una mayor cantidad de datos– o para reemplazar datos multimedia sensibles o confidenciales –por ejemplo, los datos biométricos, como se comentó anteriormente–.

Por último, los datos tabulares sintéticos se asemejan a los datos reales que aparecen en forma de tabla. Este tipo de tablas aparecen regularmente en las bases de datos en forma de filas –instancias– y columnas –atributos–. Es el tipo más común de datos sintéticos.

En este ejercicio, crearemos datos sintéticos tabulares. El código de este ejercicio se muestra en esta sección y puede ser descargado en: http://www.amese.net/libro_ia/Prog/8.9_Sinteticos.ipynb

Lo primero que vamos a hacer es cargar las librerías, como se muestra en el siguiente código:

Código:

```
# Librerías
import os
import random
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import plotly.figure_factory as ff
```

Prácticamente, todas las librerías usadas en este ejercicio ya se habían usado, probablemente con la excepción de plotly, misma que se va a utilizar para graficar más adelante. En caso de no tener alguna librería, se recomienda desde la terminal instalarla de la siguiente manera:

```
python -m pip install -U plotly
```

O:

```
pip install plotly
```

Recuerda reemplazar «plotly» por la librería que te haga falta.

Lo siguiente es poner la ruta y nombre del archivo donde se localiza la base de datos. Primeramente, se muestra el código para teclear la ruta de la base de datos. Si la base de datos se encuentra en el mismo

directorío de trabajo de la libreta, solo presionar «Enter».

Código:

```
# Pedir al usuario ingresar la ruta donde se encuentra su base de datos
print("Ejemplo de la ruta de la Base de Datos \\Users\\Aceves\\Documents\\Ejercicios\\")
path=str(input("Teclea la Ruta donde se localiza tu Base de Datos:"))
```

Posteriormente, se ingresa el nombre de la base de datos sin la extensión, en este caso. Se asume que es .csv y se concatena al nombre del archivo posteriormente, como se muestra en el siguiente código:

Código:

```
file = str(input("¿Cómo se llama tu Archivo? "))
if not ".csv" in file:
    file += ".csv"
```

Para este ejercicio, se va a utilizar la base de datos de gorriones, como se explica a detalle en el apéndice A.3.

Como se puede ver en la figura 8.40, la base de datos se muestra de manera correcta. En dicha base de datos, se cuenta con cinco atributos de la longitud de diferentes gorriones medidos durante una tormenta. Los atributos son: X1, X2, X3, X4 y X5. El sexto atributo es el de decisión llamado «supervivencia», el cual muestra si el gorrión sobrevivió a la tormenta o no.

39	163	249	33.40	19.5	22.8	Superviviente
40	163	242	31.00	18.1	28.7	Superviviente
41	159	238	31.20	18.1	28.7	No superviviente
42	159	238	31.20	18.4	29.3	No superviviente
43	161	245	32.10	19.1	28.8	No superviviente
44	162	245	31.90	17.1	19.4	Superviviente
45	162	247	31.90	18.1	28.4	No superviviente
46	153	237	30.60	18.6	28.4	No superviviente
47	162	245	32.50	18.5	21.1	No superviviente
48	164	248	32.30	18.8	28.9	Superviviente

Figura 8.40. Muestra de la base de datos de gorriones para datos sintéticos

Ahora vamos a mostrar el número total de instancias de todos los atributos incluyendo el atributo de decisión, como se muestra en el siguiente código:

Código:

```
# Obtener el número de celdas que tiene la base de datos
total_cells = np.product(data.shape)
total_cells
```

En el caso de la base de datos usada para el presente ejercicio, dará una salida de 294. Posteriormente, se va a ingresar –en porcentaje– el número de datos sintéticos que se quiere agregar, como se muestra en el siguiente código:

Código:

```
# Pedir al usuario ingresar el porcentaje de datos sintéticos que requiere generar
num_data=(input("Ingresa el porcentaje de datos sintéticos que quieras generar: "))
```

Lo cual dará la siguiente salida. En este ejemplo, se creará el 30 % de datos sintéticos, como se muestra a continuación:

Ingresá el porcentaje de datos sintéticos que quieras generar: 30.

De acuerdo al porcentaje ingresado, define el número de instancias sintéticas que se van a crear y hace una copia de la base de datos original. Es importante esta copia para poder comparar la distribución de los atributos antes y después de la generación de los datos y, así, asegurarse de que no cambie mucho la distribución. El código para realizar esto se muestra a continuación:

```
Código:
# Definir el número de datos a generar con base en el porcentaje indicado
N_datos=total_cells*(int(num_data)/100)
N_datos=int(N_datos)
N_datos

# Realizar copia de la base de datos
datos=data.copy()
datos
```

En este caso que quisimos el 30 % de datos sintéticos, nos da un resultado de 88. Es decir, creará 88 datos nuevos tomando en cuenta la distribución de los datos originales.

Lo siguiente a considerar son las características de la base de datos. Como se ha comentado en algunas ocasiones, es fundamental conocer la base de datos. Las características que se van a necesitar se muestran mediante «`datos.describe()`», mismas que muestran el número de instancias, media, desviación estándar y los valores en sus diferentes cuartiles. Esto se muestra en la figura 8.41.

```
Código:
# Conocer los datos que están ingresando
datos.describe(include = 'all')

# Definir la desviación estándar de los datos originales
data_std=datos.std()
```

	X1	X2	X3	X4	X5	Supervivencia
count	49.000000	49.000000	49.000000	49.000000	49.000000	49
unique	NaN	NaN	NaN	NaN	NaN	2
top	NaN	NaN	NaN	NaN	NaN	No superviviente
freq	NaN	NaN	NaN	NaN	NaN	29
mean	157.979592	241.326531	31.459584	18.469388	20.826531	NaN
std	3.654277	5.067822	0.795199	0.564286	0.991374	NaN
min	152.000000	230.000000	36.100000	17.200000	18.800000	NaN
25%	155.000000	238.000000	36.900000	18.100000	20.200000	NaN
50%	158.000000	242.000000	31.500000	18.500000	20.700000	NaN
75%	161.000000	245.000000	32.000000	18.800000	21.500000	NaN
max	165.000000	252.000000	33.400000	19.800000	23.100000	NaN

Figura 8.41. Distribución estadística de los datos por atributo para el ejemplo de datos sintéticos

Ahora hay que definir los atributos y mostrar la distribución de los datos. Para esto, se utilizará «`displot`» de la librería seaborn, como se muestra a continuación:

```
Código:
# Definir los atributos
columns_df=list(datos.columns)
columns_df

# Obtener la distribución de datos
sns.displot(datos,kde=True)
for i in range(len(columns_df)):
    colum=columns_df[i]
    sns.displot(datos[colum],kde=True)
plt.title('Distribucion')
```

En la figura 8.42, se muestra un ejemplo de un histograma con la distribución de los datos por atributo –en este caso, X4–.

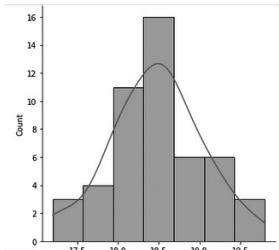


Figura 8.42. Ejemplo de un histograma para la distribución de datos para el ejemplo de datos sintéticos

Hasta el momento, no hemos creado datos sintéticos, se ha definido la base de datos, porcentaje de datos sintéticos deseados y algunas estadísticas para conocer los datos a partir de los cuales se crearán los datos sintéticos. El paso posterior es un preprocessamiento de datos, cómo obtener datos faltantes en total y por atributo e imputar los datos. Esto se muestra en el siguiente código:

Código:

```
# Inspeccionar si hay datos faltantes
datos.info()

# Eliminar atributos duplicados
datos=datos.drop_duplicates()

# Contar valores faltantes
missing_values_count = datos.isnull().sum()
missing_values_count

# Contar valores Nan
nan_values_count = datos.isna().sum()
nan_values_count

# Obtener el porcentaje de datos faltantes en la base de datos
total_missing = missing_values_count.sum()
total_nan = nan_values_count.sum()

# Porcentaje de datos faltantes
total_missing_porcent = total_missing/total_cells*100
print(total_missing_porcent)

# Porcentaje de datos no numéricos (nan)
total_nan_porcent = total_nan/total_cells*100
print(total_nan_porcent)

# Porcentaje de datos nulos por columna
for col in datos.columns:
    pct_missing = np.mean(datos[col].isnull())
    print('{} - {}%'.format(col, round(pct_missing*100)))

# Crear indicador para las características con datos faltantes
for col in datos.columns:
    missing = datos[col].isnull()
    num_missing = np.sum(missing)

# Indicador de atributo donde es necesario imputar
if num_missing > 0:
```

```

print('Indicador de Dato Faltante para: {}'.format(col))

# Imputar los valores perdidos y crear las variables indicadoras
# de valores perdidos para cada columna numérica.
datos_numeric = datos.select_dtypes(include=[np.number])
numeric_cols = datos_numeric.columns.values

for col in numeric_cols:
    missing = datos[col].isnull()
    num_missing = np.sum(missing)

    if num_missing > 0: #imputacion a celdas nulas
        print('imputing missing values for: {}'.format(col))
        med = datos[col].median()
        datos[col] = datos[col].fillna(med)

# Imputar los valores perdidos y crear las variables indicadoras
# de valores perdidos para cada columna no numérica.
datos_non_numeric = datos.select_dtypes(exclude=[np.number])
non_numeric_cols = datos_non_numeric.columns.values

for col in non_numeric_cols:
    missing = datos[col].isnull()
    num_missing = np.sum(missing)

    if num_missing > 0: # imputación para las columnas que tienen valores perdidos.
        print('imputing missing values for: {}'.format(col))
        top = datos[col].describe()['top'] # imputar con el valor más frecuente.
        datos[col] = datos[col].fillna(top)

# Datos resultantes al concluir la limpieza de la base de datos
datos

```

En este caso, no hay valores faltantes. Sin embargo, en caso de encontrar valores que no haya datos en algunos atributos, realizará una imputación simple tanto para los casos donde no hay un valor como en los casos en los cuales el valor se espere numérico, pero no lo sea. Por último, en el código anterior, se muestra los datos resultantes después de este preprocesamiento.

Antes de realizar la implementación para crear datos sintéticos, se vuelve a obtener la distribución de los datos y la desviación estándar. Esto se hace debido a que es importante mantener una distribución lo más parecida posible a la original para no sesgar el entrenamiento posterior y que las pruebas del algoritmo me muestren resultados diferentes a los esperados y un rendimiento irreal debido solamente a que eliminé instancias, las imputé o agregué datos sintéticos. El código para volver a realizar esta verificación en la distribución se muestra a continuación:

Código:

```

# Revisar la desviación estándar original
data_std

# Obtener desviación después de realizar imputación para comprobar que esta sea correcta
datos.std()

```

Una vez realizado todo este procedimiento, comenzamos a crear los datos sintéticos. El primer paso es revisar la cantidad de datos únicos.

datos.nunique(axis=0) #Cantidad de datos únicos en cada atributo, esto es, el número de observaciones de cada atributo. El obtener cada observación nos va a ayudar a la creación de los datos sintéticos, pues cada valor tendrá una probabilidad de ser tomado. El código se muestra a continuación:

Código:

```
datos.nunique(axis=0) #Cantidad de datos únicos en cada atributo

# Obtener los datos únicos por atributo
uniq_data_clums=[]
for i in range(len(columns_df)):
    variables = datos[columns_df[i]].unique()
    uniq_data_clums.append(variables)
uniq_data_clums
```

El resultado de realizar esto se muestra como una cadena de valores, como se muestra en la figura 8.43.

```
[array([156, 154, 153, 155, 163, 157, 164, 158, 160, 161, 159, 152, 165,
       162]),
 array([245, 240, 236, 243, 247, 238, 239, 248, 244, 246, 235, 237, 242,
       232, 250, 231, 252, 230, 249]),
 array([31.6 , 30.4 , 31. , 30.9 , 31.5 , 32. , 32.8 , 32.7 , 31.3 ,
       31.1 , 32.32, 31.4 , 30.5 , 30.3 , 32.5 , 32.6 , 31.7 , 32.2 ,
       33.1 , 30.1 , 31.8 , 31.9 , 30.8 , 31.2 , 33.4 , 32.1 , 30.7 ,
       30.6 , 32.3 ]),
 array([18.5 , 19.1 , 18.4 , 17.7 , 18.6 , 19. , 19.1 , 18.8 , 19.3 , 18.1 , 18. ,
       18.2 , 17.2 , 19.1 , 19.8 , 17.3]),
 array([28.5 , 19.6 , 20.6 , 20.2 , 20.3 , 20.9 , 21.2 , 21.1 , 22. , 21.8 , 20. ,
       19.8 , 21.6 , 20.1 , 21.9 , 21.5 , 20.7 , 21.7 , 22.5 , 21.4 , 22.7 , 23.1 ,
       21.3 , 19. , 22.2 , 18.6 , 19.3 , 22.8 , 20.8 , 20.4]),
 array(['Superviviente', 'No superviviente'], dtype=object)]
```

Figura 8.43. Instancias con observaciones únicas para el ejercicio de creación de datos sintéticos

 A partir de los valores únicos, se crearán los datos sintéticos. La manera más sencilla de crear datos sintéticos tabulares es tomando valores de manera aleatoria de las observaciones del código anterior, mismas que se muestran en la figura 8.43. Sin embargo, en este ejercicio se optó por realizarlo mediante un método parecido al método estocástico de ruleta que se utiliza mucho como operador genético de selección en cómputo evolutivo. Un ejemplo de cómo funciona el método de ruleta tradicional en este tipo de algoritmos se muestra en la figura 8.44.

Código:

```
# Generar datos sintéticos a partir de método ruleta
bd=[]
for j in range(N_datos):
    row=[]
    for i in range(len(columns_df)):
        # Valor único
        item_counts_key = datos[columns_df[i]].value_counts().index.tolist()

        # Probabilidad de que este valor aparezca en la base de datos
        item_counts = datos[columns_df[i]].value_counts(normalize=True)

        # Suma acumulativa de la probabilidad para priorizar valores
        cumsumP=item_counts.cumsum()
        w=cumsumP.to_numpy()

        # Generar un porcentaje aleatorio
        r=random.random()
```

```

# Buscar el porcentaje más cercano alcanzado por los valores únicos
magico=np.argwhere(r <= w)

# Obtener el índice de cada valor
x=int(magico[0])

# Obtener el valor para generar dato sintético
dato=item_counts_key[x] #Obtener el valor para generar dato sintético
row.append(dato)

# Agregar el valor creado sintéticamente a la BD
bd.append(row)

# Crear un macro de datos de los datos sintéticos
d_syn=pd.DataFrame(bd, columns=columns_df)
d_syn

```

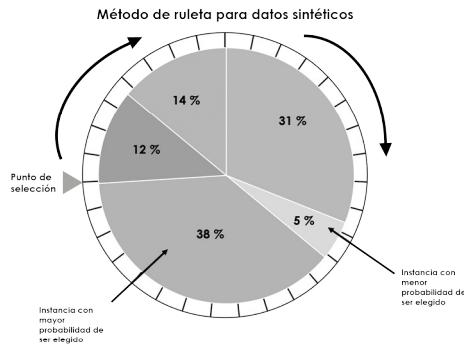


Figura 8.44. Ejemplo de ruleta para la selección de datos aleatorios de las observaciones de datos sintéticos

La razón por la cual se realizó en este ejemplo un método de ruleta es debido a que, si solo se toma en cuenta el mínimo y máximo y se genera un número aleatorio entre dichos valores, la distribución de los datos después de agregar los datos sintéticos puede cambiar drásticamente, lo cual, ya se comentó, puede ser un problema en la etapa de entrenamiento.

Por último, se agregan los datos creados artificialmente a los datos originales y se muestra la distribución anterior y posterior a modo de comparación. La figura 8.45 muestra la cantidad de datos creados de manera sintética junto con los datos originales, mientras que la figura 8.46 muestra un ejemplo de las distribuciones de los datos, misma en la que puede observarse que las distribuciones son casi idénticas. El código se muestra a continuación:

Código:

```

# Crear un macro de datos de los datos sintéticos
d_syn=pd.DataFrame(bd, columns=columns_df)
d_syn

# Agregar datos sintéticos a la copia de los datos originales
d_final=pd.concat([datos, d_syn], ignore_index=True)
d_final

# Obtener la distribución de datos
sns.displot(d_final,kde=True)
for i in range(len(columns_df)):

```

```

colum=columns_df[i]
sns.displot(d_final[colum],kde=True)
plt.title('Distribucion')

# Desviacion est醖ard de los datos sint閠icos
d_final.std()

# Desviacion est醖ard original
data_std

for i in range(len(columns_df)):
    colum=columns_df[i]
    datos[colum].plot.kde()
    d_final[colum].plot.kde()
    plt.title('Distribucion')
    plt.show()

```

	X1	X2	X3	X4	X5	Supervivencia
0	156	245	31.6	18.5	20.5	Superviviente
1	154	240	30.4	17.9	19.6	No superviviente
2	153	240	31.0	18.4	20.6	No superviviente
3	153	236	30.9	17.7	20.2	No superviviente
4	155	243	31.5	18.6	20.3	Superviviente
...
131	164	249	31.4	19.1	20.1	No superviviente
132	163	244	30.6	17.2	19.0	No superviviente
134	155	246	33.4	19.1	20.1	No superviviente
135	153	232	30.4	18.6	22.5	Superviviente
136	155	248	31.6	19.1	21.1	Superviviente

137 rows x 6 columns

Figura 8.45. Extracto de los datos creados sint閠icamente en la base de datos

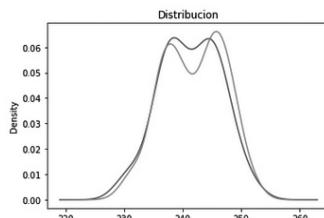


Figura 8.46. Comparaci髇 entre la distribuci髇 de datos original y cuando se agregan datos sint閠icos

8.10. T閏nica de imputaci髇 m閑ltiple MICE

Como se comentó en la sección 4.5 «Imputación de datos», cuando hacen falta datos es útil evaluar las diferentes estrategias para poder reemplazar dichos datos faltantes.

Una de las técnicas más utilizadas es la de imputación. En esta práctica, se abordará la implementación de una de las imputaciones múltiples con mejor funcionamiento: la de imputación múltiple por ecuaciones encadenadas (o MICE, por sus siglas en inglés).

El código se muestra en esta sección y puede ser descargado en:
http://www.amese.net/libro_ia/Prog/8.

El objetivo de la imputación es habilitar el uso de los datos para realizar inferencias y conclusiones válidas. Si la variable aleatoria analizada es **X** y contiene datos faltantes, se debe tomar en cuenta que, una vez hecha la imputación el estimador de **X**, debe ser cercano a los valores de los parámetros de la distribución de **X**, es decir, la varianza del estimador será pequeña y con sesgo mínimo. Esto justifica conocer *a priori* las distribuciones de los datos para cada uno de los atributos de la base de datos y proporciona una base para medir, al menos cualitativamente, la calidad de la imputación.

Existen estudios que muestran que la selección de un método de imputación simple contra un método de imputación múltiple depende de la correlación que exista entre los atributos de la base de datos. Si existe una correlación, es recomendable seleccionar un algoritmo de imputación múltiple, por lo que primeramente se leerán los datos y se hará una correlación entre sus atributos. La base de datos utilizada en este ejemplo es el de datos ambientales de partículas contaminantes PM10, cuyas características se muestra en el apéndice A.7.

El algoritmo de imputación múltiple por ecuaciones encadenadas asume que los datos faltantes son MAR (faltantes de manera aleatoria, por sus siglas en inglés), lo que significa que la probabilidad de un valor faltante depende de los valores observados. La correlación entre los atributos podría sustentar este hecho. De igual forma, cuando los datos faltantes no cumplen esta propiedad, el algoritmo generalmente converge, pero los estimadores son sesgados.

Una gran ventaja de utilizar MICE es que es posible ejecutar diferentes modelos de regresión para cada variable. Esto significa que cada atributo puede ser modelado bajo su propia distribución, es decir, atributos binarios se pueden estimar con una regresión logística mientras que valores continuos, con una regresión lineal o polinómica.

El algoritmo se resume en cuatro pasos:

1. Se realiza una imputación simple para todos los valores faltantes.
2. Se selecciona un atributo y los valores que se imputaron se remueven.
3. Los valores en el atributo seleccionado en el paso dos se imputan utilizando un modelo de regresión sobre los demás atributos —aquellos que aún contienen la imputación simple—. El modelo de regresión puede incluir todos los atributos o un subconjunto.
4. Los pasos 2 y 3 se repiten seleccionando diferentes atributos hasta que todos los atributos hayan sido imputados con un modelo de regresión.

Código:

```
#Se importan las librerías
import matplotlib.pyplot as plt
from sklearn import linear_model
from copy import deepcopy
import pandas as pd
import numpy as np
import csv

# Leer los datos
df = pd.read_csv('2020PM10.csv')
df = df.drop(columns=['FECHA', 'HORA'])
df = df.replace(-99, np.nan)

# df_clean contiene los datos con datos sin atributos inválidos
df_clean = df.dropna(axis=1, how='all')
origi_df = df_clean
df_clean.corr()
```

	ACO	AJM	ATI	BJU	CAM	CUA	CUT	FAC	GAM	HOM	—	MER	PED	SAC	SPE	TA
ACO	1.000000	0.307682	0.408689	0.537182	0.480664	0.305601	0.516079	0.507420	0.597294	NAN	—	0.545022	0.487354	0.644601	0.48861	0.8891
AJM	0.307682	1.000000	0.430379	0.627712	0.586064	0.740360	0.282692	0.480681	0.438061	0.518430	—	0.51912	0.48664	0.38272	0.47819	0.4045
ATI	0.408689	0.430379	1.000000	0.537182	0.480664	0.740360	0.282692	0.480681	0.438061	0.518430	—	0.51912	0.48664	0.38272	0.47819	0.4045
BJU	0.537182	0.627712	0.537182	1.000000	0.710803	0.488711	0.470709	0.586088	0.720440	0.588897	—	0.77020	0.70794	0.680303	0.68101	0.8249
CAM	0.480664	0.590564	0.480681	0.710803	1.000000	0.454666	0.420517	0.598477	0.575108	0.410006	—	0.71130	0.69854	0.65217	0.60023	0.9055
CUA	0.305601	0.740354	0.415108	0.488711	0.454666	1.000000	0.306021	0.40707	0.520587	0.53813	0.620803	0.51419	0.78711	0.41056	—	0.41056
CUT	0.516079	0.280382	0.537087	0.470709	0.420517	0.306021	1.000000	0.610111	0.560730	0.480517	—	0.43811	0.57718	0.50890	0.54811	0.57245
FAC	0.507420	0.480581	0.742302	0.586088	0.538477	0.520587	0.610111	1.000000	0.595018	0.552088	—	0.57778	0.59418	0.53965	0.58003	0.48911
GAM	0.597294	0.430005	0.546213	0.732040	0.687408	0.40707	0.560730	0.595018	1.000000	0.695423	—	0.50303	0.49850	0.47697	0.51957	0.51865
HOM	0.518430	0.819134	0.586088	0.537087	0.520587	0.480517	0.560730	0.595018	0.695423	1.000000	—	0.74048	0.70827	0.68741	0.61517	0.49511
INN	0.305728	0.680795	0.377701	0.646182	0.386343	0.648072	0.282743	0.420516	0.334172	0.398606	—	0.59864	0.64486	0.59803	0.70603	0.39511
ICT	0.584600	0.620201	0.319345	0.719108	0.854056	0.804657	0.480517	0.57778	0.518097	0.717085	—	0.71130	0.69854	0.65217	0.60023	0.48621
MER	0.545045	0.619134	0.481550	0.772020	0.717100	0.584057	0.480517	0.57778	0.518097	0.717085	—	1.00000	0.62936	0.64746	0.58154	0.48956
PED	0.487354	0.809000	0.586088	0.537087	0.520587	0.480517	0.560730	0.595018	0.695423	0.717085	—	0.54747	0.59418	0.58745	0.48587	—
SAC	0.507041	0.280382	0.537079	0.600000	0.214108	0.586088	0.537087	0.520587	0.480517	0.560730	—	0.54747	0.59418	0.58745	0.48587	0.50069
SPE	0.440051	0.810100	0.511123	0.646182	0.586088	0.500000	0.57778	0.518097	0.610125	0.581154	—	0.501125	0.52031	0.53204	1.00000	0.50504
TIN	0.389128	0.490001	0.414440	0.480517	0.500000	0.419000	0.371021	0.487184	0.510023	0.487502	—	0.49002	0.48837	0.49860	0.50500	1.00000
TLA	0.490579	0.470987	0.617104	0.537080	0.702218	0.444041	0.502040	0.69600	0.691714	0.50302	—	0.42908	0.51715	0.56872	0.52017	0.54496
TUJ	0.205032	0.641500	0.769696	0.536967	0.502078	0.510194	0.720041	0.746742	0.550021	NAN	—	0.53979	0.69457	0.57777	0.62902	0.47276
UZ	0.603035	0.594995	0.495498	0.754457	0.646518	0.535055	0.480512	0.560591	0.771057	0.702488	—	0.77173	0.59153	0.40201	0.60464	0.54111
VF	0.601494	0.228101	0.522081	0.482028	0.404042	0.296037	0.666001	0.600699	0.704095	0.47773	—	0.47773	0.50588	0.59801	0.30763	0.50602
XAL	NAN	0.587095	0.479355	0.480654	0.458666	0.296180	0.536349	0.550093	0.790016	0.572098	—	0.681549	0.498321	0.65761	0.280740	0.50504

22 rows x 22 columns

Figura 8.47. Extracto de la base de datos ambiental del apéndice A.7

Se reemplaza todos los NAN (no es un número, por sus siglas en inglés) y se hace un conteo del porcentaje de datos válidos que cada estación de monitoreo tiene, posteriormente, se muestra gráficamente la distribución de cada característica, como se muestra en la figura 8.48 y 8.49, respectivamente.

Código:

```
# Generar histograma de las características
# y remover los valores inválidos
plt.rcParams.update({'figure.max_open_warning': 0})
for column in df:
    col_nan = df[column].replace(-99, np.nan)
    # Porcentaje de los datos por atributo
    percentage = (1 - col_nan.isna().sum() / (col_nan.count() + col_nan.isna().sum())) * 100
    print(f'{column} = {percentage}%')
    col_clean = col_nan.dropna()
    if len(col_clean.index) != 0:
        fig, axi = plt.subplots()
        axi.set_title(str(column))
        col_clean.hist(bins=50, ax=axi)
```

```
ACO = 52.800546448087424%
AJM = 20.480418943533695%
ATI = 92.213114754099365%
BJU = 87.51138433515483%
CAM = 74.70400728597451%
CHO = 0.0%
CUP = 44.66074481239615%
FAC = 83.8683970856102%
PAR = 0.0%
GAM = 55.04326047358834%
HOM = 9.384%
INN = 18.18159396071044%
IZT = 59.1871946594565%
MER = 78.10792349726776%
MGH = 0.0%
MPA = 0.0%
PED = 92.47495446265938%
SAC = 0.0%
```

Figura 8.48. Síntesis de porcentaje de valores inválidos de la base de datos

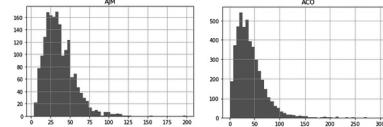


Figura 8.49. Ejemplo de una distribución de cada característica de la base de datos de partículas contaminantes.

De la lista de la figura 8.48, se puede observar que las siguientes estaciones no tienen ningún dato válido, por lo que no puede ser imputada:

- 1. CHO.
- 2. FAR.
- 3. MGH.

- 4. MPA.
- 5. SAC.

Así mismo, las siguientes estaciones tienen datos insuficientes para ser imputados:

- 1. HGM.
- 2. TLI.
- 3. XAL.

Posteriormente, se definen las estaciones –atributos– que vamos a tomar en cuenta para ser imputadas con las siguientes etiquetas:

Código:

```
# Tabla LUT (Look-up Table) que considera atributos con un alto porcentaje de
# disponibilidad de datos, es decir, por lo menos 50 % de datos
lookup_table = [
    'ACO',
    'AJM',
    'ATI',
    'BJU',
    'CAM',
    'CUA',
    'CUT',
    'FAC',
    'GAM',
    'IZT',
    'MER',
    'PED',
    'SAG',
    'SFE',
    'TAH',
    'TLA',
    'UIZ',
    'VIF'
]
```

El siguiente paso es realizar una imputación simple de los valores inválidos, como se muestra en el siguiente código:

Código:

```
# Reemplazar los -99 con una imputación simple (media)
df_clean = df_clean
[df_clean.columns.intersection(lookup_table)]
df_temp = df_clean.fillna(-99)
data = df_temp.to_numpy().tolist()
df_clean = df_clean.fillna(df_clean.mean())
means = df_clean.mean().to_numpy().tolist()
```

Una vez que se realizó la imputación simple, se identifican las instancias que serán imputadas con MICE con el siguiente código:

Código:

```
# Crear una LUT para guardar cuáles instancias serán imputadas
marked_cells = np.zeros([len(data), len(data[0])])

for row in range(len(data)):
    for col in range(len(data[0])):
        if data[row][col] == float(-99):
            marked_cells[row][col] = 1
            data[row][col] = means[col]

# Guardar una copia de los datos para compararlo con el error
reference = deepcopy(data[:])
```

Después se realiza una regresión multivariable con imputación MICE de la siguiente manera:

Código:

```
# Algoritmo MICE
finished = False
while not finished:
    for test_col in range(0, len(lookup_table)):
        # Se realiza la regresión lineal multivariable a todos los valores faltantes
        # que se requieren imputar
        identified_rows = []
        for row in range(len(data)):
            for col in range(len(data[0])):
                if test_col == col:
                    if marked_cells[row][col] == float(1):
                        identified_rows.append(row)
        # Se identifica los datos fijos que se usarán para
        # estimar los datos faltantes utilizando la regresión
        data_to_fit = []
        for row in range(len(data)):
            if not (row in identified_rows):
                data_to_fit.append(data[row])
        # En este punto, scikitlearn se utilizará para implementar MICE
        # el modelo de regresión lineal puede en este punto ser reemplazado por otro
        # modelo de regresión
        df = pd.DataFrame(data_to_fit)
        x_cols = [v for v in range(len(lookup_table)) if v != test_col]

        Y = df[test_col]
        X = df[x_cols]

        regr = linear_model.LinearRegression()
        regr.fit(X, Y)
        # Se predicen los valores faltantes
        for row in range(len(data)):
            for col in range(len(data[0])):
```

```

if col == test_col:
    if row in identified_rows:
        features = [data[row][v] for v in range(len(data[0])) if v != test_col]
        data[row][col] = regr.predict([features])

# Se compara la estimación previa y la actual
R = np.array(data) - np.array(reference)
finished = True
for row in range(len(R)):
    for col in range(len(R[0])):
        if R[row][col] > 5:
            finished = False
reference = deepcopy(data)

# Se convierten todos los valores a flotantes
for row in range(len(data)):
    for col in range(len(data[0])):
        data[row][col] = float(data[row][col])

```

Por último, se compara la distribución del algoritmo MICE con la distribución original sin imputar, como se muestra en el siguiente código y la figura 8.50.

Código:

```

# Se genera el histograma con las características
df = pd.DataFrame(data)
df.columns = lookup_table
plt.rcParams.update({'figure.max_open_warning': 0})
for column in df:
    col_nan = df[column].replace(-99, np.nan)
    col_clean = col_nan.dropna()

    if len(col_clean.index) != 0:

        min_val = df[column].min()
        max_val = df[column].max()
        mean = df[column].mean()
        std_dev = df[column].std()
        mice_minmax = df[column].apply(lambda x: (x-min_val)/(max_val-min_val))
        mice_zscore = df[column].apply(lambda x: (x-mean)/(std_dev))

        stats = pd.DataFrame(index=['count', 'mean', 'std', 'min', '25%', '75%', 'max'])

        stats['original_'+ column] = origi_df[column].describe()
        stats['mice_'+ column] = df[column].describe()
        stats['mice_minmax_'+ column] = mice_minmax.describe()
        stats['mice_zscore_'+ column] = mice_zscore.describe()

        col_origi = origi_df[column]

        fig, axs = plt.subplots(4, 2, figsize=(20, 15))
        axs[0,0].set_title('Original Distribution')

```

```

axs[1,0].set_title('MICE with Linear Regression Distribution')
axs[2,0].set_title('MICE and Norm by MinMax')
axs[3,0].set_title('MICE and Norm by z-score')
col_origi.hist(bins=50, ax=axs[0,0])
col_origi.plot.box(ax=axs[0,1])

col_clean.hist(bins=50, ax=axs[1,0])
col_clean.plot.box(ax=axs[1,1])

mice_minmax.hist(bins=50, ax=axs[2,0])
mice_minmax.plot.box(ax=axs[2,1])

mice_zscore.hist(bins=50, ax=axs[3,0])
mice_zscore.plot.box(ax=axs[3,1])

print(stats)

```

	original_A0	mice_A0	mice_minmax_A0	mice_zscore_A0
count	4636.000000	8784.000000	8784.000000	8.784000e+03
mean	40.650554	41.416086	0.186229	3.419780e-15
std	21.799554	25.140479	0.000055	1.000000e+00
min	1.000000	-66.694177	0.000000	-4.174968e+00
25%	21.000000	23.000000	0.237478	-7.110810e-01
75%	54.000000	54.825157	0.321748	5.179706e-01
max	311.000000	311.000000	1.000000	1.041115e+01
	original_AM	mice_AM	mice_minmax_AM	mice_zscore_AM
count	1799.000000	8784.000000	8784.000000	8.784000e+03
mean	19.522345	30.298909	0.182810	2.412980e-15
std	19.54129	15.298902	0.074273	1.000000e+00
min	4.000000	-6.983172	0.000000	-2.450557e+00
25%	22.000000	19.695861	0.129520	-7.067100e-01
75%	47.000000	38.157963	0.219150	5.000460e-01
max	199.000000	199.000000	1.000000	1.101335e+01
	original_ATI	mice_ATI	mice_minmax_ATI	mice_zscore_ATI
count	8106.000000	8784.000000	8784.000000	8.784000e+03
mean	34.394874	33.781863	0.034777	1.476708e-15

Figura 8.50. Diferencia de distribuciones de datos originales e imputados con MICE

8.11. Regresión lineal y polinomial

La regresión es un método para estudiar la relación entre una variable dependiente y y una variable independiente x . En esta práctica, abordaremos las más simples de las regresiones: la regresión lineal y la regresión polinomial. La regresión lineal se puede expresar de la siguiente forma:

$$y=f(x)=\beta_0+\beta_1x \quad (8.6)$$

Donde β_0, β_1 son los coeficientes del modelo lineal.

Como su nombre nos indica, establece una relación lineal entre x e y . También podemos establecer una relación no lineal con el fin de tener un modelo que asemeje mejor el comportamiento de los datos, a esto lo conocemos como regresión polinomial, y se expresa de la siguiente forma:

$$f(x)=\beta_0 + \beta_1x + \beta_2x^2 + \beta_3x^3 + \dots + \beta_nx^n \quad (8.7)$$

Normalmente, cuando se hace una regresión polinomial, no se toman polinomios de grado mayor a 3 o 4, porque para grados bastante altos, el polinomio tiende a tener demasiadas curvas, haciendo que se comporte de forma extraña.

El código se muestra en esta sección y puede ser descargado en:

http://www.ameise.net/libro_ia/Prog/8.

En este ejercicio, se usará la base de datos de partículas contaminantes PM10, que se explica a detalle en el apéndice A.7.

Primeramente, importamos librerías de utilidad como se muestra en el siguiente código:

Código:

```
import numpy as np
```

```

import pandas as pd
import matplotlib.pyplot as plt

```

Además, vamos a importar la clase *Polynomial* de *numpy*, la cual contiene la función *fit* que nos será muy útil para realizar las regresiones.

Código:

```
from numpy.polynomial import Polynomial as poly
```

Abrimos la base de datos que vamos a utilizar. Este es un fragmento de la base de datos completa 2020PM10.csv para exemplificar la regresión. Recuerda que se debe tener la base de datos «2020PM10_Muestra.csv» en el ambiente de Jupyter.

Código:

```

# Abrimos la BD
df = pd.read_csv("2020PM10_Muestra.csv")
df.head()

```

	FECHA	HORA	ATI	BJU	CUA	CUT	FAC
0	1/1/2020	1	102	62	116	138	70
1	1/1/2020	2	113	61	200	137	107
2	1/1/2020	3	123	63	138	192	114
3	1/1/2020	4	101	66	143	201	124
4	1/1/2020	5	76	103	88	160	91

Figura 8.51. Muestra de datos utilizadas para el ejercicio de regresión

Esta muestra contiene las lecturas de cinco locaciones distintas, aunque, para el propósito de esta práctica, solo necesitamos una de estas cinco columnas. Usaremos las lecturas de ATI y tomaremos veinticuatro observaciones que corresponden a las veinticuatro horas de un día. Entonces, filtramos las columnas sobre las cuales trabajaremos, usando la función *filter* de **pandas**, como se muestra en el siguiente código:

Código:

```

# Filtramos las columnas que ocuparemos en la regresión
df = df.filter(['FECHA', 'HORA', 'ATI'])
df.head()

```

	FECHA	HORA	ATI
0	1/1/2020	1	102
1	1/1/2020	2	113
2	1/1/2020	3	123
3	1/1/2020	4	101
4	1/1/2020	5	76

Figura 8.52. Muestra de datos filtrada por columnas utilizada para regresión

Determinamos el tamaño de la muestra que usaremos y filtramos los datos. Para exemplificar, en esta práctica, se utilizarán los primeros veinticuatro datos, cuyos índices corresponden al intervalo [0, 23], como se muestra en el siguiente código:

Código:

```

# Creamos un array con los índices que ocuparemos
muestra = np.arange(24)
# Filtramos los datos de acuerdo con el array
df = df.iloc[muestra]

```

Algo que debemos tomar en cuenta es que, en los datos que tenemos, se encuentran horas o días donde no se reportaron las concentraciones de PM10. Esos datos faltantes están representados con un -99. Como son

datos faltantes, los debemos filtrar para poder hacer nuestra regresión, ya que, de otro modo, nos afectarían al momento de realizar la regresión y tendríamos modelos erróneos. Otra estrategia sería realizar una técnica de imputación como se mostró en la sección 4.5 y la práctica de la sección 8.10, aunque, por simplicidad, se filtrarán los datos para esta práctica.

Código:

```
# Filtramos las columnas donde no se tenga la lectura (-99)
df = df[df['ATI'] != -99]
```

Preparamos los datos correspondientes al eje x y al eje y. Al eje x le corresponden el total de datos, y al eje y le corresponde la respuesta que se obtiene, que son las lecturas de partículas PM10, como se muestra en el siguiente código:

Código:

```
# Eje x corresponde a la hora
x = np.arange(len(df))
# Eje y corresponde al dato
y = df['ATI']
```

Ya que preparamos nuestros datos, vamos a crear una función llamada **regresión**, donde podamos elegir el grado del polinomio que queremos y lo podemos ver gráficamente. Además, vamos a calcular la raíz del error cuadrático medio (RMSE por sus siglas en inglés, *root mean square error*), un parámetro que nos indica la diferencia que hay entre los valores observados y los valores predichos por nuestro modelo. El RMSE se calcula de la siguiente forma:

$$RMSE = \sqrt{\frac{\sum_{i=1}^N (\hat{r}_i - r_i)^2}{N}} \quad (8.8)$$

\hat{r}_i es el valor predicho por el modelo, r_i es el valor observado y N es el número de observaciones que se tienen.

Esta función va a recibir como parámetros **x**, **y** y el grado del polinomio que queremos, además, va a tener un parámetro opcional «**precision**», que tiene por defecto el valor 4, este parámetro es para controlar el número de decimales que queremos ver cuando mostramos el RMSE.

Dentro de la función, lo primero es llamar a la función *fit* de *Polynomial*, esta función recibe **x**, **y** y el grado del polinomio, y nos devuelve el polinomio correspondiente que guardaremos en **f**. Después, inicializamos en 0 dos variables, **sumatoria**, donde iremos sumando la diferencia al cuadrado del valor observado y el valor predicho, e **i**, que nos servirá como contador para acceder a los datos del polinomio.

Este proceso lo implementaremos dentro de un *for* y se repetirá tantas veces como datos tengamos. Lo siguiente es dividir **sumatoria** entre el número de observaciones que tenemos y al resultado de esa división, le sacamos raíz cuadrada para obtener el RMSE.

Finalmente, procedemos a graficar.

Código:

```
def regresion(x, y, grado, precision=4):
    # Realizamos la regresión
    f = poly.fit(x, y, grado)
    # Calculamos la varianza
    sumatoria = 0
    i = 0
```

```

for v_observado in y:
    sumatoria += (v_observado - f(i)) ** 2
i += 1
# Número de observaciones
N = len(x)
# Dividimos entre el número de observaciones
sumatoria /= N
# Sacamos el RMSE y lo redondeamos
RMSE = round(np.sqrt(sumatoria), precision)
# Graficamos los datos observados
plt.plot(x, y, 'bo')
# Graficamos el modelo
plt.plot(x, f(x), 'r-')
# Ponemos en el título el grado del polinomio
plt.title("Polinomio grado " + str(grado) + "\n")
# Ponemos el RMSE en el título también
plt.title("RMSE: " + str(RMSE), loc="left")
# Ponemos las leyendas
plt.legend(['Datos', 'Polinomio'])
plt.show()

```

Ya que tenemos la función, podemos usarla para realizar la regresión del grado que queramos —para realizar la regresión lineal, basta con enviarle a nuestra función un 1 en el parámetro **grado**. Este grado lo podemos pedir por teclado, si así lo deseamos.

Código:

```

grado = int(input("¿Qué grado de polinomio desea?\n"))
regresion(x, y, grado)

```

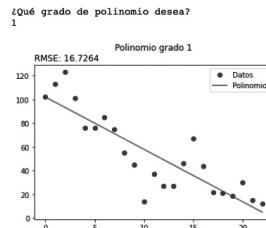


Figura 8.53. Regresión lineal con datos de contaminantes ambientales

Si se desea cambiar de grado, el algoritmo pedirá el número para el grado del polinomio. Las figuras 8.54 y 8.55 muestran los mismos datos, pero modelados con un polinomio de grado 3 y 5, respectivamente.

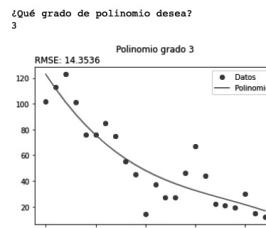


Figura 8.54. Regresión polinomial de grado 3 con datos de contaminantes ambientales

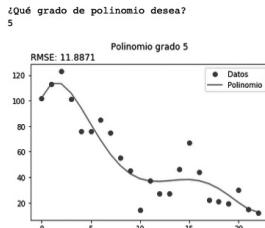


Figura 8.55. Regresión polinomial de grado 5 con datos de contaminantes ambientales

En esta práctica, se pueden ajustar tanto el número de datos como el grado del polinomio. También se puede reemplazar la sección que filtra los datos no válidos por una técnica de imputación (múltiple, MICE, por ejemplo).

8.12. Gradiente descendente

Para entender de manera simple cómo trabaja el algoritmo de gradiente descendente –en español, algunos textos lo manejan como descendiente al gradiente–, pongamos el ejemplo de un excursionista que está perdido en una montaña en un día con mucha neblina.

Debido a la neblina, le es imposible al excursionista ver más que unos cuantos metros a su alrededor. La primera impresión es que el excursionista está perdido, pero podría encontrar su camino. En este caso, la única manera de bajar la montaña es mirando sus propios pies e intentando caminar un paso a la vez, evaluando si está descendiendo conforme avanza.

Asumamos ahora que la montaña es convexa y existe un único punto mínimo –es decir, un óptimo o extremo global–. En este contexto, el excursionista podría saber que va en la dirección correcta caminando hacia donde la pendiente sea más pronunciada, pues significará que, efectivamente, está descendiendo la pendiente de la mejor manera. El excursionista repetiría ese proceso de la pendiente más pronunciada conforme vaya descendiendo de la montaña y, finalmente, llegue hasta abajo. El algoritmo de gradiente descendente funciona de esta manera.

El algoritmo del gradiente descendente es un algoritmo iterativo utilizado en problemas de optimización, donde el objetivo es minimizar una función tomando la dirección opuesta del gradiente. El gradiente es la primera derivada de una función. Dicho de otro modo, es la tasa de cambio de una función o qué tan rápido cambia una función en un punto dado en dirección ascendente. Entonces, para encontrar el mínimo de una función, utilizamos el gradiente negativo.

Si regresamos al ejemplo del excursionista, el algoritmo de gradiente descendente puede utilizar la dirección de la pendiente a partir de la posición del espacio de búsqueda. Si se conoce esta información, el algoritmo podría encontrar pesos ajustados de manera aleatoria para moverse a una nueva posición cerca del anterior destino –el cual no es el destino final–, y repetir estos pasos hasta que el error sea mínimo y se halla encontrado el mínimo global.

El código se muestra en esta sección y puede ser descargado en:
http://www.amese.net/libro_ia/Prog/8.

Para este método, se requiere una tasa de aprendizaje o paso **alfa** que nos indica qué tanto se «desplaza» nuestro algoritmo de optimización en cada iteración. Uno de los inconvenientes de este método es que la tasa de aprendizaje se mantiene constante durante toda la ejecución. Para mejorar esto, se han creado otros métodos derivados como, por ejemplo, el método Adam (*adaptive movement estimation algorithm*), que es el algoritmo que abordaremos en este ejemplo.

Veamos en qué consiste este algoritmo y lo que necesitamos para implementarlo:

```

Pseudocódigo:

 $\alpha$ : peso
 $\beta_1, \beta_2 \in [0, 1]$ : tasa de decremento exponencial para la estimación de momentos.
 $f(\theta)$ : función con parámetros  $\theta$ .
 $\theta_0$ : vector de parámetros iniciales.
 $m_0 \leftarrow 0$ : inicializar el vector del primer momento en 0.
 $v_0 \leftarrow 0$ : inicializar el vector del segundo momento en 0.
 $t \leftarrow 0$ : inicializar el tiempo en 0.
while  $\theta_t$  no converge do
     $t \leftarrow t + 1$ : incrementar el tiempo.
     $g_t \leftarrow \nabla_{\theta} f_t(\theta_{t-1})$ : obtener el gradiente de la función en el tiempo  $t$ .
     $m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$ : actualizar estimación sesgada del primer momento.
     $v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$ : actualizar estimación sesgada del segundo momento en bruto.
     $\hat{m}_t \leftarrow m_t (1 - \beta_2)$ : cálculo de la estimación del primer momento corregido por sesgo.
     $\hat{v}_t \leftarrow v_t (1 - \beta_2)$ : cálculo de la estimación del segundo momento en bruto corregido por sesgo.
     $\hat{\theta}_t \leftarrow \theta_{t-1} - \alpha \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$ : actualización de parámetros.
end while
return  $\theta_t$  (parámetros resultantes)

```

El pseudocódigo mencionado anteriormente corresponde al algoritmo Adam. En este método, se inicia en un punto aleatorio y , a partir de ahí, comienza la búsqueda del mínimo global.

Así mismo, se le pasan tres hiperparámetros (β_1 , β_2 y α). Los hiperparámetros β nos ayudan a controlar la tasa de decremento exponencial de la media móvil, mientras que α es el tamaño del paso máximo que el algoritmo puede dar. Es importante resaltar que el tamaño del paso se calcula en función del gradiente, permitiendo, de este modo, utilizar pasos de longitud variable que es justo lo que vuelve a este algoritmo más eficiente en comparación del algoritmo del gradiente descendente normal.

Para este ejemplo utilizaremos la función:

$$f(x_1, x_2) = a + e - ae^{-b\sqrt{\frac{1}{d}(x_1^2+x_2^2)}} - e^{\frac{1}{d}(\cos(cx_1)+\cos(cx_2))}$$

Donde a, b, c, d son constantes, y $x_1, x_2 \in [-15, 30]$

Importamos librerías de utilidad.

Código:

```

import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d
import random

```

Para poder trabajar con la función, debemos asignar valores a las constantes, en este caso, nosotros le asignaremos los siguientes valores $a = 20$, $b = 0.1$, $c = 3$, $d = 2$.

Ya con las constantes definidas, podemos implementar la función en Python. Dicha función tiene dos parámetros –dos entradas– y a la salida nos devuelve un único valor que es el resultado de la función evaluada en esos valores.

Código:

```

# Valores de las constantes
a, b, c, d = 20, 0.1, 3, 2

# Definición de la función f(t)
def funcion(x1,x2):
    return (
        a+np.exp(1)-a*np.exp(-b*np.sqrt((1/d)*(x1**2+x2**2)))
        -np.exp((1/d)*(np.cos(c*x1)+np.cos(c*x2)))
    )

# Se establece el intervalo de la función
lim = np.asarray([-15,30])

```

Podemos graficar la función para observarla en un espacio tridimensional. Para esto, requerimos los valores en los que será evaluada nuestra función, valores que sabemos que están en el intervalo $[-15, 30]$

para las dos entradas de nuestra función.

Para esto, creamos dos arreglos: **ejex**, **ejey**, que contengan esos valores, utilizando la función *arange* de *numpy* que nos permite establecer el límite inferior, el superior y el incremento, en este caso, esos valores serían -15, 30, 1 respectivamente. Lo siguiente es crear los pares ordenados, **x**, **y**, empleando la función *meshgrid* de *numpy*. Esta función nos devuelve una matriz coordenada a partir de los vectores coordinados que recibe como parámetros.

Después, evaluamos la función en los valores y guardamos los resultados en **z**, y se grafica en 3D utilizando la función *plot_surface*, mandando como parámetros **x**, **y**, **z** que son nuestros tres ejes, además, mandamos el parámetro *cmap* = 'jet' para visualizar la superficie con colores fríos en los puntos «más bajos» y con colores más cálidos en los puntos «más altos», como se muestra en la figura 8.56.

Código:

```
# Valores en los que será evaluada la función
ejex = ejey = np.arange(lim[0], lim[1], 1)
# Formación de las parejas ordenadas
x, y = np.meshgrid(ejex, ejey)
# Datos de la función evaluada
z = funcion(x, y)
# Se procede a graficar
fig = plt.figure()
axis = fig.gca(projection='3d')
axis.plot_surface(x, y, z, cmap='jet')
plt.show()
```

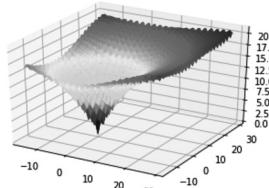


Figura 8.56. Función para representar el ejemplo de gradiente descendente

Lo siguiente que necesitamos es definir el gradiente, para el cual, ocupamos las derivadas parciales de nuestra función, que corresponden a:

$$\begin{aligned}\frac{\partial f(x_1, x_2)}{\partial x_1} &= \frac{1}{d\sqrt{\frac{1}{d}(x_1^2+x_2^2)}} \left(abx_1 e^{-b\sqrt{\frac{1}{d}(x_1^2+x_2^2)}} + c \sin(cx_1) e^{\frac{1}{d}(\cos(cx_1)+\cos(cx_2))} \sqrt{\frac{1}{d}(x_1^2+x_2^2)} \right) \\ \frac{\partial f(x_1, x_2)}{\partial x_2} &= \frac{1}{d\sqrt{\frac{1}{d}(x_1^2+x_2^2)}} \left(abx_2 e^{-b\sqrt{\frac{1}{d}(x_1^2+x_2^2)}} + c \sin(cx_2) e^{\frac{1}{d}(\cos(cx_1)+\cos(cx_2))} \sqrt{\frac{1}{d}(x_1^2+x_2^2)} \right)\end{aligned}\quad (8.10)$$

A diferencia de la función que nos devuelve un único valor, el gradiente es un vector, por lo que nos va a devolver un vector –un *array* de *numpy*–.

Código:

```
# Definición del gradiente g(t)
def gradiente(x1,x2):
    return np.asarray([
        (1/(d*np.sqrt((1/d)*(x1**2+x2**2)))) *
        ((a*b*x1*np.exp(-b*np.sqrt((1/d)*(x1**2+x2**2)))) +
        c*np.sin(c*x1)*np.exp((1/d)*(np.cos(c*x1)+np.cos(c*x2))) *
        np.sqrt((1/d)*(x1**2+x2**2))),
```

```

(1/(d*np.sqrt((1/d)*(x1**2+x2**2))))*
((a*b*x2*np.exp(-b*np.sqrt((1/d)*(x1**2+x2**2))))*
+c*np.sin(c*x2)*np.exp((1/d)*(np.cos(c*x1)+np.cos(c*x2)))*
np.sqrt((1/d)*(x1**2+x2**2)))]
)

```

También crearemos un gráfico en 2D de la función que nos resultará útil para observar cómo «avanza» el algoritmo en su búsqueda del mínimo global. Podemos utilizar la función `contourf` de `matplotlib` para este propósito. Esta función recibe como parámetros nuestros valores en **x**, **y**, **z**, `cmap = 'jet'` y `levels = 50`, que nos permite ajustar la «resolución» de los colores, como se muestra en el siguiente código y la figura 8.57.

Código:

```

plt.contourf(x, y, z, cmap='jet', levels=50)
plt.show()

```

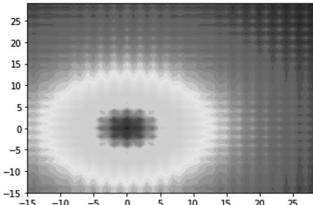


Figura 8.57. Función para representar el ejemplo de gradiente descendente mostrando los contornos de la función

El siguiente paso consiste en la construcción del algoritmo Adam.

Este algoritmo lo implementaremos como una función que recibe cuatro parámetros obligatorios, **lim**, **alfa**, **beta1**, **beta2**, y uno opcional, **epsilon**, este último tendrá un valor por defecto de 10^{-8} , un valor muy pequeño, solo para evitar que se produzca una división por cero.

Dentro de la función crearemos una lista llamada **resultados** donde almacenaremos los valores evaluados en cada iteración, a continuación, se crea el punto «aleatorio» **x** —que se encontrará dentro de los límites de valores permitidos **lim**— utilizando la función `randrange` de la librería `random`, la cual nos permite generar un número aleatorio dentro de un rango específico. **x** es un vector con dos elementos porque nuestra función $f(x_1, x_2)$ es de dos entradas y lo multiplicamos por 1.0 para que asegurarnos que sea un vector con elementos tipo `float`. Después, creamos una variable auxiliar **aux** que nos será de utilidad para medir qué tanto varia la función, dicho de otro modo, cuando la función pueda converger.

Lo siguiente es inicializar los momentos **m**, **v** y el tiempo **t** en cero, debido a que los momentos también son vectores, los creamos multiplicando el vector **lim** por 0.0, mientras que a **t** simplemente le asignamos el valor de cero.

Posteriormente, procedemos a realizar la actualización de parámetros de acuerdo con el algoritmo Adam. Esta actualización se llevará a cabo dentro de un ciclo `while`, nuestra condición de paro será cuando la función converja —cuando la diferencia entre $x(t)$ y $x(t-1)$, sea menor a 0.0001, $\|x(t) - x(t-1)\| < 0.0001$ — o cuando se hayan realizado cinco mil iteraciones. Dentro de este ciclo, obtendremos el gradiente **g** que corresponde a **x** y actualizaremos nuestros momentos **m**, **v**, **m2**, **v2** —**m2** y **v2** corresponden a \hat{m} y \hat{v} , respectivamente—, guardamos el valor actual de **x** en **aux** para realizar la comparación y calculamos el nuevo valor de **x**.

Finalmente, guardamos el valor de **x** en **resultados**, evaluamos nuestra función en **x**, imprimimos los resultados para ver cómo se está comportando el algoritmo, incrementamos **t** en una unidad y agregamos nuestra condición de paro de cinco mil iteraciones.

Código:

```
def adam(lim, alfa, beta1, beta2, epsilon=1e-8):
    # Creamos la lista donde guardaremos los resultados
    resultados = []
    # Generamos un punto inicial aleatorio dentro de los límites
    x = np.asarray([
        random.randrange(lim[0], lim[1]+1, 1),
        random.randrange(lim[0], lim[1]+1, 1)
    ]) * 1.0
    # Declaramos una variable auxiliar para medir la variación
    aux = [x[0] + 1, x[1] + 1]
    # Inicializamos los momentos «m» y «v» en cero
    m = v = lim * 0.0
    # Iniciamos el ciclo de optimización
    t = 0
    while (abs(x[0] - aux[0]) > 0.0001 or abs(x[1] - aux[1]) > 0.0001):
        # Obtenemos el gradiente
        g = gradiente(x[0], x[1])
        # Actualizamos una variable a la vez
        for i in range(lim.shape[0]):
            # m(t) = beta1 * m(t-1) + (1 - beta1) * g(t)
            m[i] = beta1 * m[i] + (1.0 - beta1) * g[i]
            # v(t) = beta2 * v(t-1) + (1 - beta2) * g(t)^2
            v[i] = beta2 * v[i] + (1.0 - beta2) * g[i]**2.0
            # m2(t) = m(t) / (1 - beta1(t))
            m2 = m[i] / (1.0 - beta1**(t+1.0))
            # v2(t) = v(t) / (1 - beta2(t))
            v2 = v[i] / (1.0 - beta2**(t+1.0))
            # Asignamos el valor actual de x a aux
            aux[i] = x[i]
            # Actualizamos x
            # x(t) = x(t-1) - alfa * m2(t) / (raiz(v2(t)) + epsilon)
            x[i] = x[i] - alfa * m2 / (np.sqrt(abs(v2)) + epsilon)
        # Guardamos el resultado
        resultados.append(x.copy())
        # evaluamos el punto calculado
        x3 = funcion(x[0], x[1])
        # Imprimimos cada punto para ver el progreso
        print("t={}, f(t) = f{} = {}".format(t, x, x3))
        # Incrementamos t
        t += 1
        # Restringimos el ciclo a 5000 iteraciones
        if t > 5000:
            break
    print("Gradiente: {}".format(g))
    return resultados
```

Una vez construido, vamos a correrlo, para ello, debemos definir los hiperparámetros que recibe nuestra función **adam**. Una selección de parámetros para algoritmos de *aprendizaje máquina* que se recomienda es la siguiente:

- $\alpha = 0.001$
- $\beta_1 = 0.9$
- $\beta_2 = 0.999$
- $\epsilon = 10^{-8}$

Estos hiperparámetros los podemos variar dependiendo de nuestro problema con el fin de obtener mejores resultados. Hay que tener en cuenta dos cosas: la primera, si α es muy pequeño, el algoritmo se puede estancar en algún mínimo local o donde el gradiente sea muy pequeño, o si es muy grande, puede que se pase del mínimo y nunca converja la función, y la segunda es que $\beta_1, \beta_2 \in [0,1]$.

Para este ejemplo, se encontró —después de algunas ejecuciones— que los parámetros $\alpha = 1$, $\beta_1 = 0.61$, $\beta_2 = 0.984$ dan buenos resultados.

La semilla que se establece `random.seed(1)` tiene el objetivo de iniciar siempre en el mismo punto con el propósito de poder realizar los ajustes correspondientes, de otro modo, si en cada ejecución se inicia en un punto distinto, sería mucho más difícil realizar ajustes en el algoritmo.

Ya con los parámetros establecidos, ejecutemos el algoritmo y guardemos los resultados en **resultados** para graficar, creando un gráfico de contorno, como lo hicimos previamente y agregándole la «trayectoria» que sigue el algoritmo.

Código:

```
# Establecemos la semilla
random.seed(1)

# Establecemos nuestros hiperparámetros
# Magnitud del paso
alfa = 1

# Factores de decremento para la estimación del momento
# Factor del primer momento
beta1 = 0.61

# Factor del segundo momento
beta2 = 0.984

resultados = adam(lim, alfa, beta1, beta2)

# Creamos un gráfico de contorno
plt.contourf(x, y, z, cmap='jet', levels=50)
# Graficamos los resultados obtenidos
resultados = np.asarray(resultados)
plt.plot(resultados[:, 0], resultados[:, 1], '--k')
plt.show()
```

```

t=2767, f(t) = f(-0.57955102 -0.43821561) = 2.6757775538302226
t=2768, f(t) = f(-0.16788182 0.22595908) = 0.8251566833648987
t=2769, f(t) = f( 0.55239103 -0.85038212) = 3.4699063399530696
t=2770, f(t) = f(-0.05634756 -1.18465628) = 3.2917299629213
t=2771, f(t) = f(-0.37275328 -0.75427519) = 2.9689817450339784
t=2772, f(t) = f(0.39030174 0.18926641) = 1.4705324132079372
t=2773, f(t) = f(-0.76562054 -0.39262396) = 3.0299212985180306
t=2774, f(t) = f(-0.98501508 0.27022016) = 3.2483867404142615
t=2775, f(t) = f(-0.26823611 -0.23630592) = 1.1502989531699468
t=2776, f(t) = f(0.91158229 0.67643959) = 3.7545160994024163
t=2777, f(t) = f(1.44903365 0.37012483) = 3.6808082785699385
t=2778, f(t) = f( 1.15959715 -0.50340745) = 3.752170902671722
t=2779, f(t) = f(2.22283976 0.18659943) = 3.208819952097735
t=2780, f(t) = f( 1.19735822 -0.56265768) = 3.98104490892062
t=2781, f(t) = f( 0.22917359 -0.13407458) = 0.7587800696618441
t=2782, f(t) = f(-1.20643281 0.78028069) = 4.197490724454227
t=2783, f(t) = f(-2.18341532 0.94175981) = 4.806466335081821
t=2784, f(t) = f(-2.36927344 0.34126005) = 4.0113488147066505
t=2785, f(t) = f(-1.34946627 -0.78896536) = 4.297110138272946

```

Figura 8.58. Extracto de los resultados por iteración del algoritmo de gradiente descendiente

En la figura 8.58, podemos apreciar que, a partir de la iteración 2700, el algoritmo comienza a converger, pero no está claro hacia qué valor exactamente, además, el gradiente no está cerca de cero aún. Esto puede ser debido a que nuestro alfa es muy grande, por lo que podemos agregar una condición para hacer a alfa más pequeño a partir de la iteración 2700.

Agregando la condición de que después de la iteración 2700 alfa sea igual a 0.001, nuestra función se vería de la siguiente forma:

Código:

```

def adam(lim, alfa, beta1, beta2, epsilon=1e-8):
    # Creamos la lista donde guardaremos los resultados
    resultados = []
    # Generamos un punto inicial aleatorio dentro de los límites
    x = np.asarray([
        random.randrange(lim[0], lim[1]+1, 1),
        random.randrange(lim[0], lim[1]+1, 1)
    ]) * 1.0
    # Declaramos una variable auxiliar para medir la variación
    aux = [x[0] + 1, x[1] + 1]
    # Inicializamos los momentos «m» y «v» en cero
    m = v = lim * 0.0
    # Iniciamos el ciclo de optimización
    t = 0
    while (abs(x[0] - aux[0]) > 0.0001 or abs(x[1] - aux[1]) > 0.0001):
        # Obtenemos el gradiente
        g = gradiente(x[0], x[1])
        # Actualizamos una variable a la vez
        for i in range(lim.shape[0]):
            # m(t) = beta1 * m(t-1) + (1 - beta1) * g(t)
            m[i] = beta1 * m[i] + (1.0 - beta1) * g[i]
            # v(t) = beta2 * v(t-1) + (1 - beta2) * g(t)^2
            v[i] = beta2 * v[i] + (1.0 - beta2) * g[i]**2.0
            # m2(t) = m(t) / (1 - beta1(t))
            m2 = m[i] / (1.0 - beta1**(t+1.0))
            # v2(t) = v(t) / (1 - beta2(t))
            v2 = v[i] / (1.0 - beta2**(t+1.0))
            # Asignamos el valor actual de x a aux
            aux[i] = x[i]
        # Actualizamos x
        # x(t) = x(t-1) - alfa * m2(t) / (raiz(v2(t)) + epsilon)

```

```

x[i] = x[i] - alfa * m2 / (np.sqrt(abs(v2)) + epsilon)
# Guardamos el resultado
resultados.append(x.copy())
# Evaluamos el punto calculado
x3 = funcion(x[0], x[1])
# Imprimimos cada punto para ver el progreso
print("t={}, f(t) = f{} = {}".format(t, x, x3))
# Incrementamos t
t += 1
# Restringimos el ciclo a 1000 iteraciones
if t > 5000:
    break
if t == 2700:
    alfa = 0.001
print("Gradiente: {}".format(g))
return resultados

```

Corramos de nuevo el programa para ver nuestros resultados.

Código:

```

# Establecemos la semilla
random.seed(1)

# Establecemos nuestros hiperparámetros
# Magnitud del paso
alfa = 1
# Factores de decremento para la estimación del momento
# Factor del primer momento
beta1 = 0.61
# Factor del segundo momento
beta2 = 0.984

resultados = adam(lim, alfa, beta1, beta2)

# Creamos un gráfico de contorno
plt.contourf(x, y, z, cmap='jet', levels=50)
# Graficamos los resultados obtenidos
resultados = np.asarray(resultados)
plt.plot(resultados[:, 0], resultados[:, 1], '--k')
plt.show()

```

```

t=3125, f(t) = f[-2.02569809 -2.02492701] = 3.7243333059541763
t=3126, f(t) = f[-2.02569275 -2.02503131] = 3.7243324049369146
t=3127, f(t) = f[-2.02569712 -2.02512936] = 3.724331672908199
Gradiente: [1.59145907e-05 8.03316664e-03]

```

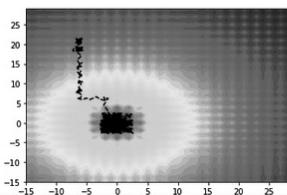


Figura 8.59. Ejecución del algoritmo de gradiente descendente mostrando su convergencia

En esta ejecución, se ve claramente que la función converge al punto $(-2.02, -2.02, 3.72)$ y que el gradiente también es mucho menor.

Veamos la gráfica en 3D para tener una mejor apreciación. Graficaremos como lo hicimos al principio, activando la proyección 3D, y le agregaremos los puntos de la forma (x, y, z) para visualizar la trayectoria, los valores de x, y son x_1, x_2 (**resultados**`[:, 0]` y **resultados**`[:, 1]`, respectivamente) que obtuvimos en la ejecución del algoritmo Adam, nos faltarían los valores de z , que podemos obtener fácilmente llamando a la función «**funcion**» y evaluarla en los valores de x, y . Obtenidos estos valores, graficamos una línea que vaya por estos puntos.

Código:

```
# Graficamos en 3D
fig = plt.figure()
axis = fig.gca(projection='3d')
axis.plot_surface(x, y, z, cmap='jet')
z = funcion(resultados[:, 0], resultados[:, 1])
axis.plot(resultados[:, 0], resultados[:, 1], z, '-k', linewidth=4)
plt.show()
```

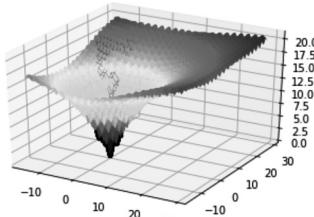


Figura 8.60. Función graficada en 3D mostrando los resultados de gradiente descendente

Si quitamos la superficie se puede apreciar de mejor manera la convergencia, como se muestra en la figura 8.61.

Código:

```
# Graficamos en 3D
fig = plt.figure()
axis = fig.gca(projection='3d')
z = funcion(resultados[:, 0], resultados[:, 1])
axis.plot(resultados[:, 0], resultados[:, 1], z, '-k', linewidth=4)
plt.show()
```

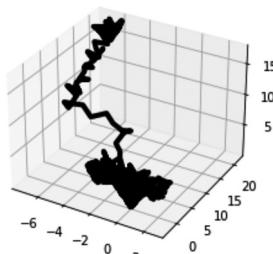


Figura 8.61. Resultados finales del algoritmo gradiente descendente

Como ejercicio complementario, modifica tu programa para poder llegar al mínimo global para otras funciones. Por ejemplo, encuentra el mínimo global para la función de Himmelblau (figura 8.62) que se describe mediante la ecuación 8.11.

$$f(x,y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2 \quad (8.11)$$

Recuerda que la función Himmelblau tiene un mínimo global en $x = -0.270844$ e $y = -0.923038$ donde: $f(x,y) = 181.616$.

Esta función también tiene cuatro mínimos locales, los cuales son:

```
f(3.0, 2.0) = 0.0
f(-2.805118, 3.131312) = 0.0
f(-3.77931, -3.283186) = 0.0
f(3.584428, -1.848126) = 0.0
```

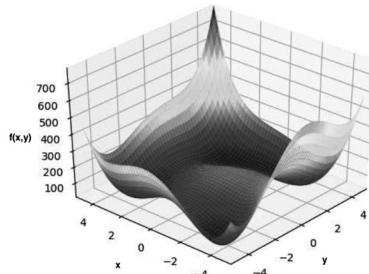


Figura 8.62. Función Himmelblau para el ejercicio de descendiente al gradiente

8.13. Método de validación cruzada (K-fold)

Como se comentó en la sección 5.12, la validación cruzada (*K-fold*) es uno de los métodos más utilizados para realizar pruebas a los algoritmos de inteligencia artificial.

Dividir un conjunto de datos conjuntos de entrenamiento y prueba es una buena técnica para cuando queremos evaluar un algoritmo de *aprendizaje máquina*. Sin embargo, dividir los datos en solo dos conjuntos y realizar una sola prueba puede arrojar resultados muy sesgados.

K-fold es una alternativa para evitar este problema, ya que nos permite dividir los datos en *k* particiones homogéneas, donde se utilizará cada una de estas particiones como conjunto de prueba mientras las demás se utilizan como entrenamiento, en un proceso repetitivo de *k* iteraciones.

En la figura 8.63 se ilustra cómo se dividen los datos cuando $k = 5$. El color verde se refiere al conjunto de entrenamiento y el color amarillo al conjunto de prueba.

Iteración	DATOS				
	C1	C2	C3	C4	C5
1	C1	C2	C3	C4	C5
2	C1	C2	C3	C4	C5
3	C1	C2	C3	C4	C5
4	C1	C2	C3	C4	C5
5	C1	C2	C3	C4	C5

Figura 8.63. División de datos para prueba de validación cruzada cuando $k = 5$

El código se muestra en esta sección y puede ser descargado en:
http://www.amese.net/libro_ia/Prog/8.

Para realizar esta práctica, se utilizará la base de datos mencionada en el apéndice A.2. «Enfermedad coronaria».

Comenzamos importando las librerías y creando las particiones —en este caso, cinco—.

Código:

```
import pandas as pd
import numpy as np
```

```
df = pd.read_csv("HeartDisease.csv")
print(df.head())
```

Las particiones las realizaremos a través de una función que llamaremos *particion* –así, sin acento–, en la cual haremos las particiones de la manera más homogénea que sea posible. Esta función recibirá como parámetros la base de datos *df* y el número de particiones a realizar *k*.

Posteriormente, creamos una lista donde guardaremos las divisiones llamada **datos**, obtenemos el número de elementos en la base de datos y lo guardamos en **tamano**. Este valor tiene que ser entero, por lo tanto, al momento de guardar el valor, lo convertimos a entero –a esto se le conoce como *casting*–, porque la división entre el número de datos y las particiones a crear, usualmente, será un número tipo *float*.

El siguiente paso es hacer un ciclo *for* donde iremos asignando los índices de cada partición, el rango de cada partición lo determinamos utilizando la función **range**, donde le mandaremos los parámetros, *i*tamano* (límite inferior) y *(i+1)*tamano* (límite superior). Este rango lo vamos a guardar como un arreglo y lo agregamos a la lista **datos**, este ciclo lo ejecutaremos *k-1* veces, ya que, para la última partición, los límites serán *(i+1)*tamano* (límite inferior) y *len(df)* (límite superior), de modo que, si se realizó un *casting* cuando obtuvimos **tamano**, no se pierdan elementos. Por ejemplo, si tenemos veinticiete datos en total y queremos cinco particiones, el tamaño de la partición sería de $27/5 = 5.4$, al realizar el *casting*, el tamaño nos quedaría de cinco, pero, si hacemos todas las particiones de cinco elementos, al final tendríamos veinticinco datos en lugar de veintisiete, para evitar esto, lo que hacemos es que las primeras cuatro particiones las hacemos de cinco elementos y la última partición la hacemos de los elementos restantes, siete en este caso: y así, al final, tenemos los veintisiete datos, aunque nuestra última partición es ligeramente más grande.

Código:

```
def particion(df, k):
    # Lista donde guardaremos las particiones
    datos = []
    # Obtenemos el total de datos en la BD
    tamano = int(len(df)/k)
    # Comenzamos a realizar las particiones
    for i in range(k-1):
        # Índices de cada partición
        division = np.array(range(i*tamano, (i+1)*tamano))
        # Guardamos la partición en datos
        datos.append(np.array(division))
    # Última división
    datos.append(np.array(range((i+1)*tamano, len(df))))
    return datos
```

Ya con las particiones creadas, haremos otra función llamada *conjunto*, que nos servirá para obtener los conjuntos de entrenamiento y prueba. Esta función recibe tres parámetros, la base de datos *df*, la lista de particiones **datos** y el índice llamado **indice** que indica el número de iteración y, por ende, la posición del conjunto de prueba.

Para crear un conjunto de datos a partir de un **data frame**, usamos la función *iloc* de **pandas** que filtra los datos de acuerdo con los índices que recibe como parámetro. El conjunto de prueba lo podemos crear mandando *datos[indice]*, como parámetro a *iloc*, mientras que el conjunto de entrenamiento consiste en todos los datos, excepto los que usamos como prueba. De esta manera, para obtener estos datos, podemos obtener los índices de todos los datos en *df* usando *np.array(range(len(df))*) y eliminando los índices del

conjunto de datos con la función `delete` de `numpy`, esta función recibe como parámetros el `array` sobre el que se va a trabajar y las posiciones que se van a eliminar. Posteriormente, usando `np.delete(ind, datos[indice])`, le indicamos que trabaje sobre los índices de todo el archivo y después elimine los que usamos como prueba. Hecho esto, usamos de nuevo `iloc` para obtener los datos correspondientes al conjunto de entrenamiento, como se muestra en el siguiente código:

Código:

```
def conjunto(df, datos, indice):  
    # Creamos el conjunto de prueba  
    prueba = df.iloc[datos[indice]]  
  
    # Sacamos los indices de toda la BD  
    ind = np.array(range(len(df)))  
  
    # Eliminamos los indices del conjunto de prueba  
    ind_ent = np.delete(ind, datos[indice])  
  
    # Creamos el conjunto de entrenamiento  
    entrenamiento = df.iloc[ind_ent]  
  
    return entrenamiento, prueba
```

Probemos nuestro algoritmo.

Código:

```
# Número de particiones que queremos  
k = 4  
  
# Mandamos llamar a division() para crear las particiones  
datos = particion(df, k)  
  
datos
```

```

[array] [ 8, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114], [array] [ 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, 165, 166, 167, 168, 169, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180, 181, 182, 183, 184, 185, 186, 187, 188, 189, 190, 191, 192, 193, 194, 195, 196, 197, 198, 199, 200, 201, 202, 203, 204, 205, 206, 207, 208, 209, 210, 211, 212, 213, 214, 215, 216, 217, 218, 219, 220, 221, 222, 223, 224, 225, 226, 227, 228, 229], [array] [ 230, 231, 232, 233, 234, 235, 236, 237, 238, 239, 240, 241, 242, 243, 244, 245, 246, 247, 248, 249, 250, 251, 252, 253, 254, 255, 256, 257, 258, 259, 260, 261, 262, 263, 264, 265, 266, 267, 268, 269, 270, 271, 272, 273, 274, 275, 276, 277, 278, 279, 280, 281, 282, 283, 284, 285, 286, 287, 288, 289, 290, 291, 292, 293, 294, 295, 296, 297, 298, 299, 300, 301, 302, 303, 304, 305, 306, 307, 308, 309, 310, 311, 312, 313, 314, 315, 316, 317, 318, 319, 320, 321, 322, 323, 324, 325, 326, 327, 328, 329, 330, 331, 332, 333, 334, 335, 336, 337, 338, 339, 340, 341, 342, 343, 344, 345, 346, 347, 348, 349, 350, 351, 352, 353, 354, 355, 356, 357, 358, 359, 360, 361, 362, 363, 364, 365, 366, 367, 368, 369, 370, 371, 372, 373, 374, 375, 376, 377, 378, 379, 380, 381, 382, 383, 384, 385, 386, 387, 388, 389, 390, 391, 392, 393, 394, 395, 396, 397, 398, 399, 400, 401, 402, 403, 404, 405, 406, 407, 408, 409, 410, 411, 412, 413, 414, 415, 416, 417, 418, 419, 420, 421, 422, 423, 424, 425, 426, 427, 428, 429, 430, 431, 432, 433, 434, 435, 436, 437, 438, 439, 440, 441, 442, 443, 444, 445, 446, 447, 448, 449, 450, 451, 452, 453, 454, 455, 456, 457, 458, 459, 460, 461]

```

Figura 8.64. Resultado de la división de k-fold para k = 4

En la figura 8.64, se muestra un extracto del resultado de una división de validación cruzada con $k = 4$, si se requiere un número diferente de particiones, se cambia el valor de la variable k . El siguiente código creará las listas para guardar los datos de entrenamiento y prueba para cada k , cuyo resultado se muestra en la figura 8.65.

Código:

```
# Creamos dos listas donde guardaremos nuestros datos
train = []
test = []
# Obtenemos los conjuntos llamando a conjunto()
for i in range(k):
```

```

entrenamiento, prueba = conjunto(df, datos, i)
print("Iteración:", i + 1)
print("Entrenamiento:")
print(entrenamiento)
print("Prueba:")
print(prueba)

# Agregamos los conjuntos a las listas
train.append(entrenamiento)
test.append(prueba)

```

```

Iteración: 1
Entrenamiento:
   sbp famhist obesity age chd
115 128      1    28.41  48     0
116 140      0    21.91  32     1
117 154      0    20.60  42     0
118 150      1    23.35  61     1
119 130      0    28.82  27     0
...
457 114      0    28.45  58     0
458 182      0    28.61  52     1
459 108      0    20.89  55     0
460 118      0    27.35  40     0
461 132      1    14.70  46     1
[347 rows x 5 columns]
Prueba:
   sbp famhist obesity age chd
0   160      1    25.30  52     1

```

Figura 8.65. Extracto de la creación de listas para las particiones con $k = 4$ para validación cruzada

Por último, preguntamos si queremos exportar los datos a un archivo. Si la respuesta es afirmativa, guardaremos cada subconjunto de entrenamiento con su respectivo subconjunto de prueba, enumerándolos desde 0 hasta $k-1$.

Código:

```

guardar = input("¿Desea guardar los datos en un archivo? s/n\n")
if guardar.lower() == "s":
    for i in range(k):
        n_train = "train_" + str(i) + ".csv"
        train[i].to_csv(n_train)
        n_test = "test_" + str(i) + ".csv"
        test[i].to_csv(n_test)

```

8.14. Algoritmo K-medias (K-means)

En este apartado, revisaremos uno de los algoritmos de aprendizaje no supervisado más utilizados: *k-means*.

Este algoritmo nos permite la agrupación de un conjunto de entrada en diversos subgrupos que cuentan con características similares. Esto tiene una gran utilidad en proyectos donde no se tiene información etiquetada y se quiere realizar una clasificación basada en características de cada instancia del conjunto de datos.

El código se muestra en esta sección y puede ser descargado en:
http://www.amese.net/libro_ia/Prog/8.

Comencemos a revisar la implementación del algoritmo, inicialmente, importamos algunas librerías de utilidad, como se muestra en el siguiente código:

Código:

```

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

```

El algoritmo de k-medias se basa en la medición del centroide de cada subgrupo por medio de la media de los integrantes de dicho grupo, asignando cada instancia al grupo más cercano.

Revisemos el algoritmo por pasos:

- Definir la cantidad de clases.
- Inicializar centroides de manera aleatoria.
- Medición de la distancia euclídea de cada punto a cada centroide.
- Asignación de cada punto a cada clase según la distancia mínima calculada.
- Actualización de centroides con la media de los puntos de cada grupo.
- Repetir 3-5 hasta alcanzar algún criterio de paro.

El criterio de paro no está establecido, comúnmente, se realiza por:

- Número de iteraciones.
- Ausencia de cambios en los grupos.
- Poca cantidad de cambios en la asignación de cada punto.
- Otros.

Para esta práctica en específico, seguiremos la metodología propuesta en la sección 5.6.

1. Recolectar

En este ejercicio, usaremos una base de datos muy popular dentro del área de clasificación, Iris, la cual se explica más a detalle en el apéndice A.1. Realizaremos la clasificación de tres distintos tipos de flores según sus características y sin conocimiento previo de las clases.

A continuación, crearemos nuestra clase **myKmeans**, la cual será un clasificador no supervisado. Dentro de los métodos que tendrá esta clase están:

- **__init__**: se inicializan las cinco variables que nos servirán para realizar este algoritmo.
- **set_random_centroids**: inicializar los centroides de manera aleatoria al inicio del algoritmo.
- **set_new_centroids**: actualizar los centroides en cada iteración.
- **normalizeX**: normalizamos nuestros datos para que no existan sesgos por las magnitudes de los datos.
- **get_distance**: obtener la distancia de cada punto a cada centroide.
- **assign_clusters**: se basa en las distancias para asignar cada punto al grupo que corresponde según la distancia mínima.
- **fit**: aquí se unen la mayor parte de los métodos para seguir el algoritmo.
- **predict**: regresa la clasificación de cierto punto o grupo de puntos basado en el entrenamiento del algoritmo.
- **plot_iter**: nos permite observar de manera gráfica el comportamiento de nuestro algoritmo.

Código:

```
class myKmeans:  
    """  
    Este es nuestro propio clasificador no supervisado kmeans.  
    """  
  
    import numpy as np  
    import random  
    import matplotlib.pyplot as plt
```

```

def __init__(self):
    self.k=0
    self.dimensions=0
    self.centroids=np.array([])
    self.clusters=np.array([], dtype=int)
    self.colors=['red','blue','green','orange','cyan','black', 'magenta']

def set_k(self, k):
    self.k=k

def get_k(self):
    return self.k

def set_dimensions(self, X):
    self.dimensions=X.shape[1]

def get_dimensions(self):
    return self.dimensions

def set_random_centroids(self):
    c=np.zeros([self.k, self.dimensions])
    for rows in range(self.k):
        for cols in range(self.dimensions):
            c[rows, cols]= self.random.random()
    self.centroids=c

def set_new_centroids(self, X):
    c=np.zeros([self.k, self.dimensions])
    for rows in range(self.k):
        for cols in range(self.dimensions):
            c[rows, cols]= self.np.mean(X[self.clusters==rows,cols])
    self.centroids=c

def get_centroids(self):
    return self.centroids

def normalize(self, feature):
    return (feature-min(feature))/(max(feature)-min(feature))

def normalizeX(self, X):
    X=self.np.array(X)
    for index in range(self.dimensions):
        X[:,index]=self.normalize(X[:,index])
    return X

def get_distance(self, X):
    dist=np.zeros([X.shape[0], self.centroids.shape[0]])
    for c in range(self.centroids.shape[0]):
        for p in range(X.shape[0]):
            dist[p,c]=self.euclidean_distance(X[p,:],self.centroids[c,:])
    return dist

```

```

def euclidean_distance(self, point, centroid):
    euclideanDistance=0
    for dim in range(self.dimensions):
        euclideanDistance+=(point[dim]-centroid[dim])**2
    return euclideanDistance**(1/2)

def assign_clusters(self, distances):
    prev_clusters=self.clusters.copy()
    self.clusters=np.zeros(distances.shape[0], dtype=int)
    for index in range(distances.shape[0]):
        self.clusters[index]=np.where(min(distances[index,:])==distances[index,:])[0][0]
    return self.np.array_equal(self.clusters, prev_clusters)

def get_clusters(self):
    return self.clusters

def fit(self, X, k=2, iterations=20, verbose=False):
    self.set_k(k)
    self.set_dimensions(np.array(X))
    self.set_random_centroids()
    X=self.normalizeX(X)
    for iteration in range(iterations):
        print('Iteración', iteration)
        if iteration!=0:
            self.set_new_centroids(X)
        distances=self.get_distance(X)
        same=self.assign_clusters(distances)
        if verbose:
            self.plot_iter(X)
        if same==True:
            print('Los centroides no han cambiado')
            return None
        print('Máximo de iteraciones alcanzado')
        return None

    def predict_clusters(self, distances):
        predicted_clusters=np.zeros(distances.shape[0], dtype=int)
        for index in range(distances.shape[0]):
            predicted_clusters[index]=np.where(min(distances[index,:])==distances[index,:])[0][0]
        return predicted_clusters

    def predict(self, X):
        distances=self.get_distance(self.normalizeX(X))
        return self.predict_clusters(distances)

    def set_colors(self):
        while self.get_k()>len(self.get_colors()):
            allColors=[k for k, i in mcolors.cnames.items()]
            selected_color=allColors[random.randint(0,len(allColors))]
            if selected_color not in np.array(self.get_colors()):

```

```

    self.colors.append(selected_color)

def get_colors(self):
    return self.colors

def plot_iter(self, X):
    self.set_colors()
    self.plt.figure()
    for index in range(self.k):
        self.plt.scatter(X[index==self.clusters,0],X[index==self.clusters,1], color=self.colors[index])
        self.plt.scatter(self.centroids[0], self.centroids[1], marker='s', color='yellow')
    self.plt.pause(1)

```

Ya que tenemos nuestro clasificador, procederemos a crear una función que nos permita observar, de forma gráfica, la manera en que están organizados los grupos separando las clases por colores, como se muestra en el siguiente código:

Código:

```

import matplotlib.colors as mcolors
import random

def plot_classes(X, y):
    plt.figure()
    classes=np.unique(y)
    colors=['red','blue','green','orange','cyan']
    # Si necesitas más colores puedes agregarlos.
    #allColors=[k for k, i in mcolors.cnames.items()]
    # Si ejecutas esta linea de código puedes
    # obtener todos los colores disponibles.
    for index in np.unique(y):
        plt.scatter(X[index==y,0], X[index==y,1], color=colors[np.where(index==np.unique(y))[0][0]])

```

2. Preparar

Antes de leer nuestros datos para iniciar con la clasificación, crearemos una función que nos entregue cuatro subconjuntos que llamaremos **X_train**, **y_train**, **X_test** y **y_test**. Para entrenar y probar nuestro clasificador.

Nota. El subconjunto **y_train** no se utilizará debido a que es un algoritmo no supervisado, por lo cual, no requiere de los valores etiquetados para clasificar el subconjunto **X_train**. Sin embargo, **y-test** se usará con propósitos de evaluación del desempeño del algoritmo.

Código:

```

def split_train_test(X, y, division=0.8):
    index=np.random.permutation(np.arange(0,X.shape[0]))
    split=int(division*X.shape[0])
    X_train=X[index[:split],:]
    X_test=X[index[split:]]
    y_train=y[index[:split]]
    y_test=y[index[split:]]
    return X_train, X_test, y_train, y_test

```

3. Analizar

A continuación, leemos el archivo y guardamos nuestro valor real de clasificación en la variable **y**. De igual manera, guardamos nuestros datos en la variable **X**.

Dado que no podemos observar gráficamente los puntos en dimensiones muy grandes, en este ejercicio lo simplificaremos a dos dimensiones para observarlo de una mejor manera.

Nota. Puede realizarse el cálculo de clases a dimensiones más grandes, pero la visualización siempre será en 2D.

Desplegamos una vista previa de los datos. Nota: hay que revisar si el archivo iris.csv está en el directorio correcto. Esto se muestra en la figura 8.66.

Código:

```
df=pd.read_csv('iris.csv', header=None)
y=np.array(df[4])
X=np.array(df.drop([2,3,4], axis=1))
df.head()
```

	0	1	2	3	4
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

Figura 8.66. Extracto de la apertura de archivo iris.csv para K-medias

Revisamos si existen valores nulos en alguna columna con el siguiente código. Esto se muestra en la figura 8.67.

Código:

```
for c in df.columns:
    print('Column',c,'- Nulls:',np.sum(pd.isnull(df[c])))
```

```
Column 0 - Nulls: 0
Column 1 - Nulls: 0
Column 2 - Nulls: 0
Column 3 - Nulls: 0
Column 4 - Nulls: 0
```

Figura 8.67. Listado de valores nulos por atributo (columnas)

Esto puede ser muy útil para identificar si se requiere realizar imputación o se tienen demasiados valores faltantes por atributo.

Ahora, utilizamos el método **describe** para tener un resumen de los datos mediante variables estadísticas, como se muestra en el siguiente código:

Código:

```
df.describe()
```

	0	1	2	3
count	150.000000	150.000000	150.000000	150.000000
mean	5.843333	3.054000	3.758667	1.198667
std	0.829066	0.433594	1.764420	0.763161
min	4.300000	2.000000	1.000000	0.100000
25%	5.100000	2.800000	1.600000	0.300000
50%	5.800000	3.000000	4.350000	1.300000
75%	6.400000	3.300000	5.100000	1.800000
max	7.900000	4.400000	6.900000	2.500000

Figura 8.68. Estadísticas de la base de datos por atributo (columnas)

Procedemos por dividir **X** e **Y** en nuestros subconjuntos de entrenamiento y prueba. En este ejemplo en particular, se realizó una partición 80-20, como se muestra en la sección 5.10, y no una validación cruzada como en la sección 5.12 y la práctica anterior (8.13).

Código:

```

x_train, X_test, y_train, y_test = split_train_test(X, y, division=0.8)
print("Dimensión de X_train:", X_train.shape)
print("Dimensión de X_test:", X_test.shape)
print("Dimensión de y_train:", y_train.shape)
print("Dimensión de y_test:", y_test.shape)

Dimension de X_train: (120, 2)
Dimension de X_test: (30, 2)
Dimension de y_train: (120,)
Dimension de y_test: (30,)

```

Figura 8.69. Separación de datos para K-medias por el método 80-20

Es importante mencionar que en nuestra función **split_train_test** hicimos la división en un 80-20.

80 % de los datos son para el entrenamiento –ciento veinte datos– y 20 % son para prueba –treinta datos–.

Nota. Se utilizó una permutación aleatoria de los índices para que siempre se obtengan distintos subconjuntos de entrenamiento y prueba. Esto quiere decir que, a partir de este punto, tus resultados pueden variar un poco de los que aquí se muestran.

4. Entrenar

Ahora creamos y entrenamos el clasificador.

Como argumentos colocamos el conjunto de datos **X**, un número de grupos **k** igual a 3, 20 iteraciones como máximo y ponemos el valor *True* en **verbose** para permitir que nuestra función grafique los resultados de cada iteración.

Código:

```

clf=myKmeans()
clf.fit(X_train, k=3, iterations=20, verbose=True)

```

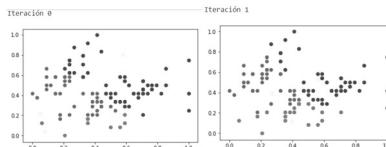


Figura 8.70. Ejecución del algoritmo de K-medias, mostrando las iteraciones 0 y 1

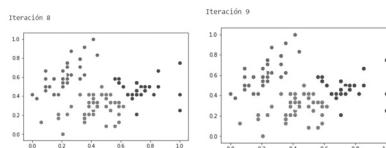


Figura 8.71. Ejecución del algoritmo de K-medias mostrando las iteraciones 8 y 9

Se puede observar en las figuras 8.70 y 8.71 que los centroides, en color amarillo, se van moviendo en cada iteración junto con el resto de los grupos. Al final, se muestra por cuál de los dos criterios de paro se detuvo el algoritmo –este proceso se puede encontrar en el método **fit**–.

5. Probar

Después, realizamos la predicción de nuestro conjunto de prueba para evaluar la capacidad de clasificación de nuestro algoritmo.

Código:

```

predictions=clf.predict(X_test)

```

Para tener una visualización del conjunto real y el conjunto evaluado utilizamos la función **plot_classes** dos veces, la primera, con nuestro valor conocido en **y**, la segunda, con nuestros valores predichos por el clasificador.

Código:

```
plot_classes(X_test, y_test)
```

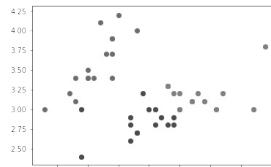


Figura 8.72. Ejecución del algoritmo de K-medias mostrando el entrenamiento

Código:

```
plot_classes(X_test, predictions)
```

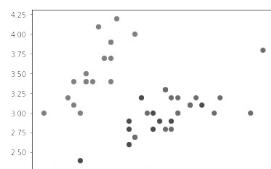


Figura 8.73. Ejecución del algoritmo de K-medias mostrando las pruebas

De forma gráfica, puede verse que los grupos se asemejan, pero, para observar de forma numérica los resultados, creamos una función que nos entregue la precisión de la clasificación. Llamaremos **get_accuracy** a dicha función.

Nota. Uno de los inconvenientes de los métodos de aprendizaje no supervisado es que etiquetan los datos de manera aleatoria, así que, si ejecutas el algoritmo varias veces, es probable que cada clase sea nombrada de manera distinta.

Para resolver el problema anterior, dentro de la función **get_accuracy** haremos que se determine cuál es la clasificación que mejor se acopla a nuestro valor conocido **y-test** al realizar permutaciones en las etiquetas que se asignan a cada clase y determinando cuál nos entrega un mejor valor de precisión. Esto no se realizaría en un problema en el que no tenemos datos etiquetados, ya que es indistinto el nombre de las clases si no se tiene un etiquetado previo.

Código:

```
from itertools import permutations

def get_accuracy(y, predictions):
    allPermutations=np.array(list(permutations(np.unique(y) )))
    acc=[]
    for perm in allPermutations:
        classes=np.arange(0,y.shape[0])
        for index in range(classes.shape[0]):
            classes[index]=np.where(y[index]==perm)[0][0]
        acc.append(np.sum(classes==predictions))
    acc=np.array(acc)
    bestAccIndex=np.where(max(acc)==acc)[0][0]
    return dict(zip(np.arange(0,allPermutations.shape[1]),allPermutations[bestAccIndex])), acc[bestAccIndex]
```

Guardamos las clases y precisión que entrega la función **get_accuracy**. Posteriormente, imprimimos en pantalla el orden de los grupos y la precisión obtenida.

Código:

```
classes_dict, acc = get_accuracy(y_test, predictions)
print('Orden de los grupos:', classes_dict)
print('Aciertos:', acc)
```

```
Orden de los grupos: {0: 'Iris-virginica', 1: 'Iris-versicolor', 2: 'Iris-setosa'}
Aciertos: 37
```

Es importante mencionar que no siempre dará exactamente los mismos aciertos, puesto que depende de los valores de los centroides iniciales para realizar el entrenamiento y los resultados de las pruebas dependerán de ello.

Por último, creamos nuestra variable **predicted_classes**, la cual, es una traducción de los valores numéricos de **predictions** al nombre real de nuestras clases.

Desplegamos los resultados y el porcentaje de precisión.

Adicionalmente, creamos un dataframe de la librería **pandas** para visualizar nuestros datos de una mejor manera. Colocamos el valor esperado **y_test** en la columna «Resultado esperado», el valor predicho **predicted_classes** en la columna «Resultado obtenido» y si la predicción fue acertada en la columna «Correcto».

Código:

```
predicted_classes=[classes_dict[_] for _ in predictions]
results=pd.DataFrame(data=np.transpose([y_test, predicted_classes, y_test==predicted_classes]),
columns=['Resultado esperado','Resultado obtenido','Correcto'])
print('Aciertos:',acc,'/',y_test.shape[0],'- Porcentaje:',100*acc/y_test.shape[0],'%')
results.head()
```

Aciertos: 37 / 45 - Porcentaje: 82.222222222223 %		
	Resultado esperado	Resultado obtenido
0	Iris-versicolor	Iris-versicolor
1	Iris-setosa	Iris-setosa
2	Iris-virginica	Iris-virginica
3	Iris-versicolor	Iris-versicolor
4	Iris-virginica	Iris-virginica

Figura 8.74. Resultados del algoritmo de K-medias, el porcentaje de predicción correcta

Como podemos ver que en la figura 8.74 se tiene un valor aceptable de exactitud, y los primeros valores de predicción son idénticos a los que esperábamos. En caso de que un valor no corresponda, en la columna «Correcto» aparecería el valor «False».

Nota. Es probable que, cuando implementes el algoritmo por tu cuenta, encuentres valores similares a los que aquí se muestran, aunque podrían existir variaciones debido a que creamos grupos de entrenamiento y prueba aleatorios al inicio del algoritmo.

6. Usar

Una vez que revisamos el proceso completo para el uso de nuestra librería k-means, podemos utilizarla en cualquier conjunto de datos deseado.

Para usar otra base de datos, elige una dentro de tu carpeta **database** e impórtalo por medio del comando **read_csv** de la librería **pandas** como lo hemos hecho hasta ahora.

A partir de ahí, sigue los mismos pasos desde el inicio de este capítulo, ya que realizamos el algoritmo de manera general para que pueda ser usado con cualquier conjunto de datos.

Como consideraciones generales, utiliza conjuntos de datos que no tengan una cantidad muy alta de grupos, debido a que será difícil visualizar las clases.

Por último, intenta crear un conjunto de datos que sea visiblemente separable para que compruebes la efectividad del algoritmo. Puedes crear un conjunto de datos sintético con dos, tres o más grupos visiblemente separables y probar el algoritmo que acabas de implementar.

8.15. Algoritmo K-vecinos más cercanos (KNN)

En este apartado, revisaremos el algoritmo de KNN. Este algoritmo es del tipo supervisado y nos permite realizar una clasificación/agrupación de nuestros datos de entrada.

El código se muestra en esta sección y puede ser descargado en:
http://www.amese.net/libro_ia/Prog/8.

El algoritmo se compone de los siguientes pasos:

- Determinar un valor de **k**-vecinos adecuado para nuestro conjunto de datos.
- Iterar cada punto de nuestro conjunto de prueba para determinar sus **k**-vecinos más cercanos.
- Asignar a dicho punto la clase que más se haya repetido dentro de sus vecinos más cercanos.
- Evaluar la precisión del clasificador.

1. Recolectar

Comenzamos por importar librerías, leer el archivo y desplegar una muestra de los datos. En esta ocasión, utilizaremos nuestro conjunto de datos de enfermedad coronaria (apéndice A.2), el cual contiene datos de problemas cardiacos de algunos pacientes junto con otros parámetros médicos y físicos de cada paciente.

Comenzamos importando las librerías y leyendo la base de datos, como se muestra en el siguiente código:

Código:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

df=pd.read_csv("HeartDisease.csv")
print("Dimensión:",df.shape)
df.head()
```

El código nos muestra las instancias y atributos de la base de datos así como un extracto del contenido de la base de datos.

Utilizamos la librería **seaborn** para visualizar rápidamente y de forma gráfica el comportamiento entre las variables en pares.

Código:

```
sns.pairplot(df, vars=df.columns[:3], hue=df.columns[-1])
```

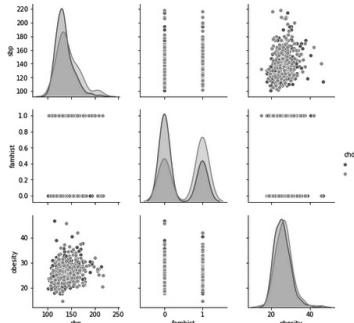


Figura 8.75. Distribución de datos de enfermedad coronaria para ejercicio de KNN

2. Preparar

Creamos una función para normalizar los valores entre 0 y 1. Para que las distancias no estén sesgadas por las dimensiones de estas.

Código:

```
def scale_zero_one(df):
    df2=df.copy()
    for col in df2.columns:
        minCol=min(df[col])
        maxCol=max(df[col])
        df2[col]=(df[col]-minCol) / (maxCol-minCol)
    return df2
df2=scale_zero_one(df)
df2.head()
```

```
Out[3]:
      sbp   famhist   obesity     age   chd
0  0.504274    1.0  0.332497  0.755102  1.0
1  0.367521    0.0  0.444479  0.979592  1.0
2  0.145299    1.0  0.452949  0.632653  0.0
3  0.589744    1.0  0.542346  0.877551  1.0
4  0.282051    1.0  0.354141  0.693878  1.0
```

Figura 8.76. Base de datos para el ejercicio de KNN, mostrando la normalización de los datos

3. Analizar

Nuevamente, revisamos los datos y observamos que no hay una diferencia significativa para este conjunto de datos debido a que el escalamiento no modifica la distribución de estos.

Código:

```
sns.pairplot(df2, vars=df2.columns[:3], hue=df2.columns[-1])
```

Procedemos a crear una clase que será nuestro clasificador KNN, en él agregamos métodos que nos faciliten el uso de nuestro clasificador.

Nota. No todos los métodos son necesarios para que funcione el clasificador KNN. A continuación, solo se explicarán los que son indispensables.

En nuestro método `__init__`, creamos las variables iniciales del clasificador:

- **best_k**: guardaremos la k que dé un mayor rendimiento durante el entrenamiento del clasificador.
- **best_acc**: la precisión asociada a la k elegida.

- **origin**: servirá para definir el punto de origen para medir distancias.
- **distances**: distancias de cada punto al origen.
- **X_pool**: conjunto X para comparación, adquirido en el entrenamiento.
- **y_pool**: valor objetivo asociado a la variable **X_pool**.

El método **setOrigin** nos permite cambiar el origen actual.

Después, **setDistances** es el que se encarga de calcular las distancias de cada punto al origen.

DistancePoints es usado por **setDistances** para hacer en sí el cálculo y regresarlo para que sea guardado. El método **fit** es uno de los más importantes, ya que se encarga del entrenamiento del clasificador. Dentro de **fit**, se usa **evaluate** para revisar la precisión que tiene cada cantidad de vecinos y así determinar la mejor. Aquí también se utiliza la función **belongsTo** que determina a qué clase pertenece cada punto en realidad.

Por último, **predict** se encarga de determinar a qué conjunto pertenece cada punto de prueba, de acuerdo a la **k** definida en el entrenamiento. Todos los métodos se muestran en el siguiente código:

Código:

```
class myKNN():
    """
    Este es mi clasificador sencillo de vecinos más cercanos. Busca la mejor k en intervalos de
    un factor determinado (predeterminado a 5).
    """

    def __init__(self):
        import numpy as np
        self.best_k=2
        self.best_acc=0
        self.origin=np.array([])
        self.distances=np.array([])
        self.X_pool=[]
        self.y_pool=[]

    def setOrigin(self, p1):
        self.origin=p1

    def distancePoints(self, p1, p2):
        mySum=0
        for x in range(p1.shape[0]):
            mySum+=((p2[x]-p1[x])**2)
        return mySum**(1/2)

    def getOrigin(self):
        return self.origin

    def setDistances(self, X):
        self.distances=np.array([])
        for x in range(X.shape[0]):
            self.distances=np.append(self.distances, self.distancePoints(self.origin, X[x,:]))

    def getDistances(self):
        return self.distances

    def belongsTo(self, sel):
```

```

uniqueVars=np.unique(sel)
counter=np.zeros(uniqueVars.shape[0])
for ind in sel:
    counter[np.where(ind == uniqueVars)[0][0]]+=1
higherCount=np.where(np.max(counter)==counter)[0][0]
cluster=uniqueVars[higherCount]
return cluster

def evaluate(self, k, factor):
    X_aux=self.X_pool.copy()
    index=np.arange(0,X_aux.shape[0])
    train_index=np.random.permutation(index)
    train_index=train_index[:int(0.2*index.shape[0])]
    accuracy=np.zeros(k)
    for ind in train_index:
        self.setOrigin(X_aux[ind])
        self.setDistances(np.delete(X_aux, ind, axis=0))
        db=np.zeros([(index.shape[0]-1),3])
        db[:,0]=np.delete(index, ind, axis=0)
        db[:,1]=self.distances
        db[:,2]=self.y_pool[index!=ind]
        db=db[db[:,1].argsort()]
        if self.belongsTo(db[:k*factor,2])==self.y_pool[ind]:
            accuracy[k-1]+=1
    accuracy[k-1]=accuracy[k-1]/train_index.shape[0]
    print("Precisión:",accuracy[k-1], '/', 1, end="\n\n")
    if self.best_acc<accuracy[k-1]:
        self.best_acc=accuracy[k-1]
        self.best_k=k*factor

    def fit(self, X, y, factor=5):
        self.X_pool=X
        self.y_pool=y
        for k in range(1,10):
            print("k calculada:",k*factor)
            self.evaluate(k, factor)
        print("k determinada:",self.best_k, "Precisión esperada:",self.best_acc)

    def predict(self, X):
        predictions=np.zeros(X.shape[0])
        cluster=[]
        nclusters=np.unique(self.y_pool).shape[0]
        for c in range(nclusters):
            cluster.append([])
        X_aux=X.copy()
        index_pool=np.arange(0,self.X_pool.shape[0])
        index=np.arange(0,X_aux.shape[0])
        for ind in index:
            self.setOrigin(X_aux[ind])

```

```

self.setDistances(self.X_pool)
db=np.zeros([(index_pool.shape[0]),3])
db[:,0]=index_pool
db[:,1]=self.distances
db[:,2]=self.y_pool
db=db[db[:,1].argsort()]
cluster[int(self.belongsTo(db[:,(self.best_k),2]))].append(ind)
predictions[ind]=int(self.belongsTo(db[:,(self.best_k),2]))
return predictions, np.array(cluster)

```

Revisemos un resumen de nuestra clase por medio de la función **help**:

Código:

```
help(myKNN)
```

```

Help on class myKNN in module __main__:

class myKNN(builtins.object)
| Este es mi clasificador sencillo de vecinos más cercanos. Busca la mejor k en intervalos de
| un factor determinado (predeterminado a 5).
|
| Methods defined here:
|
| __init__(self)
|     Initialize self. See help(type(self)) for accurate signature.
|
| belongsTo(self, sel)
|
| distancePoints(self, p1, p2)
|
| evaluate(self, k, factor)
|
| fit(self, X, y, factor=5)
|
| getDistances(self)
|
| getOrigin(self)
|
| predict(self, X)
|
| setDistances(self, X)
|
| setOrigin(self, p1)

_____
|
| Data descriptors defined here:
|
| __dict__
|     dictionary for instance variables (if defined)
|
| __weakref__
|     list of weak references to the object (if defined)

```

Figura 8.77. Resumen de la clase para el ejercicio de KNN

Podemos ver que todos los métodos aparecen, así como nuestra descripción de la clase.

Ahora, continuaremos con un pequeño análisis de nuestro conjunto de datos.

Tenemos 462 observaciones para este conjunto de datos, por lo cual tomaremos el 80, % para entrenamiento y el 20 % para pruebas.

Entrenamiento: 370

Prueba: 92

De igual manera, hay que considerar que debemos guardar nuestro valor objetivo en una variable **y**. Adicionalmente, guardando el resto de los datos en una variable **X**. Cada una con su partición **train** y **test**.

Código:

```

y_train, y_test = df["chd"][:370], df["chd"][370:]
X_train, X_test = df.values[:370,:], df.values[370,:,:]
print("Valores en X")
print("Entrenamiento:", X_train.shape, "Prueba:", X_test.shape, end="\n\n")
print("Valores en y")
print("Entrenamiento:", y_train.shape, "Prueba:", y_test.shape)

```

4. Entrenar

Ahora hacemos uso de la clase creada previamente y creamos el clasificador. También realizamos el entrenamiento con los conjuntos de entrenamiento mediante el siguiente código:

Código:

```

clf=myKNN()
clf.fit(X_train, y_train)

```

```

k calculada: 5
Precisión: 0.5540540540540541 / 1
k calculada: 10
Precisión: 0.6216216216216216 / 1
k calculada: 15
Precisión: 0.6486486486486487 / 1
k calculada: 20
Precisión: 0.6486486486486487 / 1
k calculada: 25
Precisión: 0.6756756756756757 / 1
k calculada: 30
Precisión: 0.5495495495495496 / 1
k calculada: 35
Precisión: 0.6621621621621622 / 1
k calculada: 40
Precisión: 0.6621621621621622 / 1
k calculada: 45
Precisión: 0.5945945945945946 / 1
k determinada: 25 Precisión esperada: 0.6756756756756757

```

Figura 8.78. Resultados del entrenamiento para el ejercicio de KNN

5. Probar

Creamos una función para facilitar el cálculo de la precisión obtenida por nuestro clasificador. En la cual se determina la cantidad de verdaderos positivos.

Código:

```

def accuracy(y, predictions):
    return (np.array([y==predictions]).sum())/y.shape[0]

```

Obtenemos las predicciones para nuestro conjunto de prueba y , posteriormente, desplegamos la precisión.

En esta sección, también obtenemos como variable los *clusters* con los índices obtenidos en el proceso de predicción.

Código:

```

predictions, clusters=clf.predict(X_test)
print("Precisión:")
accuracy(y_test, predictions)

```

Precisión:

0.7282608695652174

Finalmente, usamos los *clusters* obtenidos durante la predicción para comparar con el grupo real que ya conocemos. Seleccionamos dos dimensiones para una fácil visualización. Se usará la columna 0 y la 2.

Código:

```

plt.subplot(1,2,1)
for i in range(clusters.shape[0]):
    plt.scatter(X_test[clusters[i],0],X_test[clusters[i],2])
plt.title("Predicción")
plt.subplot(1,2,2)
for i in range(np.unique(clf.y_pool).shape[0]):
    plt.scatter(X_test[i==y_test,0],X_test[i==y_test,2])
plt.title("Real")
plt.show()

```

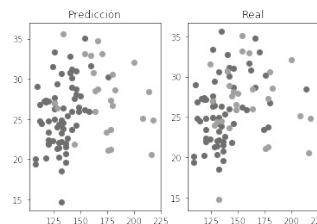


Figura 8.79. Datos de predicción vs datos reales para el algoritmo KNN

Puede observarse en la figura 8.79 que los grupos son muy parecidos —recuerda que el clasificador tiene un 72 % de precisión—, por lo cual el clasificador construido tiene un desempeño aceptable, aunque aún puede ser mejorado. Intenta modificar algunas secciones del código para que se haga una mejor predicción.

Nota. Puede ser posible que, debido a que en este ejemplo las clases están muy sobreplantadas, no se pueda tener una precisión muy alta en este conjunto de datos. Intenta con otro conjunto de datos para comparar tus resultados.

6. Usar

Hemos creado un clasificador **KNN** sencillo que nos permite determinar la clase a la que pertenece un conjunto de puntos, dado otro conjunto de puntos conocido por medio de los puntos adyacentes más cercanos.

Este clasificador puede mejorar, haciendo más eficiente la búsqueda de la cantidad de vecinos más cercanos que dan un mejor resultado. Usa el código proporcionado e intenta mejorar el clasificador.

Dentro de los usos que tiene KNN, se encuentra la agrupación de datos que tiene la característica de estar muy sobreplantadas. Esto presenta una ventaja frente a otros algoritmos, ya que, cuando los grupos no sean separables, el algoritmo tendrá un buen rendimiento si se ha realizado un buen entrenamiento.

8.16. Árboles de decisión

En la sección 5.7, «Aprendizaje por árboles de decisión», se comentó respecto a las ventajas y desventajas que tienen este tipo de algoritmos. Los árboles de decisión son útiles cuando se trabaja con variables categóricas, tiene relativamente pocas instancias con pocas observaciones y pocos atributos. A mayor número de dimensiones, mayor será la complejidad de realizar los cálculos para el buen funcionamiento del árbol, y mayor será la profundidad del árbol.

Sin embargo, es útil cuando se tienen pocas observaciones por atributo y para aprendizaje en línea, pues este tipo de algoritmos se pueden ajustar conforme se van obteniendo nuevos datos.

En este ejercicio, se muestra el código para poder implementar un árbol de decisión ID3, mismo que se muestra en: http://www.ameise.net/libro_ia/Prog/8.

La base de datos que se utilizará en este ejercicio es la base de datos de clima, que se explica a mayor detalle en el apéndice A.5. Normalmente, un árbol de decisión comienza con un nodo raíz y de ahí se calculan las ganancias de los demás atributos para formar el árbol. Un ejemplo de un árbol que se utiliza para decidir si la persona va a usar lentes o no, mostrando los atributos, nodos finales y las ganancias de cada atributo, se muestra en la figura 8.80.

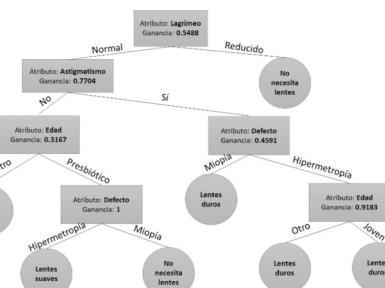


Figura 8.80. Ejemplo de un árbol de decisión mostrando sus atributos y las ganancias por atributo

Para implementar el árbol ID3 se siguen los siguientes pasos:

- Calcular la entropía para la base de datos.

- Para cada atributo:
 - Calcular la entropía para todos los otros atributos.
 - Calcular el promedio de ganancia de información.
 - Calcular la ganancia para el atributo.
- Seleccionar el atributo cuya ganancia sea la mayor.
- Repetir hasta obtener el árbol.

En este caso, no se realizó una separación de datos de entrenamiento y prueba, debido a la naturaleza del algoritmo y al número de instancias con el que cuenta esta base de datos –en este caso, catorce–. Si se desea tener más instancias, se puede hacer uso de la creación de datos sintéticos para poder realizar más pruebas.

También, es importante mencionar que, en este tipo de algoritmos, se busca que la cobertura sea cercana o igual al 100 %, por lo que no se realizó otro tipo de pruebas.

El código utiliza una técnica llamada de recursividad. La recursividad, en términos de programación, es una técnica donde una función se llama a sí misma en su definición. La estructura debe contener una condición de paro para evitar infinitas llamadas a sí misma.

En el caso del algoritmo ID3, la recursividad es una herramienta muy utilizada, ya que, para la construcción de un árbol en computación, suele utilizar la recursividad. Esto se asemeja mucho a iterar sobre los nodos de un árbol usando un algoritmo de búsqueda por profundidad. Al utilizar la recursividad, se construirá el árbol por profundidad y no por anchura, lo que permite reducir el tamaño del árbol hasta llegar a una condición de paro –nodo final–. El pseudocódigo es el siguiente:



Pseudocódigo Árbol ID3

```
dataset = leer_csv(path)
etiquetas =
'columna_que_contiene_las_etiquetas'
árbol = {}
árbol = helper_fn(dataset, etiquetas, [clases_de_las_etiquetas], árbol imprimir(árbol)
```




Pseudocódigo helper_fn

```
función helper_fn(dataset, etiquetas, [clases], árbol)
s = calcular_entropía(dataset, etiquetas, [clases])
ganancias = ganancia_por_atributo(dataset, s, etiquetas, [clases])
nodo, ganancia = obtener_nodo(ganancias)
árbol[nodo] = {}
ramas = obtener_clases(dataset, nodo)
dataset_particionado = particionar_dataset(dataset_nodo)
for rama y partición en ramas y particiones:
    árbol[nodo][rama] = {} validar_si_es_hoja(partición, etiquetas, clases) si es hoja:
        asignar_clase_final(árbol) si no:
            helper_fn(partición, etiquetas, clases, árbol[nodo][rama])
```

Para implementar el código, primero importamos las librerías y definimos la variable donde se guardará la base de datos, como se muestra en el siguiente código:

Código:

```
# Importamos las librerías y definimos la variable
# donde se guardará la base de datos, se convierte a
# formato .csv
import pandas as pd

def read_csv(input_csv):
    return pd.read_csv(input_csv)
```

Posteriormente, se leen los valores de la base de datos y se calcula la entropía total del sistema, como se muestra en el siguiente código:

Código:

```
# Se obtienen los valores de la base de datos
def get_p_and_n(df, values):
    denominator = []
    for value in values:
        denominator.append(len((df.loc[df[test_col] == value]).index))
    return denominator

# Se calcula la entropía total del sistema

def entropy(df, test_col, values):
    denominator = get_p_and_n(df, values)
    s = 0
    for value in denominator:
        if value == 0:
            s += 0
        else:
            s += (-value/(sum(denominator)))*math.log(value/(sum(denominator)), 2)
    return s
```

Luego, se calcula la ganancia por cada atributo con el siguiente código:

Código:

```
# Se calculan las ganancias por cada atributo
def gain_per_attr(df, s, test_col, values):

    denominator = get_p_and_n(df, values)
    gains = {}
    for feature in df.columns:
        if feature != test_col:
            gains[feature] = 0
            # Obtener los valores únicos de la característica
            values_in_feature = df[feature].unique()
            for value_in_feature in values_in_feature:
                # Construir el árbol a partir de la característica
                sub_df = df.loc[df[feature] == value_in_feature]
                # Obtener los contadores positivos y negativos de cada valor
                sub_denominator = get_p_and_n(sub_df, values)
                s_i = entropy(sub_df, test_col, values)
                gains[feature] += ((sum(sub_denominator))/(sum(denominator)))*s_i
            gains[feature] = -gains[feature] + s
    return gains
```

Una vez que se calculan las ganancias, se determina el nodo raíz, los nodos intermedios y el nodo final – también llamado hoja– de la siguiente manera:

Código:

```

# Se obtiene el nodo raíz
def get_root_node(gains):
    return max(gains.items(), key = lambda k : k[1])

# Se obtienen los nodos intermedios
def get_node_subsets(df, node):
    datasets = []
    branches = df[node].unique()
    for branch in branches:
        datasets.append(df.loc[df[node] == branch])
    return datasets

# Se determina si es un nodo final o no (hoja)
def is_leaf(df, test_col, values):
    rows = len(df.index)
    counter = 0
    for value in values:
        counter += 1
        if len((df.loc[df[test_col] == value]).index) == rows:
            return counter
    return 0

```

Por último, se manda llamar la función y se construye el árbol y se validan las instancias del atributo de salida: «¿Salir?», como se muestra en el siguiente código:

Código:

```

def helper_fn(df, test_col, values, tree):
    s = entropy(df, test_col, values)
    gains = gain_per_attr(df, s, test_col, values)
    node, gain = get_root_node(gains)
    tree[node] = {}
    branches = df[node].unique()
    subsets = get_node_subsets(df, node)
    for branch, subset in zip(branches, subsets):
        tree[node][branch] = {}
        # Validar si la rama es un nodo final
        leaf = is_leaf(subset, test_col, values)
        if leaf == 1:
            tree[node][branch] = values[0]
        elif leaf == 2:
            tree[node][branch] = values[1]
        elif leaf == 3:
            tree[node][branch] = values[2]
        else:
            helper_fn(subset, test_col, values, tree[node][branch])
    return tree

df = read_csv(r'Clima.csv')
test_col = '¿Salir?'

```

```

positive_value = 'Si'
negative_value = 'No'

tree = {}
tree = helper_fn(df, test_col, [positive_value, negative_value], tree)
print(tree)

```

Para realizar la validación, se crea la función **predict** para determinar si el árbol define las instancias como «¿Salir? = Sí», o «¿Salir? = No». Como se requiere una cobertura del 100 %, se prueban todas las instancias, como se muestra en el siguiente código:

Código:

```

# Predicción del árbol
def clima_predict(clima, temperatura, humedad, viento):
    if clima == 'soleado':
        if humedad == 'alta':
            print('No')
        elif humedad == 'normal':
            print('Si')
        elif clima == 'nublado':
            print('Si')
        elif clima == 'lluvioso':
            if viento == 'débil':
                print('Si')
            elif viento == 'fuerte':
                print('No')
    # Pruebas de ID3
    clima_predict('soleado', 'calido', 'alta', 'debil')
    clima_predict('soleado', 'calido', 'alta', 'fuerte')
    clima_predict('nublado', 'calido', 'alta', 'debil')
    clima_predict('lluvioso', 'templado', 'alta', 'debil')
    clima_predict('lluvioso', 'frio', 'normal', 'debil')
    clima_predict('lluvioso', 'frio', 'normal', 'fuerte')
    clima_predict('nublado', 'frio', 'normal', 'fuerte')
    clima_predict('soleado', 'templado', 'alta', 'debil')
    clima_predict('soleado', 'frio', 'normal', 'debil')
    clima_predict('lluvioso', 'templado', 'normal', 'debil')
    clima_predict('soleado', 'templado', 'normal', 'fuerte')
    clima_predict('nublado', 'templado', 'alta', 'fuerte')
    clima_predict('nublado', 'calido', 'normal', 'debil')
    clima_predict('lluvioso', 'templado', 'alta', 'fuerte')

```

El listado de resultados es el siguiente:

No No Sí Sí Sí No Sí

No Sí Sí Sí Sí Sí No

Esto nos indica que el árbol muestra los resultados correctos. Como ejercicio adicional, la base de datos de meningitis del apéndice A.6 puede ser utilizada con este ejercicio. Busca que el árbol se construya de manera correcta y se tenga una cobertura del 100 %.

8.17. Algoritmo de bosques aleatorios

Como se mencionó en la sección 5.8, los bosques aleatorios son un algoritmo utilizado para problemas de clasificación y regresión. Este algoritmo crea subconjuntos de datos aleatorios a partir del conjunto de datos original y, con cada uno de esos subconjuntos, crea un árbol cuyos nodos selecciona de manera aleatoria también.

El código se muestra en esta sección y puede ser descargado en: http://www.amese.net/libro_ia/Prog/8.

Como ejemplo de este algoritmo, haremos una clasificación de los pasajeros del Titanic, con la base de datos mostrado en el apéndice A.8.1, los clasificaremos en supervivientes y no supervivientes.

Para la construcción del algoritmo, usaremos la librería **scikit-learn (sklearn)**, específicamente, la clase **RandomForestClassifier**. Esta clase trabaja únicamente con valores numéricos –sklearn también cuenta con una clase para realizar modelos de regresión que se llama *RandomForestRegressor*–.

Si no tenemos instalado **scikit-learn**, podemos usar el siguiente comando:

```
pip install -U scikit-learn
```

Ahora procedemos a realizar el código.

Importamos las librerías.

Código:

```
import numpy as np
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
```

Abrimos el archivo que contiene los datos del Titanic con el siguiente código:

Código:

```
df= pd.read_csv("titanic.csv")
df.head()
```

Un extracto de la base de datos se muestra en la figura 8.81.

PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	3	Braund, Mr. Owen Harris	male	22.0	1	0	313338	7.2500	Nan	S
1	2	1	Cumings, Mrs. John Bradley (Florence Brigitte Th...)	female	38.0	1	0	PC 17599	71.2833	C85	C
2	3	1	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O/O 3101280	7.9250	Nan	S
3	4	1	Futrelle, Mrs. Jacques Heath (Clytie Purcell)	female	35.0	1	0	113803	53.1000	C123	S
4	5	0	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0000	Nan	S

Figura 8.81. Datos de la BD de Titanic para el algoritmo de bosques aleatorios

Como podemos observar en la figura 8.81, no todas las columnas son relevantes para nuestro propósito, por lo que vamos a seleccionar solo los atributos que nos interesan, los cuales son: «Survived, Pclass, Sex, Age, SibSp, Parch y Embarked», como se muestra en la figura 8.82.

Código:

```
df = df[["Survived", "Pclass", "Sex", "Age", "SibSp", "Parch", "Embarked"]]
df.head()
```

Survived	Pclass	Sex	Age	SibSp	Parch	Embarked	
0	0	3	male	22.0	1	0	S
1	1	1	female	38.0	1	0	C
2	1	3	female	26.0	0	0	S
3	1	1	female	35.0	1	0	S
4	0	3	male	35.0	0	0	S

Figura 8.82. Listado de atributos de la BD de Titanic

Vemos que tenemos dos atributos no numéricos, por lo que debemos convertirlos para poder utilizar la función **RandomForestClassifier**. La conversión la podemos hacer con una función de **pandas** que se llama **get_dummies**; **get_dummies** nos devuelve columnas cualitativas convertidas en columnas numéricas con 0 y 1. Esta función recibe como parámetros el *dataframe*, las columnas sobre las cuales se hará la conversión, a modo de lista, y vamos a mandar verdadero a la variable **drop_first** que es falsa por defecto. Si dejamos **drop_first** como falso, tendríamos una columna por cada observación, por ejemplo, en el caso de la columna «Sex», tendríamos una columna llamada *Sex_male* y otra llamada *Sex_female*. En cambio, al volver verdadero a **drop_first**, vamos a tener una sola columna llamada *Sex_male* donde se usa 1 para hombre y 0 para mujer. El código se muestra a continuación:

Código:

```
df = pd.get_dummies(df, columns=["Sex", "Embarked"], drop_first=True)
df.head()
```

En la figura 8.83 se muestra dicha conversión.

	Survived	Pclass	Age	SibSp	Parch	Sex_male	Embarked_Q	Embarked_S
0	0	3	22.0	1	0	1	0	1
1	1	1	38.0	1	0	0	0	0
2	1	3	26.0	0	0	0	0	1
3	1	1	35.0	1	0	0	0	1
4	0	3	35.0	0	0	1	0	1

Figura 8.83. Conversión de atributos numéricos mediante la función *get_dummies*

Ya que todas nuestras columnas tienen valores numéricos, procedemos a verificar si existen valores nulos. Lo realizamos con la función **info** de **pandas** mediante el siguiente código (figura 8.84):

Código:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 8 columns):
 #   Column   Non-Null Count  Dtype  
 --- 
 0   Survived  891 non-null   int64  
 1   Pclass    891 non-null   int64  
 2   Age       714 non-null   float64 
 3   SibSp    891 non-null   int64  
 4   Parch    891 non-null   int64  
 5   Sex_male 891 non-null   uint8  
 6   Embarked_Q 891 non-null   uint8  
 7   Embarked_S 891 non-null   uint8  
dtypes: float64(1), int64(4), uint8(3)
memory usage: 37.5 KB
```

Figura 8.84. Información de los atributos de la BD de Titanic con la observación de si existen datos nulos

En este caso, tenemos 177 valores nulos, los cuales eliminaremos por practicidad de este ejemplo, aunque se recomienda que se imputen, ya sea con una imputación simple o con una imputación múltiple como MICE. En este caso, eliminamos los valores nulos con **dropna**. El resultado de eliminar los valores nulos se muestra en la figura 8.85.

Código:

```
df = df.dropna()
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 714 entries, 0 to 890
Data columns (total 8 columns):
 #   Column   Non-Null Count  Dtype  
 --- 
 0   Survived  714 non-null   int64  
 1   Pclass    714 non-null   int64  
 2   Age       714 non-null   float64 
 3   SibSp    714 non-null   int64  
 4   Parch    714 non-null   int64  
 5   Sex_male 714 non-null   uint8  
 6   Embarked_Q 714 non-null   uint8  
 7   Embarked_S 714 non-null   uint8  
dtypes: float64(1), int64(4), uint8(3)
memory usage: 35.6 KB
```

Figura 8.85. Información de los atributos de la BD de Titanic sin instancias nulas

Ahora, ya terminamos de preparar nuestros datos y estamos listos para implementar el bosque aleatorio.

Para la construcción de nuestro bosque, vamos a implementar la validación cruzada **k-fold** vista anteriormente, así que importamos nuestras funciones de la validación, como se muestra en el siguiente código:

```
Código:  
def division(df, k):  
    datos = []  
    tamano = int(len(df)/k)  
    for i in range(k-1):  
        division = np.array(range(i*tamano, (i+1)*tamano))  
        datos.append(np.array(division))  
    datos.append(np.array(range((i+1)*tamano, len(df))))  
    return datos  
  
# Se filtran los datos para tener entrenamiento y prueba  
def conjunto(df, datos, indice):  
    prueba = df.iloc[datos[indice]]  
    ind = np.array(range(len(df)))  
    ind_ent = np.delete(ind, datos[indice])  
    entrenamiento = df.iloc[ind_ent]  
    return entrenamiento, prueba
```

Creamos una función en donde ejecutaremos la validación **k-fold** e iremos creando un bosque por cada subconjunto que tengamos. Esta función **crear_bosque** recibe como parámetros un **dataframe df**, el número de particiones en los que se dividirán los datos **k**, el número de estimadores llamado así –**estimadores**–, el número mínimo de hojas –también llamado **hojas**–, la semilla para el estado aleatorio –**semilla**– y el nombre de la columna donde se encuentra la salida o variable dependiente –el cual, en este caso, se llamará **salida**–.

Dentro de la función, inicializaremos una variable **precisión** donde iremos sumando la precisión de cada bosque para poder obtener la media, una vez que termine con los diferentes bosques.

Posteriormente, llamamos a **división** para dividir nuestros datos e implementamos un ciclo en el cual cada iteración corresponderá a un bosque aleatorio. En el ciclo obtenemos los datos de entrenamiento y de prueba, sepáramos la salida de los demás atributos –requisito para poder usar la clase **RandomForestClassifier**–, para eliminar una columna de un **dataframe** usamos la función **drop**.

Esta función nos permite eliminar filas o columnas: **axis = 0** para indicar que vamos a eliminar filas y **axis = 1** para indicar que se trata de columnas. También, hay que mandarle el índice de las filas o el nombre de las columnas a eliminar en forma de lista.

Ya que hemos separado la salida de los demás atributos, instanciamos **RandomForestClassifier** mandándole como parámetros el número de estimadores –número de árboles–, la semilla para inicializar el estado aleatorio –solo en la construcción del primer árbol– y el número mínimo de hojas de los árboles.

Posteriormente, entrenamos el modelo llamando **fit**, mandándole los datos de entrenamiento –los atributos y la salida– y, finalmente, probamos el bosque con los datos de prueba, esto lo hacemos con la función **score**, la cual recibe como parámetros los datos de prueba y nos devuelve la precisión del árbol como un número entre 0 y 1. Recordemos que, por cada subconjunto creado por la validación cruzada **k-fold**, estamos creando un bosque y vamos a retornar el promedio de la precisión de todos.

Código:

```
def crear_bosque(df, k, estimadores, hojas, semilla, salida):
    # Inicializamos la variable de la precisión
    precision = 0.0
    # Hacemos las divisiones del conjunto de datos
    datos = division(df, k)
    # Por cada subconjunto se crea un bosque y luego promediamos la precisión
    for i in range(k):

        # Obtenemos los datos de entrenamiento y prueba
        entrenamiento, prueba = conjunto(df, datos, i)

        # Separamos la salida de los demás atributos
        entrenamiento_atributos = entrenamiento.drop([salida], axis=1)
        entrenamiento_salida = entrenamiento[salida]
        prueba_atributos = prueba.drop([salida], axis=1)
        prueba_salida = prueba[salida]

        # Damos los hiperparámetros del bosque

        # Establecemos la semilla en la primera iteración
        if i == 0:
            bosque = RandomForestClassifier(n_estimators=estimadores, random_state=semilla, min_samples_leaf=hojas)
        else:
            bosque = RandomForestClassifier(n_estimators=estimadores, min_samples_leaf=hojas)
        # Entrenamos el bosque
        bosque.fit(entrenamiento_atributos, entrenamiento_salida)
        # Probamos el bosque y sumamos la precisión obtenida
        precision += bosque.score(prueba_atributos, prueba_salida)
        # Retornamos el promedio de la precisión obtenida
    return round(precision/k, 3)
```

Ahora, vamos a probar nuestro código y crear bosques con distinto número de árboles, mientras los demás parámetros los mantenemos igual –k, estimadores, semilla, número de hojas y la salida que se deseé–. En la figura 8.86, se muestra la salida de la ejecución del árbol.

The screenshot shows three code cells in a Jupyter Notebook. The first cell, labeled 'Un árbol', contains the command `In [9]: 1 crear_bosque(df, 5, 1, 8, 1, "Survived")` and the output `Out[9]: 0.759`. The second cell, labeled 'Cinco árboles', contains the command `In [10]: 1 crear_bosque(df, 5, 5, 8, 1, "Survived")` and the output `Out[10]: 0.795`. The third cell, labeled 'Diez árboles', contains the command `In [11]: 1 crear_bosque(df, 5, 10, 8, 1, "Survived")` and the output `Out[11]: 0.807`.

Figura 8.86. Salida del algoritmo de bosque aleatorio

Como podemos observar, a medida que incrementamos el número de árboles, incrementa la precisión del bosque hasta un cierto límite. Es importante mencionar que, debido a que los estimadores los realiza de forma aleatoria, los resultados pueden no ser exactamente iguales a los mostrados en la figura 8.86.

8.18. Red neuronal simple (perceptrón)

Para poder entender el funcionamiento de las redes neuronales, como se vio en el capítulo 6 de este libro,

comenzaremos con construir una. En este sencillo ejemplo, construiremos un perceptrón de una sola capa. El código se muestra en esta sección y puede ser descargado en: http://www.amese.net/libro_ia/Prog/8.

Para esta práctica, no se requerirán datos de las bases de datos contenidas en el apéndice A. Sin embargo, es importante mencionar que se utilizó la librería **neurolab** versión 0.3.5, misma que puede ser descargada sin pago de regalías desde la página de **neurolab** (pypi.org).

Una vez que se descargue, se puede instalar ya sea con la plataforma de Anaconda o desde la terminal de comandos de la siguiente manera:

```
conda install -c labfabulous neurolab
```

0:

```
pip install neurolab
```

El perceptrón es lo que podemos llamar una red neuronal de una sola neurona. El diagrama de un perceptrón simple de una capa se muestra en la figura 8.87. En este ejemplo, abordaremos cómo utilizar este modelo como un clasificador.

Para exemplificar el perceptrón, vamos a replicar el comportamiento de la función lógica **OR**, cuyo diagrama se muestra en la figura 8.88. Durante este ejemplo, nos vamos a apoyar de la librería **neurolab** que nos permite crear varios modelos de redes neuronales, entre ellos, el perceptrón.

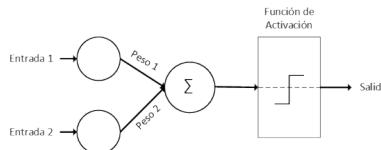


Figura 8.87. Diagrama de un perceptrón simple de una capa



Figura 8.88. Diagrama de una compuerta OR de dos entradas

Como se puede ver en las figuras 8.87 y 8.88, existe cierta relación entre el funcionamiento de una compuerta lógica y el funcionamiento de un perceptrón simple, mismo que será ejemplificado para poder entrenar un perceptrón y que pueda aprender el comportamiento de dicha compuerta. Como recordatorio, una compuerta lógica OR tiene una salida de un «1» cuando, por lo menos, una de sus entradas sea «1», de lo contrario —todas sus entradas son «0»—, tendrá un «0» a la salida.

Para comenzar el perceptrón, importamos librerías que vamos a utilizar, como se muestra en el siguiente código:

Código:

```
import numpy as np
import matplotlib.pyplot as plt
import neurolab as nl
```

Creamos nuestros conjuntos —las entradas y salidas que tendrá nuestro perceptrón—, donde las entradas son las posibles combinaciones que puede tener la función lógica **OR** y las salidas son el resultado esperado. Como podemos observar, tenemos dos neuronas de entrada y una de salida por cada ejemplo.

Código:

```

entradas = np.array([[0, 1], [1, 1], [0, 0], [1, 0]])
entradas

Out: array([[0, 1],
[1, 1],
[0, 0],
[1, 0]])

Código:
salidas = np.array([[1], [1], [0], [1]])
salidas

```

Definimos las dimensiones de nuestras entradas, es decir, el intervalo en el que se encuentran los valores de entrada, en este caso, son $[0, 1]$ para ambas entradas.

```

Código:
dim1_min, dim1_max, dim2_min, dim2_max = 0, 1, 0, 1
dim1 = [dim1_min, dim1_max]
dim2 = [dim2_min, dim2_max]

```

Definimos el número de capas de salida, en este caso, esperamos un solo valor, por lo que será una salida de una sola capa.

```

Código:
neu_salida = 1

```

Creamos nuestro perceptrón, que es una instancia de la clase **newp**. Para crearlo, es necesario enviarle como parámetros las dimensiones de las entradas en forma de lista y el número de neuronas de salida.

```

Código:
perceptron = nl.net.newp([dim1, dim2], neu_salida)

```

Una vez creado nuestro perceptrón, lo entrenamos con la función **train** propia de la clase **newp**, a esta función hay que mandarle, en los parámetros, los datos de entrenamiento, es decir, los datos de entrada y las salidas que se esperan, el número de épocas **epoch**s, que es el número máximo de ciclos de entrenamiento que puede tener la red, **show**, que imprime por consola el progreso del entrenamiento cada n veces, y la tasa de aprendizaje **lr (learning rate)**. Además, en nuestro ejemplo, vamos a graficar el proceso para hacerlo más visual, como se muestra en la figura 8.89.

```

Código:
# Entrenamos el perceptrón
progreso_error = perceptron.train(entradas, salidas, epoch=10, show=1, lr=0.3)
# Visualizamos el progreso del error
plt.figure()
plt.plot(progreso_error)
plt.xlabel('Número de épocas')
plt.ylabel('Error')
plt.title('Progreso del error')
plt.grid()
plt.show()

```

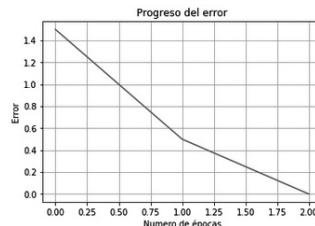


Figura 8.89. Error por número de épocas de entrenamiento de una compuerta lógica OR por medio de un perceptrón simple

```

Epoch: 1; Error: 1.5;
Epoch: 2; Error: 0.5;
Epoch: 3; Error: 0.0;
The goal of learning is reached

```

Ahora construiremos un conjunto de datos de prueba con el que evaluaremos nuestro perceptrón.

Código:

```
# Crear un conjunto de pruebas
prueba = np.array([[0, 0], [1, 1], [1, 0], [0, 1]])
```

Posteriormente, evaluaremos el perceptrón con su función **sim**, que recibe los datos de entrada y nos devuelve la salida de acuerdo con el entrenamiento que tuvo la neurona.

Código:

```
for dato in prueba:
    salida = perceptron.sim([dato])
    print("{} OR {} = {}".format(dato[0], dato[1], salida[0]))

0 OR 0 = [0.]
1 OR 1 = [1.]
1 OR 0 = [1.]
0 OR 1 = [1.]
```

Como podemos observar, las predicciones de nuestro perceptrón son correctas. Es importante mencionar que se pueden hacer tantas pruebas como se deseen.

Ahora hagamos la función lógica **XOR**. La compuerta lógica XOR define su salida de acuerdo al número de entradas en 1 que tiene la compuerta. En ocasiones, se refiere a ella como «si son iguales es 0 y si son diferentes es 1». Esto no es del todo correcto, debido a que, en una compuerta XOR de tres entradas, no se podría tener esa lógica para las combinaciones: 110, 011, 101 y 111, pues darían resultados incorrectos.

En este caso, se cuentan las entradas que tienen 1. Si el número de entradas que tienen uno es 0, la salida es uno, de lo contrario, la salida es 0. El diagrama lógico de la compuerta XOR se muestra en la figura 8.90.



Figura 8.90. Diagrama de una compuerta XOR de dos entradas

Definimos los datos de entrenamiento, como se muestra en el siguiente código:

```
Código:
entradas = np.array([[0, 1], [1, 0], [1, 1], [0, 0]])
entradas
```

```

array([[0, 1],
[1, 0],
[1, 1],
[0, 0]])

Código:
salidas = np.array([[1], [1], [0], [0]])
salidas

array([[1],
[1],
[0],
[0]])

```

Las dimensiones y el número de neuronas de salida permanecen constantes. Entonces, procedemos a crear nuestro perceptrón con estos nuevos datos, lo llamaremos **perceptron1**.

```

Código:
perceptron1 = nl.net.newp([dim1, dim2], neu_salida)

```

Realizamos el proceso de entrenamiento y mostramos los resultados como se puede observar en la figura 8.91.

```

Código:
# Entrenamos el perceptrón
progreso_error = perceptron1.train(entradas, salidas, epochs=100, show=20, lr=0.01)
# Visualizamos el progreso del error
plt.figure()
plt.plot(progreso_error)
plt.xlabel('Número de épocas')
plt.ylabel('Error')
plt.title('Progreso del error')
plt.grid()
plt.show()

Epoch: 20; Error: 1.0;
Epoch: 40; Error: 1.0;
Epoch: 60; Error: 1.0;
Epoch: 80; Error: 1.0;
Epoch: 100; Error: 1.0;
The maximum number of train epochs is reached

```



Figura 8.91. Error por número de épocas de entrenamiento de una compuerta lógica XOR por medio de un perceptrón simple

Pero ahora observamos que, a pesar de ser ejecutado por el máximo de 100 épocas, nuestro error se mantiene en 1.

Ejecutando el perceptrón con un *dataset* de prueba obtenemos los siguientes resultados.

Código:

```
# Crear un conjunto de pruebas
prueba = np.array([[1, 0], [0, 0], [1, 1], [0, 1]])

for dato in prueba:
    salida = perceptron.sim([dato])
    print("{} XOR {} = {}".format(dato[0], dato[1], salida[0]))
```

1 XOR 0 = [0.]
0 XOR 0 = [0.]
1 XOR 1 = [0.]
0 XOR 1 = [0.]

Como podemos observar, nuestro modelo se ha equivocado en los casos en que 1 XOR 0 y 0 XOR 1, esto se debe a que el perceptrón de una capa solo puede clasificar datos que sean linealmente separables, y el comportamiento de la función lógica **XOR** no es linealmente separable. Esta es una de las limitaciones que presenta el perceptrón de una sola capa y que puede resolverse con un perceptrón multicapa, comúnmente llamado MLP.

8.19. Perceptrón multicapa (MLP)

Una red multicapa nos sirve para resolver problemas más complejos. En las redes multicapa, las salidas de algunas neuronas se convierten en la entrada de las siguientes neuronas, lo que nos permite obtener una clasificación o regresión no lineal. En el ejemplo del perceptrón simple (práctica 8.18), no se logró modelar con éxito una compuerta lógica XOR, precisamente, debido a la linealidad que tiene dicha red. En este ejemplo, crearemos una red multicapa con varias neuronas dentro de cada capa.

El código se muestra en esta sección y puede ser descargado en:

http://www.amese.net/libro_ia/Prog/8.

Una red multicapa tiene una forma parecida a como se muestra en la figura 8.92.

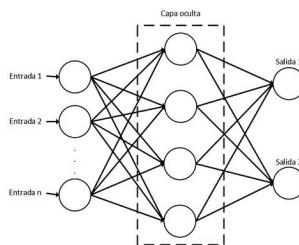


Figura 8.92. Ejemplo de un perceptrón multicapa (MLP)

En la figura 8.92, se muestra un ejemplo de un perceptrón multicapa. El número de neuronas de entrada y de capas ocultas puede ser muy variable, lo mismo que el número de neuronas en la/las capas ocultas. El número de neuronas de la capa de salida, usualmente, es igual al número de clases que tienen los datos cuando se requiere clasificar.

Comenzamos por importar las librerías de nuevo:

Código:

```

import numpy as np
import matplotlib.pyplot as plt
import neurolab as nl

```

Para este ejemplo, tampoco vamos a requerir alguna base de datos como las que se describen en el apéndice A, aunque la misma librería **neurolab**, como se mostró en el ejercicio del perceptrón simple, es fundamental instalarla. Así mismo, vamos a utilizar una ecuación muy conocida que se conoce como función de Rosenbrock. Crearemos una red que nos permita predecir los valores que toma esta ecuación, como se muestra en la ecuación 8.12.

$$\sum_{i=1}^{n-1} \left[100(x_i^2 - x_{i+1})^2 + (x_i - 1)^2 \right] \quad 8.12)$$

Vamos a plantear $[-5, 10]$ como el espacio en el que trabajaremos esta función. Para los datos de entrenamiento, crearemos un vector de números enteros en este intervalo.

Código:

```

lim_inf = -5
lim_sup = 10
ejex = ejey = np.linspace(lim_inf, lim_sup, 50)

```

Ya con nuestro espacio creado, definimos la función **Rosenbrock**. La función la vamos a trabajar de manera vectorial, es decir, en lugar de evaluar un punto y, a la vez, vamos a evaluar todo el vector con el fin de ahorrarnos tiempo computacional. **Rosenbrock** va a recibir dos variables: **x**, **y**, ambas vectores que nos facilitarán las operaciones algebraicas. Con estos vectores, definimos la función y la guardamos en **z** –al elevar al cuadrado un vector de *numpy* se realiza elemento por elemento, no es la multiplicación de matrices que se realiza en el álgebra lineal–.

Finalmente, vamos a devolver el resultado en **z**.

Código:

```

def rosenbrock(x, y):
    z = 100*(x**2 - y)**2 + (x-1)**2
    return z

```

Ahora, vamos a crear las matrices que forman las parejas ordenadas que le mandaremos a la función para crear una superficie utilizando **meshgrid**, después, llamamos a la función **Rosenbrock** mandándole las matrices que creamos y guardando la matriz que nos retorna en **z**. Finalmente, procedemos a graficar (figura 8.93).

Código:

```

# Creamos las parejas ordenadas
x, y = np.meshgrid(ejex, ejey)
# Evaluamos la función
z = rosenbrock(x, y)
# Se procede a graficar
plt.gca(projection='3d').plot_surface(x, y, z, cmap='jet')
plt.title("Función de Rosenbrock")
plt.show()

```

Función de Rosenbrock

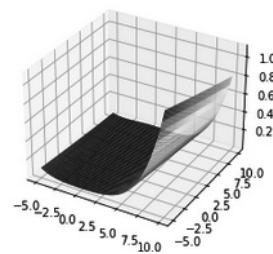


Figura 8.93. Función de Rosenbrock

Ya que tenemos los valores que toma la función y los valores que nos retorna, hay que preparar esos datos para entrenar nuestra red neuronal.

Definimos el rango que tienen nuestros datos de entrada, que es $[-5, 10]$ para ambas variables, y normalizamos los datos de **z**, debido a que la función nos devuelve valores muy grandes –algunos mayores a 800 000–.

Código:

```
# Establecemos las dimensiones de nuestras entradas
dim = [lim_inf, lim_sup]
# Normalizamos los datos
z = z/numpy.linalg.norm(z)
```

Luego procedemos a transponer los datos –requisito para utilizar crear nuestra red con **neurolab**–.

Código:

```
# Transponemos nuestros datos
x = x.reshape(-1, 1)
y = y.reshape(-1, 1)
z = z.reshape(-1, 1)
```

Y, finalmente, vamos a concatenar **x** e **y**, que son nuestros valores de entrada, ya que tienen que estar en una sola matriz cuando los pasemos a nuestra red para entrenarla. Este vector tiene las mismas dimensiones que establecimos anteriormente para ambas entradas, **z** lo dejamos igual porque solo queremos una neurona de salida.

Código:

```
X = np.concatenate((x, y), axis=1)
```

```
X
```

```
array([[-5. , -5. ],
[-4.69387755, -5. ],
[-4.3877551 , -5. ],
...,
[ 9.3877551 , 10. ],
[ 9.69387755, 10. ],
[10. , 10. ]])
```

En este punto, se puede elegir entre importar la red ya creada y omitir el entrenamiento de la red que se mostrará a continuación o entrenar la red con los datos que tenemos. La red de ejemplo puede ser descargada en: http://www.amese.net/libro_ia/Prog/

Si se desea cargar la red ya creada, se carga el archivo con el siguiente código:

Código:

```
# IMPORTAR LA RED YA CREADA
net = nl.load("Rosenbrock.net")
```

En este caso en particular, se creará una red con la siguiente topología: tres capas, la primera con diez neuronas, la segunda con siete y la tercera con tres, con una sola neurona de salida. En este ejemplo, se utiliza la función **newff** que recibe como parámetros las dimensiones de las neuronas de entrada y como lista las neuronas que habrá en cada capa exceptuando la capa de entrada.

Vamos a establecer el error de la red a MAE (*mean absolute error*) y procedemos a entrenarla. Le mandamos las entradas que están en **X** y las salidas deseadas que están en **z**, la limitamos a 200 épocas y a un error mínimo de 0.001.

Posteriormente, mostramos y graficamos el progreso del error como se muestra en la figura 8.94.

Código:

```
# Creamos el modelo
net = nl.net.newff([dim, dim], [10, 7, 3, 1])

# Cambiamos el error
net.errorf = nl.error.MAE()

# Entrenamos la red
error = net.train(X, z, epochs=200, show=20, goal=0.001)

# Graficamos el error
plt.figure()
plt.plot(error)
plt.xlabel("Número de épocas")
plt.ylabel("Error")
plt.title("Progreso del error")
```

```
Epoch: 20; Error: 0.000296776313550927;
Epoch: 40; Error: 0.001714817744785535;
The goal of learning is reached
```

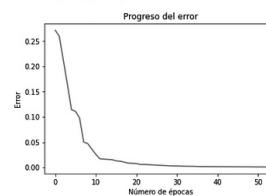


Figura 8.94. Error de aprendizaje para el ejercicio MLP de la función de Rosenbrock

Vamos a generar datos de prueba para ver cómo se comporta nuestra red. El conjunto de datos de prueba que generaremos es un conjunto más denso que el que utilizamos para entrenar la red, pues este consta de cien elementos. Una vez preprocesados los datos, los evaluamos con la red con la función **sim** y los graficamos para ver los datos reales vs los datos predichos. Despues de preparar los datos para evaluarlos con nuestra red, tenemos que reacomodarlos a su forma original para poder graficar sin problemas. La forma original la podemos tomar de **z** y la guardamos en **forma** para reordenarlos después de evaluarlos y se grafica la diferencia como se muestra en la figura 8.95.

Código:

```

# Generamos datos de prueba
ejex = ejey = np.linspace(lim_inf, lim_sup, 100)
# Creamos las parejas ordenadas
x, y = np.meshgrid(ejex, ejey)
# Evaluamos la función con los datos de prueba
z = rosenbrock(x, y)
# Guardamos la forma de los datos
forma = z.shape
# Normalizamos los datos
z = z/np.linalg.norm(z)
# Transponemos nuestros datos
x = x.reshape(-1, 1)
y = y.reshape(-1, 1)
# Concatenamos los datos de entrada
X = np.concatenate((x, y), axis=1)
# Evaluamos la red
z_pred = net.sim(X)
# Reacomodamos los datos para graficarlos
x = x.reshape(forma)
y = y.reshape(forma)
z_pred = z_pred.reshape(forma)

```

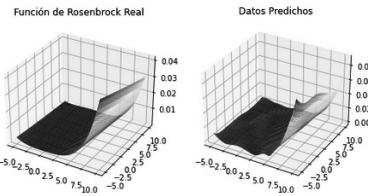


Figura 8.95. Comparación entre el entrenamiento (izquierda) y la prueba (derecha) de una función de Rosenbrock con un MLP

Se puede observar que la estimación no es perfecta. Sin embargo, en base a los hiperparámetros y la topología de la red, se puede estimar con una buena aproximación. Si se quiere guardar la red entrenada para este ejemplo, se utiliza el siguiente código (opcional):

Código:

```
# Línea para guardar la red porque cada vez que se ejecuta cambia
net.save("Rosenbrock.net")
```

8.20. Red neuronal convolutiva (CNN)

Como se comentó en la sección 6.2, las redes neuronales convolucionales son muy utilizadas hoy en día, principalmente, para aplicaciones que tienen que ver con imágenes.

En este ejemplo, abordaremos el uso y construcción de una CNN para realizar la clasificación de algunas imágenes. Estaremos utilizando PyTorch, que es un *framework* de inteligencia artificial de código abierto, de fácil uso y nos ahorra tiempo al momento de programar.

Se puede instalar PyTorch desde el entorno de Conda con el siguiente comando:

Windows

Con cuda:

```
conda install pytorch torchvision torchaudio cudatoolkit=10.2 -c pytorch
```

Sin cuda:

```
conda install pytorch torchvision torchaudio cpuonly -c pytorch
```

O utilizando pip:

Con cuda:

```
pip3 install torch==1.8.1+cu102 torchvision== 0.9.1+cu102 torchaudio== 0.8.1 -f  
https://download.pytorch.org/whl/torch\_stable.html
```

Sin cuda:

```
pip3 install torch==1.8.1+cpu torchvision== 0.9.1+cpu torchaudio== 0.8.1 -f  
https://download.pytorch.org/whl/torch\_stable.html
```

Mac

Conda

```
conda install pytorch torchvision torchaudio -c pytorch
```

O utilizando pip:

```
pip3 install torch torchvision torchaudio
```

Linux

Conda

Con cuda:

```
conda install pytorch torchvision torchaudio cudatoolkit=10.2 -c pytorch
```

Sin cuda:

```
conda install pytorch torchvision torchaudio cpoonly -c pytorch
```

O utilizando pip:

Con cuda:

```
pip3 install torch torchvision torchaudio
```

Sin cuda:

```
pip3 install torch==1.8.1+cpu torchvision== 0.9.1+cpu torchaudio== 0.8.1 -f  
https://download.pytorch.org/whl/torch\_stable.html
```

El código se muestra en esta sección y puede ser descargado en: http://www.amese.net/libro_ia/20CNN

Por otro lado, las imágenes se tuvieron que comprimir en un archivo con extensión .zip llamado «img_cnn.zip» ubicado en: http://www.amese.net/libro_ia/img_cnn.zip

En ese directorio, se tiene que descomprimir y crear una carpeta llamada «Gatos». Dicha carpeta contendrá otras dos carpetas: una llamada «Gatos» y la otra «Random», esta última contiene las demás imágenes que no son gatos.

Una vez instalado PyTorch, comenzemos con el ejemplo. Tenemos imágenes varias, algunas de animales como de perros y gatos. En este ejemplo, crearemos una red neuronal convolutiva (CNN) que sea capaz de

clasificarlos entre los que son gatos y los que no.

Comenzaremos importando librerías de utilidad, como `numpy`, `matplotlib`, `torch`, `torchvision` y algunos otros módulos de `torch`, como se muestra en el siguiente código:

Código:

```
import numpy as np
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
from torchvision import transforms, datasets, models
from torch.utils.data import random_split
```

Una vez importadas las librerías, creamos un diccionario, mismo que servirá más adelante para poder realizar la clasificación de las imágenes. Este diccionario contendrá dos valores: 0 y 1, que apuntarán a «Gato» y «No es gato», respectivamente, como se muestra en el siguiente código:

Código:

```
# Crear diccionario para clasificación de imágenes
clases = dict({0:"Gato", 1:"No es gato"})
clases

{0: 'Gato', 1: 'No es gato'}
```

Lo siguiente es cargar el dataset de imágenes. Para esto, utilizaremos las funcionalidades de **torchvision**, la primera proviene de **dataset** y nos permite cargar un directorio de imágenes. Esta funcionalidad trabaja con la segunda, que es **transforms**. Con **transforms** podemos implementar cambios en las imágenes con el fin de estandarizarlas y algunos otros cambios que pueden resultar útiles.

Cabe mencionar que, si las imágenes son en escala de grises, tendrán 1 canal, mientras que, si son a color, tendrán 3 canales (RGB). Esto será importante al momento que tengamos que considerar las dimensiones que se esperan de nuestras imágenes. En nuestro caso, estamos utilizando imágenes a color. Un ejemplo del número de canales en una imagen se muestra en la figura 8.96.

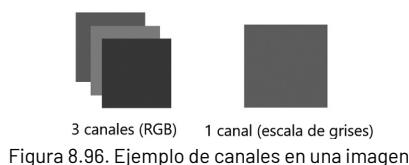


Figura 8.96. Ejemplo de canales en una imagen

Para este ejercicio en particular, es importante realizar un cierto preprocesamiento. En esta práctica, se aplicará a las imágenes lo siguiente:

- **RandomResizedCrop**. Se utiliza para que las imágenes tengan el mismo tamaño —recibe como parámetro el tamaño deseado en píxeles—.
- **ToTensor**. Convierte las imágenes en tensores —que son la base de PyTorch—.
- **Normalize**. Sirve para normalizar los tensores. En nuestro caso, hemos agregado dos métodos más:
- **RandomRotation**. Gira las imágenes x grados —recibe como parámetro el número de grados a girar—, y:
- **RandomHorizontalFlip**. Aleatoriamente, voltear una imagen horizontalmente. Estos dos últimos métodos no son estrictamente necesarios que se apliquen, pero es una buena práctica para que nuestra red pueda reconocer las imágenes en cualquier posición u orientación.

Ahora, los tensores, mencionados anteriormente, son la base de PyTorch, todas las operaciones que realiza PyTorch son hechas con tensores. Son muy parecidos a los *array* de *numpy*, tanto así que se pueden transformar *arrays* a *tensores* de una manera sumamente fácil.

Código:

```
# Ejemplo de un tensor
torch.tensor([1, 2, 3, 3])

tensor([1, 2, 3, 3])
```

Cargamos la ubicación del directorio que contiene las imágenes, dentro de este directorio existen dos más, en uno están las imágenes de gatos y en el otro hay imágenes diferentes, las cuales se encuentran en el directorio «random». Todas estas imágenes se van a cargar en lo que será nuestra base de datos. Se hará un preetiquetado de los datos, dependiendo de en qué carpeta se encuentren, será 0 para el directorio de gatos y 1 para el directorio «random». Este procedimiento se muestra en el siguiente código:

Código:

```
# Directorio de las imágenes
dataset_dir = "Gatos"

# Ajustamos las imágenes
transformadas = transforms.Compose([transforms.RandomRotation(30),
transforms.RandomResizedCrop(224),
transforms.RandomHorizontalFlip(),
transforms.ToTensor(),
transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])])

# Cargamos el dataset de imágenes
dataset = datasets.ImageFolder(dataset_dir, transform=transformadas)
dataset

Dataset ImageFolder
Number of datapoints: 2000
Root location: Gatos
StandardTransform
Transform: Compose(
    RandomRotation(degrees=[-30.0, 30.0], interpolation=nearest, expand=False, fill=0)
    RandomResizedCrop(size=(224, 224), scale=(0.08, 1.0), ratio=(0.75, 1.3333), interpolation=bilinear)
    RandomHorizontalFlip(p=0.5)
    ToTensor()
    Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
)
```

Podemos ver que nuestra base de datos contiene dos mil imágenes, mil de gatos y mil en el directorio «random». Ya que tenemos nuestra base de datos, crearemos tres conjuntos: uno de entrenamiento, uno de validación y uno de prueba. En este caso en particular, se crearán los conjuntos de tal modo que el 30 % serán datos de prueba y del restante 70%, el 30 % serán datos de validación y los demás serán los datos de entrenamiento, como se muestra en el siguiente código:

Código:

```
# Dividimos el dataset en conjuntos de entrenamiento, validación y prueba
```

```

# Tamaño del dataset
len_dataset = len(dataset)
# definimos el tamaño de los datos de entrenamiento (70 % del dataset)
len_entre = int(len_dataset * 0.7)

# Definimos el tamaño de los datos de prueba
len_prueba = int((len_dataset - len_entre)/2)

# Tamaño del set de validación
len_val = len_dataset - len_entre - len_prueba

# Creamos los conjuntos de entrenamiento y prueba
entrenamiento, validacion, prueba = random_split(dataset, (len_entre, len_val, len_prueba))

```

Lo siguiente es usar un tipo de objeto llamado *data loader* que nos permitirá dividir los datos en lotes o *batches*, el cual nos permitirá cargar la información más rápidamente. Para el conjunto de entrenamiento, crearemos *batches* de 64, mientras que, para los conjuntos de prueba y validación, no crearemos *batches* – que es lo mismo que tener *batches* de 1–. Por último, mandaremos *shuffle* como verdadero para que ordene de manera aleatoria los datos de cada conjunto, como se muestra en el siguiente código:

Código:

```

# Definimos el dataloader con los datos de entrenamiento
loader_entrenamiento = torch.utils.data.DataLoader(entrenamiento, batch_size=64, shuffle=True)
# Definimos el dataloader con los datos de validación
loader_validacion = torch.utils.data.DataLoader(validacion, batch_size=1, shuffle=True)
# Definimos el dataloader con los datos de prueba
loader_prueba = torch.utils.data.DataLoader(prueba, batch_size=1, shuffle=True)

```

Una vez terminado el preprocessamiento, se comienza con la construcción, en este caso, de un modelo preentrenado llamado **AlexNet**.

PyTorch nos permite crear redes personalizadas –con las capas que queramos–, gracias a su clase **Module**. **Module** nos permite heredar todos los métodos que contienen las redes neuronales de PyTorch. Así que, para crear una red propia, basta con crear una clase hija de **Module**. En nuestro caso, crearemos la clase **CNN**.

Para que nuestra clase funcione adecuadamente, se necesitan implementar al menos dos métodos: el primero, el «**constructor**», donde se crea la arquitectura de la red, definiendo las capas del modelo, el segundo, el método *forward*, que es donde se define cómo debe calcularse la entrada que se le dé al modelo.

Para declarar las capas de nuestro modelo, se pueden declarar cada una como un atributo del modelo o se puede utilizar la función **Sequential**, que nos permite agrupar las capas del modelo, aunque hacer esto no es imprescindible, nos sirve para que, visiblemente, sea más ordenado.

Para construir nuestra red, nos vamos a basar en la arquitectura de la red **AlexNet**, la cual es muy utilizada para procesamiento de imágenes. Podemos ver desde *torchvision* la estructura de esta red.

Código:

```

models.alexnet()

AlexNet(
(features): Sequential(
(0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))

```

```

(1): ReLU(inplace=True)
(2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
(3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
(4): ReLU(inplace=True)
(5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
(6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(7): ReLU(inplace=True)
(8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(9): ReLU(inplace=True)
(10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(11): ReLU(inplace=True)
(12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
)
(avgpool): AdaptiveAvgPool2d(output_size=(6, 6))
(classifier): Sequential(
(0): Dropout(p=0.5, inplace=False)
(1): Linear(in_features=9216, out_features=4096, bias=True)
(2): ReLU(inplace=True)
(3): Dropout(p=0.5, inplace=False)
(4): Linear(in_features=4096, out_features=4096, bias=True)
(5): ReLU(inplace=True)
(6): Linear(in_features=4096, out_features=1000, bias=True)
)
)

```

Para poder implementar prácticamente cualquier algoritmo de inteligencia artificial, es necesario entender el modelo. En este caso, **AlexNet** cuenta con lo siguiente:

Vemos que está dividido en dos partes, *features* y *classifier*. *features* está compuesta de capas convolucionales, ReLU y *max pooling*, mientras que *classifier*, está compuesta por capas lineales, ReLU, y contiene lo que se conoce como *dropouts*.

La capa convolucional (**Conv2d**) recibe varios parámetros, el primero es el número de canales de entrada, como estamos utilizando imágenes a color, este valor será 3, luego son los canales que queremos de salida. Para este caso son 64. Después viene el *kernel_size* o tamaño del filtro. Este filtro es el que se encargará de recorrer toda la imagen con el único propósito de recolectar la información más importante de la imagen. Seguido, tenemos el *stride* o zancada, que es el número de píxeles que avanza el filtro cada que se mueve, y, por último, tenemos el *padding* o relleno, que es agregar información en los bordes de la imagen.

Para ejemplificar mejor estos términos, ampliaré un poco la información que se mostró en el capítulo 6 de redes neuronales.

Con lo que respecta al *kernel*, podemos definirlo en redes convolutivas como el filtro que irá recorriendo la imagen para aplicar la convolución y obtener información de los píxeles que abarca. No confundir este término de **kernel** con, por ejemplo, una máquina de soporte de vectores, donde el *kernel* es la función que se le va a aplicar a los datos que atraviesan el máximo hiperplano. Un *kernel* de CNN de 4x4 será un filtro de 16 píxeles, mientras que uno de 3x3 será un filtro de 9 píxeles, como se muestra en la figura 8.97.

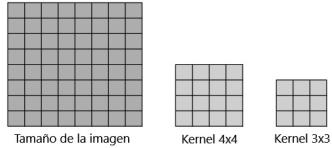


Figura 8.97. Ejemplos de kernel para una red profunda convolutiva

El **stride** o zancada se refiere a cuántos píxeles avanzará el **kernel** cada vez, tanto horizontalmente como verticalmente. Un **stride** pequeño tardará más en recorrer la imagen y recabará más información, mientras que uno grande recorrerá la imagen en poco tiempo, pero recabarán menos información.

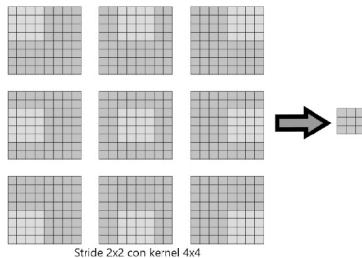


Figura 8.98. Ejemplo de una zancada 2x2 para un kernel 4x4 en una red profunda convolutiva

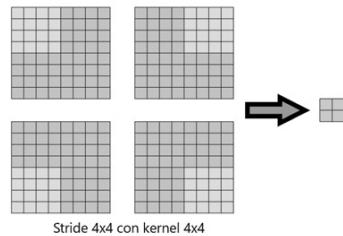


Figura 8.99. Ejemplo de una zancada 4x4 para un kernel 4x4 en una red profunda convolutiva

Las figuras 8.98 y 8.99 muestran una imagen ejemplo de 8x8 píxeles, ambas con un **kernel** de 4x4. En la figura 8.98, se observa que, al aplicar una zancada de 2x2, se condensa la información al final de proceso en 9 píxeles. Por otro lado, en el ejemplo de la figura 8.99, se tiene la misma imagen con el mismo **kernel**, pero con zancada de 4x4. En este caso, observamos que se ha condensado la información en 4 píxeles.

Otro de los conceptos que conviene explicar es el de **padding**. El **padding** es la cantidad de píxeles extra que se le agregan a la imagen original, normalmente, se rellena con 0, aunque se pueden poner valores personalizados. El **padding** se utiliza para que el **kernel** pueda obtener más información de la imagen al tener una superficie mayor por la cual moverse. Un ejemplo de **padding** 1x1 y **padding** 2x2 se muestra en la figura 8.100.

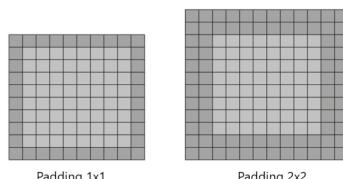


Figura 8.100. Ejemplo de un padding de 1x1 y 2x2

Por su parte, el término **MaxPooling** hace referencia a una disminución de la imagen, reduciendo los píxeles de esta. Esto lo logra por la operación que emplea, misma que permite que nos quedemos con la información más importante y se deseche la demás. Además, esta operación se realiza utilizando un filtro con dilatación, el cual es un filtro con espacios entre sus píxeles –llamado usualmente **dilation**–. Las figuras 8.101 y 8.102 muestran un ejemplo de **MaxPooling** y **dilation**, respectivamente.

2	4	8	5
9	5	2	3
6	1	6	2
7	3	4	0



9	8
7	6

Figura 8.101. Ejemplo de MaxPooling para una CNN

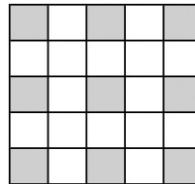


Figura 8.102. Ejemplo de dilatación = 1 para una CNN

Un *kernel* con dilatación es un filtro de tamaño $n \times m$ que deja cierto espacio entre los píxeles, de modo que no todo lo que esté dentro del *kernel* se tomará como información, solo los píxeles seleccionados, como se mostró en la figura 8.102.

Algo que es fundamental cuando se pasa de una capa hacia otra es que los canales de salida de la primera deben ser los canales de entrada de la segunda, de otro modo, nuestra red no funcionará.

Durante la primera etapa vemos que tenemos cinco capas convolucionales y, después, tenemos un *adaptive average pooling*, que, a diferencia del *maxpooling* donde configuramos el *kernel* y los demás hiperparámetros, aquí solamente especificamos el tamaño de la salida que queremos y se asignan los hiperparámetros de manera automática.

En la segunda parte, tenemos lo que es el clasificador –perceptrón multicapa–, lo que nos dirá a qué clase pertenecen nuestras imágenes. El clasificador contiene algunos *dropouts*, que eliminan algunos datos de forma aleatoria, poniéndolos en 0, con el fin de evitar el sobreentrenamiento. A los *dropouts* se les manda la probabilidad con la que eliminarán información, en un rango de 0 a 1.

En el modelo original de **AlexNet**, se tienen como salida mil neuronas, utilizadas para clasificar imágenes en mil clases distintas, dado que nosotros solo tenemos dos clases distintas, modificaremos la salida a dos neuronas. También modificaremos un poco la estructura del clasificador, en nuestro caso, solo tendremos una capa lineal, una ReLU, un *dropout* de 0.5 y, por último, una capa lineal. Esto se muestra en el siguiente código:

Código:

```
# Creamos nuestro modelo personalizado
class CNN(torch.nn.Module):

    # Constructor
    def __init__(self):
        super().__init__()
        # Parte convolutiva
        self.convolutiva = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2)),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False),
            nn.Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2)),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False),
            nn.Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)),
            nn.ReLU(inplace=True),
```

```

nn.Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)),
nn.ReLU(inplace=True),
nn.Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)),
nn.ReLU(inplace=True),
nn.MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False),

nn.AdaptiveAvgPool2d(output_size=(6, 6))
)

# Clasificador
self.clasificador = nn.Sequential(
nn.Linear(in_features=9216, out_features=4096, bias=True),
nn.ReLU(inplace=True),
nn.Dropout(p=0.5, inplace=False),
nn.Linear(in_features=4096, out_features=2, bias=True)
)

def forward(self, x):
x = self.convolutiva(x)
x = torch.flatten(x, start_dim=1, end_dim=-1)
x = self.clasificador(x)
return x

```

Finalmente, en la función *forward*, enviamos las imágenes primero a las capas convolucionales y después al clasificador o capas totalmente conectadas, pero antes de enviarlas, aquí es importante que «aplanemos» los tensores. Esto significa que se tenga una sola dimensión, de otra forma, no se podrá realizar la clasificación.

Ya creada nuestra clase, vamos a instanciarla, pero antes vamos a utilizar una funcionalidad más de PyTorch. Para esto, vamos a emplear la gpu para realizar los cálculos de manera más rápida, si es que tenemos alguna disponible, usaremos **torch.device** para seleccionar un dispositivo, ya sea una gpu —si hay una disponible— o la cpu —en caso de que no haya cpu—, y definimos el criterio de la función de pérdida, también elegiremos el algoritmo optimizador de la red, nosotros escogeremos *CrossEntropyLoss* como función de pérdida y *SGD*(gradiente descendente estocástico).

Nuestro optimizador recibe como parámetros los parámetros de nuestra red (**cnn.parameters**) para ir actualizando los pesos de las capas, también recibe la tasa de aprendizaje, (*learning rate*), como se muestra en el siguiente código:

Código:

```

# Usamos una gpu si es que hay alguna disponible
disp = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Instanciamos nuestra
cnn = CNN().to(disp)
# Función de pérdida
# criterio = nn.NLLLoss()
criterio = nn.CrossEntropyLoss()
# Optimizador
optimizador = torch.optim.Adam(cnn.parameters(), lr=0.00001)

```

Ahora, crearemos una función para evaluar la exactitud del modelo. Esta función nos servirá después en la sección de entrenamiento.

Definimos nuestra función **evaluar** que recibe como parámetros la red (**modelo**), un objeto tipo *data loader* (**prueba**), el dispositivo donde se trabajará (**disp**) y tiene un parámetro opcional que es **mostrar**, por defecto, mostrar es «False», si lo cambiamos a «True», nos imprimirá la exactitud al término de la función, si no, nos retornará la exactitud.

Empezamos la función poniendo en modelo en modo de evaluación. Después inicializamos dos variables: **correctas** y **total**, en las que llevaremos el conteo de las predicciones correctas y el total de datos, respectivamente. Después vamos obteniendo las imágenes y sus etiquetas iterando el *data loader*, copiamos esos datos al dispositivo elegido previamente y ejecutamos el modelo con estas entradas. Posteriormente, se obtiene la clase a la que pertenecen y vamos sumando el total de datos y el total de datos correctos. Finalmente, calculamos la exactitud de nuestro modelo. Para obtener la pérdida hacemos algo similar, es decir, vamos obteniendo la pérdida de cada imagen, las sumamos y calculamos la media. Esto se muestra en el siguiente código:

Código:

```
def evaluar(modelo, prueba, disp, criterio, mostrar=False):

    # Ponemos el modelo en modo de evaluación
    modelo.eval()

    # Inicializamos variables
    correctas = 0
    total = 0
    perdida_val = 0.0

    # Obtenemos los datos del data loader
    for imagenes, etiquetas in iter(prueba):
        # Copiamos los datos al dispositivo seleccionado (gpu o cpu)
        imagenes = imagenes.to(disp)
        etiquetas = etiquetas.to(disp)

        # Obtenemos las salidas del modelo
        salidas = modelo(imagenes)

        # Obtenemos las predicciones seleccionando la clase con la
        # mayor probabilidad
        predichos = torch.max(salidas.data, 1)[1]

        # Acumulamos el número de datos
        total += len(etiquetas)

        # Pérdida
        perdida_val += criterio(salidas, etiquetas).data
        # Acumulamos las predicciones correctas
        correctas += (predichos == etiquetas).sum()

    perdida_val = float(perdida_val/float(total))
    perdida_val = round(perdida_val, 4)
    # Calculamos la exactitud
    exactitud = 100 * correctas / float(total)
```

```

exactitud = round(float(exactitud), 4)

if mostrar:
    print("Exactitud: {}%".format(exactitud))
else:
    return exactitud, perdida_val

```

El siguiente paso es definir la función de entrenamiento del modelo. Esta función va a entrenar el modelo con los datos de entrenamiento y va a evaluar la exactitud con los datos de validación para evitar tener resultados sesgados.

Los parámetros de **entrenar_modelo** son los siguientes:

- **Modelo.**
- **Entrenamiento.** Es el set de entrenamiento.
- **Validacion.** Es el set de validación.
- **Criterio.** Es el criterio utilizado.
- **Optim.** Es el optimizador utilizado.
- **Disp.** Es el dispositivo utilizado.
- **ex_d.** Define la exactitud deseada. Por defecto es 100. Si se deja así, el entrenamiento terminará hasta finalizar todas las épocas.
- **Epochs.** Recibe las épocas. Por defecto, son 100, aunque puede modificarse.
- **Mostrar.** Indica cada cuántas épocas muestra el progreso.

Comenzamos inicializando dos listas vacías donde se irán guardando los valores de pérdida y exactitud para graficarlos posteriormente. También, inicializamos **ex_ant** que guardará la exactitud anterior para realizar la comparación entre la exactitud anterior y la actual. Utilizamos un ciclo **for** para recorrer las épocas, este **for** se detendrá cuando haya alcanzado todas las épocas o cuando se haya alcanzado la exactitud deseada. Lo siguiente es poner el modelo en modo de entrenamiento y vamos obteniendo las imágenes y sus etiquetas. Copiamos esos datos al dispositivo seleccionado, establecemos el gradiente en cero para que no se vayan acumulando valores que nos pueden llevar a tener ajustes de manera errónea, mandamos las imágenes a nuestro modelo y guardamos las salidas que nos devuelve, después calculamos la pérdida y actualizamos los pesos de las capas.

Lo siguiente es evaluar la exactitud del modelo con el conjunto de validación. Para realizar esto, llamamos a nuestra función **evaluar** y guardamos la pérdida y la exactitud en las listas que creamos al principio de la función. Lo siguiente es evaluar la condición de paro así como tomar la decisión si ya es momento de mostrar el progreso o aún no. Finalmente, convertimos a **arrays** las listas y las devolvemos: como se muestra en el siguiente código.

Código:

```

def entrenar_modelo(modelo, entrenamiento, validacion, criterio, optim, disp, ex_d=100, epochas=100,
mostrar=20):

    # Listas donde guardaremos los datos para graficar
    l_perdida = []
    l_exact = []
    l_perd_val = []

    # Vamos recorriendo las épocas una a una

```

```

for epoca in range(epocas + 1):

    perdida_data = 0.0
    total = 0

    # Configuramos el modelo en modo de entrenamiento
    modelo.train()

    # Obtenemos las imágenes y las etiquetas
    for imagenes, etiquetas in iter(entrenamiento):
        # Copiamos los datos al dispositivo seleccionado
        imagenes = imagenes.to(disp)
        etiquetas = etiquetas.to(disp)

        # Establecemos el gradiente en 0
        optim.zero_grad()

        # Obtenemos la salida del modelo
        salidas = modelo(imagenes)

        # Acumulamos el número de datos
        total += len(etiquetas)

        # Calculamos la pérdida (valor esperado - valor obtenido)
        perdida = criterio(salidas, etiquetas)
        perdida_data += perdida.data

        # Calculamos el gradiente usando la pérdida estimada
        perdida.backward()

        # Actualizamos los pesos
        optim.step()

        # Calculamos la pérdida
        perdida_data = float(perdida_data/float(total))
        perdida_data = round(perdida_data, 4)

    # Calculamos la exactitud y la pérdida de validación
    exact, perd_val = evaluar(modelo, validacion, disp, criterio)

    # Guardamos los datos para graficar posteriormente
    l_perdida.append(perdida_data)
    l_exact.append(exact)
    l_perd_val.append(perd_val)

    # Si se llega a la pérdida aceptada y se pierde exactitud,
    # se detiene el entrenamiento
    if ex_d <= exact:
        # Imprimimos el progreso
        print("Época: {} \t Pérdida ent: {} \t Pérdida val: {} \t Exactitud val: {}".format(
            epoca, perdida_data, perd_val, exact
        ))
        break

```

```

# Mostramos el progreso de acuerdo a 'mostrar'
if epoca % mostrar == 0:
    # Imprimimos el progreso
    print("Época: {} \t Pérdida ent: {} \t Pérdida val: {} \t Exactitud val: {}".format(
        epoca, perdida_data, perd_val, exact
    ))

# Convertimos las listas a arrays de numpy
l_perdida = np.asarray(l_perdida)
l_exact = np.asarray(l_exact)
l_perd_val = np.asarray(l_perd_val)
l_epoca = np.arange(epoca + 1)

return l_perdida, l_perd_val, l_exact, l_epoca

```

Llamamos a la función de entrenamiento con el siguiente código:

Código:

```

perdida, perd_val, exactitud, epoca = entrenar_modelo(
    cnn, loader_entrenamiento, loader_validacion, criterio,
    optimizador, disp, ex_d=80, mostrar=10
)

```

Esto puede tardar bastante. Hay que tomar en cuenta que son muchas capas, neuronas, hiperparámetros, cálculos e imágenes para poder realizar la clasificación.

Graficamos la función de pérdida del entrenamiento y la exactitud que se obtiene con el set de validación. Esto se muestra en el siguiente código:

Código:

```

# Escalamos la pérdida de validación
perd_val1 = perd_val * (perdida.mean() / perd_val.mean())
perd_val1

# Función de pérdida
plt.plot(epoca, perdida, color='r')
plt.plot(epoca, perd_val1, color='b')
plt.xlabel("Época")
plt.ylabel("Pérdida")
plt.title("Pérdida")
plt.legend(["Entrenamiento", "Validación"])
plt.show()

# Exactitud
plt.plot(epoca, exactitud, color='b')
plt.xlabel("Época")
plt.ylabel("Exactitud")
plt.title("Exactitud")

```

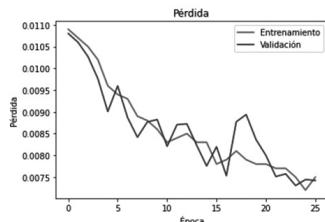


Figura 8.103. Pérdida de entrenamiento y validación para una CNN

Como puede observarse en la figura 8.103, la pérdida tanto del entrenamiento como de la validación van consistentemente decreciendo hasta la época 25 cuando terminó el entrenamiento, debido a que la red CNN llegó a la exactitud requerida —en este caso, mayor a 0.8 u 80 %—.

Como se comentó en la sección 5.10, se busca una buena generalización, y, en este caso, aunque la diferencia entre ambas pérdidas no es la misma, sigue decreciendo a la par. Hay que cuidar que la pérdida de la validación no incremente, mientras la pérdida del entrenamiento siga bajando. Esto implicaría que el algoritmo ya no está generalizando bien y, posiblemente, sobreaprendiendo.

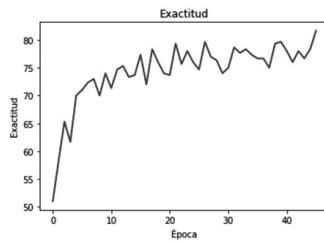


Figura 8.104. Época vs exactitud para una CNN para clasificación de imágenes

Como puede verse en la figura 8.104, la exactitud (accuracy) está continuamente mejorando hasta que llega al punto en el cual se decidió que podría detenerse, en este caso, en 82.66 %.

Probablemente, no sea el mejor resultado que se pueda obtener, pues que casi el 20 % de las imágenes que no son gatos las confunda con gatos y viceversa, no parece un buen resultado de exactitud. Hay que tomar en cuenta que podemos modificar el umbral, aunque tardaría mucho más y llegaría a un límite.

Ya hemos creado nuestra red neuronal convolucional, ahora lo único que nos falta es poder ver las imágenes y ver en qué clase las ha puesto nuestra red.

Vamos a agregarle un límite de imágenes para que nos imprima, a este límite lo llamaremos **k**.

Como al principio las hemos convertido a tensores y las hemos normalizado, ahora es necesario revertir esa transformación para poderlas ver con **matplotlib**.

Vamos a crear una función para ilustrar este proceso. La función la llamaremos **mostrar_imagen**, y recibirá como parámetro un *data loader*, dentro de esta función, estableceremos dos tensores, uno que corresponde a la media, **mean** y otro que corresponde a la desviación estándar, **std**. Les asignaremos los mismos valores que les asignamos a los utilizados para normalizar las imágenes.

Después, obtendremos las imágenes una a una y las multiplicaremos por **std** y les sumaremos **mean**. Con esto, ya hemos revertido la normalización. Posteriormente, se convierten a un *array* de *numpy*, transponerlas y graficarlas con **imshow** de *pyplot*, como se muestra en el siguiente código:

Código:

```
evaluar(cnn, loader_prueba, disp, criterio, mostrar=True)
def mostrar_imagen(loader, k=-1):

    mean = torch.tensor([0.485, 0.456, 0.406])
    std = torch.tensor([0.229, 0.224, 0.225])

    # Contador
    i = 0

    for imagenes, etiquetas in iter(loader):
        for imagen in imagenes:
            # Incrementamos el contador
            i += 1
            # Condición de paro
            if k > 0 and k < i:
                break
            # Revertimos la normalización de la imagen
            img = imagen * std[:,None,None] + mean[:,None,None]
            plt.imshow(img.numpy().transpose(1, 2, 0))
            plt.show()
            # Condición de paro
            if k > 0 and k < i:
                break
    mostrar_imagen(loader_prueba, k=5)
```

Como puede observarse, la red obtuvo una clasificación del 82.66 %. En las imágenes, se muestran algunas figuras que la mente humana no confundiría con un gato. Sin embargo, en una primera instancia, la red demostró que puede identificarlas con un buen margen de exactitud.

Posteriormente, mostramos la exactitud máxima lograda y mostramos las imágenes con su clasificación. Ahora, vamos a combinar con la función **evaluar** para poder acceder a las predicciones de estas.

Le daremos dos parámetros más a nuestra función **evaluar**, que serán **predecir**, por defecto, en «False», cuando lo mandemos como «True», nos imprimirá las imágenes junto con su etiqueta, y **k**, que es nuestro contador, por defecto en -1. Para ver su clase, utilizaremos el diccionario **clases**, que creamos previamente. De modo que, dependiendo de si la clase es 1 o 0, podamos ver si se trata de un gato o no, como se muestra en el siguiente código:

Código:

```
def evaluar(modelo, prueba, disp, criterio, mostrar=False, predecir=False, k=-1):
```

```

# Ponemos el modelo en modo de evaluación
modelo.eval()

# Inicializamos variables
correctas = 0
total = 0
perdida_val = 0.0

# Inicializamos nuestro contador
j = 0

# Obtenemos los datos del data loader
for imagenes, etiquetas in iter(prueba):
    # Copiamos los datos al dispositivo seleccionado (gpu o cpu)
    imagenes = imagenes.to(disp)
    etiquetas = etiquetas.to(disp)

    # Obtenemos las salidas del modelo
    salidas = modelo(imagenes)

    # Obtenemos las predicciones seleccionando la clase con la
    # mayor probabilidad
    predichos = torch.max(salidas.data, 1)[1]

    # Acumulamos el número de datos
    total += len(etiquetas)

    # Pérdida
    perdida_val += criterio(salidas, etiquetas).data

    # Acumulamos las predicciones correctas
    correctas += (predichos == etiquetas).sum()

    # Implementamos el método para ver las imágenes
    if predecir:
        if j < k or k == -1:
            # Copiamos las imágenes a la cpu
            imagenes = imagenes.to('cpu')
            predichos = predichos.to('cpu')
            predichos = predichos.numpy()
            # Índice de la predicción
            i = 0
            mean = torch.tensor([0.485, 0.456, 0.406])
            std = torch.tensor([0.229, 0.224, 0.225])
            for imagen in imagenes:
                # Incrementamos el contador
                if k > 0:
                    j += 1
                # Desnormalizamos la imagen
                img = imagen * std[:,None,None] + mean[:,None,None]
                plt.imshow(img.numpy().transpose(1, 2, 0))
                # Obtenemos la predicción de la imagen
                prediccion = clases.get(predichos[i])
                # Ponemos la predicción como título

```

```

plt.title(prediccion)
i += 1
plt.show()

perdida_val = float(perdida_val/float(total))
perdida_val = round(perdida_val, 4)
# Calculamos la exactitud
exactitud = 100 * correctas / float(total)
exactitud = round(float(exactitud), 4)
if mostrar:
    print("Exactitud: {}%".format(exactitud))
else:
    return exactitud, perdida_val

```

Ejecutamos **evaluar** con **predecir = True** y **k = 10**, para ver diez imágenes y guardamos el modelo generado, como se muestra en el siguiente código:

Código:

```

evaluar(cnn, loader_prueba, disp, criterio, mostrar=True, predecir=True, k=10)

# Guardar el modelo utilizando torch.save
# Se guardará en un archivo llamado cnn_checkpoint.pth
# en el mismo directorio
torch.save({
    'model_state_dict': cnn.state_dict(),
    'optimizer_state_dict': optimizador.state_dict(),
}, "cnn_checkpoint.pth")

```

Si se requiere cargar el modelo guardado para no volver a reentrenar todo de nuevo, se utiliza el siguiente código:

Código:

```

# Usamos una gpu si es que hay alguna disponible
disp = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Instanciamos la clase
cnn = CNN().to(disp)
# Función de pérdida
criterio = nn.CrossEntropyLoss()
# Optimizador
optimizador = torch.optim.SGD(cnn.parameters(), lr=0.01)

# Cargamos los valores previamente guardados
checkpoint = torch.load("cnn_checkpoint.pth")
cnn.load_state_dict(checkpoint['model_state_dict'])
optimizador.load_state_dict(checkpoint['optimizer_state_dict'])

```

Ahora que ya tenemos un clasificador de imágenes puedes hacer mejoras a este ejercicio. Por ejemplo, intenta mejorar la exactitud modificando los hiperparámetros como se muestra en los capítulos 6 y 7. Cada que cambies un hiperparámetro, observa el efecto que este tiene sobre la clasificación.

También puedes utilizar otras imágenes para ver qué tan bien funciona tu clasificador o puedes mejorar el tiempo de ejecución del algoritmo cambiando el número de capas, el *pooling* y el optimizador.

Otro de los ejercicios que puedes hacer es modificar la topología de la red. Intenta con GoogleNet, Resnet o alguna otra red convolutiva.

Por último, modifica tu código para que pueda clasificar más de dos diferentes objetos. Es decir, que el número de clases sea mayor a dos.

8.21. Red neuronal recurrente (RNN)

Las redes neuronales recurrentes (RNN, por sus siglas en inglés) son muy útiles cuando se trata de analizar datos en el tiempo. Existen varios tipos de redes recurrentes, como son GRU, ELMAN, LSTM, etc.

Durante este ejemplo, realizaremos una red LSTM (*long short term memory*), para datos en dominio de tiempo altamente no lineales y predecir su comportamiento.

Utilizaremos un conjunto de datos sobre el registro de concentraciones de partículas PM10, en la Ciudad de México del 2018 (apéndice A.7), mismos que pueden ser descargados en: http://www.amese.net/libro_ia/datos_pm10.csv

El código se muestra en esta sección y puede ser descargado en: http://www.amese.net/libro_ia/Prog/8.21_RNN.ipynb

Importamos las librerías, como se muestra en el siguiente código:

Código:

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
from numpy.polynomial import Polynomial as poly
```

Posteriormente, se leen los datos con los que vamos a trabajar y te toma un atributo, es decir, una estación solamente (BJU) para poder trabajar con la red recurrente, como se muestra en el siguiente código:

Código:

```
df = pd.read_csv("2018PM10.csv")
df = df[["BJU"]]
df.head()
```

Posteriormente, seleccionamos los primeros dos mil datos, que son los que estaremos trabajando. Dentro de los registros hay horas donde no se hubo registro, estos datos están denotados como -99 y tendremos que imputarlos. Las diferentes técnicas de imputación se muestran en la sección 4.5 y un ejemplo de una imputación múltiple llamada MICE se muestra en la sección 8.10. El siguiente código muestra la selección de dos mil datos de ejemplo y la imputación:

Código:

```
bju = df.iloc[0:2000]
bju
bju = list(bju["BJU"])
def imputar(valor, lista, lim_inf=3, lim_sup=3, grado=3):
    # Mientras haya datos nulos en el dataset se ejecuta el ciclo
```

```

while valor in lista:
    # Guardamos el valor de lim_sup + 1 en m
    m = lim_sup + 1
    # Índices de los datos nulos
    i = []
    # Índices de los datos no nulos para realizar el polinomio
    indices = []
    # Datos no nulos
    datos = []

    # Se agrega el primer valor nulo
    i.append(lista.index(valor))

    # Se toman los datos anteriores
    for j in range(i[0]-lim_inf, i[0]):
        indices.append(j)
        datos.append(lista[j])

    # Se inicializan variables
    # k funciona como bandera
    # l es el avance para el rango inferior
    k = 0
    l = 1

    # Se ejecuta el ciclo hasta que se encuentren el número
    # de valores consecutivos no nulos solicitados
    while k == 0:
        k = 1
        for j in range(i[0]+l, i[0]+m):
            if lista[j] == valor:
                k = 0
                l += 1
                m += 1
                break

    # Se guardan los índices de los valores nulos así como
    # los valores no nulos para realizar el polinomio
    for j in range(i[0]+l, i[0]+m):
        if lista[j] == valor:
            m += 1
            i.append(j)

    else:
        indices.append(j)
        datos.append(lista[j])

    # Se realiza la regresión polinomial para imputar los
    # datos nulos
    f = poly.fit(indices, datos, grado)

    # Se sustituyen los datos nulos

```

```

for dato in i:
    lista[dato] = int(f(dato))
imputar(-99, bju)

```

Como se muestra en el código anterior, la imputación que se realizó fue una imputación por regresión con un polinomio de grado tres. Esto se realizó por simplicidad, como ejercicio, se puede combinar la imputación MICE (sección 8.10) con la red recurrente y observar el efecto que tienen diferentes imputaciones en el resultado final. En este caso, tomaremos los tres datos anteriores al dato nulo y los tres siguientes para crear un polinomio que se ajuste e imputar el dato nulo.

Para esto, crearemos una función llamada **imputar** que recibirá como parámetros el valor que es considerado nulo (**valor**), la lista donde se encuentran los datos (**lista**), el número de datos anteriores a tomar (**lim_inf**), el número de datos posteriores a tomar (**lim_sup**) y un parámetro opcional que es el grado del polinomio (**grado**) por defecto 3. Dentro de esta función, ejecutaremos un ciclo *while* que se detendrá hasta que ya no exista ningún valor nulo. Dentro de este ciclo, primero tomamos los primeros tres valores no nulos y guardamos su índice y su valor.

Después, comenzamos el proceso para tomar los tres valores siguientes no nulos, como estamos buscando los tres siguientes valores consecutivos, realizamos un ciclo *for* que nos dirá qué tan lejos se encuentran esos valores de la posición del valor nulo en turno, esto lo realizamos porque es posible que haya más de un valor nulo en el rango que pedimos de valores posteriores. De ser así, se agrega el índice de este valor nulo encontrado a la lista y se buscan los valores no nulos en una posición más adelante. Hecho este proceso, se realiza la regresión polinomial y se sustituyen los valores nulos.

Ahora vamos a dividir nuestros datos en tres conjuntos: entrenamiento, validación y prueba. El conjunto de entrenamiento abarca el 70 % de los datos mientras que los conjuntos de validación y prueba abarcan el 15 % y 15 % restantes. Es importante mantener el orden de los datos durante este proceso, ya que son datos que dependen del tiempo. El código se muestra a continuación:

Código:

```

# Dividimos los datos en conjuntos de entrenamiento, validación y prueba

# Total de datos
len_ds = len(bju)
# Definimos el tamaño de los datos de entrenamiento (70 % de los datos)
len_entre = int(len_ds * 0.7)
# Definimos el tamaño de los datos de prueba y de validación (15 % y 15 %)
len_val = int((len_ds - len_entre)/2)
# Creamos los conjuntos de entrenamiento, validación y prueba
entre, val, prueba = bju[:len_entre], bju[len_entre:len_val+len_entre], bju[len_val+len_entre:]

```

Vamos a hacer uso de las funcionalidades de torch para etiquetar nuestros datos de una forma más rápida. Primero, tenemos que saber qué entrada va a recibir nuestra red neuronal, para este caso, la entrada será un vector de longitud n y la etiqueta corresponderá al valor que se quiere predecir, es decir, el valor siguiente (la entrada $n+1$).

Creamos una clase hija **RNNData** para construir objetos tipo *dataset* de acuerdo a nuestras necesidades. Para que nuestra clase funcione adecuadamente, necesita al menos los siguientes métodos: **__init__**, **__len__** y **__getitem__**. El método **__len__** nos retorna la cantidad de datos dentro del *dataset* y **__getitem__** es el método por el que podemos acceder a los datos dentro del *dataset*.

Esta clase recibirá la lista de datos **data** y la secuencia o número de elementos que conforman el vector de entrada **seq**. Dentro del constructor, convertiremos nuestros datos a tensores y guardaremos los valores máximo y mínimo para después normalizar los datos mediante el método **min-max**, también guardaremos el parámetro **seq**. En **__len__** retornaremos el número de datos que nos retornará el *dataset*, en este caso, es la cantidad de datos menos **seq**. Y en **__getitem__** se retorna la secuencia de valores (vector de entrada) y la etiqueta (salida).

Código:

```
class RNNDATA(Dataset):
    def __init__(self, data, seq):
        # Convertimos los datos a tensores
        self.data = torch.tensor(data)
        self.max = self.data.max()
        self.min = torch.min(self.data)
        # Normalizamos
        self.data = (self.data - self.min)/(self.max - self.min)
        # Cantidad de valores anteriores
        self.seq = seq

    def __len__(self):
        # Retornamos el número de datos en el dataset
        return len(self.data) - self.seq

    def __getitem__(self, index):
        # Se obtiene la secuencia
        sequence = self.data[index:index + self.seq]
        # Se obtiene el valor siguiente
        valor = self.data[index + self.seq]

        return sequence, valor
```

Posteriormente, seleccionamos algunos hiperparámetros que nos serán de utilidad y creamos los sets de datos –uno por cada conjunto–, como se muestra en el siguiente código:

Código:

```
# Hiperparámetros

# Tamaño de la secuencia
seq = 20
# Tamaño del batch
batch = 20

# Crear sets de datos
ds_entre = RNNDATA(entre, seq)
ds_val = RNNDATA(val, seq)
ds_prueba = RNNDATA(prueba, seq)
```

Probamos nuestro *dataset* pidiendo que nos retorne los valores correspondientes al índice cinco, el cual nos devuelve una tupla con dos tensores, en el primero, se encuentran veinte valores y, en el segundo, se

encuentra el valor siguiente que es justo lo esperado. Posteriormente, se crean los objetos iterativos (llamados **dataloaders**) a partir de los sets de datos, como se muestra en el siguiente código:

Código:

```
# Probar el set de datos
ds_entre.__getitem__(5)

# Definimos el dataloader con los datos de entrenamiento
loader_entre = DataLoader(ds_entre, batch_size=batch)
# Definimos el dataloader con los datos de validación
loader_val = DataLoader(ds_val, batch_size=batch)
# Definimos el dataloader con los datos de prueba
loader_prueba = DataLoader(ds_prueba, batch_size=batch)
```

Ya con nuestros datos preparados procedemos a realizar nuestra red. Creamos una clase hija de **Module** a la que llamaremos **LSTM**. En esta clase, vamos a guardar el tamaño de la entrada, el tamaño del estado oculto, el tamaño del lote y el número de capas ocultas LSTM que tendremos que pueden ser desde 1 hasta n.

Para este modelo, tenemos tres métodos, **__init__()**, **init_hidden()** y **forward()**. Empecemos con **__init__()**, aquí definiremos las variables mencionadas y las capas que tendrá nuestro modelo, utilizaremos la(s) capa(s) LSTM que nos provee PyTorch, la cual se crea con las dimensiones de la entrada, el estado oculto y el número de capas, además, le agregaremos un dropout de 0.5 entre cada capa y definimos una capa de salida lineal con una activación ReLU. Después, en **init_hidden()**, vamos a crear e inicializar en cero el estado oculto, que es una matriz cuyas dimensiones son el número de capas LSTM, el tamaño del lote y la dimensión del estado oculto. Finalmente, tendremos el método **forward()**, que es donde se ejecutará el modelo y donde obtendremos las predicciones.

Código:

```
class LSTM(nn.Module):

    def __init__(self, input_dim, hidden_dim, batch_size, output_dim=1, num_layers=2):
        super().__init__()
        # Tamaño de la entrada
        self.input_dim = input_dim
        # Tamaño del estado oculto
        self.hidden_dim = hidden_dim
        # Tamaño del batch
        self.batch_size = batch_size
        # Número de capas lstm
        self.num_layers = num_layers

        # Definimos la capa LSTM, con un dropout después de cada capa (excepto en la última)
        self.lstm = nn.LSTM(self.input_dim, self.hidden_dim, self.num_layers, dropout=0.5)
        # Agregamos un dropout para la última capa
        self.drop = nn.Dropout(0.5)
        # Definimos la capa de salida
        self.linear = nn.Linear(self.hidden_dim, output_dim)
        self.relu = nn.ReLU(inplace=True)

    def init_hidden(self):
```

```

# Inicializamos en ceros nuestro estado oculto
return (torch.zeros(self.num_layers, self.batch_size, self.hidden_dim))

def forward(self, x):
    # Estructura de la salida de la capa lstm: [seq, batch_size, input_size]
    lstm_out, self.hidden = self.lstm(x)

    # Tomamos el último valor
    y_pred = self.linear(lstm_out[-1, :, :])
    return y_pred

```

Se definen los hiperparámetros para nuestro modelo, los criterios de término del modelo y, por último, el optimizador a utilizar –Adam, en este caso–.

Código:

```

# Hiperparámetros

# Tamaño de la entrada (características)
in_size = 1
# Número de capas
n_layers = 3
# Tamaño de la salida
o_size = 1
# Tamaño del estado oculto
hid_size = 30
# Tasa de aprendizaje (learning rate)
lr = 0.001

# Instanciamos el modelo
lstm = LSTM(in_size, hid_size, batch, o_size, n_layers)
# Establecemos el criterio
criterio = nn.MSELoss()
# Establecemos el optimizador
optimizador = torch.optim.Adam(lstm.parameters(), lr)

```

Lo siguiente es realizar el entrenamiento del modelo. Para realizar el entrenamiento de nuestra red, utilizaremos los conjuntos de entrenamiento y validación, graficaremos la pérdida de ambos conjuntos, el RSME del conjunto de validación y las predicciones del conjunto de validación para ver cómo va progresando el entrenamiento de nuestro modelo.

Primero crearemos una función llamada **evaluar**, que es donde evaluaremos el modelo, esta función recibe como parámetros el modelo **modelo**, el tamaño de la secuencia **seq**, el tamaño del lote **batch**, el tamaño de la entrada **in_size**, el *data loader* y *data set* que evaluará, **loader** y **ds**, respectivamente, y dos parámetros opcionales, **criterio** y **loss**, los cuales serán nuestros indicadores para calcular la pérdida, si es que estamos entrenando el modelo u omitir este paso si solo lo estamos evaluando.

Dentro de esta función iremos recuperando los datos del *data loader*, en cada iteración «reiniciamos» el estado oculto y reacomodamos los datos que recibe nuestro modelo a la forma que piden las capas LSTM, si la opción de pérdida está activada, la calculamos, después regresamos los valores sin normalizar y los agregamos a las listas. Finalmente, calculamos el MSE (*mean squared error*) y retornamos los datos, como se muestra en el siguiente código:

```

Código:

def evaluar(modelo, seq, batch, in_size, loader, ds, criterio=None, loss=False):
    # Creamos las listas
    l_pred = []
    Y = []

    # Ponemos el modelo en modo evaluación
    modelo.eval()
    # Inicializamos variables
    total = 0
    perdida_data = 0
    # Comenzamos a extraer los datos
    for x, y in loader:
        # Inicializamos el estado oculto
        modelo.hidden = modelo.init_hidden()
        # Evaluamos los datos
        y_pred = modelo(x.reshape([seq, batch, in_size]))
        # Pérdida
        if loss:
            perdida = criterio(y_pred.view(-1), y)
            perdida_data += perdida.data
            total += len(y)

        # Guardamos los datos de predicción y los reales
        for i in range(batch):
            l_pred.append(y_pred[i].item() * (ds.max - ds.min) + ds.min)
            Y.append(y[i].item() * (ds.max - ds.min) + ds.min)

    # Convertimos las listas a arrays
    Y = np.asarray(Y)
    y_pred = np.asarray(l_pred)
    # Calculamos el mse
    mse = (Y - y_pred)**2
    mse = round(mse.sum() / (len(Y) - seq), 4)

    if loss:
        # Calculamos la pérdida
        perdida_data = float(perdida_data / float(total))
        perdida_data = round(perdida_data, 6)

    # Retornamos la pérdida y el mse
    return mse, perdida_data

else:
    # Retornamos los datos desnormalizados y el rmse
    return Y, y_pred, round(mse**0.5, 4)

```

Después construiremos la función **entrenamiento**, esta función recibe como parámetros los mismos que **evaluar** más el *loader* de entrenamiento **loader_entre**, el optimizador **optim**, el número de épocas **epocas** y cada cuánto mostrar el progreso **mostrar**. Dentro de esta función, realizaremos la actualización de los pesos

y retornaremos el número de épocas, el RSME de validación y la pérdida del conjunto de entrenamiento y de validación.

Código:

```
def entrenamiento():
    modelo, loader_entre, loader_val, ds_val, in_size, batch,
    hid_size, seq, optim, crit, epochas=30, mostrar=10
):
    # Creamos las listas
    l_perdida = []
    l_per_val = []
    l_mse = []

    for epoca in range(epochas + 1):
        # Ponemos el modelo en modo entrenamiento
        modelo.train()
        # Inicializamos variables
        perdida_data = 0
        total = 0
        # Extraemos los datos
        for x, y in loader_entre:
            # Inicializamos el estado oculto
            modelo.hidden = modelo.init_hidden()
            # Evaluamos
            y_pred = modelo(x.reshape([seq, batch, in_size]))

            # Función de pérdida
            perdida = crit(y_pred.view(-1), y)

            # Ponemos el gradiente en cero
            optim.zero_grad()
            # Calculamos la pérdida
            perdida.backward()
            # Actualizamos los pesos
            optim.step()

            perdida_data += perdida.data
            total += len(y)

        # Calculamos la pérdida
        perdida_data = float(perdida_data/float(total))
        perdida_data = round(perdida_data, 6)
        l_perdida.append(perdida_data)

        # Evaluamos el conjunto de validación
        mse, per_val = evaluar(
            modelo, seq, batch, in_size, loader_val, ds_val,
            criterio=crit, loss=True
        )
        # Guardamos los datos
```

```

l_per_val.append(per_val)
l_mse.append(mse)

if epoca % mostrar == 0:
    # Imprimimos el progreso
    print("Época: {} \t Pérdida ent: {} \t Pérdida val: {}".format(epoca, perdida_data, per_val))

# Convertimos las listas a arrays
l_perdida = np.asarray(l_perdida)
l_epoca = np.arange(epoca + 1)
l_per_val = np.asarray(l_per_val)
l_rmse = np.sqrt(np.asarray(l_mse))

return l_perdida, l_epoca, l_per_val, l_rmse

```

Para comenzar con el entrenamiento, se utiliza el siguiente código:

Código:

```

per, ep, per_val, rmse = entrenamiento(
    lstm, loader_entre, loader_val, ds_val, in_size,
    batch, hid_size, seq, optimizador, criterio
)

```

```

Época: 0 Pérdida ent: 0.001651 Pérdida val: 0.002334
Época: 10 Pérdida ent: 0.000688 Pérdida val: 0.001228
Época: 20 Pérdida ent: 0.00039 Pérdida val: 0.000855
Época: 30 Pérdida ent: 0.000355 Pérdida val: 0.000777

```

Para poder evaluar los resultados, se escala la pérdida de validación y se grafica como se muestra en el siguiente código (el resultado se muestra en la figura 8.105):

Código:

```

# Escalamos la pérdida de validación
per_val1 = per_val*(per.mean()/per_val.mean())
plt.plot(ep, per)
plt.plot(ep, per_val1)
plt.xticks(ep)
plt.locator_params(axis='x', nbins=12)
plt.xlabel("Épocas")
plt.ylabel("Pérdida")
plt.legend(["Entrenamiento", "Validación"])
plt.title("Pérdida")
plt.show()

```

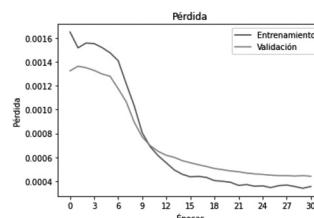


Figura 8.105. Pérdida de entrenamiento y validación por número de épocas para una red recurrente

Para poder realizar una adecuada evaluación, se calcula el error por la raíz de la suma de cuadrados (RMSE) y se grafica como se muestra en la figura 8.106.

Código:

```
plt.plot(ep, rmse)
plt.xticks(ep)
plt.locator_params(axis='x', nbins=12)
plt.xlabel("Épocas")
plt.ylabel("RMSE")
plt.title("RMSE validación")
plt.show()
```

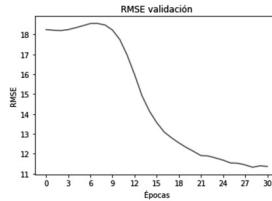


Figura 8.106. Error RMSE para una red recurrente RNN

Siempre es útil graficar para este tipo de datos tanto el valor real como la predicción, además de calcular el error –en este caso, RMSE–, como se muestra en el siguiente código y la figura 8.107:

Código:

```
# Evaluamos el conjunto de validación.
y, y_pred, rmse = evaluar(lstm, seq, batch, in_size, loader_val, ds_val)

# Graficamos el modelo de la validación
plt.plot(y)
plt.plot(y_pred)
plt.xlabel("Horas")
plt.ylabel("PM10")
plt.title("Set de validación\n")
plt.title("RMSE: " + str(rmse), loc="right")
plt.legend(["Real", "Predicción"])
plt.show()
```

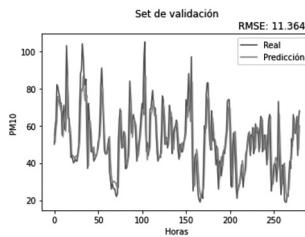


Figura 8.107. Datos reales vs datos predichos por una red recurrente RNN

En este ejercicio, se consideró pertinente calcular como métrica de error además de RMSE, el coeficiente de correlación o CC.

El coeficiente de correlación es un indicador de qué tan asociadas están dos variables. El valor de este coeficiente, r , va de -1 hasta 1, mientras más cercano sea r a 1 o -1, indica que las variables están

estrechamente relacionadas, mientras que, si es cercano a 0, quiere decir que su correlación es débil. La ecuación para sacar el coeficiente de correlación es la siguiente:

$$r = \sigma_{x,y} / \sigma_x \sigma_y$$

Donde:

$\sigma_{x,y}$ es la **covarianza** de (x,y) .

σ_x es la **desviación estándar** de x .

σ_y es la **desviación estándar** de y .

El siguiente código muestra el cálculo del coeficiente de correlación:

Código:

```
def correlacion(x, y):
    # Sacamos el valor promedio
    x_prom = np.average(x)
    y_prom = np.average(y)
    # Sacamos la covarianza
    cov = (x-x_prom)*(y-y_prom)
    cov = cov.sum()/len(x)
    # Sacamos la desviación estándar (std)
    x_std = x.std()
    y_std = y.std()

    return cov/(x_std*y_std)
correlacion(y, y_pred)
```

0.7849037783137425

Tenemos un coeficiente de correlación de 0.78, lo que nos indica que es un buen modelo que ha aprendido los patrones de la serie sin caer en el sobreentrenamiento.

APÉNDICE

Datos Usados en este Libro

En el presente libro, se abordan una serie de bases de datos que intentan clarificar algunos conceptos tratados a manera de casos de estudio. Las bases de datos que se presentan en este libro están libres de regalías, son muy utilizadas por la comunidad de desarrolladores en ciencia de datos y pueden ser localizadas tanto en algunas universidades a manera de repositorio como en sitios web si necesidad del pago de regalías.

Tanto los códigos como las bases de datos usadas en el presente libro pueden ser localizados en:
http://www.amese.net/libro_ia/datos/

A.1. Lirios

La base de datos de lirios es probablemente la más conocida para entrenar algoritmos de aprendizaje máquina que hay. Esta base de datos consiste en identificar tres tipos de lirios por medio de cuatro atributos, los cuales son:

- Longitud del sépalo (en cm).
- Anchura del sépalo (en cm).
- Longitud del pétalo (en cm).
- Anchura del pétalo (en cm).

El atributo de decisión es el quinto parámetro, presentado de manera categórica, son tres tipos de lirios, los cuales son:

- *Iris setosa*.
- *Iris versicolor*.
- *Iris virginica*.

Esta base de datos es con lo que muchas veces se comienza a trabajar cuando se requiere implementar algoritmos de clasificación. Esto es debido a que se tienen igual cantidad de instancias para cada lirio –en este caso, 50– y una de las categorías se presenta completamente separada de las otras dos, lo que hace su clasificación más fácil que con otros algoritmos. Básicamente, con respecto al tamaño de sépalo y pétalo, se tiene que determinar qué tipo de lirio se trata. Un ejemplo de cada clase es el siguiente:

No.	L. Sépalo	A. Sépalo	L. Pétalo	A. Pétalo	Lirio
10	4.9	3.1	1.5	0.1	Setosa
79	6	2.9	4.5	1.5	Versicolor
115	5.8	2.8	5.1	2.4	Virginica

Tabla A.1. Ejemplo de cada clase BD de lirios

Puede ser encontrada en: http://www.amese.net/libro_ia/datos/iris.csv. Este archivo contiene tanto la base de datos ordenada en sus instancias por tipo de atributo en un archivo .txt y un archivo .csv. Los archivos contienen la misma información, lo que cambia es el formato del archivo.

A.2. Enfermedad coronaria

Esta base de datos también puede ser encontrada en el repositorio de UCI o en: http://www.amese.net/libro_ia/datos/HeartDisease.csv. Esta base de datos consiste en determinar si una persona tiene una enfermedad coronaria o no por medio de sus atributos. Consta de 462 instancias y tiene los siguientes cinco atributos:

- **sbp.** «Sistolic Blood Pressure» o presión sanguínea sistólica.
- **famhist.** Historial familiar. Si la persona –en este caso, cada instancia– menciona que tienen un historial familiar con problemas de enfermedad coronaria, la observación de tipo binario sería 1 si tiene historial familiar con enfermedad coronaria y 0 si no tiene historial familiar.
- **obesity.** Se refiere al índice de masa corporal de acuerdo a su peso y su estatura. En este sentido, la base de datos no cuenta con información acerca del peso y la estatura de cada individuo –instancia– ni tampoco su género, pero existe en este atributo el cálculo de obesidad.
- **age.** Se refiere a la edad –en años cumplidos– del individuo.
- **chd.** Es el atributo de salida, llamado también atributo de decisión es de tipo nominal (binario). Se refiere a si el individuo presenta una enfermedad coronaria, donde, si su instancia tiene un 0, significa que el individuo no presenta una enfermedad coronaria, mientras que un 1, si la presenta.

A pesar de que, en esta base de datos, podría faltar más información, principalmente médica, es algo muy común que sucede cuando nos enfrentamos a datos que no recolectamos por nuestra cuenta. Sin embargo, este es un ejercicio interesante en varios sentidos. En primer lugar, para identificar qué atributos tienen una mayor relación con la enfermedad coronaria. En este sentido, tanto la presión como el índice de masa corporal, el historial familiar y la edad podrían parecer todos importantes para determinar si un individuo podría tener una enfermedad coronaria. En este caso, algunos parámetros son más importantes que otros en cuestión de los atributos que se tienen en la base de datos.

Así mismo, los atributos presentan diferencias fundamentales como lo son tipo de atributo, rango, intervalo, etcétera, lo cual representa un buen reto para clasificación.

A.3. Gorriones

Esta base de datos es un claro ejemplo del tipo de base de datos que se encuentran disponibles. Esta base de datos no cuenta con mucha información, salvo la premisa de que se midieron a un grupo de gorriones después de una tormenta. Algunos de ellos sobrevivieron y otros no.

Esto es importante mencionarlo, debido a que no cuenta con más información geográfica de la tormenta, qué tan severa fue la tormenta ni la ubicación de los gorriones supervivientes en relación con los no supervivientes.

Es decir, solo se encuentra con información estadística, pero es un buen ejemplo de lo que se puede extraer de una base de datos con poca información. Los atributos son cinco y están numerados de X1 a X5. Todas las mediciones están hechas en milímetros (mm) y están organizadas de la siguiente manera:

- X1: longitud total.
- X2: extensión del ala.
- X3: longitud del pico y cabeza.
- X4: longitud del húmero.
- X5: longitud del esternón.

Las tablas A.2 y A.3 muestran la lista de las mediciones tanto para los gorriones supervivientes como para los no supervivientes. El archivo «Gorriones.csv» puede ser descargado en:
http://www.amese.net/libro_ia/datos/Gorriones.csv

Supervivientes					
x1	x2	x3	x4	x5	
156	245	31.6	18.5	20.5	
154	240	30.4	17.9	19.6	
153	240	31	18.4	20.6	
153	236	30.9	17.7	20.2	
155	243	31.5	18.6	20.3	
163	247	32	19	20.9	
157	238	30.9	18.4	20.2	
155	239	32.8	18.6	21.2	
164	248	32.7	19.1	21.1	
158	238	31	18.8	22	
158	240	31.3	18.6	22	
160	244	31.1	18.6	20.5	
161	246	32.3	19.3	21.8	
157	245	32	19.1	20	
157	235	31.5	18.1	19.8	
154	237	31.6	18	20.3	
158	244	31.4	18.5	21.6	
153	238	30.5	18.2	20.9	
155	236	30.3	18.5	20.1	
163	246	32.5	18.6	21.9	
159	236	31.5	18	21.5	

Tabla A.2. Lista de supervivientes para la BD de gorriones

No Supervivientes					
x1	x2	x3	x4	x5	
155	240	31.4	18	20.7	
156	240	31.5	18.2	20.6	
160	242	32.6	18.8	21.7	
152	232	30.3	17.2	19.8	
160	250	31.7	18.8	22.5	
155	237	31	18.5	20	
157	245	32.2	19.5	21.4	
165	245	33.1	19.8	22.7	
153	231	30.1	17.3	19.8	
162	239	30.3	18	23.1	
162	243	31.6	18.8	21.3	
159	245	31.8	18.5	21.7	
159	247	30.9	18.1	19	
155	243	30.9	18.5	21.3	
162	252	31.9	19.1	22.2	
152	230	30.4	17.3	18.6	
159	242	30.8	18.2	20.5	
155	238	31.2	17.9	19.3	
153	249	33.4	19.5	22.8	
163	242	31	18.1	20.7	
156	237	31.7	18.2	20.3	
159	238	31.5	18.4	20.3	
161	245	32.1	19.1	20.8	
155	235	30.7	17.7	19.6	
162	247	31.9	19.1	20.4	
153	237	30.6	18.6	20.4	
162	245	32.5	18.5	21.1	
164	248	32.3	18.8	20.9	

Tabla A.3. Lista de no supervivientes para la BD de gorriones

A.4. Distancia ciudades en américa

Este ejercicio se utiliza principalmente para problemas de optimización para encontrar mínimos globales cuando el espacio de búsqueda es demasiado grande como para explorar todas las posibles soluciones. En este caso, se tienen dieciocho ciudades del continente americano y las posibles distancias de cada ciudad a cada otra ciudad, como se muestra en la figura A.1. Las ciudades que se muestran son:

- Ciudad de México.
- Quito.
- Miami.
- San Salvador.
- Mendoza.
- Guadalajara.
- Mérida.
- Washington.
- Monterrey.
- Managua.
- Caracas.
- Boston.
- Buenos Aires.
- Nueva York.
- Ciudad de Panamá.
- Brasilia.
- Montevideo.
- Bogotá.

Tabla A.4. Tabla de distancias de ciudades de América

En la tabla A.4, se puede observar que existen dieciocho! posibles combinaciones de distancias, es decir: 6.4×10^{12} posibles combinaciones, lo que hace muy complicado explorar todas las posibles combinaciones de distancias, mientras que la figura A.1 muestra el mapa de las ciudades que se presentaron en la tabla A.4. También el no utilizar métodos metaheurísticos, conlleva a encontrar soluciones que no son cercanamente la mejor distancia, es decir, que converja en una solución que no es la mejor, la diferencia en kilómetros para este ejercicio puede ser considerable. El nombre del archivo con la tabla de distancias se llama «DistanciaCiudades.csv» y puede ser descargado en:

http://www.amese.net/libro_ia/datos/DistanciasCiudades.csv



Figura A.1. Mapa de las ciudades en el continente americano usada para ejercicios de optimización

A.5. Condiciones climáticas

Esta base de datos es muy corta y este tipo de datos con pocas observaciones, pocas instancias y pocos atributos son usados usualmente en ejemplos de clasificación y regresión por árboles de decisión –ejemplo: algoritmos ID3 y CART–.

Consta de seis atributos contando el atributo de decisión. Está separado por días («Atributo Día») en donde se muestra el clima, la temperatura, la humedad y el viento en diferentes días. Por último, el atributo de decisión llamado: «¿Salir?» muestra si, de acuerdo a los parámetros, se puede salir o no. Es importante mencionar que la base de datos no tiene acentos para evitar problemas de importación por el formato.

Día	Clima	Temperatura	Humedad	Viento	¿Salir?
1 Soleado	Cálido	Alta	Débil	No	
2 Soleado	Cálido	Alta	Fuerte	No	
3 Nublado	Cálido	Alta	Débil	Si	
4 Lluvioso	Templado	Alta	Débil	Si	
5 Lluvioso	Frión	Normal	Débil	Si	
6 Lluvioso	Frión	Normal	Fuerte	No	
7 Nublado	Frión	Normal	Fuerte	Si	
8 Soleado	Templado	Alta	Débil	No	
9 Soleado	Frión	Normal	Débil	Si	
10 Lluvioso	Templado	Normal	Débil	Si	
11 Soleado	Templado	Normal	Fuerte	Si	
12 Nublado	Templado	Alta	Fuerte	Si	
13 Nublado	Cálido	Normal	Débil	Si	
14 Lluvioso	Templado	Alta	Fuerte	No	

Tabla A.5. Condiciones climáticas para la base de datos clima

Los datos presentados en la tabla A.5. presentan los siguientes atributos:

- **Día.** Es un atributo ordinal, representa los días del 1 al 14.
- **Clima.** Es un atributo nominal con tres diferentes observaciones, las cuales son: soleado, nublado y lluvioso.
- **Temperatura.** También es un atributo nominal con tres diferentes observaciones, las cuales son: frío, templado y cálido.
- **Humedad.** Atributo nominal con dos observaciones, alta o normal.
- **Viento.** Atributo nominal binario. Sus dos observaciones son: débil y fuerte.

- **Salir.** Atributo nominal de decisión. Determina si se puede salir o no y sus observaciones son: sí y no.

La tabla puede encontrarse en: http://www.amese.net/libro_ia/datos/

El archivo «Clima.xls» se encuentra en la misma ruta y contiene la misma información, solo tiene un diferente formato.

A.6. Síntomas de meningitis

Este ejercicio es también corto y muy parecido a los datos del apéndice A.5. «Condiciones climáticas». El de síntoma de meningitis describe la presencia o ausencia de la enfermedad meningitis en un grupo reducido de instancias.

En este ejemplo, se toman en cuenta diferentes personas (10) separadas por un ID, como se muestra en la tabla A.6. Los atributos son dolor de cabeza, fiebre y vómito. Con esos atributos dicotómicos –binarios–, se define el atributo de decisión también dicotómico llamado meningitis, que determina si con los otros atributos se tiene meningitis o no. La tabla se encuentra en:

http://www.amese.net/libro_ia/datos/Meningitis.csv

ID	Dolor de Cabeza	Fiebre	Vómito	Meningitis
1Sí	Sí	No	No	
2No	Sí	No	No	
3Sí	No	Sí	No	
4Sí	No	Sí	No	
5No	Sí	No	Sí	
6Sí	No	Sí	No	
7Sí	No	Sí	No	
8Sí	No	Sí	Sí	
9No	Sí	No	No	
10Sí	No	Sí	Sí	

Tabla A.6. Tabla de la BD de meningitis

A.7. Datos de partículas contaminantes PM10

Este ejercicio es ideal cuando se tiene que modelar, predecir o imputar datos continuos.

Esta base de datos contiene los datos de partículas contaminantes PM10 en la Ciudad de México en el 2020 de todas las estaciones que monitorea en la ciudad y área metropolitana, mismo que fue descargado de la página oficial de la red de monitoreo RAMA, la cual es libre de regalías y se encuentra en la página oficial de calidad del aire del gobierno de la Ciudad de México: <http://wwwaire.cdmx.gob.mx/default.php>

Los atributos están separados por la fecha, la hora en la que se tomó la muestra y el acrónimo de cada estación de monitoreo. El número está dado en microgramos por metro cúbico (ugm-3) y consiste en 8746 instancias y 27 atributos con datos de cada estación de monitoreo.

Existe un gran número de instancias en prácticamente todos los atributos que tienen datos no válidos, lo que hace más complejo la predicción de estos datos. Estas instancias aparecen con un valor de -99 para que no se confundan con un 0, que implicaría que, en esa hora, en esa particular estación de monitoreo, no hubo contaminantes PM10. Un ejemplo con solo veinticuatro datos –un día– y nueve estaciones de monitoreo se muestra en la figura A.2.

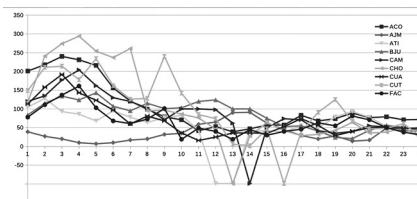


Figura A.2. Ejemplo de datos continuos de partículas contaminantes PM10

Los datos se encuentran en: http://www.amese.net/libro_ia/dat

En la sección 5.16.2 se abordó las métricas de exactitud para datos continuos. A pesar de que los datos reales se tomaron de esta base de datos, los datos de predicción para poder calcular el error se pueden obtener en: http://www.amese.net/libro_ia/datos_continuos.xlsx.

A.8. Otras bases de datos

En este apartado, se presentan otras bases de datos utilizadas en el presente libro. Se separaron de esta manera, debido a que, o se realizaron modificaciones para un ejemplo en específico, o se agregaron posteriormente para exemplificar una práctica o solo se utilizan una vez.

A.8.1. Datos de pasajeros del Titanic

Esta base de datos es de las más populares tanto en foros de aprendizaje máquina como concursos de minería de datos. Puede ser descargada en: http://www.amese.net/libro_ia/datos/Titanic.csv

Esta base de datos consiste en determinar, por medio de sus atributos, si los pasajeros del Titanic sobrevivieron o no. Consta de 891 instancias y tiene los siguientes doce atributos:

- **PassengerId**. Es un atributo ordinal. Representa la numeración de los pasajeros.
- **Survived**. Es el atributo de salida –de decisión–. Es un atributo dicotómico nominal. El 0 representa que no sobrevivió, mientras que el 1 representa que sí sobrevivió.
- **Pclass**. Es un atributo nominal con tres observaciones. El 1 representa primera clase, el 2, segunda clase y, por último, el 3, tercera clase.
- **Name**. Este atributo representa el nombre del pasajero.
- **Gender**. Atributo nominal dicotómico –binario–. Representa el género del pasajero donde *male* es hombre y *female*, mujer.
- **Age**. Atributo tipo intervalo. Representa la edad del pasajero en años cumplidos.
- **SibSp**. Atributo tipo ordinal. Representa el número de hermanos, hermanas o consorte abordo del Titanic.
- **Parch**. Atributo tipo ordinal. Representa el número de padres o hijos a bordo del Titanic.
- **Ticket**. Número de ticket con el cual se abordó el Titanic.
- **Fare**. Atributo que muestra la tarifa que pagó el pasajero.
- **Cabin**. Número de cabina asignada en el Titanic.
- **Embarked**. Puerto de embarcación. La «C» corresponde al puerto de Cherbourg, la «Q», al puerto de Queenstown y, por último, la «S», al puerto de Southampton.

A.8.2. Datos aleatorios de edades

Esta base de datos fue creada específicamente para exemplificar un ejercicio cuando se tiene errores de cardinalidad en una base de datos. Puede ser descargada en: http://www.amese.net/libro_ia/datos/

La base de datos consta de dieciocho instancias y dos atributos: edad y género. Ambos atributos tienen errores de cardinalidad. El atributo edad tiene un error de cardinalidad de doscientos diez años que no puede confundirse con un dato atípico válido –comúnmente llamado outlier–, mientras que el atributo género tiene errores de cardinalidad diferentes a «h» (hombre) y «m» (mujer).

Glosario de Términos

A

Agrupación. El proceso de dividir un grupo heterogéneo de objetos en subgrupos homogéneos. Los algoritmos de agrupación encuentran grupos de instancias que son similares entre sí.

Algoritmo. Un set de instrucciones secuenciales –paso a paso– para resolver un problema.

Algoritmo genético. Un tipo de algoritmo derivado del cómputo evolutivo inspirado en la teoría de evolución de Darwin. Un algoritmo genético genera una población aplicando operadores genéticos, cruza y mutación. Al repetir este proceso por muchas generaciones, el algoritmo genético provee una solución óptima al problema.

Algoritmo genético híbrido. Un algoritmo genético que se combinan con otras técnicas de optimización.

Algoritmo genético multiisla. Una implementación de un algoritmo genético paralelo en la cual existen subpoblaciones que evolucionan de manera separada con comunicación limitada entre cada una.

Algoritmo evolutivo. Un programa de cómputo que utiliza el concepto de evolución biológica para resolver problemas. Por ejemplo, algoritmos genéticos, programación genética, estrategias evolutivas y programación evolutiva.

Alelo. Los valores que puede tomar un gene.

Aprendizaje. El proceso por el cual se ajusta un algoritmo para alcanzar el comportamiento deseado.

Aprendizaje competitivo. Aprendizaje no supervisado en el que los individuos –o neuronas, depende del algoritmo– compiten entre ellos de tal forma que solo un individuo gana la «competencia» y, por tanto, se posiciona como el mejor.

Aprendizaje máquina. Mecanismo adaptativo que permite que los sistemas aprendan de la experiencia, del ejemplo o de la analogía. Las capacidades de aprendizaje mejoran el sistema conforme pasa el tiempo.

Aprendizaje no supervisado. También conocido como aprendizaje automático. Durante el proceso de aprendizaje, el sistema descubre características significativas durante el aprendizaje y aprende a clasificar datos de entrada en sus categorías apropiadas.

Aprendizaje supervisado. El tipo de aprendizaje que requiere una secuencia de ejemplos etiquetados. Cada ejemplo contiene el patrón de entrada y el patrón deseado de salida.

Aptitud. Lo opuesto al costo. La aptitud es un valor asociado con un cromosoma que asigna un mérito relativo a ese cromosoma.

Árbol de decisión. Es una representación gráfica de un set de datos que describe los mismos como estructuras parecidas a las de un árbol. Un árbol de decisión consiste en nodos, ramas y hojas. El árbol siempre comienza en un nodo raíz y separa los datos de acuerdo a características y observaciones en un subnivel con nuevos nodos. Son particularmente buenos resolviendo problemas de clasificación con pocas observaciones y atributos.

Arquitectura. Ver «Topología».

Atributo. Una propiedad de un objeto. Por ejemplo, en la base de datos «Lirios», los atributos son: longitud del sépalo, anchura del sépalo, longitud del pétalo y anchura del pétalo.

Atributo de decisión. Es el tributo propio de la clasificación. En el mismo caso de la base de datos «Lirios», el atributo de decisión es el tipo de lirio, el cual tiene las observaciones: *Iris setosa*, *Iris versicolor* e *Iris*

virginica.

Autoadaptación. La inclusión de un mecanismo de evolucionar no solo la solución, sino la información de cómo cada solución genera nuevos hijos.

Axón. Utilizada en redes neuronales, el axón es modelada por la salida de la neurona.

B

Base de datos. Una colección de datos estructurados.

Bit. Un dígito binario que puede tomar valores dicotómicos, normalmente «0» y «1».

Búsqueda. Es el proceso de examinar el set de posibles soluciones para encontrar la solución aceptable.

Búsqueda exhaustiva. Una técnica de solución de problemas en donde cada posible solución es examinada hasta que se encuentra una posible solución.

Búsqueda heurística. Una técnica de solución de problemas en donde se aplican técnicas para guiar la búsqueda y obtener resultados viables sin que se exploren todas las soluciones posibles.

Búsqueda tabú. Una técnica de búsqueda donde el algoritmo de búsqueda lleva un seguimiento de las soluciones con un alto costo, para poder evitar dichas soluciones en el futuro.

C

Caja negra. Modelo que es opaco al usuario. Aunque el modelo puede producir resultados correctos, sus parámetros e interconexiones no son conocidas por el usuario.

Capa. Llamada también *layer*. Es un grupo de neuronas que tienen una función específica y que son procesadas como un todo.

Capa de entrada. Es la primera capa de neuronas de una red neuronal artificial. La capa de entrada acepta las señales de entrada y las redistribuye en neuronas a la siguiente capa. La capa de entrada rara vez incluye neuronas de cómputo y no procesa patrones de entrada.

Capa de salida. La última capa de neuronas de una red neuronal artificial. La capa de salida produce el patrón de salida de toda la red.

Capa oculta. Una capa de neuronas entre las capas de entrada y de salida. Se llama «oculta» debido a que, en esta capa, no se puede observar el comportamiento de la red neuronal artificial.

Característica. En aprendizaje máquina y reconocimiento de patrones, una característica es una propiedad medible de un fenómeno siendo observado.

CART. Árbol de clasificación y regresión, por sus siglas en inglés. CART provee de un set de reglas que pueden ser aplicadas a sets nuevos de datos. CART segmenta las instancias de datos en forma de árbol.

Centroide. Un centroide es el centro de gravedad donde sus elementos están posicionados alrededor del centro de masa del centroide.

Clase. Un grupo de objetos con atributos comunes.

Clonación. Crear un hijo el cual es una copia exacta del padre.

Clustering. Ver «Agrupación».

Codificación. El proceso de transformar información de un esquema a otro.

Cálculo evolutivo. Algoritmo usado para simular la evolución de los seres vivos en una computadora. El campo de cálculo evolutivo incluye algoritmos genéticos, estrategias evolutivas y programación genética.

Complemento. En la teoría clásica, el complemento de A es el set de elementos que no son miembros de A.

En la teoría difusa, el complemento de un set es lo opuesto a ese set.

Conexión. Se conoce también como sinapsis. Es asociada con el acoplamiento desde una neurona a la otra para transferir señales.

Conocimiento. Entendimiento teórico o práctico de un tema. Conocimiento es lo que nos permite realizar decisiones informadas.

Convergencia. Llegar a una solución. En algoritmos genéticos, se llega a una convergencia cuando sus cromosomas dejan de encontrar mejores soluciones. En redes neuronales, se habla de convergencia cuando el error ha llegado al umbral determinado, indicando que la tarea se ha aprendido.

Costo. El resultado de la función costo.

Cromosoma. Un arreglo de genes que son evaluados por la función de aptitud. Una cadena de genes que representa a un individuo.

Cruza. Un operador de reproducción que forma un nuevo cromosoma a partir de dos cromosomas padres, combinando parte de la información de cada uno.

Cruza de dos puntos. Un esquema de crusa que selecciona dos puntos en el cromosoma con el fin de intercambiar los genes entre esos dos puntos.

Cruza de un punto. Un punto en dos padres –cromosomas– que se selecciona. Los hijos se forman tomando el material genético de la parte izquierda de un padre con la parte derecha del otro padre y viceversa.

Cruza por ciclo. Método de crusa para problemas de permutación en el cual un punto es inicialmente seleccionado para un intercambio de genes entre los pares, después, el remanente del operador «cicla» a través de los padres para eliminar repeticiones de genes en los hijos.

Cruza por correspondencia parcial. PMx por sus siglas en inglés. Es un método de crusa para problemas de permutación en el cual dos puntos de crusa se eligen, se escogen los valores entre esos puntos y se refiere a la matriz de correspondencia entre esos puntos, en caso de repeticiones.

Cruza por orden. OX por sus siglas en inglés. Un método de crusa que lida con el operador de permutación que procura preservar porciones del orden absoluto de cada parente.

Cruza uniforme. Método de crusa que asigna aleatoriamente a cada hijo un gene de un parente o del otro. Usualmente, la crusa uniforme funciona con el 50 % de probabilidad de ser asignado a ese hijo en cuestión.

D

Datos. La representación simbólica de mediciones u observaciones.

Datos categóricos. Son los datos que pueden ser representados en un pequeño número de categorías discretas. Por ejemplo: género, estado civil, etcétera.

Defuzzificación. El último paso de la inferencia difusa. Es el proceso de convertir una salida combinada de reglas difusas a sus correspondientes valores numéricos.

Dendrita. Una rama de las neuronas biológicas, la cual se encarga de transferir información de una parte de la célula a otra. En redes neuronales, las dendritas son modeladas como entradas a una neurona.

Distancia de Hamming. El número de genes que un cromosoma difiere de otro.

Distancia euclíadiana. La distancia más corta entre dos puntos en coordenadas cartesianas.

E

Edad de un cromosoma. El número de generaciones que un cromosoma ha existido.

Elitismo. Técnica mediante la cual se conserva el mejor cromosoma de generación en generación.

Entrenamiento. En aprendizaje máquina, el entrenamiento es el modelo que se genera para predecir los datos de entrada. Básicamente, los datos se dividen en datos de entrenamiento, validación y pruebas.

Época. La presentación de un set de entrenamiento en una red neuronal.

Error. La diferencia entre la salida deseada y la actual.

Espacio de búsqueda. El set de todas las posibles soluciones a un problema.

Especiación. El proceso de desarrollar nuevas especies, principalmente, en algoritmos genéticos. La forma más común de especiación ocurre cuando una especie esté geográficamente separada en el espacio de búsqueda por suficiente tiempo de la población principal para que sus genes tengan diferencias fundamentales.

Estancamiento. Ver «Estagnación».

Estagnación. El punto en el cual el algoritmo no encuentra una solución mejor, a pesar de no haber encontrado la solución óptima.

Estrategia evolutiva. Un tipo de algoritmo evolutivo que utiliza solo operadores de mutación y no requiere que el problema sea representado de forma codificada.

Evolución. Una serie de cambios genéticos en los cuales los organismos vivos adquieren las características que los distinguen de otros organismos.

Extremo. Un mínimo o un máximo.

F

Factor de certeza. Un valor asignado a un hecho o regla para indicar la certeza o confianza que se tiene de que dicho hecho o regla es válido.

Factor de confianza. Ver «Factor de certeza».

Falso negativo. *False negative* (FN, por sus siglas en inglés). Es una instancia en la cual se tienen un valor positivo, a pesar de haber predicho un valor negativo.

Falso positivo. *False positive* (FP, por sus siglas en inglés). Es una instancia en la cual se tiene un valor negativo, a pesar de haber predicho un valor positivo.

Función costo. Función para optimizar.

Función de activación. Una función matemática también llamada función de transferencia. Su uso es el de generar un mapa de una neurona hacia su salida de acuerdo a su valor de entrada. Algunos ejemplos de función de activación son: stepwise (por escalón), sigmoide, tanh, ReLU, SeLU, lineal, entre otros.

Función de aptitud. Función matemática usada en calcular la aptitud de un individuo.

Función de membresía. Función matemática que define el set difuso en el universo de discurso.

Función de optimización. El proceso de encontrar el mejor extremo a una función determinada.

Función objetivo. La función para optimizar.

Fuzzificación. El primer paso de la inferencia difusa. El proceso de mapear entradas numéricas a grados en los cuales dichas entradas pertenecen a sus respectivos sets difusos.

G

Gen. Gene. La unidad de un parámetro codificado que pertenece a un cromosoma.

Generación. Una iteración de un algoritmo genético.

Generalización. La habilidad de una red neuronal para producir resultados correctos a partir de datos que no han sido entrenados previamente.

Grado de membresía. El valor numérico entre 0 y 1 que representa el grado en el cual un elemento pertenece a un set en particular. También se conoce como valor de membresía.

H

Heurística. Una estrategia que puede ser aplicada a problemas complejos. Son usadas para reducir problemas complejos y resolverlos por medio de operaciones más simples basados en reglas.

I

Individuo. Un miembro de la población que consiste en un cromosoma y su función costo.

Inferencia difusa. El proceso de razonamiento basado en lógica difusa. La inferencia difusa incluye cuatro pasos: la fuzzificación de las variables de entrada, la evaluación de las reglas, agregar las reglas de salida y la defuzzificación.

Instancia. Un objeto en específico de una clase.

Instanciación. El proceso de asignar un valor específico a una variable. Por ejemplo, la observación de una instancia llamada «lunes» es una instanciación un atributo que se llame «día de la semana».

Inteligencia. La habilidad de aprender y entender, resolver problemas y tomar decisiones. En cómputo, la inteligencia se conoce como una máquina que puede alcanzar un nivel cognitivo de tal forma que pueda resolver una tarea a nivel humano o superior.

Inteligencia artificial. La rama de las ciencias computacionales que se encarga de desarrollar algoritmos que se comporten de manera «inteligente» y que resuelva problemas específicos de manera semiautónoma o automática o aprenda de experiencias previas.

Inversión. Un operador de reordenamiento que funciona seleccionando dos puntos de cruza en un cromosoma e invierte el origen de los genes entre dichos puntos.

K

K-medias. (K-means). Es un algoritmo de agrupamiento de los más simples y populares en aprendizaje máquina. K-means funciona identificando un número k de centroides y asignando cada dato a un agrupamiento específico de acuerdo a la cercanía del centroide.

L

Lógica booleana. Un sistema de lógica basada en la álgebra de Boole. Maneja dos valores: «verdadero» y «falso». Las condiciones booleanas de verdadero y falso se representan como un «1» para «verdadero» y como un «0» para «falso».

Lógica difusa. A diferencia de la lógica booleana, la lógica difusa adquiere diversos valores y maneja el concepto de verdades parciales –valores entre completamente verdadero o 1 y completamente falso o 0–. También se refiere a teoría de sets difusos.

M

Mapa de bits. La representación de una imagen por medio de filas y columnas de puntos.

Memoria asociativa. El tipo de memoria que nos permite asociar un concepto con otro. Un ejemplo muy común es asociar sonidos o escenas con escuchar una nota de música. También, la memoria asociativa en redes neuronales aplica cuando se toma un patrón entrenado previamente que presenta un patrón similar a la entrada. La red neuronal Hopfield es un buen ejemplo de esto.

Migración. La transferencia de los genes de un individuo de una subpoblación a otra.

Minería de datos. La extracción de conocimiento a partir de los datos. También se le conoce minería de datos a la exploración y análisis de grandes cantidades de datos para poder descubrir patrones y reglas a partir de dichos datos.

Mínimo global. El mínimo verdadero del espacio de búsqueda entero. Puede verse también como el valor mínimo del rango entero de sus parámetros de entrada.

Mínimo local. El valor mínimo en un subespacio del espacio de búsqueda entero. Puede verse también como el valor mínimo de una función de un rango limitado de sus parámetros de entrada.

Modelo determinista. Modelo matemático que determina su salida con completa certeza —sin variables aleatorias o incertidumbre—.

Mutación. Un operador de reproducción que altera el valor de algunos de los genes de un individuo de manera aleatoria. Esto se realiza con el fin de explorar nuevas soluciones dentro del espacio de búsqueda.

N

Neurona. Una célula que es capaz de procesar información. Es un elemento básico de las redes neuronales artificiales.

Nicho. La estrategia de supervivencia de un organismo. Esto aplica principalmente en algoritmos genéticos, donde especies en diferentes nichos —por ejemplo: una especie comiendo plantas y la otra comiendo insectos— pueden coexistir lado a lado sin competir. Sin embargo, si dos especies ocupan el mismo nicho, la especie más débil se extinguirá. En algoritmos genéticos, cada pico de la función costo es análogo a un nicho.

Nodo. Un punto de un árbol de decisión.

Nodo raíz. El nodo más alto de un árbol de decisión. El árbol siempre inicia del nodo raíz y crece separando los datos en cada nivel en nuevos nodos.

O

Operador de reproducción. El algoritmo usado en implementar la reproducción.

Operador genético. Actúa en un cromosoma para poder producir un nuevo individuo. Tanto para algoritmos genéticos como para programación genética, algunos operadores genéticos incluyen cruce y mutación.

Optimización. El proceso de iterativamente mejorar la solución a un problema con respecto a la función objetivo específica.

Optimización global. El proceso iterativo de encontrar el óptimo verdadero en el espacio de búsqueda entero.

Optimización multiobjetivo. Proceso de optimización en el cual la función objetivo regresa más de un valor.

Optimización por colonia de hormigas (*ant colony optimization - ACO*). Método de optimización global que mimetiza el camino óptimo que lleva a las hormigas de su nido a una fuente de comida.

Optimización por enjambre de partículas. Método de optimización global que mimetiza el comportamiento de enjambres —o cardumen— de algunos animales como pájaros o peces.

P

Padre. Un individuo que produce uno o más individuos diferentes conocidos como hijos.

Perceptrón. La forma más simple de red neuronal. La operación de un perceptrón es basada en el modelo de neuronas. Consiste en una única neurona con pesos sinápticos ajustables. El perceptrón aprende haciendo pequeños ajustes a los pesos para reducir la diferencia entre las salidas actual y deseada.

Peso. El valor asociado con la conexión entre dos neuronas en una red neuronal artificial. Este valor determina la fortaleza de la conexión e indica cuando de la salida de una neurona es alimentada a la entrada de otra.

Población. Un grupo de individuos de una generación que se componen de cromosomas y cada cromosoma de genes.

Porcentaje de crusa. Un número entre 0 y 1 que indica qué tan frecuentemente la crusa es aplicada a cada población.

Potencial de acción. Una señal de salida —también llamada impulso nervioso— de una neurona que no pierde fuerza en distancias largas.

Probabilidad de mutación. Un número entre 0 y 1 que indica la probabilidad que la mutación ocurra en un gene.

Programación evolutiva. Es un algoritmo evolutivo que típicamente utiliza selección por torneo, variables continuas y no realiza crusa.

Pruebas. Cuando a un subset de datos se aplica al modelo —entrenamiento— para evaluar su rendimiento.

R

Rama. La conexión entre los nodos en un árbol de decisión.

Red neuronal artificial. Un paradigma de procesamiento de información inspirado en la estructura y las funciones del cerebro humano. Consiste en un número de elementos interconectados llamadas neuronas, que tienen conexiones con cierto pesos llamadas sinapsis.

Recombinación. Combinar la información de dos cromosomas padres mediante la crusa.

Regla bayesiana. Método estadístico para actualizar las probabilidades inherentes a ciertos hechos cuando se obtiene nueva evidencia.

Regla difusa. Una regla condicional del tipo: si x es A, ENTONCES y es B, donde x e y son variables lingüísticas y A y B son valores lingüísticos determinados por los sets difusos.

Rendimiento. Evaluación estadística de la aptitud.

Reordenamiento. Cambio en el orden de los genes en un cromosoma para tratar de juntar genes relacionados entre sí y mejorar la aptitud del individuo.

Reproducción. La creación de hijos a partir del cromosoma de los padres.

Ruido. En ciencia de datos, el ruido se concibe como los errores asociados a la manera en que los datos son recolectados, medidos o interpretados.

S

Selección. El proceso de escoger padres para su posterior crusa. Usualmente, la selección se basa en el criterio de aptitud.

Selección natural. Es el proceso mediante el cual los individuos mejor adaptados tienen una mayor probabilidad de reproducirse y pasar su material genético a la siguiente generación.

Selección por ruleta. Un método de selección en el cual un individuo en específico se selecciona de la población padre con una probabilidad igual a su aptitud dividida entre la sumatoria de todas las aptitudes de la población de esa generación.

Selección por torneo. Método de selección en el cual se selecciona un subset de la población y se selecciona el miembro con la mejor aptitud.

Sensitividad. También llamada *recall* además de *sensitivity*. Una técnica para determinar qué tan sensible es la salida de un modelo con respecto a una entrada en particular.

Set. Una colección de elementos también llamado miembros.

Set de entrenamiento. Set de datos utilizado para modelar el comportamiento de los datos.

Set de pruebas. Un set de datos que es usado para verificar la habilidad de un modelo a generalizarse. El set de pruebas debe ser estrictamente independiente del set de entrenamiento.

Set difuso. Para representar un set difuso, se expresa como su función, en la cual sus elementos se representan como su grado de membresía.

Sinapsis. Una conexión química entre dos neuronas en una red neuronal biológica.

Sistema experto. Un programa de computadora que es capaz de realizar al nivel de un experto humano en un área en particular.

Sistema híbrido. Un sistema que combina por lo menos dos herramientas inteligentes, como lo pueden ser las redes neuronales, algoritmos genéticos, sistemas difusos, etcétera.

Sobreentrenamiento. El estado en el cual un algoritmo se ha aprendido todos los datos de entrenamiento, pero no puede generalizar.

Soma. El cuerpo de una neurona biológica.

Subpoblación. Una porción de la población principal en la que individuos solamente se cruzan con otros de la misma población. Muy usado en algoritmos genéticos multiislas.

Supervivencia. Los criterios mediante los cuales solo los individuos con la aptitud más alta pueden sobrevivir y pasar sus genes a la siguiente generación.

T

Tasa de aprendizaje. Un número positivo menor que la unidad que controla la tasa de cambio que los pesos de una red neuronal cambian de una iteración a la siguiente. La tasa de cambio (*learning rate*) afecta directamente la velocidad del entrenamiento de una red.

Tasa de aprendizaje adaptativo. La tasa de aprendizaje que se ajusta al cambio del error durante la ejecución del entrenamiento.

Tasa de convergencia. La velocidad en la cual el algoritmo se aproxima a una solución.

Tasa de crusa. Ver «Porcentaje de crusa».

Tasa de mutación. Porcentaje de genes en la población mutada por cada iteración de un algoritmo genético.

Test de Turing. Un test diseñado para determinar si una máquina puede pasar por una persona en términos de inteligencia humana y tareas cognitivas. El test es pasado si el interrogador que hace preguntas tanto a una persona como a una máquina no puede distinguir cuál es la persona.

Topología. Una estructura interna de una red neuronal artificial que se refiere al número de neuronas en cada capa, el número de capas y las conexiones entre neuronas.

U

Umbral. Un valor específico que deberá de ser rebasado para que la salida de la neurona se pueda generar.

V

Validación. Un subset de datos además de los sets de entrenamiento y pruebas. El set de datos de validación se encarga de ver qué tan exacto es el modelo y realizar ajustes si es necesario previo a las pruebas.

Verdadero negativo. Se refiere TN a sus siglas en inglés (*true negative*). Es una instancia en el set de pruebas en la que se espera un valor negativo de su característica y se predice un valor negativo también.

Verdadero positivo (TP). Se refiere TP a sus siglas en inglés (*true positive*). Se dice de una instancia en el set de pruebas en la que se espera que tenga un valor positivo de su característica y se predice un valor positivo también.

Visualización de datos. La representación de datos que ayuda a entender la estructura y el significado de la información contenida en los datos.



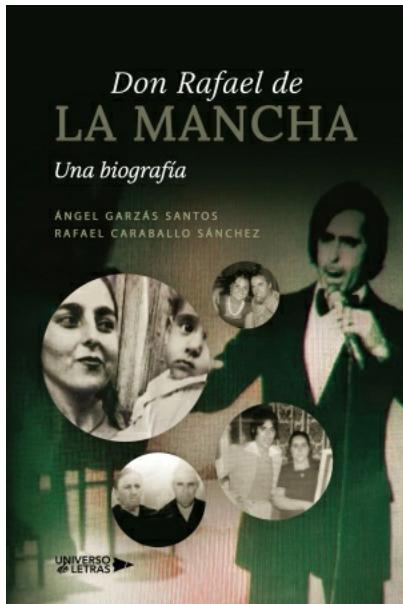
Las máscaras

Rodríguez Guerrero, Rafael
9788418676703

[Cómpralo y empieza a leer](#)

Cuando Benjamín sobrevive a una grave infección, abandona su rutinaria vida y la búsqueda de compañía femenina se convierte en una obsesión. Su periplo comienza en París, y durante una improvisada sesión fotográfica, descubre el poder de una cámara para generar intimidad y hacer que lo impensable ocurra con toda naturalidad. Sus aventuras lo arrastran a un torbellino emocional y erótico donde lo carnal se confunde con lo sublime, pero una nueva cirugía destruye su potencia sexual. Él no se amilana y para reavivar su libidinoso plan, se somete a un doloroso procedimiento que le repone con creces la capacidad que tenía de joven. Cristina, una diseñadora de modas que conoció de niña, se cruza en su camino y su belleza e intelecto le son irresistibles. La diferencia de edad hace imposible una relación, pero la palabra imposible no existe en el vocabulario de Benjamín. ¿Podrán ambos dar un inconcebible vuelco a sus vidas?

[Cómpralo y empieza a leer](#)



Don Rafael de la Mancha

Garzáns Santos, Ángel

9788418855931

[Cómpralo y empieza a leer](#)

Rafael de la Mancha es un cantante de canción española nacido en 1945. Desde pequeño tiene una vida llena de aventuras. Nace en un hospital de Ciudad Real y le ingresan en un hospicio, Casa Cuna, y su abuela desde Los Cortijos recorre 50 kilómetros en burro para traerse al bebé a vivir con ellos al pueblo. Lo crían como un hijo más, hermano de sus tíos y de su madre. Vive muy feliz en aquel pueblo de su infancia hasta que marcha a Urda para aprender un oficio con su madre y su padrastro. Luego sigue con su empeño de ser artista y canta. Canta con un circo y actúa en la radio y en la televisión en el programa La Gran Ocación. Con otros amigos cantantes hacen giras por toda España. Hasta que decide salir de España. Burdeos, París, Londres, Alemania, Austria, Italia, Grecia, son algunos de los lugares donde Rafael lleva su arte y su voz, representando la canción española. Hasta que finalmente se afincá en Turquía, donde permanece en activo cantando en distintas cadenas de la televisión turca.

[Cómpralo y empieza a leer](#)



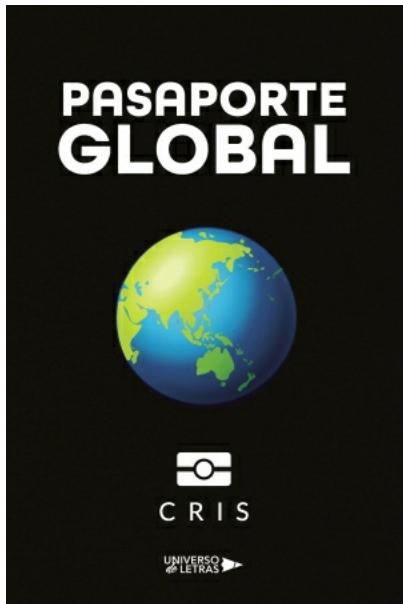
La urgencia de un sueño

Toro Vicencio, Mario
9788418855580

[Cómpralo y empieza a leer](#)

Joao Gilberto, brasileño, vuela a América para conocer ese país, de paso por su viaje a España para hacer un post grado. En Nueva York coincide con Birgit, sueca, y deciden no separarse más, transfundiendo en sus corrientes sanguíneas pasados, antepasados portugueses y escandinavos, existencias, vivencias, experiencias, sentimientos, sentidos y.... más sentidos que se expresan en un irresistible impulso de entrega, entrelazando sus espíritus para transponer el umbral del perpetuo horizonte, siempre sin límites y sin distancia conocida. Estando en pleno apogeo de su juventud crearon su propio Paraíso, del que nunca serían echados y del que nunca renegarían. Pero la más brutal de todas las incertidumbres y pesadillas, la pandemia, se encargaría de destrozar el infinito futuro de ellos, contagiándolos y separándolos. Birgit desaparece inexplicablemente en el hospital que la ingresa y nada hace suponer un desenlace feliz.

[Cómpralo y empieza a leer](#)



Pasaporte Global

Cris

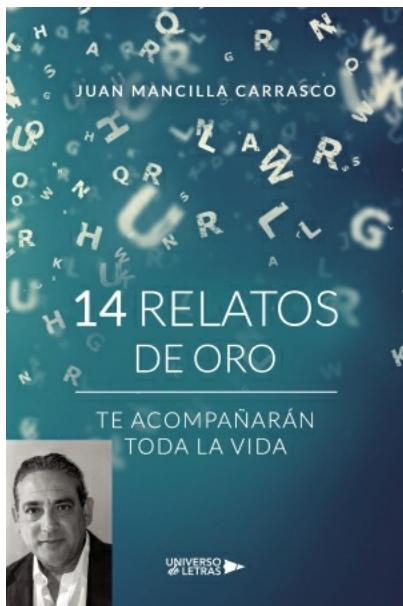
9788418571541

189 Páginas

[Cómpralo y empieza a leer](#)

¿Si tuvieras la oportunidad de elegir, ¿qué nacionalidad elegirías? ¿Por qué tenemos una nacionalidad? ¿Qué es el Estado-nación? ¿Puedo dejar de ser argentino, suizo, senegalés, chino o australiano? ¿Debo resignarme al arbitraje caprichoso y azaroso del destino que determinó mi nacimiento y con ello impuso en mi identidad un elemento cultural, social y artificialmente construido como lo es la nacionalidad? ¿Existe el derecho universal para ser simplemente ciudadano del mundo? Todas estas preguntas son necesarias para develar quiénes somos en realidad, ya que nuestros gustos, deseos, personalidades, expectativas y posibilidades están profundamente supeditadas a la geografía, historia y sociedad que nos vio nacer. Es necesario un pasaporte global para ser ciudadano del mundo, un pasaporte que nos reconozca a todos como hijos de la misma tierra y con el mismo derecho a habitarla, gozarla y disfrutarla sin importar el lugar donde hayamos nacido. ¡Todos tenemos derecho a nuestro PASAPORTE GLOBAL, todos somos ciudadanos del mundo! Te invito a darle la vuelta al mundo, a viajar juntos por los cinco continentes y conocer pequeñas historias de vida, lucha, amor y sed de justicia. ¡Pasaporte global en mano y bienvenidos a bordo!

[Cómpralo y empieza a leer](#)



14 relatos de oro

Mancilla Carrasco, Juan

9788418855641

[Cómpralo y empieza a leer](#)

Catorce relatos de oro es una invitación al lector a realizar un viaje para evadirse de la realidad. Un viaje que consta de catorce historias independientes con tramas y estilos diferentes, pero teniendo como nexo la intimidad con la que están contadas. En su debut, Juan Mancilla deja la impronta de su personalidad en esta antología que reúne una serie de relatos que van desde los clásicos cuentos hasta alguno más discursivo e inconformista, siempre desde un lado poético, filosófico, íntimo y personal. Un viaje a través de las palabras que quedará en el recuerdo y te acompañará para toda la vida.

[Cómpralo y empieza a leer](#)