

Inteligencia artificial avanzada

Raúl Benítez

Gerard Escudero

Samir Kanaan

PID_00174137



Universitat Oberta
de Catalunya

www.uoc.edu

Los textos e imágenes publicados en esta obra están sujetos –excepto que se indique lo contrario– a una licencia de Reconocimiento-NoComercial-SinObraDerivada (BY-NC-ND) v.3.0 España de Creative Commons. Podéis copiarlos, distribuirlos y transmitirlos públicamente siempre que citéis el autor y la fuente (FUOC. Fundació per a la Universitat Oberta de Catalunya), no hagáis un uso comercial y no hagáis una obra derivada. La licencia completa se puede consultar en <http://creativecommons.org/licenses/by-nc-nd/3.0/es/legalcode.es>

Índice

Introducción	7
1. Introducción a la inteligencia artificial (IA)	9
1.1. Neuronas y transistores	9
1.2. Breve historia de la IA	12
1.3. Ámbitos de aplicación de la inteligencia artificial	15
2. Recomendadores y agrupamientos	19
2.1. Métricas y medidas de similitud	20
2.1.1. Ejemplo de aplicación	20
2.1.2. Distancia euclídea	21
2.1.3. Correlación de Pearson	23
2.1.4. Procesamiento de datos reales	25
2.1.5. Conclusiones	25
2.2. Recomendadores basados en memoria	26
2.2.1. Conceptos generales	26
2.2.2. Aproximaciones simples	26
2.2.3. Recomendación ponderada	27
2.2.4. Conclusiones	28
2.3. Algoritmos de agrupamiento (<i>clustering</i>)	29
2.3.1. Ejemplo de aplicación	29
2.3.2. Conceptos generales	31
2.3.3. Agrupamiento jerárquico. Dendrogramas	32
2.3.4. k-medios (<i>k-means</i>)	35
2.3.5. c-medios difuso (<i>Fuzzy c-means</i>)	37
2.3.6. Agrupamiento espectral (<i>spectral clustering</i>)	37
2.3.7. Recomendadores basados en modelos	39
3. Extracción y selección de atributos	40
3.1. Técnicas de factorización matricial	42
3.1.1. Descomposición en valores singulares (SVD)	43
3.1.2. Análisis de componentes principales (PCA)	47
3.1.3. Análisis de componentes independientes (ICA)	61
3.1.4. Factorización de matrices no-negativas (NMF)	72
3.2. Discriminación de datos en clases	79
3.2.1. Análisis de discriminantes lineales (LDA)	79
3.3. Visualización de datos multidimensionales	86
3.3.1. Escalamiento multidimensional (MDS)	86
4. Clasificación	93
4.1. Introducción	93

4.1.1.	Categorización de textos	94
4.1.2.	Aprendizaje automático para clasificación	96
4.1.3.	Tipología de algoritmos para clasificación	97
4.2.	Métodos basados en modelos probabilísticos	98
4.2.1.	Naïve Bayes	98
4.2.2.	Máxima entropía	102
4.3.	Métodos basados en distancias	104
4.3.1.	kNN	105
4.3.2.	Clasificador lineal basado en distancias	108
4.3.3.	<i>Clustering</i> dentro de clases	110
4.4.	Métodos basados en reglas	112
4.4.1.	Árboles de decisión	112
4.4.2.	AdaBoost	120
4.5.	Clasificadores lineales y métodos basados en kernels	125
4.5.1.	Clasificador lineal basado en producto escalar	125
4.5.2.	Clasificador lineal con kernel	130
4.5.3.	Kernels para tratamiento de textos	135
4.5.4.	Máquinas de vectores de soporte	143
4.6.	Protocolos de test	157
4.6.1.	Protocolos de validación	157
4.6.2.	Medidas de evaluación	158
4.6.3.	Tests estadísticos	160
4.6.4.	Comparativa de clasificadores	161
5.	Optimización	164
5.1.	Introducción	164
5.1.1.	Tipología de los métodos de optimización	167
5.1.2.	Características de los metaheurísticos de optimización	167
5.2.	Optimización mediante multiplicadores de Lagrange	168
5.2.1.	Descripción del método	169
5.2.2.	Ejemplo de aplicación	170
5.2.3.	Análisis del método	171
5.3.	Recocción simulada	171
5.3.1.	Descripción del método	172
5.3.2.	Ejemplo de aplicación	173
5.3.3.	Análisis del método	175
5.3.4.	Código fuente en Python	176
5.4.	Algoritmos genéticos	178
5.4.1.	Descripción del método	179
5.4.2.	Ampliaciones y mejoras	181
5.4.3.	Ejemplo de aplicación	181
5.4.4.	Análisis del método	182
5.4.5.	Código fuente en Python	183
5.5.	Colonias de hormigas	185
5.5.1.	Descripción del método	186
5.5.2.	Ejemplo de aplicación	187
5.5.3.	Análisis del método	190

5.5.4.	Código fuente en Python.....	191
5.6.	Optimización con enjambres de partículas.....	193
5.6.1.	Descripción del método	194
5.6.2.	Ejemplo de aplicación	197
5.6.3.	Análisis del método	197
5.6.4.	Código fuente en Python.....	199
5.7.	Búsqueda tabú	201
5.7.1.	Descripción del método	201
5.7.2.	Ejemplo de aplicación	202
5.7.3.	Análisis del método	204
5.7.4.	Código fuente en Python.....	205
6.	Anexo: conceptos básicos de estadística	208
	Actividades	211
	Bibliografía	213

Introducción

Este módulo está organizado de la forma siguiente: los métodos de búsqueda y optimización se describen en el apartado 2, donde se detallarán las técnicas de extracción de información de bases de datos que contengan información semántica, como por ejemplo web de noticias o las conversaciones entre diversos miembros de una red social.

Las técnicas de caracterización de datos se estudiarán en el apartado 3, describiendo las técnicas principales basadas en descomposición de los datos en modos principales. En el apartado 3 también se estudiarán las técnicas de extracción de características y un método de visualización de datos multidimensionales.

Los algoritmos de clasificación de datos se presentan en el apartado 4, en el que se estudiarán los principales métodos de clasificación y reconocimiento de patrones.

En el apartado 5 se explican algunas técnicas avanzadas de inteligencia evolutiva, algoritmos que utilizan reglas heurísticas inspiradas en el funcionamiento evolutivo de los sistemas biológicos.

1. Introducción a la inteligencia artificial (IA)

1.1. Neuronas y transistores

Empezaremos planteando la pregunta filosófica fundamental, y así podremos dedicar nuestros esfuerzos a aspectos de carácter científico-técnico.

¿Es físicamente posible que una máquina presente capacidad de abstracción similar a la inteligencia humana?

Para responder esta pregunta, hay que tener en cuenta que el cerebro humano es el sistema de reconocimiento de patrones más complejo y eficiente que conocemos. Los humanos realizamos acciones tan sorprendentes como identificar a un conocido entre la multitud o reconocer de oído el solista de un concierto para violín. En el cerebro humano, las funciones cognitivas se realizan mediante la activación coordinada de unas 90.000.000.000 células nerviosas interconectadas mediante enlaces sinápticos. La activación neuronal sigue complejos procesos biofísicos que garantizan un funcionamiento robusto y adaptativo, y nos permite realizar funciones como el procesamiento de información sensorial, la regulación fisiológica de los órganos, el lenguaje o la abstracción matemática.

La neurociencia actual todavía no aporta una descripción detallada sobre cómo la activación individual de las neuronas da lugar a la formación de representaciones simbólicas abstractas. Lo que sí parece claro es que en la mayoría de procesos cognitivos existe una separación de escalas entre la dinámica a nivel neuronal y la aparición de actividad mental abstracta. Esta separación de escalas supone la ruptura del vínculo existente entre el hardware (neuronas) y el software de nuestro cerebro (operaciones abstractas, estados mentales), y constituye la hipótesis de partida para que los símbolos abstractos puedan ser manipulados por sistemas artificiales que no requieran un substrato fisiológico natural. La posibilidad de manipular expresiones lógicas y esquemas abstractos mediante sistemas artificiales es la que permite la existencia de lo que conocemos como *inteligencia artificial*.

Por supuesto, el cerebro no es el único sistema físico en el que se produce una separación de la dinámica a diferentes escalas. Esta característica también se observa en otros muchos sistemas complejos que presentan fenómenos de autoorganización no lineal. De la misma forma que es posible describir las co-

Lecturas complementarias

C. Koch (1999). *Biophysics of Computation: Information Processing in Single Neurons*. USA: Oxford University Press.

rrientes oceánicas sin necesidad de referirse al movimiento microscópico de las moléculas de agua, el pensamiento abstracto puede analizarse sin necesidad de referirse la activación eléctrica cerebral a nivel neuronal.

En cualquier caso, una de las cuestiones de mayor relevancia y aún no resueltas de la neurociencia actual es saber si existen procesos mentales -como la conciencia, la empatía o la creatividad-, que estén intrínsecamente ligados a la realidad biofísica del sistema nervioso humano y sean por tanto inaccesibles a un sistema artificial.

Otro aspecto importante en el funcionamiento del cerebro humano es el papel que tiene la experiencia y el aprendizaje. El cerebro humano actual no es solo resultado de una evolución biológica basada en alteraciones genéticas, sino también del conjunto de técnicas y conocimientos que la humanidad ha ido acumulando con el tiempo. Aspectos como la cultura o el lenguaje, transmitidas de generación en generación, también determinan la forma en la que se establecen patrones de activación neuronal en nuestro cerebro, y por lo tanto, contribuyen a la emergencia de procesos de abstracción en los que se basan áreas como las matemáticas o la literatura. A nivel biológico, existen diversos mecanismos que permiten la existencia de procesos de activación neuronal dependientes de la experiencia previa y del entrenamiento. El mecanismo principal es conocido como *plasticidad sináptica*, un fenómeno por el que las conexiones sinápticas entre neuronas modulan su intensidad en función de la actividad que hayan experimentado previamente. De esta forma, cuanto más veces se active un cierto canal de activación neuronal, más fácil resultará activarlo en el futuro e integrarlo a nuevos procesos cognitivos de mayor complejidad.

La plasticidad neuronal es la base de la mayoría de procesos de aprendizaje y memoria. A este paradigma se le conoce como *aprendizaje reforzado*, ya que la actividad sináptica se refuerza en función del número de veces que se establece una conexión entre neuronas. Esta regla -que relaciona la actividad neuronal con la función cognitiva-, se le conoce como *regla de Hebb* por los trabajos del neuropsicólogo canadiense Donald O. Hebb publicados en su libro de 1949 *The organization of behavior*. Algunas técnicas de inteligencia artificial como los métodos de *aprendizaje supervisado* también se basan en reglas similares que permiten modificar de forma adaptativa la forma en que el sistema artificial procesa la información.

La inteligencia artificial (IA) es una disciplina académica relacionada con la teoría de la computación cuyo objetivo es emular algunas de las facultades intelectuales humanas en sistemas artificiales. Con inteligencia humana nos referimos típicamente a procesos de percepción sensorial (visión, audición, etc.) y a sus consiguientes procesos de reconocimiento de patrones, por lo que las

aplicaciones más habituales de la IA son el tratamiento de datos y la identificación de sistemas. Eso no excluye que la IA, desde sus inicios en la década del 1960, haya resuelto problemas de carácter más abstracto como la demostración de teoremas matemáticos, la adquisición del lenguaje, el jugar a ajedrez o la traducción automática de textos. El diseño de un sistema de inteligencia artificial normalmente requiere la utilización de herramientas de disciplinas muy diferentes como el cálculo numérico, la estadística, la informática, el procesado de señales, el control automático, la robótica o la neurociencia. Por este motivo, pese a que la inteligencia artificial se considera una rama de la informática teórica, es una disciplina en la que contribuyen de forma activa numerosos científicos, técnicos y matemáticos. En algunos aspectos, además, se beneficia de investigaciones en áreas tan diversas como la psicología, la sociología o la filosofía.

Pese a que se han producido numerosos avances en el campo de la neurociencia desde el descubrimiento de la neurona por Santiago Ramón y Cajal a finales del siglo XIX, las tecnologías actuales están muy lejos de poder diseñar y fabricar sistemas artificiales de la complejidad del cerebro humano. De hecho, a día de hoy estamos lejos de reproducir de forma sintética las propiedades electroquímicas de la membrana celular de una sola neurona. Pero como hemos comentado anteriormente, la manipulación de conceptos y expresiones abstractas no está supeditada a la existencia de un sistema biológico de computación. En definitiva, un ordenador no es más que una máquina que procesa representaciones abstractas siguiendo unas reglas predefinidas.

Avances de la microelectrónica

En la actualidad, la única tecnología que permite implementar sistemas de inteligencia artificial son los sistemas electrónicos basados en dispositivos de estado sólido como el transistor. En efecto, gracias a los grandes avances de la microelectrónica desde los años setenta, los ordenadores actuales disponen de una gran capacidad de cálculo que permite la implementación de sistemas avanzados de tratamiento de datos y reconocimiento de patrones. Los sistemas digitales basados en transistor constituyen una tecnología rápida, robusta y de tamaño reducido que permite la ejecución secuencial de operaciones aritmético-lógicas. En muchas aplicaciones concretas, los sistemas artificiales pueden llegar a presentar un rendimiento notablemente mejor que el del propio cerebro humano. Este es el caso, por ejemplo, en aquellas situaciones que requieran gestionar grandes cantidades de datos o que exijan una rápida ejecución de cálculos matemáticos.

Un sistema de inteligencia artificial requiere de una secuencia finita de instrucciones que especifique las diferentes acciones que ejecuta la computadora para resolver un determinado problema. Esta secuencia de instrucciones constituye la *estructura algorítmica* del sistema de inteligencia artificial.

Se conoce como *método efectivo o algoritmo* al procedimiento para encontrar la solución a un problema mediante la reducción del mismo a un conjunto de reglas.

En ocasiones, los sistemas de IA resuelven problemas de forma *heurística* mediante un procedimiento de ensayo y error que incorpora información relevante basada en conocimientos previos. Cuando un mismo problema puede resolverse mediante sistemas naturales (cerebro) o artificiales (computadora), los algoritmos que sigue cada implementación suelen ser completamente diferentes puesto que el conjunto de instrucciones elementales de cada sistema son también diferentes. El cerebro procesa la información mediante la activación coordinada de redes de neuronas en áreas especializadas (cortex visual, cortex motor, etc.). En el sistema nervioso, los datos se transmiten y reciben codificados en variables como la frecuencia de activación de las neuronas o los intervalos en los que se generan los potenciales de acción neuronales. El elevado número de neuronas que intervienen en un proceso de computación natural hace que las fluctuaciones fisiológicas tengan un papel relevante y que los procesos computacionales se realicen de forma estadística mediante la actividad promediada en subconjuntos de neuronas.

En un sistema IA, en cambio, las instrucciones básicas son las propias de una computadora, es decir operaciones aritmético-lógicas, de lectura/escritura de registros y de control de flujo secuencial. La tabla 1 describe las diferencias fundamentales entre sistemas de inteligencia artificial y natural en las escalas más relevantes.

Tabla 1. Comparación entre inteligencia natural y artificial a diferentes niveles

Nivel	Natural	Artificial
Abstracción	Representación y manipulación de objetos abstractos	Representación y manipulación de objetos abstractos
Computacional	Activación coordinada de áreas cerebrales	Algoritmo / procedimiento efectivo
Programación	Conexiones sinápticas plasticidad	Secuencia de operaciones aritmético-lógicas
Arquitectura	Redes excitatorias e inhibitorias	CPU + memoria
Hardware	Neurona	Transistor

La idea principal es que, a pesar de las enormes diferencias entre sistemas naturales y artificiales, a un cierto nivel de abstracción ambos pueden describirse como sistemas de procesamiento de objetos abstractos mediante un conjunto de reglas.

1.2. Breve historia de la IA

El nacimiento de la IA como disciplina de investigación se remonta a 1956, durante una conferencia sobre informática teórica que tuvo lugar en el Dartmouth College (Estados Unidos). A esa conferencia asistieron algunos de los científicos que posteriormente se encargaron de desarrollar la disciplina en

diferentes ámbitos y de dotarla de una estructura teórica y computacional apropiada. Entre los asistentes estaban John McCarthy, Marvin Minsky, Allen Newell y Herbert Simon. En la conferencia, A. Newell y H. Simon presentaron un trabajo sobre demostración automática de teoremas al que denominaron *Logic Theorist*. El *Logic Theorist* fue el primer programa de ordenador que emulaba características propias del cerebro humano, por lo que es considerado el primer sistema de inteligencia artificial de la historia. El sistema era capaz de demostrar gran parte de los teoremas sobre lógica matemática que se presentaban en los tres volúmenes de los *Principia Mathematica* de Alfred N. Whitehead y Bertrand Russell (1910-1913).

Minsky y McCarthy fundaron más tarde el laboratorio de inteligencia artificial del Massachusetts Institute of Technology (MIT), uno de los grupos pioneros en el ámbito. La actividad de los años cincuenta es consecuencia de trabajos teóricos de investigadores anteriores como Charles Babbage (autor de la Máquina analítica, 1842), Kurt Gödel (teorema de incompletitud, 1930), Alan Turing (máquina universal, 1936), Norbert Wiener (cibernética, 1943) y John von Neumann (arquitectura del computador, 1950). La arquitectura de von Neumann consta de una unidad central de proceso (CPU) y de un sistema de almacenamiento de datos (memoria), y fue utilizada en 1954 por RAND Corporation para construir JOHNIAC (John v. Neumann Numerical Integrator and Automatic Computer), una de las primeras computadoras en las que más tarde se implementaron sistemas de inteligencia artificial como el *Logic Theorist* de Newell y Simon.

En 1954 también apareció el IBM 704, la primera computadora de producción en cadena, y con ella se desarrollaron numerosos lenguajes de programación específicamente diseñados para implementar sistemas de inteligencia artificial como el LISP. Junto con estos avances, se produjeron los primeros intentos para determinar la presencia de comportamiento inteligente en una máquina. El más relevante desde el punto de vista histórico fue propuesto por Alan Turing en un artículo de 1950 publicado en la revista *Mind* y titulado *Computing Machinery and Intelligence*.

En este trabajo se propone un test de inteligencia para máquinas según el cual una máquina presentaría un comportamiento inteligente en la medida en que fuese capaz de mantener una conversación con un humano sin que otra persona pueda distinguir quién es el humano y quién el ordenador. Aunque el *test de Turing* ha sufrido innumerables adaptaciones, correcciones y controversias, pone de manifiesto los primeros intentos de alcanzar una definición objetiva de la inteligencia.

En este contexto, es de especial relevancia el *Teorema de incompletitud de Gödel* de 1931, un conjunto de teoremas de lógica matemática que establecen las li-

mitaciones inherentes a un sistema basado en reglas y procedimientos lógicos (como lo son todos los sistemas de IA).

Tras los primeros trabajos en IA de los años cincuenta, en la década de los sesenta se produjo un gran esfuerzo de formalización matemática de los métodos utilizados por los sistemas de IA.

Los años setenta, en parte como respuesta al test de Turing, se produjo el nacimiento de un área conocida como *procesado del lenguaje natural* (NLP, Natural Language Processing), una disciplina dedicada a sistemas artificiales capaces de generar frases inteligentes y de mantener conversaciones con humanos. El NLP ha dado lugar a diversas áreas de investigación en el campo de la lingüística computacional, incluyendo aspectos como la desambiguación semántica o la comunicación con datos incompletos o erróneos. A pesar de los grandes avances en este ámbito, sigue sin existir una máquina que pueda pasar el test de Turing tal y como se planteó en el artículo original. Esto no es tanto debido a un fracaso de la IA como a que los intereses del área se han ido redefiniendo a lo largo de la historia. En 1990, el controvertido empresario Hugh Loebner y el Cambridge Center for Behavioral Studies instauraron el *premio Loebner*, un concurso anual ciertamente heterodoxo en el que se premia al sistema artificial que mantenga una conversación más indistinguible de la de un humano. Hoy en día, la comunidad científica considera que la inteligencia artificial debe enfocarse desde una perspectiva diferente a la que se tenía en los años cincuenta, pero iniciativas como la de Loebner expresan el impacto sociológico que sigue teniendo la IA en la sociedad actual.

En los años ochenta empezaron a desarrollarse las primeras aplicaciones comerciales de la IA, fundamentalmente dirigidas a problemas de producción, control de procesos o contabilidad. Con estas aplicaciones aparecieron los primeros sistemas expertos, que permitían realizar tareas de diagnóstico y toma de decisiones a partir de información aportada por profesionales expertos. En torno a 1990, IBM construyó el ordenador ajedrecista Deep Blue, capaz de plantarle cara a un gran maestro de ajedrez utilizando algoritmos de búsqueda y análisis que le permitían valorar cientos de miles de posiciones por segundo.

Más allá del intento de diseñar robots humanoides y sistemas que rivalicen con el cerebro humano en funcionalidad y rendimiento, el interés hoy en día es diseñar e implementar sistemas que permitan analizar grandes cantidades de datos de forma rápida y eficiente. En la actualidad, cada persona genera y recibe a diario una gran cantidad de información no sólo a través de los canales clásicos (conversación, carta, televisión) sino mediante nuevos medios que nos permiten contactar con más personas y transmitir un mayor número de datos en las comunicaciones (Internet, fotografía digital, telefonía móvil). Aunque el cerebro humano es capaz de reconocer patrones y establecer relaciones útiles entre ellos de forma excepcionalmente eficaz, es ciertamente limitado cuando la cantidad de datos resulta excesiva. Un fenómeno similar

HAL 9000

El impacto social de la IA durante la década de los sesenta se pone de manifiesto en la película "2001: A Space Odyssey", dirigida en 1968 por Stanley Kubrick y basada en una novela homónima de ciencia ficción de Arthur C. Clarke. El protagonista principal de la película es HAL 9000 (Heuristically programmed ALgorithmic computer), un ordenador dotado de un avanzado sistema de inteligencia artificial que es capaz de realizar tareas como mantener una conversación, reconocimiento de voz, lectura de labios, jugar a ajedrez e incluso manifestar un cierto grado de sensibilidad artística.

Aplicaciones de la IA

En la actualidad, la IA se ha consolidado como una disciplina que permite diseñar aplicaciones de gran utilidad práctica en numerosos campos. Actualmente, existe una enorme lista de ámbitos de conocimiento en los que se utilizan sistemas de IA, entre los que son de especial relevancia la minería de datos, el diagnóstico médico, la robótica, la visión artificial, el análisis de datos bursátiles o la planificación y logística.

ocurre en el ámbito empresarial, donde cada día es más necesario barajar cantidades ingentes de información para poder tomar decisiones. La aplicación de técnicas de IA a los negocios ha dado lugar a ámbitos de reciente implantación como la inteligencia empresarial, *business intelligence* o a la minería de datos, *data mining*. En efecto, hoy más que nunca la información está codificada en masas ingentes de datos, de forma que en muchos ámbitos se hace necesario extraer la información relevante de grandes conjuntos de datos antes de proceder a un análisis detallado.

En resumen, hoy en día el objetivo principal de la inteligencia actual es el tratamiento y análisis de datos.

Algunas veces nos interesará caracterizar los datos de forma simplificada para poder realizar un análisis en un espacio de dimensión reducida o para visualizar los datos de forma más eficiente. Por ejemplo, puede ser interesante saber qué subconjunto de índices bursátiles internacionales son los más relevantes para seguir la dinámica de un cierto producto o mercado emergente. En otras ocasiones, el objetivo será identificar patrones en los datos para poder clasificar las observaciones en diferentes clases que resulten útiles para tomar decisiones respecto un determinado problema. Un ejemplo de este segundo tipo de aplicación sería el análisis de imágenes médicas para clasificar a pacientes según diferentes patologías y así ayudar al médico en su diagnóstico. Por último, en muchos casos se hace necesario realizar búsquedas entre una gran cantidad de datos u optimizar una determinada función de coste, por lo que también será necesario conocer métodos de búsqueda y optimización. En esta clase de problemas encontramos, por ejemplo, el diseño de los horarios de una estación de trenes de forma que se minimice el tiempo de espera y el número de andenes utilizados.

1.3. Ámbitos de aplicación de la inteligencia artificial

Las aplicaciones más frecuentes de la inteligencia artificial incluyen campos como la robótica, el análisis de imágenes o el tratamiento automático de textos. En *robótica*, uno de los campos de investigación actual con mayor proyección es el del *aprendizaje adaptativo*, en el que un sistema robotizado explora diferentes configuraciones con el objetivo de realizar un movimiento complejo (caminar, agarrar un objeto, realizar una trayectoria, jugar a golf, etc.). El objetivo podría ser, por ejemplo, que un robot cuadrúpedo se levante y ande de forma autónoma, de forma que siga un proceso de exploración y aprendizaje similar al que realiza un recién nacido durante los primeros meses de vida. Un sistema de IA en este caso se encargaría de explorar diferentes movimientos de forma aleatoria, midiendo las variables de cada articulación y comprobando en cada momento el grado de éxito alcanzado por cada secuencia de

acciones (altura del centro de masas, desplazamiento horizontal, fluctuaciones en la posición vertical, velocidad de desplazamiento, etc.). El sistema de IA modularía la ejecución de las diferentes acciones, incrementando la probabilidad de aquellas que presenten un mejor rendimiento y restringiendo las que no comporten una mejora de la función objetivo. Un área afín a la robótica es la de las *interfaces cerebro-computadora* (BCI, Brain-computer Interfaces), sistemas artificiales que interactúan con el sistema nervioso mediante señales neurofisiológicas con el objetivo asistir a personas discapacitadas durante la ejecución de determinadas tareas motoras.

Dentro del campo de la IA, una de las ramas con mayor proyección son los denominados *sistemas expertos*, en los que el objetivo es diseñar un sistema que permita analizar un conjunto de datos y realizar tareas típicamente asociadas a la figura de un profesional experto como el diagnóstico, la detección de fallos, la planificación o la toma de decisiones. Los datos con los que trabaja el sistema experto pueden ser de naturaleza muy diversa.

En un sistema de diagnóstico clínico, por ejemplo, se puede partir de imágenes radiológicas, de una serie temporal de la frecuencia del ritmo cardíaco de un paciente o de un conjunto de datos con valores extraídos de análisis de sangre o de orina. La utilización de todas las señales anteriores a la vez constituye un sistema de *fusión multimodal*, en el que el estado clínico del paciente se describe desde una perspectiva multiorgánica más completa. En los sistemas expertos se combina información extraída de datos con el conocimiento del sistema que aporta un experto especializado. A estos sistemas se les conoce como *sistemas basados en conocimiento* (KBS, *knowledge-based systems*), y permiten integrar reglas heurísticas y árboles de decisiones elaborados por una comunidad de expertos durante años de trabajo y experimentación. Estas reglas no pueden ser inferidas directamente de los datos observados, y son de gran utilidad en aplicaciones sobre diagnóstico y toma de decisiones.

Ejemplo

Un ejemplo de este tipo de técnicas serían el protocolo de diagnóstico que realiza un médico especialista a partir de un conjunto de pruebas, o los criterios que aplica un ingeniero de caminos para validar la resistencia mecánica de un puente.

La información aportada por humanos expertos es también necesaria para diseñar sistemas artificiales que jueguen a juegos como el ajedrez o el go.

Go

Go es un tradicional juego de mesa chino que se practica en un tablero reticular y tiene por objetivo ocupar con las fichas una región mayor que el adversario. El desarrollo de máquinas que jueguen a Go con humanos a un alto nivel es un ámbito en el que actualmente se dedican grandes esfuerzos desde el campo de la IA. En este juego, el gran número de posibles movimientos en cada jugada impide aplicar técnicas de búsqueda global, por lo que los sistemas inteligentes deben incorporar información sobre estrategias y tácticas aportada por jugadores humanos expertos.

El *análisis de textos* es otro ejemplo interesante en el que se desarrollan numerosos sistemas de IA. Aspectos como la traducción automática de textos han evolucionado de forma sorprendente durante los últimos años gracias a técnicas de IA. Hoy en día es fácil encontrar plataformas web gratuitas que permitan traducir textos entre más de 50 lenguas diferentes. Estos sistemas de traducción automática presentan resultados de notable calidad, y tienen en cuenta aspectos semánticos y contextuales antes de proponer una traducción del texto. El éxito de estas plataformas se debe en gran parte a sistemas de IA que utilizan enormes cantidades de información textual aportada por los usuarios del servicio. Sistemas similares permiten realizar búsquedas inteligentes en la web, de forma que los vínculos que se ofrecen tengan en cuenta las preferencias estadísticas propias y del resto de usuarios, o enviar publicidad de forma selectiva a partir de la información que aparece en el campo asunto de nuestro buzón de correo electrónico.

Otro ejemplo de sistema en el ámbito de la *ingeniería de procesos* es un sistema de detección de fallos en una planta compleja. Una planta industrial dispone de múltiples sensores distribuidos que permiten monitorizar el proceso de forma continua. En la fase de entrenamiento, el sistema aprende un conjunto de patrones dinámicos de los sensores que corresponden a situaciones en las que se produce un error en la planta. En el futuro, cuando se produce un error, el sistema de IA detecta su existencia de forma automática y es capaz de *diagnosticar* el origen del fallo comparando la respuesta de los sensores con las respuestas características de cada error. En una tercera fase, el sistema puede incluso tomar decisiones sobre qué acciones hay que realizar para resolver el problema de la forma más rápida y eficiente. Las aplicaciones industriales de la IA es un campo con una gran proyección en el que los sistemas van dirigidos a mejorar procesos de fabricación, control de calidad, logística o planificación de recursos.

En muchas ocasiones, los sistemas de IA constan de dos fases, una primera fase de *aprendizaje* y una segunda de *predicción*. En la fase de aprendizaje se aporta un conjunto de datos representativo de aquellas situaciones que se desea analizar, de forma que el sistema IA aprende las características fundamentales de los datos y es capaz de *generalizar* su estructura. Dicha generalización no es más que la construcción de un modelo de los datos que permita realizar una predicción acertada a partir de nuevas observaciones.

Reconocimiento de caras

Consideremos el ejemplo de un sistema automático para el reconocimiento de caras. Al sistema se le proporcionan 100 imágenes faciales de 10 personas diferentes, 10 caras por persona. Las imágenes son tomadas en diferentes condiciones (diferente expresión facial, ropa, iluminación, exposición, fondo, etc.), de forma que sean representativas de las características faciales de cada persona. El sistema identifica las características principales de cada una de las fotos y es capaz de agruparlas en un determinado número de grupos (presumiblemente 10, excepto si se han incluido gemelos monocigóticos). Entre las características utilizadas pueden aparecer, entre otras, el color de los ojos, el grosor de los labios o el perímetro de la cabeza. En la fase de predicción, se parte de una imagen de uno de los 10 individuos que no haya sido incluida en el conjunto de datos de entrenamiento. El sistema calcula las características faciales del nuevo individuo, y debe

ser capaz de identificar esa imagen como perteneciente a uno de los grupos definidos durante el proceso de aprendizaje (presumiblemente al grupo del individuo correcto, a no ser que la nueva imagen no tenga apenas rasgos comunes con las 10 imágenes de entrenamiento). Si la nueva imagen es identificada de forma correcta, diremos que el sistema de IA acierta en su predicción, mientras que cuando el sistema asigna la imagen a otro grupo de forma errónea, se produce un error de clasificación.

Tabla 2. Principales ámbitos de aplicación de los sistemas de inteligencia artificial

Área	Aplicaciones
Medicina	Ayuda al diagnóstico Análisis de imágenes biomédicas Procesado de señales fisiológicas
Ingeniería	Organización de la producción Optimización de procesos Cálculo de estructuras Planificación y logística Diagnóstico de fallos Toma de decisiones
Economía	Análisis financiero y bursátil Análisis de riesgos Estimación de precios en productos derivados Minería de datos Marketing y fidelización de clientes
Biología	Análisis de estructuras biológicas Genética médica y molecular
Informática	Procesado de lenguaje natural Criptografía Teoría de juegos Lingüística computacional
Robótica y automática	Sistemas adaptativos de rehabilitación Interfaces cerebro-computadora Sistemas de visión artificial Sistemas de navegación automática
Física y matemáticas	Demostración automática de teoremas Análisis cualitativo sistemas no-lineales Caracterización de sistemas complejos

2. Recomendadores y agrupamientos

Actualmente la gran mayoría de instrumentos de medida de cualquier tipo de magnitud (desde magnitudes físicas como longitud, temperatura, tiempo hasta magnitudes de comportamiento como patrones de búsqueda y navegación en la web y preferencias de compra *online*, pasando por herramientas comunes como las cámaras digitales) son capaces de volcar sus mediciones a algún formato digital; de esa forma todos esos datos pueden estar fácilmente disponibles para su tratamiento.

En este momento el problema no es disponer de datos, pues se dispone en gran abundancia de ellos; el reto es conseguir extraer información a partir de los datos, o sea, darles un sentido y extraer conclusiones útiles de ellos. Esta tarea se conoce por el nombre de **minería de datos** (*data mining*).

Uno de los principales retos en el procesamiento de datos es el de integrar los procedentes de múltiples fuentes, para así dotar de diferentes perspectivas al conjunto de estos, lo que permite extraer información más rica. Esta tendencia se da en casi todas las áreas: recopilación de la actividad de los usuarios en un sitio web; integración de sensores de temperatura, presión atmosférica y viento en el análisis meteorológico; uso de diferentes datos financieros y bursátiles en la previsión de inversiones; entre otros ejemplos.

La tarea de integración de múltiples fuentes de datos recibe el nombre de **filtrado colaborativo** (*collaborative filtering*).

De la misma forma que en las ciencias experimentales es fundamental utilizar un instrumental adecuado y unas unidades consistentes, un aspecto clave en cualquier tarea de procesamiento de datos es el uso de **métricas**, o sea medidas de distancia, adecuadas para el tipo de datos que se está tratando.

Una de las aplicaciones en las que se centra este módulo es la de los **recomendadores**. Un recomendador es un sistema que recoge y analiza las preferencias de los usuarios, generalmente en algún sitio web (comercios, redes sociales, sitios de emisión o selección de música o películas, etc.). La premisa básica de

Ved también

En el subapartado 2.1 de este módulo se estudian algunas de las métricas más habituales.

Ved también

En el subapartado 2.2 se describen algunas estrategias sencillas para construir recomendadores.

los recomendadores es que usuarios con actividad o gustos similares continuarán compartiendo preferencias en el futuro. Al recomendar a un usuario productos o actividades que otros usuarios con gustos similares han elegido previamente el grado de acierto acostumbra a ser más elevado que si las recomendaciones se basan en tendencias generales, sin personalizar.

La tarea de encontrar a los usuarios más afines y utilizar esta información para predecir sus preferencias puede inscribirse en una tarea más general que recibe el nombre de **agrupamiento** (*clustering*), y que consiste en encontrar la subdivisión óptima de un conjunto de datos, de forma que los datos similares pertenezcan al mismo grupo.

2.1. Métricas y medidas de similitud

Una **métrica** es una función que calcula la distancia entre dos elementos y que por tanto se utiliza para medir cuán diferentes son. Existen varias formas de medir la distancia entre dos elementos, y elegir la métrica adecuada para cada problema es un paso crucial para obtener buenos resultados en cualquier aplicación de minería de datos.

En este subapartado utilizaremos la **distancia euclídea** y estudiaremos una medida de similitud habitual, la **correlación de Pearson**.

Por último, una forma habitual y segura* de convertir una función de distancia d en una función de similitud s es la siguiente:

$$s(P,Q) = \frac{1}{1 + d(P,Q)} \quad (1)$$

2.1.1. Ejemplo de aplicación

En un sitio web de visualización de películas a la carta se recoge la valoración de cada usuario sobre las películas que va viendo, con el objetivo de poder proponer a los usuarios las películas que más se adapten a sus gustos. Tras ver una película, un usuario ha de dar una valoración entre 1 y 5, donde las valoraciones más bajas corresponden a películas que han disgustado al usuario, y las más alta a las que le han gustado más. Se desea descubrir similitudes entre usuarios de manera que a cada usuario se le propongan las películas que más han gustado a los usuarios con gustos más parecidos al suyo.

Para poner en práctica estas pruebas se utilizarán los conjuntos de datos disponibles en <http://www.grouplens.org/node/73>, concretamente el conjunto de 100k

Ved también

En el subapartado 2.3 se presentan los métodos de agrupamiento más importantes.

A menudo se utilizan funciones que miden la similitud entre dos elementos en lugar de su distancia.

Ved también

Las distancias y similitudes se utilizan en gran cantidad de métodos, por lo que en este módulo se presentan otras métricas como la distancia de Hamming en el subapartado 4.3.1 y la información mutua en el subapartado 3.1.3.

* Segura porque no incurre en divisiones por cero si la distancia es cero.

valoraciones (fichero *ml-data_0.zip*), que contiene 100.000 valoraciones (del 1 al 5) de 1.682 películas realizadas por 943 usuarios. Si bien el conjunto de datos contiene 23 ficheros, en este módulo sólo se utilizará el fichero *u.data*, que es el que contiene las valoraciones de los usuarios. En la tabla 3 se muestra un ejemplo meramente ilustrativo de valoraciones.

Tabla 3. Valoraciones de 8 películas (ejemplo)

IdUsuario/IdPelícula	1	2	3	4	5	6	7	8
1		3	1		4	3	5	
2	4	1	3		5			2
3	2	1		5				1
4	3		2			5		4

Los datos se presentan en forma matricial por legibilidad y concisión.

En el fichero *u.data* cada valoración aparece en una fila, y las columnas son IdUsuario IdPelícula Valoración Fecha.

El fichero *u.item* contiene información sobre cada película, y el fichero *u.user* contiene información genérica sobre cada usuario (edad, sexo, etc.). Como se puede observar, los usuarios no tienen por qué valorar todas las películas, sino sólo las que han visto. La información de la tabla puede expresarse en Python mediante un diccionario en el que cada identificador de usuario es una clave, y el valor asociado a dicha clave es otro diccionario que asocia cada identificador de película a la puntuación asignada por ese usuario. El fragmento de código 2.1 muestra la representación correspondiente a la tabla 3.

Código 2.1: Diccionario de valoración de películas

```
1 valoraciones = {1: {2:3, 3:1, 5:4, 6:3, 7:5}, 2: {1:4, 2:1, 3:3, 5:5, 8:2},
2               3: {1:2, 2:1, 4:5, 8:1},      4: {1:3, 3:2, 6:5, 8:4}}
```

2.1.2. Distancia euclídea

La distancia euclídea de dos puntos $P = (p_1, p_2, \dots, p_n)$ y $Q = (q_1, q_2, \dots, q_n)$ en el espacio n -dimensional \mathbb{R}^n viene dada por la fórmula:

$$d(P, Q) = \sqrt{\sum_{i=1}^n (p_i - q_i)^2} \tag{2}$$

La distancia euclídea no es más que la generalización a n dimensiones del teorema de Pitágoras. Si las distancias en sí no son importantes, sino sólo su comparación, a menudo se utiliza la **distancia euclídea cuadrada**, es decir, sin la raíz cuadrada, pues la comparación entre distancias euclídeas cuadradas da los mismos resultados que entre las distancias euclídeas y puede resultar mucho más rápida de calcular, ya que la raíz cuadrada es una operación computacionalmente costosa.

La función que calcula la distancia euclídea entre las valoraciones de dos usuarios (según el formato visto en el fragmento de programa 2.1) en Python se muestra en el fragmento de programa 2.2, así como la función necesaria para convertir la distancia en una similitud.

Código 2.2: Distancia euclídea entre dos diccionarios

```

1 from math import sqrt
2
3 def distEuclidea(dic1, dic2):
4     # Calcular la suma de cuadrados de los elementos comunes
5     # a los dos diccionarios
6     suma2 = sum([pow(dic1[elem]-dic2[elem], 2)
7                 for elem in dic1 if elem in dic2])
8     return sqrt(suma2)
9
10 def similEuclidea(dic1, dic2):
11     return 1/(1+distEuclidea(dic1, dic2))

```

En la tabla 4 se muestran las similitudes entre las valoraciones de los usuarios utilizando la distancia euclídea. Como se puede comprobar, $s(P,P) = 1$ (ya que $d(P,P) = 0$) y $s(P,Q) = s(Q,P)$, o sea, la matriz de similitudes es simétrica.

Tabla 4. Similitud euclídea de las valoraciones

Id usuario	1	2	3	4
1	1,0	0,25	0,33	0,31
2	0,25	1,0	0,31	0,29
3	0,33	0,31	1,0	0,24
4	0,31	0,29	0,24	1,0

Supongamos que el usuario 4 desea que se le recomiende alguna película; el sistema, de momento, puede sugerirle que vea las películas que han gustado al usuario 1 (y no ha visto el 4), ya que es el usuario con mayor similitud; en el ejemplo, el usuario 4 podría elegir las películas 5 y 7, que son las que más han gustado al usuario 1 y aún no ha visto el 4.

Una limitación de la similitud (y la distancia) euclídea es que es muy sensible a la escala: si un usuario tiende a puntuar las películas que le gustan con un 4 y otro con un 5, habrá una cierta distancia entre ellos, en especial si el número de valoraciones es alto, aunque en el fondo ambos usuarios comparten gustos a pesar de utilizar las puntuaciones de formas diferentes.

Además, a mayor número de dimensiones (valoraciones, en este caso), la distancia euclídea tiende a ser mayor, lo que puede distorsionar los resultados. Por ejemplo, si dos usuarios comparten 2 valoraciones y hay una diferencia de 1 entre cada una de ellas, su distancia será $d = \sqrt{1^2 + 1^2} = 1,41$; sin embargo, si comparten 5 valoraciones con una diferencia de 1 entre ellas, su distancia será $d = \sqrt{1^2 + 1^2 + 1^2 + 1^2 + 1^2} = 2,24$; una distancia bastante mayor para unas valoraciones en apariencia similares. Cuantas más valoraciones compartan dos usuarios (aunque no sean muy diferentes), más alejados estarán, lo que parece contrario a la lógica.

Utilidad de la distancia euclídea

La distancia euclídea es una métrica útil en numerosas aplicaciones, en especial si las magnitudes son lineales y su escala es uniforme; además, es sencilla y rápida de calcular.

2.1.3. Correlación de Pearson

El coeficiente de correlación de Pearson es una medida de similitud entre dos variables que resuelve los problemas de la similitud euclídea. Se trata de una medida de cómo las dos variables, una frente a otra, se organizan en torno a una línea recta (línea de mejor ajuste), tal y como se puede ver en la figura 1. Cuanto más similares son las valoraciones de dos usuarios, más se parecerá su recta a la recta $y = x$, ya que las valoraciones serán de la forma (1,1), (3,3), (4,4), etc.

Figura 1. Correlación entre las valoraciones los usuarios 1 y 2 de la tabla 3

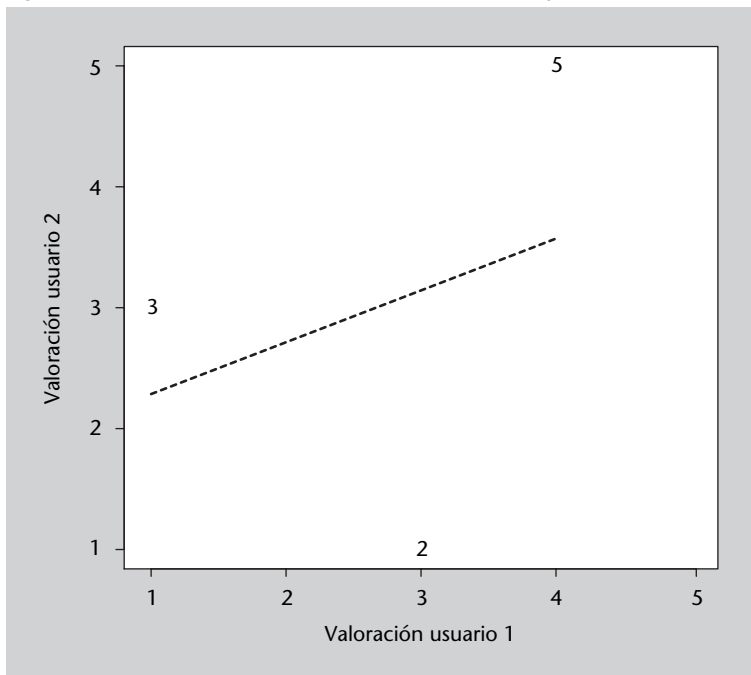


Figura 1

Diagrama de dispersión (scatter plot) de las valoraciones de dos usuarios: se representa la valoración de cada película en común tomando el eje x como la valoración de un usuario, y el eje y como la valoración del otro usuario. Los números en el diagrama corresponden a los identificadores de las películas comunes.

El coeficiente de correlación de Pearson (en este subapartado, simplemente *correlación*) está relacionado con la pendiente de la recta representada en la figura 1, y puede tomar un valor en el rango $[-1,1]$. Si su valor es 1 indica que las dos variables están perfectamente relacionadas; si es 0, no hay relación lineal entre ellas*; si es negativo es que existe una correlación negativa, en este caso que las valoraciones de un usuario son opuestas a las del otro.

* El coeficiente de correlación de Pearson sólo mide relaciones lineales; aunque valga 0, puede haber relaciones no lineales entre las dos variables.

El cálculo del coeficiente de correlación de Pearson sobre dos muestras de datos alineados (valoraciones de usuarios, en nuestro caso) x_i e y_i viene dado por la fórmula:

$$r_{xy} = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^n (x_i - \bar{x})^2} \sqrt{\sum_{i=1}^n (y_i - \bar{y})^2}} \quad (3)$$

donde \bar{x} es la media de los valores de x y \bar{y} la media de los valores de y .

Nótese que para efectuar el cálculo los datos deben estar alineados: en nuestro caso, sólo se deben tomar las valoraciones comunes a los dos usuarios. También hay que prever que el denominador pueda valer cero. Con todas esas consideraciones, en el código 2.3 se muestra la función en Python que calcula el coeficiente de correlación de Pearson de dos diccionarios de valoraciones.

Código 2.3: Coeficiente de Pearson entre dos diccionarios

```

1 def coefPearson(dic1, dic2):
2     # Obtener los elementos comunes a los dos diccionarios
3     comunes = [x for x in dic1 if x in dic2]
4     nComunes = float(len(comunes))
5
6     # Si no hay elementos comunes, se devuelve cero; si no
7     # se calcula el coeficiente
8     if nComunes==0:
9         return 0
10
11    # Cálculo de las medias de cada diccionario
12    media1 = sum([dic1[x] for x in comunes])/nComunes
13    media2 = sum([dic2[x] for x in comunes])/nComunes
14
15    # Cálculo del numerador y del denominador
16    num = sum([(dic1[x]-media1)*(dic2[x]-media2) for x in comunes])
17    den1 = sqrt(sum([pow(dic1[x]-media1, 2) for x in comunes]))
18    den2 = sqrt(sum([pow(dic2[x]-media2, 2) for x in comunes]))
19    den = den1*den2
20
21    # Cálculo del coeficiente si es posible, o devuelve 0
22    if den==0:
23        return 0
24
25    return num/den

```

Como se deduce de su definición, el coeficiente de correlación de Pearson es simétrico y vale 1 al calcularlo respecto a la misma variable. En la tabla 5 puede verse los valores correspondientes a las valoraciones de los usuarios de la tabla 3. En este ejemplo la correlación entre los usuarios 1 y 3 es 0 porque sólo tienen una película en común, con lo que la recta de ajuste no se puede definir.

Tabla 5. Coeficiente de Pearson de las valoraciones

Id usuario	1	2	3	4
1	1,0	0,33	0,0	1,0
2	0,33	1,0	0,95	-0,5
3	0,0	0,95	1,0	-1,0
4	1,0	-0,5	-1,0	1,0

También se observa que los usuarios 1 y 4 tienen una correlación de 1,0; eso es debido a que, independientemente de factores de escala, su tendencia a valorar las películas es igual. Nótese la diferencia con la similitud euclídea mostrada en la tabla 4, si bien el usuario más similar al 4 sigue siendo el 1 y por lo tanto las películas que podría elegir serían las mismas. Por otra parte, los usuarios 3 y 4 hacen valoraciones contrarias, por lo que su coeficiente es -1,0.

No obstante, sería conveniente disponer de más datos para obtener medidas más realistas.

El coeficiente de correlación de Pearson resulta útil como medida de similitud porque es independiente de los desplazamientos y escalas de los valores estudiados; para conseguir estas propiedades utilizando la similitud euclídea sería necesario normalizar los datos previamente a su análisis. Uno de sus principales inconvenientes en su uso en filtrado colaborativo es que requiere que haya al menos dos valores comunes para poder dar un resultado significativo; eso limita su aplicación en aquellos casos en que se desea sugerir productos similares a un cliente que ha hecho una única elección.

2.1.4. Procesamiento de datos reales

Hasta ahora se han utilizado las funciones de similitud sobre datos de prueba introducidos manualmente; sin embargo, es interesante probarlas con un volumen de datos apreciable para poder valorar mejor su comportamiento.

En el fragmento de código 2.4 se leen los datos del fichero *u.data* de la base de datos de valoraciones de películas *MovieLens* utilizada en este subapartado, y se produce como resultado un diccionario con la misma estructura que el dado como ejemplo en el código de programa 2.1.

Para utilizarlo basta con aplicar alguna de las funciones de similitud descritas anteriormente a cualquier par de usuarios (claves del diccionario principal), o bien escribir un programa que genere una tabla o diccionario con todas las correlaciones.

Código 2.4: Función que lee un fichero de valoraciones de *MovieLens*

```
1 def leeValoraciones(nomFich="u.data"):
2     lineas = [(l.strip()).split("\t")]
3     for l in (open(nomFich).readlines()):
4         diccio = {int(l[0]) : {} for l in lineas}
5
6     for l in lineas:
7         diccio[int(l[0])][int(l[1])] = int(l[2])
8     return diccio
```

2.1.5. Conclusiones

Las principales limitaciones de los procedimientos explicados en este subapartado son dos: primera, que no permiten sugerir productos directamente, sino sólo usuarios afines; esta limitación se resolverá en el siguiente subapartado. La segunda limitación estriba en que es necesario almacenar en memoria todas las valoraciones de los usuarios y recalcularlas todas cada vez que se incorpora una nueva valoración o usuario, lo que puede suponer un coste computacio-

nal alto en aplicaciones grandes; visto de otra manera, no se genera ninguna abstracción o modelo de los datos que permita trabajar con una versión reducida de los datos. Esta limitación se resolverá en el subapartado 2.3, que trata sobre los algoritmos de agrupamiento.

2.2. Recomendadores basados en memoria

En este subapartado se mejorará el sistema de recomendación de películas visto en el subapartado 2.1 para que, en lugar de sugerir usuarios con gustos similares, sugiera directamente las películas que pueden ser más interesantes para el usuario.

2.2.1. Conceptos generales

Se habla de recomendadores **basados en memoria** porque requieren mantener todos los datos disponibles para poder realizar una recomendación; además, cada vez que se introduce un nuevo dato o que se solicita una recomendación estos recomendadores han de efectuar sus operaciones sobre el conjunto completo de datos, lo que provoca que sean relativamente ineficientes, especialmente en aplicaciones con volúmenes de datos grandes. No obstante, tienen interés porque son muy sencillos conceptualmente y su programación es relativamente rápida.

Estos recomendadores están estrechamente relacionados con los clasificadores basados en distancias estudiados en el subapartado 4.3, pues emplean una metodología similar (por ejemplo buscar el elemento más cercano al que se está evaluando); la diferencia radica en el uso que se hace de esta información, pues en el caso de los clasificadores se toma un nuevo elemento y se decide a qué clase debe pertenecer, mientras que en el caso de los recomendadores se toma un elemento y se sugieren los elementos más semejantes, sin necesidad de realizar una clasificación.

En la práctica se trata de generar una valoración (*ranking*) de todos los datos o preferencias registradas y que tenga en cuenta las particularidades del solicitante (gustos del usuario, etc.).

2.2.2. Aproximaciones simples

La forma más sencilla de generar una valoración global de un conjunto de elementos es promediar su grado de valoración. En el ejemplo de las películas, si una película tiene cuatro valoraciones (3, 5, 4, 3), su valoración media será 3,75; las películas con mayor valoración global serán las sugeridas al usuario. En el código 2.5 se muestra la función que devuelve una lista ordenada (de

mayor a menor valoración) de películas, a partir del diccionario de valoraciones de los usuarios.

Código 2.5: Función de recomendación media global de películas

```

1 # Genera una lista ordenada de valoraciones globales a partir de
2 # un diccionario de valoraciones de usuarios, de la forma
3 # [(idPelícula, valoraciónGlobal)]
4 def listaValoracionesSimple(diccio):
5     # Diccionario auxiliar {idPelícula: [valoraciones]}
6     aux = {}
7     for valorUsuario in diccio.values():
8         for idPeli in valorUsuario:
9             if not aux.has_key(idPeli):
10                aux[idPeli] = []
11                aux[idPeli].append(valorUsuario[idPeli])
12
13     # Cálculo y ordenación de las valoraciones globales
14     media = lambda x: sum(x)/float(len(x))
15     result = [(p, media(aux[p])) for p in aux]
16     result.sort(key = lambda x: x[1], reverse=True)
17     return result

```

Obviamente es posible añadir un *if* para omitir de la lista las películas que ya ha visto el usuario, aunque no se ha incluido esa operación para simplificar el código fuente mostrado.

La principal limitación de este método radica en que no tiene en cuenta las preferencias individuales de cada usuario. No obstante, esta estrategia puede ser útil con usuarios nuevos, de los que no se conoce sus gustos. Una ventaja es que no depende de ninguna métrica para efectuar sus recomendaciones.

Una estrategia sencilla que tiene en cuenta las preferencias del usuario consiste en seleccionar las películas favoritas del usuario más semejante, que era lo que se dejaba hacer manualmente en el subapartado 2.1 tras sugerir el usuario más cercano. La desventaja de esta estrategia consiste en que está limitada a las películas que ha visto el usuario más cercano. Si no ha visto una película que muchos otros usuarios relativamente afines han visto y valorado positivamente, el sistema nunca sugerirá esa película a pesar de que podría tratarse de una buena recomendación. Esta estrategia es similar al método de clasificación del vecino más cercano (*1-Nearest-Neighbour*) que se estudia en el subapartado 4.3.1.

2.2.3. Recomendación ponderada

Es posible utilizar un método que reúna las ventajas de los dos métodos de recomendación simples que se acaban de exponer. La idea es sencilla: se trata de obtener una valoración global del conjunto de elementos (productos) pero en lugar de calcular la media aritmética de las valoraciones, hay que calcular la media de las valoraciones ponderada por la afinidad del usuario correspondiente. De esa manera, la valoración global de los productos está personalizada para cada usuario y, por tanto, debería resultar más útil y acertada. En el pro-

grama 2.6 se muestra una función que calcula la valoración media de todas las películas ponderada por la afinidad de los usuarios con el usuario actual. $vp_j(x)$ es la valoración ponderada de una película x para el usuario j , y viene dada por la expresión:

$$vp_j(x) = \begin{cases} \frac{\sum_{i \in V(x)} (s(i,j)v_i(x))}{\sum_{i \in V(x)} s(i,j)} & \text{si } |V(x)| > 0 \\ 0 & \text{si } |V(x)| = 0 \end{cases} \quad (4)$$

donde $v_i(x)$ es la valoración de la película x por el usuario i y $s(i,j)$ es la similitud entre los usuarios i y j . Por último, $V(x)$ es el conjunto de usuarios que han valorado la película x .

Código 2.6: Función de recomendación media ponderada de películas

```

1 # Genera una lista ordenada de valoraciones ponderadas a partir de
2 # un diccionario de valoraciones de usuarios y un número de usuario.
3 # Se puede elegir la función de similitud entre usuarios.
4 def valoracionPonderada(diccio, usuario, similitud = coefPearson):
5     # En primer lugar se genera un diccionario con las similitudes
6     # de nuestro usuario con todos los demás.
7     # Este diccionario podría almacenarse para evitar recalcularlo.
8     simils = {x: similitud(diccio[usuario], diccio[x])
9               for x in diccio if x != usuario}
10
11     # Diccionario auxiliar {idPelicula: [valoracion*similitud
12     # usuarios]} y {idPelicula: [similitud usuarios]} (numerador y
13     # denominador de la valoración ponderada)
14     numerador = {}
15     denominador = {}
16
17     # Se recorre el diccionario de valoraciones y se rellenan los
18     # diccionarios auxiliares con los valores encontrados
19     for idUsuario in simils:
20         for idPeli in diccio[idUsuario]:
21             if not numerador.has_key(idPeli):
22                 numerador[idPeli] = []
23                 denominador[idPeli] = []
24             s = simils[idUsuario]
25             numerador[idPeli].append(diccio[idUsuario][idPeli]*s)
26             denominador[idPeli].append(s)
27
28     # Se calculan y ordenan las valoraciones ponderadas
29     result = []
30     for idPeli in numerador:
31         s1 = sum(numerador[idPeli])
32         s2 = sum(denominador[idPeli])
33         if s2 == 0:
34             media = 0.0
35         else:
36             media = s1/s2
37         result.append((idPeli, media))
38
39     result.sort(key = lambda x: x[1], reverse=True)
40     return result

```

2.2.4. Conclusiones

Los recomendadores vistos en este subapartado permiten mejorar de una forma muy sencilla sitios web y otras aplicaciones similares para sintonizar mejor

con los usuarios y sugerirles productos que les puedan interesar, mejorando así tanto su satisfacción con la aplicación como los éxitos potenciales de venta o acceso.

Las limitaciones de los métodos vistos hasta ahora son que, como se ha explicado, son métodos basados en memoria: requieren almacenar y procesar todos los datos cada vez que se realiza una consulta. De este modo, una operación sencilla no resulta excesivamente costosa, pero como se tiene que repetir completamente en cada consulta puede dar lugar a una carga computacional y de memoria desmesurada en aplicaciones medianas y grandes.

Otra limitación fundamental es que los métodos de recomendación que se acaban de estudiar no abstraen los datos de ninguna forma, es decir, no proporcionan información global acerca de los datos de los que se dispone, con lo que su uso está limitado a producir recomendaciones de productos, pero no pueden utilizarse para abordar estudios más complejos sobre los tipos de usuarios o productos de que se dispone, estudios que son esenciales para analizar el funcionamiento de la aplicación y diseñar su futuro. A continuación se estudiarán métodos que sí abstraen y producen información de alto nivel a partir de los datos disponibles.

2.3. Algoritmos de agrupamiento (*clustering*)

2.3.1. Ejemplo de aplicación

Una empresa de juegos en línea está analizando el comportamiento de sus clientes con el fin de ofrecerles los productos y características más adecuados a sus intereses. En este estudio se miden dos características: número de horas diarias jugando y número de horas diarias charlando (chateando) con otros jugadores, ambas en valor promedio para cada jugador. Los objetivos son dos: el primero, determinar qué tipos o clases de jugadores existen, según su actividad; el segundo, clasificar a los nuevos jugadores en alguna de esas clases para poder ofrecerles las condiciones y productos más adecuados. El segundo objetivo se trata extensamente en el apartado 3 dedicado a la clasificación, mientras que en este subapartado nos centraremos en cómo organizar un conjunto de datos extenso en unas cuantas clases o grupos, desconocidos *a priori*.

En la figura 2 se muestran los datos recogidos de 2.084 jugadores. Se trata de datos sintéticos (generados artificialmente) con el propósito de que ilustren más claramente los métodos que se expondrán a continuación. Concretamente, se pueden observar cinco clases o grupos claramente diferenciados, cada uno marcado con un símbolo diferente. En los datos reales no suele observarse una separación tan clara, y por consiguiente la ejecución de los métodos de agrupamiento no produce resultados tan definidos.

Figura 2. Actividad de los jugadores en línea

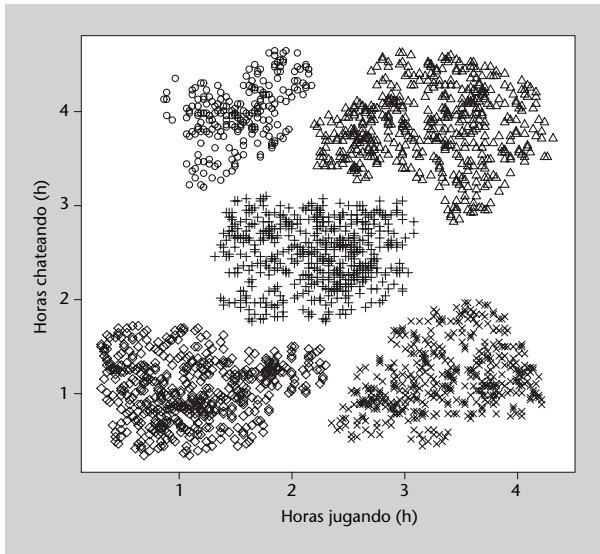


Figura 2

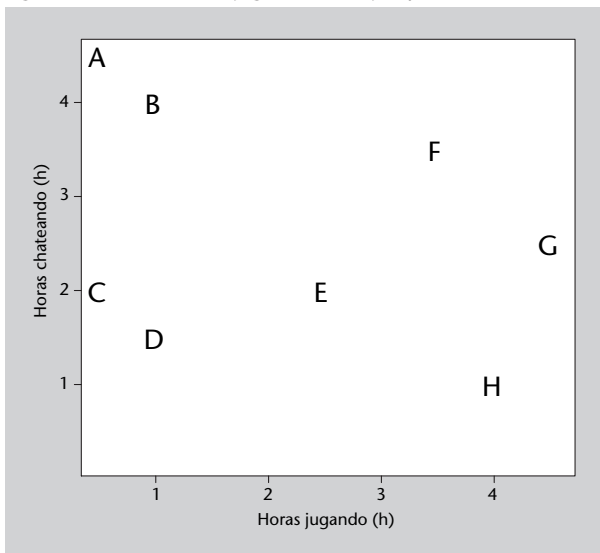
Las dos variables utilizan las mismas unidades (horas) y magnitudes similares y por tanto pueden utilizarse directamente para agrupar; en un caso más general, en el que las variables tengan magnitudes diferentes, suele ser necesario normalizarlas para que su magnitud se equipare y no tengan más influencia unas que otras (salvo que se pretenda justamente eso por la naturaleza del problema).

De hecho, para mostrar la ejecución paso a paso de algunos algoritmos de agrupamiento es necesario utilizar muchos menos datos, que en este caso son los que se muestran en la tabla 6 y en la figura 3.

Tabla 6. Jugadores de ejemplo

Jugador	Horas juego	Horas chat
A	0,5	4,5
B	1	4
C	0,5	2
D	1	1,5
E	2,5	2
F	3,5	3,5
G	4,5	2,5
H	4	1

Figura 3. Actividad de 8 jugadores de ejemplo



2.3.2. Conceptos generales

La tarea de agrupamiento de datos es una tarea **no supervisada**, ya que los datos que se le proporcionan al sistema no llevan asociada ninguna etiqueta o información añadida por un revisor humano; por el contrario, es el propio método de agrupamiento el que debe descubrir las nuevas clases o grupos a partir de los datos recibidos.

Si bien los algoritmos de agrupamiento se han introducido en el subapartado anterior como herramienta para crear recomendadores, sus aplicaciones van mucho más allá y se emplean en numerosos ámbitos, como por ejemplo:

- Procesamiento de imagen, para separar unas zonas de otras (típicamente con imágenes de satélite).
- Agrupamiento de genes relacionados con una determinada característica o patología.
- Agrupamiento automático de textos por temas.
- Definir tipos de clientes y orientar las estrategias comerciales en función de esos grupos.
- En general, definir grupos en conjuntos de datos de los que no se conoce una subdivisión previa: astronomía, física, química, biología, medicina, farmacología, economía, sociología, psicología, etc.

A menudo el agrupamiento de los datos precede a la clasificación de nuevos datos en alguno de los grupos obtenidos en el agrupamiento; por esa razón, el agrupamiento y la clasificación de datos están estrechamente relacionados, como se podrá observar en este subapartado, que comparte algunas técnicas con el subapartado 4.3.

Los algoritmos de agrupamiento se pueden clasificar en dos tipos: los **jerárquicos** con progresivos, es decir que van formando grupos progresivamente, y se estudiarán en primer lugar; los **particionales** sólo calculan una partición de los datos: los restantes algoritmos estudiados en este subapartado pertenecen a esta categoría.

Utilizando algoritmos de agrupamiento es posible construir recomendadores **basados en modelos**, llamados así porque a diferencia de los recomendadores basados en datos no necesitan almacenar todos los datos de que se dispone, sino que producen una abstracción de los datos dividiéndolos en grupos; para producir una recomendación sólo es necesario asociar un usuario o producto a un grupo existente, por lo que la información a almacenar se reduce a la descripción de los grupos obtenidos previamente. De esta forma la generación

de una recomendación se simplifica notablemente, y por tanto los recursos requeridos para generarla.

Más allá de los recomendadores, una de las características más importantes de los algoritmos de agrupamiento es que permiten organizar datos que en principio no se sabe o puede clasificar, evitando criterios subjetivos de clasificación, lo que produce una información muy valiosa a partir de datos desorganizados.

Una característica común a casi todos los algoritmos de agrupamiento es que son capaces de determinar por sí mismos el número de grupos idóneo, sino que hay que fijarlo de antemano o bien utilizar algún criterio de cohesión para saber cuándo detenerse (en el caso de los jerárquicos). En general ello requiere probar con diferentes números de grupos hasta obtener unos resultados adecuados.

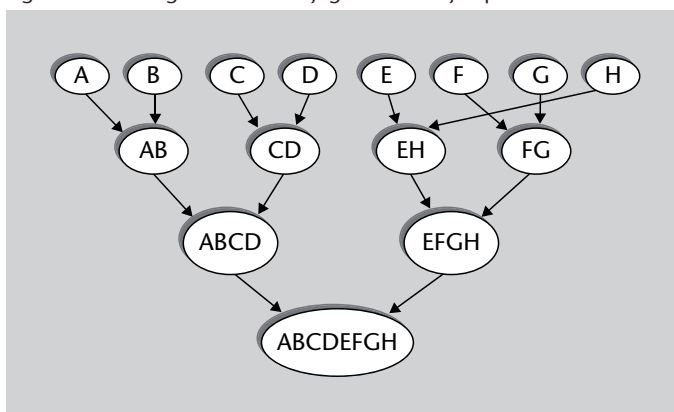
2.3.3. Agrupamiento jerárquico. Dendrogramas

Hay dos tipos de algoritmos de agrupamiento jerárquicos. El algoritmo **aglomerativo** parte de una fragmentación completa de los datos (cada dato tiene su propio grupo) y fusiona grupos progresivamente hasta alcanzar la situación contraria: todos los datos están reunidos en un único grupo. El algoritmo **divisivo**, por su parte, procede de la forma opuesta: parte de un único grupo que contiene todos los datos y lo va dividiendo progresivamente hasta tener un grupo para cada dato.

El **dendrograma*** es un diagrama que muestra las sucesivas agrupaciones que genera (o deshace) un algoritmo de agrupamiento jerárquico. En la figura 4 puede verse el dendrograma resultante de aplicar agrupamiento aglomerativo a los datos mostrados en la tabla 6.

*O diagrama de árbol, del griego *dendron*, árbol, y *gramma*, dibujo.

Figura 4. Dendrograma de los 8 jugadores de ejemplo de la tabla 6



En los algoritmos jerárquicos de agrupamiento, se parte del estado inicial y se aplica un criterio para decidir qué grupos unir o separar en cada paso, hasta que se alcance el estado final. Si bien conceptualmente ambos tipos de algoritmos jerárquicos (aglomerativo y divisivo) son equivalentes, en la práctica el algoritmo aglomerativo es más sencillo de diseñar por la razón de que sólo hay una forma de unir dos conjuntos, pero hay muchas formas de dividir un conjunto de más de dos elementos.

Criterios de enlace

En cada paso de un algoritmo de agrupamiento jerárquico hay que decidir qué grupos unir (o dividir). Para ello hay que determinar qué grupos están más cercanos, o bien qué grupo está menos cohesionado. Supongamos que estamos programando un algoritmo aglomerativo; la distancia entre dos grupos puede determinarse según diferentes expresiones, que reciben el nombre de **criterios de enlace**. Algunos de los criterios más utilizados para medir la distancia entre dos grupos A y B son los siguientes:

- Distancia máxima entre elementos de los grupos (enlace completo):

$$\max\{d(x,y) : x \in A, y \in B\}$$

- Distancia mínima entre elementos de los grupos (enlace simple):

$$\min\{d(x,y) : x \in A, y \in B\}$$

- Distancia media entre elementos de los grupos (enlace medio):

$$\frac{1}{|A||B|} \sum_{x \in A} \sum_{y \in B} d(x,y)$$

Una desventaja del criterio de enlace simple es que puede provocar que un único elemento fuerce la unión de todo su conjunto con otro conjunto que, por lo demás, no sea especialmente cercano. Por esa razón en general se preferirán otros criterios (completo, medio u otros).

Código del algoritmo aglomerativo

En el siguiente programa se definen las funciones necesarias para calcular el agrupamiento aglomerativo sobre conjunto de datos utilizando distancia euclídea (podría utilizarse otra distancia) y criterios de enlace completo o mí-

nimo. La función *fusionaGrupos* toma un agrupamiento, selecciona los dos grupos más cercanos según el criterio indicado y los fusiona; la función *agrupamientoAglomerativo* inicialmente genera un grupo para cada dato y los va fusionando hasta que sólo queda un grupo; en cada fusión muestra el agrupamiento resultante. Finalmente se incluye una función *leeJugadores* que lee un fichero con los datos de los jugadores en formato “horasJuego horasChat clase” (se incluye la clase para evaluar el rendimiento del programa).

Código 2.7: Agrupamiento aglomerativo

```

1 from math import sqrt
2
3 # Calcula la distancia euclídea entre dos vectores/tuplas
4 def distEuclidea(v1, v2):
5     return sqrt(sum(pow(x-y,2) for x,y in zip(v1,v2)))
6
7
8 # Calcula el enlace completo (máximo) entre dos grupos
9 # (distancia máxima entre dos puntos de cada grupo)
10 def enlaceCompleto(puntos, g1, g2, dist = distEuclidea):
11     # Busca el máximo en las combinaciones de puntos
12     maxima = 0.0
13     for p1 in g1:
14         for p2 in g2:
15             d = dist(puntos[p1], puntos[p2])
16             if d > maxima:
17                 maxima = d
18     return maxima
19
20
21 # Calcula el enlace simple (mínimo) entre dos grupos
22 # (distancia mínima entre dos puntos de cada grupo)
23 def enlaceSimple(puntos, g1, g2, dist = distEuclidea):
24     # Busca el mínimo en las combinaciones de puntos
25     minima = float("inf")
26     for p1 in g1:
27         for p2 in g2:
28             d = dist(puntos[p1], puntos[p2])
29             if d < minima:
30                 minima = d
31     return minima
32
33 # Dado un conjunto de puntos y un agrupamiento, fusiona
34 # los dos grupos más cercanos con el criterio indicado.
35 # "grupos" debe contener al menos dos grupos, y vuelve
36 # modificado, con los grupos elegidos fusionados.
37 def fusionaGrupos(puntos, grupos, criterio=enlaceCompleto,
38                  dist=distEuclidea):
39     if len(grupos) < 1: return
40
41     # Busca el par de grupos más adecuados (valor mínimo
42     # del criterio utilizado).
43     minimo = float("inf")
44     nombres = grupos.keys()
45     for i in range(len(nombres)-1):
46         for j in range(i+1, len(nombres)):
47             d = criterio(puntos, grupos[nombres[i]],
48                         grupos[nombres[j]], dist)
49             if d < minimo:
50                 minimo = d
51                 candidato = (nombres[i], nombres[j])
52
53     # El nombre del nuevo grupo será el más bajo de los dos
54     nombreGrupo = min(candidato)
55     grupoBorrar = max(candidato)
56
57     # Fusiona los dos grupos: añade los elementos a uno de
58     # ellos y elimina el otro del diccionario "grupos".
59     grupos[nombreGrupo].extend(grupos[grupoBorrar])

```

```

60     del(grupos[grupoBorrar])
61
62
63 # Agrupamiento jerárquico aglomerativo: fusiona grupos
64 # hasta obtener un único grupo
65 def agrupamientoAglomerativo(puntos, criterio=enlaceCompleto,
66                               dist=distEuclidea):
67     # Generación del agrupamiento inicial (cada punto un grupo)
68     grupos = {x:[x] for x in puntos}
69     print(grupos)
70
71     # Fusión de grupos hasta conseguir un único grupo
72     while len(grupos) > 1:
73         fusionaGrupos(puntos, grupos, criterio, dist)
74     print(grupos)
75
76 # Lee un fichero con los datos de los jugadores, formato
77 # "horasJuego horasChat clase" (la clase se ignora)
78 def leeJugadores(nomFich="Jugadores.txt"):
79     lineas = [(l.strip()).split("\t")]
80     for l in (open(nomFich).readlines())
81     # Se asigna un id=0,1,2,... a cada jugador
82     diccio = {}
83     for i in range(len(lineas)):
84         diccio[i] = (float(lineas[i][0]), float(lineas[i][1]))
85     return diccio
86
87
88 # Diccionario de puntos de ejemplo con sus coordenadas
89 puntos = {'A':(0.5,4.5), 'B':(1,4), 'C':(0.5,2), 'D':(1,1.5),
90           'E':(2.5,2), 'F':(3.5,3.5), 'G':(4.5,2.5), 'H':(4,1)}

```

Una posible optimización del programa consistiría en almacenar las distancias entre grupos para no tener que recalcularlas todas en cada fusión. En el programa de ejemplo no se ha hecho así para no complicarlo por cuestiones de simple eficiencia.

Conclusiones

El principal interés de los algoritmos jerárquicos radica en que generan diferentes grados de agrupamiento, y su evolución, lo que puede ser tan útil como la obtención de un agrupamiento definitivo. Además son sencillos conceptualmente y desde el punto de vista de la programación. Por otra parte, los algoritmos jerárquicos de agrupamiento actúan de forma **voraz** (*greedy*), ya que en cada paso toman la mejor decisión en ese momento, sin tener en cuenta la futura conveniencia de tal decisión. Por ese motivo, en general dan peor resultado que otros algoritmos de agrupamiento.

2.3.4. k-medios (*k-means*)

El algoritmo de agrupamiento k-medios busca una partición de los datos tal que cada punto esté asignado al grupo cuyo centro (llamado *centroide*, pues no tiene por qué ser un punto de los datos) sea más cercano. Se le debe indicar el número *k* de clusters deseado, pues por sí mismo no es capaz de determinarlo.

A grandes rasgos el algoritmo es el siguiente:

- 1) Elegir k puntos al azar como centroides iniciales. No tienen por qué pertenecer al conjunto de datos, aunque sus coordenadas deben estar en el mismo intervalo.
- 2) Asignar cada punto del conjunto de datos al centroide más cercano, formándose así k grupos.
- 3) Recalcular los nuevos centroides de los k grupos, que estarán en el centro geométrico del conjunto de puntos del grupo.
- 4) Volver al paso 2 hasta que las asignaciones a grupos no varíen o se hayan superado las iteraciones previstas.

El programa de k -medios está escrito en el código 4.4. Si bien se trata de un algoritmo sencillo y rápido, al tener en cuenta sólo la distancia a los centroides puede fallar en algunos casos (nubes de puntos de diferentes formas o densidades), ya que por lo general tiende a crear esferas de tamaño similar que particionen el espacio. Así, para los datos mostrados en la figura 2, la partición en grupos aproximada es la mostrada en la figura 5, en la que se ve que, por ejemplo, algunos puntos del grupo superior derecho (triángulos) se han asignado al superior izquierdo (círculos) por estar más cercanos al centroide, sin tener en cuenta el espacio que los separa. El mismo problema se da en otros puntos. Además, su resultado puede variar de una ejecución a otra, pues depende de los centroides generados aleatoriamente en el primer paso del algoritmo.

Figura 5. K-medios ejecutado sobre los datos de la figura 2

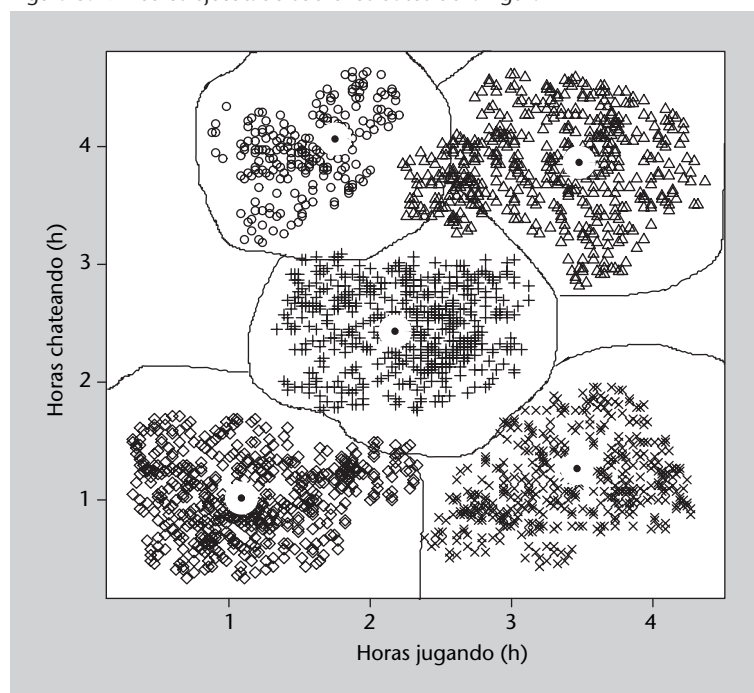


Figura 5

Los puntos negros rodeados de una zona blanca representan los centroides de los grupos obtenidos.

2.3.5. c-medios difuso (*Fuzzy c-means*)

Resulta lógico pensar que, en k-medios, un punto que esté junto al centroide estará más fuertemente asociado a su grupo que un punto que esté en el límite con el grupo vecino. El agrupamiento difuso (*fuzzy clustering*) representa ese diferente grado de vinculación haciendo que cada dato tenga un grado de pertenencia a cada grupo, de manera que un punto junto al centroide puede tener 0,99 de pertenencia a su grupo y 0,01 al vecino, mientras que un punto junto al límite puede tener 0,55 de pertenencia a su grupo y 0,45 al vecino. Esto es extensible a más de dos grupos, entre los que se repartirá la pertenencia de los datos. Se puede decir que el agrupamiento discreto (no difuso) es un caso particular del difuso en el que los grados de pertenencia son 1 para el grupo principal y 0 para los restantes.

El algoritmo de c-medios difuso es prácticamente idéntico al algoritmo k-medios visto anteriormente, las principales diferencias son:

- Cada dato x tiene asociado un vector de k valores reales que indican el grado de pertenencia $m_x(k)$ de ese dato a cada uno de los k grupos. El grado de pertenencia de un punto a un grupo depende de su distancia al centroide correspondiente. Habitualmente la suma de los grados de pertenencia de un dato es igual a 1.
- En lugar de crear k centroides aleatoriamente, se asigna el grado de pertenencia de cada punto a cada grupo aleatoriamente y después se calculan los centroides a partir de esa información.
- El cálculo de los centroides está ponderado por el grado de pertenencia de cada punto al grupo correspondiente $m_x(k)$.

A grandes rasgos las ventajas e inconvenientes de c-medios difuso son las mismas que las de k-medios: simplicidad, rapidez, pero no determinismo y excesiva dependencia de la distancia de los puntos a los centroides, sin tener en cuenta la densidad de puntos de cada zona (por ejemplo, espacios vacíos). Se trata de un algoritmo especialmente utilizado en procesamiento de imágenes.

2.3.6. Agrupamiento espectral (*spectral clustering*)

El **espectro** de una matriz es el conjunto de sus **valores propios**. La técnica de análisis de componentes principales descrita en el subapartado 3.1.2 está estrechamente relacionada con el agrupamiento espectral, pues determina los ejes en los que los datos ofrecen mayor variabilidad tomando los vectores asociados a valores propios mayores; en la técnica que aquí nos ocupa, por el contrario, se buscan los valores propios menores, pues indican escasa variabilidad y por tanto pertenencia a un mismo grupo.

Ved también

Los valores y vectores propios de una matriz y su obtención con Python están explicados en el subapartado 3.1.1.

Si bien el fundamento matemático detallado queda fuera del ámbito de estos materiales, el algoritmo de agrupamiento espectral consta de los siguientes pasos:

- 1) Calcular la matriz de distancias W de los datos que se desea agrupar.
- 2) Calcular la **laplaciana** de la matriz de distancias mediante la fórmula $L_{rw} = D - W$, donde I es la matriz identidad y D es una matriz en la que los elementos de la diagonal son la suma de cada fila de W .
- 3) Calcular los valores y vectores propios de L_{rw} .
- 4) Ordenar los valores propios de menor a mayor; reordenar las columnas de la matriz de vectores propios según ese mismo orden.
- 5) El número de grupos k sugerido por el algoritmo es el número de valores propios muy pequeños (menores que 10^{-10} si los grupos están realmente separados).
- 6) Aplicar un algoritmo de *clustering* convencional (k-medios por ejemplo) a las k primeras columnas de la matriz de vectores propios, indicando que se desea obtener k grupos.

El código 2.8 muestra el programa correspondiente a la descripción anterior, en la que faltaría la llamada a uno de los métodos de agrupamiento vistos anteriormente, como k-medios, para completar el proceso.

Código 2.8: Agrupamiento espectral

```

1 from numpy import *
2 from numpy.linalg import eig
3
4
5 # Calcula la distancia euclídea entre dos vectores/tuplas
6 def distEuclidea(v1, v2):
7     return sqrt(sum(pow(x-y,2) for x,y in zip(v1,v2)))
8
9
10 # Cálculo de la matriz de distancias de una lista de puntos
11 def matrizDist(puntos):
12     # Crea una matriz de p x p, siendo p el número de puntos
13     dist = zeros((len(puntos), len(puntos)))
14
15     # Calcula las distancias entre puntos
16     for i in range(len(puntos)):
17         for j in range(i):
18             d = distEuclidea(puntos[i], puntos[j])
19             dist[i, j] = d
20             dist[j, i] = d
21     return dist
22
23
24 # Cálculo de la laplaciana sin normalizar
25 def laplaciana(W):
26     I = identity(len(W), float)
27     D = diag([sum(Wi) for Wi in W])
28     L = D - W
29     return L
30
31
32 # Generación del modelo espectral

```

Lectura complementaria

U. von Luxburg (2007). *A Tutorial on Spectral Clustering*. *Statistics and Computing* (vol. 4, núm. 17).

Laplaciana no normalizada

La laplaciana calculada en el algoritmo de agrupamiento espectral se trata de la laplaciana no normalizada. Existen otras dos laplacianas aplicables en agrupamiento espectral, la simétrica y la de recorrido aleatorio; en muchos casos su comportamiento es equivalente.

```

33 def modeloEspectral(puntos):
34     # Cálculo de los valores y vectores propios
35     W = matrizDist(puntos)
36     L = laplaciana(W)
37     valp, vecp = eig(L)
38
39     # Ordenación por valor propio (menor a mayor) de los valores
40     # propios y de los vectores propios (por columnas)
41     orden = sorted(range(len(valp)), key = valp.__getitem__)
42     valp.sort()
43     vecp2 = zeros((len(vecp), len(vecp)))
44     for col in range(len(orden)):
45         vecp2[:, col] = vecp[:, orden[col]]
46
47     return valp, vecp2
48
49 # Lee un fichero con los datos de los jugadores, formato
50 # "horasJuego horasChat clase" (la clase se ignora), y
51 # devuelve una lista
52 def leeJugadores(nomFich="Jugadores.txt"):
53     lineas = [(l.strip()).split("\t")]
54     for l in (open(nomFich).readlines())
55
56     lista = []
57     for i in range(len(lineas)):
58         lista.append((float(lineas[i][0]), float(lineas[i][1])))
59     return lista
60
61
62 # Conjunto de puntos de ejemplo en forma de lista
63 puntos = [(0.5,4.5), (1,4), (0.5,2), (1,1.5),
64           (2.5,2), (3.5,3.5), (4.5,2.5), (4,1)]

```

El agrupamiento espectral es un método de agrupamiento muy potente, pues es capaz de agrupar datos teniendo en cuenta las diferencias de densidad, datos convexos (por ejemplo un círculo rodeando a otro), etc. Además indica el número de grupos que deben formarse.

Habitualmente se aplica un núcleo gaussiano* al calcular la matriz de distancias para mejorar su rendimiento.

*Véase el subapartado 4.5.2 en adelante.

2.3.7. Recomendadores basados en modelos

La aplicación de algoritmos de agrupamiento para la construcción de recomendadores se basa, por lo general, en utilizar los grupos obtenidos para decidir qué recomendar a un usuario, en lugar de utilizar todos los datos disponibles. Esto agiliza enormemente el proceso de decisión y permite disponer de varias “vistas” de los datos: grupos de usuarios, de productos, etc. De esta manera, en cada circunstancia se puede encontrar una recomendación adecuada de forma rápida.

3. Extracción y selección de atributos

Supongamos que deseamos analizar un conjunto de datos para resolver un problema determinado. Imaginemos por ejemplo una estación meteorológica que cada minuto realice una medida de la temperatura y de la humedad relativa. Al cabo de una hora habremos obtenido un conjunto de datos formado por $m = 60$ observaciones de $n = 2$ variables, que representaremos con la matriz de tamaño 60×2 . Si representamos las medida i -ésima de la temperatura con la variable $a_{i,1}$ y la de humedad relativa con $a_{i,2}$, la matriz de datos vendrá dada por

$$\begin{pmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \\ \vdots & \vdots \\ a_{60,1} & a_{60,2} \end{pmatrix} \quad (5)$$

Antes de proceder al análisis de los datos, debemos encontrar una forma adecuada de representarlos. A lo largo del apartado, representaremos los conjuntos de datos mediante una matriz multidimensional A de tamaño $m \times n$, en la que las n columnas representan las variables observadas y las m filas corresponden a las observaciones realizadas:

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \dots & a_{m,n} \end{pmatrix} \quad (6)$$

Una buena representación de datos resulta esencial porque por lo general los datos son redundantes o contienen información espuria. Antes de proceder a su análisis, es recomendable obtener un conjunto reducido de variables que permita describir los datos de forma simplificada. En el ejemplo anterior, si la temperatura y la humedad mantienen una relación determinada, los datos podrían simplificarse mediante una única variable que sea función de la temperatura y la humedad.

La simplificación de datos también se conoce como *reducción de dimensionalidad*, y pretende eliminar aspectos supérfluos como correlaciones entre variables (redundancias) o fluctuaciones estocásticas (ruido). Estos aspectos son agrupados en lo que comúnmente llamamos *ruido*, es decir, todos aquellos factores que no representen información relevante del proceso que vayamos a estudiar.

En el caso anterior, el ruido provendrá de las fluctuaciones de temperatura y humedad debidas a los flujos micrometeorológicos, o del ruido electrónico introducido por los equipos electrónicos de medida.

Por contra, denominaremos *señal* a aquellos factores que contribuyen de forma relevante a los datos, de forma que en general se cumplirá la identidad siguiente:

$$\text{datos} = \text{señal} + \text{ruido} \quad (7)$$

Las fuentes más habituales de ruido se asocian a los procesos de obtención y transmisión de los datos. Cuando los datos son obtenidos por algún tipo de sistema instrumental, se introducirán ciertas fluctuaciones inherentes al sistema físico de medida o a las etapas de amplificación y acondicionamiento. Al ruido en los sistemas de medida debe añadirse el que se produce durante la transmisión de los datos a través de líneas de comunicación, típicamente debidos a interferencias electromagnéticas o a la pérdida o corrupción de datos en canales analógicos o digitales. Finalmente, también habrá que considerar cualquier distorsión de los datos debida a otros factores como redundancias, errores o a la incorporación de información que no resulte relevante. Como resulta evidente, uno de los problemas fundamentales del tratamiento de datos es separar señal y ruido. Pero ese no es el único aspecto que consideraremos en este subapartado. En ocasiones, el interés no será la eliminación del ruido sino la identificación de los diferentes aspectos que contribuyen a los datos. En estos casos será necesario agrupar algunas de las características de los datos para disponer de una representación de los mismos en función de k componentes elementales más simples que nos facilitan su interpretación

$$\text{datos} = \text{componente}_1 + \text{componente}_2 + \dots + \text{componente}_k. \quad (8)$$

Las componentes se suelen ordenar siguiendo algún criterio objetivo que determine su importancia, lo que nos permite una interpretación simplificada de los datos mediante reconstrucciones parciales obtenidas a partir de algunas de las componentes. En general, asumiremos que los datos se obtienen mediante

la realización de medidas experimentales de un sistema determinado, de forma que la estructura de los datos necesariamente refleja las características de dicho sistema. Por ejemplo, al analizar los datos de la estación meteorológica anterior, podremos separar los datos en dos factores, uno correspondiente a los efectos diurnos y otro a los nocturnos.

El objetivo entonces será separar los datos en componentes que den cuenta de alguna de las características principales del sistema subyacente. Consideremos, por ejemplo, un conjunto de datos que consista en una secuencia de imágenes por satélite de una determinada zona geográfica. Resulta lógico pensar que la presencia de diferentes características geográficas en la zona (regiones montañosas, campos de cultivo, ríos, etc.) se reflejarán de alguna forma en el conjunto de datos a analizar. Las técnicas que estudiaremos en este subapartado nos van a permitir identificar las diferentes subzonas geográficas a partir de un análisis de la estructura de los datos. El procedimiento consistirá en definir un subconjunto de características de los datos (por ejemplo, el tamaño en píxeles de zonas de color verde en las imágenes) para poder identificar las distintas subzonas que se reflejan en los datos, y por lo tanto para descomponer los mismos en componentes diferenciadas.

El nombre genérico de este conjunto de las técnicas descritas en el subapartado 3.1 es *métodos de factorización matricial* porque descomponen en factores una matriz de datos. Una de las formas más eficientes de separar los datos consiste en identificar las componentes que mejor describen su variabilidad. El objetivo será encontrar un sistema de variables que describan los ejes en los que los datos presentan una mayor variabilidad. Esta es la idea básica de la técnica de descomposición en componentes principales (PCA), que describiremos más adelante, en el subapartado 3.1.2. Otra de las formas consiste en descomponer los datos en sus fuentes independientes, es decir, en aquellos mecanismos de naturaleza independiente del sistema subyacente a los datos. Eso es lo que aprenderemos en el subapartado 3.1.3 cuando estudiemos la técnica de descomposición en componentes independientes (ICA). La tercera técnica que estudiaremos en el subapartado 3.1.4 es conocida como factorización de matrices no-negativas (NMF) y permite una descomposición de los datos de forma que cada componente contribuye de forma acumulativa a la matriz global.

En el subapartado 3.3.1 también aprenderemos cómo utilizar otras técnicas para la visualización de datos multidimensionales o para identificar qué variables permiten separar los datos en grupos diferentes en el subapartado 3.2.1.

3.1. Técnicas de factorización matricial

Las técnicas de extracción y selección de atributos nos permiten describir los datos en función de unas nuevas variables que pongan de manifiesto algu-

nos de los aspectos de los datos en los que estemos interesados, omitiendo o atenuando aquellos factores que no son relevantes.

Las técnicas que se describen en este apartado (SVD, PCA, ICA y NMF) son conocidas como *técnicas de factorización matricial*, puesto que todas ellas descomponen una matriz de datos como el producto de matrices más simples. En cualquier caso, la factorización de una matriz no es única, y cada técnica pone de manifiesto aspectos diferentes de la información contenida en los datos originales.

3.1.1. Descomposición en valores singulares (SVD)

La descomposición en valores singulares es una herramienta de álgebra lineal que tiene innumerables aplicaciones en campos como el cálculo numérico, el tratamiento de señales o la estadística. El motivo por el que resulta tan útil es que SVD permite descomponer una matriz en una forma en la que nos resultará muy sencillo calcular sus valores y vectores propios (diagonalización de matrices). En sistemas de inteligencia artificial, SVD permite descomponer una matriz de datos expresándola como la suma ponderada de sus vectores propios. Una de las formas más simples de simplificar un conjunto de datos es descomponerlos utilizando SVD y reconstruirlos a partir de los vectores propios con mayor valor propio. Los vectores propios con menor valor propio suelen corresponder a aspectos de detalle de los datos que pueden no ser relevantes en una primera exploración. Además, SVD también se utiliza en la descomposición en componentes principales que describiremos en el subapartado 3.1.2.

En el caso particular de una matriz cuadrada de tamaño $n \times n$, la *diagonalización* consiste en encontrar el conjunto de valores y vectores propios $\{\lambda_i, v_i\}, i = 1 \dots n$ que cumplan la identidad

$$A \cdot v_i = \lambda_i v_i, \quad (9)$$

para luego expresar la matriz A en la forma

$$A = V^{-1} \cdot \Lambda \cdot V, \quad (10)$$

donde Λ es una matriz diagonal $n \times n$ cuyos componentes son los valores propios λ_i y V es una matriz $n \times n$ cuyas n columnas son los vectores propios v_i (cada uno de tamaño $1 \times n$).

Diagonalizar una matriz

Diagonalizar una matriz consiste en obtener sus valores y vectores propios. Los vectores propios definen un nuevo sistema de coordenadas en el que la matriz de datos es diagonal.

El código 3.1 indica cómo obtener los valores y vectores propios de una matriz en Python. En la primera línea se importa la librería *numpy*, que incluye gran parte de las funciones que necesitaremos en este apartado. La línea 3 introduce una matriz A de tamaño 3×3 , y en la línea 10 se obtienen los valores y vectores propios de A utilizando la rutina *svd* del paquete de rutinas de álgebra lineal *linalg* incluido en la librería *numpy*. En las instrucciones de las líneas 20 y 23 se comprueba que la ecuación 9 se cumple para el primero de los valores propios v_1 (las líneas 20 y 23 corresponden respectivamente a los términos izquierdo y derecho de la ecuación 9).

Código 3.1: Ejemplo diagonalización en Python

```

1  >>> from numpy import *
2
3  >>> A = array([[1, 2, 3],[4, 5, 6],[7, 8, 9]])
4
5  >>> A
6  array([[1, 2, 3],
7         [4, 5, 6],
8         [7, 8, 9]])
9
10 >>> val, vec = linalg.eig(A)
11
12 >>> val
13 array([ 1.61168440e+01, -1.11684397e+00, -3.21775239e-16])
14
15 >>> vec
16 array([[ -0.23197069, -0.78583024,  0.40824829],
17         [-0.52532209, -0.08675134, -0.81649658],
18         [-0.8186735 ,  0.61232756,  0.40824829]])
19
20 >>> dot(A, vec[:,0])
21 array([ -3.73863537, -8.46653421, -13.19443305])
22
23 >>> val[0]*vec[:,0]
24 array([ -3.73863537, -8.46653421, -13.19443305])

```

Como las matrices de datos no suelen ser cuadradas, SVD generaliza la diagonalización en matrices con un tamaño arbitrario $m \times n$. El resultado consiste en una descomposición factorial de la matriz A de tamaño $m \times n$ como producto de tres matrices U , S y V

$$A = U \cdot S \cdot V^T, \quad (11)$$

donde S es una matriz diagonal de tamaño $m \times n$ con componentes no negativos y U, V son matrices unitarias de tamaño $m \times m$ y $n \times n$ respectivamente. A los componentes diagonales no nulos de la matriz S se les conoce como *valores singulares*, mientras que las m columnas de la matriz U y las n columnas de la matriz V se denominan *vectores singulares* por la izquierda y por la derecha, respectivamente. El número máximo de valores singulares diferentes de la matriz A está limitado por el rango máximo de la matriz A , $r = \min\{m, n\}$.

El código 3.2 describe cómo se realiza una descomposición SVD en Python utilizando las herramientas de la librería *numpy*. El código empieza definiendo

una matriz A de tamaño 4×2 , y en la línea 8 se obtiene su descomposición SVD. La última instrucción (línea 28) comprueba que los términos izquierdo y derecho de la ecuación 11 sean iguales utilizando la instrucción *allclose* de Python.

Código 3.2: Ejemplo SVD en Python

```

1 >>> from numpy import *
2 >>> A = array([[1, 3, 5, 7], [2, 4, 6, 8]]).T
3 >>> A
4 array([[1, 2],
5         [3, 4],
6         [5, 6],
7         [7, 8]])
8 >>> U, S, Vt = linalg.svd(A, full_matrices=True)
9 >>> U
10 array([[ -0.15248323, -0.82264747, -0.39450102, -0.37995913],
11         [-0.34991837, -0.42137529,  0.24279655,  0.80065588],
12         [-0.54735351, -0.0201031 ,  0.69790998, -0.46143436],
13         [-0.74478865,  0.38116908, -0.5462055 ,  0.04073761]])
14 >>> S
15 array([ 14.2690955 ,  0.62682823])
16 >>> Vt
17 array([[ -0.64142303, -0.7671874 ],
18         [ 0.7671874 , -0.64142303]])
19 >>> S1 = zeros((4, 2))
20 >>> S1[:2, :2] = diag(S)
21 >>> S1
22 array([[ 14.2690955 ,  0.          ],
23         [  0.          ,  0.62682823],
24         [  0.          ,  0.          ],
25         [  0.          ,  0.          ]])
26 >>> S1.shape
27 (4, 2)
28 >>> allclose(A, dot(U, dot(S1, Vh)))
29 True

```

SVD también puede interpretarse como una descomposición en la que la matriz A se expresa como la suma de $r = \min\{m, n\}$ matrices, conocidas como *componentes SVD*

$$A = U \cdot S \cdot V^T = S(1,1) u_1 \cdot v_1^T + S(2,2) u_2 \cdot v_2^T + \dots + S(r,r) u_r \cdot v_r^T, \quad (12)$$

donde u_1, u_2, \dots, u_m son los vectores columna de la matriz de vectores singulares por la izquierda $U = \{u_1, u_2, \dots, u_m\}$, y v_1, v_2, \dots, v_n son los vectores columna de la matriz de vectores singulares por la derecha $V = \{v_1, v_2, \dots, v_n\}$. En esta representación, cada una de las r matrices $u_j \cdot v_j^T$ tienen un peso que viene dado por el valor singular correspondiente $S(j,j)$. Así, los vectores singulares con mayor valor singular serán más representativos de la matriz A , mientras que los vectores singulares con un menor valor singular aportarán muy poco a la descomposición en componentes SVD de A . Este hecho es el que permite que SVD sea utilizada como técnica de representación de datos, pues resulta una herramienta útil para identificar qué vectores singulares son relevantes para describir la matriz A y cuáles pueden considerarse como residuales.

Aplicando este procedimiento a una matriz de datos obtendremos una representación de la información en función de unos pocos vectores singulares, y

por lo tanto dispondremos de una representación de los datos en un espacio de dimensionalidad reducida. Representar los datos de forma fiable en un espacio de dimensionalidad reducida tiene evidentes ventajas para realizar un análisis exploratorio de la información, y puede ser utilizado como técnica de reducción del ruido o de compresión de datos. El código 3.3 implementa un ejemplo de esta descomposición en componentes SVD, comprobando que la matriz A puede reconstruirse como la suma de sus componentes (la línea 7 corresponde a la parte derecha de la ecuación 12, el símbolo \backslash es simplemente un salto de línea).

Observación

La instrucción `z.reshape(-1,1)` convierte el vector z en un vector columna. Para convertir un vector z en un vector fila utilizaríamos la instrucción `z.reshape(1,-1)`.

Código 3.3: descomposición de una matriz en sus componentes SVD

```

1  #!/usr/bin/env python
2
3  >>> A = array([[1, 2],[3, 4]])
4
5  >>> U,S,Vh = linalg.svd(A)
6
7  >>> S[0]*U[0].reshape(-1,1)*Vh[0] + S[1]*U[1].reshape(-1,1)*Vh[1]
8  array([[ 1.,  2.],
9         [ 3.,  4.]])

```

Otra propiedad interesante de la descomposición SVD de la matriz A es que permite calcular directamente los vectores y valores propios de la matriz $A^T \cdot A$. En efecto, tenemos que

$$A = U \cdot S \cdot V^T, \quad (13)$$

$$A^T = V \cdot S \cdot U^T, \quad (14)$$

y aplicando la unicidad de U , $U^T \cdot U = 1$, tenemos que

$$A^T \cdot A = V \cdot S \cdot U^T \cdot U \cdot S \cdot V^T = V \cdot S^2 \cdot V^T, \quad (15)$$

lo que demuestra que los vectores propios de $A^T \cdot A$ vienen dados (con la posible excepción del signo) por el vector V^T obtenido en la descomposición SVD de A , y sus vectores propios por S^2 . Por lo tanto, la descomposición SVD de una matriz de datos A permite conocer los valores y vectores propios de la matriz de correlación $A^T \cdot A$. Esta propiedad se describe con un ejemplo en el código 3.4, en el que se comparan los resultados de aplicar SVD a una matriz A con los obtenidos de diagonalizar la matriz $A^T \cdot A$.

Código 3.4: SVD permite calcular los valores y vectores propios de la matriz $A^T \cdot A$

```

1  >>> from numpy import *
2  >>> A = array([[1, 2, 3],[4, 5, 6],[7, 8, 9]])
3
4  >>> cov_A = dot(A.T,A)
5  >>> cov_A
6  array([[ 66,  78,  90],

```

```
7      [ 78, 93, 108],
8      [ 90, 108, 126]])
9
10 >>> val, vec = linalg.eig(cov_A)
11
12 >>> U, S, Vt = linalg.svd(A)
13
14 >>> val
15 array([ 2.83858587e+02,  1.14141342e+00, -2.38229079e-15])
16
17 >>> S*S.T
18 array([ 2.83858587e+02,  1.14141342e+00,  1.18098937e-31])
19
20 >>> vec
21 array([[ -0.47967118, -0.77669099,  0.40824829],
22        [ -0.57236779, -0.07568647, -0.81649658],
23        [ -0.66506441,  0.62531805,  0.40824829]])
24
25 >>> Vt.T
26 array([[ -0.47967118, -0.77669099,  0.40824829],
27        [ -0.57236779, -0.07568647, -0.81649658],
28        [ -0.66506441,  0.62531805,  0.40824829]])
```

3.1.2. Análisis de componentes principales (PCA)

Ejemplo de aplicación

A modo de ejemplo introductorio, supongamos que disponemos de un conjunto de datos con información sobre los jugadores de fútbol de primera división. Teniendo en cuenta que la liga española dispone de 20 equipos y que cada uno consta de 25 jugadores con licencia federativa, dispondremos de un total de $m = 500$ observaciones. Para cada jugador disponemos de tres variables con la media de pases, goles y balones robados por partido respectivamente ($n = 3$ variables, la matriz de datos tendrá un tamaño $m = 500 \times 3$). Los datos, representados en el sistema de ejes tridimensional de las variables, adoptan un aspecto similar al de una nube de puntos en el espacio.

PCA es una técnica que permite determinar los ejes principales de la distribución de la nube de puntos. Encontrar los ejes principales de la nube de puntos equivale a encontrar un nuevo conjunto de variables en los que los datos presentan una dispersión máxima, es decir, aquellos ejes en los que la variabilidad de los datos es mayor. En el ejemplo de los datos de futbolistas, estos ejes corresponden a combinaciones de las tres variables en las que los futbolistas tienen una mayor variabilidad. Podría pasar, por ejemplo, que la mayoría de futbolistas realicen un número promedio de pases por partido similar, y que las discrepancias entre jugadores se deban a una combinación entre el número promedio de balones que roban y de goles que marcan. Esta combinación de variables describirá un nuevo eje de coordenadas (el primer componente principal), en el que se producirá una mayor dispersión de los datos en el espacio original. Una proyección de los datos en un subconjunto de los ejes principales permite una simplificación de los datos a partir de una representación en un espacio de dimensionalidad reducida. Más adelante veremos un ejemplo genérico en el que aplicaremos la técnica PCA a un

conjunto de datos tridimensional. La técnica PCA se basa en la diagonalización de la matriz de covarianza de los datos, y sus características principales se describen a continuación.

Descripción de la técnica

Si A representa una matriz de datos de tamaño $m \times n$ con m observaciones de n variables, la matriz $A^T \cdot A$ es conocida como *matriz de autocorrelación* y expresa las relaciones estadísticas existentes entre las n variables. Básicamente, la técnica PCA consiste en diagonalizar la matriz de autocorrelación $A^T \cdot A$ o la matriz de covarianza $(A - \bar{A})^T \cdot (A - \bar{A})$ de los datos A .

Como las matrices de autocorrelación o autocovarianza indican correlación estadística entre variables, sus vectores y valores propios nos permitirán obtener unas nuevas variables entre las que la correlación estadística sea mínima. Los vectores propios darán lugar a las *componentes principales*, definiendo unos nuevos ejes de coordenadas orientados en las direcciones en las que los datos presenten una mayor variabilidad. Los vectores propios con mayor valor propio corresponderán a las direcciones en las que los datos presentan una mayor dispersión, y aquellos vectores propios con un autovalor pequeño indicarán una dirección en la que los datos apenas varían su valor.

Como hemos visto en el subapartado 3.1.1, la descomposición SVD de una matriz A permite calcular los vectores y valores propios de la matriz $A^T \cdot A$, por lo que SVD nos será de gran utilidad para descomponer A en sus componentes principales.

La relación entre SVD y PCA se pone de manifiesto al escribir la descomposición SVD de la matriz A en la forma

$$A = R \cdot V^T \quad (16)$$

donde $R = U \cdot S$ es conocida como *matriz de resultados* (en inglés, *scores matrix*), y a la matriz de vectores singulares por la derecha V^T se le conoce como *matriz de cargas* (en inglés, *loadings matrix*).

La ecuación 16 expresa la descomposición PCA de la matriz A como una factorización en las matrices R y V^T .

Los pasos a seguir para realizar una descomposición PCA de una matriz de datos en Python se indican en el ejemplo descrito en el código 3.5.

Código 3.5: Ejemplo de PCA en Python

```

1 from numpy import *
2 import pylab
3 from mpl_toolkits.mplot3d import Axes3D
4 import matplotlib.pyplot as plot
5 set_printoptions(precision = 3)
6
7 # Datos: distribución normal multivariada en 3d
8 mean = [1,5,10]
9 cov = [[-1,1,2],[-2,3,1],[4,0,3]]
10 d = random.multivariate_normal(mean,cov,1000)
11
12 # representación gráfica de los datos:
13 fig1 = plot.figure()
14 sp = fig1.gca(projection = '3d')
15 sp.scatter(d[:,0],d[:,1],d[:,2])
16 plot.show()
17
18 # ANALISIS PCA:
19 # Paso 1: Calcular la matriz de covarianza de los datos (N x N):
20 d1 = d - d.mean(0)
21 matcov = dot(d1.transpose(),d1)
22
23 # Paso 2: Obtener los valores y vectores propios de la matrix de
24 # covarianza:
25 valp1,vecp1 = linalg.eig(matcov)
26
27 # Paso 3: Decidir qué vectores son los relevantes representando
28 # los valores propios en orden decreciente (scree plot):
29 ind_decre = argsort(valp1) # orden creciente
30 ind_decre = ind_decre[::-1] # orden decreciente
31 val_decre = valp1[ind_decre] # valores propios en orden decreciente
32 vec_decre = vecp1[:,ind_decre] # ordenar tambien vectores propios
33 pylab.plot(val_decre, 'o-')
34 pylab.show()
35
36 # Proyectar datos a la nueva base definida por los
37 # todos los vectores propios (espacio PCA 3D)
38 d_PCA = zeros((d.shape[0],d.shape[1]))
39 for i in range(d.shape[0]):
40     for j in range(d.shape[1]):
41         d_PCA[i,j] = dot(d[i,:],vecp1[:,j])
42
43 # recuperar datos originales invirtiendo la proyección (reconstrucción)
44 :
45 d_recon = zeros((d.shape[0],d.shape[1]))
46 for i in range(d.shape[0]):
47     for j in range(d.shape[1]):
48         d_recon[i] += d_PCA[i,j]*vecp1[:,j]
49
50 # comprobar que se recuperan los datos originales:
51 allclose(d,d_recon)
52
53 # Proyectar datos a la nueva base definida por los dos
54 # vectores propios con mayor valor propio (espacio PCA 2D)
55 d_PCA2 = zeros((d.shape[0],2))
56 for i in range(d.shape[0]):
57     for j in range(2):
58         d_PCA2[i,j] = dot(d[i,:],vec_decre[:,j])
59
60 # reconstruir datos invirtiendo la proyección PCA 2D:
61 d_recon2 = zeros((d.shape[0],d.shape[1]))
62 for i in range(d.shape[0]):
63     for j in range(2):
64         d_recon2[i] += d_PCA2[i,j]*vec_decre[:,j]
65
66 # representación gráfica de los datos:
67 fig2 = plot.figure()
68 sp2 = fig2.gca(projection = '3d')
69 sp2.scatter(d_recon2[:,0],d_recon2[:,1],d_recon2[:,2],c='r',marker='x')
70 plot.show()

```

Nuevo sistema de coordenadas

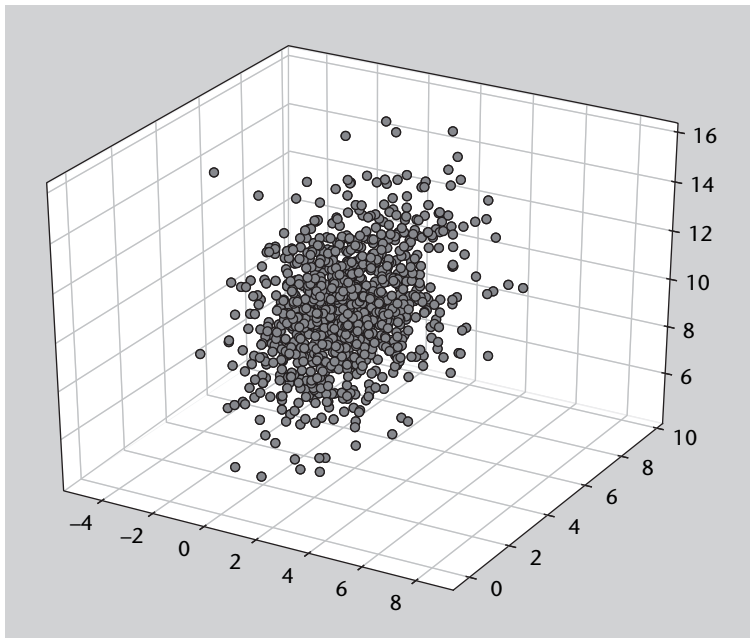
PCA permite caracterizar un conjunto de datos mediante un nuevo sistema de coordenadas en las direcciones en las que los datos presentan una mayor variabilidad.

En la línea 10 se genera una matriz 1000×3 de datos distribuidos aleatoriamente según una densidad de probabilidad normal multivariada con media $(1,5,10)$ y matriz de covarianza

$$\begin{pmatrix} -1 & 1 & 2 \\ -2 & 3 & 1 \\ 4 & 0 & 3 \end{pmatrix}. \quad (17)$$

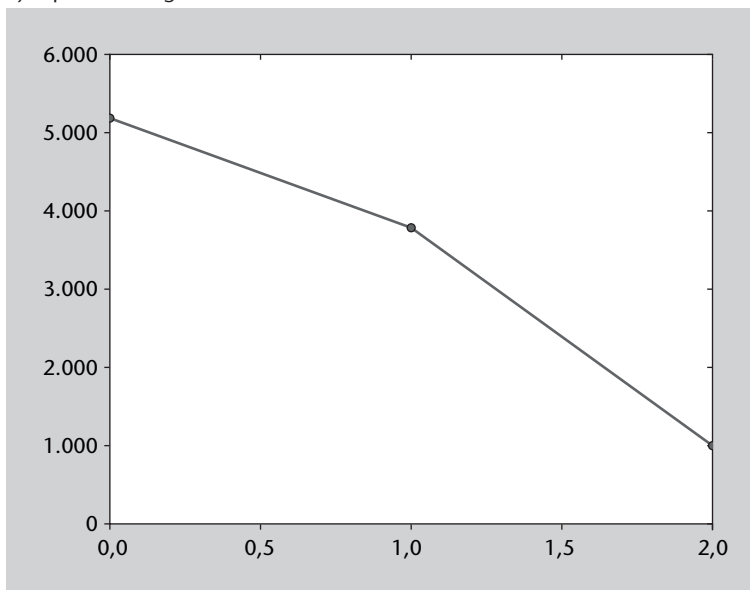
Esta matriz de datos es similar a la descrita en el ejemplo de los datos de futbolistas al que hacíamos referencia en la introducción de este subapartado. Los datos se representan en la figura 6 mediante una gráfica tridimensional, en la que se aprecia la distribución de datos en forma de nube de puntos.

Figura 6. Datos del ejemplo PCA descrito en el código 3.5



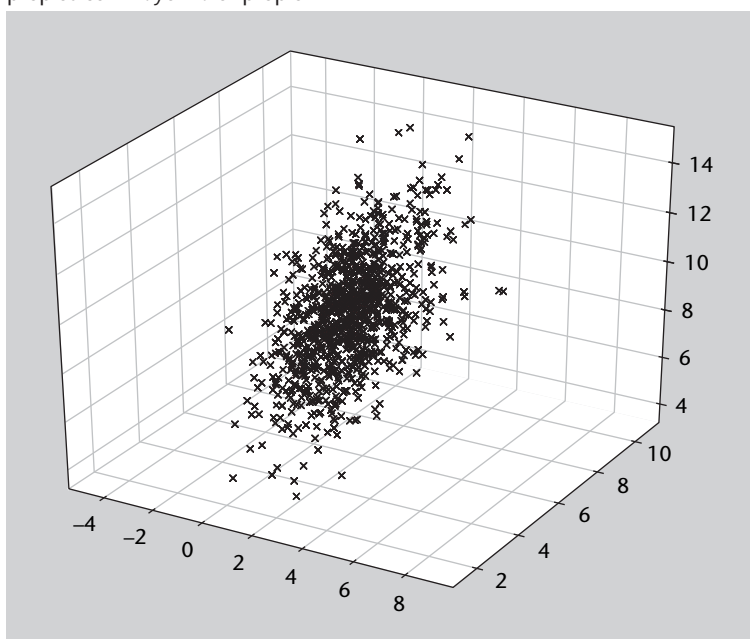
El primer paso para realizar PCA es obtener la matriz de autocovarianza de los datos (línea 21), que se obtiene tras sustraer la media a cada columna de la matriz d (línea 20). El segundo paso es obtener los valores y vectores propios de la matriz de autocovarianza (línea 24). Para decidir qué vectores propios son más relevantes, lo habitual es representar gráficamente los valores propios en orden decreciente en lo que se conoce como un *scree plot* (diagrama de pedregal, por la forma abrupta en la que decrecen los autovalores que recuerda a la ladera de una montaña). Los valores y vectores propios son ordenados siguiendo un orden decreciente del valor propio (líneas 28-31). Este diagrama aparece representado en la figura 7, en la que se observa que uno de los autovalores es significativamente menor que los otros dos. Cada uno de los vectores propios es un vector que apunta en tres direcciones ortogonales en las que los datos presentan una mayor variabilidad (espacio PCA).

Figura 7. Scree plot de los valores propios de la matriz de autocovarianza del ejemplo del código 3.5



En las líneas de la 37 a 40 se proyectan los datos originales al espacio PCA realizando el producto escalar entre los datos y los vectores propios (línea 40). Los datos originales se pueden recuperar invirtiendo la proyección (líneas 43), lo que se comprueba en la línea 49 del código. A continuación, se realiza una proyección de los datos a un espacio PCA que sólo tiene en cuenta los dos vectores propios con mayor valor propio asociado (líneas 53-56). Al considerar los dos componentes más relevantes que la reconstrucción de los datos a partir del espacio PCA de dimensionalidad reducida se asemeja a los datos originales (figura 8).

Figura 8. Reconstrucción de los datos del ejemplo 3.5 utilizando los dos vectores propios con mayor valor propio



Ejemplo: PCA para imágenes

Una de las aplicaciones más interesantes de PCA es el análisis de imágenes. En este caso, PCA se aplica a una secuencia de imágenes con el objetivo de identificar las características principales que aparecen en las mismas. Las aplicaciones son numerosas, e incluyen aspectos como el reconocimiento de caras, el análisis de imágenes médicas o el procesamiento de información geográfica obtenida mediante técnicas de teledetección por satélite.

Matemáticamente, una imagen no es más que una matriz de $n \times m$ valores enteros. Cada uno de los componentes de la matriz representa un píxel de la imagen, y adopta valores enteros entre 0 y $2^k - 1$, donde k es el número de bits utilizados para representar cada píxel. En caso de que se utilicen 8 bits por píxel ($k = 8$), por ejemplo, los componentes de la matriz tomarán valores enteros en el rango $[0 - 255]$. Por lo tanto, una secuencia de imágenes no es más que un conjunto de NIM matrices cada una de ellas de tamaño $n \times m$. Para poder trabajar con PCA, los datos se organizan de forma que cada imagen sea almacenada en un único vector fila que contenga sus $NPIX = n \cdot m$ píxeles. De esta manera, la secuencia de imágenes se describe matemáticamente como una matriz de tamaño $NIM \times NPIX$ que adopta la forma

$$\begin{pmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,NPIX} \\ a_{2,1} & a_{2,2} & \dots & a_{2,NPIX} \\ a_{3,1} & a_{3,2} & \dots & a_{3,NPIX} \\ \dots & \dots & \dots & \dots \\ a_{NIM,1} & a_{NIM,2} & \dots & a_{NIM,NPIX} \end{pmatrix}, \quad (18)$$

donde cada fila corresponde a una imagen de la secuencia y cada columna representa uno de los píxeles de la imagen.

El problema principal con el que nos encontramos al aplicar PCA a una secuencia de imágenes es que normalmente el número de imágenes NIM es considerablemente menor que el número de píxeles NPIX de cada imagen. El problema radica en que para calcular PCA habría que diagonalizar la matriz de covarianza de tamaño $NPIX \times NPIX$, lo que resulta tremendamente costoso incluso para imágenes de tamaño reducido (con imágenes de 128×128 píxeles tendríamos que diagonalizar una matriz de 16384×16384). Para solventar esta situación, se utiliza una interesante propiedad algebraica según la cual las matrices $A^T \cdot A$ (de tamaño $NPIX \times NPIX$) y $A \cdot A^T$ ($NIM \times NIM$) comparten los valores propios no nulos. En efecto, las condiciones de diagonalización de ambas matrices vienen dadas por

Imágenes NIM

Una secuencia de NIM imágenes de NPIX píxeles se puede representar como una matriz de datos con NPIX variables y NIM observaciones.

$$A^T \cdot A \cdot u_i = \lambda_i u_i \quad (19)$$

$$A \cdot A^T \cdot w_i = \lambda_i w_i. \quad (20)$$

Multiplicando la segunda ecuación por A^T por la izquierda tenemos

$$A^T \cdot A \cdot A^T \cdot w_i = A^T \lambda_i w_i, \quad (21)$$

e identificando 21 con la ecuación de 19 tenemos que los valores propios de $A^T \cdot A$ (u_i) y los de $A \cdot A^T$ (w_i) están relacionados de la forma

$$A^T \cdot w_i = u_i \quad (22)$$

por lo que podemos obtener los vectores propios de $A^T \cdot A$ diagonalizando la matriz $A \cdot A^T$ que, en el caso de imágenes, tiene un tamaño menor. Esta estrategia es la que se implementa en el código 3.6, en el que se aplica PCA a una secuencia de imágenes. El código supone que las imágenes tienen extensión *.jpg y que se encuentran en el mismo directorio en el que se ejecute el programa. Las líneas 13-23 construyen una lista Python con los nombres de los ficheros con extensión *.jpg encontrados en el directorio. Tras determinar el número de imágenes NIM y el tamaño de las mismas NPIX, se construye una matriz de datos de tamaño $NIM \times NPIX$ con las NIM imágenes desplegadas en forma de vectores fila de longitud NPIX (línea 33). A continuación se procede al centrado de los datos (líneas 41-43), se obtiene la matriz $M = A \cdot A^T$ (línea 45) y se calculan los vectores y valores propios de la matriz M (línea 46). Los vectores propios de $A^T \cdot A$ se obtienen utilizando la ecuación 22 (línea 47). Las líneas 49 ordenan los vectores propios en orden decreciente de sus valores propios, y la siguiente determina los valores propios de la matriz de datos A .

Código 3.6: Ejemplo de aplicación de PCA para el análisis de imágenes

```

1      import os
2  from PIL import Image
3  import numpy
4  import pylab
5  from pca_im import *
6  import matplotlib.pyplot as plt
7
8  #####
9  # CONSTRUIR MATRIZ DE DATOS
10 #####
11
12 # Obtener directorio actual en el que están las imágenes
13 path= os.getcwd()
14 dirList=os.listdir(path)
15
16 # construir una lista con los ficheros con extensión '.jpg'
17 imlist = []
18 i = 0
19 for fname in dirList:
20     filename, filext = os.path.splitext(fname)
21     if filext == '.jpg':

```

```

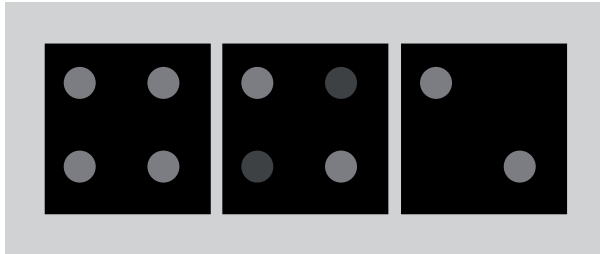
22     imlist.append(fname)
23     i = i +1
24
25 # Abrir la primera imagen y obtener su tamaño en píxeles
26 im = numpy.array(Image.open(imlist[0]))
27 m,n = im.shape[0:2]
28
29 NPIX = n*m    # Numero de píxeles de cada imagen:
30 NIM = len(imlist) # Numero de imágenes a analizar:
31
32 # Crear una matriz de tamaño NIM x NPIX:
33 A = numpy.array([numpy.array(Image.open(imlist[i])).flatten()
34                 for i in range(NIM)], 'f')
35
36 #####
37 # ANALISIS PCA
38 #####
39
40 # Centrado de los datos restando la media:
41 im_media = A.mean(axis=0)
42 for i in range(NIM):
43     A[i] -= im_media
44
45 M = dot(A,A.T) # matriz AA' (NIM x NIM)
46 lam,vec = linalg.eigh(M) # obtener los NIM vectores y valores propios
47     de M = AA'
48 aux = dot(A.T,vec).T # aplicar w = A'v para obtener los vectores
49     propios de A'A
50 V = aux[:,:-1] # ordenar vectores propios
51 S = sqrt(lam)[::-1] # ordenar valores propios de A
52
53 # Reconponer la imagen media:
54 im_media = im_media.reshape(m,n)
55
56 # Representar los autovalores PCA
57 pylab.plot(S[0:10], 'o-')
58
59 # Obtener los dos primeros modos PCA y representarlos:
60 modo1 = V[0].reshape(m,n)
61 modo2 = V[1].reshape(m,n)
62
63 # Representar graficamente:
64 fig1 = pylab.figure()
65 fig1.suptitle('Imagen_promedio')
66 pylab.gray()
67 pylab.imshow(im_media)
68 ##
69 fig2 = pylab.figure()
70 fig2.suptitle('Primer_modos_PCA')
71 pylab.gray()
72 pylab.imshow(modo1)
73 ##
74 fig3 = pylab.figure()
75 fig3.suptitle('Segundo_modos_PCA')
76 pylab.gray()
77 pylab.imshow(modo2)
78 pylab.show()

```

El análisis de imágenes con PCA trabaja con una matriz de datos $NIM \times NPIX$, y puede interpretarse como la descomposición en componentes principales de $NPIX$ variables de las que tenemos NIM observaciones. Los vectores propios indican las combinaciones lineales de píxeles que mayor variabilidad presentan en la secuencia. Así, un píxel que adopte un valor constante durante la secuencia de imágenes no será incluido en ninguno de los componentes con mayor valor propio. En cambio, aquellos píxeles o grupos de píxeles que presenten mayor variabilidad a lo largo de la secuencia darán lugar a los com-

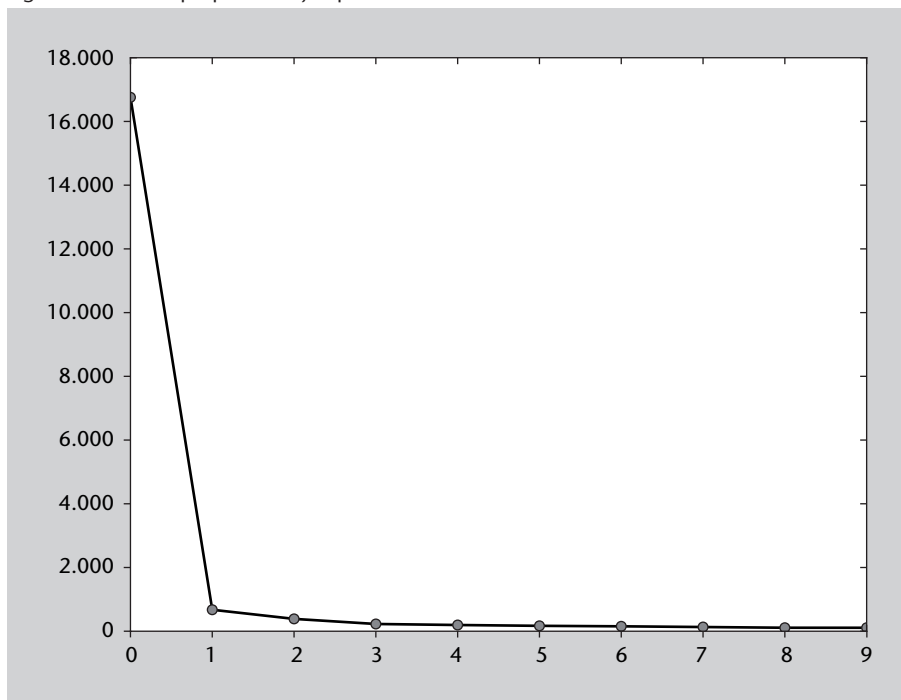
ponentes principales de la descomposición. Esto es lo que puede observarse cuando aplicamos el código 3.6 a una secuencia de 125 imágenes como las que se representan en la figura 9.

Figura 9. Imágenes 1,4 y 7 de la secuencia del ejemplo 3.6



En este ejemplo, los píxeles de los círculos superior izquierdo e inferior derecho adoptan un valor constante durante la secuencia, mientras que los círculos superior derecho e inferior izquierdo cambian su valor siguiendo una señal sinusoidal, completando cuatro períodos a lo largo de la secuencia. La figura 10 representa los diez primeros valores propios que se obtienen al aplicar el código PCA 3.6 a la secuencia de imágenes del ejemplo.

Figura 10. Valores propios del ejemplo 3.6



El primer valor propio es significativamente mayor que el resto, lo que nos indica que la mayor parte de la variabilidad de la secuencia puede resumirse en un único componente. La reconstrucción que se obtiene a partir de esa primera componente se representa en la figura 11, y contiene únicamente los círculos superior derecho e inferior izquierdo que son los que cambian a lo largo de la secuencia. El segundo modo PCA, con un valor propio mucho menor, se representa en la figura 12 y corresponde a valores residuales en el entorno de la primera componente PCA. Ni el fondo de la imagen ni los círculos supe-

rior izquierdo e inferior derecho aparecen representados en los cinco primeros modos del análisis PCA debido a que adoptan un valor constante durante la secuencia.

Figura 11. Primer modo PCA del ejemplo 3.6

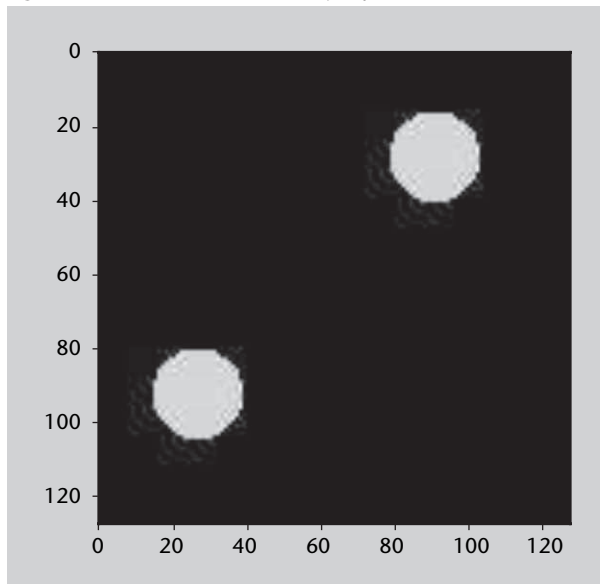
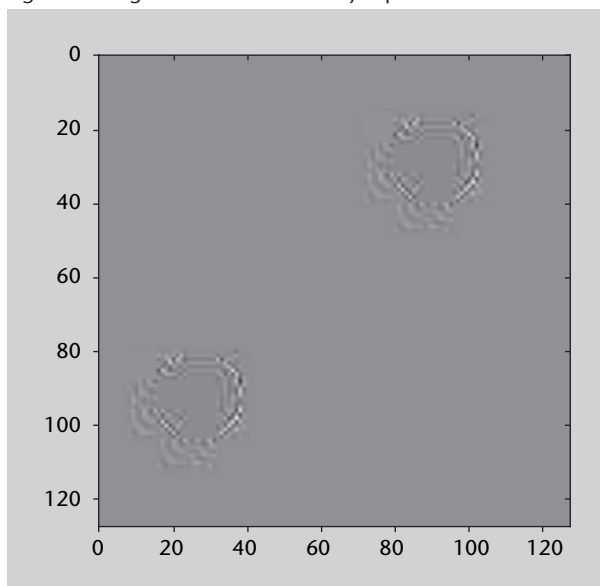


Figura 12. Segundo modo PCA del ejemplo 3.6



Análisis PCA de una secuencia de caras: Eigenfaces (autocarar)

Sin duda, el ejemplo paradigmático de PCA para imágenes es el análisis de una secuencia de caras. En este ejemplo partimos de una secuencia de 25 imágenes de caras obtenidas de la base de datos AR Face database de la Universidad de Ohio. Las 25 caras se representan en la figura 13 y corresponden a un conjunto de personas con diferentes expresiones faciales (sonriendo, serio, gritando, etc.).

Lectura recomendada

A. M. Martínez; R. Benavente (junio, 1998).
The AR Face Database. CVC Technical Report (núm. 24).

Figura 13. Secuencia de imágenes original (Ejemplo 3.7)



Igual que en el ejemplo anterior, el conjunto de datos de partida consta de NIM filas y NPIX columnas (matriz A, código 3.7). El código PCA es similar al del ejemplo anterior, pero en este caso la descomposición PCA se realiza mediante la descomposición SVD de la matriz de datos A. En el código 3.7 del ejemplo, las líneas 13-38 se dedican a abrir las imágenes contenidas en un directorio y a construir la matriz de datos A. El análisis PCA se realiza a partir de la línea 44, empezando por el centrado de datos y siguiendo con la descomposición SVD de la matriz A (línea 55). Tal y como se explicó en el subapartado 3.1.1 (ecuación 15 y el código 3.4), la descomposición SVD de A permite encontrar los valores y vectores propios de las matrices $A^T A$ y AA^T . Las líneas de la 58-68 simplemente comprueban que ambas matrices se recuperan a partir de los vectores singulares por la derecha (V) y por la izquierda (U), respectivamente. A partir de la línea 70 se procede a la reconstrucción PCA de la matriz de datos A. Cuando se utilizan todos los vectores propios (*eigenfaces*), la reconstrucción recupera la matriz de datos original A (línea 75).

Una reconstrucción parcial a partir de un conjunto reducido de *eigenfaces* permite realizar una descripción simplificada de la matriz de datos A y por tanto de la secuencia de imágenes original. En las líneas 78-81 se reconstruye la matriz de datos a partir de un número de componentes igual a 24, 10, 2 o 1 único modo PCA (1 única eigenface). Para observar cómo de fiables son cada una de las reconstrucciones, las figuras representan las imágenes reconstruidas. Es importante notar que como la descomposición SVD se ha aplicado a la matriz de datos centrada (habiendo extraído la media), a las reconstrucciones finales debe añadirse la imagen promedio para recuperar la imagen correcta (línea 90 y siguientes de la sección de figuras del código 3.7).

Código 3.7: Ejemplo *eigenfaces*: descomposición PCA de una secuencia de imágenes con caras

```

1 import os
2 from PIL import Image
3 from PIL import ImageOps
4 import numpy
5 import pylab
6 from pca_im import *
7 import matplotlib.pyplot as plt
8
9 #####
10 # COSNTRUIR MATRIZ DE DATOS
11 #####
12
13 # Obtener directorio actual en el que están las imágenes
14 path= os.getcwd()
15 dirList=os.listdir(path)
16
17 # construir una lista con los ficheros con extensión '.jpg'
18 imlist = []
19 i = 0
20 for fname in dirList:
21     filename, filext = os.path.splitext(fname)
22     if filext == '.jpg':
23         imlist.append(fname)
24         i = i +1
25
26 # Abrir la primera imagen y obtener su tamaño en píxeles
27 z = Image.open(imlist[0])
28 # Convertir a escala de grises (PIL, ImageOps)
29 ImageOps.grayscale(z)
30 im = numpy.array(z)
31 m,n = im.shape[0:2]
32
33 NPIX = n*m # Número de píxeles de cada imagen:
34 NIM = len(imlist) # Número de imágenes a analizar:
35
36 # Crear una matriz de tamaño NIM x NPIX:
37 A = numpy.array([numpy.array(ImageOps.grayscale(Image.open(imlist[i])))
38     .flatten()
39
40     for i in range(NIM)], 'f')
41 #####
42 # ANALISIS PCA
43 #####
44 # Centrado de los datos restando la media:
45 im_media = A.mean(axis=0)
46 for i in range(NIM):
47     A[i] -= im_media
48
49 im_media = im_media.reshape(m,n)
50
51 AAT = dot(A,A.T) # matriz AA' (NIM x NIM)
52 ATA = dot(A.T,A) # matriz A'A (NPIX x NPIX)
53
54 # Calcular PCA a partir de la descomposición SVD de A:
55 U,S,Vt = linalg.svd(A)
56 SS = S*S.T
57
58 # reconstrucción matriz ATA:
59 S2 = zeros((NPIX,NPIX))
60 for i in range(size(SS)):
61     S2[i,i] = SS[i]
62 resultATA = dot(Vt.T,dot(S2,Vt))
63
64 # reconstrucción matriz AAT:
65 S3 = zeros((NIM,NIM))
66 for i in range(size(SS)):
67     S3[i,i] = SS[i]
68 resultAAT = dot(U,dot(S3,U.T))

```

```
69
70 # reconstrucción matriz A:
71 S1 = zeros(A.shape)
72 for i in range(size(S)):
73     S1[i,i] = S[i]
74
75 resulta = dot(U,dot(S1,Vt)) # recuperar matriz original
76
77 # reconstrucciones parciales a partir de unos pocos modos PCA:
78 reconA24 = dot(U[:, :24], dot(S1[:24, :24], Vt[:24, :]))
79 reconA10 = dot(U[:, :10], dot(S1[:10, :10], Vt[:10, :]))
80 reconA2 = dot(U[:, :2], dot(S1[:2, :2], Vt[:2, :]))
81 reconA1 = dot(U[:, :1], dot(S1[:1, :1], Vt[:1, :]))
82
83 # Figuras
84
85 # reconstrucción de la primera imagen de la secuencia:
86 fig1 = pylab.figure()
87 pylab.gray()
88 pylab.axis('off')
89 fig1.suptitle('Imagen_n1_original')
90 pylab.imshow(A[0].reshape(m,n)+im_media)
91
92 fig2 = pylab.figure()
93 pylab.gray()
94 pylab.axis('off')
95 fig2.suptitle('Imagen_n1_reconstrucción_PCA_24_modos')
96 pylab.imshow(reconA24[0].reshape(m,n)+im_media)
97
98 fig3 = pylab.figure()
99 pylab.gray()
100 pylab.axis('off')
101 fig3.suptitle('Imagen_n1_reconstrucción_PCA_10_modos')
102 pylab.imshow(reconA10[0].reshape(m,n)+im_media)
103
104 fig4 = pylab.figure()
105 pylab.gray()
106 pylab.axis('off')
107 fig4.suptitle('Imagen_n1_reconstrucción_PCA_2_modos')
108 pylab.imshow(reconA2[0].reshape(m,n)+im_media)
109
110 fig5 = pylab.figure()
111 pylab.gray()
112 for i in range(NIM):
113     fig5.suptitle('Conjunto_de_imagenes_original')
114     pylab.subplot(5,5,i)
115     pylab.imshow(A[i].reshape(m,n)+im_media)
116     pylab.axis('off')
117
118 # reconstrucción de todas las imágenes de la secuencia
119
120 fig6 = pylab.figure()
121 pylab.gray()
122 fig6.suptitle('reconstrucción_PCA_10_modos')
123 for i in range(NIM):
124     pylab.subplot(5,5,i)
125     pylab.imshow(reconA10[i].reshape(m,n)+im_media)
126     pylab.axis('off')
127
128 fig7 = pylab.figure()
129 pylab.gray()
130 pylab.axis('off')
131 fig7.suptitle('reconstrucción_PCA_2_modos')
132 for i in range(NIM):
133     pylab.subplot(5,5,i)
134     pylab.imshow(reconA2[i].reshape(m,n)+im_media)
135     pylab.axis('off')
136
137 pylab.show()
```

En la figura 14, por ejemplo, se muestra la primera imagen de la secuencia (arriba izquierda), así como las reconstrucciones parciales a partir de 24 modos (arriba derecha), 10 modos (abajo izquierda) o 2 modos (abajo derecha). Resulta evidente que a medida que se va reduciendo el número de modos, la reconstrucción de la cara es más diferente a la imagen original. De hecho, una reconstrucción con 24 modos es casi perfecta, lo que resulta evidente puesto que el número de imágenes NIM = 25 y, como explicamos anteriormente, sólo hay 25 vectores propios con valores propios no nulos.

Figura 14. Reconstrucción PCA de la primera cara del ejemplo 3.7



Figura 14

Arriba izquierda: Imagen original. Arriba derecha: Reconstrucción PCA a partir de 24 modos. Abajo izquierda: Reconstrucción PCA 10 modos. Abajo derecha: Reconstrucción PCA 2 modos.

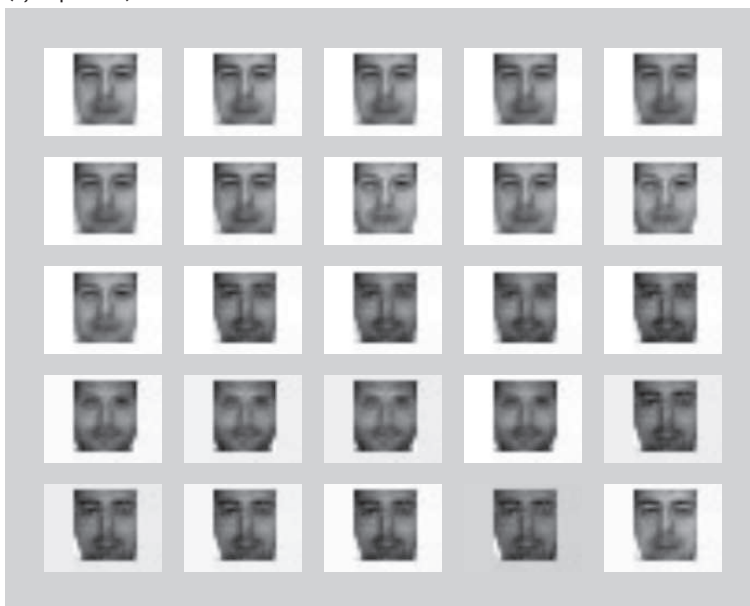
Las figuras 15 y 16 muestran la reconstrucción del conjunto de datos completo a partir de 10 y 2 modos, respectivamente. Comparando con la secuencia de imágenes original (figura 13), resulta evidente que la reconstrucción con 10 modos resulta suficiente para capturar las diferentes expresiones faciales de la secuencia original. Una reconstrucción a partir de 2 modos, en cambio, presenta deficiencias evidentes y sólo captura los rasgos generales de las caras, sin permitir una distinción clara de las expresiones faciales o de las características de cada una de las personas. En este contexto, la técnica PCA se utiliza tanto para reducir el tamaño de las imágenes*, como para disponer de un conjunto reducido de características que permitan discriminar entre diferentes personas o diferentes expresiones faciales. El número de *eigenfaces* escogidas dependerá de los objetivos concretos de cada aplicación.

*Reducir el número de componentes equivale a reducir el número de componentes no nulos de la matriz de datos y por lo tanto del tamaño necesario para almacenar los datos.

Figura 15. Reconstrucción de la secuencia de imágenes a partir de 10 modos PCA (Ejemplo 3.7)



Figura 16. Reconstrucción de la secuencia de imágenes a partir de 2 modos PCA (Ejemplo 3.7)



3.1.3. Análisis de componentes independientes (ICA)

Ejemplo de aplicación

Como ejemplo de aplicación de esta técnica, consideremos una fiesta en la que varias personas hablan de forma simultánea. En esta fiesta, cada persona percibe una señal acústica en la que se mezclan las voces del resto de los asistentes. Para que dos personas puedan mantener una conversación, es necesario que cada uno identifique las palabras emitidas por su interlocutor. Como

cada persona recibe una mezcla de todos los asistentes, el cerebro debe realizar algún tipo de descomposición que le permita identificar la señal emitida por la persona con la que está hablando en ese momento. El cerebro realiza esta separación, además de por características acústicas (intensidad de la voz, tono) o visuales (movimiento de los labios), por el hecho de que cada persona emite una señal que, *estadísticamente*, es *independiente* de la del resto de los asistentes. En efecto, aunque las conversaciones compartan una temática común, el hecho de que cada asistente es un sistema físico independiente garantiza que sus señales sean estadísticamente independientes del resto.

El análisis de componentes independientes (ICA) es una técnica estadística que permite identificar fuentes estadísticamente independientes a partir de un conjunto de señales mezcladas. En otras palabras, ICA es una técnica de gran utilidad que permite descomponer una determinada señal en las fuentes *independientes* que la componen. A ICA también se la conoce como *separación ciega de fuentes**. En términos generales, ICA es una técnica que utiliza la *independencia estadística* para identificar las diferentes fuentes que contribuyen a una determinada mezcla de señales.

Conviene resaltar aquí la importancia del término *independiente*. Las técnicas descritas anteriormente (SVD, PCA) descomponían en factores que no eran necesariamente independientes entre sí desde un punto de vista estadístico.

Descripción de la técnica

Como hemos comentado anteriormente, ICA es un método que resulta útil cuando las fuentes a identificar provienen de sistemas distintos que emiten señales independientes. Esa es la primera hipótesis de aplicación de este método. La segunda condición necesaria es disponer de un número de señales mezcladas igual al número de componentes independientes que se desea identificar.

El subapartado siguiente está dedicado a aclarar algunos aspectos básicos sobre la estadística de señales, y en particular sobre los conceptos de independencia y correlación estadística. Volveremos al ejemplo de aplicación de la técnica ICA más adelante en este subapartado. Aquellos estudiantes que dispongan de conocimientos sobre estadística y variables aleatorias pueden dirigirse directamente a la descripción del código ICA en Python y al ejemplo de aplicación descrito en el subapartado 3.1.3.

Independencia y correlación estadística

En el lenguaje coloquial, a dos señales que no están correlacionadas se les suele llamar independientes. Desde el punto de vista matemático, la afirmación anterior no es cierta: independencia y correlación son propiedades estadísticas diferentes.

*En inglés, *blind source separation (BSS)*.

Ved también

En el subapartado siguiente explicaremos con mayor detalle qué se entiende por *estadísticamente independiente* desde un punto de vista matemático, lo que nos permitirá formular y comprender las bases de la técnica ICA.

Independencia y falta de correlación

Dos variables estadísticamente independientes no presentan correlación, pero la ausencia de correlación no implica necesariamente independencia estadística. Independencia y falta de correlación son por tanto conceptos matemáticamente diferentes.

En concreto, la independencia estadística es una condición más restrictiva que la ausencia de correlación, ya que dos señales independientes necesariamente estarán descorrelacionadas pero dos señales descorrelacionadas no tiene porqué ser independientes.

Para formular este concepto de forma precisa hay que trabajar con las *densidades de probabilidad* de cada una de las señales. Supongamos, pues, que conocemos los n valores $\{x_1, x_2, \dots, x_n\}$ que una señal $x(t)$ toma en un conjunto discreto de tiempos $\{t_1, t_2, \dots, t_n\}$. El *histograma* de valores de $x(t)$ se construye a partir del número de instantes t_i en que la señal toma valores en un determinado intervalo. Si los valores máximo y mínimo de la señal vienen dados por x_{min} y x_{max} respectivamente, dividiremos el rango de valores que toma la señal $x(t)$ en m intervalos de tamaño $\Delta x = (x_{max} - x_{min})/m$. El intervalo k -ésimo viene dado por

$$b_k = [x_{min} + (k-1)\Delta x, x_{min} + k\Delta x], k = 1 \dots m, \quad (23)$$

de forma que el primer intervalo es $b_1 = [x_{min}, x_{min} + \Delta x]$ y el último $b_m = [x_{max} - \Delta x, x_{max}]$. Si la señal $x(t)$ toma un total de h_k valores en el intervalo b_k , el histograma de $x(t)$ vendrá dado por $\{h_1, h_2, \dots, h_m\}$. El código 3.8 indica cómo calcular histogramas y función de densidad de probabilidad en Python. En este ejemplo se parte de un conjunto de datos gaussianos con media $\mu = 200$ y desviación típica $\sigma = 30$.

Código 3.8: Cálculo de histograma y función densidad de probabilidad en Python

```

1 import numpy as np
2 import matplotlib.mlab as mlab
3 import pylab
4
5 # datos gaussianos media mu desviación estandard sigma:
6 mu, sigma = 200, 30
7 y_datos = mu + sigma*np.random.randn(10000)
8
9 # histograma recuento de valores en cada bin:
10 nbins = 50 #número de intervalos
11
12 fig1 = pylab.figure()
13 n, bins, patches = pylab.hist(y_datos, nbins, normed=0, facecolor='
14 green', alpha=0.75)
15 pylab.title(r'\mathrm{Histograma de los datos gaussianos} \mu=200, \sigma
16 =30')
17 pylab.ylabel('Contaje')
18 pylab.xlabel('Intervalos (bin)')
19 pylab.show()
20
21 # histograma normalizado de los datos (densidad de probabilidad)
22 fig2 = pylab.figure()
23 p, bins, patches = pylab.hist(y_datos, nbins, normed=1, facecolor='blue
24 ', alpha=0.75)
25
26 # Ajuste de los datos con una función de densidad

```

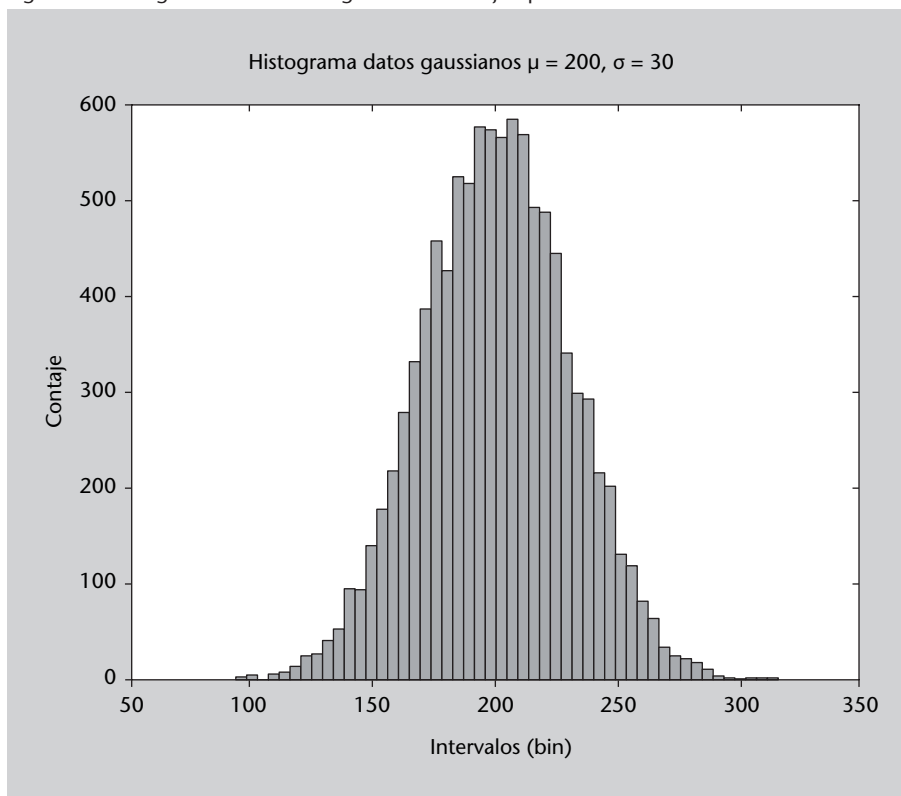
```

24 # de probabilidad gaussiana:
25 y_pdf = mlab.normpdf(bins, mu, sigma)
26 l = pylab.plot(bins, y_pdf, 'r--', linewidth=1)
27 pylab.ylabel('Probabilidad')
28 pylab.xlabel('Intervalos_(bin)')
29 pylab.title(r'$\mathrm{Densidad\_de\_probabilidad}\_\mu=200,\_\sigma=30$'
30 )
31 pylab.grid(True)
32 pylab.show()
33 # Comprobar que la integral de la función densidad de probabilidad
34 # es igual a 1:
35
36 Integral = sum(p*np.diff(bins))
37 print Integral

```

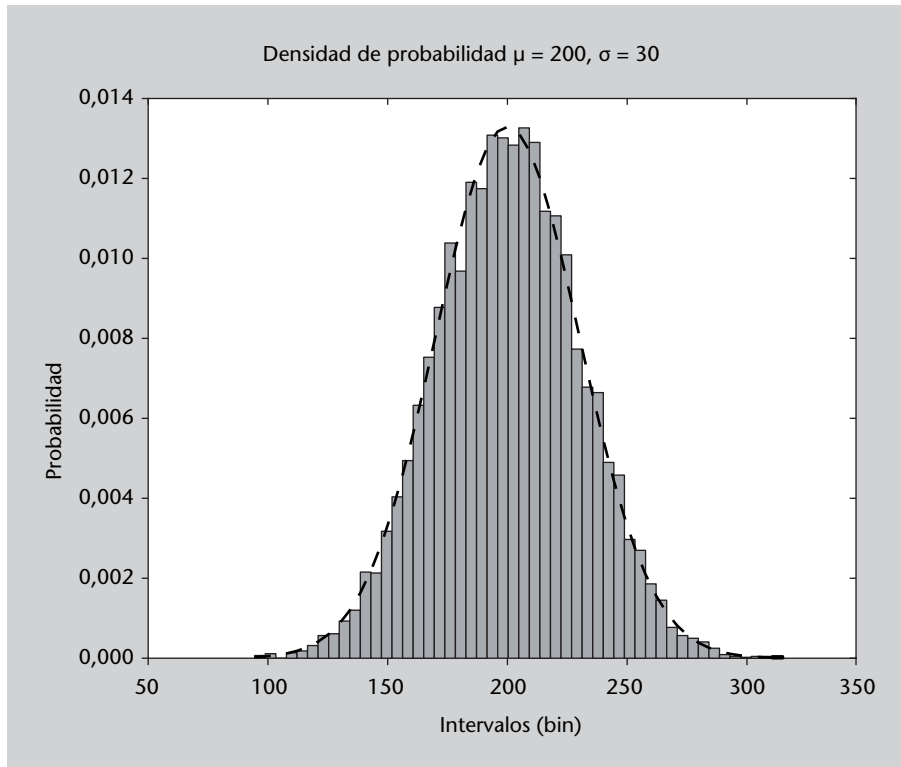
La figura 17 representa el número de datos que toman valores dentro de cada intervalo (contaje).

Figura 17. Histograma de los datos gaussianos del ejemplo 3.8



En la figura 18 el histograma de los datos ha sido normalizado al número de valores, de forma que lo que se representa es la función de densidad de probabilidad. La línea discontinua indica la forma que tiene una distribución de probabilidad gaussiana con media $\mu = 200$ y desviación típica $\sigma = 30$.

Figura 18. Densidad de probabilidad de los datos del ejemplo 3.8



La probabilidad de que la señal $x(t)$ tome un valor en el intervalo b_k vendrá dada por el cociente entre el número de veces que $x(t)$ toma valores en el intervalo b_k y el número total de valores de la señal, n . Nos referiremos entonces a la *función probabilidad* de $x(t)$ como

$$p_k = \frac{h_k}{n}. \quad (24)$$

Como cabría esperar, la suma de todas las probabilidades $\sum_{k=1}^m p_k = 1$ es un puesto que corresponde a la probabilidad de que la señal adopte un valor en algún intervalo b_k .

La *densidad de probabilidad* $\rho_x(k)$ en un intervalo b_k se define entonces como el número de valores que se han producido en dicho intervalo h_k normalizado con el área total del histograma

$$\rho_x(k) = \frac{h_k}{n\Delta x}. \quad (25)$$

En el límite en el que el tamaño del intervalo Δx se hace infinitesimalmente pequeño (o, equivalentemente, $m \rightarrow \infty$) y el número de valores de la señal n tiende a infinito, $\rho_x(k)$ da lugar a la *función densidad de probabilidad continua*

$$\rho_x(x) = \lim_{\Delta x \rightarrow 0, m \rightarrow \infty} \rho_x(k). \quad (26)$$

Volvamos al asunto de la independencia y a la ausencia de correlación estadística entre dos señales $x(t)$ e $y(t)$. De forma similar a lo que hicimos anteriormente, podemos dividir los rangos de valores que toman las variables x e y en m intervalos de tamaño Δx y Δy respectivamente. Entonces podemos definir la *densidad de probabilidad conjunta* $\rho_{xy}(x,y)$ de $x(t)$ e $y(t)$ a partir del número de veces que la señal x e y tomen valores en diferentes intervalos *de forma simultánea*. Esta función nos indica, por ejemplo, en cuántos instantes de tiempo t_1, t_2, \dots, t_n la señal x ha tomado valores en el primer intervalo y en el último, lo que nos permite conocer la probabilidad de que se produzcan un par de valores (x,y) a la vez.

Cuando las señales x e y son *estadísticamente independientes*, la densidad de probabilidad conjunta $\rho_{xy}(x,y)$ viene dada por el producto de las *densidades de probabilidad marginales* de cada una de las señales $\rho_x(x)$ y $\rho_y(y)$

$$\rho_{xy}(x,y) = \rho_x(x)\rho_y(y), \quad (27)$$

puesto que, según el teorema de Bayes, la probabilidad de que se dé el par de valores (x_0, y_0) es el producto de las probabilidades de que x_0 se dé en x y de que y_0 se dé en y de forma independiente. Cuando las señales no son independientes tenemos, en cambio que

$$\rho_{xy}(x,y) = \rho_x(x)\rho_y(y) + \rho_{xy}(x|y) \quad (28)$$

donde $\rho_{xy}(x|y)$ es la *probabilidad condicionada* de que se dé un valor de x habiéndose dado un valor y . De la expresión anterior se deriva que *los momentos centrales conjuntos* de cualquier orden r,s cumplen la condición

$$E[(x - \bar{x})^r (y - \bar{y})^s] = E[(x - \bar{x})^r] E[(y - \bar{y})^s] \quad (29)$$

para cualquier valor entero de r,s . En particular, para $r = s = 1$ tenemos que la *covarianza* $Cov(x,y) = E[(x - \bar{x})(y - \bar{y})]$ entre x e y es nula

$$Cov(x,y) = E[(x - \bar{x})(y - \bar{y})] = E[(x - \bar{x})] E[(y - \bar{y})] = 0, \quad (30)$$

Ved también

El anexo de este módulo se dedica a repasar los conceptos básicos de estadística que son necesarios en este apartado.

puesto que los primeros momentos centrales de cada una de ellas son nulos

$$E[(x - \bar{x})] = \int_{-\infty}^{\infty} (x - \bar{x})\rho_x(x)dx = \int_{-\infty}^{\infty} x\rho_x(x)dx - \bar{x} = 0 \quad (31)$$

$$E[(y - \bar{y})] = \int_{-\infty}^{\infty} (y - \bar{y})\rho_y(y)dy = \int_{-\infty}^{\infty} y\rho_y(y)dy - \bar{y} = 0. \quad (32)$$

En efecto, cuando dos señales son estadísticamente independientes, su covarianza es nula.

A partir de la covarianza se puede definir la *correlación* entre las dos señales

$$\text{Corr}(x,y) = \frac{\text{Cov}(x,y)}{\sigma_x\sigma_y} \quad (33)$$

que da cuenta de la covarianza normalizada con las desviaciones estándar de cada una de las señales σ_x y σ_y .

El código 3.9 describe cómo calcular las matrices de covarianza y correlación entre dos variables.

Código 3.9: Cálculo de matrices de covarianza y correlación entre dos conjuntos de datos

```

1 from scipy import stats
2 from numpy import *
3
4 # datos:
5 mu1, sigma1 = 200, 30
6 x = mu1 + sigma1*random.randn(3).T
7 mu2, sigma2 = 10, 5
8 y = mu2 + sigma2*random.randn(3).T
9
10 # matriz de covarianza:
11 covmat = cov(x,y,bias=1)
12
13 # matriz de correlación:
14 corrmatrix = corrcoef(x,y)
15
16 >>> print covmat
17 [[ 902.94004704 -86.47788485]
18  [-86.47788485  26.52406588]]
19
20 >>> print var(x), var(y)
21 902.940047043 26.5240658823
22
23 >>> print corrmatrix
24 [[ 1.          -0.55879891]
25  [-0.55879891  1.          ]]
26
27 # relación entre ambas:
28 >>> cov(x/std(x),y/std(y),bias=1)
29 array([[ 1.          , -0.55879891],
30        [-0.55879891,  1.          ]])

```

Todo esto nos lleva a concluir que dos señales independientes tienen una correlación nula. Lo que no es necesariamente cierto es que dos señales con correlación nula sean independientes. Para que sean independientes, además de la correlación también deben ser nulos los momentos centrales conjuntos de cualquier orden que cumplan la ecuación 29.

Por ejemplo, para $r = s = 2$ tenemos, utilizando la ecuación 71 que

$$E[(x - \bar{x})^2(y - \bar{y})^2] = E[(x - \bar{x})^2]E[(y - \bar{y})^2] = \text{Var}[x]\text{Var}[y] \quad (34)$$

Información mutua

La información mutua es una medida de la relación estadística entre dos señales. De forma cualitativa, la información mutua puede entenderse como una función de correlación que tiene en cuenta relaciones lineales y no-lineales entre dos variables.

En concreto, lo que mide la información mutua es la probabilidad de que una variable tome un valor determinado condicionada a que la otra variable haya tomado otro. O lo que es lo mismo, la cantidad de información que tenemos de una de ellas en caso de conocer la otra.

Para eso introducimos el concepto de *entropía de Shannon* de una señal x , definida como el valor esperado del logaritmo de su densidad de probabilidad

$$I(x) = -E[\log(\rho_x(x))] = - \int \rho_x(x) \log(\rho_x(x)) dx. \quad (35)$$

Cuanto mayor es la entropía de Shannon de una señal, menor es su contenido informacional.

De esta forma una señal aleatoria presentará mayor entropía y menor contenido informacional que una señal periódica. En el ámbito de las comunicaciones, la ecuación 35 permite cuantificar la información que se transmite a través de un determinado canal. De forma equivalente, es posible definir la

información mutua entre dos señales como el valor esperado del logaritmo de su distribución conjunta

$$I(x,y) = -E[\log(\rho_{xy}(x,y))] = -\int \int \rho_{xy}(x,y) \log(\rho_{xy}(x,y)) dx dy \quad (36)$$

En el caso de que x e y sean señales estadísticamente independientes, la probabilidad conjunta factoriza como el producto de las probabilidades marginales $\rho_{xy}(x,y) = \rho_x(x)\rho_y(y)$ (ecuación 27), y la información mutua viene dada por la

$$I(x,y) = -E[\log(\rho_{xy}(x,y))] = -E[\log(\rho_x(x))]E[\log(\rho_y(y))] = I(x)I(y) \quad (37)$$

Cuando ambas señales tienen alguna relación, sea lineal o no lineal, la probabilidad condicional $\rho_{xy}(x|y)$ no es nula, por lo que se cumple la ecuación 28 y la información mutua toma valores positivos $I(x,y) = I(x)I(y) + I(x|y) > I(x)I(y)$.

Puesto que la definición incluye directamente la densidad de probabilidad conjunta, la entropía mutua tiene en cuenta todos los momentos conjuntos de las distribuciones y no solamente la covarianza, y por lo tanto, la información mutua es una medida de la independencia estadística entre dos señales.

ICA descompone una matriz de datos en componentes estadísticamente independientes.

Ejemplo: separación de fuentes independientes

Las diferentes implementaciones de ICA obtienen las señales independientes de una mezcla mediante un proceso de minimización de la información mutua. Adicionalmente, la mayoría de los métodos incluyen alguna estrategia que garantice que las fuentes, además de independientes, sean lo más simples posible (principio de parsimonia). Para ello utilizan diferentes medidas de complejidad computacional de las señales fuente obtenidas.

La versión que vamos a utilizar está contenida en la librería python de acceso libre MDP*, y utiliza este procedimiento que garantiza fuentes independientes y de baja complejidad. La versión utilizada implementa el algoritmo FASTICA desarrollado por Aapo Hyvarinen.

El código 3.10 describe la utilización de la técnica ICA en Python. El programa empieza definiendo dos señales sinusoidales con frecuencias diferentes. En el ejemplo introductorio, cada una de estas señales se corresponderían a las voces de dos de los participantes en la fiesta. Estas señales aparecen representadas en la figura 19 y serán consideradas como las fuentes originales a identificar mediante la técnica ICA.

*MDP (Modular toolkit for Data Processing)
<http://mdp-toolkit.sourceforge.net/>.

Lectura recomendada

Aapo Hyvarinen. (1999). Fast and Robust Fixed-Point Algorithms for Independent Component Analysis IEEE Transactions on Neural Networks, 10(3):626-634, .
A. Hyvarinen, E. Oja (1997). A fast Fixed-Point Algorithms for Independent Component Analysis Neural Computation, 9(7)1483-1492.
<http://www.cis.hut.fi/projects/ica/fastica/>

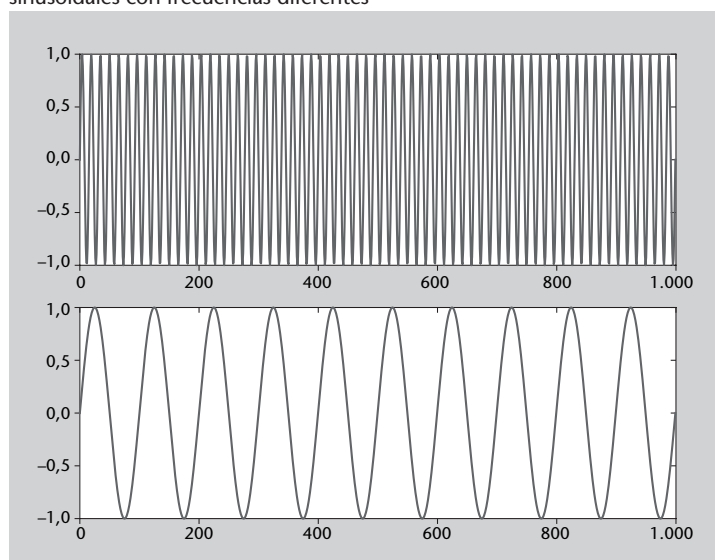
Código 3.10: Ejemplo de descomposición ICA en Python

```

1 import numpy as np
2 import pylab as py
3 import scipy as sc
4 import mdp
5 from mdp import fastica
6
7 # Fuentes originales
8 t = np.linspace(0,10,1000)
9 Sa = np.sin(2*sc.pi*t)
10 Sb = np.sin(13*sc.pi*t)
11
12 S1 = Sa.reshape(1,-1)
13 S0 = Sb.reshape(1,-1)
14 S = np.concatenate((S0,S1))
15
16 # Datos mezclados
17 A = [[1, 4], [-3, 2]] # Matriz de mezcla
18
19 X = np.dot(A, S) # Observaciones
20
21 # Descomposicion ICA:
22
23 y_ICA = mdp.fastica(X.T, dtype='float32')
24
25 # representacion de resultados:
26 fig1 = py.figure()
27 py.subplot(211)
28 py.plot(S.T[:,0])
29 py.subplot(212)
30 py.plot(S.T[:,1])
31 fig1.suptitle('Fuentes_originales')
32
33 fig2 = py.figure()
34 py.subplot(211)
35 py.plot(X.T[:,0])
36 py.subplot(212)
37 py.plot(X.T[:,1])
38 fig2.suptitle('Fuentesmezcladas')
39
40 fig3 = py.figure()
41 py.subplot(211)
42 py.plot(y_ICA[:,0])
43 py.subplot(212)
44 py.plot(y_ICA[:,1])
45 fig3.suptitle('Componentes_independientes_(ICA)')
46
47 py.show()

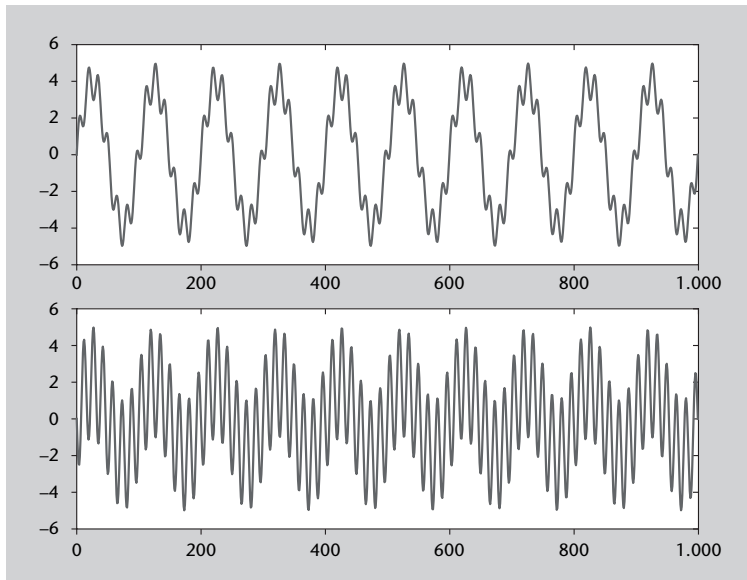
```

Figura 19. En el código 3.10, las fuentes originales son dos señales sinusoidales con frecuencias diferentes



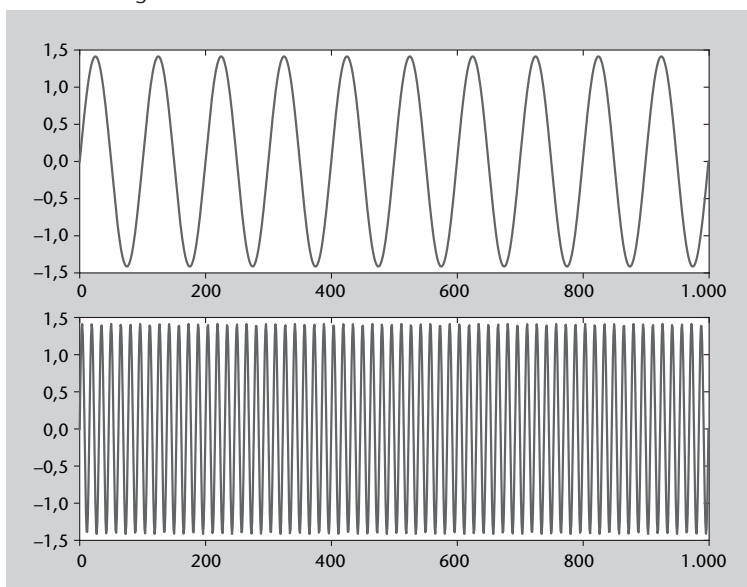
A continuación se construyen los datos mezclados aplicando una matriz de mezclas A a las fuentes originales. En el ejemplo de la fiesta, los datos mezclados corresponden a la señal total mezclada que oíría cada uno de los asistentes. Como resultado, se obtienen dos señales de mezcla que son combinación lineal de las fuentes originales y que se representan en la figura 20.

Figura 20. Resultado de mezclar las fuentes originales con una matriz de mezcla. 3.10



La descomposición ICA se aplica a estas dos señales mezcladas con la intención de poder identificar las fuentes independientes subyacentes. El resultado de la descomposición son las dos señales representadas en la figura 21, y como puede verse, cada una de ellas corresponde a una de las dos fuentes originales de partida. El código puede utilizarse para explorar los resultados obtenidos con la técnica ICA al aplicarla a otro tipo de señales y con matrices de mezcla diferentes.

Figura 21. Los modos independientes identificados por la técnica ICA recuperan las fuentes originales 3.10



3.1.4. Factorización de matrices no-negativas (NMF)

Ejemplo de aplicación

NMF es una herramienta utilizada de forma habitual en aplicaciones de *minería de textos*, por lo que el ejemplo más representativo es el análisis de aparición de palabras en textos. El método NMF resulta útil para caracterizar conjuntos de datos que no tomen valores negativos, como por ejemplo, datos de recuento (número de votos por candidatura, aparición de palabras en documentos, frecuencia de eventos en una planta industrial, etc.). Supongamos que tenemos un conjunto de N textos que queremos caracterizar mediante un subconjunto de K palabras. El sistema puede representarse mediante una matriz de ocurrencias A de tamaño $N \times K$ cuyos componente i, j indicasen el número de veces que la palabra j aparece en el texto i .

$$\begin{array}{r}
 \text{palabra 1} \quad \text{palabra 2} \quad \dots \quad \text{palabra } K \\
 \text{texto 1} \\
 \text{texto 2} \\
 \dots \\
 \text{texto } N
 \end{array}
 \begin{pmatrix}
 a_{1,1} & a_{2,1} & \dots & a_{1,K} \\
 a_{2,1} & a_{2,2} & \dots & a_{2,K} \\
 \vdots & \vdots & \ddots & \vdots \\
 a_{N,1} & a_{N,2} & \dots & a_{N,K}
 \end{pmatrix}$$

Eligiendo las K palabras de forma estratégica podremos caracterizar los textos según el número de veces que aparezca cada palabra. Por ejemplo, en caso de que los textos provengan de una web de noticias, parece lógico que aquellos textos en los que más veces aparezca la palabra *internacional* o *Europa* estarán referidos a noticias de ámbito internacional, mientras que en los que aparezcan las palabras *alcalde* o *ayuntamiento* tendrán que ver con noticias de ámbito local. Evidentemente, también puede suceder que una noticia local sobre ayudas agrarias incluya la palabra *Europa*, o que una de ámbito internacional se refiera al alcalde de una determinada ciudad. El método NMF supera esta ambigüedad definiendo un conjunto de características que permitan agrupar los textos de forma eficiente. En resumen, NMF nos va a permitir agrupar los textos siguiendo un criterio que permita identificarlos mediante la aparición de palabras clave.

Descripción de la técnica

NMF realiza una descomposición en factores una matriz de datos no-negativa A de la forma

$$A \approx W \cdot F \quad (38)$$

donde las matrices no-negativas F y W son conocidas como *matriz de características* y *matriz de pesos* respectivamente. La matriz de características F , de

tamaño $M \times K$, define M características y pondera cada una de ellas mediante la importancia que tiene cada una de las K palabras. Las características corresponden a temas genéricos que han sido definidos a partir de la agrupación de palabras en diferentes textos. La componente i,j de F representa la importancia de la palabra j en la característica i . Su aspecto sería el siguiente

$$\begin{array}{l}
 \text{palabra 1} \quad \text{palabra 2} \quad \dots \quad \text{palabra } K \\
 \text{característica 1} \\
 \text{característica 2} \\
 \dots \\
 \text{característica } M
 \end{array}
 \begin{pmatrix}
 f_{1,1} & f_{1,2} & \dots & f_{1,K} \\
 f_{2,1} & f_{2,2} & \dots & f_{2,K} \\
 \vdots & \vdots & \ddots & \vdots \\
 f_{M,1} & f_{M,2} & \dots & f_{M,K}
 \end{pmatrix}$$

Por otra parte, la matriz de pesos W le atribuye un peso a cada una de las características en función de su importancia en cada uno de los N textos. Sus M columnas corresponden a las características y sus N filas a los textos analizados, de forma que la componente i,j de W indica la importancia que tiene la característica j en el texto i

$$\begin{array}{l}
 \text{característica 1} \quad \text{característica 2} \quad \dots \quad \text{característica } M \\
 \text{texto 1} \\
 \text{texto 2} \\
 \dots \\
 \text{texto } N
 \end{array}
 \begin{pmatrix}
 w_{1,1} & w_{1,2} & \dots & w_{1,M} \\
 w_{2,1} & w_{2,2} & \dots & w_{2,M} \\
 \vdots & \vdots & \ddots & \vdots \\
 w_{N,1} & w_{N,2} & \dots & w_{N,M}
 \end{pmatrix}$$

La ecuación 38 debe interpretarse como una transformación que permite definir M características de forma que los datos originales puedan factorizarse en la forma

$$\text{textos} \times \text{palabras} \approx \text{textos} \times \text{características} \cdot \text{características} \times \text{palabras}. \quad (39)$$

Factorización

NMF permite factorizar una matriz de datos definida positiva en el producto de dos matrices no-negativas con una estructura más simple.

Evidentemente, uno de los parámetros que deben definirse para aplicar NMF es el número de características M que se desean. Un valor demasiado alto resultaría en un etiquetado demasiado específico de los textos, mientras que un valor excesivamente pequeño agruparía los textos en unas pocas características demasiado genéricas y por tanto carentes de información relevante. La matriz de características F es equivalente a la matriz de vectores propios que obteníamos con otras técnicas de factorización de datos como PCA o ICA. La diferencia principal entre NMF y PCA radica en que en NMF la matriz F es no-negativa y por tanto la factorización de A puede interpretarse como una descomposición acumulativa de los datos originales.

Conviene precisar que la descomposición NMF no es única, es decir, dado un número de características M hay más de una forma de descomponer los datos en la forma de la ecuación 38. Lo ideal es aplicar el procedimiento e interpretar los resultados para obtener información adicional de los datos. El algoritmo que se encarga de realizar esta factorización no es trivial y escapa al alcance de este libro. En términos generales, las diferentes implementaciones de NMF siguen un proceso de optimización multidimensional que minimiza el residuo R de la descomposición

$$R = W \cdot F - A. \quad (40)$$

Implementación en Python

Aquí utilizaremos una implementación que se describe en el libro *Programming Collective Intelligence* de T. Segaran. El código de la rutina *nmf.py* allí descrito es el que aparece en el código 3.11.

Código 3.11: Implementación de NMF en Python descrita en el libro *Programming Collective Intelligence*

```

1 from numpy import *
2
3 def difcost(a,b):
4     dif=0
5     # Loop over every row and column in the matrix
6     for i in range(shape(a)[0]):
7         for j in range(shape(a)[1]):
8             # Add together the differences
9             dif+=pow(a[i,j]-b[i,j],2)
10    return dif
11
12 def factorize(v,pc=10,iter=50):
13     ic=shape(v)[0]
14     fc=shape(v)[1]
15     # Initialize the weight and feature matrices with random values
16     w=matrix([[random.random() for j in range(pc)] for i in range(ic)])
17     h=matrix([[random.random() for i in range(fc)] for i in range(pc)])
18     # Perform operation a maximum of iter times
19     for i in range(iter):
20         wh=w*h
21     # Calculate the current difference
22     cost=difcost(v,wh)
23     if i%10==0: print cost
24     # Terminate if the matrix has been fully factorized
25     if cost==0: break
26     # Update feature matrix
27     hn=(transpose(w)*v)
28     hd=(transpose(w)*w*h)
29     h=matrix(array(h)*array(hn)/array(hd))
30     # Update weights matrix
31     wn=(v*transpose(h))
32     wd=(w*h*transpose(h))
33     w=matrix(array(w)*array(wn)/array(wd))
34     return w,h

```

El código 3.12 presenta un ejemplo de utilización de la rutina *nmf.py*. Primero se define una matriz de datos A , que en este caso contiene datos aleatorios de

Lectura complementaria

T. Segaran (2007).
Programming Collective Intelligence. O'Really

contaje de 10 palabras en 100 textos. A continuación se procede a factorizar la matriz A utilizando NMF. En este caso hemos utilizado un total de 3 características. La validez del resultado puede analizarse comprobando el residuo de la aproximación, es decir las diferencias entre la matriz de datos original A y la aproximación NMF $W \cdot F$.

Código 3.12: Ejemplo de utilización del método NMF

```

1 from numpy import *
2 import nmf
3
4 # Matriz de datos aleatoria (100 textos x 10 palabras):
5 A = random.randint(0,100,(100,10))
6
7 #descomposición NMF utilizando 3 características:
8 W,F = nmf.factorize(A,pc=3,iter=500)
9
10 print(W)
11 print(F)
12
13 # Residuo de la aproximación:
14 R = A - dot(W,F)
15
16 # máximo error relativo de reconstrucción (%):
17 R_rel = 100*R/A
18 print(R_rel.max())

```

Otras implementaciones

Existen otras implementaciones que utilizan algoritmos de optimización basados en el método de gradientes proyectados, como la que puede obtenerse en la dirección web <http://www.csie.ntu.edu.tw/~cjlin/nmf/> basada en el artículo siguiente: C.-J. Lin. *Projected gradient methods for non-negative matrix factorization*. *Neural Computation*, 19(2007), 2756-2779.

Análisis de textos

Como hemos comentado, una de las aplicaciones más importantes de la técnica NMF es el análisis de documentos. Para poder construir la matriz de recuento de palabras, primero hay que realizar algunas operaciones de filtrado de los textos obtenidos. Supongamos que disponemos de un conjunto de ficheros de texto en formato .txt. Cada texto puede leerse mediante la rutina Python descrita en el código 3.13.

Código 3.13: Lectura de ficheros de texto y creación de diccionarios en Python

```

1 # -*- coding: cp1252 -*-
2 import glob
3 import numpy
4 import string
5
6 def textos(path):
7     # listar todos los ficheros con extensión .txt del directorio path:
8     filelist = glob.glob(path)
9     docs = []
10
11     # bucle sobre todos los ficheros en el directorio:
12     for kfile in filelist:
13         fp = file(kfile) # abrir fichero
14         txt = fp.read() # leer fichero
15         fp.close() # cerrar fichero

```

```

16
17     # filtrar caracteres:
18     txt = txt.lower() # convertir a minusculas
19     # substituir retornos de carro por espacios
20     txt = txt.replace('\n', '_')
21     # borrar caracteres ASCII que no sean el espacio
22     #(ascii 32), o las letras a-z (ascii 97-122):
23     char_ok = [32] + range(97,122)
24     for i in range(256):
25         if i not in char_ok:
26             txt = txt.replace(chr(i), '')
27
28     # construir una lista con las palabras del texto:
29     palabras = txt.split()
30     # crear un diccionario:
31     dicc = { }
32     for ipal in range(len(palabras)):
33         pl = palabras[ipal];
34         if len(pl)>4: # excluir palabras de menos de 5 caracteres
35             if pl in dicc:
36                 dicc[pl].append(ipal) # localización de las
37                                     apariciones
38             else:
39                 dicc[pl] = [ipal]
40     # añadir número de apariciones al final del diccionario
41     for ipal in range(len(palabras)):
42         pl = palabras[ipal];
43         if pl in dicc:
44             #dicc[pl].append(len(dicc[pl])) # número de apariciones
45             dicc[pl].insert(0, len(dicc[pl]))
46     # agregar diccionario de cada documento
47     docs.append(dicc)
48     return docs

```

El código 3.13 utiliza un *string* de entrada *path* en el que se indique el directorio en el que están ubicados los ficheros de texto. En las líneas 6-9 del código genera una lista *filelist* con los nombres de los ficheros de texto. El bucle de la línea 12 recorre todos los ficheros de texto y ejecuta la secuencia de acciones siguientes para cada uno de ellos: abrir fichero, leer fichero, cerrar fichero, eliminar todos los caracteres ASCII que sean diferentes a las letras a-z o el espacio (líneas 23-26). Una vez filtradas, se construye una lista con las palabras que aparecen en el texto mediante la función Python *split*.

El siguiente paso es crear un diccionario Python *dicc*, que consiste en una lista de las palabras que aparecen en el texto que va acompañada de las posiciones en las que aparece cada palabra (líneas 30-38). En el ejemplo sólo se han incorporado al diccionario las palabras con más de cuatro caracteres. Al principio de esta lista añadimos un nuevo valor que indica el número de veces que cada palabra aparece en el texto, por lo que cada entrada del diccionario final contiene la palabra, y a continuación un vector con el número de veces que aparece en el texto y las posiciones del texto en las que aparece. La última instrucción del bucle de ficheros incorpora el diccionario de cada texto en una variable biblioteca *docs* que agrupa todos los diccionarios (un diccionario por fichero de texto). La rutina retorna la variable *docs* con todos los diccionarios de todos los textos encontrados en la carpeta *path*.

El código 3.14 indica la instrucción necesaria para generar una biblioteca de diccionarios a partir de los ficheros de texto ubicados en la carpeta *C:/textos/*.

Código 3.14: Llamada a la rutina textos para generar una biblioteca de diccionarios a partir de un conjunto de ficheros de texto

```

1 >>> docs = textos('C:\\textos\\*.txt')
2
3 >>> docs[0]
4 {'arrhythmogenic': [3, 2, 11, 135], 'catecholaminergic': [1, 147], '
  significant': [1, 80], 'family': [1, 52], 'providers': [1, 67], '
  abridge': [1, 125], 'correlations': [1, 154], 'outcomes': [1, 49],
  'helped': [1, 14], 'namely': [1, 137], 'determined': [1, 100], '
  decisions': [1, 72], 'occur': [1, 85], 'recognition': [1, 105], '
  current': [1, 127], 'polymorphic': [1, 148], 'realm': [1, 33], '
  knowledge': [1, 128], 'mutated': [1, 6], 'implementation': [1,
  114], 'technology': [1, 92], 'coupled': [1, 93], 'pathogenesis':
  [1, 151], 'research': [1, 1], 'better': [1, 55], 'their': [9, 8,
  7, 6, 5, 18, 48, 57, 65, 74], 'genetic': [3, 2, 35, 131], '
  clinical': [1, 87], 'identification': [1, 4], 'treatment': [1,
  116], 'which': [1, 110], 'inherited': [3, 2, 10, 134], 'forward':
  [1, 15], 'progress': [3, 2, 26, 79], 'tachycardia': [1, 150], '
  fatal': [1, 108], 'cause': [1, 9], 'possible': [1, 40], '
  potentially': [1, 107], 'practice': [1, 88], 'genotypephenotype':
  [1, 153], 'diseases': [5, 4, 3, 12, 109, 136], 'advances': [1,
  90], 'genes': [1, 7], 'background': [1, 132], 'symptomatic': [1,
  44], 'understanding': [3, 2, 16, 97], 'pathophysiology': [1, 19],
  'genetically': [1, 99], 'assists': [1, 102], 'understand': [1,
  56], 'discussed': [1, 157], 'members': [1, 53], 'improving': [1,
  96], 'article': [1, 123], 'conjunction': [1, 63], 'continued': [3,
  2, 0, 78], 'earlier': [3, 2, 104, 113], 'improve': [1, 47], '
  leads': [1, 111], 'short': [1, 141], 'decades': [1, 24], 'syndrome
  ': [5, 4, 3, 140, 143, 145], 'arrhythmias': [3, 2, 36, 101], '
  ventricular': [1, 149], 'patients': [1, 45], 'continue': [1, 83],
  'risks': [1, 58], 'allow': [1, 60], 'brugada': [1, 144], 'changes'
  : [1, 81]}

```

En la línea 3 se obtiene el diccionario correspondiente al primero de los textos encontrados en la carpeta. Cada entrada del diccionario consta de un campo para la palabra y de un vector en el que la primera componente indica el número de veces que aparece la palabra en el texto y los siguientes, las posiciones en las que aparece la palabra a lo largo del texto. Por ejemplo, la palabra *continued* aparece un total de tres veces en el primer documento, concretamente en las palabras primera [0], tercera [2] y en la que ocupa la posición 79 [78].

La rutina 3.15 lista las 10 palabras más frecuentes de cada uno de los textos analizados. Con esta información y la biblioteca de diccionarios, podemos construir la matriz de ocurrencias de palabras en textos.

Código 3.15:

```

1 from textos import *
2 import numpy
3 import operator
4
5
6 # obtener un conjunto de diccionarios:
7 docs = textos('C:\\textos\\*.txt')
8
9 # calcular la matriz de recuento de palabras
10 # tamaño N textos x K palabras
11
12 dicc_sort = []
13 # loop over dictionaries
14 for k in range(len(docs)):
15 # ordenar diccionario por número de apariciones
16     ds = sorted(docs[k].iteritems(), key=operator.itemgetter(1), reverse=
        True)

```

```

17 dicc_sort.append(ds)
18 # Seleccionar un grupo de K palabras que
19 # apezcen en los diferentes diccionarios:
20 print k
21 # listar las 10 palabras más utilizadas en cada texto:
22 for kw in range(1,10):
23     print ds[kw][0]

```

Consideremos una matriz de ocurrencias de $K = 5$ palabras en $N = 4$ textos como la siguiente:

$$A = \begin{matrix} & \text{palabra 1} & \text{palabra 2} & \text{palabra 3} & \text{palabra 4} & \text{palabra 5} \\ \text{texto 1} & \left(\begin{array}{ccccc} 14 & 1 & 3 & 2 & 1 \\ 1 & 2 & 0 & 14 & 1 \\ 15 & 1 & 3 & 2 & 4 \\ 0 & 1 & 3 & 21 & 0 \end{array} \right) \\ \text{texto 2} \\ \text{texto 3} \\ \text{texto 4} \end{matrix}$$

En este caso la primera palabra aparece mucho en los textos 1 y 3 y poco en los otros, mientras que los textos 2 y 4 comparten una alta aparición de la palabra 4. Las matrices F, W resultantes de aplicar el código 3.11 a la matriz anterior se indican en el código 3.16.

Código 3.16:

```

1 >>> F
2 matrix([[ 1.56842028e+01,  1.10030496e+00,  3.18461101e+00,
3          2.07222144e+00,  2.79548846e+00],
4          [ 8.29519007e-14,  6.26129490e-01,  7.91626659e-01,
5            8.31254256e+00,  1.45667562e-01]])
6 >>> W
7 matrix([[ 8.78583342e-01,  2.08132147e-02],
8          [ 5.58125546e-02,  1.66257600e+00],
9          [ 9.69522495e-01,  1.99310161e-05],
10         [ 5.33686016e-03,  2.53004685e+00]])

```

La interpretación de los resultados es la siguiente: La matriz F tiene dos filas que corresponden a cada una de las dos características que se han extraído por la técnica NMF. En esta matriz F , las columnas corresponden a cada una de las palabras. Analizando los valores de F que aparecen en el código 3.16, la primera conclusión a la que llegamos es que la primera característica tiene que ver principalmente con la primera palabra puesto que la componente $F(1,1)$ de la matriz toma un valor elevado. Lo mismo sucede con la segunda característica, que aparece relacionada principalmente a la cuarta palabra (componente $F(2,4)$ de la matriz F). Por lo tanto la matriz F nos describe dos características que permiten analizar los textos, una relacionada con la palabra 1 y otra con la palabra 4. Esta información de hecho es relevante puesto que pone de manifiesto algunas de las características de la matriz de ocurrencias inicial.

La matriz W , a su vez, pondera la importancia de cada una de las dos características en los cuatro textos analizados. La característica 1 (primera columna de W), por ejemplo, adopta valores más altos en los textos 1 y 3 (componen-

tes $W(1,1)$ y $W(3,1)$). De forma similar, la segunda característica (relacionada con la palabra 4), permite caracterizar los textos 2 y 4 (componentes $W(2,2)$ y $W(4,2)$).

En resumen, la descomposición NMF nos ha permitido encontrar dos características que permiten agrupar los textos según los patrones de aparición de palabras en cada uno de ellos. En este ejemplo las características están relacionadas principalmente con una de las palabras, pero en general serán una combinación lineal de algunas de las palabras analizadas. Conviene recordar que la inicialización aleatoria de las matrices W, F que aparece en el código 3.11 ocasiona que los resultados obtenidos sean ligeramente diferentes cada vez que se ejecuta el código. También cabe notar que los resultados pueden ser mejorados mediante el ajuste de parámetros como el número máximo de iteraciones.

3.2. Discriminación de datos en clases

En los subapartados anteriores hemos visto que la técnica PCA sirve fundamentalmente para *caracterizar* un conjunto de datos a partir de las variables que presentan una mayor variabilidad. De igual forma, ICA se utiliza para *identificar* los diferentes mecanismos subyacentes que dan lugar a la estructura de los datos. El análisis de discriminantes sirve para determinar qué variables de un conjunto de datos son las que discriminan entre dos o más clases predefinidas.

En términos generales, puede decirse que PCA es útil para caracterizar, ICA para identificar y LDA para *discriminar*.

Aquí nos limitaremos al caso en el que se pretendan distinguir dos clases diferentes, por lo que lo trataremos como método de *dicotomía*. Nos limitaremos al caso de dos clases porque las expresiones matemáticas son más simples y permiten una comprensión más directa de la técnica. La extensión de la técnica a casos en los que haya que distinguir más de dos grupos no supone una excesiva dificultad adicional.

Dicotomía según la RAE

Según el diccionario de la Real Academia Española *dicotomía* es el método de clasificación en que las divisiones y subdivisiones solo tienen dos partes.

3.2.1. Análisis de discriminantes lineales (LDA)

Ejemplo de aplicación

Es frecuente que un conjunto de datos esté compuesto por observaciones que puedan ser clasificadas en dos grupos. Por ejemplo, imaginemos una base de datos con mediciones biomecánicas de atletas realizadas durante la ejecución de un ejercicio de salto de altura. Resultaría interesante saber qué variables

biomecánicas de los saltadores permiten discriminar entre los atletas que alcanzan el podio del resto de participantes. La base de datos incluirá medidas de varias magnitudes relevantes como la velocidad de salida, la longitud de la primera zancada o la inclinación del cuerpo al realizar el último paso. Estas n variables biomecánicas se medirán para un conjunto de m saltadores durante campeonatos internacionales. Las mediciones darán lugar a un conjunto de n variables y m observaciones, y por tanto, a una matriz de datos de tamaño $m \times n$. A las observaciones deberemos añadir un vector de etiquetas que identifique cada observación como grupo 1 si el saltador ocupó el podio o como grupo 2 si finalizó el campeonato a partir de la cuarta posición.

Nuestro interés es saber qué variables o combinación de las mismas nos permiten identificar a los saltadores que ocupan el podio en competiciones internacionales. Un análisis PCA de estos datos nos permitiría identificar las variables que mejor expresen la variabilidad entre saltadores, pero éstas no son necesariamente las que mejor *discriminarán* entre los saltadores que presentan un mejor rendimiento. Puede suceder, por ejemplo, que la máxima variabilidad se presente en la forma en la que se realiza el primer paso sin que ésta variable tenga un impacto decisivo en la altura final superada por el saltador y por tanto en el hecho de alcanzar el podio.

Descripción de la técnica

El método LDA parte de un conjunto de datos, al que llamaremos *conjunto de entrenamiento*, del que tenemos un conocimiento previo de la asignación de cada una de las observaciones a un grupo. En el ejemplo de los saltadores de altura, el conjunto de entrenamiento consistiría en un conjunto de m observaciones de n variables biomecánicas durante una ejecución del salto. Esto nos permite construir un conjunto n -dimensional de datos (n variables, m observaciones) a los que se les ha asignado una etiqueta de pertenencia a uno de los dos grupos (p. ej., etiquetas que identifican a los m saltadores como del grupo 1 (podio) o del grupo 2 (fuera del podio)).

Como en numerosas técnicas de inteligencia artificial, LDA se compone de dos fases: una primera de *entrenamiento* y una segunda de *predicción*. En la fase de entrenamiento se parte de los m datos del conjunto de entrenamiento y se genera un modelo lineal de las n variables que permita discriminar entre los dos grupos. En la fase de predicción, el modelo lineal se utiliza para predecir a qué grupo pertenece una nueva observación que no haya sido utilizada durante la fase de entrenamiento.

En el ejemplo anterior, la predicción nos permitiría estimar si un nuevo saltador será candidato al podio (pertenecer al grupo 1) o no (grupo 2) a partir de una medición de sus n variables biomecánicas.

La formulación matemática de la técnica LDA pasa por modelizar probabilísticamente la distribución de datos en cada una de las clases. Nosotros nos limitamos al caso de dos clases a las que nos referiremos como α_1, α_2 . Durante la fase de entrenamiento, los datos de entrenamiento de cada clase se utilizan para hacer una estimación estadística de los parámetros del modelo (media, varianza). El proceso de entrenamiento consiste por lo tanto en un proceso de *estimación estadística*, en el que se determinan parámetros de una cierta densidad de probabilidad.

La versión más simple del análisis de discriminantes asume que los datos de las clases α_1, α_2 están repartidos siguiendo una distribución de probabilidad normal. Eso se expresa mediante la densidad de probabilidad condicionada de que un dato \mathbf{x} pertenezca a la clase α_i

$$p(\mathbf{x}|\alpha_i) = \frac{1}{(2\pi)^{N/2}\sigma_i} \exp\left\{-\frac{1}{2\sigma_i^2}(\mathbf{x} - \mathbf{m}_i)^T \cdot (\mathbf{x} - \mathbf{m}_i)\right\}, \quad (41)$$

donde N es la dimensionalidad de los datos (número de variables), y μ_i y σ_i son la media y desviación típica de los datos pertenecientes a la clase α_i .

Una vez modelizados los datos, se introduce una *función discriminante* $g(\mathbf{x})$ que permite asignar el dato \mathbf{x} a una de las dos clases. La asignación de un nuevo dato \mathbf{x}_0 se realiza de la forma siguiente: si $g(\mathbf{x}_0) > 0$, el dato \mathbf{x}_0 se asigna a la clase α_1 , mientras que si $g(\mathbf{x}_0) < 0$ será asignado al grupo α_2 . En caso de que la función discriminante sea cero $g(\mathbf{x}_0) = 0$, el valor \mathbf{x}_0 se encontraría en la frontera de decisión y será igualmente asignable a cualquiera de las clases.

La función discriminante $g(\mathbf{x})$ permite definir la frontera de decisión de los datos para asignarlos a una clase. En el caso en que tengamos dos clases a discriminar, la función discriminante vendría dada por

$$g(\mathbf{x}) = g_1(\mathbf{x}) - g_2(\mathbf{x}), \quad (42)$$

donde $g_i(\mathbf{x})$ es la función de asignación a la clase i , que viene dada por la expresión

$$g_i(\mathbf{x}) = \ln p(\mathbf{x} | \alpha_i) + \ln P(\alpha_i), \quad i = 1, 2 \quad (43)$$

En la ecuación anterior, $p(\mathbf{x}|\alpha_i)$ es la densidad de probabilidad condicionada de que un dato \mathbf{x} pertenezca a la clase α_i descrita en la ecuación 41, y $P(\alpha_i)$ es la *densidad de probabilidad a priori* a la que hacíamos referencia anteriormente. $P(\alpha_i)$ es por tanto la probabilidad con la que a priori asignaríamos un dato a la clase α_i . Sustituyendo la expresión 41 en la ecuación 43 e introduciendo ésta a su vez en 42 es fácil ver que la función discriminante de la ecuación viene dada por

$$g(\mathbf{x}) = -\frac{1}{\sigma_1^2}(\mathbf{x} - \mathbf{m}_1)^T \cdot (\mathbf{x} - \mathbf{m}_1) + \frac{1}{\sigma_2^2}(\mathbf{x} - \mathbf{m}_2)^T \cdot (\mathbf{x} - \mathbf{m}_2) - N \ln \frac{\sigma_1}{\sigma_2} + \ln P(\alpha_1) - \ln P(\alpha_2). \quad (44)$$

En el caso en el que no haya información a priori sobre la pertenencia a una de las dos clases, tendremos $P(\alpha_1) = P(\alpha_2) = \frac{1}{2}$ y la función discriminante se simplifica adoptando la forma

$$g(\mathbf{x}) = -\frac{1}{\sigma_1^2}(\mathbf{x} - \mathbf{m}_1)^T \cdot (\mathbf{x} - \mathbf{m}_1) + \frac{1}{\sigma_2^2}(\mathbf{x} - \mathbf{m}_2)^T \cdot (\mathbf{x} - \mathbf{m}_2) - N \ln \frac{\sigma_1}{\sigma_2}. \quad (45)$$

La ecuación 45 puede interpretarse de forma geométrica: la asignación del dato \mathbf{x} a una u otra clase pasa por calcular a qué distancia se encuentra el dato \mathbf{x} del centroide de cada clase i , que viene dada por

$$d_i(\mathbf{x}) = \frac{1}{\sigma_i^2}(\mathbf{x} - \mathbf{m}_i)^T \cdot (\mathbf{x} - \mathbf{m}_i). \quad (46)$$

La distancia $d_i(\mathbf{x})$ expresada en la ecuación anterior es conocida como *distancia de Mahalanobis*, y constituye una métrica que pondera la distancia del dato al centroide de la clase i \mathbf{m}_i utilizando la varianza de los datos de la clase i , σ_i^2 . La función discriminante $g(\mathbf{x})$ está compuesta por las distancias de Mahalanobis del dato a cada una de las clases, e incluye una corrección debida a la posible disparidad entre las varianzas de los datos de cada clase $-N \ln \frac{\sigma_1}{\sigma_2}$. Así, para asignar un dato \mathbf{x}_0 a la clase α_1 , deberá cumplirse $g(\mathbf{x}_0) > 0$ o lo que es lo mismo, que $g_1(\mathbf{x}_0) > g_2(\mathbf{x}_0)$, y por lo tanto

$$\frac{1}{\sigma_1^2}(\mathbf{x}_0 - \mathbf{m}_1)^T \cdot (\mathbf{x}_0 - \mathbf{m}_1) + N \ln \sigma_1 < \frac{1}{\sigma_2^2}(\mathbf{x}_0 - \mathbf{m}_2)^T \cdot (\mathbf{x}_0 - \mathbf{m}_2) + N \ln \sigma_2. \quad (47)$$

Según la ecuación 47, cuando las varianzas de ambos grupos sean iguales $\sigma_1 = \sigma_2$, el discriminador lineal LDA asignará el dato \mathbf{x}_0 a la clase cuyo centroide esté más cerca del dato en distancia Mahalanobis. En este caso, la frontera de decisión será equidistante en distancia Mahalanobis a los centroides de los dos grupos. En caso de que exista una discrepancia entre las dos desviaciones típicas (p. ej., $\sigma_1 \neq \sigma_2$), el clasificador penalizará la asignación al grupo que presente mayor dispersión estadística mediante el término $N \ln \sigma_i$, $i = 1, 2$, lo que ocasionará un desplazamiento efectivo de la frontera de decisión del problema. Además de la interpretación geométrica de la asignación de pertenencia a clases, el método puede entenderse desde un punto de vista estadístico ya que el nuevo dato es asignado al clúster al que su pertenencia es más *estadísticamente verosímil*.

Función de verosimilitud

Otras muchas técnicas estadísticas de reconocimiento de patrones utilizan la *función de verosimilitud* para definir los criterios de agrupamiento.

Ejemplo: separación de datos bidimensionales

Las dos fases de la técnica LDA se describen en el ejemplo descrito en el código 3.17. Se parte de un conjunto de entrenamiento con datos gaussianos

($n = 2$ variables, $m = 100$ observaciones) que han sido etiquetados como pertenecientes al grupo 1 o al grupo 2. Los datos etiquetados como grupo 1 tienen la misma desviación típica que los del grupo 2 ($\sigma_1 = \sigma_2 = 5,5$), pero presentan una media distinta ($\mu_1 = 10, \mu_2 = -10$). El conjunto completo de datos, junto a las etiquetas de pertenencia a cada grupo, se le pasan a la rutina LDA para que construya un modelo lineal que permita discriminar entre ambas clases. En este ejemplo utilizaremos el código LDA del paquete Python de inteligencia artificial *Scikit-Learn* que podemos obtener de <http://scikit-learn.sourceforge.net>. Este paquete utiliza a su vez los paquetes estándar de computación y representación gráfica de Python *numpy*, *scipy*, *matplotlib*. La fase de entrenamiento corresponde a las líneas 26-28, en las que se genera un modelo lineal (fit) a partir de los datos XT y sus respectivas etiquetas de pertenencia a los dos grupos, almacenadas en la variable *labelT*. La opción *priors = None* de la rutina LDA especifica que no se considera una mayor probabilidad de asignación *a priori* del nuevo dato a una de las dos clases. En ocasiones, un conocimiento previo de los datos nos permite favorecer la asignación de un nuevo dato a uno de los grupos. Este sesgo anticipado se expresa mediante la *densidad de probabilidad a-priori*, que determina una cierta probabilidad de asignación a los grupos antes de conocer el nuevo dato a clasificar.

Código 3.17: Ejemplo de aplicación del análisis de discriminantes lineales (LDA)

```

1 import numpy as np
2 import pylab
3 from scikits.learn.lda import LDA
4
5 # Datos gaussianos:
6 mu1, sigma1 = 10, 5.5
7 X1 = mu1 + sigma1*np.random.randn(100,2)
8 mu2, sigma2 = -10, 5.5
9 X2 = mu2 + sigma2*np.random.randn(100,2)
10
11 # Representar gráficamente los datos de entrenamiento:
12 fig1 = pylab.figure()
13 pylab.scatter(X1[:,0],X1[:,1],marker='^',c='r')
14 pylab.scatter(X2[:,0],X2[:,1],marker='o',c='b',hold='on')
15 pylab.legend(('Grupo_1', 'Grupo_2'))
16
17 # Concatenar los dos conjuntos de puntos:
18 XT = np.concatenate((X1,X2))
19
20 # Etiquetar los datos como tipo 1 o tipo 2:
21 label1 = np.ones(X1.shape[0])
22 label2 = 2*np.ones(X2.shape[0])
23 labelT = np.concatenate((label1 ,label2))
24
25 # Fase de entrenamiento:
26 clf = LDA()
27 clf.fit(XT, labelT)
28 LDA(priors=None)
29
30 #Fase de predicción:
31 print clf.predict([[20, 0]]) #predicción para el dato [20,0]
32 print clf.predict([[5, -20]])#predicción para el dato [5,-20]
33
34 # Representación de la predicción de los datos [20,0] y [5,-20]:
35 pylab.scatter(20,0,s=100,marker='x',c='k')
36 pylab.annotate('LDA_grupo_1',xy=(20,-2),xycoords='data',
37               xytext=(-50,-50),
38               textcoords='offset_points',
39               arrowprops=dict(arrowstyle="->"))

```

```

40 pylab.scatter(-15,-5,s=100,marker='x',c='k')
41 pylab.annotate('LDA_grupo_2',xy=(-15,-5),xycoords='data',
42               xytext=(-50,50),
43               textcoords='offset_points',
44               arrowprops=dict(arrowstyle="->"))
45
46 # Predicción de datos en una retícula:
47 fig2 = pylab.figure()
48 for i in range(-20,21,1):
49     for k in range(-20,21,1):
50         p = clf.predict([[i, k]])
51         print i,k,p
52         if p == 1:
53             pylab.scatter(i,k,s=20,marker='o',c='r',hold='on')
54         else:
55             pylab.scatter(i,k,s=20,marker='o',c='b',hold='on')
56
57 pylab.axis([-20,20,-20,20])
58 pylab.text(5,5,'GRUPO_1',fontsize=25,fontweight='bold',color='k')
59 pylab.text(-10,-10,'GRUPO_2',fontsize=25,fontweight='bold',color='k')
60
61 pylab.show()

```

La fase de predicción permite utilizar el modelo lineal construido tras el entrenamiento para asignar un nuevo dato a uno de los dos grupos. En el código de ejemplo, los nuevos datos son (20,0) y (5, -20), y son asignados por el clasificador lineal LDA a los grupos 1 y 2, respectivamente (líneas 31 y 32). El resultado de dicha asignación se representa en la figura 22, en la que los datos de entrenamiento son los círculos azules (etiquetados como grupo 1) y los triángulos rojos (grupo 2). Los dos nuevos datos (20,0) y (5, -20) para los que se realiza la predicción se indican en la figura 22 con una cruz, y el método LDA los ha asignado correctamente a los grupos 1 y 2, respectivamente. La frontera de decisión que determina la asignación de los datos a cada grupo puede obtenerse calculando la predicción LDA de los puntos de una retícula de tamaño (-20,20) en ambas variables.

Figura 22. Representación gráfica de la fase de entrenamiento del ejemplo LDA 3.17

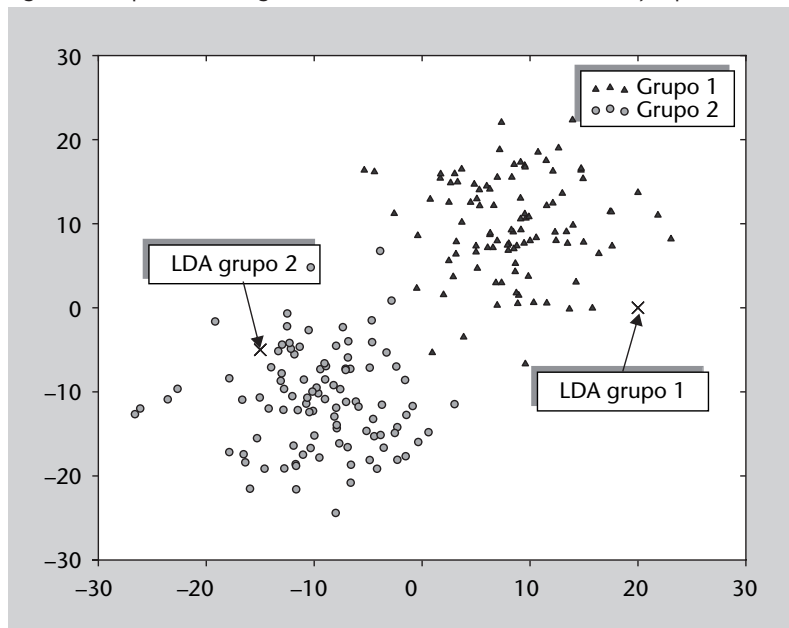


Figura 22

Los círculos azules han sido etiquetados como grupo 1 y los triángulos rojos como grupo 2 (datos de entrenamiento). Tras el entrenamiento, los nuevos datos [20,0] y [5, -20] son correctamente asignados por el clasificador lineal LDA a los grupos 1 y 2, respectivamente.

En la figura 23, los puntos de la retícula que han sido asignados al grupo 1 se representan de azul, mientras que los que han sido asignados al grupo 2 se representan en rojo. Como cabe esperar, la frontera es una función lineal que separa de forma equidistante los centroides de ambos grupos. Como veremos más adelante, la frontera es equidistante debido a que la desviación típica de los datos de cada uno de los grupos es la misma (i.e, $\sigma_1 = \sigma_2$).

Figura 23

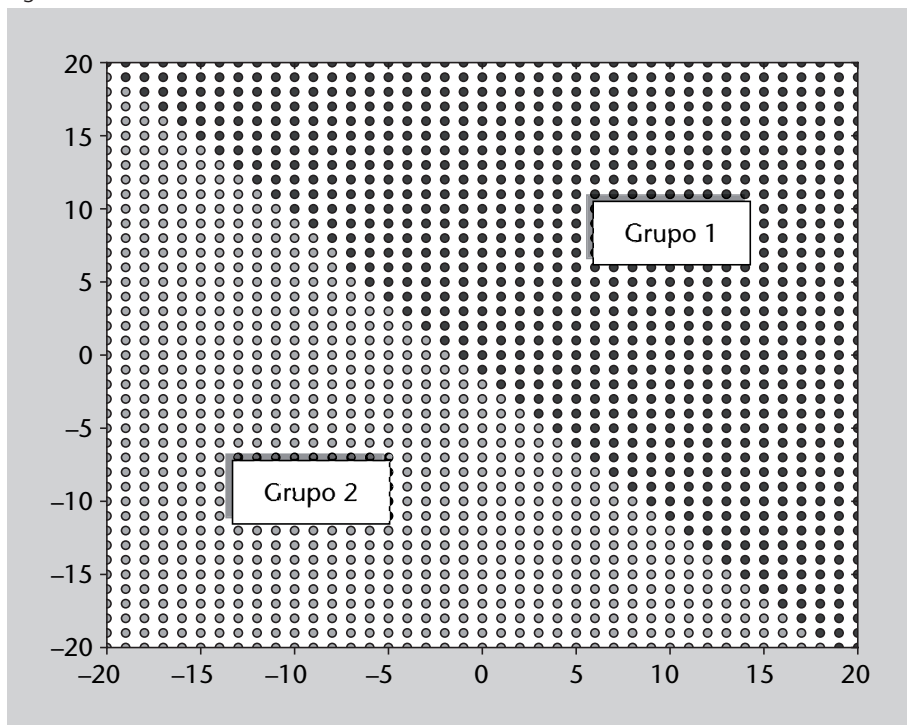
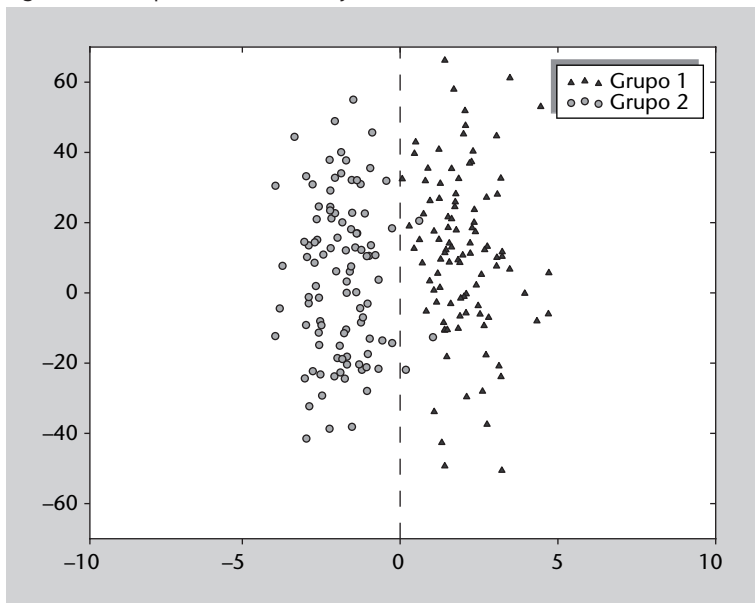


Figura 23

Predicción de la asignación a las dos clases mediante el modelo lineal generado por la técnica LDA a partir de los datos de entrenamiento en el ejemplo 3.17.

Conviene aclarar que, en muchos casos, técnicas como PCA no son útiles para realizar discriminación de datos en clases. Como ejemplo, consideremos los datos gaussianos multivariados de los dos grupos de la figura 24. Un análisis PCA de los datos de ambas clases proporcionaría una componente principal PCA a lo largo del eje de ordenadas (y), puesto que esta es la dirección, los datos presentan una mayor variabilidad. Si observamos la distribución de los datos de ambas clases, resulta evidente que una variable de este tipo no resulta útil para poder diferenciar los datos de ambos grupos, puesto que las diferencias entre el grupo 1 y el 2 es su ubicación en diferentes posiciones del eje de abscisas (x). En cambio, la línea vertical discontinua corresponde a la función discriminante obtenida al aplicar LDA a los datos, que sí resulta eficaz para separar de forma automática los datos en dos grupos que en gran medida coinciden con los grupos definidos inicialmente. En este caso, todos los datos del grupo 1 serían correctamente clasificados por la función discriminante, mientras que sólo tres puntos del grupo 2 serían asignados erróneamente al grupo 1.

Figura 24. Comparación entre LDA y PCA



3.3. Visualización de datos multidimensionales

3.3.1. Escalamiento multidimensional (MDS)

Ejemplo de aplicación

En ocasiones deseamos visualizar datos multidimensionales en un plano. Imaginemos por ejemplo un conjunto de datos que contenga múltiples variables sobre las preferencias musicales de un conjunto de personas. Las preferencias individuales de cada persona se representan mediante un conjunto de datos que define numerosos aspectos mediante n variables de carácter musical (grupos favoritos, el estilo de música, compositores, solistas, orquestas, grabaciones, etc.) y sociológico (edad, género, formación, salario, etc.). A las discográficas les resultará útil saber si existen relaciones entre los consumidores habituales de sus productos y otros colectivos desconocidos con los que puedan establecerse conexiones.

Resulta evidente que para establecer estas relaciones de forma manual no es posible trabajar en el espacio de dimensión n . Una de las posibilidades es proyectar los datos del espacio original a un espacio de dimensión 2 (plano), pero de forma que se *mantengan las distancias relativas entre cada uno de los individuos*. En otras palabras, personas que estaban lejos en el espacio original de dimensión n también estarán lejos en la proyección bidimensional de los datos, mientras que personas cercanas en el espacio original lo seguirán siendo en el espacio proyectado. Si las distancias entre individuos se conservan en el plano, será muy fácil ver qué personas están cercanas en el espacio original y por tanto quiénes son susceptibles de adquirir un determinado producto al que podrían ser afines. Podría suceder, por ejemplo, que los amantes incondi-

cionales del rock sinfónico y los Rolling Stones, mayores de 40 años y con un alto nivel de estudios, sean un buen objetivo de mercado para una grabación inédita de una sonata de F. Schubert del pianista ruso S. Richter. Este tipo de relaciones permiten establecer nuevos nichos de mercado o explorar nuevas posibilidades en un estudio de marketing.

Descripción de la técnica

Como hemos indicado, el escalamiento multidimensional* se utiliza principalmente como herramienta de visualización de datos. En general, un análisis exploratorio previo es enormemente útil para establecer relaciones iniciales entre las variables, y por tanto, para obtener información cualitativa sobre las características estructurales de los datos. Cuando el conjunto de datos multidimensionales tiene un elevado número de variables, el análisis exploratorio se complica debido al número de variables a considerar. Por supuesto, siempre es posible visualizar los datos mediante proyecciones bidimensionales o tridimensionales en subespacios de variables representativas, pero el procedimiento no deja de ser laborioso y dificulta una visión general de los datos. MDS es una herramienta que permite proyectar los datos en un espacio de dimensionalidad reducida de forma que la distribución de los datos en el espacio proyectado mantenga una similitud con la distribución de los datos en el espacio original.

*En inglés *multidimensional scaling* (MDS).

En resumen, MDS busca un espacio de dimensionalidad reducida en el que se mantengan las distancias relativas entre los datos originales. Puntos cercanos en el espacio original son escalados de forma que también sean cercanos en el espacio MDS, lo que nos va a permitir una mejor visualización de los datos sin desvirtuar la estructura original de los mismos.

La estructura general del algoritmo MDS es la siguiente (código 3.18): en primer lugar se toman los datos en el espacio N -dimensional original (N variables, M observaciones) y se calcula la *matriz de distancias* entre cada una de las M observaciones. Esto da lugar a una matriz simétrica $M \times M$, cuyo componente i,j corresponde a la distancia euclídea entre los puntos \mathbf{x}_i e \mathbf{x}_j . A continuación, las M observaciones son distribuidas aleatoriamente en un espacio bidimensional. Para cada par de puntos, se calcula una función error como la diferencia entre la distancia entre los puntos en el espacio original y la distancia en el espacio proyectado. Entonces se aplica un proceso iterativo, en el que la posición de cada punto en el espacio 2D es corregida para minimizar el error entre la distancia en el espacio original. El proceso continúa hasta que el error global no se ve afectado de forma sustancial por nuevos desplazamientos de los nodos.

El escalamiento multidimensional (MDS) permite visualizar datos multidimensionales en un espacio bidimensional.

Código 3.18: Código MDS en el libro *Programming Collective Intelligence*

```

1 from numpy import *
2 from pearson import *
3 from euclidean import *
4
5 def scaledown(data, distance=euclidean, rate=0.01):
6     n=len(data)
7     # The real distances between every pair of items
8     realdist=[[distance(data[i],data[j]) for j in range(n)]
9                for i in range(0,n)]
10    outersum=0.0
11    # Randomly initialize the starting points of the locations in 2D
12    loc=[[random.random(),random.random( )] for i in range(n)]
13    fakedist=[[0.0 for j in range(n)] for i in range(n)]
14    lasterror=None
15    for m in range(0,1000):
16        # Find projected distances
17        for i in range(n):
18            for j in range(n):
19                fakedist[i][j]=sqrt(sum([pow(loc[i][x]-loc[j][x],2)
20                                         for x in range(len(loc[i]))]))
21    # Move points
22    grad=[[0.0,0.0] for i in range(n)]
23    totalerror=0
24    for k in range(n):
25        for j in range(n):
26            if j==k: continue
27            # The error is percent difference between the distances
28            errorterm=(fakedist[j][k]-realdist[j][k])/realdist[j][k]
29            # Each point needs to be moved away from or towards the other
30            # point in proportion to how much error it has
31            grad[k][0]+=((loc[k][0]-loc[j][0])/fakedist[j][k])*errorterm
32            grad[k][1]+=((loc[k][1]-loc[j][1])/fakedist[j][k])*errorterm
33            # Keep track of the total error
34            totalerror+=abs(errorterm)
35    print totalerror
36    # If the answer got worse by moving the points, we are done
37    if lasterror and lasterror<totalerror: break
38    lasterror=totalerror
39    # Move each of the points by the learning rate times the gradient
40    for k in range(n):
41        loc[k][0]-=rate*grad[k][0]
42        loc[k][1]-=rate*grad[k][1]
43    return loc

```

La rutina descrita en el código 3.18 corresponde a la implementación de MDS descrita en el libro *Programming Collective Intelligence*, T. Segaran (O'Really, 2007).

Ejemplo: Proyección 2D de datos multidimensionales. Utilización de diferentes métricas

Un ejemplo de utilización de la rutina se describe en el código 3.19. En este ejemplo se generan 30 datos, 4 dimensionales, aleatoriamente distribuidos en tres grupos distintos de igual tamaño (líneas 1-15). A continuación se llama a la rutina MDS (línea 23), y previamente se define la métrica con la que deseamos trabajar para determinar la distancia entre puntos. En este ejemplo se ha escogido una distancia euclídea, ya que en el espacio original los datos no presentan ninguna correlación. La opción de distancia de Pearson resulta interesante en casos en los que los datos no se distribuyen por grupos separados en el espacio sino por la presencia de correlaciones internas entre grupos de datos.

Código 3.19: MDS con datos descorrelacionados distribuidos en un espacio

```

1 from numpy import *
2 from scaledown import *
3 import pylab as py
4
5 # Matriz de datos (N variables x M observaciones):
6 N = 4
7 M = 30
8
9 # Generar datos a partir de 3 clusters diferentes:
10
11 X1 = 10 + 2*random.randn(M/3,N) # cluster 1 (dispersión media)
12 X2 = -10 + 5*random.randn(M/3,N) # cluster 2 (dispersión alta)
13 X3 = 1*random.randn(M/3,N) # cluster 3 (dispersión baja)
14
15 A = concatenate((X1,X2,X3))
16
17 # Escoger métrica de espacio original (Euclídea o Pearson):
18 metrica = pearson
19 # metrica = euclidean
20
21 # Técnica MDS: Devuelve la posición de los datos
22 # originales en un espacio 2D:
23 mds = scaledown(A, metrica)
24
25 # Representación de los datos en el espacio 2D MDS:
26 fig1 = py.figure()
27 for i in range(0,M/3,1):
28     py.scatter(mds[i][0], mds[i][1], marker='o', c='r')
29 for i in range(M/3,2*M/3,1):
30     py.scatter(mds[i][0], mds[i][1], marker='v', c='g')
31 for i in range(2*M/3,M,1):
32     py.scatter(mds[i][0], mds[i][1], marker='x', c='b')
33 fig1.suptitle('Métrica_Pearson')
34 py.show()

```

El algoritmo para calcular la distancia de correlación de Pearson se describe en el código 3.20. Esta distancia toma un valor 1 cuando los datos están completamente correlacionados y un valor 0 cuando no existe ninguna correlación entre las observaciones.

Código 3.20: Distancia de correlación de Pearson

```

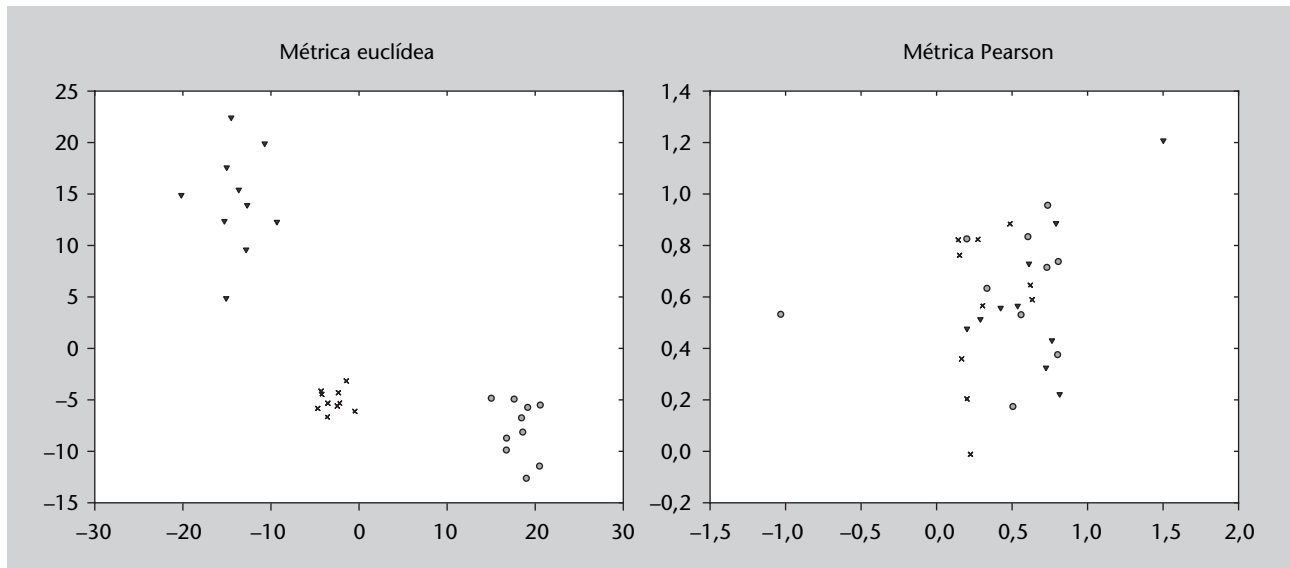
1 from math import sqrt
2
3 def pearson(v1,v2):
4     # Simple sums
5     sum1=sum(v1)
6     sum2=sum(v2)
7     # Sums of the squares
8     sum1Sq=sum([pow(v,2) for v in v1])
9     sum2Sq=sum([pow(v,2) for v in v2])
10    # Sum of the products
11    pSum=sum([v1[i]*v2[i] for i in range(len(v1))])
12    # Calculate r (Pearson score)
13    num=pSum-(sum1*sum2/len(v1))
14    den=sqrt((sum1Sq-pow(sum1,2)/len(v1))*(sum2Sq-pow(sum2,2)/len(v1)))
15    if den==0: return 0
16    return 1.0-num/den

```

La proyección bidimensional MDS de los datos descritos en el ejemplo 3.19 utilizando una métrica euclídea se representan en el panel izquierdo de la

figura 25. Como puede observarse, los tres grupos definidos en el espacio original de cuatro dimensiones se mantienen agrupados en la proyección MDS 2D. Si utilizamos la distancia de correlación de Pearson, los resultados no son satisfactorios puesto que la distribución de los datos en el espacio proyectado no refleja la auténtica distribución de los mismos en el espacio original (panel derecho de la figura 25).

Figura 25. MDS ejemplo 3.19



El ejemplo 3.21 describe una aplicación de la técnica MDS a un conjunto de puntos formado por tres subconjuntos en los que las observaciones tienen una correlación interna. Cada uno de los grupos presenta distintos valores de correlación. En este caso, la distancia que resulta idónea es la distancia de Pearson.

Código 3.21: Aplicación de MDS a un conjunto de datos que presentan correlaciones internas organizadas en tres grupos

```

1 from numpy import *
2 from scaledown import *
3 import pylab as py
4
5 # Matriz de datos (N variables x M observaciones):
6 N = 10
7 M = 30
8
9 # Generar datos a partir de 3 grupos con correlaciones diferentes:
10
11 Xinit1 = random.randn(1,N) # cluster 1 (correlación baja)
12 X1 = Xinit1
13 for i in range(1,M/3,1):
14     X1 = concatenate((X1,100*Xinit1 + random.rand(1,N)))
15
16 Xinit2 = random.randn(1,N) # cluster 2 (correlación media)
17 X2 = Xinit2
18 for i in range(1,M/3,1):
19     X2 = concatenate((X2,50*Xinit2 + random.rand(1,N)))
20
21 Xinit3 = random.randn(1,N) # cluster 3 (correlación alta)
22 X3 = Xinit3
23 for i in range(1,M/3,1):

```

```

24 X3 = concatenate((X3,10*Xinit3 + random.rand(1,N)))
25
26 #Representación correlaciones de cada grupo:
27 fig1 = py.figure()
28 py.scatter(X1[0],X1[1],marker='o',hold='on')
29 py.scatter(X2[0],X2[1],marker='x',hold='on')
30 py.scatter(X3[0],X3[1],marker='v',hold='on')
31
32 A = concatenate((X1,X2,X3))
33
34 # Escoger métrica de espacio original (Euclídea o Pearson):
35 metrica = euclidean
36 # metrica = pearson
37
38 # Técnica MDS: Devuelve la posición de los datos
39 # originales en un espacio 2D:
40 mds = scaledown(A, metrica)
41
42 # Representación de los datos en el espacio 2D MDS:
43 fig1 = py.figure()
44 for i in range(0,M/3,1):
45     py.scatter(mds[i][0],mds[i][1],marker='o',c='r')
46 for i in range(M/3,2*M/3,1):
47     py.scatter(mds[i][0],mds[i][1],marker='x',c='g')
48 for i in range(2*M/3,M,1):
49     py.scatter(mds[i][0],mds[i][1],marker='v',c='b')
50 fig1.suptitle('Métrica_Euclídea')
51 py.show()

```

Las figuras 26 y 27 reflejan la distribución de los puntos en el espacio MDS bidimensional cuando se utiliza una métrica euclídea (figura 26) y una métrica de Pearson (figura 27). En este caso resulta evidente que la métrica de Pearson permite agrupar los tres subconjuntos de observaciones en tres grupos bien diferenciados en el espacio de dimensión reducida.

Figura 26. Proyección MDS 2D de los datos del ejemplo 3.21 utilizando una métrica euclídea

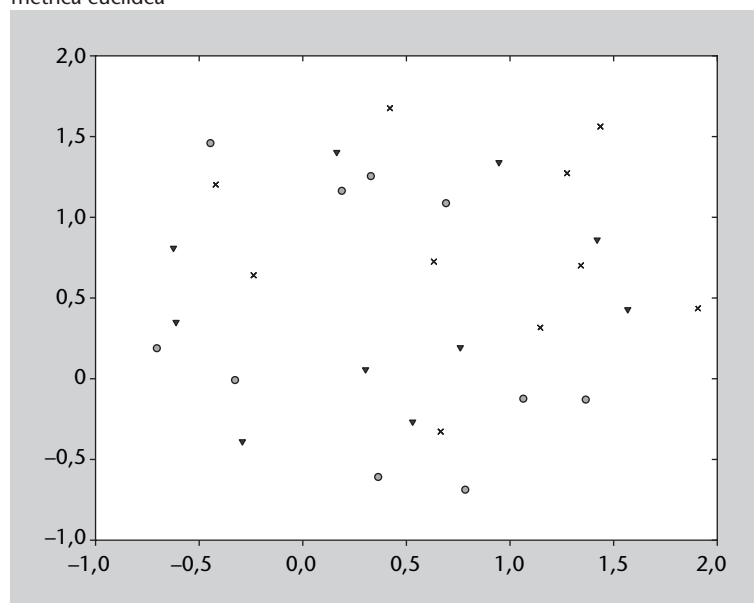
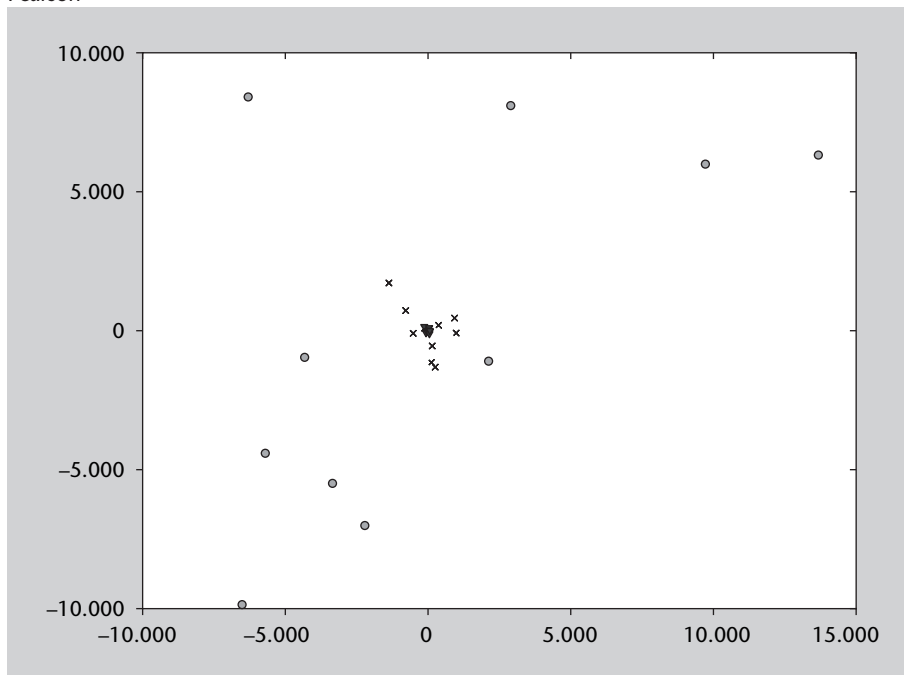


Figura 27. Proyección MDS 2D de los datos del ejemplo 3.21 utilizando una métrica de Pearson



4. Clasificación

4.1. Introducción

La clasificación es una de las tareas de reconocimiento de patrones en la que queremos etiquetar a un individuo a partir de ciertas propiedades que lo caracterizan; entendiendo como individuo una entidad de cualquier tipo. A continuación se muestran tres ejemplos de clasificación que utilizaremos a lo largo de este apartado:

- Clasificación de setas (Fuente: Problema “mushroom” del repositorio UCI (Frank y Asunción, 2010)). Supongamos que queremos realizar una aplicación para teléfonos móviles que ayude a los aficionados a la recolección de setas a discernir las setas venenosas de las comestibles a partir de sus propiedades: forma y color del sombrero y grosor y color del tronco.
- Clasificación de flores (Fuente: Problema “iris” del repositorio UCI (Frank y Asunción, 2010)). Supongamos que queremos realizar una aplicación para un almacén de flores, donde llegan a diario miles de productos. Disponemos de un sistema láser que nos suministra una serie de medidas sobre las flores y nos piden que el sistema las clasifique automáticamente para transportarlas mediante un robot a las diferentes estanterías del almacén. Las medidas que envía el sistema láser son: la longitud y anchura del sépalo y el pétalo de cada flor.
- Clasificación de documentos. Supongamos que queremos realizar una aplicación en la que dados los textos que van llegando a una agencia de noticias, los clasifique automáticamente para distribuirlos automáticamente a los clientes interesados. Los temas pueden ser: deporte, sociedad, política... o más específicamente: fútbol, motociclismo, atletismo... Los clientes pueden ser cadenas de televisión, radios, periódicos, revistas de toda índole...

Ved también

El problema de la clasificación de documentos, junto con sus alternativas, se describe con más profundidad en el subapartado 4.1.1.

La clasificación, desde el punto de vista del reconocimiento de patrones, involucra tres fases importantes: la definición de las clases, la representación de la información en forma de atributos y el aprendizaje mediante algoritmos.

La definición de clases viene dada en muchos problemas. En el ejemplo de la clasificación de setas tenemos que las clases son: venenosa y comestible. En

los otros dos ejemplos tenemos que tener en cuenta la granularidad de las clases. En la clasificación de flores podemos escoger entre familias de flores: rosa, clavel, margarita... o dentro de una familia, la flor específica: iris versicolor, iris setosa e iris virgínica. Este último será el que utilizaremos durante todo el apartado. En la clasificación de documentos también nos afecta la granularidad. En este caso, además, nos puede ocurrir que un documento tenga más de una etiqueta: una noticia sobre la relación sentimental entre un futbolista y una cantante puede ser de deporte y sociedad... La definición de clases tiene una relación directa con la aplicación final de la clasificación.

La representación de la información tiene que ver con como representamos las características de los individuos o ejemplos a etiquetar. Tiene una relación directa con el comportamiento general del clasificador que construyamos; es tan o más importante que el algoritmo de aprendizaje que utilicemos. Este tema se explica con más detalle en el apartado 3.

Este apartado está dedicado a la tercera de las fases, los algoritmos de aprendizaje para clasificación. Empezamos por desarrollar un poco más el problema de la clasificación de documentos (o categorización de textos) y por formalizar el problema. Una vez hecho esto damos una tipología de este tipo de algoritmos y entramos a describirlos. Para finalizar se dan los protocolos y medidas de evaluación.

4.1.1. Categorización de textos

La categorización de textos es uno de los problemas que se intenta resolver en el campo del procesamiento del lenguaje natural*. Este problema consiste en clasificar documentos con una o varias clases.

* En inglés, *Text Categorisation y Natural Language Processing*.

Un ejemplo práctico de este problema, al que se están dedicando muchos recursos y esfuerzos, es el tratamiento de textos por parte de las agencias de noticias. A estas empresas les suelen llegar varias noticias por segundo y tienen que etiquetar estas noticias en función de la temática: deportes, política, política internacional, sociedad... Una de las características de este problema es que es multietiqueta*, es decir, un ejemplo puede estar asociado a más de una etiqueta.

* En inglés, *Multilabel*.

Para representar los documentos en forma de atributos se suelen utilizar conjuntos de palabras*. Esta técnica viene del campo de la recuperación de la información**. Los conjuntos de palabras suelen aparecer en la bibliografía de dos formas: como conjuntos de palabras propiamente dichos (sobre todo cuando se trabaja a nivel de frase) o como conjuntos de palabras anotando el número de veces que aparecen en el documento (o su frecuencia de aparición).

* En inglés, *bag of words o vector space model (VSM)*.

** En inglés, *Information Retrieval*.

Cuando se trabaja con conjuntos de atributos, se suelen procesar los datos para eliminar las palabras sin contenido semántico (como preposiciones, ar-

títulos...), para eliminar signos de puntuación y para trabajar directamente con los lemas de las palabras (se elimina el género, número y tiempo verbal). Para poder realizar esto, se utilizan dos tipos de recursos: listas de palabras sin contenido semántico* y procesadores lingüísticos.

* En inglés, *lists of stop words*.

Enlace de interés

En la página web del profesor Lluís Padró (<http://www.lsi.upc.edu/~padro>) se pueden encontrar listas de palabras sin contenido semántico para el inglés, el castellano y el catalán. Las tenéis disponibles en el repositorio de la asignatura.

El segundo tipo de recurso son los procesadores lingüísticos. Suelen incluir tokenizadores, lematizadores, etiquetadores morfosintácticos, analizadores sintácticos superficiales... para diferentes lenguas. Para este problema, con el tokenizador y el lematizador ya cumpliremos con nuestras necesidades. Hay diversos procesadores lingüísticos libres disponibles en la red. Dos de ellos son el *FreeLing** y el *NLTK***.

* <http://nlp.lsi.upc.es/freeling>

** <http://www.nltk.org>

Un caso particular de la categorización de textos son los filtros anti spam. En este problema hay que pensar en cómo añadir como atributos los componentes de los correos, así como las direcciones de origen o el asunto.

El conjunto de datos más utilizado en categorización de textos es el Reuters-21578*. A los conjuntos de datos de textos se les denomina corpus. Este corpus ha sido generado a partir de datos de la agencia de noticias Reuters. Los documentos que contiene están sin procesar y tienen asociados una serie de clases o categorías. Este conjunto de datos también está disponible en el repositorio de la UCI (Frank y Asunción, 2010).

* <http://www.daviddlewis.com/resources/testcollections/reuters21578/>

El siguiente corpus* es un subconjunto del anterior. T. Joachims procesó una parte del corpus anterior calculando las frecuencias de aparición de las diferentes palabras en el documento. A esta forma de representar la información se le denomina VSM**. La tarea asociada a este corpus es distinguir si los documentos pertenecen o no a la clase: *adquisiciones corporativas*. El corpus está pensado para discernir si un documento es de esta clase o no. El conjunto de datos contiene 1.000 ejemplos positivos y 1.000 negativos en el conjunto de entrenamiento y 300 de cada en el conjunto de test. El número de atributos asciende a 9.947 y contienen información de la frecuencia relativa de las palabras en los documentos.

* <http://svmlight.joachims.org>

** Del inglés, *Vector Space Model*. También suele recibir el nombre de bolsa de palabras (del inglés, *bag of words*).

El tercer conjunto de datos es el *TechTC-100**, que ha sido recopilado por Gabrilovich y Markovitch. Este conjunto consta de una serie de páginas web que están asociadas a una ontología o jerarquía. Los datos están disponibles sin procesar y procesados. Este proceso consiste en generar el VSM para los documentos guardando el número de veces que aparece un término o palabra en cada documento. Las clases son las ramas de la jerarquía ODP** a la que pertenecen las páginas.

* <http://techtc.cs.technion.ac.il/techtc100/techtc100.html>

** <http://www.dmoz.org>

4.1.2. Aprendizaje automático para clasificación

En este subapartado vamos a ver la formalización del problema de clasificación tal y como se usa comúnmente en la bibliografía del campo del aprendizaje automático y el reconocimiento de patrones.

El objetivo del aprendizaje automático* para clasificación (o aprendizaje supervisado) consiste en inducir una aproximación (modelo o hipótesis) h de una función desconocida f definida desde un espacio de entrada X hacia un espacio discreto y desordenado $Y = 1, \dots, K$, dado un conjunto de entrenamiento S .

El conjunto de entrenamiento $S = \{(x_1, y_1), \dots, (x_n, y_n)\}$ contiene n ejemplos de entrenamiento, que corresponden a pares (x, y) donde $x \in X$ e $y = f(x)$. La componente x de cada ejemplo es un vector $x = (x_1, \dots, x_n)$ de atributos con valores continuos o discretos que describen la información relevante o propiedades del ejemplo. Los valores del espacio de salida Y asociados a cada ejemplo son las clases del problema. Así, cada ejemplo de entrenamiento queda totalmente caracterizado por un conjunto de pares atributo–valor y una etiqueta de clase.

Dado un conjunto de entrenamiento S , un algoritmo de entrenamiento induce un clasificador h que corresponde a una hipótesis acerca de la función f . Al hacer esto, el algoritmo puede escoger entre un conjunto de posibles funciones H llamado *espacio de hipótesis*. Los diferentes algoritmos de aprendizaje difieren en el espacio de hipótesis que tienen en cuenta (algunos tratarán funciones lineales, otros crearán hiperplanos que particionan dominios...); en el lenguaje de representación que utilizan (por ejemplo: árboles de decisión, conjuntos de probabilidades condicionales, redes neuronales...) y en el *sesgo* que usan al escoger la “mejor” hipótesis de entre las que son compatibles con el conjunto entrenamiento (por ejemplo: simplicidad, margen máximo...).

Partiendo de nuevos vectores x , utilizaremos h para predecir sus correspondientes valores de y para clasificar los nuevos ejemplos; esperando que coincidan con f en la mayoría de casos, o lo que es equivalente, que se produzcan el mínimo número de errores.

La medida del error de los nuevos ejemplos (o ejemplos no vistos) se llama *error de generalización**. Resulta obvio que el error de generalización no se puede minimizar sin conocer a priori la función f o la distribución $P(X, Y)$. Esto hace que necesitemos un principio de inducción. La manera más usual de proceder consiste en intentar minimizar el error sobre el conjunto de entrenamiento (o error empírico). Este principio se conoce como *minimización empírica del riesgo*** y da una buena estimación del error de generalización en presencia de muchos ejemplos de entrenamiento.

* En inglés, *machine learning*.

Regresión

Cuando el espacio de salida es continuo, el problema a tratar será de *regresión* en lugar de clasificación.

Clustering

Cuando no utilizamos las etiquetas de clase y_i en el conjunto de entrenamiento $S = \{x_1, \dots, x_n\}$ hablamos de *clustering* o aprendizaje no supervisado. Este tema se describe en el subapartado 2.3 de este módulo.

Lecturas complementarias

S. Kulkarni; G. Harman (2011). *An Elementary Introduction to Statistical Learning Theory*. USA: John Wiley and Sons, Inc.

* En inglés, *generalisation or true error*.

** En inglés, *empirical risk minimisation*.

No obstante, en dominios con pocos ejemplos de entrenamiento, forzar a que la minimización del error empírico tienda a cero puede llevar a que se sobreentrene* el conjunto de entrenamiento y a que se generalice de forma errónea. La complejidad de las funciones inducidas** y la presencia de ruido y ejemplos atípicos*** (mal clasificados o inconsistentes) en el conjunto de entrenamiento también puede aumentar el riesgo de sobreentrenamiento. Al definir un marco experimental es necesario establecer un compromiso entre el error experimental y la complejidad de los clasificadores inducidos para garantizar un error de generalización pequeño.

* En inglés, *overfitting*.

** La complejidad del modelo se mide con la llamada dimensión de Vapnik-Chervonenkis.

*** En inglés, *outliers*.

Toda la nomenclatura utilizada en los dos párrafos anteriores viene de la *teoría del aprendizaje estadístico**. La forma clásica de tratar este tema es a partir del sesgo** y la varianza. El sesgo de un estimador corresponde a la diferencia entre su esperanza matemática y el valor que está estimando. Todo lo que restringe el espacio de búsqueda está relacionado con el sesgo. La complejidad del modelo se mide a través de la varianza, que mide la dispersión de los valores en torno a la esperanza. La complejidad del modelo está directamente relacionada con la varianza e inversamente relacionada con el sesgo. El error de generalización está formado por la suma del sesgo y la varianza del modelo. En la construcción de modelos siempre tenemos que tratar con un compromiso entre el sesgo y la varianza.

* En inglés, *statistical learning theory*.

** En inglés, *bias*.

A modo de ejemplo, la tabla 7 muestra las características de dos ejemplos del conjunto de datos de las setas descrito anteriormente. Los atributos del primer ejemplo $x = (x_1, x_2, x_3, x_4)$ son *cap-shape*, *cap-color*, *gill-size* y *gill-color* con valores *convex*, *brown*, *narrow* y *black* respectivamente; el valor de y es *poisonous*. El conjunto Y contiene las etiquetas $\{poisonous, edible\}$.

Tabla 7. Ejemplo del tratamiento de setas

class	cap-shape	cap-color	gill-size	gill-color
poisonous	convex	brown	narrow	black
edible	convex	yellow	broad	black

Fuente: Problema "mushroom" del repositorio UCI (Frank y Asunción, 2010).

Toda la formulación que hemos visto hasta aquí corresponde a problemas monoetiqueta, en los que cada ejemplo sólo tiene asociada una única etiqueta. Para formalizar los problemas multietiqueta, como la categorización de textos, tenemos que cambiar las características de la función $f(x)$. En los problemas monoetiqueta esta función devuelve un único valor de $Y = \{1, \dots, k\}$, donde k es el número de clases; en los problemas multietiqueta, la función $f(x)$ devuelve un conjunto de valores de Y , que puede ser vacío.

4.1.3. Tipología de algoritmos para clasificación

Podemos clasificar los algoritmos en función del principio de inducción que usan para adquirir los modelos o reglas de clasificación a partir de los ejemplos. Estos métodos están basados en: probabilidades, distancias, reglas o kernels. Esta tipología no tiene la intención de ser exhaustiva o única. Por su-

puesto, la combinación de diferentes paradigmas es otra posibilidad que se está utilizando en los últimos tiempos.

Los cuatro siguientes subapartados están enfocados a explicar estos paradigmas y algunos de sus algoritmos más característicos. El último subapartado nos presenta los protocolos de evaluación que se utilizan en los procesos de clasificación.

4.2. Métodos basados en modelos probabilísticos

Los métodos estadísticos suelen estimar un conjunto de parámetros probabilísticos, que expresan la probabilidad condicionada de cada clase dadas las propiedades de un ejemplo (descrito en forma de atributos). A partir de aquí, estos parámetros pueden ser combinados para asignar las clases que maximizan sus probabilidades a nuevos ejemplos.

4.2.1. Naïve Bayes

Supongamos que queremos realizar una aplicación para teléfonos móviles que ayude a los aficionados a la recolección de setas a discernir las setas venenosas de las comestibles a partir de sus propiedades: forma y color del sombrero y grosor y color del tronco. La tabla 8 muestra ejemplos de este tipo de datos.

Tabla 8. Conjunto de entrenamiento

class	cap-shape	cap-color	gill-size	gill-color
poisonous	convex	brown	narrow	black
edible	convex	yellow	broad	black
edible	bell	white	broad	brown
poisonous	convex	white	narrow	brown
edible	convex	yellow	broad	brown
edible	bell	white	broad	brown
poisonous	convex	white	narrow	pink

Fuente: problema "mushroom" del repositorio UCI (Frank y Asunción, 2010).

El Naïve Bayes es el representante más simple de los algoritmos basados en probabilidades. Está basado en el teorema de Bayes.

El algoritmo de Naïve Bayes lo describen Duda y Hart en su forma más clásica en 1973.

El algoritmo de Naïve Bayes clasifica nuevos ejemplos $x = (x_1, \dots, x_m)$ asignándole la clase k que maximiza la probabilidad condicional de la clase dada la secuencia observada de atributos del ejemplo. Es decir,

$$\operatorname{argmax}_k P(k|x_1, \dots, x_m) = \operatorname{argmax}_k \frac{P(x_1, \dots, x_m|k)P(k)}{P(x_1, \dots, x_m)} \approx \operatorname{argmax}_k P(k) \prod_{i=1}^m P(x_i|k)$$

donde $P(k)$ y $P(x_i|k)$ se estiman a partir del conjunto de entrenamiento, utilizando las frecuencias relativas (estimación de la máxima verosimilitud*).

* En inglés, *maximum likelihood estimation*.

Ejemplo de aplicación

La tabla 8 muestra un ejemplo de conjunto de entrenamiento en el que tenemos que detectar si una seta es venenosa o no en función de sus propiedades.

Durante el proceso de entrenamiento empezaremos por calcular $P(k)$ para cada una de las clases $k \in Y$. Así, aplicando una estimación de la máxima verosimilitud obtenemos $P(poisonous) = 3/7 = 0,43$ y $P(edible) = 4/7 = 0,57$. El segundo paso consiste en calcular $P(x_i|k)$ para cada pareja atributo-valor y para cada clase, de la misma forma que en el caso anterior. La tabla 9 muestra los resultados de este proceso. El 1 de la celda correspondiente a la fila *cap-shape: convex* y columna *poisonous* sale de que los tres ejemplos etiquetados como *poisonous* tienen el valor *convex* para el atributo *cap-shape*.

Tabla 9. Valores de $P(x_i|k)$

atributo-valor	poisonous	edible
cap-shape: convex	1	0,5
cap-shape: bell	0	0,5
cap-color: brown	0,33	0
cap-color: yellow	0	0,5
cap-color: white	0,67	0,5
gill-size: narrow	1	0
gill-size: broad	0	1
gill-color: black	0,33	0,25
gill-color: brown	0,33	0,75
gill-color: pink	0,33	0

Con esto habremos terminado el proceso de entrenamiento. A partir de aquí, si nos llega un ejemplo nuevo como el que muestra la tabla 10 tendremos que aplicar la fórmula $\text{argmax}_k P(k) \prod_{i=1}^m P(x_i|k)$ para clasificarlo.


Tabla 10. Ejemplo de test

class	cap-shape	cap-color	gill-size	gill-color
poisonous	convex	brown	narrow	black

Fuente: Problema "mushroom" del repositorio UCI (Frank y Asunción, 2010).

Empezamos por calcular la parte del valor *poisonous*. Para ello tendremos que multiplicar entre sí las probabilidades: $P(poisonous)$, $P(\text{cap-shape-convex}|poisonous)$, $P(\text{cap-color-brown}|poisonous)$, $P(\text{gill-size-narrow}|poisonous)$ y $P(\text{gill-color-black}|poisonous)$; que equivale a un valor de 0,05. Realizando el mismo proceso para la clase *edible* obtenemos un valor de 0. Por tanto, la clase que maximiza la fórmula de las probabilidades es *poisonous*; con lo que el método está clasificando correctamente el ejemplo de test, dado este conjunto de entrenamiento.

Análisis del método

Uno de los problemas del Naïve Bayes, que ha sido mencionado frecuentemente en la literatura, es que el algoritmo asume la independencia de los diferentes atributos que representan a un ejemplo. 

A modo de ejemplo, es poco probable que el color del tallo de una seta no esté relacionado con el color de su sombrero.

Hasta ahora, hemos hablado exclusivamente de atributos nominales. En el caso de tener un problema representado con algún atributo continuo, como los numéricos, es necesario algún tipo de proceso. Hay diversas formas de hacerlo; una consiste en categorizarlos, dividiendo el continuo en intervalos. Otra opción consiste en asumir que los valores de cada clase siguen una distribución gaussiana y aplicar la fórmula:

$$P(x = v|c) = \frac{1}{\sqrt{2\pi\sigma_c^2}} \exp\left(-\frac{(v - \mu_c)^2}{2\sigma_c^2}\right)$$

donde x corresponde al atributo, v a su valor, c a la clase, μ_c al promedio de valores de la clase c y σ_c^2 a su desviación estándar.

El algoritmo presenta un problema numérico a tener en cuenta para ciertos conjuntos de datos. Cuando nos encontramos con un ejemplo de test que tiene alguna pareja atributo–valor que no ha aparecido en el conjunto de entrenamiento, no tendremos ningún valor para $P(x_i|k)$. La tabla 11 muestra un ejemplo para el conjunto de entrenamiento anterior; el valor “white” del atributo “gill-color” no aparece en el conjunto de entrenamiento. Para solucionar este problema se suele aplicar alguna técnica de *suavizado**.

* En inglés, *smoothing*.

Tabla 11. Ejemplo de test (suavizado)

class	cap-shape	cap-color	gill-size	gill-color
edible	bell	yellow	broad	white

Fuente: Problema “mushroom” del repositorio UCI (Frank y Asunción, 2010).

Un ejemplo de técnica de suavizado para este algoritmo consiste en sustituir la $P(x_i|k)$ que contiene el contador a cero por $P(k)/n$ donde n corresponde al número de ejemplos de entrenamiento.

Otra característica a destacar es que cuando el conjunto de entrenamiento no está balanceado, tiende a clasificar los ejemplos hacia la clase que tiene más ejemplos dentro del conjunto de entrenamiento.

Conjunto de entrenamiento balanceado

Un conjunto de entrenamiento está balanceado cuando tiene el mismo número de ejemplos de cada una de las clases que contiene.

Dos de las ventajas que presenta el método son su simplicidad y eficiencia computacional.

No obstante, y a pesar de sus inconvenientes, este método ha sido muy utilizado históricamente y se han obtenido buenos resultados para muchos conjuntos de datos. Esto ocurre cuando el conjunto de entrenamiento representa bien las distribuciones de probabilidad del problema.

Lectura complementaria

Esta técnica de suavizado aparece en: H. T. Ng. Exemplar-based Word Sense Disambiguation: Some Recent Improvements. En las actas de la conferencia *Empirical Methods in Natural Language Processing*, EMNLP, 1997.

Implementación en Python

En inteligencia artificial se ha usado clásicamente la programación funcional (representada por el lenguaje LISP) y la programación lógica (representada por PROLOG). En este apartado los programas se darán en Python, pero usando la parte de programación funcional que contiene. El programa 4.1 corresponde a la implementación, siguiendo estas directrices, del algoritmo Naïve Bayes.

Código 4.1: Naïve Bayes en Python

```

1 from functools import reduce
2
3 # declaración de funciones
4 def contar(l):
5     p = {}
6     for x in l:
7         p.setdefault(x, 0)
8         p[x] += 1
9     return p
10
11 def contar2(l, k):
12     p = {}
13     for i in range(len(l)):
14         p.setdefault(k[i], {})
15         p[k[i]].setdefault(l[i], 0)
16         p[k[i]][l[i]] += 1
17     return p
18
19 def Pxik(atr, N, k):
20     if atr in N[k]:
21         return N[k][atr] / Nk[k]
22     else:
23         return Nk[k] / n ** 2
24
25 def classify(t):
26     l = [(k, Nk[k] / n *
27           reduce(lambda x, y: x * y,
28                 map(Pxik, t, Nxik,
29                   [k for a in range(len(t)])))]
30     for k in Nk.keys()]
31     return max(l, key=lambda x: x[1])[0]
32
33 # carga del archivo
34 l = list(map(lambda l: (l.strip()).split(','),
35             open('mushroom.data.txt', 'r').readlines()))
36
37 # construcción del training: 2 de cada 3
38 train = list(map(lambda x: x[1],
39                 filter(lambda v: v[0] % 3 != 0, enumerate(l))))
40 n = len(train)
41
42 # construcción del test: 1 de cada 3
43 test = list(map(lambda x: x[1],
44                filter(lambda v: v[0] % 3 == 0, enumerate(l))))
45
46 # transpuesta del training
47 m = list(zip(*train))
48
49 # numerador de P(k)
50 Nk = contar(m[0])
51
52 # numerador de P(xi|k)
53 Nxik = [contar2(m[1], m[0]) for i in range(1, len(m))]
54
55 # clasificación
56 clases = list(map(lambda x: x.pop(0), test))
57 predicciones = list(map(classify, test))
58

```

```

59 # número de correctos
60 print('Prec.: ', len(list(filter(lambda x: x[0] == x[1],
61                               zip(*[predicciones, clases]))))
62       / len(test) * 100, '%')
    
```

La implementación contempla el uso de la técnica de suavizado comentada en el subapartado anterior. Este programa servirá para cualquier conjunto de datos en que todos sus atributos sean nominales (no importando el número de ellos), sin valores ausentes y que tenga la clase como primera columna. Están marcados, mediante comentarios, los diferentes procesos que realiza.

Si aplicamos el programa al archivo de datos del problema “mushroom” del UCI (Frank y Asunción, 2010) obtendremos una precisión del 55,7%. Este programa selecciona como conjunto de tests uno de cada tres ejemplos del archivo. El resto de ejemplos pertenecerá al conjunto de entrenamiento.

Ved también

Los archivos de datos y el programa están disponibles en el repositorio de la asignatura.

4.2.2. Máxima entropía

Supongamos que queremos realizar una aplicación para teléfonos móviles que ayude a los aficionados a la recolección de setas a discernir las setas venenosas de las comestibles a partir de sus propiedades: forma y color del sombrero. La tabla 12 muestra ejemplos de este tipo de datos.

Tabla 12. Conjunto de entrenamiento

class	cap-shape	cap-color
poisonous	convex	brown
edible	convex	yellow
edible	bell	white
poisonous	convex	white

Fuente: Problema “mushroom” del repositorio UCI (Frank y Asunción, 2010).

El algoritmo de máxima entropía proporciona una manera flexible de combinar evidencias estadísticas de diferentes fuentes. La estimación de probabilidades no asume conocimientos previos de los datos y se ha probado que es muy robusto.

El principio de máxima entropía se ha utilizado en muchos ámbitos, no es un algoritmo de clasificación. En este subapartado vamos a ver la adaptación del principio de la máxima entropía para crear un clasificador.

Dado un conjunto de entrenamiento $\{(x_1, y_1), \dots, (x_n, y_n)\}$ donde n es el número de ejemplos de entrenamiento, nuestro trabajo será encontrar las probabilidades condicionadas $p(y|f)$, donde f son los atributos que nos permitan definir el modelo de clasificación.

Algoritmo de máxima entropía

El algoritmo de máxima entropía apareció por primera vez en 1957 en dos artículos de E. T. Jaynes.

Lectura complementaria

En este subapartado vamos a utilizar la aproximación que se hace del principio de máxima entropía en: A. L. Berger, S. A. Della Pietra; V. J. Della Pietra (1996). “A Maximum Entropy Approach to Natural Language Processing”. *Computational Linguistics* (vol. 1, núm. 22).

El principio de máxima entropía consiste en definir un problema a partir de restricciones, y a partir de aquí definimos la distribución de probabilidades que sea más uniforme. Una medida matemática de la uniformidad de una distribución condicional $p(y|x)$ nos la da la entropía condicional:

$$H(p) = - \sum_{x,y} \tilde{p}(x)p(y|x) \log(p(y|x))$$

donde $\tilde{p}(x)$ corresponde a la distribución de probabilidad empírica.

Representaremos las parejas atributo–valor a partir de funciones $f_i(x,y)$ que asignarán un valor de 1 si el ejemplo x le corresponde como valor del atributo a la pareja representada por f_i , o 0 en caso contrario. A modo de ejemplo, si x_1 corresponde al primer ejemplo de la tabla 12 y f_1 al atributo correspondiente a la pareja (*cap-shape*, *convex*), la función quedaría como:

$$f_1(x,y) = \begin{cases} 1 & \text{si } x \text{ tiene } \textit{convex} \text{ como valor del atributo } \textit{cap-shape} \\ 0 & \text{si no lo tiene} \end{cases}$$

y el valor de $f_1(x_1,y_1)$ sería 1.

El valor esperado de f con respecto a la distribución empírica $\tilde{p}(x,y)$ es:

$$\tilde{p}(f) = \sum_{x,y} \tilde{p}(x,y)f(x,y)$$

y el valor esperado de f con respecto al modelo $p(y|x)$ que queremos:

$$p(f) = \sum_{x,y} \tilde{p}(x)p(y|x)f(x,y)$$

que nos genera la restricción:

$$p(f) = \tilde{p}(f)$$

o

$$\sum_{x,y} \tilde{p}(x,y)f(x,y) = \sum_{x,y} \tilde{p}(x)p(y|x)f(x,y)$$

Si tenemos el conjunto de restricciones $C = \{p \in P | p(f_i) = \tilde{p}(f_i), \forall 1 \leq i \leq n\}$, el principio de máxima entropía trata de buscar la distribución de probabilidad con entropía máxima:

$$p^* = \operatorname{argmax}_{p \in C} H(p)$$

Añadiremos las restricciones que siguen para garantizar que p son distribuciones de probabilidad condicional:

$$p(y|x) \geq 0, \forall x, y$$

$$\sum_y p(y|x) = 1, \forall x$$

Para atacar este problema, aplicaremos el método de los multiplicadores de Lagrange de la teoría de la optimización de restricciones:

$$\begin{aligned} \zeta(p, \Lambda, \gamma) = & - \sum_{x,y} \tilde{p}(x) p(y|x) \log(p(y|x)) + \\ & + \sum_i \lambda_i (\sum_{x,y} \tilde{p}(x, y) f_i(x, y) - \tilde{p}(x) p(y|x) f_i(x, y)) + \\ & + \gamma \sum_x p(y|x) - 1 \end{aligned}$$

De las soluciones a este problema tendremos:

$$p_\lambda(y|x) = \frac{1}{Z_\lambda(x)} \exp \left(\sum_i \lambda_i f_i(x, y) \right)$$

donde $Z_\lambda(x)$ corresponde a un factor de normalización:

$$Z_\lambda(x) = \sum_i \exp \left(\sum_i \lambda_i f_i(x, y) \right)$$

Uso en Python

El algoritmo de la máxima entropía está implementado en la librería *SciPy**. Esta librería lleva como ejemplos de uso de la máxima entropía los ejemplos que aparecen en el artículo de Berger y otros.

* <http://www.scipy.org>

4.3. Métodos basados en distancias

Los métodos de esta familia clasifican los nuevos ejemplos a partir de medidas de similitud o distancia. Esto se puede hacer comparando los nuevos ejemplos con conjuntos (uno por clase) de prototipos y asignando la clase del prototipo más cercano, o buscando en una base de ejemplos cuál es el más cercano.

Ved también

El método de los multiplicadores de Lagrange se estudia en el subapartado 5.2 de este módulo.

Referencia bibliográfica

A. L. Berger; S. A. Della Pietra; V. J. Della Pietra (1996). "A Maximum Entropy Approach to Natural Language Processing". *Computational Linguistics* (vol. 1, núm. 22).

4.3.1. kNN

Supongamos que queremos realizar una aplicación para un almacén de flores, donde llegan a diario miles de productos. Disponemos de un sistema láser que nos suministra una serie de medidas sobre las flores y nos piden que el sistema las clasifique automáticamente para transportarlas mediante un robot a las diferentes estanterías del almacén. Las medidas que envía el sistema láser son: la longitud y anchura del sépalo y el pétalo de cada flor. La tabla 13 muestra ejemplos de este tipo de datos.

Tabla 13. Conjunto de entrenamiento

class	sepal-length	sepal-width	petal-length	petal-width
setosa	5,1	3,5	1,4	0,2
setosa	4,9	3,0	1,4	0,2
versicolor	6,1	2,9	4,7	1,4
versicolor	5,6	2,9	3,6	1,3
virginica	7,6	3,0	6,6	2,1
virginica	4,9	2,5	4,5	1,7

Fuente: Problema "iris" del repositorio UCI (Frank y Asunción, 2010).

Uno de los algoritmos representativos de esta familia es el kNN (k vecinos más cercanos*). En este algoritmo la clasificación de nuevos ejemplos se realiza buscando el conjunto de los k ejemplos más cercanos de entre un conjunto de ejemplos etiquetados previamente guardados y seleccionando la clase más frecuente de entre sus etiquetas. La generalización se pospone hasta el momento de la clasificación de nuevos ejemplos**.

* En inglés, *k nearest neighbours*.

** Por esta razón es por la que en ocasiones es llamado aprendizaje perezoso (en inglés, *lazy learning*)

Una parte muy importante de este método es la definición de la medida de distancia (o similitud) apropiada para el problema a tratar. Esta debería tener en cuenta la importancia relativa de cada atributo y ser eficiente computacionalmente. El tipo de combinación para escoger el resultado de entre los k ejemplos más cercanos y el valor de la propia k también son cuestiones a decidir de entre varias alternativas.

Este algoritmo, en su forma más simple, guarda en memoria todos los ejemplos durante el proceso de entrenamiento y la clasificación de nuevos ejemplos se basa en las clases de los k ejemplos más cercanos*. Para obtener el conjunto de los k vecinos más cercanos, se calcula la distancia entre el ejemplo a clasificar $x = (x_1, \dots, x_m)$ y todos los ejemplos guardados $x_i = (x_{i1}, \dots, x_{im})$. Una de las distancias más utilizadas es la euclídea:

* Por esta razón se lo conoce también como basado en memoria, en ejemplos, en instancias o en casos.

$$de(x, x_i) = \sqrt{\sum_{j=1}^m (x_j - x_{ij})^2}$$

Ejemplo de aplicación

La tabla 13 muestra un conjunto de entrenamiento en el que tenemos que clasificar flores a partir de sus propiedades. En este ejemplo, aplicaremos el kNN para valores de k de 1 y 3, utilizando como medida de distancia la euclidiana.

El proceso de entrenamiento consiste en guardar los datos; no tenemos que hacer nada. A partir de aquí, si nos llega un ejemplo nuevo como el que muestra la tabla 14, tenemos que calcular las distancias entre el nuevo ejemplo y todos los del conjunto de entrenamiento. La tabla 15 muestra estas distancias.

Tabla 14. Ejemplo de test

class	sepal-length	sepal-width	petal-length	petal-width
setosa	4,9	3,1	1,5	0,1

Fuente: Problema "iris" del repositorio UCI (Frank y Asunción, 2010).

Tabla 15. Distancias

0,5	0,2	3,7	2,5	6,1	3,5
-----	-----	-----	-----	-----	-----

Para el 1NN escogemos la clase del ejemplo de entrenamiento más cercano que coincide con el segundo ejemplo (distancia 0,2) que tiene por clase *setosa*. Para el 3NN escogemos los tres ejemplos más cercanos: primero, segundo y cuarto; con distancias respectivas: 0,5, 0,2 y 2,5. Sus clases corresponden a *setosa*, *setosa* y *versicolor*. En este caso ñp asignaremos también a *setosa* por ser la clase más frecuente. En los dos casos el resultado es correcto.

Análisis del método

Un aspecto a tener en cuenta tiene que ver con la eficiencia computacional; este tiene que ver con que el algoritmo realiza todos los cálculos en el proceso de clasificación. Así, aun siendo un método rápido globalmente, tenemos que tener en cuenta que el proceso de clasificación no lo es. Esto puede llegar a ser crítico para aplicaciones de tiempo real que necesiten una respuesta rápida.

En el momento en que queramos aplicar este algoritmo a un problema, la primera decisión que tenemos que tomar es la medida de distancia y la segunda el valor de k . Se suele escoger un número impar o primo para minimizar la posibilidad de empates en las votaciones. La siguiente cuestión a decidir es el tratamiento de los empates cuando la k es superior a uno. Algunos heurísticos posibles son: no dar predicción en caso de empate, dar la clase más frecuente en el conjunto de entrenamiento de entre las clases seleccionadas para votar. . . Se suele escoger el heurístico en función del problema a tratar.

Las distancias más utilizadas son la euclídea y la de Hamming, en función del tipo de atributos que tengamos. La primera se utiliza mayoritariamente para atributos numéricos y la segunda para atributos nominales o binarios. Otros tipos de distancias pueden ser utilizadas dependiendo del problema a tratar.

Distancia de Hamming

La distancia de Hamming es $dh(x, x_i) = \sum_{j=1}^m \delta(x_j, x_{ij})$ donde $\delta(x_j, x_{ij})$ es la distancia entre dos valores que corresponde a 0 si $x_j = x_{ij}$ y a 1 si son diferentes.

Un ejemplo de esto es la distancia MVDM. Esta medida sustituye a la de Hamming en el tratamiento de atributos nominales y ha dado buenos resultados en problemas de procesamiento del lenguaje natural parecidos a la categorización de textos. Su inconveniente principal es su alto coste computacional.

Una variante del algoritmo muy utilizada es la utilización de ejemplos ponderados. Consiste en la introducción de una modificación en el esquema de votación de los k vecinos más cercanos, que hace la contribución de cada ejemplo proporcional a su importancia. Esta importancia se mide teniendo en cuenta la cercanía al ejemplo de test.

La ponderación de los atributos también se ha utilizado. Esto consiste en realizar un “ranking” de los atributos en función de su relevancia y hacer que su contribución afecte al cálculo de la distancia. Los pesos de los atributos se pueden calcular de diversas formas; una de ellas es la distancia RLM*. A modo de ejemplo, aplicar este concepto a la distancia de Hamming, una vez calculados los pesos, da como resultado:

$$dh(x, x_i) = \sum_{j=1}^m w_j \delta(x_j, x_{ij})$$

Otra forma de tratamiento de atributos es su selección, sobre todo en problemas en los que su número es importante. La selección de atributos consiste en reducir el número de atributos de los ejemplos. La idea es doble: selección de los más relevantes y eliminación de los que producen ruido. Una forma de realizar esto consiste en dividir el conjunto de entrenamiento en dos: uno de entrenamiento más pequeño y otro llamado de validación. Se aplica un algoritmo de optimización (como los descritos en el apartado 5) escogiendo como función objetivo el resultado de la clasificación sobre el conjunto de validación. El conjunto de entrenamiento de estos clasificadores es la parte del de entrenamiento que no pertenece al de validación. Una vez seleccionados los atributos, se entrena el clasificador final con todo el conjunto de entrenamiento y se prueba con el de test real. Otra forma de realizar esto consiste en aplicar alguno de los métodos descritos en el apartado 3, como el *PCA*.

Una de las grandes ventajas de este método es la conservación de excepciones en el proceso de generalización, y uno de los grandes inconvenientes conocidos es la sensibilidad a procesos de selección de atributos.

Implementación en Python

El programa 4.2 corresponde a la implementación en Python del kNN básico. En esta versión se ha escogido como distancia la euclídea y 3 como valor de la k . Este programa funcionará para todos los problemas con datos numéricos sin valores ausentes que tengan la clase como última columna.

MVDM

MVDM son las siglas del inglés, *modified value difference metric*. Propuesta por: S. Cost; S. Salzberg (1993). “A Wighted Nearest Neighbor Algorithm for Learning with Symbolic Features”. *Machine Learning* (núm. 10)

* Distancia por Ramón López de Mántaras en: R. López de Mántaras (1991). “A distance-based attribute selection measure for decision tree induction”. *Machine Learning* (núm. 6, págs. 81-92).

Lectura recomendada

En la referencia siguiente se utilizan algoritmos genéticos para seleccionar atributos y el ajuste de parámetros de un algoritmo basado en kNN: B. Decadt; V. Hoste; W. Daelemans; A. van den Bosch (2004). GAMBL, Genetic Algorithm Optimization of Memory-Based WSD. En las actas del *International Workshop on Evaluating WSD Systems, Senseval* (núm. 3)

Código 4.2: kNN en Python

```

1 # parámetros
2
3 k = 3
4
5 # declaración de funciones
6
7 def deulidea(x, y):
8     return sum(map(lambda a, b: (float(a) - float(b)) ** 2,
9                    x, y)) ** (1 / 2)
10
11 def contar(l):
12     p = {}
13     for x in l:
14         p.setdefault(x, 0)
15         p[x] += 1
16     return p
17
18 def classify(t):
19     ds = list(map(deulidea, train, [t for x in range(len(train))]))
20     kcl = contar([sorted([(ds[i], clasesTrain[i])
21                          for i in range(len(train))],
22                          key=lambda x: x[0])[i][1]
23                    for i in range(k)])
24     return max([(x, kcl[x]) for x in kcl.keys()],
25               key=lambda x: x[1])[0]
26
27 # carga del archivo
28 l = list(map(lambda l: (l.strip()).split(','),
29             open('iris.data.txt', 'r').readlines()))
30
31 # construcción del training: 2 de cada 3
32 train = list(map(lambda x: x[1],
33                 filter(lambda v: v[0] % 3 != 0, enumerate(1))))
34 nc = len(train[0])
35 clasesTrain = list(map(lambda x: x.pop(nc - 1), train))
36
37 # construcción del test: 1 de cada 3
38 test = list(map(lambda x: x[1],
39                filter(lambda v: v[0] % 3 == 0, enumerate(1))))
40 clasesTest = list(map(lambda x: x.pop(nc - 1), test))
41
42 # clasificación
43 predicciones = list(map(classify, test))
44
45 # número de correctos
46 print('Prec.: ', len(list(filter(lambda x: x[0] == x[1],
47                                zip(*[predicciones, clasesTest]))))
48       / len(test) * 100, '%')

```

Si aplicamos el programa al archivo de datos del problema “iris” del UCI (Frank y Asunción, 2010) obtendremos una precisión del 98%. Este programa selecciona como conjunto de tests uno de cada tres ejemplos del archivo. El resto de ejemplos pertenecerá al conjunto de entrenamiento.

4.3.2. Clasificador lineal basado en distancias

Una forma muy parecida a la del algoritmo anterior de abordar el problema de la clasificación de flores se basa en el uso de centros de masa o centroides. El modelo de clasificación de este algoritmo consta de un centroide, que representa cada una de las clases que aparecen en el conjunto de entrenamiento*.

Repositorio

Los archivos de datos y el programa están disponibles en el repositorio de la asignatura.

* Este método también recibe el nombre de método basado en centroides o en centros de masas.

El valor de cada atributo del centroide se calcula como el promedio del valor del mismo atributo de todos los ejemplos del conjunto de entrenamiento que pertenecen a su clase. La fase de clasificación consiste en aplicar el 1NN con distancia euclídea, seleccionando como conjunto de entrenamiento los centroides calculados previamente.

Ejemplo de aplicación

Aplicaremos como ejemplo el algoritmo a los ejemplos de la tabla 13. El proceso de entrenamiento consiste en calcular los centroides. La tabla 16 muestra el resultado. A modo de ejemplo, el 5 del atributo *sepal-length* del centroide *setosa* sale del promedio de 4.9 y 5.1 de la tabla 13.

Tabla 16. Centroides

class	sepal-length	sepal-width	petal-length	petal-width
setosa	5	3,25	1,4	0,2
versicolor	5,85	2,9	4,15	1,35
virginica	6,25	2,75	5,55	1,9

A partir de aquí, si nos llega un ejemplo nuevo como el que muestra la tabla 14, tenemos que calcular las distancias entre el nuevo ejemplo y todos los centroides para el 1NN. La tabla 17 muestra estas distancias. Al escoger el más cercano (distancia 0,2) que tiene por clase *setosa*, vemos que el ejemplo queda bien clasificado.

Tabla 17. Distancias

0,2	3,1	4,6
-----	-----	-----

Análisis del método

Este algoritmo puede aplicarse utilizando un producto escalar y se ha utilizado mucho desde este otro punto de vista. En el subapartado 4.5.1 se plantea este nuevo punto de vista y se analiza en más profundidad.

Implementación en Python

El programa 4.3 corresponde a la implementación en Python del clasificador lineal. Este programa funcionará para todos los problemas con datos numéricos sin valores ausentes que tengan la clase como última columna.

Código 4.3: Clasificador lineal basado en distancias en Python

```

1 from functools import reduce
2
3 # declaración de funciones
4
5 def deulidea(x, y):
6     return sum(map(lambda a, b: (float(a) - float(b)) ** 2,
7                   x[1], y)) ** (1 / 2)

```

```

8
9 def contar(l):
10     p = {}
11     for x in l:
12         p.setdefault(x, 0)
13         p[x] += 1
14     return p
15
16 def classify(t):
17     ds = list(map(deuclidea, centroides,
18                 [t for x in range(len(centroides))]))
19     return min([(ds[i], centroides[i][0])
20                for i in range(len(centroides))],
21               key=lambda x: x[0])[1]
22
23 def calcularCentroides(clase):
24     filt = list(filter(lambda x: x[0] == x[1],
25                      [(clasesTrain[i], clase, train[i])
26                       for i in range(len(train))]))
27     transp = list(zip(*[filt[i][2] for i in range(len(filt))]))
28     return (clase,
29            list(map(lambda l: reduce(lambda a, b:
30                                   float(a) + float(b),
31                                   l) / clases[clase], transp)))
32
33 # carga del archivo
34 l = list(map(lambda l: (l.strip()).split(','),
35            open('iris.data.txt', 'r').readlines()))
36
37 # construcción del training: 2 de cada 3
38 train = list(map(lambda x: x[1],
39                filter(lambda v: v[0] %3 != 0, enumerate(1))))
40 nc = len(train[0])
41 clasesTrain = list(map(lambda x: x.pop(nc - 1), train))
42
43 # construcción del test: 1 de cada 3
44 test = list(map(lambda x: x[1],
45               filter(lambda v: v[0] %3 == 0, enumerate(1))))
46 clasesTest = list(map(lambda x: x.pop(nc - 1), test))
47
48 # entrenamiento
49 clases = contar(clasesTrain)
50 centroides = [calcularCentroides(c) for c in clases.keys()]
51
52 # clasificación
53 predicciones = list(map(classify, test))
54
55 # número de correctos
56 print('Prec.:', len(list(filter(lambda x: x[0] == x[1],
57                               zip(*[predicciones, clasesTest]))))
58       / len(test) * 100, '%')

```

Si aplicamos el programa al archivo de datos del problema “iris” del UCI (Frank y Asunción, 2010), obtendremos una precisión del 96%. Este programa selecciona como conjunto de tests uno de cada tres ejemplos del archivo. El resto de ejemplos pertenecerá al conjunto de entrenamiento.

Repositorio

Los archivos de datos y el programa están disponibles en el repositorio de la asignatura.

4.3.3. Clustering dentro de clases

Este algoritmo está a medio camino entre los dos anteriores. En este caso, abordaremos el problema de la clasificación de las flores teniendo las ventajas y/o inconvenientes de los dos.


```

49         for ej in conj]
50     centr2 = [nextCentr(list(filter(
51         lambda x: x[0] == x[1],
52         zip(*[list(repeat(c, len(conj))),
53             cercanos, conj])))
54         for c in range(k)]
55     if cercanos == anteriores: break
56     anteriores = cercanos
57
58     return [(clase, centr2[i]) for i in range(k)]
59
60 # carga del archivo
61 l = list(map(lambda l: (l.strip()).split(','),
62             open('iris.data.txt', 'r').readlines()))
63
64 # construcción del training: 2 de cada 3
65 train = list(map(lambda x: x[1],
66                 filter(lambda v: v[0] %3 != 0, enumerate(1))))
67 nc = len(train[0])
68 clasesTrain = list(map(lambda x: x.pop(nc - 1), train))
69
70 # construcción del test: 1 de cada 3
71 test = list(map(lambda x: x[1],
72                filter(lambda v: v[0] %3 == 0, enumerate(1))))
73 clasesTest = list(map(lambda x: x.pop(nc - 1), test))
74
75 # entrenamiento
76 clases = contar(clasesTrain)
77 centroides = reduce(lambda x, y: x + y,
78                    [kmeans(c, k, maxit) for c in clases.keys()])
79
80 # clasificación
81 predicciones = list(map(classify, test))
82
83 # número de correctos
84 print('Prec.: ',
85       len(list(filter(lambda x: x[0] == x[1],
86                       zip(*[predicciones, clasesTest]))))
87       / len(test) * 100, '%')

```

Si aplicamos el programa al archivo de datos del problema “iris” del UCI (Frank y Asunción, 2010), con una k de 3 y un máximo de iteraciones de 10, obtendremos una precisión del 98%. Este programa selecciona como conjunto de tests uno de cada tres ejemplos del archivo. El resto de ejemplos pertenecerá al conjunto de entrenamiento.

4.4. Métodos basados en reglas

Estos métodos adquieren reglas de selección asociadas a cada una de las clases. Dado un ejemplo de test, el sistema selecciona la clase que verifica algunas de las reglas que determinan una de las clases.

4.4.1. Árboles de decisión

Supongamos que queremos realizar una aplicación para teléfonos móviles que ayude a los aficionados a la recolección de setas a discernir las setas venenosas de las comestibles a partir de sus propiedades: forma y color del sombrero y color del tronco.

Repositorio

Los archivos de datos y el programa están disponibles en el repositorio de la asignatura.

Un árbol de decisión es una forma de representar reglas de clasificación inherentes a los datos, con una estructura en árbol n -ario que particiona los datos de manera recursiva. Cada rama de un árbol de decisión representa una regla que decide entre una conjunción de valores de un atributo básico (nodos internos) o realiza una predicción de la clase (nodos terminales).

El algoritmo de los árboles de decisión básico está pensado para trabajar con atributos nominales. El conjunto de entrenamiento queda definido por $S = \{(x_1, y_1), \dots, (x_n, y_n)\}$, donde cada componente x corresponde a $x_i = (x_{i_1}, \dots, x_{i_m})$ donde m corresponde al número de atributos de los ejemplos de entrenamiento; y el conjunto de atributos por $A = \{a_1, \dots, a_m\}$ donde $\text{dom}(a_j)$ corresponde al conjunto de todos los posibles valores del atributo a_j , y para cualquier valor de un ejemplo de entrenamiento $x_{ij} \in \text{dom}(a_j)$.

El proceso de construcción del árbol es un proceso iterativo, en el que, en cada iteración, se selecciona el atributo que mejor particiona el conjunto de entrenamiento. Para realizar este proceso, tenemos que mirar la bondad de las particiones que genera cada uno de los atributos y, en un segundo paso, seleccionar el mejor. La partición del atributo a_j genera $|\text{dom}(a_j)|$ conjuntos, que corresponde al número de elementos del conjunto. Existen diversas medidas para mirar la bondad de la partición. Una básica consiste en asignar a cada conjunto de la partición la clase mayoritaria del mismo y contar cuántos quedan bien clasificados y dividirlo por el número de ejemplos. Una vez calculadas las bondades de todos los atributos, escogemos el mejor.

Cada conjunto de la mejor partición pasará a ser un nuevo nodo del árbol. A este nodo se llegará a través de una regla del tipo *atributo = valor*. Si todos los ejemplos del conjunto han quedado bien clasificados, lo convertimos en nodo terminal con la clase de los ejemplos. En caso contrario, lo convertimos en nodo interno y aplicamos una nueva iteración al conjunto ("reducido") eliminando el atributo que ha generado la partición. En caso de no quedar atributos, lo convertiríamos en nodo terminal asignando la clase mayoritaria.

Para realizar el test, exploramos el árbol en función de los valores de los atributos del ejemplo de test y las reglas del árbol hasta llegar al nodo terminal, y damos como predicción la clase del nodo terminal al que lleguemos.

Ejemplo de aplicación

La tabla 18 es una simplificación de la tabla 8. Para construir un árbol de decisión a partir de este conjunto tenemos que calcular la bondad de las particiones de los tres atributos: *cap-shape*, *cap-color* y *gill-color*. El atributo *cap-shape* nos genera una partición con dos conjuntos: uno para el valor *convex* y otro para *bell*. La clase mayoritaria para el conjunto de *convex* es *poisonous* y la de

bell es *edible*; su bondad es $bondad(cap - shape) = (3 + 2)/7 = 0,71$. Si realizamos el mismo proceso para el resto de atributos obtenemos: $bondad(cap - color) = (1 + 2 + 2)/7 = 0,71$ y $bondad(gill - color) = (1 + 3 + 1)/7 = 0,71$.

Tabla 18. Conjunto de entrenamiento

class	cap-shape	cap-color	gill-color
poisonous	convex	brown	black
edible	convex	yellow	black
edible	bell	white	brown
poisonous	convex	white	brown
edible	convex	yellow	brown
edible	bell	white	brown
poisonous	convex	white	pink

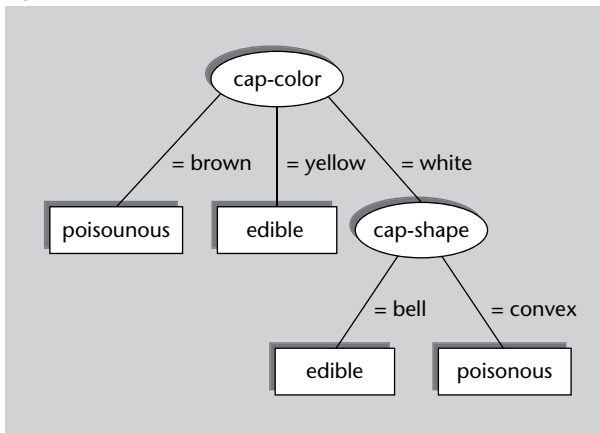
Fuente: Problema "mushroom" del repositorio UCI (Frank y Asunción, 2010).

El siguiente paso consiste en seleccionar el mejor atributo. Hemos obtenido un empate entre los tres atributos, podemos escoger cualquiera de ellos; escogemos *cap-color*. Los nodos generados por el conjunto de *brown* y *yellow* son terminales y les asignamos las clases *poisonous* y *edible*, respectivamente. Esto es debido a que obtienen los dos conjuntos una bondad de 1. El nodo de *white* no queda con los datos que muestra la tabla 19. Este conjunto lo obtenemos de eliminar el atributo *cap-color* de los ejemplos que tienen el valor *white* para el atributo *cap-color*. Volvemos a iterar; la bondad de las nuevas particiones es: $bondad(cap - shape) = (2 + 2)/4 = 1$ y $bondad(gill - color) = (2 + 1)/4 = 0,75$. El mejor atributo es *cap-shape*, que genera dos nodos terminales con las clases *edible* para *bell* y *poisonous* para *convex*. La figura 28 muestra el árbol construido en este proceso.

Tabla 19. Conjunto de la segunda iteración

class	cap-shape	gill-color
edible	bell	brown
poisonous	convex	brown
edible	bell	brown
poisonous	convex	pink

Figura 28. Árbol de decisión



Para etiquetar un ejemplo de test como el que muestra la tabla 20 tenemos que recorrer el árbol, partiendo de la raíz, escogiendo las ramas correspondientes

a los valores de los atributos de los ejemplos de test. Para este caso, miramos el valor del atributo *cap-color* y bajamos por la rama que corresponde al valor *brown*. Llegamos a un nodo terminal con clase *poisonous* con lo que se la asignaremos al ejemplo de test como predicción. Esta predicción es correcta.

Tabla 20. Ejemplo de test

class	cap-shape	cap-color	gill-color
poisonous	convex	brown	black

Fuente: Problema "mushroom" del repositorio UCI (Frank y Asunción, 2010).

Análisis del método

Este algoritmo tiene la gran ventaja de la facilidad de interpretación del modelo de aprendizaje. Un árbol de decisión nos da la información clara de la toma de decisiones y de la importancia de los diferentes atributos involucrados. Por esta razón se ha utilizado mucho en diversos campos, como en el financiero.

Dos de los grandes inconvenientes que plantea son la elevada fragmentación de los datos en presencia de atributos con muchos valores y el elevado coste computacional que esto implica. Esto hace que no sea un método muy adecuado para problemas con grandes espacios de atributos; como pueden ser aquellos que contienen información léxica, como la categorización de textos o los filtros anti spam.

Otro inconveniente a tener en cuenta es que los nodos terminales correspondientes a reglas que dan cobertura a pocos ejemplos de entrenamiento no producen estimaciones fiables de las clases. Tiende a sobreentrenar el conjunto de entrenamiento*. Para suavizar este efecto se puede utilizar alguna técnica de poda**. Más adelante se explica una de ellas.

*Este efecto se conoce en inglés como *overfitting*.

** En inglés, *prunning*.

Tratamiento de atributos numéricos

Supongamos que ahora queremos realizar una aplicación para un almacén de flores, donde llegan a diario miles de productos. Disponemos de un sistema láser que nos suministra la longitud del sépalo de las flores y nos piden que el sistema las clasifique automáticamente para transportarlas mediante un robot a las diferentes estanterías del almacén.

El algoritmo de los árboles de decisión fue diseñado para tratar con atributos nominales, pero existen alternativas para el tratamiento de atributos numéricos. El más común se basa en la utilización de puntos de corte. Estos son un punto que divide el conjunto de valores de un atributo en dos (los menores y los mayores del punto de corte).

Para calcular el mejor punto de corte de un atributo numérico, se ordenan los valores y se eliminan los elementos repetidos. Se calculan los posibles puntos de corte como el promedio de cada dos valores consecutivos. Como último paso se calcula el mejor de ellos como aquél con mejor bondad.

Tenemos que tener en cuenta que el tratamiento de atributos numéricos genera árboles (y decisiones) binarios. Esto implica que en los siguientes niveles del árbol tendremos que volver a procesar el atributo numérico que acabamos de seleccionar, a diferencia de los atributos nominales.

Ejemplo del cálculo del mejor punto de corte de un atributo numérico

A continuación se expone el cálculo del mejor punto de corte para el atributo *sepal-length* del conjunto de datos que muestran la tabla 13. La tabla 21 muestra estos cálculos: las columnas 1 y 2 muestran la clase y el atributo ordenados por el atributo, la 3 muestra los valores sin los repetidos, la 4 los puntos de corte como el promedio de cada dos valores consecutivos y la última columna muestra las bondades de los puntos de corte. El mejor punto de corte es 5,35 con una bondad de 66,7 %.

Tabla 21. Cálculos para el mejor punto de corte

clase	sepal-length	sin los repetidos	puntos de corte	bondad
setosa	4,9			
virgínica	4,9	4,9	5	$(1 + 2)/6 = 0,5$
setosa	5,1	5,1	5,35	$(2 + 2)/6 = 0,67$
versicolor	5,6	5,6	5,85	$(2 + 1)/6 = 0,5$
versicolor	6,1	6,1	6,85	$(2 + 1)/6 = 0,5$
virgínica	7,6	7,6		

Algoritmo ID3

El algoritmo ID3 es una modificación de los árboles de decisión que utiliza la ganancia de información* como medida de bondad para analizar los atributos. La ganancia de información es un concepto que está basado en la entropía. El uso de la entropía tiende a penalizar más los conjuntos mezclados.

* En inglés, *Information Gain*.

La entropía es una medida de la teoría de la información que cuantifica el desorden. Así, dado un conjunto S , su fórmula viene dada por:

$$H(S) = - \sum_{y \in Y} p(y) \log_2(p(y))$$

donde $p(y)$ es la proporción de ejemplos de S que pertenece a la clase y . La entropía será mínima (0) cuando en S hay una sola clase y máxima (1) cuando el conjunto es totalmente aleatorio.

A modo de ejemplo, la entropía del conjunto que muestra la tabla 18 viene dada por:

$$H(S) = -\frac{3}{7} \log_2 \frac{3}{7} - \frac{4}{7} \log_2 \frac{4}{7} = 0,985$$

La fórmula de la ganancia de información para el conjunto S y el atributo a_j queda como:

$$G(S, a_j) = H(S) - \sum_{v \in a_j} p(v) H(S_v)$$

donde $p(v)$ es la proporción de ejemplos de S que tienen el valor v para el atributo a_j y S_v es el subconjunto de S de los ejemplos que toman el valor v para el atributo a_j .

A modo de ejemplo, la ganancia de información del conjunto que muestra la tabla 18 y el atributo cap_shape viene dada por:

$$G(S, cs) = H(S) - \frac{5}{7} H(S_{cs=convex}) - \frac{2}{7} H(S_{cs=bell}) = 0,292$$

donde cs es $cap - shape$,

$$H(S_{cs=convex}) = -\frac{3}{5} \log_2 \frac{3}{5} - \frac{2}{5} \log_2 \frac{2}{5} = 0,971$$

y

$$H(S_{cs=bell}) = -\frac{0}{2} \log_2 \frac{0}{2} - \frac{2}{2} \log_2 \frac{2}{2} = 0$$

Existen otras variantes de árboles de decisión que utilizan otras medidas para computar la bondad de los atributos. Algunas de ellas son el uso de la varianza cuando nos encontramos atributos numéricos. Otra medida utilizada es la información mutua*, que está relacionada con la entropía cruzada.

Poda de los árboles de decisión

El objetivo de la poda de los árboles de decisión es obtener árboles que no tengan en las hojas reglas que afecten a pocos ejemplos del conjunto de entrenamiento. Es deseable no forzar la construcción del árbol para evitar el sobreentrenamiento*.

* En inglés, *mutual information*.

Ved también

Sobre la información mutua podéis ver el subapartado 3.1.3.

* En inglés, *overfitting*.

La primera aproximación para abordar la solución de este problema consiste en establecer un umbral de reducción de la entropía. Es decir, pararemos una rama cuando la disminución de entropía no supere dicho umbral. Esta aproximación plantea el problema de que puede ser que no se reduzca en un cierto paso pero sí en el siguiente.

Otra aproximación, que soluciona el problema de la anterior y que es la que se suele utilizar, consiste en construir todo el árbol y, en una segunda fase, eliminar los nodos superfluos (poda del árbol*). Para realizar este proceso de poda recorreremos el árbol de forma ascendente (de las hojas a la raíz) y vamos mirando si cada par de nodos incrementa la entropía por encima de un cierto umbral si los juntáramos. Si esto pasa, deshacemos la partición.

* En inglés, *prunning*.

Implementación en Python

El programa 4.5 corresponde a la implementación en Python de los árboles de decisión básicos con atributos nominales. Este programa funcionará para todos los problemas con datos nominales sin valores ausentes que tengan la clase como primera columna.

Código 4.5: Árboles de decisión en Python

```

1 # declaración de funciones
2
3 def bondad(clases , conjunto):
4     p = {}
5     for i in range(len(clases)):
6         p.setdefault(conjunto[i], {})
7         p[conjunto[i]].setdefault(clases[i], 0)
8         p[conjunto[i]][clases[i]] += 1
9
10    return sum([max([(val, p[atr][val])
11                    for val in p[atr].keys()])[1]
12               for atr in p.keys()])
13
14 def claseMF(clases):
15     p = {}
16     for x in clases:
17         p.setdefault(x, 0)
18         p[x] += 1
19     return max([(p[cl], cl) for cl in p.keys()])[1]
20
21 def iteracion2(c):
22     c.pop(1)
23     return (c[0][0], iteracion(list(c[2]), list(zip(*c[1]))))
24
25 def iteracion(cl, cj):
26     l = sorted(set(cl))
27     if len(l)==1:
28         return ("clase", l[0])
29     else:
30         (b, col) = max([(bondad(cl, cj[i]), i)
31                        for i in range(len(cj))])
32         l = cj.pop(col)
33         lu = sorted(set(l))
34         cj = list(zip(*cj))
35         if len(cj) == 0:
36             l = [list(filter(lambda x: x[0] == x[1], y))
37                  for y in [[(val, l[i], cl[i])
38                             for i in range(len(l))]
39                             for val in lu]]
40         return ("atr", col,

```

```

41         [(lp[0][0],
42            ("clase", claseMF(list(list(zip(*lp))[2])))
43            for lp in l])
44     elif b == len(cl):
45         l = [list(filter(lambda x: x[0] == x[1], y))
46              for y in [(val, l[i], cl[i])
47                       for i in range(len(l))
48                       for val in lu]]
49     return ("atr", col,
50            [(lp[0][0],
51               ("clase", claseMF(list(list(zip(*lp))[2])))
52               for lp in l])
53     else:
54         l = [list(filter(lambda x: x[0] == x[1], y))
55              for y in [(val, l[i], list(cj[i]), cl[i])
56                       for i in range(len(l))
57                       for val in lu]]
58     return ('atr', col, [iteracion2(list(zip(*x))
59                                for x in l])
60
61 def testEj(ej, arbol):
62     if isinstance(arbol, tuple):
63         if arbol[0] == "clase":
64             return arbol[1]
65         elif arbol[0] == "atr":
66             valor = ej.pop(arbol[1])
67             arbol = list(filter(lambda x: x[0][0] == x[1],
68                               [(1, valor) for l in arbol[2]]))
69         if len(arbol)==0:
70             return None
71         else:
72             return testEj(ej, arbol[0][0][1])
73     else:
74         return None
75     else:
76         return None
77
78 # carga del archivo
79 l = list(map(lambda l: (l.strip()).split(','),
80             open('mushroom.data.txt', 'r').readlines()))
81
82 # construcción del training: 2 de cada 3
83 train = list(map(lambda x: x[1],
84                 filter(lambda v: v[0] %3 != 0, enumerate(1))))
85 clasesTrain = list(map(lambda x: x.pop(0), train))
86
87 # construcción del test: 1 de cada 3
88 test = list(map(lambda x: x[1],
89                 filter(lambda v: v[0] %3 == 0, enumerate(1))))
90 clasesTest = list(map(lambda x: x.pop(0), test))
91
92 # Construcción del árbol
93 arbol = iteracion(clasesTrain, list(zip(*train)))
94 print(arbol)
95
96 # clasificación
97 predicciones = [testEj(ej, arbol) for ej in test]
98
99 # número de correctos
100 print('Prec.: ', len(list(filter(lambda x: x[0] == x[1],
101                                zip(*[predicciones, clasesTest]))))
102        / len(test) * 100, '%')

```

Si aplicamos el programa al archivo de datos del problema “mushroom” del UCI (Frank y Asunción, 2010) obtendremos una precisión del 100%. El programa muestra el árbol generado como salida con formato parentizado. Este programa selecciona como conjunto de tests uno de cada tres ejemplos del archivo. El resto de ejemplos pertenecerá al conjunto de entrenamiento.

Repositorio

Los archivos de datos y el programa están disponibles en el repositorio de la asignatura.

4.4.2. AdaBoost

Supongamos que queremos realizar una aplicación para teléfonos móviles que ayude a los aficionados a la recolección de setas a discernir las setas venenosas de las comestibles a partir de sus propiedades: forma y color del sombrero.

Este método está específicamente diseñado para combinar reglas. Se basa en la combinación lineal de muchas reglas muy sencillas pero muy precisas (llamadas *reglas débiles**), para crear un clasificador muy robusto con un error arbitrariamente bajo en el conjunto de entrenamiento. Estas reglas débiles se aprenden secuencialmente manteniendo una distribución de pesos sobre los ejemplos de entrenamiento. Estos pesos se van actualizando a medida que vamos adquiriendo nuevas reglas. Esta actualización depende de la dificultad de aprendizaje de los ejemplos de entrenamiento. Es decir, los pesos de los ejemplos serán directamente proporcionales a su dificultad de aprendizaje, y por tanto, la adquisición de nuevas reglas se irá dirigiendo a los ejemplos con mayor peso (y dificultad).

AdaBoost básico

AdaBoost es un algoritmo que pretende obtener una regla de clasificación muy precisa combinando muchos clasificadores débiles, cada uno de los cuales obtiene una precisión moderada. Este algoritmo trabaja eficientemente con espacios de atributos muy grandes y ha sido aplicado con éxito a muchos problemas prácticos. A continuación se muestra el pseudocódigo del algoritmo:

algoritmo AdaBoost

entrada: $S = \{(x_i, y_i), \forall i = 1..m\}$

S es el conjunto de ejemplos de entrenamiento

D_1 es la distribución inicial de pesos

m es el número de ejemplos de entrenamiento

$y_i \in \{-1, +1\}$

$D_1(i) = \frac{1}{m}, \forall i = 1..m$

Se realizan T iteraciones:

para $t := 1$ **hasta** T **hacer**

Se obtiene la hipótesis débil $h_t : X \rightarrow \{-1, +1\}$

$h_t = \text{ObtenerHipótesisDébil}(X, D_t)$

$\epsilon_t = \sum_{i: h_t(x_i) \neq y_i} D_t(i)$

$\alpha_t = \frac{1}{2} \ln\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$

Se actualiza la distribución D_t

$D_{t+1}(i) = \frac{D_t(i) \exp(-\alpha_t y_i h_t(x_i))}{Z_t}, \forall i = 1..m$

Z_t es un factor de normalización tal que D_{t+1} sea una

distribución, por ejemplo: $Z_t = \sum_{i=1}^m D_t(i) \exp(-\alpha_t y_i h_t(x_i))$

fin para

devuelve la combinación de hipótesis: $f(x) = \sum_{t=1}^T \alpha_t h_t(x)$

fin AdaBoost

* En inglés, *weak rules* o *weak hypotheses*.

Lectura complementaria

El algoritmo AdaBoost fue propuesto en: Y. Freund; R. E. Schapire (1997). "A Decision-theoretic Generalization of On-line Learning and an Application to Boosting". *Journal of Computer and System Sciences* (vol. 1, núm. 55).

Referencia bibliográfica

Y. Freund; R. E. Schapire (1999). "A Short Introduction to Boosting". *Journal of Japanese Society for Artificial Intelligence* (vol. 5, núm. 14, págs. 771-780).

Las hipótesis débiles se aprenden secuencialmente. En cada iteración se aprende una regla que se sesga para clasificar los ejemplos con más dificultades por parte del conjunto de reglas precedentes. AdaBoost mantiene un vector de pesos como una distribución D_t sobre los ejemplos. En la iteración t , el objetivo del algoritmo es encontrar una hipótesis débil, $h_t : X \rightarrow \{-1, +1\}$, con un error moderadamente bajo con respecto a la distribución de pesos D_t . En este marco, las hipótesis débiles $h_t(x)$ dan como predicciones valores reales que corresponden a valores de confianza. Inicialmente, la distribución de pesos D_1 es uniforme, y en cada iteración, el algoritmo de *boosting* incrementa (o decreta) exponencialmente los pesos $D_t(i)$ en función de si $h_t(x_i)$ realiza una buena (o mala) predicción. Este cambio exponencial depende de la confianza $|h_t(x_i)|$. La combinación final de hipótesis, $h : X \rightarrow \{-1, +1\}$, calcula sus predicciones ponderando con pesos los votos de las diferentes hipótesis débiles,

$$f(x) = \sum_{t=1}^T \alpha_t \cdot h_t(x)$$

Para cada ejemplo x , el signo de $f(x)$ se interpreta como la clase predicha (el AdaBoost básico trabaja sólo con dos clases de salida, -1 o $+1$), y la magnitud $|f(x)|$ como una medida de la confianza de la predicción. Esta función se puede usar para clasificar nuevos ejemplos o para realizar un ranking de los mismos en función de un grado de confianza.

Ejemplo de aplicación

El algoritmo Adaboost tiene dos parámetros a determinar: el número máximo de iteraciones y la forma de construcción de las reglas débiles. En este subapartado vamos a ver la aplicación del algoritmo escogiendo como conjunto de entrenamiento los datos de la tabla 22 y como test el de la tabla 23. Fijamos el número máximo de iteraciones a 5 y como reglas débiles, reglas del estilo: *atributo = valor* y *atributo ≠ valor*.

Tabla 22. Conjunto de entrenamiento

class	cap-shape	cap-color
+1	convex	brown
-1	convex	yellow
-1	bell	white
+1	convex	white

Fuente: Problema "mushroom" del repositorio UCI (Frank y Asunción, 2010).

Tabla 23. Ejemplo de test

class	cap-shape	cap-color
-1	bell	yellow

Fuente: Problema "mushroom" del repositorio UCI (Frank y Asunción, 2010).

Como primer paso del algoritmo inicializamos los 4 elementos del vector de pesos D_1 a 0,25. A continuación, empezamos el proceso iterativo. Lo primero es calcular la regla débil. La tabla 24 muestra las diferentes reglas aplicables* y su error. La regla $cs = bell$ asignará las predicciones: -1, -1, +1 y -1, respectivamente. Para calcular el error sumamos el peso $D_t(i)$ de los ejemplos cuyas predicciones sean diferentes a sus clases. En este caso sumaríamos el peso del primer, segundo y cuarto ejemplos (en total 0,75). El resto de casos los calcularíamos de manera similar.

* Donde cs corresponde a *cap-shape* y cc a *cap-color*.

Tabla 24. Reglas débiles

Regla	Error
$cs = bell$	0,75
$cs \neq bell$	0,25
$cs = convex$	0,25
$cs \neq convex$	0,75
$cc = brown$	0,25
$cc \neq brown$	0,75
$cc = white$	0,5
$cc \neq white$	0,5
$cc = yellow$	0,75
$cc \neq yellow$	0,25

Escogemos como regla débil una de las que tengan error mínimo (en el ejemplo hemos escogido $cs \neq bell$). Así, el error de la primera iteración es $\epsilon_1 = 0,25$ y por tanto $\alpha_1 = \frac{1}{2} \ln(\frac{1-\epsilon_1}{\epsilon_1}) = 0,5493$.

El paso siguiente consiste en actualizar el vector de pesos. Empezamos por calcular $D_1(i) \exp(-\alpha_1 y_i h_1(x_i))$. La tabla 25 muestra el valor de esta expresión para los 4 ejemplos y el valor de todas sus componentes. La última columna muestra los valores finales del vector de pesos para la segunda iteración.

Tabla 25. Cálculo de D_2

i	$D_1(i)$	α_1	y_i	$h_1(x_i)$	numerador	$D_2(i)$
1	0,25	0,5493	+1	+1	0,1443376	0,1667
2	0,25	0,5493	-1	+1	0,4330127	0,5
3	0,25	0,5493	-1	-1	0,1443376	0,1667
4	0,25	0,5493	+1	+1	0,1443376	0,1667

El resto de iteraciones se calcularían de forma similar. La tabla 26 muestra los resultados de las 5 iteraciones del entrenamiento.

Tabla 26. Evolución de los pesos

t	α_t	regla $_t$	D_t
1	0,549	$cs \neq bell$	(0,25,0,25,0,25,0,25)
2	0,805	$cc = brown$	(0,167,0,5,0,167,0,167)
3	1,099	$cc \neq yellow$	(0,999,0,3,0,099,0,499)
4	0,805	$cs \neq bell$	(0,056,0,167,0,5,0,278)
5	0,805	$cc = brown$	(0,033,0,5,0,3,0,167)
6			(0,019,0,3,0,18,0,5)

El clasificador da una predicción correcta sobre el ejemplo que muestra la tabla 23. Para obtener la predicción tenemos que calcular:

$$\begin{aligned} \text{signo}(\sum_{t=1}^5 \alpha_t h_t(x)) &= \\ &= \text{signo}(0,549 \times (-1) + 0,805 \times (-1) + \\ &+ 1,099 \times (-1) + 0,805 \times (-1) + 0,805 \times (-1)) = \\ &= \text{signo}(-4,063) = -1 \end{aligned}$$

Análisis del método

Para garantizar la estabilidad del algoritmo tenemos que asegurarnos de que el error se mantenga entre 0 y 0,5. Si el error sube de esta medida, no podemos asegurar la convergencia del mismo.

Este algoritmo se ha aplicado con éxito a muchos problemas prácticos, incluidos la categorización de textos o el filtro anti spam.

Las ventajas que nos aporta este método son varias. En primer lugar, tiene pocos parámetros: el número máximo de iteraciones y la forma de las hipótesis débiles. La implementación de este algoritmo es relativamente sencilla.

Por otro lado, al finalizar el proceso, nos deja información muy útil. De las reglas generadas se puede extraer conocimiento que puede ser interpretado por humanos. Los pesos finales indican qué ejemplos no se han podido modelizar. Esto nos da información de posibles *outliers* o posibles errores de etiquetado.

El mayor inconveniente que tiene es su coste computacional. Se han dedicado esfuerzos a tratar este tema, como la versión LazyBoosting* en que en cada iteración sólo se explora un subconjunto aleatorio de las posibles reglas débiles. En este trabajo se reporta una reducción sustancial del coste computacional con una reducción mínima de los resultados.

El AdaBoost es un algoritmo teóricamente bien fundamentado. El error de generalización de la hipótesis final puede ser acotado en términos del error de entrenamiento. Este algoritmo está basado en la maximización del margen, lo que hace que tenga muchas similitudes con las máquinas de vectores de soporte. Estos dos conceptos están tratados con mayor profundidad en el subapartado 4.5.

* G. Escudero; L. Màrquez; G. Rigau (2000). "Boosting Applied to Word Sense Disambiguation". In *Proceedings of the 12th European Conference on Machine Learning*.

El algoritmo básico sólo permite clasificar entre dos clases. Para tratar problemas multiclase, en lugar de binarios, tenemos que utilizar la variante AdaBoost.MH.

Implementación en Python

El programa 4.6 corresponde a la implementación en Python del AdaBoost binario básico con atributos nominales. Este programa funcionará para todos los problemas con datos nominales sin valores ausentes que tengan la clase como primera columna y codificada como -1 y $+1$. Tiene definido el máximo de iteraciones en 50.

Código 4.6: Adaboost en Python

```

1 from itertools import repeat
2 from math import log
3 from math import exp
4
5 # parámetros
6 T = 50
7
8 # declaración de funciones
9
10 def hti(atvl, val, igual):
11     if (igual and (val == atvl)) or (not igual and (val != atvl)):
12         pred = +1
13     else:
14         pred = -1
15     return pred
16
17 def error(Dt, cl, atvl, val, igual):
18     s = 0
19     for i in range(len(cl)):
20         if hti(atvl[i], val, igual) != int(cl[i]):
21             s += Dt[i]
22     return s
23
24 def WeakLearner(Dt, tr):
25     ll = []
26     for atr in range(len(tr)-1):
27         for val in sorted(set(tr[atrl+1])):
28             et = error(Dt, tr[atrl], tr[atrl+1], val, True)
29             ll.append([et, atr+1, val, True])
30             et = error(Dt, tr[atrl], tr[atrl+1], val, False)
31             ll.append([et, atr+1, val, False])
32     return min(ll)
33
34 def testEj(ej, modelo):
35     return sum([regla[0] * hti(ej[regla[2]], regla[3], regla[4])
36               for regla in modelo])
37
38 # carga del archivo
39 l = list(map(lambda l: (l.strip()).split(','),
40             open('mushroom.adaboost.txt', 'r').readlines()))
41
42 # construcción del training: 2 de cada 3
43 train = list(map(lambda x: x[1],
44                 filter(lambda v: v[0] % 3 != 0, enumerate(1))))
45
46 # construcción del test: 1 de cada 3
47 test = list(map(lambda x: x[1],
48                filter(lambda v: v[0] % 3 == 0, enumerate(1))))
49
50 # entrenamiento
51 tt = list(zip(*train))
52 m = len(train)

```

Lectura complementaria

El AdaBoost.MH fue propuesto en: R. E. Schapire; Y. Singer (2000). "Boostexter: A Boosting-based System for Text Categorization". *Machine Learning* (vol. 39, núm. 2-3)

```

53 Dt = list(repeat(1/m, m))
54 modelo = []
55 for i in range(T):
56     [et, atr, val, igual] = WeakLearner(Dt, tt)
57     if et > 0.5:
58         print("Error: _et_out_of_range!")
59         break
60     if et == 0:
61         print("Esta_todo_bien_clasificado!")
62         break
63     alfat = 1 / 2 * log((1 - et) / et)
64     Dt2 = [Dt[j] * exp(-alfat * hti(tt[atr][j], val, igual) *
65             float(train[j][0])) for j in range(m)]
66     s = sum(Dt2)
67     Dt = [Dt2[j]/s for j in range(m)]
68     modelo.append((alfat, et, atr, val, igual))
69 print(len(modelo), "reglas_aprendidas!")
70
71 # clasificación
72 predicciones = [testEj(ej, modelo) for ej in test]
73
74 # número de correctos
75 print('Prec.: ', len(list(filter(
76     lambda x: x[0] * float(x[1][0]) > 0,
77     zip(*[predicciones, test]))))
78     / len(test) * 100, '%')

```

Si aplicamos el programa al archivo de datos del problema “mushroom” del UCI (Frank y Asunción, 2010) obtendremos una precisión del 99,889%. Este programa selecciona como conjunto de tests uno de cada tres ejemplos del archivo. El resto de ejemplos pertenecerá al conjunto de entrenamiento.

Repositorio

Los archivos de datos y el programa están disponibles en el repositorio de la asignatura.

4.5. Clasificadores lineales y métodos basados en kernels

En este subapartado empezaremos por ver un algoritmo equivalente al clasificador lineal que se describe en el subapartado 4.3.2. En la versión de este subapartado sustituiremos la distancia euclídea por el producto escalar. Esta modificación nos permitirá introducir el uso de los kernels: una técnica muy utilizada en la última década para el tratamiento de conjuntos no lineales con algoritmos lineales. Para finalizar el subapartado veremos el algoritmo de las máquinas de vectores de soporte; posiblemente el algoritmo de aprendizaje supervisado más importante del momento.

4.5.1. Clasificador lineal basado en producto escalar

Supongamos que queremos realizar una aplicación para un almacén de flores, donde llegan a diario miles de productos. Disponemos de un sistema láser que nos suministra una serie de medidas sobre las flores y nos piden que el sistema las clasifique automáticamente para transportarlas mediante un robot a las diferentes estanterías del almacén. Las medidas que envía el sistema láser son: la longitud y anchura del sépalo y el pétalo de cada flor.

Un clasificador lineal (binario) es un hiperplano en un espacio n -dimensional de atributos que puede ser representado por un vector de pesos w y un umbral* b que está relacionado con la distancia del hiperplano al origen: $h(x) = \text{signo}(\langle w, x \rangle + b)$ (donde $\langle \cdot, \cdot \rangle$ representa el producto escalar). Cada componente del vector de pesos se corresponde con uno de los atributos.

* En inglés, *bias*.

El ejemplo más simple de clasificador lineal es el método basado en los centroides descrito en el subapartado 4.3.2. Una vez calculados los centroides p y n como el de los ejemplos con clases $+1$ y -1 respectivamente, podemos expresar la predicción de un nuevo ejemplo como:

$$h(x) = \begin{cases} +1, & \text{si } de(x,p) < de(x,n) \\ -1, & \text{en otro caso} \end{cases}$$

donde $de(x,p)$ corresponde a la distancia euclídea entre x y p .

Sabiendo que la distancia euclídea se puede expresar como $\|x - p\|^2$, la expresión anterior se puede reformular como:

$$h(x) = \text{signo}(\|x - n\|^2 - \|x - p\|^2)$$

Teniendo en cuenta que $\|a - b\|^2 = \langle a, a \rangle + \langle b, b \rangle - 2\langle a, b \rangle$, desarrollando el argumento de la función signo obtenemos:

$$\begin{aligned} \|x - n\|^2 - \|x - p\|^2 &= \\ &= \langle x, x \rangle + \langle n, n \rangle - 2\langle x, n \rangle - \langle x, x \rangle - \langle p, p \rangle + 2\langle x, p \rangle = \\ &= 2\langle x, p \rangle - 2\langle x, n \rangle - \langle p, p \rangle + \langle n, n \rangle = \\ &= 2\langle x, p - n \rangle - \langle p, p \rangle + \langle n, n \rangle \end{aligned}$$

Para finalizar este desarrollo, obtendremos la relación con w y b :

$$\begin{aligned} h(x) &= \text{signo}(2\langle x, p - n \rangle - \langle p, p \rangle + \langle n, n \rangle) = \\ &= \text{signo}(\langle x, p - n \rangle - \frac{1}{2}(\langle p, p \rangle - \langle n, n \rangle)) = \text{signo}(\langle w, x \rangle - b) \end{aligned}$$

Y por tanto:

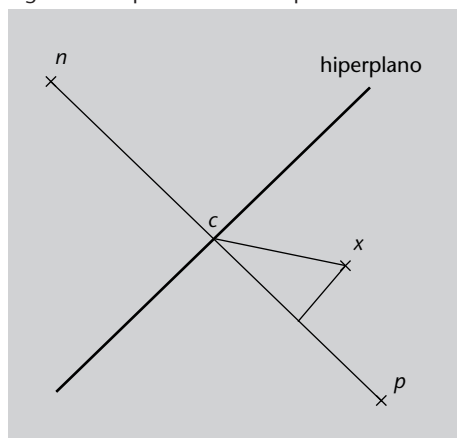
$$\begin{aligned} w &= p - n \\ b &= \frac{1}{2}(\langle p, p \rangle - \langle n, n \rangle) \end{aligned}$$

Producto escalar

El producto escalar es la proyección de un vector sobre el otro. Si tenemos un punto x como el de la figura 29 y los dos centroides p para los ejemplos de la clase +1, n para los -1 y c como el punto medio entre ambos, podemos aprovechar el producto escalar para obtener la predicción. La predicción para este punto vendrá dada por el signo del producto escalar de los vectores $c \rightarrow p$ y $c \rightarrow x$:

$$h(x) = \text{signo}(\langle p - c, x - c \rangle)$$

Figura 29. Representación del producto escalar



Desarrollando esta expresión obtenemos:

$$y = \text{signo}(\langle p - c, x \rangle + \langle c, c - p \rangle) = \text{signo}(\langle w, x \rangle + b)$$

Y por tanto:

$$w = p - c = p - \frac{p + n}{2} = \frac{p - n}{2}$$

$$b = \langle c, c - p \rangle = \langle \frac{p+n}{2}, \frac{p+n}{2} - p \rangle = \langle \frac{p+n}{2}, \frac{n-p}{2} \rangle =$$

$$= \frac{\langle p, n \rangle - \langle p, p \rangle + \langle n, n \rangle - \langle n, p \rangle}{4} = \frac{\langle n, n \rangle - \langle p, p \rangle}{4}$$

La magnitud del producto escalar nos está dando una medida de la distancia del punto a clasificar al hiperplano, que se puede interpretar como una medida de la confianza de la predicción.

Fijaos en que con los dos desarrollos hemos llegado al mismo resultado*. Existe una relación directa entre las distancias euclídeas y el producto escalar.

* Notad que multiplicar por un valor positivo no cambia el resultado de la función signo.

Ejemplo de aplicación

La tabla 27 muestra un conjunto de entrenamiento en el que tenemos que clasificar flores a partir de sus propiedades. Los centroides se calculan a partir de los promedios de los diferentes atributos y corresponden respectivamente a: $p = (5,0, 3,25, 1,4, 0,2)$ y $n = (5,85, 2,9, 4,15, 1,35)$.

Tabla 27. Conjunto de entrenamiento

class	sepal-length	sepal-width	petal-length	petal-width
+1	5,1	3,5	1,4	0,2
+1	4,9	3,0	1,4	0,2
-1	6,1	2,9	4,7	1,4
-1	5,6	2,9	3,6	1,3

Fuente: Problema "iris" del repositorio UCI (Frank y Asunción, 2010).

Una vez obtenidos los centroides, pasamos a obtener los valores del hiperplano w y b .

$$w = \frac{p-n}{2} = (-0,425, 0,175, -1,375, -0,575)$$

$$b = \frac{\langle n,n \rangle - \langle p,p \rangle}{4} = 6,02875$$

A partir de este momento, si queremos clasificar el ejemplo que muestra la tabla 28 evaluamos la fórmula:

$$\begin{aligned} y &= \text{signo}(\langle w,x \rangle + b) = \\ &= \text{signo}(\langle (-0,425, 0,175, -1,375, -0,575), (4,9, 3,1, 1,5, 0,1) \rangle + 6,02875) = \\ &= \text{signo}(2,36875) = +1 \end{aligned}$$

Tabla 28. Ejemplo de test

class	sepal-length	sepal-width	petal-length	petal-width
+1	4,9	3,1	1,5	0,1

Fuente: Problema "iris" del repositorio UCI (Frank y Asunción, 2010).

Análisi del método

Este tipo de método ha sido muy utilizado en el área de la recuperación de la información*, ya que es una manera simple, eficiente y efectiva de construir modelos para la categorización de textos. Hay muchos algoritmos para entrenar este tipo de clasificadores (perceptrón, Widrow-Hoff, Winnow...). Un algoritmo de esta familia ha ganado mucha importancia en la última década, las máquinas de vectores de soporte.

* En inglés, *information retrieval*.

Implementación en Python

El programa 4.7 corresponde a la implementación en Python del clasificador lineal binario basado en los centroides que utiliza el producto escalar en lugar de la distancia euclídea. Este programa funcionará para todos los problemas con datos numéricos sin valores ausentes que tengan la clase como última columna y codificada como -1 y $+1$.

Código 4.7: Lineal con producto escalar en Python

```

1 from functools import reduce
2
3 # declaración de funciones
4
5 def prodEscalar(a, b):
6     return sum(a[i]*b[i] for i in range(len(a)))
7
8 def h(t, w, b):
9     y = prodEscalar(w, t) - b
10    if y > 0:
11        return +1
12    else:
13        return -1
14
15 # carga del archivo
16 l = list(map(lambda l: (l.strip()).split(','),
17             open('iris.data.txt', 'r').readlines()))
18
19 # construcción del training: 2 de cada 3
20 train = list(map(lambda x: x[1],
21                filter(lambda v: v[0] %3 != 0, enumerate(1))))
22
23 # construcción del test: 1 de cada 3
24 test = list(map(lambda x: x[1],
25                filter(lambda v: v[0] %3 == 0, enumerate(1))))
26 clasesTest = list(map(lambda x: int(x.pop(len(x)-1)), test))
27
28 # entrenamiento
29 tp = list(zip(*filter(
30     lambda x: x[len(train[0])-1] == '+1', train)))
31 tp.pop(len(tp)-1)
32 tn = list(zip(*filter(
33     lambda x: x[len(train[0])-1] == '-1', train)))
34 tn.pop(len(tn)-1)
35 p = list(map(lambda l: reduce(lambda x, y: float(x)+float(y), l) /
36             len(l), tp))
37 n = list(map(lambda l: reduce(lambda x, y: float(x)+float(y), l) /
38             len(l), tn))
39 w = list(map(lambda a, b: a - b, p, n))
40 b = (prodEscalar(p, p) - prodEscalar(n, n)) / 2
41 # clasificación
42 predicciones = [h(list(map(lambda x: float(x), t)), w, b)
43                 for t in test]
44
45 # número de correctos
46 print('Prec.:', len(list(filter(lambda x: x[0] == x[1],
47                               zip(*[predicciones, clasesTest]))))
48       / len(test) * 100, '%')

```

Si aplicamos el programa al archivo de datos del problema “iris” del UCI (Frank y Asunción, 2010), utilizando la clase *Iris-setosa* como clase $+1$ y *Iris-versicolor* como -1 , obtendremos una precisión del 100%. Este programa selecciona como conjunto de tests uno de cada tres ejemplos del archivo. El resto de ejemplos pertenecerá al conjunto de entrenamiento.

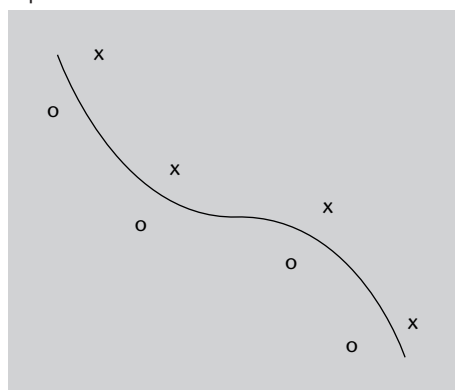
Repositorio

Los archivos de datos y el programa están disponibles en el repositorio de la asignatura.

4.5.2. Clasificador lineal con kernel

Los algoritmos lineales como el anterior están pensados para clasificar conjuntos linealmente separables. En el caso de un espacio de atributos que se pueda representar en el plano, el hiperplano correspondería a una línea recta. En muchos problemas reales, no se suele dar esta circunstancia; son conjuntos para los que no existen hiperplanos lineales que los puedan separar. La figura 30 muestra un ejemplo de uno de estos casos. Se han dedicado muchos esfuerzos en la literatura a crear algoritmos no lineales para abordar este tipo de problemas. El gran inconveniente que plantean la mayoría de estos métodos es su elevado coste computacional.

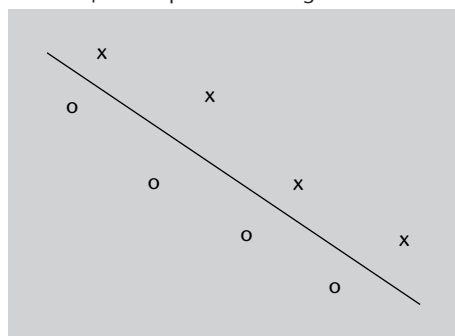
Figura 30. Ejemplo de conjunto no linealmente separable



En las dos últimas décadas se llevan usando con mucha efectividad los así llamados *kernels* (o funciones núcleo). El uso de los mismos permite atacar problemas no lineales con algoritmos lineales. La estrategia que se sigue consiste en proyectar los datos a otros espacios de atributos en los que sí que exista un hiperplano lineal que los pueda separar*. Con esto conseguimos el coste computacional de los algoritmos lineales con sólo un pequeño incremento que conlleva la proyección de los datos a otro espacio de atributos.

Así, si conocemos una función ϕ que proyecte los puntos de la figura 30 a un espacio de atributos como el que muestra la figura 31, podremos encontrar un hiperplano lineal en el nuevo espacio que separe los dos tipos de puntos.

Figura 31. Ejemplo de proyección con cierta función ϕ de los puntos de la figura 30



Lecturas complementarias

J. Shawe-Taylor; N. Cristianini (2004). *Kernel Methods for Pattern Analysis*. UK: Cambridge University Press.

Ved también

En el subapartado 4.5.4 se da una interpretación geométrica de los kernels y su uso.

* En la literatura se le suele citar como el truco o estratagema del kernel, en inglés *kernel trick*.

De hecho, no se trabaja con las funciones tipo ϕ , sino con las así llamadas funciones kernel, $\kappa(x,z) = \langle \phi(x), \phi(z) \rangle$. Cuando se trabaja con kernels se suele utilizar una estructura modular en la que primero se proyectan los puntos (ejemplos de entrenamiento) dos a dos con la función κ , creando así una matriz cuadrada. A esta matriz se la conoce como matriz kernel*. Los algoritmos de aprendizaje pueden soler actuar sobre el kernel κ o sobre esta matriz directamente. La notación estándar para mostrar las matrices kernel es:

$$\begin{pmatrix} \kappa(x_1, x_1) & \kappa(x_1, x_2) & \cdots & \kappa(x_1, x_l) \\ \kappa(x_2, x_1) & \kappa(x_2, x_2) & \cdots & \kappa(x_2, x_l) \\ \vdots & \vdots & \ddots & \vdots \\ \kappa(x_l, x_1) & \kappa(x_l, x_2) & \cdots & \kappa(x_l, x_l) \end{pmatrix}$$

donde l corresponde al número de ejemplos de entrenamiento. Las matrices kernel han de cumplir que sean simétricas y semidefinidas positivamente. De manera equivalente, su función kernel asociada tiene que cumplir la misma propiedad.

* En inglés, *kernel matrix*.

Matriz semidefinida positiva

Se dice que una matriz simétrica es semidefinida positiva cuando todos sus valores propios son superiores o iguales a cero.

A partir de este momento vamos a desarrollar la formulación teniendo en cuenta que conocemos una supuesta función kernel κ , pero no conocemos su función ϕ relativa. Como en el primer desarrollo del algoritmo anterior, supongamos que p y n corresponden a los centroides de los ejemplos con clases +1 y -1 respectivamente, y $|p|$ y $|n|$ al número de ejemplos de las clases +1 y -1 respectivamente. La predicción de un nuevo ejemplo x viene dada por:

$$h(x) = \text{signo}(\|\phi(x) - \phi(n)\|^2 - \|\phi(x) - \phi(p)\|^2)$$

Para continuar desarrollando esta expresión tenemos que tener en cuenta que $\kappa(n,n) = \langle \phi(n), \phi(n) \rangle$ es equivalente al promedio de las proyecciones dos a dos de los elementos con clase -1. Con p y los ejemplos de la clase +1 funciona de forma similar. Si desarrollamos el argumento de la función signo teniendo en cuenta estas observaciones, obtenemos:

$$\begin{aligned} & \|\phi(x) - \phi(n)\|^2 - \|\phi(x) - \phi(p)\|^2 = \\ & \kappa(x,x) + \frac{1}{|n|^2} \sum_{\{i|y_i=-1\}} \sum_{\{j|y_j=-1\}} \kappa(x_i, x_j) - \frac{2}{|n|} \sum_{\{i|y_i=-1\}} \kappa(x, x_i) - \\ & -\kappa(x,x) - \frac{1}{|p|^2} \sum_{\{i|y_i=+1\}} \sum_{\{j|y_j=+1\}} \kappa(x_i, x_j) + \frac{2}{|p|} \sum_{\{i|y_i=+1\}} \kappa(x, x_i) = \\ & = \frac{2}{|p|} \sum_{\{i|y_i=+1\}} \kappa(x, x_i) - \frac{2}{|n|} \sum_{\{i|y_i=-1\}} \kappa(x, x_i) - \\ & -\frac{1}{|p|^2} \sum_{\{i|y_i=+1\}} \sum_{\{j|y_j=+1\}} \kappa(x_i, x_j) + \frac{1}{|n|^2} \sum_{\{i|y_i=-1\}} \sum_{\{j|y_j=-1\}} \kappa(x_i, x_j) \end{aligned}$$

Y de aquí obtenemos:

$$b = \frac{1}{2} \left(\frac{1}{|p|^2} \sum_{\{i|y_i=+1\}} \sum_{\{j|y_j=+1\}} \kappa(x_i, x_j) - \frac{1}{|n|^2} \sum_{\{i|y_i=-1\}} \sum_{\{j|y_j=-1\}} \kappa(x_i, x_j) \right)$$

$$h(x) = \text{signo} \left(\frac{1}{|p|} \sum_{\{i|y_i=+1\}} \kappa(x, x_i) - \frac{1}{|n|} \sum_{\{i|y_i=-1\}} \kappa(x, x_i) - b \right)$$

Notad que en estas expresiones resultantes no aparece ϕ por ningún lado. Esta forma en la que no aparece ϕ recibe el nombre de *formulación dual*, en contraposición a la *formulación primal*. Otra cuestión a tener en cuenta de cara a la implementación es que la b es una constante que se tiene que calcular una sola vez, ya que no aparece el ejemplo de test. No tendremos que recalcularlo para cada uno de los ejemplos de test.

Construcción de kernels

Hay un conjunto de reglas llamadas propiedades de la clausura que nos ayudan en la construcción de nuevos kernels. Si tenemos que κ_1 y κ_2 son kernels sobre $X \times X$, $X \subseteq \mathbb{R}^n$, $a \in \mathbb{R}^+$, $f(\cdot)$ una función real sobre X , $\phi : X \rightarrow \mathbb{R}^N$ con κ_3 un kernel sobre $\mathbb{R}^N \times \mathbb{R}^N$, y B una matriz simétrica semidefinida positiva de $n \times n$, las siguientes funciones son kernels:

- 1) $\kappa(x, z) = \kappa_1(x, z) + \kappa_2(x, z)$
- 2) $\kappa(x, z) = a\kappa_1(x, z)$
- 3) $\kappa(x, z) = \kappa_1(x, z)\kappa_2(x, z)$
- 4) $\kappa(x, z) = f(x)f(z)$
- 5) $\kappa(x, z) = \kappa_3(\phi(x), \phi(z))$
- 6) $\kappa(x, z) = x' B z$

Kernels habituales

Hay dos kernels no lineales de aplicación general que aparecen en todas las implementaciones de métodos basados en kernels. A continuación damos su caracterización.

Si $\kappa_1(x, z)$ es un kernel sobre $X \times X$, donde $x, z \in X$, y $p(x)$ es un polinomio con coeficientes positivos y $\sigma > 0$, entonces las siguientes funciones también son kernels:

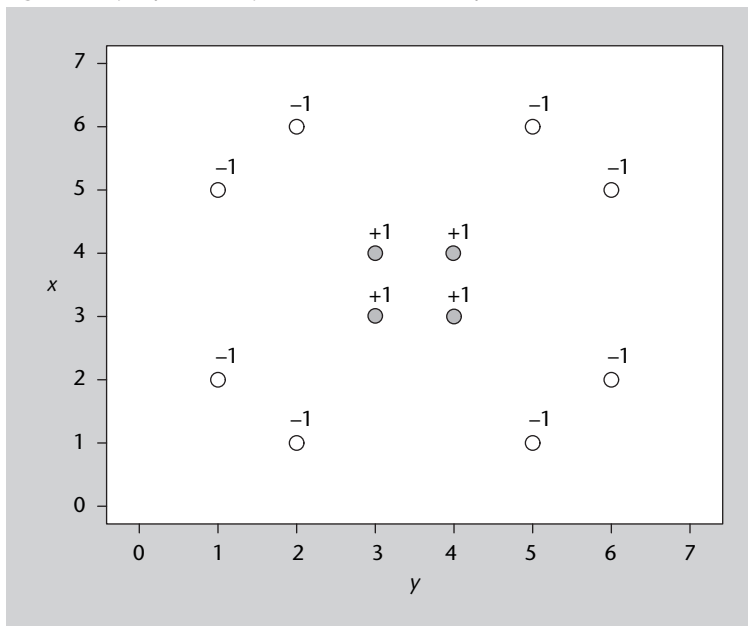
- 1) $\kappa(x, z) = p(\kappa_1(x, z))$
- 2) $\kappa(x, z) = \exp\left(-\frac{\|x-z\|^2}{2\sigma^2}\right)$

El primer kernel es conocido como *kernel polinómico* y el tercero como *kernel gaussiano* o *RBF kernel*.

Ejemplo de aplicación

El conjunto de datos que muestra la figura 32 no es linealmente separable. Si calculamos los centroides y aplicamos el conjunto de *training* como conjunto de test, logramos una precisión del 66%. ¿Cómo podemos separar este conjunto?

Figura 32. Ejemplo de conjunto linealmente no separable



Este conjunto de datos se puede aprender con un kernel radial

$$\kappa(x,z) = \exp\left(-\frac{\|x-z\|^2}{2\sigma^2}\right)$$

que en ocasiones aparece como $\kappa(x,z) = \exp(-\gamma\|x-z\|^2)$.

Si escogemos como conjunto de entrenamiento los puntos de la figura 32, como ejemplo de test el punto $x_t = (3,6)$ y un kernel radial con $\gamma = 1$, obtenemos:

$$b = 0,1629205972066024$$

$$h(x_t) = \text{signo}(-0,2057373019348639) = -1$$

La predicción de -1 es coherente con la posición del punto $(3,6)$ en la figura 32.

Implementación en Python

El programa 4.8 corresponde a la implementación en Python del clasificador lineal binario basado en los centroides con kernel gaussiano. Este programa funcionará para todos los problemas con datos numéricos sin valores ausentes que tengan la clase como última columna y codificada como -1 y $+1$.

Código 4.8: Lineal con kernel RBF en Python

```

1 from functools import reduce
2 from math import exp
3
4 # declaración de kernel
5
6 def kernel(a, b, gamma=1):
7     return exp(-gamma *
8               sum(map(lambda x, y:
9                       (float(x) - float(y)) ** 2, a, b)))
10
11 # declaración de funciones
12
13 def h(x, tp, tn, cp, cn, b):
14     y = (sum([kernel(x, xi) for xi in tp]) / cp -
15         sum([kernel(x, xi) for xi in tn]) / cn - b)
16     if y > 0:
17         return '+1'
18     else:
19         return '-1'
20
21 train = list(map(lambda l: (l.strip()).split(','),
22                 open('punts.txt', 'r').readlines()))
23
24 # entrenamiento
25 tp = list(zip(*filter(lambda x: x[len(train[0])-1] == '+1', train)))
26 tp.pop(len(tp)-1)
27 tp = list(zip(*tp))
28 cp = len(tp)
29 tn = list(zip(*filter(lambda x: x[len(train[0])-1] == '-1', train)))
30 tn.pop(len(tn)-1)
31 tn = list(zip(*tn))
32 cn = len(tn)
33
34 b = (sum([sum([kernel(xi, xj) for xi in tp])
35         for xj in tp]) / (cp ** 2) -
36      sum([sum([kernel(xi, xj) for xi in tn])
37         for xj in tn]) / (cn ** 2)) / 2
38
39 test = [['3', '6']]
40 clasesTest = ['-1']
41 predicciones = [h(x, tp, tn, cp, cn, b) for x in test]
42
43 # número de correctos
44 print('Prec.: ', len(list(filter(lambda x: x[0] == x[1],
45                                zip(*[predicciones, clasesTest]))))
46       / len(test) * 100, '%')

```

Si aplicamos el programa al archivo de datos del problema de la figura 32 obtendremos una predicción correcta para el punto (3,6).

Análisis del método

La definición de kernels está a medio camino entre la representación de la información y el algoritmo de aprendizaje. Los kernels están muy relacionados

Repositorio

Los archivos de datos y el programa están disponibles en el repositorio de la asignatura.

con el espacio de atributos de los problemas. Es decir, no podemos decir que exista un kernel mejor que otro. Cada kernel funciona con los problemas a los que mejor se adapta por sus características intrínsecas. Para el problema de las flores que muestra la tabla 27, funciona bien el kernel lineal pero no funciona el kernel radial, y para el que muestra la figura 32, pasa exactamente lo contrario.

El uso de kernels nos permite adaptar los algoritmos a espacios de atributos más complejos o específicos al problema. Con ellos se ha logrado mejorar el comportamiento de los algoritmos de aprendizaje para muchos problemas, abordar problemas que hasta el momento no tenían solución e ir más allá y crear nuevos algoritmos que tratan datos estructurados, como pueden ser los análisis sintácticos del lenguaje natural o el tratamiento de vídeos. Además, el acceso a estos espacios de atributos complejos y flexibles se consigue con un coste computacional (en espacio y tiempo) relativamente bajo.

Una de las grandes ventajas de estos métodos, además de su flexibilidad, es que están teóricamente muy bien fundamentados y se han demostrado características como su eficiencia computacional, robustez y estabilidad estadística.

En la bibliografía se habla del cuello de botella de la matriz kernel, como del límite que impone el trabajar con ella para todo. Todo lo que queramos representar de los problemas se ha de poder expresar en forma de las funciones kernel y por tanto de la matriz kernel. En algunos casos podemos encontrar limitaciones.

El uso de kernels no está acotado a los algoritmos de clasificación. Esta misma técnica se ha utilizado en métodos *clustering* como los que se describen en el subapartado 2.3 o en los algoritmos de selección de atributos descritos en el apartado 3. Así, existen algoritmos como el kernel PCA o el kernel k-means. Los algoritmos como el kernel PCA o el kernel CCA se están utilizando en problemas de visión como el reconocimiento de caras.

Una de las mayores ventajas del uso de kernels (funciones núcleo) es el diseño de kernels específicos para tareas concretas. En los últimos tiempos se está investigando mucho en la creación de kernels para tratar conjuntos, cadenas de texto, grafos o árboles, entre otros*. Algunos de estos kernels son los específicos para tratar texto**, imágenes o vídeos***.

4.5.3. Kernels para tratamiento de textos

En este subapartado vamos a profundizar en uno de los kernels específicos, el kernel para el tratamiento de textos que nos servirá para abordar el problema de la categorización de textos.

* J. Shawe-Taylor; N. Cristianini (2004). *Kernel Methods for Pattern Analysis*. UK: Cambridge University Press.

** T. Joachims (2001). *Learning to Classify Text Using Support Vector Machines*. Kluwer Academic Publishers.

*** F. Camastra; A. Vinciarelli (2008). *Machine Learning for Audio, Image and Video Analysis: Theory and Applications* (Advanced Information and Knowledge Processing). Springer.

Supongamos que queremos diseñar un clasificador para una agencia de noticias que sepa distinguir los documentos relacionados con *adquisiciones corporativas*.

El primer paso que tenemos que seguir para poder abordar este problema es pensar en la codificación de los datos. Una de las formas más simples para representar texto es mediante un conjunto de palabras*. A cada palabra se le puede asignar si aparece o no en el documento, el número de veces que aparece o la frecuencia relativa de la palabra en el documento. La frecuencia relativa evita el sesgo que el tamaño de los ejemplos (documentos) produce en los resultados.

* En inglés, *bag of words*.

En el conjunto de datos de que disponemos*, ya está preprocesado y el vector contiene la información de las frecuencias relativas. La tabla 29 muestra la primera parte del contenido del primer ejemplo de entrenamiento. El primer valor corresponde a la clase, que puede ser positiva o negativa. En este caso está indicando que el ejemplo pertenece a las *adquisiciones corporativas*. Las siguientes columnas corresponden a parejas *atributo : valor*. Así 6 : 0,0198 nos dice que el atributo número 6 tiene el valor 0,0198. Los atributos están ordenados y aquellos que no aparecen quiere decir que son valores nulos. A este tipo de representación de los atributos se le denomina representación dispersa**, en contraposición a la representación densa. En un archivo separado encontramos un índice de palabras donde podemos comprobar que el atributo 6 corresponde a la aparición de la palabra *said*. Los conjuntos de entrenamiento y test están separados en dos archivos: *train.dat* y *test.dat*.

* Este conjunto de datos es un subconjunto del Reuters 21578 procesado por T. Joachims que está disponible en: <http://svmlight.joachims.org>

** En inglés, *sparse*.

Tabla 29. Ejemplo de representación de un documento

```
+1 6:0.0198403253586671 15:0.0339873732306071 29:0.0360280968798065 ...
```

Fuente: Problema de "Text Categorisation" formateado por T. Joachims.

A partir de aquí, podemos definir que la representación de un documento d queda de la forma:

$$\phi(d) = (tf(t_1, d), tf(t_2, d), \dots, tf(t_N, d)) \in \mathbb{R}^N$$

donde t_i corresponde al término o palabra i , N al número de términos y $tf(t_i, d)$ a la frecuencia relativa del término t_i en el documento d .

Partiendo de esta definición, el conjunto de entrenamiento lo podemos expresar mediante la matriz siguiente:

$$D = \begin{pmatrix} tf(t_1, d_1) & tf(t_2, d_1) & \cdots & tf(t_N, d_1) \\ tf(t_1, d_2) & tf(t_2, d_2) & \cdots & tf(t_N, d_2) \\ \vdots & \vdots & \ddots & \vdots \\ tf(t_1, d_l) & tf(t_2, d_l) & \cdots & tf(t_N, d_l) \end{pmatrix}$$

donde cada fila corresponde a un documento y cada columna a la aparición de un término en todos los documentos.

A partir de aquí se pueden definir diferentes tipos de funciones kernel. La primera aproximación a una función de kernel es una de las que se usa para tratar conjuntos:

$$\kappa(d_1, d_2) = 2^{|A_1 \cap A_2|}$$

donde A_i corresponde al conjunto formado por las palabras de d_i .

En categorización de textos se suele utilizar la composición de kernels para ir creando kernels más sofisticados. Para ver un ejemplo de ello, vamos a ver cómo se combinaría el kernel anterior con el RBF estudiado anteriormente.

Si partimos de los kernels:

$$\kappa_2(d_1, d_2) = 2^{|A_1 \cap A_2|}$$

$$\kappa_1(x, z) = \exp(-\gamma \|x, z\|)$$

y de que

$$\|\phi_1(x) - \phi_1(z)\|^2 = \kappa_1(x, x) - 2\kappa_1(x, z) + \kappa_1(z, z)$$

el kernel resultado de su combinación es:

$$\kappa(d_1, d_2) = \exp(-\gamma(\kappa_2(d_1, d_1) - 2\kappa_2(d_1, d_2) + \kappa_2(d_2, d_2)))$$

Una segunda aproximación para abordar la creación de kernels para categorización de textos más utilizada que la de conjuntos es la “kernealización” del VSM*. Este kernel, a partir de la definición anterior de la matriz D , se formula como sigue:

$$\kappa(d_1, d_2) = \sum_{i=1}^N tf(t_i, d_1)tf(t_i, d_2)$$

Implementación en Python

El programa 4.9 corresponde a la implementación en Python del clasificador lineal binario basado en los centroides con los tres kernels anteriores. Este programa funcionará para todos los problemas con el formato descrito en el subapartado anterior.

* Del inglés, *Vector Space Model*. El kernel asociado recibe el nombre de *Vector Space Kernel*.

Código 4.9: Lineal con kernel RBF en Python

```

1 from functools import reduce
2 from math import exp
3 from time import clock
4
5 # declaración de kernel
6
7 def kernelSet(a, b):
8     da = dict((k,v) for k, v in (l.split(':') for l in a))
9     db = dict((k,v) for k, v in (l.split(':') for l in b))
10    return 2 ** sum(min(float(da[key]), float(db[key]))
11                   for key in da.keys() if key in db)
12
13 def kernelSetRBF(a, b, gamma=1):
14    return exp(-gamma * (kernelSet(a, a) - 2 *
15                       kernelSet(a, b) + kernelSet(b, b)))
16
17 def kernelVSM(a, b):
18    da = dict((k,v) for k, v in (l.split(':') for l in a))
19    db = dict((k,v) for k, v in (l.split(':') for l in b))
20    return 2 ** sum(float(da[key]) * float(db[key])
21                   for key in da.keys() if key in db)
22
23 kernel = kernelVSM
24
25 # declaración de funciones
26
27 def h(x, tp, tn, cp, cn, b):
28    y = (sum([kernel(x, xi) for xi in tp]) / cp -
29         sum([kernel(x, xi) for xi in tn]) / cn - b)
30    if y > 0:
31        return '+1'
32    else:
33        return '-1'
34
35 # carga del archivo
36 train = list(map(lambda l: (l.strip()).split('\n'),
37                  filter(lambda x: x[0] != '#',
38                          open('train.dat', 'r').readlines())))
39
40 # entrenamiento
41 tp = list(filter(lambda x: x[0] == '1', train))
42 l=list(map(lambda x: x.pop(0), tp))
43 del(l)
44 cp = len(tp)
45
46 tn = list(filter(lambda x: x[0] == '-1', train))
47 l=list(map(lambda x: x.pop(0), tn))
48 del(l)
49 cn = len(tn)
50
51 del(train)
52
53 inici = clock()
54 b = (sum([sum([kernel(xi, xj) for xi in tp])
55          for xj in tp]) / (cp ** 2) -
56       sum([sum([kernel(xi, xj) for xi in tn])
57           for xj in tn]) / (cn ** 2)) / 2
58 print("b", b, clock() - inici, "s")
59
60 test = list(map(lambda l: (l.strip()).split('\n'),
61                 filter(lambda x: x[0] != '#',
62                         open('test.dat', 'r').readlines())))
63 clasesTest=list(map(lambda x: x.pop(0), test))
64
65 inici = clock()
66 predicciones = [h(x, tp, tn, cp, cn, b) for x in test]
67 print("h", clock() - inici, "s")
68
69 # número de correctos
70 print('Prec.: ', len(list(filter(lambda x: x[0] == x[1],
71                                 zip(*[predicciones, clasesTest]))))
72       / len(test) * 100, '%')

```

La tabla 30 muestra los resultados al aplicar el programa 4.9 al conjunto de datos anterior con los tres tipos de kernel* y el tiempo de cómputo del parámetro b y las predicciones h . El programa se ha ejecutado en un portátil Mac con OSX de 1,4 GHz y 2 Gb de memoria.

Tabla 30. Tabla de resultados para el primer problema de text categorización

kernel	precisión	tiempo b	tiempo h
conjunto	84 %	452 s	246 s
conjunto + RBF	51,2 %	1805 s	998 s
VSM	88 %	440 s	244 s

* Fijaos que para seleccionar el kernel hay una línea justo debajo de sus definiciones del estilo:
kernel = kernelSet

Repositorio

Los archivos de datos y el programa están disponibles en el repositorio de la asignatura.

Supongamos que queremos diseñar un clasificador para distinguir páginas web de dos temáticas diferentes. Para este problema vamos a utilizar como temas dos de las etiquetas de la ontología ODP: *Arts/Music/Bands_and_Artists/U* y *Arts/Celebrities/S*.

En este caso, disponemos de un único archivo procesado, en el que está guardado el número de veces que aparece cada palabra en el documento. El formato es el mismo que para el problema anterior.

Implementación en Python

El programa 4.10 corresponde a la implementación en Python del clasificador lineal binario basado en los centroides con el kernel VSM y uno que lo combina con el RBF. En este caso, hemos modificado el kernel VSM, por razones de eficiencia, con respecto al que aparece en el código 4.9.

Código 4.10: Lineal con kernel RBF en Python

```

1 from functools import reduce
2 from math import exp
3 from time import clock
4
5 # declaración de kernel
6
7 def kernelVSM(a, b):
8     la = [(int(k),int(v)) for k, v in (l.split(':') for l in a)]
9     lb = [(int(k),int(v)) for k, v in (l.split(':') for l in b)]
10    s = 0
11    i = 0
12    j = 0
13    while (i < len(la)) and (j < len(lb)):
14        if la[i][0] == lb[j][0]:
15            s += la[i][1] * lb[j][1]
16            i += 1
17            j += 1
18        elif la[i][0] < lb[j][0]:
19            i += 1
20        else:
21            j += 1
22    return s
23
24 def kernelVSMRBF(a, b, gamma=1):
25     return exp(-gamma * (kernelVSM(a, a) - 2 *
26                          kernelVSM(a, b) + kernelVSM(b, b)))
27
28 kernel = kernelVSMRBF

```

```

29
30 # declaración de funciones
31
32 def h(x, tp, tn, cp, cn, b):
33     y = (sum([kernel(x, xi) for xi in tp]) / cp -
34          sum([kernel(x, xi) for xi in tn]) / cn - b)
35     if y > 0:
36         return '+1'
37     else:
38         return '-1'
39
40 # carga del archivo
41 data = list(map(lambda l: (l.strip()).split(' '),
42                filter(lambda x: x[0] != '#',
43                      open('vectors.dat', 'r').readlines())))
44
45 # construcción del train: 2 de cada 3
46 train = list(map(lambda x: x[1],
47                 filter(lambda v: v[0] % 3 != 0, enumerate(data))))
48
49 # construcción del test: 1 de cada 3
50 test = list(map(lambda x: x[1],
51                filter(lambda v: v[0] % 3 == 0, enumerate(data))))
52 clasesTest = list(map(lambda x: x.pop(0), test))
53
54 del(data)
55
56 # entrenamiento
57 tp = list(filter(lambda x: x[0] == '+1', train))
58 l=list(map(lambda x: x.pop(0), tp))
59 del(l)
60 cp = len(tp)
61
62 tn = list(filter(lambda x: x[0] == '-1', train))
63 l=list(map(lambda x: x.pop(0), tn))
64 del(l)
65 cn = len(tn)
66
67 del(train)
68
69 inici = clock()
70 b = (sum([sum([kernel(xi, xj) for xi in tp])
71         for xj in tp]) / float(cp ** 2) -
72      sum([sum([kernel(xi, xj) for xi in tn])
73         for xj in tn]) / float(cn ** 2)) / 2.0
74 print("b", b, clock() - inici, "s")
75
76 inici = clock()
77 predicciones = [h(x, tp, tn, cp, cn, b) for x in test]
78 print("h", clock() - inici, "s")
79
80 # número de correctos
81 print('Prec.:', len(list(filter(lambda x: x[0] == x[1],
82                               zip(*[predicciones, clasesTest]))))
83       / len(test) * 100, '%')

```

La tabla 31 muestra los resultados al aplicar el programa 4.10 al conjunto de datos anterior con los dos tipos de kernel* y el tiempo de cómputo del parámetro b y las predicciones. El programa se ha ejecutado en un portátil Mac con OSX de 1,4 GHz y 2 Gb de memoria.

Tabla 31. Tabla de resultados para el primer problema de text categorización

kernel	precisión	tiempo b	tiempo h
VSM	61,8%	55 s	48 s
VSM + RBF	52,7%	164 s	145 s

* Fijaos que para seleccionar el kernel hay una línea justo debajo de sus definiciones del estilo:
kernel = kernelVSM

Repositorio

Los archivos de datos y el programa están disponibles en el repositorio de la asignatura.

Ampliaciones de kernels para texto

Cuando se trabaja en clasificación de textos se suelen preprocesar los documentos. Uno de los procesos más comunes es eliminar las palabras sin contenido semántico como: preposiciones, artículos... Existen listas de estas palabras que nos ayudan a realizar esta tarea.

El segundo proceso que se suele utilizar es eliminar del diccionario aquellas palabras que no aparecen más de dos o tres veces en el conjunto de todos los documentos del corpus.

Con esto conseguimos reducir bastante el espacio de atributos y con ello mejorar la eficiencia computacional de los algoritmos esperando no degradar la precisión de los mismos.

En el segundo ejemplo de clasificación de textos que hemos visto no hemos tenido en cuenta el tratamiento del tamaño de los documentos. Esto va a sesgar el comportamiento de las predicciones hacia los documentos más grandes, ya que contienen más palabras y tienen más probabilidades de enlazarse entre ellos. Otra forma de tratar este tema, aparte de codificar frecuencias como en el primer problema tratado, es normalizar el kernel resultante. Esto se realiza como etapa final de la combinación de kernels a partir de la formulación siguiente:

$$\hat{k}(x,y) = \left\langle \frac{\phi(x)}{\|\phi(x)\|}, \frac{\phi(y)}{\|\phi(y)\|} \right\rangle = \frac{k(x,y)}{\sqrt{k(x,x)k(y,y)}}$$

Para finalizar este subapartado vamos a describir una función kernel basada en uno de los algoritmos más utilizados de la recuperación de la información, el *tf-idf*. La idea que persigue este algoritmo es doble: por un lado aporta una ponderación de los términos o palabras en función de su importancia; y por otro, aporta medidas de relación entre los términos, que nos ayudan a relacionar documentos que no comparten exactamente los mismos términos pero sí sinónimos.

Vamos a empezar retomando la definición de función kernel del VSM:

$$k(d_1, d_2) = \langle \phi(d_1), \phi(d_2) \rangle = \sum_{i=1}^N tf(t_i, d_1) tf(t_i, d_2)$$

Si queremos ponderar los términos con un peso, necesitamos una función $w(t)$ que nos lo proporcione. Existen diferentes alternativas, como pueden ser la información mutua o medidas basadas en la entropía. El *tf-idf* utiliza la función:

$$w(t) = \ln \left(\frac{l}{df(t)} \right)$$

Repositorio

Tenéis disponible en el repositorio de la asignatura listas de palabras sin contenido semántico para inglés, castellano y catalán.

Diccionario

Se conoce como diccionario el archivo índice de palabras que aparecen en los documentos del conjunto de datos.

Ved también

Para saber más sobre la información mutua podéis ver el subapartado 3.1.3. Para saber más sobre medidas basadas en la entropía podéis ver el subapartado 4.4.1.

donde l corresponde al número total de documentos y $df(t)$ al número de documentos que contienen el término t .

Si generamos un kernel a partir de esta función obtendremos:

$$\tilde{\kappa}(d_1, d_2) = \sum_t w(t)^2 tf(t, d_1) tf(t, d_2)$$

Si queremos añadir semejanzas semánticas entre términos, existen diversas formas de hacerlo. Una es explotar una ontología de libre distribución muy utilizada en el campo del procesamiento del lenguaje natural que se denomina *WordNet**. Esta ontología contiene las diferentes palabras de la lengua organizadas en una jerarquía en la que cada nodo representa un concepto y contiene todas las palabras sinónimas que lo representan. Una palabra está en los diferentes nodos a los que pertenece en función de sus diferentes significados. Los nodos están interconectados a través de la relación de hiperonimia/hiponimia**. Partiendo de esta ontología, la semejanza entre palabras tiene que ver con la inversa del camino más corto para llegar de una palabra a la otra.

*<http://wordnet.princeton.edu>

** A modo de ejemplo: felino es un hiperónimo de gato y gato es hipónimo de felino.

En recuperación de información se juega con la matriz D para obtener un resultado parecido. Recordemos que la matriz de kernel se puede generar a partir de:

$$K = D \times D'$$

y que:

$$D = \begin{pmatrix} tf(t_1, d_1) & tf(t_2, d_1) & \cdots & tf(t_N, d_1) \\ tf(t_1, d_2) & tf(t_2, d_2) & \cdots & tf(t_N, d_2) \\ \vdots & \vdots & \ddots & \vdots \\ tf(t_1, d_i) & tf(t_2, d_i) & \cdots & tf(t_N, d_i) \end{pmatrix}$$

Definimos que dos términos están relacionados entre sí, si aparecen frecuentemente juntos en los mismos documentos. Esto lo podemos conseguir a partir de:

$$D' \times D$$

Fijaos en que la matriz $D'D$ tendrá un valor diferente de cero en la posición (i, j) si, y sólo si, existe algún documento en el corpus en el que coocurren los términos t_i y t_j .

A partir de esto, podemos definir un nuevo kernel:

$$\tilde{\kappa}(d_1, d_2) = \phi(d_1)D'D\phi(d_2)'$$

donde

$$(D'D)_{i,j} = \sum_d tf(i,d)tf(j,d)$$

Al uso de estas semejanzas semánticas se le conoce como *GVSM*, donde la *G* viene de generalizado.

4.5.4. Máquinas de vectores de soporte

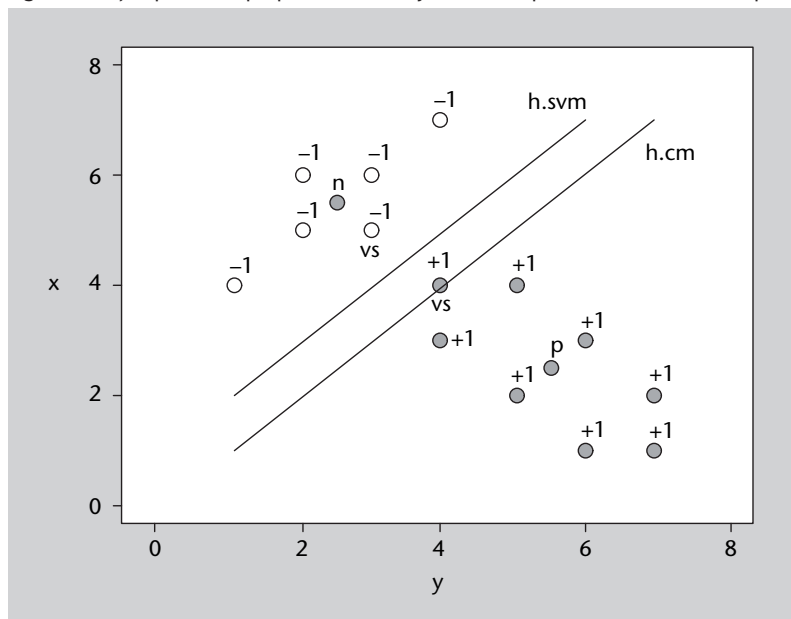
Supongamos que queremos diseñar un clasificador para una agencia de noticias que sepa distinguir los documentos relacionados con *adquisiciones corporativas*.

Las máquinas de vectores de soporte* (SVMs) son un algoritmo que mejora el clasificador lineal buscando un mejor hiperplano que el que se genera con el clasificador lineal.

* En inglés, *support vector machines*: N. Cristianini; J. Shawe-Taylor (2000). *An Introduction to Support Vector Machines (SVMs)*. Cambridge University Press.

La figura 33 ilustra este concepto: los puntos marcados con +1 corresponden a los ejemplos positivos, los -1 a los negativos, *p* al centroide de los positivos, *n* al de los negativos y *h.cm* al hiperplano generado por los centroides (está ubicado a medio camino de los dos centroides).

Figura 33. Ejemplos de hiperplanos lineales y de las máquinas de vectores de soporte



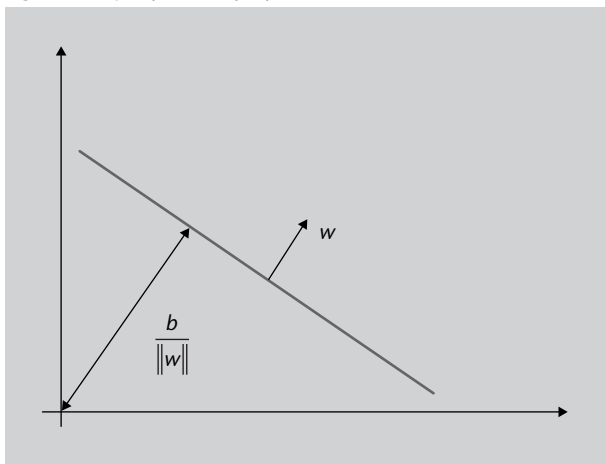
Las máquinas de vectores de soporte son un algoritmo de optimización (como los descritos en el apartado 5) que escoge el hiperplano con margen máximo de entre todos los posibles hiperplanos que separan los ejemplos positivos de los negativos (el que tiene la misma distancia a los ejemplos positivos que a los negativos). Describe el hiperplano a partir de los llamados *vectores de soporte*. Estos suelen ser los puntos más cercanos al hiperplano y los que lo definen. En la figura 33, $h.svm$ corresponde al hiperplano de margen máximo que encontrarían las SVM para este conjunto y los vectores de soporte están marcados con vs .

Interpretación geométrica

Este algoritmo combina la maximización del margen, como el AdaBoost, con el uso de los kernels descritos en el subapartado anterior y la posibilidad de flexibilizar el margen. Este subapartado pretende exponer estos conceptos de manera visual.

Muchos métodos de aprendizaje se basan en la construcción de *hiperplanos*. Un hiperplano es una figura geométrica que tiene una dimensión menos que el espacio en el que está ubicado. Ejemplos de hiperplanos son líneas en \mathbb{R}^2 o planos en \mathbb{R}^3 . La figura 34 nos muestra un ejemplo de hiperplano en \mathbb{R}^2 . Un hiperplano queda definido por un vector de pesos (w) y un umbral (b).

Figura 34. Ejemplo de hiperplano



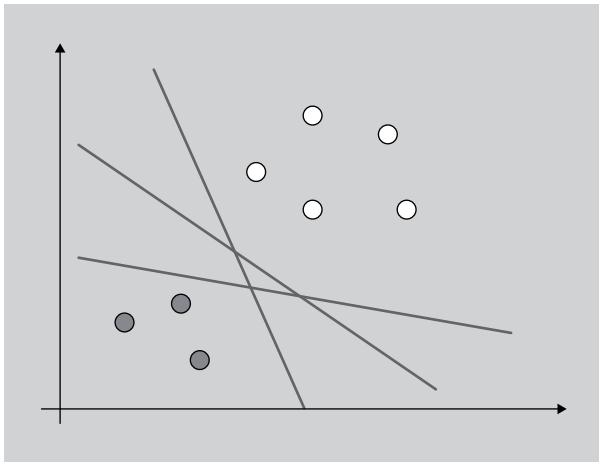
Consideremos ahora un ejemplo en \mathbb{R}^2 . Tenemos que construir un clasificador para separar dos tipos de puntos. La figura 35 muestra tres hiperplanos válidos (en este caso líneas) que nos separarían los dos conjuntos de puntos.

Podríamos encontrar infinitos hiperplanos para este problema de clasificación. Una vez hemos escogido un hiperplano para clasificar un conjunto de puntos, podemos crear una regla de clasificación a partir de él. Esta regla queda definida por la ecuación:

$$h(x) = \text{signo} \left(b + \sum_{i=1}^N w_i \cdot x_i \right)$$

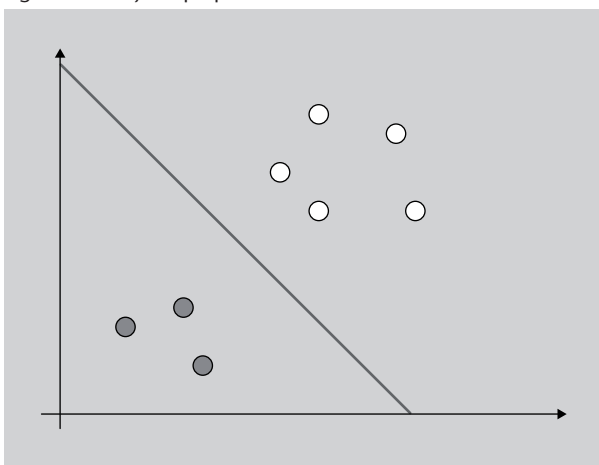
donde N es el número de atributos y el vector $w = \{w_1, \dots, w_N\}$ y b definen al hiperplano y $x = \{x_1, \dots, x_N\}$ al ejemplo a clasificar.

Figura 35. Ejemplo de hiperplanos posibles



Pero ¿cuál de estos infinitos hiperplanos nos servirá mejor como clasificador? Intuitivamente se puede decir que la mejor recta para dividir estos conjuntos sería algo parecido al de la figura 36.

Figura 36. Mejor hiperplano

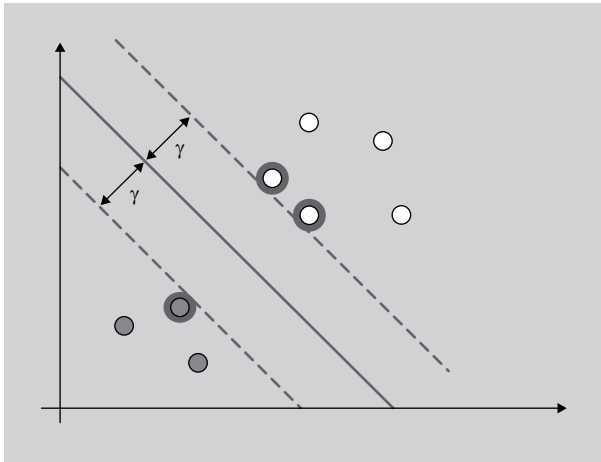


¿Por qué? Porque es la que deja más distancia a todos los puntos; es la recta que pasa más alejada de los puntos de las dos clases. A este concepto se le denomina margen. La figura 37 nos muestra la representación geométrica de este concepto; donde γ corresponde al margen y los puntos marcados en azul a los vectores de soporte. Los vectores de soporte son aquellos puntos que están rozando el margen.

Las máquinas de vectores de soporte son un método de aprendizaje que se basa en la maximización del margen. ¿Cómo lo hace? La estrategia del método consiste en buscar los vectores de soporte. Este mecanismo de búsqueda parte de un espacio de hipótesis de funciones lineales en un espacio de atributos altamente multidimensional. En él se entrena con un algoritmo de la teoría de

la optimización derivado de la teoría del aprendizaje estadístico. El subapartado siguiente describe este proceso formalmente. Este apartado está orientado a ver sus aplicaciones prácticas.

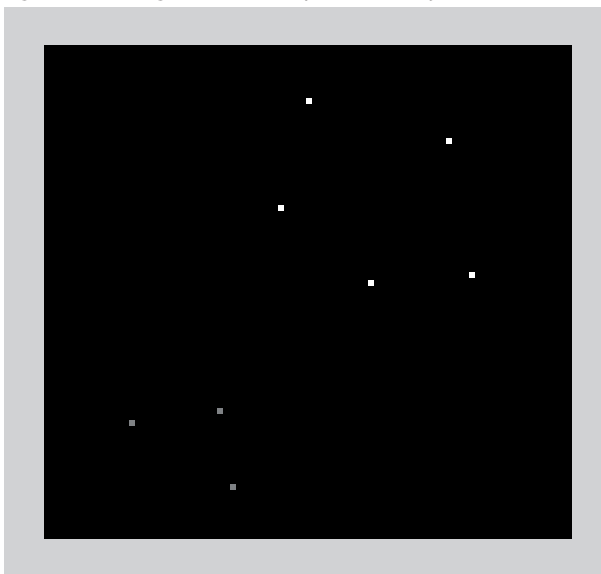
Figura 37. Margen



A partir de este momento vamos a utilizar el software *svmtoy* para ilustrar todos los conceptos. Este programa aplica las SVMs con diferentes parámetros a un conjunto con dos atributos y representa gráficamente el resultado de la clasificación.

Vamos a empezar con el caso anterior. Introduciremos una distribución parecida a la de la figura 37 (como muestra la figura 38). Los puntos se introducen con el botón del ratón. Hay que pulsar el botón change para cambiar de clase. Una vez introducidos los puntos, hay que especificar los parámetros. Para entrenar este caso especificaremos: $-t 0 -c 100$. Es decir, un kernel lineal ($-t 0$) y un valor de 100 a la rigidez (inversa de la flexibilidad) del margen. Este último concepto lo veremos más adelante.

Figura 38. Configuración inicial para el svmtoy



svmtoy

svmtoy es un programa incluido en la *libsvm*: Chih-Chung Chang; Chih-Jen Lin (2001). *LIBSVM: a library for support vector machines*. <http://www.csie.ntu.edu.tw/~cjlin/libsvm/>

Repositorio

El software *svmtoy* está disponible en el repositorio de la asignatura.

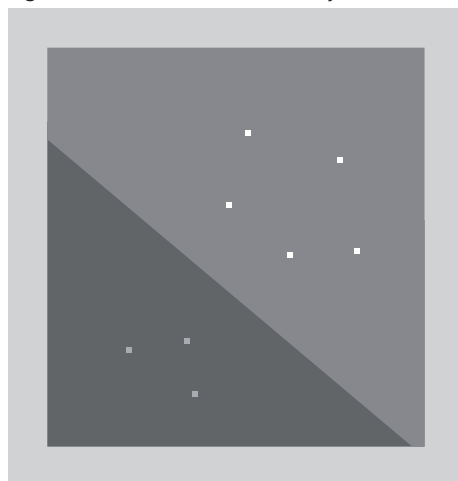
Kernel lineal

Un kernel lineal es del estilo $u' \times v$ donde u' es nuestro vector de atributos y v es otro vector.

Una vez establecidos los parámetros, pulsamos en *run* y nos aparecerá una pantalla como la que muestra la figura 39. Podemos observar cómo ha separa-

do bien los dos conjuntos de puntos. Se puede intuir un margen claro y que los vectores de soporte son dos puntos amarillos y uno azul (los más cercanos al hiperplano).

Figura 39. Kernel lineal con svmtoy

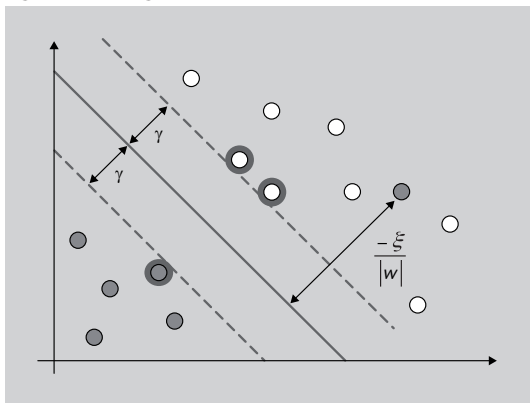


Cuando dado un problema de puntos podemos encontrar un hiperplano lineal (en el caso de \mathbb{R}^2 , una recta) que divide perfectamente a los dos conjuntos de puntos, tenemos un problema al que denominamos *linealmente separable*. El caso anterior cumple esta propiedad. Puede ocurrir (más bien suele ocurrir) que los conjuntos de datos no sean tan claramente separables. En estos casos nos solemos encontrar con dos tipologías de problemas:

- El primer problema aparece cuando hay puntos sueltos de las dos clases en la zona del margen, o puntos de una clase en la zona de la otra. El caso mostrado en el figura 40 no es linealmente separable. El punto rojo que está entre los verdes normalmente es un error o un *outlier*. La solución pasa por generar un *margen flexible**. El margen flexible admite errores en la separación a cambio de aumentar el margen. Estos errores están ponderados por la expresión $-\xi/|w|$. El uso de márgenes flexibles normalmente aumenta la generalización y disminuye la varianza del modelo. El parámetro $-c$ del *svmtoy* codifica la rigidez del margen. Cuanto menor sea el valor de este parámetro más flexible es el margen. O dicho de otro modo, la flexibilidad del margen es inversamente proporcional al valor del parámetro c . Este parámetro se escoge normalmente como una potencia de diez ($10^{\pm x}$).

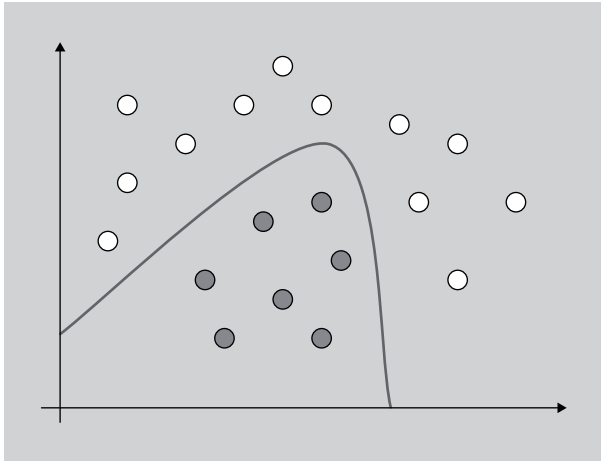
* En inglés, *soft margin*.

Figura 40. Margen flexible



- La segunda tipología de problemas aparece cuando los puntos no tienen una distribución que se pueda separar linealmente. Es decir, la estructura del problema sigue otro tipo de distribuciones. Para un problema como el que muestra la figura 41 nos iría bien una curva en lugar de una recta para separar los conjuntos de puntos. Esto se logra cambiando de kernel. Un kernel es una función de proyección entre espacios. Este mecanismo nos permite aumentar la complejidad del modelo, y por tanto la varianza.

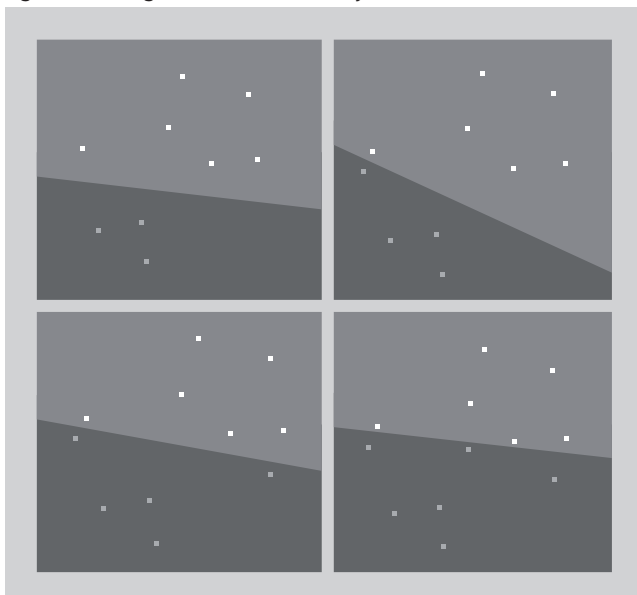
Figura 41. Conjunto linealmente no separable



De cara a construir modelos con las SVM, es muy interesante estudiar el comportamiento de estos dos parámetros: los kernels y los márgenes flexibles.

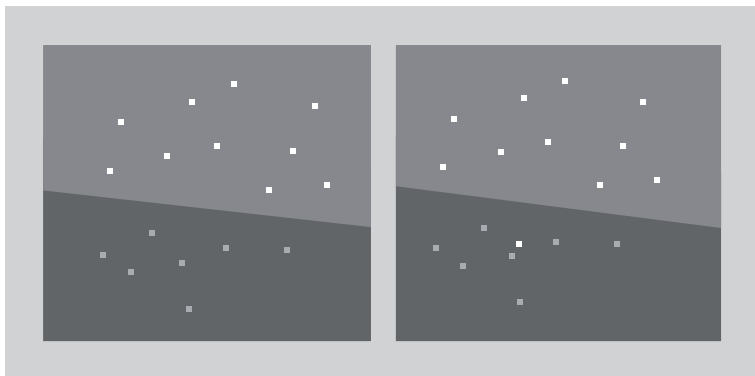
Cojamos ahora el conjunto de la figura 39 y empecemos a añadir puntos. Fijaos en cómo van cambiando el hiperplano y el margen en las imágenes de la figura 42 (tendremos que aumentar la rigidez del hiperplano escogiendo un valor de 10.000 para el parámetro c). En la última imagen de la figura 42, la distancia de los vectores de soporte al hiperplano es muy inferior a la de la primera.

Figura 42. Margen flexible con svmtoy



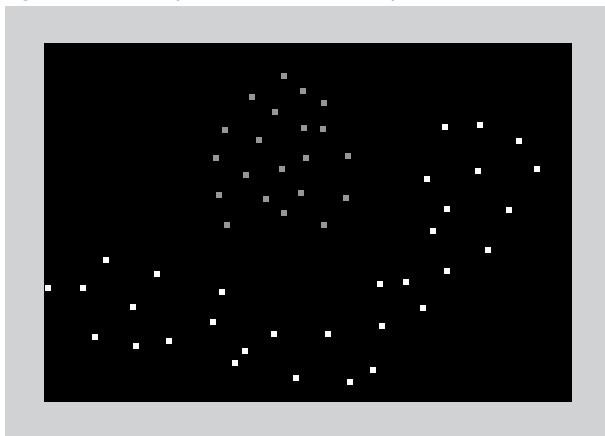
Fijaos ahora en la figura 43. En la segunda imagen tenemos un punto amarillo en la zona de los azules. Gracias a haber creado un margen flexible, lo hemos dejado mal clasificado, pero tenemos un margen igual que el anterior. Las ventajas que obtenemos tienen que ver con el error de generalización. Seguramente este caso lo encontraríamos como un error en el conjunto de entrenamiento.

Figura 43. Margen flexible con svmtoy



En este punto, vamos a ver ejemplos de la aplicación de kernels. Para empezar, fijaos en el conjunto de puntos de la figura 44. ¿Qué hiperplano puede separar estos conjuntos de puntos? Un hiperplano lineal no los separaría. La aplicación de márgenes flexibles dejaría demasiados puntos mal clasificados, y aunque fueran pocos, una línea recta no es un modelo correcto para este problema. La solución para este caso pasa por utilizar un kernel para realizar una proyección a otro espacio.

Figura 44. Puntos para kernels con svmtoy



En la figura 45 tenéis dos aproximaciones con diferentes kernels polinómicos para separar estos conjuntos de puntos y, en la figura 46 dos, con kernels radiales. Sus parámetros exactos son respectivamente:

- polinómico de grado dos $(-t\ 1\ -d\ 2\ -c\ 10000)$
- polinómico de grado cuatro $(-t\ 1\ -d\ 4\ -c\ 10000)$

Kernel polinómico

Un kernel polinómico es del estilo: $(\gamma \times u' \times v + coef0)^{grado}$ donde u' es nuestro vector de atributos, v es otro vector y los demás son valores reales. Se le pueden pasar como parámetros: $-t\ 1\ -d\ grado\ -g\ \gamma\ -r\ coef0$.

- radial con gamma 50 (-t 2 -g 50 -c 10000)
- radial con gamma 1000 (-t 2 -g 1000 -c 10000)

Kernel radial

Un kernel radial del estilo: $e^{-\gamma \times |u' \times v|^2}$ donde u' es nuestro vector de atributos, v es otro vector y los demás son valores reales. Se le pueden pasar como parámetros: -t 2 -g γ .

Figura 45. Kernels polinómicos con svmtoy

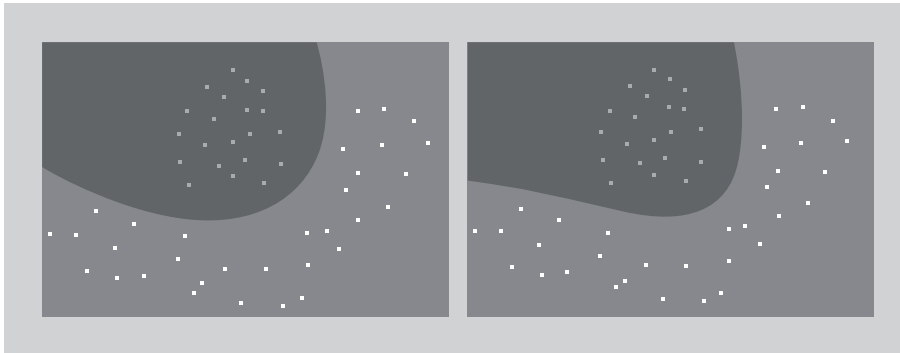
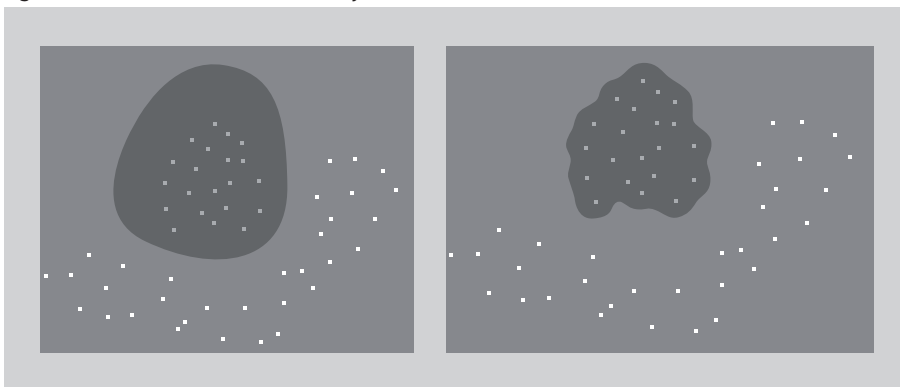


Figura 46. Kernels radiales con svmtoy



La figura 47 muestra un tercer ejemplo a trabajar. En este caso los kernels polinómicos no nos servirán. Separaremos los conjuntos de puntos utilizando kernels radiales. Fijaos en el efecto del parámetro gamma que muestra la figura 48:

- radial con gamma 1000 (-t 2 -g 1000 -c 10000)
- radial con gamma 10 (-t 2 -g 10 -c 10000)

Figura 47. Puntos para kernels radiales con svmtoy

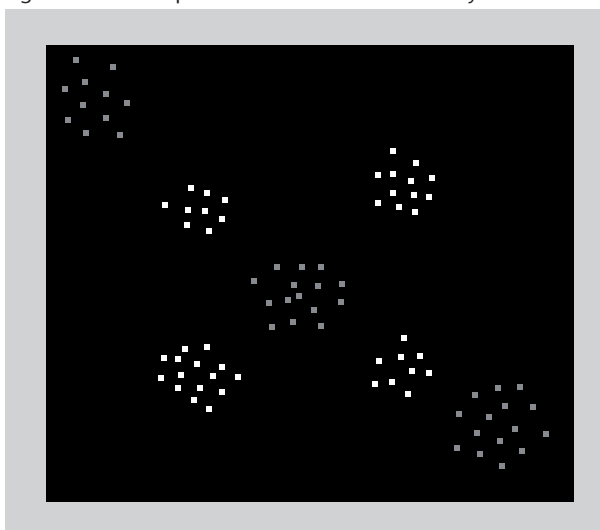
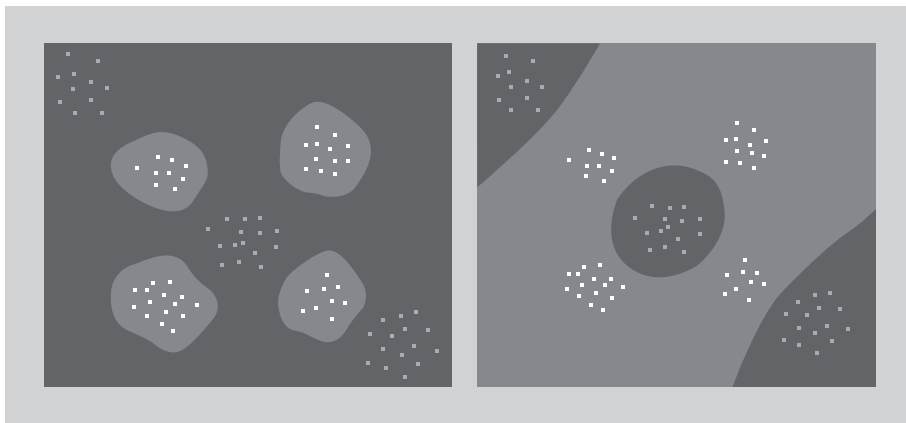


Figura 48. Kernels radiales con svmtoy



Formalización

Las máquinas de vectores de soporte se basan en el principio de la minimización del riesgo estructural* de la teoría del aprendizaje estadístico. En su forma básica, aprenden un hiperplano lineal de margen máximo que separa un conjunto de ejemplos positivos de uno de ejemplos negativos. Se ha demostrado que este sesgo de aprendizaje tiene muy buenas propiedades teniendo en cuenta la generalización de los clasificadores inducidos. El clasificador lineal queda definido por dos elementos: un vector de pesos w (con un componente por cada atributo) y un umbral b , que está relacionado con la distancia del hiperplano al origen: $h(x) = \langle x, w \rangle + b$ (figuras 34 y 36). La regla de clasificación $f(x)$ asigna +1 a un ejemplo x cuando $h(x) \geq 0$ y -1 cuando es menor. Los ejemplos positivos y negativos más cercanos al hiperplano (w, b) son los llamados vectores de soporte.

* En inglés, *structural risk minimisation*. Descrita en: V. N. Vapnik (1998). *Statistical Learning Theory*. John Wiley.

Aprender el hiperplano de margen máximo (w, b) se puede abordar como un problema de *optimización cuadrática convexa** con una única solución. Esto consiste, en su *formulación primal* en:

* En inglés, *convex quadratic optimisation*. Este es un algoritmo de optimización matemática como los que se describen en el subapartado 5.2.

$$\text{minimizar : } \|w\|$$

$$\text{sujeto a : } \gamma_i(\langle w, x_i \rangle + b) \geq 1, \forall 1 \leq i \leq N$$

donde N es el número de ejemplos de entrenamiento y las γ_i corresponden a las etiquetas.

Lo anterior se suele transformar a su *formulación dual* en que su solución se puede expresar como una combinación lineal de ejemplos de entrenamiento (vistos como vectores):

$$\text{maximizar : } \sum_{i=1}^N \alpha_i - \frac{1}{2} \sum_{i,j=1}^N \gamma_i \gamma_j \alpha_i \alpha_j \langle x_i, x_j \rangle$$

$$\text{sujeto a : } \sum_{i=1}^N \gamma_i \alpha_i = 0, \alpha_i \geq 0, \forall 1 \leq i \leq N$$

donde los α_i son valores escalares no negativos determinados al resolver el problema de optimización.

De esta formulación, el vector ortogonal al hiperplano queda definido como:

$$h(x) = \langle w, x \rangle + b = b + \sum_{i=1}^N \gamma_i \alpha_i \langle x_i, x \rangle$$

donde $\alpha_i \neq 0$ para todos los ejemplos de entrenamiento que están en el margen (vectores de soporte) y $\alpha_i = 0$ para los demás. $f(x) = +1$ cuando $h(x) \geq 0$ y -1 cuando es menor define el clasificador en su forma dual.

Aunque las máquinas de vectores de soporte son lineales en su forma básica, pueden ser transformadas en una *forma dual*, en la que los ejemplos de entrenamiento sólo aparecen dentro de productos escalares, permitiendo el uso de *funciones núcleo** para producir clasificadores no lineales. Estas funciones trabajan de manera muy eficiente en espacios de atributos de dimensionalidad muy elevada, donde los nuevos atributos se pueden expresar como combinaciones de muchos atributos básicos**.

Si suponemos que tenemos una transformación no lineal ϕ de \mathbb{R}^D a algún espacio de Hilbert \mathfrak{H} , podemos definir un producto escalar con una función núcleo tipo $\kappa(x, y) = \langle \phi(x), \phi(y) \rangle$ y obtener una formulación dual:

$$\text{maximizar : } \sum_{i=1}^N \alpha_i - \frac{1}{2} \sum_{i,j=1}^N \gamma_i \gamma_j \alpha_i \alpha_j \kappa(x_i, x_j)$$

$$\text{sujeto a : } \sum_{i=1}^N \gamma_i \alpha_i = 0, \alpha_i \geq 0, \forall 1 \leq i \leq N$$

y obtener el vector ortogonal al hiperplano:

$$h(x) = \langle w, \phi(x) \rangle + b = b + \sum_{i=1}^N \gamma_i \alpha_i \kappa(x_i, x)$$

Para algunos conjuntos de entrenamiento no es deseable obtener un hiperplano perfecto, es preferible permitir algunos errores en el conjunto de entrenamiento para obtener “mejores” hiperplanos (figuras 40 y 43). Esto se consigue con una variante del problema de optimización, llamada *margen flexible*, donde la contribución a la función objetivo de la maximización del margen y los errores en el conjunto de entrenamiento se pueden balancear a través de

* En inglés, *kernel functions*.

** Véase N. Cristianini; J. Shawe-Taylor (2000). *An Introduction to Support Vector Machines*. Cambridge University Press.

un parámetro normalmente llamado C . Este parámetro afecta a las variables ξ_i en la función (figura 40). Este problema queda formulado como:

$$\text{minimizar : } \frac{1}{2} \langle w, w \rangle + C \sum_{i=1}^N \xi_i$$

$$\text{sujeto a : } y_i(\langle w, x_i \rangle + b) \geq 1 - \xi_i, \xi_i \geq 0, \forall 1 \leq i \leq N$$

Análisis del método

Las máquinas de vectores de soporte fueron dadas a conocer en el año 92 por Boser, Guyon y Vapnik, pero no empezaron a ganar popularidad en el área del aprendizaje automático hasta finales de los noventa.

Actualmente, son uno de los algoritmos de aprendizaje supervisados más utilizados en la investigación. Han demostrado ser muy útiles y se han aplicado a una gran variedad de problemas relacionados con el reconocimiento de patrones en bioinformática, reconocimiento de imágenes, categorización de textos, entre otros. Se ha probado que su sesgo de aprendizaje es muy robusto y que se obtienen muy buenos resultados en muchos problemas. Se están dedicando también muchos esfuerzos en la comunidad investigadora internacional en las posibilidades que ha abierto el uso de kernels.

Cuando aplicamos este algoritmo, hay una cuestión que tenemos que tener en cuenta: estamos combinando el uso de kernels con un algoritmo de optimización. Si aplicamos un kernel que no encaja bien con los datos, puede pasar que el algoritmo tarde en converger o que dé una mala solución. El tiempo que tarda en construir el modelo y el número de vectores que genera nos dan pistas de la bondad de la solución en la práctica y de la separabilidad del conjunto de datos con el kernel dado. El número de vectores de soporte es un número inferior al del conjunto de entrenamiento. La diferencia entre los dos conjuntos suele ser directamente proporcional a la bondad de la solución y del hiperplano. El tiempo de ejecución del proceso de aprendizaje suele ser inversamente proporcional a la bondad.

Tanto las SVM como los métodos descritos en este subapartado son binarios. Es decir, sólo permiten separar entre dos clases. Cuando nos encontramos delante de un problema que tiene más de dos clases, tenemos que crear un marco para gestionar las predicciones de todas las clases a partir de los clasificadores binarios. Existen diversos mecanismos de binarización para abordar esta cuestión, aquí describiremos la técnica del *uno contra todos**, que es una de las más utilizadas en la bibliografía.

Referencia bibliográfica

B. Boser; I. Guyon; V. Vapnik (1992). A training Algorithm for Optimal Margin Classifiers. En las actas del *Workshop on Computational Learning Theory*. COLT

* En inglés, *one vs all*.

Esta técnica consiste en entrenar un clasificador por clase, cogiendo como ejemplos positivos los ejemplos de la clase y como negativos los ejemplos de todas las otras clases. El resultado de la clasificación es un peso (número real por clase). Para el caso monoetiqueta, se coge como predicción la clase con el peso mayor. Para el caso multietiqueta, se cogen como predicciones todas las clases con pesos positivos.

Lecturas complementarias

Para profundizar en el tema de la binarización podéis consultar:

E. Allwein; R. E. Schapire; Y. Singer (2000). "Reducing Multiclass to Binary: A Unifying Approach for Margin Classifiers". *Journal of Machine Learning Research* (núm. 1, págs. 113-141).

S. Har-Peled; D. Roth; D. Zimak (2002). "Constraint Classification for Multiclass Classification and Ranking". *Actas del Workshop on Neural Information Processing Systems*.

Uso en Python

Las dos implementaciones de SVMs de libre distribución más utilizadas son la *svmlight* de Thorsten Joachims* y la *libsvm* de Chih-Chung Chang y Chih-Jen Lin**.

SVM^{light} es una implementación libre de las SVM disponible para la mayoría de plataformas, que soporta problemas de regresión y clasificación binaria; la creación de márgenes flexibles; aprendizaje inductivo y transductivo; diferentes tipos de kernels predefinidos y permite que el usuario defina los suyos propios. La web dispone, además, de una extensión *SVM^{struct}* preparada para crear ampliaciones de kernels complejos y otras aplicaciones. Como ejemplo de la extensión se puede encontrar el tratamiento multiclase y enlaces con *Python*.

libsvm es una implementación de libre distribución que soporta menos funciones, pero que está disponible para más sistemas (como dll, jar...). Una de las funciones que sí que tiene y que no tiene la anterior es la posibilidad de trabajar directamente con la matriz kernel. Contiene el *svmtoy* que hemos utilizado anteriormente para dar la interpretación geométrica de las SVMs. En la web dispone de algunas utilidades, enlaces con otros lenguajes como *Python* y un *script* para realizar el tratamiento multiclase.

Ambas distribuciones permiten trabajar a nivel de librería y a nivel de sistema operativo. A modo de ejemplo, el código 4.11 muestra el uso de la *svmlight* utilizando llamadas al sistema operativo y el código 4.12 muestra el uso de la *libsvm* a través de su librería. El primero se ha ejecutado en un Mac OSX. En otras plataformas puede verse alterado el código.

* <http://svmlight.joachims.org>
T. Joachims (1999). *Making large-Scale SVM Learning Practical. Advances in Kernel Methods - Support Vector Learning*. En: B. Schölkopf; C. Burges; A. Smola (ed.). MIT-Press.

** <http://www.csie.ntu.edu.tw/~cjlin/libsvm>
Chih-Chung Chang; Chih-Jen Lin (2001). *LIBSVM: a library for support vector machines*.

Aprendizaje transductivo

El aprendizaje transductivo utiliza ejemplos con y sin etiquetas en el proceso de aprendizaje.

Código 4.11: svmlight vía llamadas al sistema operativo

```

1 import os
2
3 parametros = '-t_1_d_1'
4 train = 'train.dat'
5 test = 'test.dat'
6 model = 'model.svmlight.txt'
7 preds = 'predicciones.txt'
8 outs = 'output.txt'
9
10 os.system('./svm_learn_' + parametros + '_' + train + '_' + model)
11
12 os.system('./svm_classify_' + test + '_' + model + '_' + preds +
13         '_' + outs)
14
15 l = list(map(lambda l: l.strip(),
16             open(outs, 'r').readlines()))
17 print(l[0])
18 print(l[2])
19 print(l[3])

```

Código 4.12: libsvm vía librería

```

1 from libsvm.svutil import *
2 from time import clock
3
4 train = list(map(lambda l: (l.strip()).split('_'),
5                 filter(lambda x: x[0] != '#',
6                       open('train.dat', 'r').readlines()))))
7 y = list(map(lambda a: int(a.pop(0)), train))
8 x = [dict([(int(k), float(v))
9           for k, v in (l.split(':') for l in t)])
10      for t in train]
11 del(train)
12
13 inici = clock()
14 model = svm_train(y, x, '-c_4-t_0')
15 print("learn", clock() - inici, "s")
16 del(y, x)
17 svm_save_model('model.libsvm.txt', model)
18
19 test = list(map(lambda l: (l.strip()).split('_'),
20               filter(lambda x: x[0] != '#',
21                     open('test.dat', 'r').readlines()))))
22 yt = list(map(lambda a: int(a.pop(0)), test))
23 xt = [dict([(int(k), float(v))
24           for k, v in (l.split(':') for l in t)])
25      for t in test]
26 del(test)
27
28 p_label, p_acc, p_val = svm_predict(yt, xt, model)

```

Si aplicamos las *svmlight* a través del código 4.11 al corpus de la detección de documentos de *adquisiciones corporativas* obtenemos una precisión del 97,67%. Tarda 0,74 segundos de CPU en calcular el modelo.

Si aplicamos las *libsvm* a través del código 4.12 al mismo conjunto de datos obtenemos una precisión del 97,33%. Tarda 1,7 segundos de CPU en calcular el modelo.

Uso de un kernel definido por el usuario con las SVMs

Si queremos crear un kernel nuevo como los que vimos en el subapartado anterior con las *svmlight*, tenemos que implementarlo en C en los archivos *kernel.h* y *kernel.c* que acompañan la distribución y recompilarlos.

Con la *libsvm*, podemos utilizar la opción de *kernel precalculado**. En la documentación nos dice que el formato ha de ser tal y como muestra la tabla 32, donde *L* corresponde al número de ejemplos de entrenamiento, *<clase>* será +1 o -1 y en lugar del ? podemos poner cualquier cosa. Tenemos que escoger como kernel el número 4. El listado 4.13 muestra el resultado de aplicar todo esto al corpus anterior.

* En inglés, *precomputed kernel*.

Tabla 32. Ejemplo de representación de un documento en *libsvm* para el kernel precalculado

ejemplo de entrenamiento x_i
$\langle \text{clase} \rangle 0 : i 1 : \kappa(x_i, x_1) \dots L : \kappa(x_i, x_L)$
ejemplo de test x
$\langle \text{clase} \rangle 0 : ? 1 : \kappa(x, x_1) \dots L : \kappa(x, x_L)$

Código 4.13: SVM en *libsvm* con matriz de kernel

```

1 from libsvm.svmutil import *
2 from time import clock
3
4 def kernelVSM(a, b):
5     la = [(int(k), float(v)) for k, v in (l.split(':') for l in a)]
6     lb = [(int(k), float(v)) for k, v in (l.split(':') for l in b)]
7     s = 0
8     i = 0
9     j = 0
10    while (i < len(la)) and (j < len(lb)):
11        if la[i][0] == lb[j][0]:
12            s += la[i][1] * lb[j][1]
13            i += 1
14            j += 1
15        elif la[i][0] < lb[j][0]:
16            i += 1
17        else:
18            j += 1
19    return s
20
21 train = list(map(lambda l: (l.strip()).split('_'),
22                 filter(lambda x: x[0] != '#',
23                       open('train.dat', 'r').readlines())))
24 y=list(map(lambda x: int(x.pop(0)), train))
25
26 inici = clock()
27 x = [dict([(i+1, kernelVSM(train[i], train[j]))
28          for i in range(len(train))])
29       for j in range(len(train))]
30 list(map(lambda l, i: l.update({0:i+1}), x, range(len(x))))
31 print("km", clock() - inici, "s", len(x), "x", len(x[0]))
32
33 inici = clock()
34 model = svm_train(y, x, '-s_0_-t_4')
35 print("learn", clock() - inici, "s")
36 svm_save_model('model.km.txt', model)
37
38 ##model = svm_load_model('model.km.txt')
39
40 test = list(map(lambda l: (l.strip()).split('_'),
41                filter(lambda x: x[0] != '#',
42                      open('test.dat', 'r').readlines())))
43 yt=list(map(lambda x: int(x.pop(0)), test))
44 inici = clock()
45 xt = [dict([(i+1, kernelVSM(test[j], train[i]))
46          for i in range(len(train))])
47       for j in range(len(test))]
48 list(map(lambda l, i: l.update({0:-1}), xt, range(len(xt))))
49 print("kmt", clock() - inici, "s", len(xt), "x", len(xt[0]))

```

```
50
51 inici = clock()
52 p_label, p_acc, p_val = svm_predict(yt, xt, model)
53 print("predict", clock() - inici, "s")
```

Al ejecutar este programa, obtenemos una precisión del 97,7%, con unos tiempos: 31 minutos, 7 segundos, 9 minutos y 2 segundos respectivamente para: la construcción de la matriz de kernel, el proceso de aprendizaje, la construcción de la matriz de test y el proceso de clasificación.

4.6. Protocolos de test

Este subapartado está dedicada a la validación de los procesos de aprendizaje, las medidas de evaluación de sistemas de clasificación y la significación estadística.

4.6.1. Protocolos de validación

La validación es el proceso por el que comprobamos cómo de bien (o mal) ha aprendido un conjunto de datos un algoritmo de clasificación.

La validación más simple, conocida como *validación simple*, consiste en dividir el conjunto de datos en dos: uno llamado de entrenamiento* y el otro de test. Se construye el modelo sobre el conjunto de entrenamiento y acto seguido se clasifican los ejemplos de test con el modelo generado a partir del de entrenamiento. Una vez obtenidas las predicciones se aplica alguna de las medidas de evaluación como las que se describen en el siguiente subapartado. Una cuestión importante a tener en cuenta cuando dividimos un conjunto de datos en los conjuntos de entrenamiento y test es mantener la proporción de ejemplos de cada clase en los dos conjuntos.

En el caso de que tengamos un algoritmo de aprendizaje que necesita realizar ajustes de parámetros, se divide el conjunto de entrenamiento en dos conjuntos: uno de entrenamiento propiamente dicho y otro de validación*. Se ajustan los parámetros entre el conjunto de entrenamiento y el conjunto de validación y una vez encontrados los óptimos; juntamos el conjunto de entrenamiento con el de validación y generamos el modelo con el algoritmo de aprendizaje. La validación la realizamos sobre el conjunto de test. Un error de método bastante usual es el utilizar el conjunto de test para realizar el ajuste de parámetros; esto no es estadísticamente correcto. Una de las técnicas más utilizadas para ajustar parámetros son los algoritmos de optimización como los algoritmos genéticos que se describen en el subapartado 5.4.

* En inglés, *training*.

* El conjunto de validación para el ajuste de parámetros no tiene nada que ver con el proceso de validación de los algoritmos de aprendizaje.

Un problema que se ha detectado en el uso de la validación simple es que según el conjunto de datos que tengamos, puede variar mucho el comportamiento del sistema en función de la partición de ejemplos que hayamos obtenido. Es decir, diferentes particiones de ejemplos conducen a diferentes resultados. En muchos casos, los investigadores dejan disponibles los conjuntos ya divididos en entrenamiento y test para que las comparaciones entre sistemas sean más fiables.

Para minimizar este efecto, se suele utilizar la *validación cruzada** en lugar de la simple. Este tipo de validación consiste en dividir un conjunto en k subconjuntos. Acto seguido, se realizan k pruebas utilizando en cada una un subconjunto como test y el resto como entrenamiento. A partir de aquí se calcula el promedio y la desviación estándar de los resultados. Con esto podemos ver mejor el comportamiento del sistema. Un valor muy utilizado de la k es 10. Una variante conocida de la validación cruzada es el *dejar uno fuera***.

Este caso es como el anterior definiendo la k como el número de ejemplos del conjunto total. Nos quedaría el conjunto de test con un único ejemplo. Este método no se suele utilizar debido a su alto coste computacional.

* En inglés, *cross-validation* o *k-fold cross-validation*.

** En inglés, *leave one out*.

4.6.2. Medidas de evaluación

Uno de los problemas que nos encontramos al describir las medidas de evaluación es la gran diversidad de medidas que se utilizan en las diferentes áreas de investigación. Estas medidas suelen tener en cuenta las diferentes peculiaridades de los problemas de los diferentes campos. Otro problema es que en ocasiones estas medidas de evaluación reciben nombres diferentes en función del área. En este subapartado vamos a ver las medidas que se utilizan en los contextos de la clasificación, la recuperación de la información y la teoría de la detección de señales.

La mayoría de las medidas de evaluación se pueden expresar en función de la matriz de confusión o tabla de contingencia. La matriz de confusión contiene una partición de los ejemplos en función de su clase y predicción. La tabla 33 muestra el contenido de las matrices de confusiones para el caso binario. A modo de ejemplo, la celda *verdadero positivo* corresponde al número de ejemplos del conjunto de tests que tienen tanto la clase como la predicción positivas. A los *falsos positivos* también se los conoce como *falsas alarmas* o *error de tipo I*; a los *falsos negativos* como *error de tipo II*; a los *verdaderos positivos* como *éxitos*; y a los *verdaderos negativos* como *rechazos correctos*.

Tabla 33. Matriz de confusión

		clase real	
		positiva	negativa
predicción	positiva	verdadero positivo (tp)	falso positivo (fp)
	negativa	falso negativo (fn)	verdadero negativo (tn)

El *error* o *ratio de error* mide el número de ejemplos que se han clasificado incorrectamente sobre el total de ejemplos. Se pretende que los algoritmos de aprendizaje tiendan a minimizar esta medida. Su fórmula corresponde a:

$$\text{error} = \frac{fp + fn}{tp + fp + fn + tn}$$

La *exactitud** (a veces llamada también *precisión*) corresponde a los ejemplos que se han clasificado correctamente sobre el total de ejemplos. Esta medida es la complementaria de la anterior. Su fórmula corresponde a:

* En inglés, *accuracy*.

$$\text{exactitud} = \frac{tp + tn}{tp + fp + fn + tn}$$

La *precisión** o *valor predictivo positivo* corresponde a los ejemplos positivos bien clasificados sobre el total de ejemplos con predicción positiva. Su fórmula corresponde a:

* En inglés, *precision*.

$$\text{precisión} = \frac{tp}{tp + fp}$$

La *sensibilidad** corresponde a los ejemplos positivos bien clasificados sobre el total de ejemplos positivos. Su fórmula corresponde a:

* En inglés, *recall*.

$$\text{sensibilidad} = \frac{tp}{tp + fn}$$

El conjunto de precisión y sensibilidad puede verse como una versión extendida de la exactitud. Estas medidas vienen del campo de la recuperación de la información donde el valor de *tn* es muy superior al resto y sesga las medidas de evaluación. De allí nos llega también la medida *F1*, propuesta en 1979 por van Rijsbergen, que combina las dos anteriores:

$$F1 = 2 \times \frac{\text{precisión} \times \text{sensibilidad}}{\text{precisión} + \text{sensibilidad}} = \frac{2 \times tp^2}{2 \times tp + fn + fp}$$

Esta medida descarta los elementos negativos debido al sesgo producido por la diferencia entre el número de ejemplos negativos y el de positivos; en el ámbito de la recuperación de información llega a ser muy crítico.

Existen otras medidas que se utilizan a partir de la matriz de confusión, como puede ser la *especificidad*:

$$\text{especificidad} = \frac{tn}{fp + tn}$$

que se suelen utilizar en el área de la teoría de detección de señales. De aquí sale también un diagrama llamado ROC que se utiliza en varios ámbitos como el de la biotecnología.

Todas las medidas de este subapartado se han descrito a partir del problema binario, pero también se pueden utilizar en problemas multiclase.

4.6.3. Tests estadísticos

Cuando tenemos las predicciones de dos clasificadores y hemos aplicado alguna de las medidas de evaluación; estadísticamente no es suficiente con mirar cuál de ellos ha obtenido un valor más alto; hay que mirar, además, si la diferencia es estadísticamente significativa.

Para realizar esto, hay dos formas de hacerlo en función del tipo de validación que hayamos utilizado. Si hemos utilizado la validación cruzada, tenemos que usar el *test de Student* y si hemos utilizado la validación simple el *test de McNemar*. Dados dos clasificadores F y G :

- el test de *Student* con una confianza de $t_{9,0,975} = 2,262$ o equivalente (depende del número de particiones) en la validación cruzada:

$$\frac{\bar{p} \times \sqrt{n}}{\sqrt{\frac{\sum_{i=1}^n (p(i) - \bar{p})^2}{n-1}}} > 2,262$$

donde n es el número de particiones, $\bar{p} = \frac{1}{n} \sum_{i=1}^n p(i)$, $p(i) = p_F(i) - p_G(i)$, y $p_X(i)$ es el número de ejemplos mal clasificados por el clasificador X (donde $X \in \{F, G\}$).

- el test de *McNemar* con un valor de confianza de $\chi_{1,0,95}^2 = 3,842$ con la validación simple:

$$\frac{(|A - B| - 1)^2}{A + B} > 3,842$$

donde A es el número de ejemplos mal clasificados por el clasificador F pero no por el G , y B es el número de ejemplos mal clasificados por el G pero no por el F .

Demšar, en el artículo "Statistical Comparisons of Classifiers over Multiple Data Sets", recomienda como alternativa el test no paramétrico de Wilcoxon:

$$z = \frac{T - \frac{1}{4}N(N+1)}{\sqrt{\frac{1}{24}N(N+1)(2N+1)}}$$

Lectura complementaria

T. Fawcet (2006). An Introduction to ROC Analysis. *Pattern Recognition Letters*, 27, 861-874.

Lectura complementaria

T. G. Dietterich (1998). "Approximate statistical tests for comparing supervised classification learning algorithms". *Neural Computation* (núm. 10, págs. 1895-1924).

Lectura complementaria

J. Demšar (2006). "Statistical Comparisons of Classifiers over Multiple Data Sets". *Journal of Machine Learning Research* (núm. 7, págs. 1-30).

donde N es el número de conjuntos de datos y $T = \min(R^+, R^-)$, donde:

$$R^+ = \sum_{d_i > 0} \text{rank}(d_i) + \frac{1}{2} \sum_{d_i = 0} \text{rank}(d_i)$$

$$R^- = \sum_{d_i < 0} \text{rank}(d_i) + \frac{1}{2} \sum_{d_i = 0} \text{rank}(d_i)$$

teniendo en cuenta que d_i es la diferencia de la medida de evaluación entre dos clasificadores sobre el conjunto de datos i y que realizamos un *ranking* sobre estas distancias. Los rankings tal que $d_i = 0$ se dividen equitativamente entre las dos sumas. En caso de ser un número impar de ellas, se ignora una. Para $\alpha = 0,05$, la hipótesis nula se rechaza cuando z es menor que $-1,96$.

4.6.4. Comparativa de clasificadores

En este subapartado vamos a ver dos formas de comparar clasificadores entre ellos.

Comparativas de más de dos clasificadores sobre múltiples conjuntos de datos

La forma clásica de realizar una comparativa de más de dos clasificadores sobre múltiples conjuntos de datos es utilizar el método estadístico ANOVA. Desafortunadamente, este estadístico realiza muchas suposiciones que usualmente no se cumplen en aprendizaje automático. Demšar nos recomienda un test alternativo, el de Friedman.

Supongamos que tenemos k clasificadores y n conjuntos de datos. Para calcular el test de Friedman realizamos un ranking de los diferentes algoritmos para cada conjunto de datos por separado. En caso de empate promediamos los rankings. r_i^j corresponderá al ranking del algoritmo j sobre el conjunto de datos i . Bajo la hipótesis nula, el estadístico se calcula como:

$$\chi_F^2 = \frac{12n}{k(k+1)} \left(\sum_j R_j^2 - \frac{k(k+1)^2}{4} \right)$$

donde $R_j = \frac{1}{n} \sum_i r_i^j$ corresponde al promedio de los algoritmos sobre el conjunto de datos i .

Lecturas complementarias

J. Demšar (2006). "Statistical Comparisons of Classifiers over Multiple Data Sets". *Journal of Machine Learning Research* (núm. 7, págs. 1-30).

Este estadístico se distribuye de acuerdo con χ^2_F con $k - 1$ grados de libertad, cuando n y k son suficientemente grandes ($n > 10$ y $k > 5$)*.

El estadístico siguiente** es una mejora del anterior (no tan conservador):

$$F_F = \frac{(n-1)\chi_f^2}{n(k-1) - \chi_f^2}$$

que se distribuye de sobre una distribución F con $k - 1$ y $(k - 1)(n - 1)$ grados de libertad.

Medidas de acuerdo y kappa

En este subapartado vamos a ver dos de las medidas más útiles para comparar el comportamiento de diferentes clasificadores.

Dadas las predicciones de dos clasificadores F y G , el *acuerdo** (observado) se calcula como:

$$Pa = \frac{A}{N}$$

donde A corresponde al número de ejemplos para los que los dos han dado la misma predicción y N es el número total de ejemplos.

Y el estadístico *kappa* como:

$$\kappa(F,G) = \frac{Pa - Pe}{1 - Pe}$$

donde Pe (el *agreement* esperado) se calcula como:

$$Pe = \sum_{i=1}^{ns} \frac{P_F(i) \times P_G(i)}{N^2}$$

donde ns es el número de clases, y $P_X(i)$ es el número de predicciones del clasificador X (con $X \in \{F,G\}$) de la clase i .

Este estadístico tiene en cuenta cómo se distribuyen las predicciones entre las diferentes clases los clasificadores. A partir de él, se puede estudiar cómo se comportan diferentes clasificadores entre ellos e incluso crear un dendrograma de los mismos. En un experimento realizado sobre un problema de lenguaje natural*, da como resultado un parecido notable entre los algoritmos

* Para valores menores, consultad los valores críticos exactos en:
 J. H. Zar (1998). *Biostatistical Analysis* (4.ª ed.). Prentice Hall.
 D. J. Sheskin (2000). *Handbook of parametric and nonparametric statistical procedures*. Chapman & Hall/CRC.

** R. L. Iman; J. M. Davenport (1980). "Approximations of the critical region of the Friedman statistic". *Communications in Statistics*.

* En inglés, *agreement*.

Ved también

En el subapartado 2.3.3 se estudia la creación de dendrogramas.

* Capítulo 7 del libro: E. Agirre; P. Edmonds (ed.) (2006) *Word Sense Disambiguation: Algorithms and Applications*. Springer.

del AdaBoost y las SVMs diferenciándolos del resto, por poner solo un ejemplo. El experimento parte de una tabla como la 34, donde c_i corresponde al clasificador i .

Tabla 34. Matriz de kappa dos a dos

	c_1	c_2		c_{N-1}
c_2	$\kappa(c_1, c_2)$	–	–	–
c_3	$\kappa(c_1, c_3)$	$\kappa(c_2, c_3)$	–	–
\vdots	\vdots	\vdots	\ddots	\vdots
c_N	$\kappa(c_1, c_N)$	$\kappa(c_2, c_N)$	–	$\kappa(c_{N-1}, c_N)$

Conclusiones

Hay dos conceptos que afectan profundamente a todos los procesos de aprendizaje. El primero es la llamada maldición de la dimensionalidad*. Este concepto tiene que ver con la necesidad de ejemplos de entrenamiento necesarios para aprender en función de la complejidad de las reglas de clasificación; que aumenta de forma exponencial.

* En inglés, *curse of dimensionality*.

El segundo concepto es el sesgo inductivo*. Este concepto contempla que los diferentes algoritmos de aprendizaje favorecen diferentes tipos de reglas de clasificación en los procesos de aprendizaje.

* En inglés, *inductive bias*.

Sería deseable disponer de un algoritmo de aprendizaje que fuera mejor que todos los demás, independientemente del conjunto de datos a tratar. Desafortunadamente, esto no ocurre y hace que necesitemos diferentes algoritmos para diferentes tipos de conjuntos de datos. Estos resultados se han cuantificado y se los ha denominado, en inglés, *the no free lunch theorem* (Kulkarni y Harman, 2011).

Todo esto ha hecho que muchos autores consideren que el diseño de aplicaciones prácticas de procesos de aprendizaje pertenece tanto a la ingeniería y a la ciencia, como al arte.

5. Optimización

5.1. Introducción

La **optimización** de un problema P es la tarea de buscar los valores que aplicados a P satisfagan unas determinadas expectativas. Dependiendo de la naturaleza de P las expectativas que han de cumplir esos valores, o sea, lo que debe buscarse en la tarea de optimización puede tener características muy diferentes; en cualquier caso, se supone que son las características más deseables para quien propone el problema.

Así, se pueden aplicar técnicas de optimización para encontrar valores que minimicen los costes de una actividad, que minimicen el tiempo requerido para llevarla a cabo, que determinen la trayectoria óptima de un vehículo espacial, que gestionen de forma óptima un conjunto de recursos como una red de servidores, que maximicen los beneficios de una inversión o que encuentren la combinación de genes que más se asocia a una determinada patología, entre innumerables ejemplos posibles.

Para cada problema P se define una **función objetivo** f que mide de alguna forma la idoneidad de cada posible solución de P : coste, tiempo requerido, beneficio, combustible empleado, tiempo de respuesta de un sistema, productividad por hora, etc. El dominio de f , o sea el conjunto de puntos que pueden probarse como solución al problema, recibe el nombre de **espacio de soluciones** y se denota por la letra S .

El rango de f suele ser un valor escalar (entero o real, o un subconjunto), si bien en algunos casos puede tratarse de un valor de más dimensiones. Dependiendo de P y de la forma en que se defina f la tarea de optimización puede consistir en encontrar los puntos que minimicen f (coste, tiempo requerido, etc.) o los puntos que la maximicen (beneficio, productividad, eficiencia).

A menudo la función f se define como una **función de error**, que es la diferencia entre el resultado deseado y el obtenido por un punto $x \in S$. En esos casos el objetivo de la tarea de optimización siempre es, lógicamente, minimizar f .

En cualquier caso, hay que diferenciar el enunciado del problema P de la función objetivo f , pues a menudo conviene definir f de manera sustancialmente distinta a P para incrementar la efectividad de los métodos de optimización.

Si vemos f como el relieve geográfico de un país, con valles y montañas, es importante que f sea una función sin grandes llanuras, grandes zonas de valor homogéneo que no dan ninguna orientación sobre el mejor camino a tomar. El agua se estanca en las llanuras, sólo fluye y busca el mar en las pendientes. f debe definirse de forma que refleje los más pequeños cambios en la calidad de una solución, para que los métodos de optimización sepan que, poco a poco, están mejorando la solución propuesta. En el subapartado 5.7 se plantea un ejemplo, el coloreado de un mapa, en el que se aprecia claramente este requerimiento. Si f es el número de colores necesario para colorear un mapa sin que haya dos países contiguos del mismo color, tendremos una función objetivo con vastas llanuras. Sin embargo, si definimos f como el número de países contiguos del mismo color, tendremos una medida mucho más fina del progreso de la optimización, pues cada vez que se resuelva uno de esos “errores”, f disminuirá un poco.

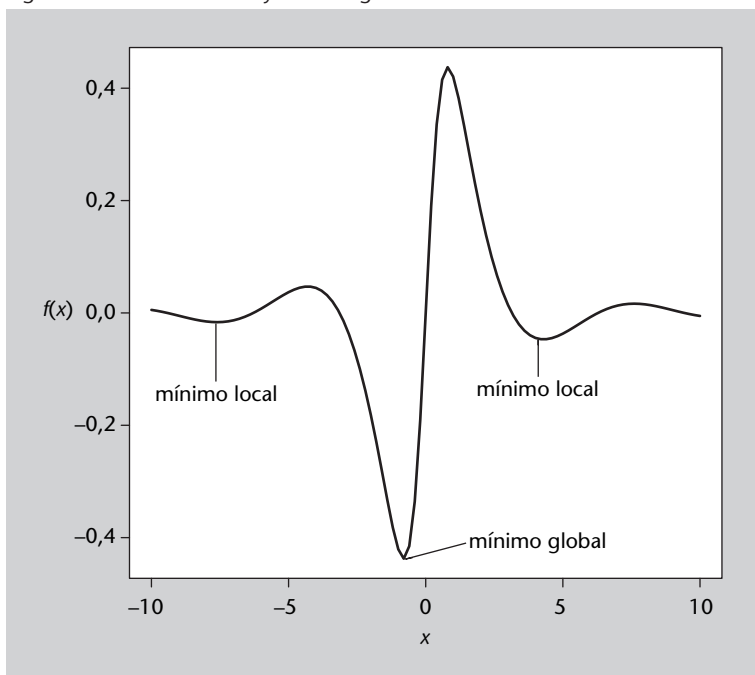
Este ejemplo nos lleva a una última consideración sobre f : en muchos problemas existen soluciones prohibidas, ya que incumplen algunas restricciones del problema; en el ejemplo anterior, un coloreado en el que dos países contiguos tienen el mismo color no es válido. Sin embargo muy a menudo conviene definir una f **relajada** respecto a P , que permita tales soluciones prohibidas como paso intermedio para alcanzar las soluciones óptimas deseadas. De lo contrario, el salto necesario para pasar de una solución válida a otra mejor puede ser tan grande que resulte, en la práctica, inalcanzable. Por ejemplo, permitiendo que dos países contiguos tengan el mismo color como paso intermedio para conseguir un coloreado válido. Si no se permite, el método tiene que obtener un nuevo coloreado del mapa en un solo paso, lo que casi seguro lo condenará al fracaso.

Siguiendo con la analogía geográfica del espacio de soluciones, y suponiendo que queremos minimizar f , lo que se busca es el valle con el punto más bajo (en el caso contrario buscaremos la montaña más alta). Dependiendo del problema es posible que exista un único valle con la solución óptima o, por el contrario, que haya numerosos valles con soluciones más o menos buenas. Si nos encontramos en el fondo de un valle nos parece que nuestra posición es la más baja, aunque es posible que haya otro valle más bajo aún en alguna otra parte. En un problema de optimización puede ocurrir lo mismo, que haya muchas zonas con soluciones relativamente buenas, si bien en otra zona de S se puedan encontrar soluciones aún mejores. En este caso se habla de **óptimos locales** y **óptimos globales**.

El objetivo de los métodos de optimización es encontrar el óptimo global y no “dejarse engañar” por los óptimos locales, aunque esto no siempre es fácil o posible.

El principal obstáculo es que, para pasar de un óptimo local a un óptimo global (o a un óptimo local mejor que el primero), es necesario pasar por puntos peores (salir de un valle para bajar a otro), y en principio eso se ve como un empeoramiento de la solución y por tanto se descarta. Puede verse un ejemplo en la figura 49.

Figura 49. Mínimos locales y mínimo global



Una última consideración se refiere al tamaño de S . En gran parte de los problemas reales, S tiene un tamaño muy grande, en especial si tiene muchas variables o dimensiones. En lo que se conoce como la **maldición de la dimensionalidad***, si tenemos un espacio de soluciones de una dimensión (una recta) de lado 1, con 100 puntos se puede explorar en intervalos de 0,01. Sin embargo, si tenemos un espacio de dos dimensiones y lado 1 (un cuadrado), serán necesarios 100^2 puntos para poder explorarlo con una resolución de 0,01; en un cubo se necesitan 100^3 puntos, y en un hipercubo de 10 dimensiones serán necesarios 100^{10} puntos. Por ese motivo una exploración exhaustiva de S suele ser impracticable, y ni siquiera métodos como el de Newton, que van subdividiendo el espacio de soluciones progresivamente, son prácticos.

Así, en muchos casos no se conoce cuál es la solución óptima, de hecho, los métodos de optimización descritos en este apartado están diseñados para encontrar soluciones aceptablemente buenas, ya que no es posible saber si son las óptimas, en espacios de soluciones muy grandes (con muchas dimensiones).

* En inglés, *curse of dimensionality*, término acuñado por Richard E. Bellman

Ved también

Una estrategia complementaria consiste en utilizar las técnicas de reducción de dimensionalidad vistas en el subapartado 3.1 de este módulo.

5.1.1. Tipología de los métodos de optimización

Existe una gran variedad de métodos de optimización debido a que cada problema requiere o permite estrategias diferentes. Básicamente se puede clasificar en las siguientes categorías:

- **Métodos analíticos:** consisten en realizar alguna transformación analítica de la función objetivo para determinar los puntos de interés. Como ejemplo se estudia el método de los multiplicadores de Lagrange en el subapartado 5.2, aunque también hay otros métodos, como la programación cuadrática.
- **Métodos matemáticos iterativos:** aplican una operación repetidamente hasta que se cumple un criterio determinado. Algunos ejemplos son el método de Newton y el método de ascenso (o descenso) de gradientes.
- **Métodos metaheurísticos:** se denomina **heurísticos** a las técnicas basadas en la experiencia o en analogías con otros procesos que resultan útiles para resolver un problema. En el área que nos ocupa, la de la optimización, los heurísticos son técnicas que suelen ayudar a optimizar un problema, habitualmente tomando soluciones aleatorias y mejorándolas progresivamente mediante el heurístico en cuestión. Los métodos que estudiaremos en este texto pertenecen a esta categoría (salvo el de Lagrange), pues son los que se consideran propios del área de la inteligencia artificial. Ejemplos: búsqueda aleatoria, algoritmos genéticos y programación evolutiva, búsqueda tabú, recocido simulado, etc.

El prefijo “meta-” de los métodos metaheurísticos se debe a que tales heurísticos son aplicables a cualquier problema, pues los métodos no presuponen nada sobre el problema que pretenden resolver y por tanto son aplicables a una gran variedad de problemas. En general se hablará de “metaheurístico” o simplemente de “método” para hablar de cada técnica en general, y de “algoritmo” para la aplicación de una técnica a un problema concreto, si bien por motivos históricos esta nomenclatura no siempre se sigue (como en el caso de los algoritmos genéticos).

Los métodos metaheurísticos no garantizan la obtención de la solución óptima, pero suelen proporcionar soluciones aceptablemente buenas para problemas inabordables con técnicas analíticas o más exhaustivas.

5.1.2. Características de los metaheurísticos de optimización

La mayoría de metaheurísticos de optimización parten de una o más soluciones elegidas aleatoriamente que se van mejorando progresivamente. Si se parte de una única solución aleatoria y las nuevas soluciones propuestas por

el heurístico dependen de ella, se corre el riesgo de quedarse estancado en un óptimo local. Por ese motivo tales métodos suelen denominarse **métodos locales**, en contraposición a los **métodos globales**, que suelen generar varias soluciones aleatorias independientes y por tanto pueden explorar diferentes áreas del espacio de soluciones; así evitan, en la medida de lo posible, limitarse a un óptimo local determinado.

Otro aspecto común a los métodos metaheurísticos que se estudiarán en este apartado es su naturaleza iterativa, lo que lógicamente lleva a la cuestión del número de iteraciones que deberá ejecutar cada algoritmo. Cuando el óptimo global sea conocido (por ejemplo, en una función de error será 0) y sea posible alcanzarlo en un tiempo aceptable, se podrá iterar el algoritmo hasta llegar al óptimo global. Sin embargo, en la mayoría de problemas el óptimo global no es conocido o no se puede garantizar que se encontrará en un tiempo determinado. Por tanto, hay que fijar algún criterio de terminación del algoritmo. Entre los criterios más habituales se encuentran:

- Alcanzar un valor predeterminado de la función objetivo (por ejemplo, un error $< \epsilon$).
- Un número fijo de iteraciones.
- Un tiempo de ejecución total determinado.
- Observar una tendencia en las soluciones obtenidas (no se observa mejora durante n iteraciones).

Por último, los métodos metaheurísticos suelen caracterizarse por una serie de parámetros que deben ajustarse de forma adecuada para resolver cada problema eficientemente. Al tratarse de heurísticos no hay reglas objetivas que determinen tales parámetros, y por tanto se recurre a valores orientativos obtenidos en estudios experimentales sobre diferentes tipos de problemas. De hecho, en algunos casos se aplica un algoritmo de optimización sobre los parámetros del algoritmo de optimización, lo que se conoce como **metaoptimización**.

5.2. Optimización mediante multiplicadores de Lagrange

Un fabricante de envases nos plantea la siguiente cuestión: ¿qué proporciones debe tener una lata cilíndrica para que contenga un volumen determinado utilizando la menor cantidad posible de metal? ¿Qué es mejor? ¿Que sea muy alta, muy ancha o igual de alta que de ancha?

La optimización mediante multiplicadores de Lagrange es un método de optimización matemática que permite encontrar el mínimo o el máximo de una función teniendo en cuenta una o más **restricciones**, que son otra función o funciones que se deben satisfacer.

En el ejemplo anterior, la función que se desea minimizar es la que determina la superficie de la lata, y la restricción es la que determina su volumen. Nótese que, si no se añade la restricción del volumen, la minimización de la función de la superficie dará como resultado una lata de altura y radio cero; un resultado correcto pero poco útil en la práctica.

5.2.1. Descripción del método

A continuación se describirá el método aplicándolo a funciones de dos variables; su generalización a más dimensiones es inmediata. Sea $f(x,y)$ la función que deseamos minimizar, y sea $g(x,y) = c$ la restricción que se aplica. Sin entrar en la explicación matemática, los puntos que buscamos son aquellos en los que los **gradientes** de f y g coinciden:

$$\nabla_{x,y}f(x,y) = \lambda \nabla_{x,y}g(x,y) \quad (48)$$

Donde λ es una constante que es necesaria para igualar las longitudes de los vectores gradiente, ya que pueden tener la misma dirección pero longitudes diferentes. La función gradiente de una función se calcula mediante las derivadas parciales de la función para cada una de sus variables:

$$\nabla_{x,y}f(x,y) = \left\langle \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y} \right\rangle \quad (49)$$

$$\nabla_{x,y}g(x,y) = \left\langle \frac{\partial g}{\partial x}, \frac{\partial g}{\partial y} \right\rangle \quad (50)$$

Por lo tanto el problema se transforma en un sistema de ecuaciones que hay que resolver para obtener la solución deseada:

$$\left. \begin{aligned} \frac{\partial f}{\partial x} &= \lambda \frac{\partial g}{\partial x} \\ \frac{\partial f}{\partial y} &= \lambda \frac{\partial g}{\partial y} \end{aligned} \right\} \quad (51)$$

En este sistema hay dos ecuaciones y tres variables: x , y y λ . En general obtendremos los valores para x e y en función de λ , que es el parámetro que nos da el grado de libertad necesario para elegir los valores deseados que cumplan la restricción impuesta; en otras palabras, nuestro problema de optimización se ha transformado en una función de una única variable, λ , que nos permite elegir el valor más adecuado a las especificaciones del problema.

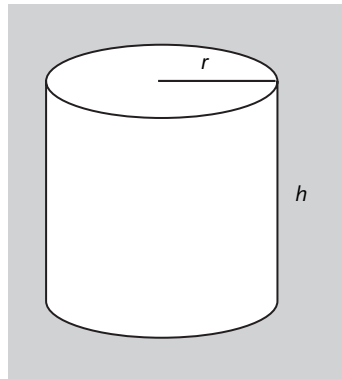
Gradiente de una función

El gradiente de una función f en un punto es un vector que apunta en la máxima pendiente de f . La función gradiente de f es un campo de vectores que apuntan, en cada punto, en la dirección de máxima pendiente de f , y se denota por ∇f .

5.2.2. Ejemplo de aplicación

Volvamos al problema de la lata cilíndrica cuya superficie deseamos minimizar para un determinado volumen. En la figura 50 se muestra la lata con las dos variables que la caracterizan, el radio r y la altura h .

Figura 50. Una lata cilíndrica



En este caso, la función que deseamos minimizar es la superficie de la lata (superficie del lado más superficial de las dos tapas), que es la que implica la cantidad de metal utilizado, y la restricción es el volumen contenido por la lata, luego en este problema f y g serán:

$$f(x,y) = 2\pi rh + 2\pi r^2 \quad (52)$$

$$g(x,y) = \pi r^2 h \quad (53)$$

Tras obtener las funciones gradiente y multiplicar ∇g por λ , hay que resolver el siguiente sistema de ecuaciones:

$$\left. \begin{aligned} 2\pi rh &= \lambda(2\pi h + 4\pi r) \\ \pi r^2 &= \lambda 2\pi r \end{aligned} \right\} \quad (54)$$

Expresando r y h en función de λ se obtiene:

$$\left. \begin{aligned} r &= 2\lambda \\ h &= 4\lambda \end{aligned} \right\} \quad (55)$$

Y de ahí se deduce que la relación entre radio y altura de una lata debe ser $h = 2r$ para utilizar la menor cantidad de material con relación al volumen que contiene.

5.2.3. Análisis del método

La naturaleza analítica de este método condiciona sus ventajas e inconvenientes. Como ventaja destaca la posibilidad de obtener no una sino un rango de soluciones óptimas de forma relativamente rápida. Por otra parte, su principal limitación consiste en que sólo es aplicable a funciones diferenciables, lo que restringe el tipo de problemas en los que puede utilizarse este método.

5.3. Recocción simulada

Supongamos que tenemos un sistema informático con N servidores y C clientes que necesitan acceder a los servidores. Cada servidor tiene un tiempo de espera base t_{eb} , si bien el tiempo de espera efectivo t_{ee} de cada servidor depende tanto de t_{eb} como del número de clientes en ese servidor, C , según la expresión $t_{ee} = t_{eb}(1+C)$. Así, si un servidor tiene $t_{eb} = 3$ y 5 clientes, tendremos $t_{ee} = 3(1 + 5) = 18$.

En este problema se busca una distribución de C clientes entre N servidores, cada uno con un t_{eb} posiblemente distinto, de manera que el tiempo de espera efectivo medio para todos los clientes sea el mínimo.

La **recocción simulada*** es una metaheurística de optimización cuyo objetivo es encontrar una aproximación razonablemente buena del óptimo global de una función determinada. Hay, por tanto, dos ideas importantes: primero que lo que se pretende encontrar es una aproximación del punto óptimo global, dando por hecho que buscar exhaustivamente el óptimo global no es posible en un plazo de tiempo razonable dado el tamaño del espacio de búsqueda. En segundo lugar, la recocción simulada está diseñada para evitar, en la medida de lo posible, quedar atascada en óptimos locales.

* En inglés, *simulated annealing*.

La recocción simulada se inspira en la técnica metalúrgica de la *recocción*, en la que los metales se recalientan y vuelven a dejar enfriar varias veces para que sus átomos se vayan recolocando en estados de menor energía. Así se forman cristales mayores y por tanto se reducen las imperfecciones del metal y mejoran sus propiedades. Este proceso se lleva a cabo de forma decreciente: al principio el recalentamiento aplicado es mayor para eliminar los defectos de mayor tamaño, y progresivamente, el recalentamiento aplicado se reduce para eliminar los defectos más finos.

De la misma forma, la recocción simulada parte de un estado al azar dentro del espacio de búsqueda y va aplicando cambios en ese estado para ver si su energía (en este caso, la función que se desea minimizar) aumenta o dismi-

nuye. Con el fin de evitar quedar atascado en óptimos locales, la recocción simulada permite cambios a estados de mayor energía (que en principio son peores) para conseguir salir de un óptimo local en busca del óptimo global. A medida que avanza la ejecución, sin embargo, los empeoramientos que se permiten son cada vez menores, de forma que poco a poco el algoritmo vaya encontrando una aproximación razonablemente buena al óptimo global.

5.3.1. Descripción del método

El objetivo básico de la recocción simulada es pasar de un estado cualquiera del espacio de búsqueda, con una energía probablemente alta, a un estado de baja energía. El término *energía* se sustituirá, en cada problema, por la medida concreta que se desee minimizar (error, valor de una función, tiempo de respuesta, coste económico, etc.).

Se dice que un estado s' es **vecino** de otro estado s si existe alguna transformación sencilla (mínima) para pasar de s a s' . En el caso de espacios de búsqueda discretos, dicha transformación suele ser evidente; por el contrario, si el espacio de búsqueda es continuo, es necesario definir una distancia máxima de transformación para considerar que dos estados son vecinos.

En el proceso de recocción se aplica un incremento de temperatura mayor en las fases iniciales, y a medida que el proceso avanza dicho incremento va reduciéndose. De la misma forma, en la recocción simulada se empieza con un margen de temperatura alto y progresivamente se va reduciendo. De esta forma, en las iteraciones iniciales es probable que el algoritmo acepte un nuevo estado s' aunque tenga una energía más alta que el estado previo s ; sin embargo, a medida que el algoritmo avanza, es menos probable que se acepte un estado con energía más alta. La temperatura, por tanto, decrece a medida que el algoritmo avanza; una formulación habitual es $T = ir * F$, donde ir es el número de iteraciones restantes y F es un factor que permite ajustar la tolerancia con los niveles de energía de los nuevos estados, cuestión que se analiza más adelante.

El primer paso de un algoritmo que utiliza la recocción simulada consiste en tomar un estado al azar en el espacio de búsqueda. También es posible tomar un estado conocido que se sepa que es mejor que la media. Denominaremos s al estado actual, y $E(s)$ será su nivel de energía.

En cada iteración del algoritmo se genera aleatoriamente un estado s' vecino del estado actual s ; en función del nivel de energía del estado actual $e = E(s)$ y del nuevo estado $e' = E(s')$, el sistema cambia a s' o por el contrario perma-

nece en s . La **probabilidad de aceptación** del nuevo estado viene dada por la expresión:

$$P(e, e', T) = \begin{cases} 1 & \text{si } e' < e \\ \exp((e - e')/T) & \text{si } e' \geq e \end{cases} \quad (56)$$

Como se ve, la temperatura T -y por tanto el número de iteraciones actual- influye en la decisión de aceptar o no un nuevo estado. Recordemos que $T = i * F$; F se puede utilizar para ajustar la probabilidad de aceptación, dado que según el orden de magnitud de la energía de un estado, un incremento de energía de 1 puede ser muy grande o muy pequeño. Una regla sencilla para ajustar el algoritmo es $i * F \simeq \Delta E_{max}$, siendo i el número total de iteraciones del algoritmo y ΔE_{max} el máximo incremento de energía aceptable, ya que así la probabilidad inicial de aceptación tendrá un valor cercano al 50%.

En resumen, en las primeras iteraciones es probable que se acepten cambios de estado que impliquen un incremento de la energía (alejándose del objetivo, que es minimizar la energía). De esta forma se favorece que el sistema salga de un mínimo local e intente buscar un mínimo global. Sin embargo, a medida que el algoritmo avanza en su ejecución se va haciendo menos probable que se acepten cambios de estado con incrementos de energía, pues se supone que lo que debe hacer el algoritmo es encontrar el mínimo del entorno en el que se encuentre.

Una mejora habitual del algoritmo consiste en almacenar el mejor estado encontrado en toda su ejecución, y devolver ese estado como resultado final, aunque no se trate del estado final alcanzado. Si bien esta mejora no tiene correspondencia con el proceso físico de recocción, es muy sencillo implementarla y puede mejorar el comportamiento del algoritmo en algunos casos.

5.3.2. Ejemplo de aplicación

La recocción simulada se utiliza frecuentemente para optimizar la planificación y asignación de recursos de diversos tipos. En este caso vamos a aplicarla a un ejemplo sencillo de asignación de recursos, concretamente al problema de la asignación de clientes a servidores enunciado anteriormente.

Para desarrollar este algoritmo, consideraremos que un estado s es una partición determinada de los C clientes entre los N servidores; por ejemplo, si tenemos 40 clientes y 5 servidores, un posible estado es $s = (10, 4, 8, 7, 11)$. Un estado s' es vecino de s si de uno a otro sólo se ha movido un cliente de un servidor a otro. Un ejemplo de estado vecino de s sería $s' = (9, 5, 8, 7, 11)$, en el que un cliente se ha movido del servidor 1 al 2.

En la tabla 35 se muestra un ejemplo de ejecución de la recocción simulada aplicada al problema de asignación de clientes a servidores. El número de servidores es $N = 5$, el de clientes $C = 40$, los tiempos de espera base de los servidores son $(5,7,7,2,1)$, y en el estado inicial los clientes se distribuyen por igual entre los cinco servidores, luego el estado inicial es $s_i = (8,8,8,8,8)$. El número de iteraciones es 10, y el factor de cálculo de la temperatura es 0.1, de forma que los incrementos máximos de energía (tiempo medio de espera) sean del orden de 1.0. Nótese que cuando un cambio de estado no se acepta, se permanece en el estado actual. Las iteraciones están numeradas en orden decreciente para que coincidan con la formulación del algoritmo.

Tabla 35. Ejemplo de asignación de clientes a servidores

#Iter	A	B	C	D	E	Tº medio espera	$P(e,e',T)$	Aceptado
Inicial	8	8	8	8	8	39,6		
10	8	7	8	8	9	37,25	1	SÍ
9	8	7	9	8	8	39,95	0,05	NO
8	8	7	7	8	10	34,95	1	SÍ
7	7	8	7	8	10	35,75	0,32	SÍ
6	8	8	7	8	9	37,25	0,08	NO
5	6	8	8	8	10	36,8	0,12	NO
4	7	8	6	9	10	34,2	1	SÍ
3	6	8	6	9	11	33	1	SÍ
2	6	8	7	9	10	34,9	0,000075	NO
1	6	8	6	10	10	33,45	0,01	NO
Final	6	8	6	9	11	33		

A,B,C,D,E son los cinco servidores del ejemplo

Como se puede observar, es más probable que cambios de estado que incrementen la energía (el tiempo medio de espera) se acepten al principio de la ejecución (por ejemplo, en la iteración 7) que hacia el final (iteración 1). En este ejemplo de ejecución el mejor estado es el último que se ha encontrado. Recordemos que no tiene por qué ser así, ya que el algoritmo almacena el mejor estado encontrado y es posible (aunque no muy probable) que el estado final sea peor.

Una última aclaración: dada su simplicidad, este ejemplo se podría haber resuelto de forma analítica, obteniéndose la solución óptima fácilmente. En este caso el uso de recocción simulada no tendría sentido, aunque se ha utilizado para presentar un ejemplo suficientemente sencillo. En problemas reales más complejos y con más variables en juego, las soluciones analíticas resultan, por lo general, imposibles, y entonces surge la necesidad de utilizar métodos como la recocción simulada.

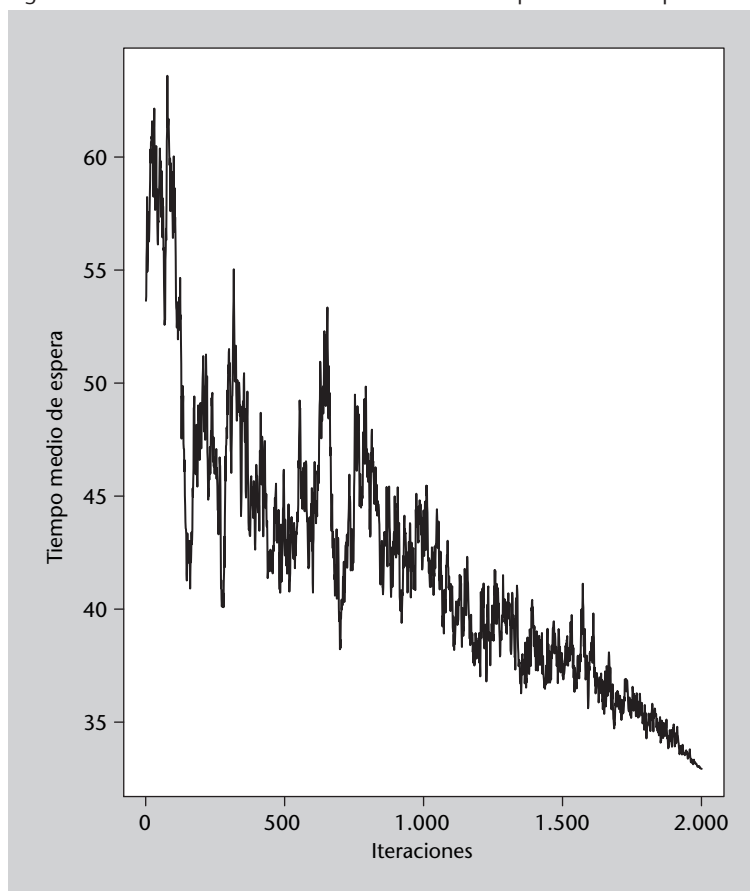
5.3.3. Análisis del método

La recocción simulada es aplicable a prácticamente cualquier problema de optimización, y una de sus principales ventajas es que no requiere elegir muchos parámetros de funcionamiento, a diferencia de otros métodos de optimización.

Su aplicación es más evidente en problemas cuyo espacio de búsqueda es discreto, pues de esa forma la definición de estado vecino es más directa; cuando el espacio de búsqueda es continuo, es necesario establecer una distancia de vecindad que introduce un parámetro más en el método.

En la figura 51 se puede ver cómo desciende el tiempo medio de espera en función del número de iteraciones. En este caso hay 50 servidores y 400 clientes. Nótese cómo en las primeras iteraciones el tiempo medio de espera sube y baja, pues el algoritmo tiene una temperatura mayor. A medida que el número de iteraciones crece, el tiempo medio disminuye o aumenta muy ligeramente, pues la temperatura es menor y no se aceptan cambios de estado que suban mucho la energía.

Figura 51. Efecto del número de iteraciones en el tiempo medio de espera



Por otra parte, en la figura 52 se compara el rendimiento de la recocción aleatoria frente a la búsqueda aleatoria. Como se ve, en las primeras iteraciones

la búsqueda aleatoria obtiene mejor resultados. Esto es debido a que en las primeras iteraciones la recocción tolera cambios a estados peores, tal y como se ha explicado anteriormente. Sin embargo, a medida que crece el número de iteraciones, la búsqueda aleatoria tiende a estancarse, mientras que la recocción sigue mejorando los resultados obtenidos, en esete caso reduciendo el tiempo medio de espera. Nótese también que el número iteraciones que requiere la recocción simulada es relativamente alto, puesto que en cada momento sólo se explora una posible solución.

Figura 52. Recoccción simulada frente a búsqueda aleatoria

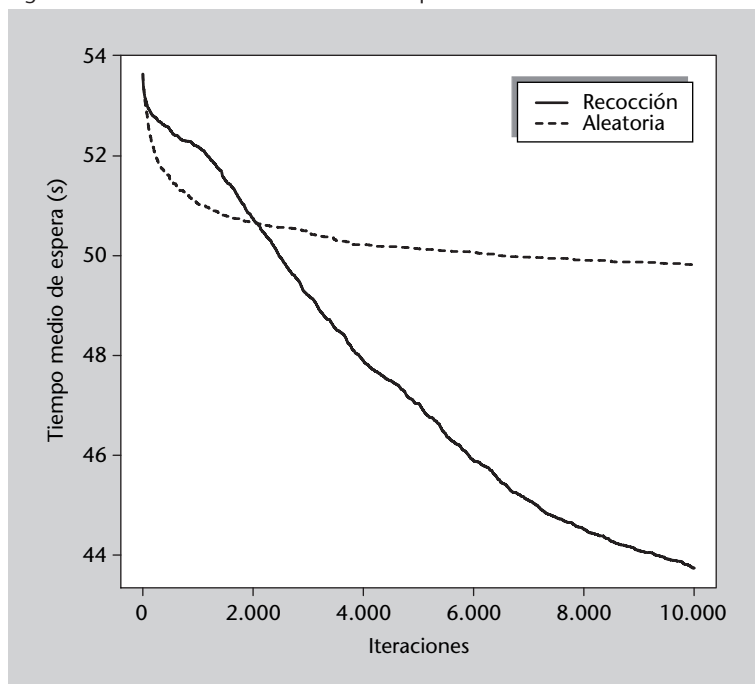


Figura 52

Comparativa entre recocción simulada y búsqueda aleatoria. Se muestra el tiempo medio de espera obtenido en un problema con 400 clientes y 50 servidores. Los valores mostrados son la media de 100 ejecuciones diferentes.

5.3.4. Código fuente en Python

Código 5.1: Algoritmo de recocción simulada

```

1 import random, sys, math, operator
2
3 # Calcula el tiempo medio de espera de un conjunto de
4 # servidores y una distribución de clientes
5 def tiempoMedioEspera(servidores, clientes):
6     tiempos = [t*(1+c) for (t,c) in zip(servidores, clientes)]
7     suma = sum(map(operator.mul, tiempos, clientes))
8     return suma / float(sum(clientes))
9
10
11 # Genera un vecino del estado actual: simplemente mueve un
12 # cliente de un servidor a otro al azar
13 def generaVecino(estado):
14     nuevo = estado[:]
15     hecho = False
16     while not hecho:
17         elegidos = random.sample(range(len(estado)), 2)
18         origen = elegidos[0]
19         destino = elegidos[1]
20
21     # El número de clientes en un servidor ha de ser >=0

```



```
22     if nuevo[origen] > 0:
23         nuevo[origen] -= 1
24         nuevo[destino] += 1
25         hecho = True
26
27     return nuevo
28
29
30 # Teniendo en cuenta la energía de cada estado, la iteración
31 # actual y el factor de tolerancia, decide si un nuevo estado
32 # se acepta o no
33 def acepta(energia, nuevaEnergia, iteraciones, factor):
34     if nuevaEnergia < energia:
35         return True
36     else:
37         valor = math.exp((energia-nuevaEnergia)/
38                         (iteraciones*factor))
39         return random.random() < valor
40
41
42 # Aplica el algoritmo de recocición simulada a un conjunto de
43 # servidores y una distribución de clientes. La tolerancia indica
44 # el empeoramiento aceptable al iniciar el algoritmo (aproximado).
45 def recocionSimulada(servidores, clientes, tolerancia,
46                     iteraciones):
47
48     # La tolerancia permite ajustar la temperatura para controlar
49 # si se acepta o no un nuevo estado "peor" (con mayor tiempo
50 # de respuesta).
51     factor = tolerancia / float(iteraciones)
52
53     # Se calcula el tiempo medio de espera del estado inicial
54     estado = clientes[:]
55     tiempo = tiempoMedioEspera(servidores, estado)
56
57     # Se almacena el mejor estado obtenido; será el que se devuelva
58     mejor = estado[:]
59     mejorTiempo = tiempo
60     for i in range(iteraciones):
61         nuevo = generaVecino(estado)
62         nuevoTiempo = tiempoMedioEspera(servidores, nuevo)
63
64         if acepta(tiempo, nuevoTiempo, iteraciones, factor):
65             estado = nuevo
66             tiempo = nuevoTiempo
67
68         if nuevoTiempo < mejorTiempo:
69             mejor = estado[:]
70             mejorTiempo = tiempo
71
72     return mejor
73
74
75 # Parámetros generales del problema
76 # Tolerancia indica aproximadamente el máximo empeoramiento
77 # que se acepta inicialmente
78 numServidores = 50
79 numClientes = 400
80 maxTRespuesta = 10
81 tolerancia = 1.0
82 iteraciones = 2000
83
84 # Generación de un conjunto de servidores
85 servidores = [random.randint(1,maxTRespuesta)
86               for i in range(numServidores)]
87 print(servidores)
88
89 # Inicialmente los clientes se distribuyen por igual entre
90 # todos los servidores; si sobra alguno, se asigna al azar
91 clientes = [numClientes/numServidores
92             for i in range(numServidores)]
93
```

```
94 resto = numClientes - sum(clientes)
95 if resto > 0:
96     for serv in random.sample(range(numServidores), resto):
97         clientes[serv] += 1
98
99 print(clientes, tiempoMedioEspera(servidores, clientes))
100
101 # Se llama a recocción simulada para que obtenga una buena
102 # distribución de los clientes en los servidores
103 clientes = recoccionSimulada(servidores, clientes, tolerancia,
104                             iteraciones)
105
106 print(clientes, tiempoMedioEspera(servidores, clientes))
```

5.4. Algoritmos genéticos

Supongamos que una empresa de inversiones *on-line* nos encarga que construyamos un asesor de inversiones para sus clientes. El planteamiento del problema es el siguiente: la empresa ofrece a sus clientes la posibilidad de invertir en bonos de varias compañías. Los bonos de cada compañía tienen tres características: precio por bono, rendimiento de los bonos (vamos a suponer que es conocido) y cantidad de bonos disponibles. El cliente, por su parte, dispone de un capital determinado. El objetivo de nuestro asesor de inversiones es que el cliente obtenga el rendimiento máximo de su capital seleccionando la mejor combinación de bonos. En este subapartado estudiaremos un potente método de optimización que podemos utilizar para resolver este problema así como muchos otros tipos de problemas.

Los seres vivos exhiben una enorme diversidad y una asombrosa adaptación a los entornos más diversos. Esto es posible mediante los mecanismos de la **selección natural**, descritos por Darwin en el siglo XIX. La idea central de su **teoría de la evolución** es que los organismos mejor adaptados al medio son los que tienen más posibilidades de sobrevivir y dejar descendencia, y por lo tanto son más abundantes que los peor adaptados, que posiblemente acaben extinguiéndose.

Las características de un organismo están determinadas por su **información genética**, que se hereda de padres a hijos. De esta manera, las características que ayudan a la supervivencia son las que se heredan con mayor probabilidad. En el caso de la reproducción sexual, la información genética del padre y la de la madre se combinan de manera aleatoria para formar la información genética de los hijos. Es lo que se llama el **cruzamiento**. Se trata de un proceso muy complejo, dado el gran volumen de información transferido, por lo que en raras ocasiones se produce un error de copia, dando lugar a una **mutación**, que en adelante heredarán los descendientes del portador.

La información genética de un individuo se divide en **genes**, que son secuencias de información genética que describen una característica del individuo*.

*Téngase en cuenta que se trata de una descripción muy simplificada. La realidad es bastante más compleja.

Los **algoritmos genéticos** emulan la selección natural sobre un conjunto de individuos para buscar la mejor solución a un problema determinado. La “información genética” de cada individuo es una posible solución al problema; por analogía, hay un “gen” para cada variable o parámetro del problema sobre el que se desea ejecutar el proceso de optimización. Para emular la selección natural, se crea una **población** o conjunto de individuos y se le hace evolucionar de forma que los mejor adaptados, o sea, los que son mejor solución para el problema, se reproduzcan con mayor probabilidad y poco a poco vayan surgiendo individuos mejor adaptados al problema; en otras palabras, mejores soluciones.

5.4.1. Descripción del método

Sea Q el problema que queremos resolver, y S el espacio de las soluciones de Q , en el que cada solución $s \in S$ se compone de n variables de la forma $s = \{x_1, x_2, x_3, \dots, x_n\}$. Esas variables pueden ser de cualquier tipo de datos; los más habituales en este caso son binarios, enteros y reales, o una combinación de los mismos. Además, cada variable tiene un rango de valores propio. En un algoritmo genético, cada individuo es una solución $s_i \in S$ con n variables (o genes, si seguimos la metáfora biológica).

También existe una función $f : S \rightarrow \mathbb{R}$ que mide la *idoneidad** de una solución $s \in S$, y que en definitiva es la función que queremos maximizar. En el ámbito de la selección natural, la idoneidad de un individuo corresponde a su adaptación al medio; los mejor adaptados sobrevivirán con mayor probabilidad.

El punto de partida de un algoritmo genético es una población P formada por k individuos tomados al azar de S . A continuación, se evalúa la idoneidad de cada solución $s \in P$ utilizando f , y los individuos se ordenan de menor a mayor idoneidad (creciendo numéricamente si se desea maximizar f , decreciendo si se desea minimizarla).

Cada iteración de un algoritmo genético recibe el nombre de **generación**, nuevamente por analogía con los organismos vivos. En cada generación se toma la población existente, ordenada por idoneidad, y se genera una nueva población mediante el **cruce** de los individuos existentes dos a dos. Para emular la selección natural, los individuos más idóneos se reproducen con mayor probabilidad que los menos idóneos; de esta forma es más probable que sus genes se transmitan a la siguiente generación de individuos.

Hay diferentes métodos para cruzar los genes de dos individuos A y B : cortar por un gen determinado de A e insertar los genes de B a partir de ese punto (cruce en un punto, ver figura 53); o cortar por dos puntos de A e insertar los genes de B correspondientes (cruce en dos puntos, que es el método que usaremos en nuestro ejemplo, ver figura 54); también es posible cruzar al azar

*En inglés *fitness*

Observación

Dependiendo del problema y de la definición de f , el objetivo puede ser *minimizar* esta función. No obstante, en este subapartado supondremos que se desea maximizar f .

cada uno de los genes (cruce uniforme, ver figura 55), entre otros métodos. Con frecuencia el método de cruce ha de tener en cuenta la naturaleza del problema, pues puede ocurrir que sobre los genes operen algunas restricciones: deban estar ordenados, no deban repetirse, su suma no deba exceder una cierta cantidad, etc. El objetivo final del cruce es, emulando a la naturaleza, combinar las características ventajosas de dos individuos para conseguir un individuo mejor adaptado incluso a su entorno.

Figura 53. Cruce en un punto

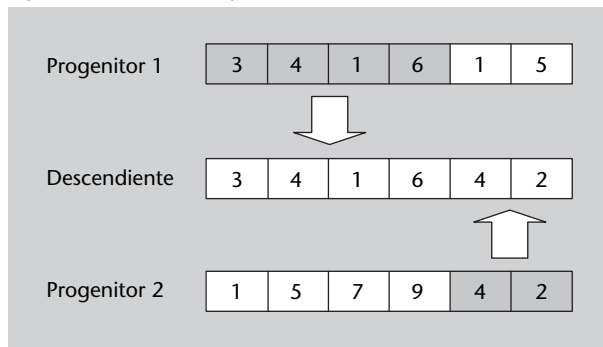


Figura 54. Cruce en dos puntos

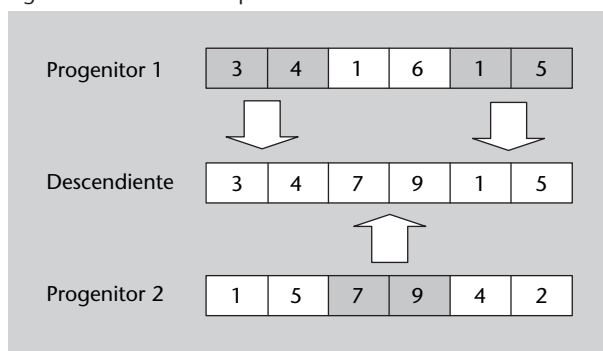
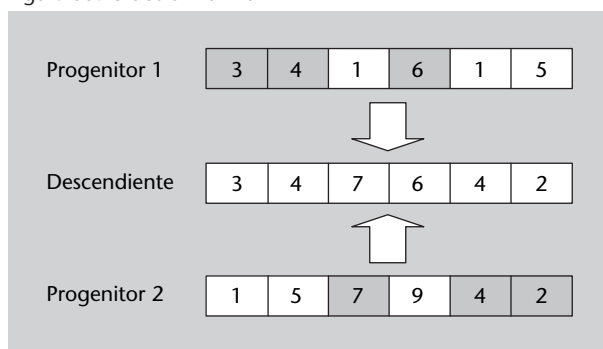


Figura 55. Cruce uniforme



El otro mecanismo que interviene en la evolución biológica, y que los algoritmos genéticos también utilizan, es el de la **mutación**. Los individuos que se han generado por cruce pueden sufrir, con una determinada probabilidad, un cambio aleatorio en alguno(s) de sus genes. El objetivo de las mutaciones es enriquecer la población introduciendo genes que no se encuentren en ella, lo que permite a su vez explorar nuevas zonas del espacio de soluciones S que

pueden contener soluciones mejores. La probabilidad o tasa de mutación no suele ser muy alta. En el ejemplo presentado a continuación es de un 1 %, a modo de orientación.

Seguidamente se evalúa la idoneidad de esta nueva población, que a su vez dará nacimiento a una nueva generación de individuos, y así sucesivamente. No existe un criterio único para establecer el número de generaciones (iteraciones) de un algoritmo genético; puede ser fijo, o hasta alcanzar una cierta idoneidad, o hasta que los nuevos individuos no aporten ninguna mejora durante algunas generaciones.

5.4.2. Ampliaciones y mejoras

Existen numerosas variantes y mejoras de los algoritmos genéticos. Una de las más habituales es la del **elitismo**: en cada generación, los E mejores individuos pasan inalterados a la siguiente generación. De esta forma el algoritmo no pierde soluciones que pueden ser mejores que la solución final, y además permite probar numerosos cruces y mutaciones de las mejores soluciones. Frecuentemente se toma $|E| = 1$, como en el ejemplo desarrollado más adelante.

Otras propuestas para mejorar los algoritmos genéticos incluyen el cruce entre más de dos individuos y los algoritmos **meméticos**, en los que los individuos tienen la capacidad de aprender y transmitir lo aprendido a sus descendientes.

5.4.3. Ejemplo de aplicación

A continuación diseñaremos y aplicaremos un algoritmo genético al problema expuesto antes del asesor de inversiones. En realidad se trata de una de las muchas variantes del conocido **problema de la mochila**.

Para hacernos una idea de la complejidad de este problema, supongamos que de cada tipo de bono existieran 10 unidades disponibles. Si tuviéramos 100 tipos de bonos distintos, el número de posibles combinaciones de inversión sería 10^{100} . Evidentemente, es imposible intentar una exploración exhaustiva de todas las combinaciones, por lo que se necesita un algoritmo que encuentre una combinación razonablemente buena en un tiempo aceptable.

En la tabla 36 se plantea un ejemplo de cinco tipos de bonos, en el que cada tipo tiene un precio, cantidad y rendimiento diferente. En la tabla 37 se puede seguir la ejecución del algoritmo durante dos generaciones, y de qué forma operan la selección de los mejores individuos, su cruce (en un punto) y las mutaciones. Además, en este algoritmo se mantiene siempre el mejor individuo de cada generación (elitismo). El último individuo de la tabla es el que mejor rendimiento proporciona de todos los obtenidos por el algoritmo.

Tabla 36. Tipos de bonos disponibles

	A	B	C	D	E
Precio	500	700	300	800	400
Cantidad	5	6	10	3	7
Rendimiento	0,15	0,05	0,03	0,1	0,1

Tabla 37. Ejecución del asesor de inversiones

Individuo	A	B	C	D	E	Capital	Rendimiento
Generación inicial							
1	1	1	2	3	1	4600	408
2	1	2	0	2	3	4700	425
3	4	3	1	0	1	4800	454
4	2	0	1	3	3	4900	519
Generación 1							
5 (4+2)	2	1	0	2	3	4500	465
6 (4+3)	4	0	1	2	2	4700	549
7 (3+2)	1	2	1	2	3	5000	434
4 (Elite)	2	0	1	3	3	4900	519
Mutación							
7 (3+2)	1	1	1	2	3	4300	399
5 (4+2)	2	1	0	2	3	4500	465
4 (Elite)	2	0	1	3	3	4900	519
6 (4+3)	4	0	1	2	2	4700	558
Generación 2							
8 (6+7)	4	0	0	2	3	4800	580
9 (6+5)	2	1	0	2	2	4100	425
10 (6+4)	3	0	1	2	3	4600	514
6(Elite)	4	0	1	2	2	4700	558
Mutación							
9 (6+5)	2	1	1	2	2	4400	434
10(6+4)	3	0	1	2	4	5000	554
6 (Elite)	4	0	1	2	2	4700	558
8 (6+7)	4	0	0	2	3	4800	580

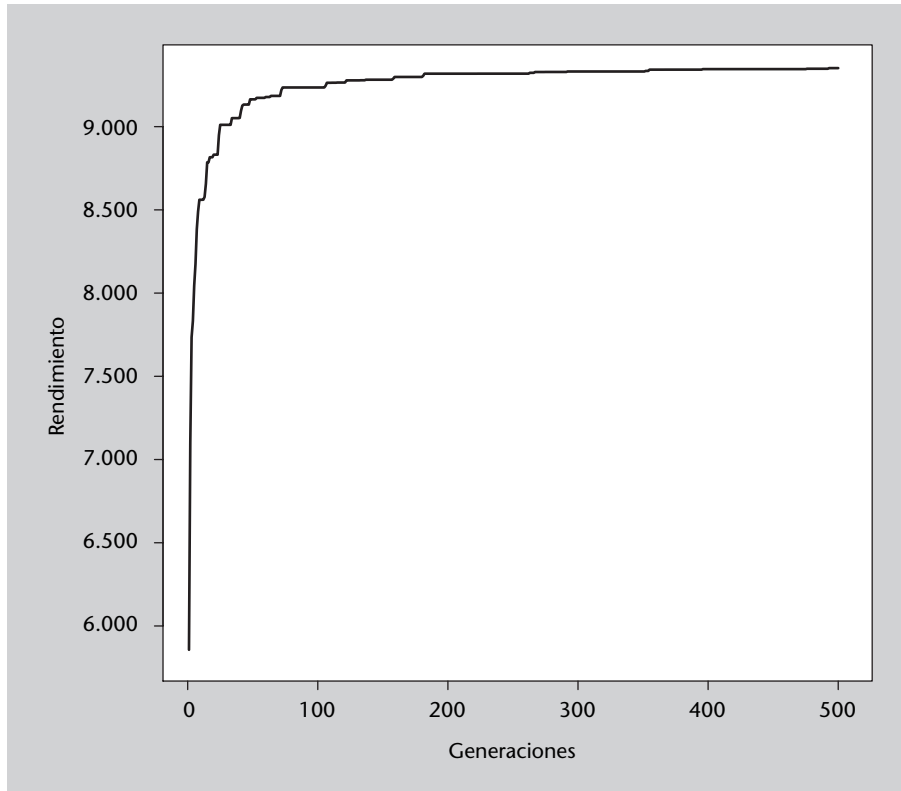
Algunas combinaciones pueden exceder el capital disponible; en tal caso, se eliminan algunas inversiones al azar (valores tachados).

5.4.4. Análisis del método

Los algoritmos genéticos son aplicables a gran número de problemas de optimización, y por lo general encuentran soluciones razonablemente buenas en un tiempo aceptable. Son, por tanto, una opción a considerar siempre que se plantee un problema de optimización.

En la figura 56 se muestra el rendimiento obtenido en el problema de las inversiones *online*, con 100 tipos de bonos y 20 individuos.

Figura 56. Efecto del número de generaciones en el rendimiento máximo obtenido



No obstante, adolecen de algunos problemas. El principal es que dependen de un gran número de parámetros y operaciones (número de individuos, tasa de mutación, estrategia de cruzamiento) y no hay reglas que indiquen claramente qué valores elegir para cada problema. Por tanto, hay que ajustar esos parámetros siguiendo recomendaciones muy generales.

Otra crítica que reciben los algoritmos genéticos es que, al proponer un gran número de soluciones al problema, se requiere calcular la función de idoneidad muchas veces; a menudo esa función es muy costosa en tiempo de computación, lo que hace que utilizar un algoritmo genético requiera mucho tiempo de ejecución.

5.4.5. Código fuente en Python

Código 5.2: Algoritmo genético del asesor de inversiones

```

1 from random import random, randint, sample
2 from collections import namedtuple
3
4 # Calcula el capital invertido por un individuo
5 def capitalInvertido(individuo):
6     return sum(map(lambda x,y: x*y.precio, individuo, inversiones))
7

```

```

8 # Calcula el rendimiento obtenido por un individuo
9 def rendimiento(individuo):
10     return sum(map(lambda x,y: x*y.precio*y.rendim,
11                  individuo, inversiones))
12
13
14 # Si un individuo gasta más capital del disponible, se eliminan
15 # aleatoriamente inversiones hasta que se ajusta al capital
16 def ajustaCapital(individuo):
17     ajustado = individuo[:]
18     while capitalInvertido(ajustado)>capital:
19         pos = randint(0,len(ajustado)-1)
20         if ajustado[pos] > 0:
21             ajustado[pos] -= 1
22
23     return ajustado
24
25
26 # Crea un individuo al azar, en este caso una selección de
27 # inversiones que no excedan el capital disponible
28 def creaIndividuo(inversiones, capital):
29     individuo = [0]*len(inversiones)
30
31     while capitalInvertido(individuo) < capital:
32         eleccion = randint(0,len(inversiones)-1)
33         individuo[eleccion] += 1
34
35     return ajustaCapital(individuo)
36
37
38 # Crea un nuevo individuo cruzando otros dos (cuyas posiciones se
39 # indican en el segundo parámetro)
40 def cruza(poblacion, posiciones):
41     L = len(poblacion[0])
42
43     # Toma los genes del primer progenitor y luego toma al azar
44     # un segmento de entre 1 y L genes del segundo progenitor
45     hijo = poblacion[posiciones[0]][:]
46     inicio = randint(0,L-1)
47     fin = randint(inicio+1,L)
48     hijo[inicio:fin] = poblacion[posiciones[1]][inicio:fin]
49
50     return ajustaCapital(hijo)
51
52
53 # Aplica mutaciones a un individuo según una tasa dada; garantiza
54 # que cumple las restricciones de capital e inversiones
55 def muta(individuo, tasaMutacion):
56     mutado = []
57     for i in range(len(individuo)):
58         if random() > tasaMutacion:
59             mutado.append(individuo[i])
60         else:
61             mutado.append(randint(0,inversiones[i].cantidad))
62
63     return ajustaCapital(mutado)
64
65
66 # Hace evolucionar el sistema durante un número de generaciones
67 def evolucionar(poblacion, generaciones):
68
69     # Ordena la población inicial por rendimiento producido
70     poblacion.sort(key=lambda x:rendimiento(x))
71
72     # Algunos valores útiles
73     N = len(poblacion)
74     tasaMutacion = 0.01
75
76     # Genera una lista del tipo [0,1,1,2,2,2,3,3,3,3,...] para
77     # representar las probabilidades de reproducirse de cada
78     # individuo (el primero 1 posibilidad, el segundo 2, etc.)
79     reproduccion = [x for x in range(N) for y in range(x+1)]

```



```

80
81 for i in range(generaciones):
82     # Se generan N-1 nuevos individuos cruzando los existentes
83     # (sin que se repitan los padres)
84     padres = sample(reproduccion,2)
85     while padres[0]==padres[1]:
86         padres = sample(reproduccion,2)
87     hijos = [cruza(poblacion, padres) for x in range(N-1)]
88
89     # Se aplican mutaciones con una cierta probabilidad
90     hijos = [muta(x, tasaMutacion) for x in hijos]
91
92     # Se añade el mejor individuo de la población anterior
93     # (elitismo)
94     hijos.append(poblacion[-1])
95     poblacion = hijos
96
97     # Se ordenan los individuos por rendimiento
98     poblacion.sort(key=lambda x:rendimiento(x))
99
100
101     # Devuelve el mejor individuo encontrado
102     return poblacion[-1]
103
104
105 # Declara una tupla con nombres para representar cada inversión
106 Inversion = namedtuple('Inversion', 'precio_cantidad_rendim')
107
108 numInver = 100
109 maxPrecio = 1000
110 maxCant = 10
111 maxRend = 0.2
112
113
114 # Genera una lista de tuplas Inversion
115 inversiones=[Inversion(random()*maxPrecio, randint(1,maxCant),
116                      random()*maxRend) for i in range(numInver)]
117 print (inversiones)
118
119 capital = 50000
120 individuos = 20
121 generaciones = 1000
122
123 poblacion = [creaIndividuo(inversiones, capital)
124             for i in range(individuos)]
125
126 # Nota: para simplificar el programa se accede a inversiones y
127 # capital de forma global (sólo se leen, no se modifican)
128
129 mejor = evoluciona(poblacion, generaciones)
130 print(mejor, capitalInvertido(mejor), rendimiento(mejor))

```

5.5. Colonias de hormigas

El **problema del viajante de comercio** es un problema clásico de optimización en el que se dispone de un conjunto de ciudades, con una distancia entre cada par de ciudades. El viajante al que se refiere el problema debe visitar todas las ciudades una y sólo una vez, intentando por su parte recorrer la mínima distancia (para gastar menos tiempo, combustible, etc.).

El algoritmo de optimización mediante colonias de hormigas* es un algoritmo probabilístico que imita la habilidad de las hormigas para encontrar el camino

* En inglés, *ant colony optimization*.

más corto desde su hormiguero hasta una fuente de alimento. Las hormigas son un excelente ejemplo de sistema emergente, dado que cada hormiga individual carece de la inteligencia suficiente para encontrar el camino más corto hasta una fuente de alimento, y sin embargo su comportamiento coordinado lo consigue.

La forma en que las hormigas consiguen encontrar el camino más corto es la siguiente: en principio las hormigas vagabundean al azar alrededor de su hormiguero, y cuando encuentran alimento toman un poco y vuelven a su hormiguero dejando un rastro de feromonas. Si otras hormigas encuentran ese rastro, es probable que dejen de vagabundear al azar y lo sigan, ya que supuestamente conduce a una fuente de alimento. A su vez, las hormigas que vuelven con alimento dejan su propio rastro de feromona, reforzando así ese camino. Si hay varios caminos hacia una misma fuente de alimento el más corto acabará siendo el preferido por las hormigas, por la sencilla razón de que al ser más corto lo recorrerán más hormigas por unidad de tiempo, y por tanto la intensidad de su rastro de feromonas será mayor. Por otra parte, las feromonas se evaporan gradualmente, con lo que los caminos que no se utilizan van perdiendo atractivo. De esta forma, partiendo de una exploración aleatoria se consigue encontrar un camino óptimo o cercano al óptimo.

Sistemas emergentes

Los sistemas emergentes son aquellos que resuelven problemas complejos y exhiben un elevado grado de inteligencia combinando subsistemas sencillos y poco inteligentes. Las neuronas son quizá el ejemplo más claro.

5.5.1. Descripción del método

La adaptación de este comportamiento a un algoritmo de optimización es relativamente directa*: se genera una “hormiga” que en principio explora el espacio del problema de forma aleatoria, y cuando encuentra una solución al problema, la marca positivamente con “feromonas”, de manera que las siguientes hormigas seguirán ese camino con una probabilidad más alta, si bien manteniendo un cierto componente aleatorio.

* Fue propuesta por primera vez por M. Dorigo en su tesis doctoral: *Optimization, Learning and Natural Algorithms*, PhD thesis, Politecnico di Milano, Italia, 1992.

La estructura general del algoritmo es muy sencilla: en cada iteración se genera una “hormiga”, que propone una solución al problema. El criterio de finalización puede ser tan sencillo como alcanzar un número de iteraciones predeterminado, o puede ser más elaborado, como alcanzar una solución de cierta calidad o comprobar que sucesivas iteraciones no mejoran la solución obtenida.

Las soluciones parciales del problema se denominan **estados de la solución**. Para generar una solución completa hay que partir del estado inicial e ir moviéndose a un nuevo estado de la solución hasta completar la solución; la naturaleza específica de cada problema determina cuál es una solución completa en cada caso. En cada momento, la probabilidad P_{xy} de que una hormiga pase de un estado x a un estado y , depende de dos factores: la facilidad ϕ_{xy}

para pasar del estado x al y , y el rastro de feromonas ρ_{xy} que en ese momento está asociado a esa transición. Si E es el conjunto de estados de la solución, entonces la probabilidad de elegir un nuevo estado viene determinada por

$$P_{xy} = \frac{(1 + \rho_{xy}^\alpha)\phi_{xy}^\beta}{\sum_{i \in E - \{x\}} (1 + \rho_{xi}^\alpha)\phi_{xi}^\beta} \quad (57)$$

donde el parámetro $\alpha \geq 0$ controla la influencia del rastro de feromonas ρ_{xy} y el parámetro $0 \leq \beta \leq 1$ controla la influencia de la facilidad de transición ϕ_{xy} . En otras palabras, la probabilidad de cambiar a un estado y se calcula comparando la facilidad y nivel de feromonas del camino xy con las del resto de caminos disponibles desde x .

Cada vez que una hormiga completa una solución es necesario actualizar el rastro de feromonas entre estados de la solución. Se aplican dos operaciones: por una parte, las feromonas tienden a evaporarse con el tiempo; por otra, la hormiga deja un rastro de feromonas por el camino que ha seguido. Así, la cantidad de feromona asociada a la transición entre los estados x e y viene dada por:

$$\rho_{xy} = (1 - \gamma)\rho_{xy} + \Delta\rho_{xy} \quad (58)$$

donde $0 \leq \gamma \leq 1$ es el coeficiente de evaporación de las feromonas y $\Delta\rho_{xy}$ es el rastro dejado por la hormiga en la transición de x a y , típicamente cero cuando la hormiga no ha utilizado esa transición en su solución, y un valor constante cuando la transición sí que forma parte de la solución.

Una mejora habitual de este algoritmo consiste en permitir que las hormigas dejen un rastro de feromonas sólo cuando han encontrado un camino mejor que el mejor camino hasta ese momento, de manera que se premie el hallazgo de caminos mejores y se reduzca el número de iteraciones necesario para encontrarlos.

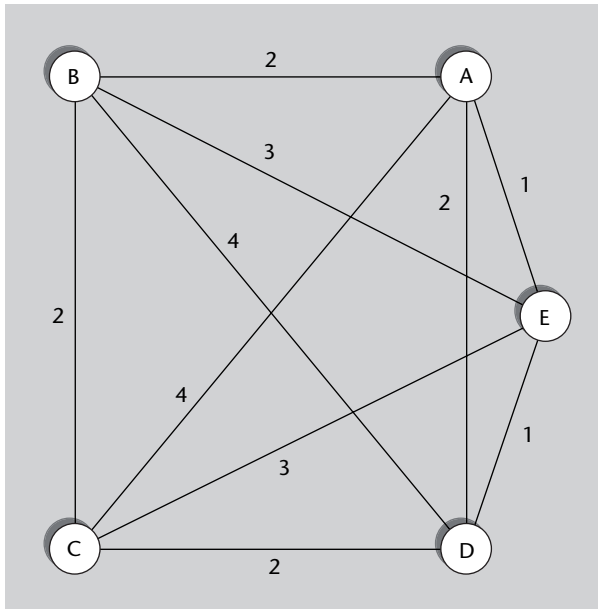
5.5.2. Ejemplo de aplicación

Las aplicaciones más evidentes de este algoritmo son aquellas en las que se busca el camino más corto entre dos puntos, si bien se han aplicado con éxito a tareas tan diversas como planificación, clasificación, partición de conjuntos, procesamiento de imágenes y plegado de proteínas.

Como ejemplo de aplicación de la optimización con colonias de hormigas resolveremos el problema del viajante de comercio descrito anteriormente. Podemos formalizar el problema representándolo como un grafo en el que cada ciudad es un nodo y los arcos están etiquetados con la distancia de una

ciudad a otra. En este ejemplo vamos a suponer que la ciudad de origen es la A, y que el viajante debe regresar a ella al final de su trayecto. En la figura 57 podemos ver un ejemplo de este problema, en el que los valores de los arcos representan las distancias entre ciudades.

Figura 57. Problema ejemplo del viajante de comercio



Para resolver este problema mediante el algoritmo de la colonia de hormigas generaremos sucesivas hormigas. La primera de ellas recorrerá las ciudades en un orden aleatorio, si bien será más probable que viaje a las ciudades más cercanas, una restricción lógica si intentamos simular el comportamiento de una hormiga real. Así, la facilidad de ir de un nodo x a un nodo y será $\phi_{xy} = 1/dist(x,y)$. Una solución de este problema será una secuencia de ciudades en la que cada ciudad aparezca una y sólo una vez. La solución óptima será aquella que minimice la distancia total recorrida por la hormiga, incluyendo la vuelta a la ciudad de partida tras recorrer las restantes ciudades.

El criterio de finalización empleado en este ejemplo es muy sencillo: un número de iteraciones (hormigas) predeterminado, exactamente 3 por no alargar excesivamente el ejemplo. Tal y como se ha explicado anteriormente, es posible utilizar un criterio más sofisticado. El ejemplo de ejecución no pretende mostrar las ventajas del algoritmo, pues hacerlo requeriría resolver un problema con gran número de ciudades y de iteraciones, sino simplemente ilustrar el funcionamiento básico del algoritmo; su verdadera potencia se aprecia en problemas mayores.

Ejecución del algoritmo

La tabla 38 muestra el peso inicial de las transiciones entre los estados (ciudades) del problema propuesto en la figura 57. Se puede observar que, inicialmente, no hay rastros de feromonas; por tanto, la probabilidad de que una

hormiga tome una transición determinada es inversamente proporcional a la distancia a esa ciudad, tal y como se ha propuesto antes. Nótese también que en este ejemplo no se distinguen dos sentidos en las transiciones entre una ciudad y otra, esa es la razón por la que sólo se muestra la mitad superior de la tabla.

Tabla 38. Estado inicial del problema del viajante

	B	C	D	E
A	$(1 + 0)^{\frac{1}{2}} = 0,5$	$(1 + 0)^{\frac{1}{4}} = 0,25$	$(1 + 0)^{\frac{1}{2}} = 0,5$	$(1 + 0)^1 = 1$
B		$(1 + 0)^{\frac{1}{2}} = 0,5$	$(1 + 0)^{\frac{1}{4}} = 0,25$	$(1 + 0)^{\frac{1}{3}} = 0,33$
C			$(1 + 0)^{\frac{1}{2}} = 0,5$	$(1 + 0)^{\frac{1}{3}} = 0,33$
D				$(1 + 0)^1 = 1$

Valor $(1 + \rho_{xy})\phi_{xy}$ para ir de la ciudad x a la ciudad y

En la primera iteración del algoritmo, una primera hormiga recorre las ciudades en un orden aleatorio, si bien condicionado por las probabilidades de cada transición.

Por tanto, esta primera hormiga ejecuta un algoritmo ávido-aleatorio, en el que elige el camino aleatoriamente, si bien las transiciones más atractivas en cada momento se eligen con mayor probabilidad, aunque a largo plazo se trate de malas decisiones.

La ruta seguida por la hormiga se marcará con feromonas, haciendo que las siguientes hormigas sigan esas transiciones con mayor probabilidad. En este caso la ruta seguida por la hormiga ha sido $A \rightarrow E \rightarrow D \rightarrow B \rightarrow C \rightarrow A$, con una longitud total de $1 + 1 + 4 + 2 + 4 = 12$.

Por otra parte, al terminar la primera iteración las feromonas sufren una cierta evaporación de acuerdo con la ecuación 58. En este ejemplo tomaremos $\gamma = 0,3$, un valor relativamente alto dado que en este ejemplo el número de iteraciones es muy reducido; por regla general conviene tomar un valor menor, por ejemplo 0,1. Toda esa información se muestra en la tabla 39.

Tabla 39. Problema del viajante tras la primera iteración

	B	C	D	E
A	$(1 + 0)^{\frac{1}{2}} = 0,5$	$(1 + 0,7)^{\frac{1}{4}} = 0,425$	$(1 + 0)^{\frac{1}{2}} = 0,5$	$(1 + 0,7)^1 = 1,7$
B		$(1 + 0,7)^{\frac{1}{2}} = 0,85$	$(1 + 0,7)^{\frac{1}{4}} = 0,425$	$(1 + 0)^{\frac{1}{3}} = 0,33$
C			$(1 + 0)^{\frac{1}{2}} = 0,5$	$(1 + 0)^{\frac{1}{3}} = 0,33$
D				$(1 + 0,7)^1 = 1,7$

De esta forma, la siguiente hormiga se encuentra unas probabilidades diferentes de tomar cada transición, guiada por el recorrido de la hormiga anterior. La segunda hormiga sigue el camino $A \rightarrow E \rightarrow C \rightarrow D \rightarrow B \rightarrow A$, con una longitud de $1 + 3 + 2 + 4 + 2 = 12$; al no mejorar la solución anterior, esta hormiga no deja feromonas, aunque las feromonas existentes sufren una cierta evaporación. En la tabla 40 se puede ver el estado de las transiciones tras la segunda iteración.

Tabla 40. Problema del viajante tras la segunda iteración

	B	C	D	E
A	$(1 + 0)\frac{1}{2} = 0,5$	$(1 + 0,49)\frac{1}{4} = 0,373$	$(1 + 0)\frac{1}{2} = 0,5$	$(1 + 0,49)1 = 1,49$
B		$(1 + 0,49)\frac{1}{2} = 0,745$	$(1 + 0,49)\frac{1}{4} = 0,373$	$(1 + 0)\frac{1}{3} = 0,33$
C			$(1 + 0)\frac{1}{2} = 0,5$	$(1 + 0)\frac{1}{3} = 0,33$
D				$(1 + 0,49)1 = 1,49$

La última hormiga sigue el camino $A \rightarrow E \rightarrow D \rightarrow C \rightarrow B \rightarrow A$, con una longitud de $1 + 1 + 2 + 2 + 2 = 8$, más corto que el mejor camino encontrado hasta ahora; de hecho, es fácil comprobar que es el camino óptimo para este problema (como el camino inverso, obviamente). En este caso la hormiga sí que deja un rastro de feromonas, que sería utilizado por las hormigas siguientes si las hubiera. En la tabla 41 puede verse el estado final de las transiciones.

Tabla 41. Problema del viajante tras la tercera iteración

	B	C	D	E
A	$(1 + 0,7)\frac{1}{2} = 0,85$	$(1 + 0,49)\frac{1}{4} = 0,373$	$(1 + 0)\frac{1}{2} = 0,5$	$(1 + 1,043)1 = 2,043$
B		$(1 + 1,043)\frac{1}{2} = 1,022$	$(1 + 0,49)\frac{1}{4} = 0,373$	$(1 + 0)\frac{1}{3} = 0,33$
C			$(1 + 0,7)\frac{1}{2} = 0,85$	$(1 + 0)\frac{1}{3} = 0,33$
D				$(1 + 1,043)1 = 2,043$

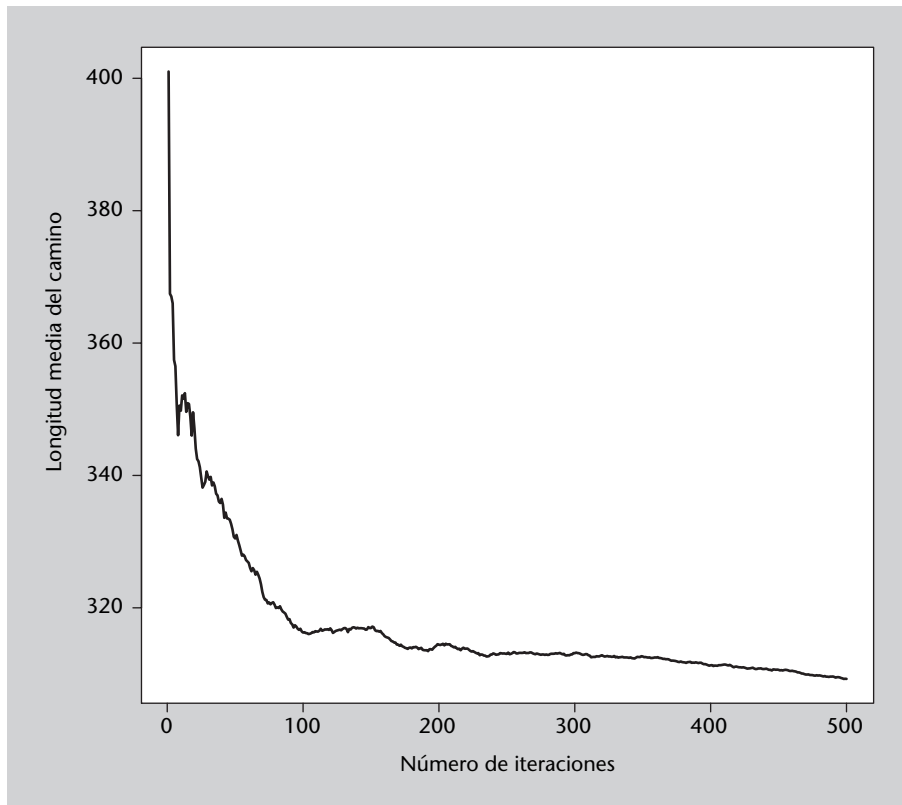
5.5.3. Análisis del método

Este método de optimización tiene numerosas ventajas, pues permite encontrar soluciones razonablemente buenas a problemas muy complejos mejorando progresivamente soluciones aleatorias. Además por lo general evita los mínimos locales. Su aplicabilidad es bastante general, pues se puede aplicar a cualquier problema que pueda traducirse a la búsqueda de un camino óptimo en un grafo. Sin embargo, en algunos casos esa traducción puede ser difícil o muy costosa.

Quizá el principal problema de este algoritmo radique en la gran cantidad de parámetros que hay que elegir: pesos α y β en el cálculo de probabilidades, coeficiente de evaporación γ , etc.

Para ilustrar el efecto del número de iteraciones en la longitud del camino obtenido, en la figura 58 se muestra la evolución de la longitud media de los caminos seguidos por las hormigas frente al número de iteraciones (hormigas). En este ejemplo se han tomado 100 ciudades con una distancia máxima entre ellas de 10. Se puede observar que la longitud de los caminos baja rápidamente y se estabiliza a medida que se incrementan las iteraciones. Para valorar la eficiencia del algoritmo, hay que tener en cuenta que el número de recorridos posibles en este problema es $n!$, siendo n el número de ciudades; así pues, para 100 ciudades el número de recorridos posibles es $100!$, o sea más de 10^{157} recorridos posibles. Evidentemente, una exploración exhaustiva de todos ellos es impracticable.

Figura 58. Efecto del número de iteraciones en la longitud media del camino



5.5.4. Código fuente en Python

Código 5.3: Algoritmo de la colonia de hormigas

```

1 import random, sys, math
2
3 # Nota: en lugar de matrices se usan listas de listas
4
5 # Genera una matriz de distancias de nCiudades x nCiudades
6 def matrizDistancias(nCiud, distanciaMaxima):
7     matriz = [[0 for i in range(nCiud)] for j in range(nCiud)]
8
9     for i in range(nCiud):
10        for j in range(i):
11            matriz[i][j] = distanciaMaxima*random.random()
12            matriz[j][i] = matriz[i][j]
13
14        return matriz
15
16 # Elige un paso de una hormiga, teniendo en cuenta las distancias
17 # y las feromonas y descartando las ciudades ya visitadas.
18 def eligeCiudad(dists, ferom, visitadas):
19     # Se calcula la tabla de pesos de cada ciudad
20     listaPesos = []
21     disponibles = []
22     actual = visitadas[-1]
23
24     # Influencia de cada valor (alfa: feromonas; beta: distancias)
25     alfa = 1.0
26     beta = 0.5
27
28     # El parámetro beta (peso de las distancias) es 0.5, alfa=1.0
29     for i in range(len(dists)):
30         if i not in visitadas:
31             fer = math.pow((1.0 + ferom[actual][i]), alfa)
32             peso = math.pow(1.0/dists[actual][i], beta) * fer

```

```

33     disponibles.append(i)
34     listaPesos.append(peso)
35
36     # Se elige aleatoriamente una de las ciudades disponibles ,
37     # teniendo en cuenta su peso relativo .
38     valor = random.random() * sum(listaPesos)
39     acumulado = 0.0
40     i = -1
41     while valor > acumulado:
42         i += 1
43         acumulado += listaPesos[i]
44
45     return disponibles[i]
46
47
48 # Genera una "hormiga", que elegirá un camino teniendo en cuenta
49 # las distancias y los rastros de feromonas. Devuelve una tupla
50 # con el camino y su longitud.
51 def eligeCamino(distancias, feromonas):
52     # La ciudad inicial siempre es la 0
53     camino = [0]
54     longCamino = 0
55
56     # Elegir cada paso según la distancia y las feromonas
57     while len(camino) < len(distancias):
58         ciudad = eligeCiudad(distancias, feromonas, camino)
59         longCamino += distancias[camino[-1]][ciudad]
60         camino.append(ciudad)
61
62     # Para terminar hay que volver a la ciudad de origen (0)
63     longCamino += distancias[camino[-1]][0]
64     camino.append(0)
65
66     return (camino, longCamino)
67
68 # Actualiza la matriz de feromonas siguiendo el camino recibido
69 def rastrosFeromonas(feromonas, camino, dosis):
70     for i in range(len(camino) - 1):
71         feromonas[camino[i]][camino[i+1]] += dosis
72
73 # Evapora todas las feromonas multiplicándolas por una constante
74 # = 0.9 (en otras palabras, el coeficiente de evaporación es 0.1)
75 def evaporaFeromonas(feromonas):
76     for lista in feromonas:
77         for i in range(len(lista)):
78             lista[i] *= 0.9
79
80 # Resuelve el problema del viajante de comercio mediante el
81 # algoritmo de la colonia de hormigas. Recibe una matriz de
82 # distancias y devuelve una tupla con el mejor camino que ha
83 # obtenido (lista de índices) y su longitud
84 def hormigas(distancias, iteraciones, distMedia):
85     # Primero se crea una matriz de feromonas vacía
86     n = len(distancias)
87     feromonas = [[0 for i in range(n)] for j in range(n)]
88
89     # El mejor camino y su longitud (inicialmente "infinita")
90     mejorCamino = []
91     longMejorCamino = sys.maxint
92
93     # En cada iteración se genera una hormiga, que elige un camino,
94     # y si es mejor que el mejor que teníamos, deja su rastro de
95     # feromonas (mayor cuanto más corto sea el camino)
96     for iter in range(iteraciones):
97         (camino, longCamino) = eligeCamino(distancias, feromonas)
98
99         if longCamino <= longMejorCamino:
100             mejorCamino = camino
101             longMejorCamino = longCamino
102
103     rastrosFeromonas(feromonas, camino, distMedia/longCamino)
104

```



```

105     # En cualquier caso, las feromonas se van evaporando
106     evaporaFeromonas(feromonas)
107
108
109     # Se devuelve el mejor camino que se haya encontrado
110     return (mejorCamino, longMejorCamino)
111
112
113 # Generación de una matriz de prueba
114 numCiudades     = 10
115 distanciaMaxima = 10
116 ciudades        = matrizDistancias(numCiudades, distanciaMaxima)
117
118 # Obtención del mejor camino
119 iteraciones     = 1000
120 distMedia       = numCiudades*distanciaMaxima/2
121 (camino, longCamino) = hormigas(ciudades, iteraciones, distMedia)
122 print("Camino:_" , camino)
123 print("Longitud_"del_camino:_" , longCamino)

```

5.6. Optimización con enjambres de partículas

Supongamos que se nos plantea buscar el mínimo (o máximo) de una función matemática cualquiera, como por ejemplo

$$f(x,y) = x^2(4 - 2,1x^2 + \frac{x^4}{3}) + xy + y^2(-4 + 4y^2) \quad (59)$$

dentro de un intervalo determinado, en este caso $[(-10, -10), (10, 10)]$. Se trata de un ejemplo puramente ilustrativo, pues f es una función que se puede derivar fácilmente y por tanto, es posible encontrar los mínimos de forma analítica. Concretamente, hay dos mínimos globales, en $(0, 0,9, -0,71)$ y en $(-0,09, 0,71)$, en los que la función toma el valor de $-1,0316$.

El método de optimización con enjambres de partículas* debe su nombre a que utiliza un conjunto de soluciones candidatas que se comportan como partículas moviéndose por un espacio: el espacio de soluciones. El movimiento de las partículas se determina por su velocidad, que depende de la posición del mejor punto que ha encontrado cada una de ellas en su recorrido y del mejor punto encontrado globalmente por el enjambre.

* En inglés, *particle swarm optimization*.

Vale la pena mencionar el origen de este método, pues curiosamente surgió a partir de un trabajo que simulaba el comportamiento de entidades sociales como rebaños de animales. Analizando el comportamiento de esa simulación, y simplificando un poco las reglas que la gobernaban, se observó que las entidades tendían a buscar el punto mínimo del espacio en el que se encontraban.

Lectura complementaria

El método de optimización con enjambres de partículas se describió en J. Kennedy; R. Eberhart (1995). "Particle Swarm Optimization". IEEE International Conference on Neural Networks

5.6.1. Descripción del método

Suponed que queremos encontrar el punto mínimo de una función $f : \mathbb{R}^n \rightarrow \mathbb{R}$, donde el espacio de soluciones $S \subset \mathbb{R}^n$ está acotado por los puntos $inf, sup \in \mathbb{R}^n$. El intervalo del espacio de soluciones S no tiene por qué ser el mismo en todas las dimensiones.

Primer paso

El primer paso del método de optimización con enjambres de partículas consiste en crear un conjunto de partículas (el enjambre) al que llamaremos E . A cada partícula $p_i \in E$ se le asigna una **posición inicial** x_i en S según la fórmula:

$$x_i \leftarrow U(inf, sup) \quad (60)$$

Donde $U : \mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}^n$ es una función que devuelve un número aleatorio con distribución uniforme dentro del subespacio de \mathbb{R}^n delimitado por los dos puntos que recibe como parámetro.

Además cada partícula tiene una **velocidad inicial** v_i que viene dada por la fórmula:

$$v_i \propto U(-|sup - inf|, |sup - inf|) \quad (61)$$

Es decir, la velocidad inicial en cada dimensión es proporcional al intervalo del espacio de soluciones en esa dimensión. Habitualmente, la velocidad inicial se multiplica por un factor $0 < k < 1$ para evitar que una velocidad inicial excesiva haga que las partículas se salgan de S en pocas iteraciones.

Por último, cada partícula recuerda el **mejor punto** m_i que ha visitado; inicialmente $m_i = x_i$, pues es el único punto que ha visitado cada partícula. También es necesario almacenar el mejor punto visitado por el enjambre, g , es decir el **mínimo global** obtenido hasta un momento dado:

$$g \leftarrow \underset{m_i}{\operatorname{argmin}} f(m_i) \quad (62)$$

Iteraciones del algoritmo

Como es habitual en este tipo de métodos, no hay un número predefinido de iteraciones; el usuario del método deberá elegir. Las opciones más habituales son: número fijo de iteraciones, repetir hasta alcanzar un valor de f suficientemente bueno, hasta que no se observen mejoras en g o hasta que transcurra un tiempo determinado.

En cada iteración las partículas cambian su velocidad influidas por su mejor punto m_i y por el mejor punto global en ese momento, g . En primer lugar se calcula la nueva velocidad de cada partícula del enjambre $i = 1, \dots, |E|$ según la expresión:

$$v_i \leftarrow \alpha v_i + \beta_c a_c (m_i - x_i) + \beta_s a_s (g - x_i) \quad (63)$$

Los parámetros y valores que aparecen en la expresión anterior son:

- α es un coeficiente que regula la **inercia** de la partícula, esto es, su tendencia a seguir con su trayectoria e ignorar la influencia de otros elementos.
- β_c (c por **cognitiva**) regula la influencia del aspecto cognitivo de la propia partícula, es decir, la influencia del mejor punto que ella recuerda en el cálculo de su nueva velocidad.
- β_s (s por **social**) regula la influencia del aspecto social sobre la partícula, es decir, la influencia del mejor punto encontrado por el enjambre. Cuanto mayor sea, más tenderán las partículas a moverse hacia el mejor punto, g , y por tanto la influencia de lo social, del enjambre, será mayor.
- a_c y a_s son dos números aleatorios calculados mediante $U(0,1)$ que asignan una influencia aleatoria a los aspectos cognitivos y sociales respectivamente.

La nueva posición de cada partícula depende de su posición actual y de su nueva velocidad:

$$x_i \leftarrow x_i + v_i \quad (64)$$

Si la nueva posición de la partícula es una solución mejor que la mejor solución que recordaba, es decir, si $f(x_i) < f(m_i)$ entonces x_i pasa a ser la nueva mejor posición de esa partícula: $m_i \leftarrow x_i$.

Tras actualizar todas las partículas, se comparan sus mejores posiciones m_i con la mejor posición global g ; si alguna $m_i < g$, entonces $g \leftarrow m_i$.

Al acabar las iteraciones, g contiene la mejor solución encontrada por el algoritmo.

Selección de los parámetros α , β_c y β_s

En la expresión 63 aparecen tres parámetros que se utilizan para calcular la nueva velocidad de una partícula. Estos parámetros permiten ajustar la in-

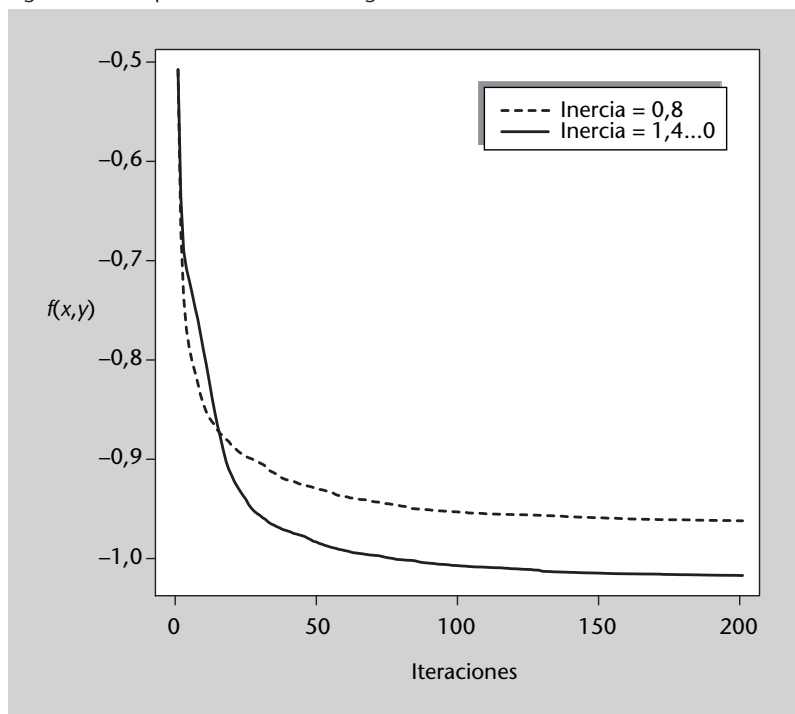
fluencia de la inercia de la partícula (α), del mejor punto que ella conoce (β_c) y del mejor punto encontrado por el enjambre (β_s). Ahora bien, ¿qué valor deben tomar? Diferentes trabajos empíricos* recomiendan que tanto β_c como β_s tengan un valor de 2,0, mientras que la inercia debe influir menos ($\alpha = 0,8$).

*Destacamos Y. Shi; R. C. Eberhart (1998). "Parameter selection in particle swarm optimization". Proceedings of Evolutionary Programming VII.

Una mejora del método consiste en empezar con una inercia relativamente alta ($\alpha = 1,4$) que se va reduciendo en cada iteración, por ejemplo, multiplicándola por un factor $r < 1$. El sentido de esta mejora es el siguiente. En las primeras iteraciones las partículas se mueven a mayor velocidad debido a su elevada inercia; con ello se consigue explorar más exhaustivamente el espacio de soluciones S y evitar así quedar atrapado en mínimos locales. Sin embargo, a medida que la ejecución del algoritmo avanza, interesa pulir las soluciones encontradas, llevando a cabo una exploración más fina de su entorno. Para conseguirlo las partículas se deben mover lentamente y explorar los alrededores de los mejores puntos encontrados, lo que se consigue haciendo que su inercia sea menor.

Esta mejora es bastante habitual y suele mejorar los resultados de los algoritmos de optimización con enjambres de partículas. En la figura 59 se compara la ejecución del algoritmo básico (inercia constante) y del algoritmo mejorado (inercia decreciente). En este caso, se muestra el valor medio de $f(x,y)$ de 10 partículas durante 200 iteraciones. Se ha elegido un número relativamente pequeño de partículas para que la obtención de puntos mínimos no se deba al azar inicial.

Figura 59. Comparación de las estrategias de inercia constante e inercia decreciente



Como se puede observar, con inercia constante el algoritmo empieza mejorando antes los resultados (minimizando $f(x,y)$), si bien llega un punto en el que

se estanca porque no es capaz de hacer una exploración más fina para seguir mejorando la posición de las partículas. Por su parte, con inercia decreciente el algoritmo empieza a optimizar más lentamente que con inercia constante, pero a medida que la ejecución avanza sigue mejorando los resultados ya que las partículas, ahora más lentas, pueden explorar con mayor precisión el área alrededor del mínimo global. Por esta razón, el código de ejemplo del subapartado 5.6.4 hace uso de inercia decreciente.

El valor del factor r depende del número de iteraciones previstas, y conviene ajustarlo de manera que hacia las últimas iteraciones la inercia tienda a cero, pero no antes. Por ejemplo, con 100 iteraciones se tiene que $0,9^{100} = 0,0000265$, luego $r = 0,9$ es adecuado si el número de iteraciones es 100, pero no si es por ejemplo 1000, ya que en ese caso a partir de la iteración 100 la inercia queda prácticamente desestimada. En ese caso sería mejor tomar $r = 0,99$, por ejemplo, ya que en la iteración 100 la inercia sigue teniendo un peso razonable, $0,99^{100} = 0,366$.

5.6.2. Ejemplo de aplicación

A modo de ejemplo se va a utilizar la optimización con enjambres de partículas para buscar el mínimo de la función 59. Se trata de un problema muy sencillo, con un espacio de soluciones muy “pequeño” (sólo dos dimensiones y unos intervalos bastante reducidos), por lo que una simple exploración al azar puede dar resultados aceptables. Sin embargo, se ha elegido así expresamente, pues permite mostrar la traza de ejecución completa, en este caso con cuatro partículas y tres iteraciones.

En la tabla 42 se puede ver, para cada iteración, cómo las partículas se van moviendo hacia la solución y van mejorando sus posiciones. Así, con tan solo cuatro partículas y tres iteraciones, el mínimo obtenido mejora claramente, pasando de $-0,0372$ a $-0,927$, bastante cercano al mínimo global que ya conocemos.

5.6.3. Análisis del método

El método de optimización con enjambres de partículas es un método adecuado a gran cantidad de problemas. Si bien el ejemplo planteado en el subapartado 5.6.2 permite comprobar paso a paso la ejecución del algoritmo en un caso sencillo, la potencia de este método se hace patente en problemas con mayor número de dimensiones, en los que la búsqueda al azar resulta infructuosa. El método ofrece las siguientes ventajas:

- Número relativamente reducido de parámetros. En muchos casos es suficiente con elegir un número de partículas, pues el método suele funcionar

Tabla 42. Ejecución del enjambre de partículas

Partícula	Posición	Velocidad	$f(x,y)$	Mejor posición	Mejor $f(x,y)$
Enjambre inicial					
1	(-1,463, 0,695)	(-0,236, -0,745)	0,195	(-1,463, 0,695)	0,195
2	(-0,0183, -0,101)	(-0,348, -0,211)	-0,0372	(-0,0183, -0,101)	-0,0372
3	(-1,625, -0,943)	(-0,164, -0,567)	3,197	(-1,625, -0,943)	3,197
4	(1,0491, -0,996)	(-0,555, -0,278)	1,225	(1,0491, -0,996)	1,225
g	(-0,0183, -0,101)		-0,0372		
Iteración 1					
1	(1,079, 0,182)	(2,542, -0,513)	2,405	(-1,463, 0,695)	0,195
2	(-0,297, -0,333)	(-0,279, -0,232)	-0,101	(-0,297, -0,333)	-0,101
3	(-0,400, -0,722)	(1,225, 0,222)	-0,123	(-0,400, -0,722)	-0,123
4	(-0,453, -1)	(-1,502, -0,754)	1,188	(-0,453, -1)	1,188
g	(-0,400, -0,722)		-0,123		
Iteración 2					
1	(0,641, -0,00817)	(-0,438, -0,190)	1,306	(-1,463, 0,695)	0,195
2	(-0,168, -0,379)	(0,129, -0,0454)	-0,317	(-0,168, -0,379)	-0,317
3	(0,580, -0,758)	(0,980, -0,0367)	-0,297	(0,580, -0,758)	-0,297
4	(-1,579, -1)	(-1,126, -0,830)	3,664	(-0,453, -1)	1,188
g	(-0,168, -0,379)		-0,317		
Iteración 3					
1	(-2, -0,932)	(-4,926, -0,924)	5,139	(-1,463, 0,695)	0,195
2	(-0,0640, -0,663)	(0,104, -0,285)	-0,927	(-0,0640, -0,663)	-0,927
3	(1,000847, -1)	(0,421, -0,824)	1,234	(0,580, -0,758)	-0,297
4	(-0,541, -0,0443)	(1,0381, 0,956)	1,0153	(-0,541, -0,0443)	1,0153
g	(-0,0640, -0,663)		-0,927		

Las coordenadas expresan (x,y) . g es la mejor posición global.

bien asignando a los parámetros los valores $\alpha = 1,4..,0$, $\beta_c = 2,0$ y $\beta_s = 2,0$, tal y como se ha explicado más arriba.

- Adecuado para espacios de soluciones continuos.
- Al haber muchas partículas no se queda atascado en mínimos locales.

Por otra parte hay que tener en cuenta algunas limitaciones o inconvenientes del método:

- Puede no ser aplicable o resultar ineficiente en espacios de soluciones discretos.
- Si el coste computacional de la función objetivo es alto, este método incurrirá en elevados tiempos de ejecución, ya que al haber muchas partículas es necesario evaluar la función objetivo muchas veces.

Para ilustrar el funcionamiento del método la figura 60 muestra el recorrido de una partícula. Nótese cómo los primeros cambios de posición son muy grandes, y a medida que se suceden las iteraciones la partícula se mueve con saltos cada vez más pequeños alrededor del punto mínimo que está buscando. Este comportamiento está potenciado por el uso de inercia decreciente, si bien también ocurre —aunque en menor grado— cuando se usa inercia constante debido a la mayor cercanía de la partícula a su propio mínimo local y al mínimo global del enjambre.

Figura 60. Recorrido de una partícula

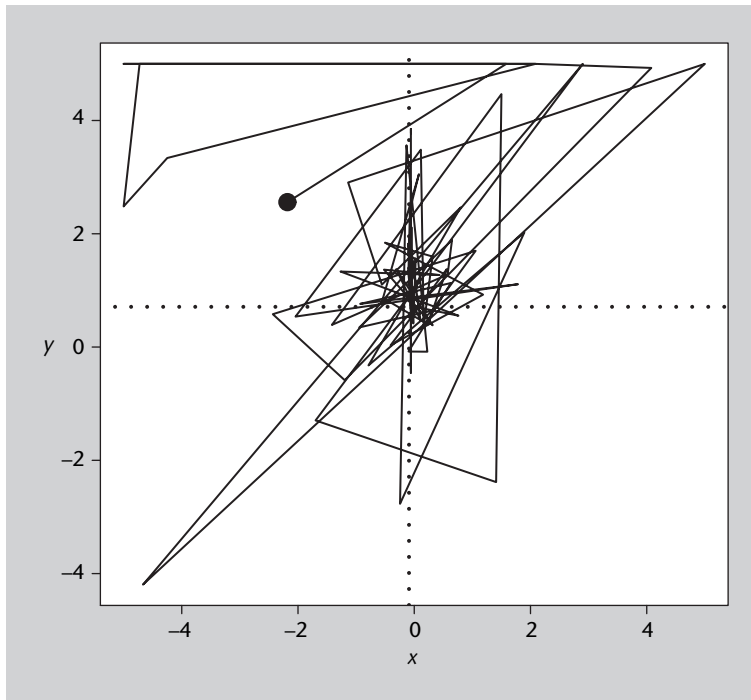


Figura 60

Recorrido de una partícula desde el punto inicial (señalado por el punto negro) hacia el punto óptimo (situado en la intersección de las líneas de puntos). Se muestra el recorrido de la mejor partícula de una simulación con 4 partículas y 100 iteraciones.

5.6.4. Código fuente en Python

Código 5.4: Algoritmo del enjambre de partículas

```

1 from random import random
2
3 # Función que se quiere minimizar
4 def funcion(x, y):
5     sum1 = x**2 * (4-2.1*x**2 + x**4/3.0)
6     sum2 = x*y
7     sum3 = y**2 * (-4+4*y**2)
8     return sum1 + sum2 + sum3
9
10
11 # Devuelve un número aleatorio dentro de un rango con
12 # distribución uniforme (proporcionada por random)
13 def aleatorio(inf, sup):
14     return random()*(sup-inf) + inf
15
16
17 # Clase que representa una partícula individual y que facilita
18 # las operaciones necesarias
19 class Particula:
20     # Algunos atributos de clase (comunes a todas las partículas)

```

```

21 # Parámetros para actualizar la velocidad
22 inercia = 1.4
23 cognitiva = 2.0
24 social = 2.0
25 # Límites del espacio de soluciones
26 infx = -2.0
27 supx = 2.0
28 infy = -1.0
29 supy = 1.0
30 # Factor de ajuste de la velocidad inicial
31 ajusteV = 100.0
32
33 # Crea una partícula dentro de los límites indicados
34 def __init__(self):
35     self.x = aleatorio(Partícula.infx, Partícula.supx)
36     self.y = aleatorio(Partícula.infy, Partícula.supy)
37     self.vx = aleatorio(Partícula.infx/Partícula.ajusteV,
38                        Partícula.supx/Partícula.ajusteV)
39     self.vy = aleatorio(Partícula.infy/Partícula.ajusteV,
40                        Partícula.supy/Partícula.ajusteV)
41     self.xLoc = self.x
42     self.yLoc = self.y
43     self.valorLoc = funcion(self.x, self.y)
44
45 # Actualiza la velocidad de la partícula
46 def actualizaVelocidad(self, xGlob, yGlob):
47     cogX = Partícula.cognitiva*random()*(self.xLoc-self.x)
48     socX = Partícula.social*random()*(xGlob-self.x)
49     self.vx = Partícula.inercia*self.vx + cogX + socX
50     cogY = Partícula.cognitiva*random()*(self.yLoc-self.y)
51     socY = Partícula.social*random()*(yGlob-self.y)
52     self.vy = Partícula.inercia*self.vy + cogY + socY
53
54 # Actualiza la posición de la partícula
55 def actualizaPosicion(self):
56     self.x = self.x + self.vx
57     self.y = self.y + self.vy
58
59 # Debe mantenerse dentro del espacio de soluciones
60 self.x = max(self.x, Partícula.infx)
61 self.x = min(self.x, Partícula.supx)
62 self.y = max(self.y, Partícula.infy)
63 self.y = min(self.y, Partícula.supy)
64
65 # Si es inferior a la mejor, la adopta como mejor
66 valor = funcion(self.x, self.y)
67 if valor < self.valorLoc:
68     self.xLoc = self.x
69     self.yLoc = self.y
70     self.valorLoc = valor
71
72
73 # Mueve un enjambre de partículas durante las iteraciones indicadas.
74 # Devuelve las coordenadas y el valor del mínimo obtenido.
75 def enjambreParticulas(particulas, iteraciones, reduccionInercia):
76
77     # Registra la mejor posición global y su valor
78     mejorParticula = min(particulas, key=lambda p:p.valorLoc)
79     xGlob = mejorParticula.xLoc
80     yGlob = mejorParticula.yLoc
81     valorGlob = mejorParticula.valorLoc
82
83 # Bucle principal de simulación
84 for iter in range(iteraciones):
85     # Actualiza la velocidad y posición de cada partícula
86     for p in particulas:
87         p.actualizaVelocidad(xGlob, yGlob)
88         p.actualizaPosicion()
89
90     # Hasta que no se han movido todas las partículas no se
91     # actualiza el mínimo global, para simular que todas se
92     # mueven a la vez

```



```

93     mejorParticula = min(particulas , key=lambda p:p.valorLoc)
94     if mejorParticula.valorLoc < valorGlob:
95         xGlob      = mejorParticula.xLoc
96         yGlob      = mejorParticula.yLoc
97         valorGlob = mejorParticula.valorLoc
98
99     # Finalmente se reduce la inercia de las partículas
100    Particula.inercia*=reduccionInercia
101
102    return (xGlob, yGlob, valorGlob)
103
104
105 # Parámetros del problema
106 nParticulas = 10
107 iteraciones = 100
108 redInercia  = 0.9
109
110 # Genera un conjunto inicial de partículas
111 particulas=[Particula() for i in range(nParticulas)]
112
113 # Ejecuta el algoritmo del enjambre de partículas
114 print(enjambreParticulas(particulas , iteraciones , redInercia))

```

5.7. Búsqueda tabú

Una editorial está elaborando un atlas del mundo y desea colorear un mapa político utilizando un número determinado de colores y evitando que países contiguos tengan el mismo color, lo que dificultaría la lectura del mapa. ¿Cómo asignar un color a cada país?

El método de la **búsqueda tabú** es una mejora de la exploración aleatoria del espacio de soluciones S , cuya ventaja radica en que se recuerdan las soluciones probadas con anterioridad. De esa forma, no se vuelven a explorar regiones que ya se han visitado previamente, y así la eficiencia de búsqueda mejora. Su nombre se debe a que las soluciones ya analizadas se convierten en tabú, ya que se impide al algoritmo acceder a ellas.

5.7.1. Descripción del método

Supongamos que queremos buscar el mínimo de una función $f : S \rightarrow \mathbb{R}$, donde S es el espacio de soluciones posibles. Expresándolo de otra manera, buscamos el punto $x^* \in S$ que cumple $x^* = \operatorname{argmin} f(x) \forall x \in S$, o bien una aproximación aceptable.

Antes de describir el método de la búsqueda tabú nos detendremos brevemente en el método de **búsqueda aleatoria local**, pues ambas están estrechamente relacionadas.

Cada punto $x \in S$ tiene un **vecindario** $V(x) \subset S$, que es el conjunto de puntos que están separados de x por un único cambio. La definición de V depende

Búsqueda aleatoria global

La búsqueda aleatoria global es diferente de la búsqueda aleatoria local, pues consiste en ir probando puntos de S al azar (sin ninguna relación entre ellos) hasta que se cumpla la condición de finalización.

del problema concreto de que se trate, pero en general dos puntos son vecinos si sólo se diferencian en una de sus variables (posiblemente en una cantidad acotada si se trata de variables numéricas).

El método de la búsqueda aleatoria local empieza evaluando un punto al azar $x_0 \in S$. A continuación toma un punto vecino al azar $x_1 \in V(x_0)$, lo evalúa y toma un vecino de x_1 . El proceso se repite hasta que se cumple la condición de finalización que se haya establecido.

Es evidente que el método de búsqueda aleatoria volverá a menudo sobre puntos explorados previamente, con la consiguiente pérdida de tiempo. Aquí entra la mejora de la búsqueda tabú: se almacenan los últimos puntos explorados, que quedan marcados como tabú, de forma que el método los evitará y se verá forzado a explorar áreas nuevas. Visto de otra forma, se define una nueva función de vecindario $V'(x,i)$ que depende de la iteración i del algoritmo, pues excluye los vecinos que se han visitado con anterioridad. En cada iteración se elige al azar un punto $x_{i+1} \in V'(x_i,i)$, se evalúa y se marca como tabú para evitar visitarlo en futuras iteraciones.

Dado que los espacios de soluciones suelen tener un número elevado de dimensiones, la probabilidad de que un punto vuelva a ser visitado es muy pequeña. Por esa razón, marcar puntos de S como tabú tendría un efecto prácticamente nulo respecto a la búsqueda aleatoria local. Para que la búsqueda tabú tenga un efecto práctico las marcas tabú no se aplican a los puntos completos, sino a cada una de las componentes de los puntos. Así, si por ejemplo el espacio de soluciones es \mathbb{Z}^3 , las marcas tabú se aplican por separado a los valores que toman x , y y z . De esa forma el efecto tabú es más potente y la mejora respecto a la búsqueda aleatoria local es apreciable.

Con la anterior estrategia se corre el riesgo de bloquear muchos puntos, lo que puede provocar que se pierdan soluciones valiosas. La forma de evitarlo es relajar la calidad de tabú de los valores, haciendo que las marcas tabú duren sólo unas iteraciones; seguidamente, los valores vuelven a estar disponibles. En el subapartado 5.7.3 se estudia la influencia de la duración de las marcas tabú en la eficiencia del método.

Dado que se trata de un método que parte de un punto al azar y explora el espacio de soluciones moviéndose entre puntos vecinos, el método de búsqueda tabú se clasifica como método de búsqueda local.

5.7.2. Ejemplo de aplicación

El ejemplo planteado inicialmente, consistente en colorear los países de un mapa sin que dos países del mismo color se toquen, es un caso particular del

problema más general del **coloreado de grafos**. Como es habitual, el mapa que queremos colorear (figura 61) puede verse como un grafo en el que cada país se representa mediante un nodo y los nodos correspondientes a países linderos están unidos mediante un arco.

Figura 61. Problema del coloreado de un mapa

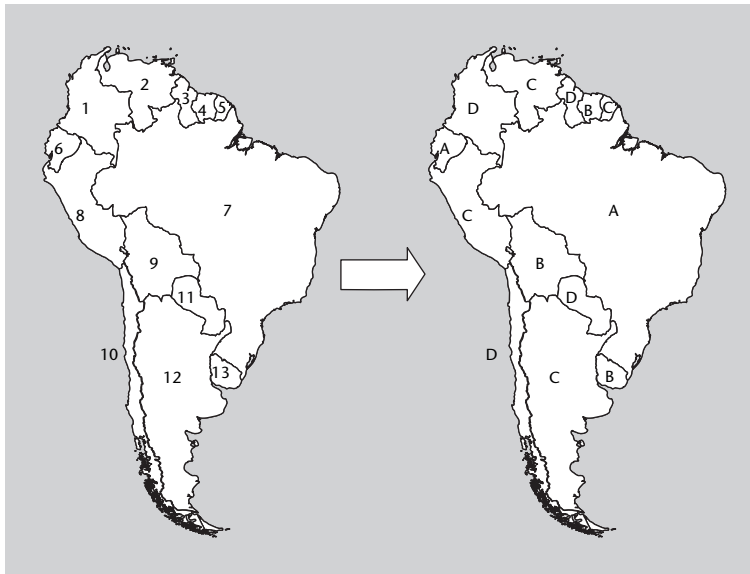


Figura 61

Ejemplo de coloreado de un mapa: tenemos un mapa con trece países (mapa izquierdo) y queremos colorearlo con cuatro colores (A, B, C y D). En el mapa derecho se muestra un posible coloreado del mapa obtenido mediante búsqueda tabú.

El problema genérico del coloreado de grafos tiene muchas más aplicaciones prácticas, como la asignación de registros del procesador al compilar, o la gestión de recursos excluyentes como pistas de aeropuertos, líneas de producción, etc. Al tratarse de un problema NP-completo, resulta interesante para probar métodos heurísticos, dado que el espacio de soluciones S tiene un tamaño $|S| = n^k$, siendo n el número de nodos y k el número de colores que se quiere utilizar; por ese motivo, una exploración exhaustiva es impracticable salvo para grafos muy pequeños.

Aunque la función objetivo más evidente para resolver este problema es el número de colores empleado en el mapa, esa función no cumple las propiedades enunciadas en el subapartado 5.1; en concreto, su principal problema es que su resolución es muy gruesa y por tanto, es incapaz de medir pequeñas mejoras en el coloreado, con lo que los algoritmos de búsqueda no reciben ninguna realimentación positiva respecto a las mejoras que van encontrando. Es necesario utilizar una función objetivo de mayor resolución, y en este caso la candidata idónea es la función que cuenta el número de colisiones, esto es de países linderos que tienen el mismo color. Como es evidente, esta función permite soluciones ilegales, pero a cambio proporciona una orientación adecuada para la búsqueda de su mínimo.

Las marcas tabú se aplican a cada una de las componentes de la solución. En este caso una solución es una correspondencia de un color para cada país, por lo que las marcas tabú serán colores prohibidos para un país determinado, debidas a que recientemente se le ha asignado ese color.

Si se deseara obtener el número mínimo de colores con los que se puede colorear un mapa dado, no habría más que iterar el algoritmo de búsqueda con un número de colores decreciente, hasta que el algoritmo sea incapaz de encontrar una asignación de colores válida (sin países linderos del mismo color).

5.7.3. Análisis del método

La búsqueda tabú es un método conceptualmente sencillo y relativamente eficiente de mejorar la búsqueda aleatoria local. No obstante, se trata de un método de búsqueda local, con las limitaciones que ello conlleva; a menudo conviene ejecutarlo repetidas veces para hacerlo más global y permitirle encontrar el mínimo global.

Como todo método que utiliza una función de vecindario, está mejor adaptado a espacios de soluciones discretos; en otro caso, es necesario discretizar el espacio o definir algún tipo de distancia de vecindad.

No hay un valor óptimo para la duración de las marcas tabú, sino que depende del número de dimensiones del espacio de soluciones y del número de valores posible de cada componente. En la figura 62 se estudia la influencia de la duración de las marcas tabú en la eficiencia del método en el ejemplo del coloreado del mapa. Una duración igual a cero equivale a la búsqueda aleatoria local, que es claramente peor que la búsqueda tabú. Una duración muy corta (<10) provoca que el tabú tenga poco efecto. Por otra parte, una duración muy larga vuelve el método muy rígido, pues excluye numerosas soluciones, y dificulta la exploración. Por tanto, tal y como se ve en la figura, es mejor elegir un número intermedio de iteraciones, en este ejemplo concreto 15.

Figura 62. Influencia de la duración del tabú

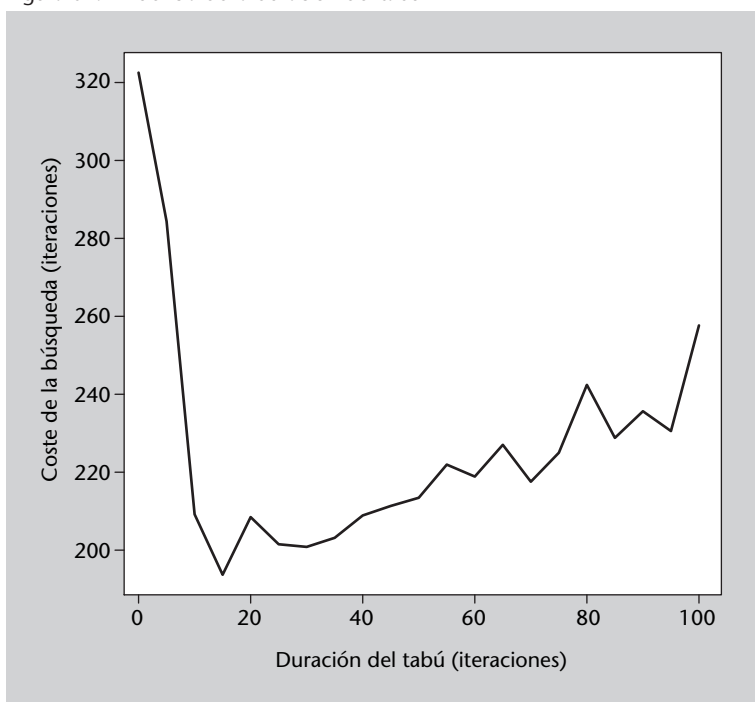


Figura 62

La duración del tabú influye claramente en la eficiencia de los algoritmos de búsqueda tabú. En la figura se muestra el número de iteraciones necesario para obtener la solución en un mapa con 200 países, una probabilidad de contacto entre ellos de 0,04 y cinco colores disponibles. Se muestran los valores medios de la búsqueda en 100 mapas diferentes.

5.7.4. Código fuente en Python

Código 5.5: Coloreado de mapas con búsqueda tabú

```

1  # Nota: los grafos se representan como listas de listas
2  from random import randint, random
3
4  # Dados dos nodos, devuelve 1 si están conectados, 0 si no
5  def conectados(grafo, nodo1, nodo2):
6      if nodo1 < nodo2:
7          nodo1, nodo2 = nodo2, nodo1
8      if nodo1 == nodo2:
9          return 0
10     else:
11         return grafo[nodo1][nodo2]
12
13
14 # Función objetivo, que cuenta los vértices conectados que tienen
15 # el mismo color
16 def objetivo(grafo, estado):
17     suma = 0
18     for i in range(len(estado)-1):
19         for j in range(i+1, len(estado)):
20             if estado[i] == estado[j]:
21                 suma += conectados(grafo, i, j)
22     return suma
23
24 # Devuelve una lista con los vértices que tienen algún problema
25 # (están conectados a otro del mismo color)
26 def verticesProblema(grafo, estado):
27     nVertices = len(estado)
28     vertices = []
29     for i in range(nVertices):
30         encontrado = False
31         j = 0
32         while j < nVertices and not encontrado:
33             if (estado[i] == estado[j]) and conectados(grafo, i, j):
34                 encontrado = True
35             j += 1
36         if encontrado:
37             vertices.append(i)
38     return vertices
39
40 # Genera la lista de vecinos de un estado y devuelve el
41 # mejor de ellos, teniendo en cuenta la información tabú y
42 # actualizándola para anotar el cambio de estado producido.
43 def vecino(grafo, estado, k, tabu, iterTabu):
44     nVertices = len(estado)
45
46     # En primer lugar se obtienen todos los vértices
47     # conectados a algún otro vértice del mismo color
48     vertices = verticesProblema(grafo, estado)
49
50     # Para cada uno de esos pares, se propone una
51     # nueva solución consistente en cambiar el color de uno de
52     # los vértices (comprobando que no tome un color tabú).
53     candidatos = []
54     for v in vertices:
55         # Si quedan colores disponibles para ese vértice
56         if len(tabu[v]) < k:
57             nuevoColor = randint(1, k)
58             while tabu[v].has_key(nuevoColor):
59                 nuevoColor = randint(1, k)
60             nuevoEstado = estado[:]
61             nuevoEstado[v] = nuevoColor
62             candidatos.append(nuevoEstado)
63
64     # Se devuelve el mejor estado obtenido (función objetivo menor).
65     # Si no hay soluciones posibles (los vértices problemáticos
66     # tienen todos los colores en tabú) se devuelve un cambio al
67     # azar para desbloquear la situación.
68     if len(candidatos) > 0:

```

```

69     elegido = min(candidatos, key=lambda e: objetivo(grafo, e))
70     posicion = candidatos.index(elegido)
71     cambiado = vertices[posicion]
72     else:
73         elegido = estado[:]
74         cambiado = sample(vertices, 1)[0]
75         elegido[cambiado] = randint(1, k)
76
77     # Se añade el cambio a la lista tabú.
78     tabu[cambiado][estado[cambiado]] = iterTabu
79     return elegido
80
81
82 # Actualiza la información tabú, descontando una iteración de
83 # todas las entradas y eliminando las que llegan a 0
84 def actualizaTabu(tabu):
85     for tabuVertice in tabu:
86         for color in tabuVertice.keys():
87             tabuVertice[color] -= 1
88             if tabuVertice[color] <= 0:
89                 del tabuVertice[color]
90
91
92 # Dados un grafo, una asignación inicial y un número de colores,
93 # busca una asignación óptima utilizando búsqueda tabú.
94 # También hay que indicar el número máximo de iteraciones, así
95 # como el número de iteraciones que se mantiene el color en tabú.
96 def busquedaTabu(grafo, asignacion, k, iteraciones, iterTabu):
97
98     estado = asignacion[:]
99     mejor = estado[:]
100    nVertices = len(estado)
101
102    # La estructura "tabu" almacena, para cada vértice (posiciones
103    # en la lista), los colores que tiene prohibidos (claves de
104    # cada diccionario) y durante cuántas iteraciones lo estarán
105    # (valores de cada diccionario)
106    tabu = [{ for n in range(nVertices)]
107    i = 0
108    while i < iteraciones and objetivo(grafo, estado) > 0:
109
110        # Selecciona el mejor vecino del estado actual
111        estado = vecino(grafo, estado, k, tabu, iterTabu)
112
113        if objetivo(grafo, estado) < objetivo(grafo, mejor):
114            mejor = estado[:]
115        print i, objetivo(grafo, estado), objetivo(grafo, mejor)
116
117        # Actualiza los estados tabú (descuenta una iteración)
118        actualizaTabu(tabu)
119
120        # Avanza a la siguiente iteración
121        i += 1
122
123    return mejor
124
125
126 # Genera un grafo en forma de matriz de conectividad, con un número
127 # de vértices y una probabilidad de contacto entre dos vértices.
128 # Devuelve la mitad inferior de la matriz como lista de listas.
129 def generaGrafo(nVertices, probContacto):
130     resultado = []
131     for i in range(nVertices):
132         # 1 si el número aleatorio es menor o igual que
133         # la probabilidad de contacto, 0 si no
134         f = lambda : random() <= probContacto and 1 or 0
135         resultado.append([f() for j in range(i)])
136
137     return resultado
138
139
140 # Programa principal

```

```
141
142 # Ejemplo: grafo que representa Sudamérica (no se resuelve)
143 sudamerica = [[] ,
144               [1] ,
145               [0,1] ,
146               [0,0,1] ,
147               [0,0,0,1] ,
148               [1,0,0,0,0] ,
149               [1,1,1,1,1,0] ,
150               [1,0,0,0,0,1,1] ,
151               [0,0,0,0,0,0,1,1] ,
152               [0,0,0,0,0,0,0,1,1] ,
153               [0,0,0,0,0,0,1,0,1,0] ,
154               [0,0,0,0,0,0,1,0,1,1,1] ,
155               [0,0,0,0,0,0,1,0,0,0,1]]
156
157 # Generación de un grafo aleatorio.
158 # Parámetros del problema: número de países y probabilidad
159 # de que dos países cualesquiera sean linderos.
160 # Lógicamente, cuantos más países haya, mayor número de
161 # contactos tendrá un país determinado.
162 nPaíses      = 100
163 probContacto = 0.05
164 mapa         = generaGrafo(nPaíses, probContacto)
165
166 # Ejecución del algoritmo: número máximo de iteraciones del
167 # algoritmo y número que iteraciones que permanece un color
168 # como tabú para un país dado.
169 iteraciones = 200
170 iterTabu    = 15
171
172 # Número de colores
173 k=4
174
175 # Se crea un estado inicial aleatoriamente
176 estado = [randint(1,k) for i in range(nPaíses)]
177
178 # Se ejecuta la búsqueda tabú con k colores.
179 solucion = busquedaTabu(prueba, estado, k, iteraciones, iterTabu)
180 print('Solución:' + str(solucion))
181 print('Colisiones=' + str(objetivo(prueba, solucion)))
```

6. Anexo: conceptos básicos de estadística

En el apartado 3 definimos la *densidad de probabilidad* $\rho_x(k)$ en un intervalo b_k se define entonces como el número de valores que se han producido en dicho intervalo h_k normalizado con el área total del histograma

$$\rho_x(k) = \frac{h_k}{n\Delta x}, \quad (65)$$

y la *función densidad de probabilidad continua* tomando el límite en el que el tamaño del intervalo Δx se hace infinitesimalmente pequeño

$$\rho_x(x) = \lim_{\Delta x \rightarrow 0, m \rightarrow \infty} \rho_x(k). \quad (66)$$

Esta función permite a su vez definir la *función de probabilidad acumulada*

$$p(x \leq x_0) = \int_{-\infty}^{x_0} \rho_x(x) dx, \quad (67)$$

que indica la probabilidad de que x tome un valor inferior o igual a x_0 . También permite definir los *momentos estadísticos* de la distribución de probabilidad de $x(t)$. El *primer momento* o media viene dado por

$$E[x] = \bar{x} = \int_{-\infty}^{\infty} x \rho_x(x) dx \quad (68)$$

que corresponde al valor medio de la señal x que denotaremos \bar{x} . Desde un punto de vista geométrico, el valor medio \bar{x} puede ser interpretado como el centro de masas de la distribución de probabilidad ρ_x .

De forma equivalente, el *segundo momento* de la distribución de $x(t)$ viene dado por

$$E[x^2] = \int_{-\infty}^{\infty} x^2 \rho_x(x) dx, \quad (69)$$

y puede ser expresado utilizando la definición del primer momento como

$$E[x^2] = E[x]^2 + E[(x - E[x])^2] = \bar{x}^2 + E[(x - \bar{x})^2]. \quad (70)$$

El segundo término de la última expresión se denomina *varianza* de la distribución

$$\text{Var}[x] = E[(x - \bar{x})^2], \quad (71)$$

y su raíz cuadrada es la *desviación típica*

$$\sigma = \sqrt{\text{Var}[x]}, \quad (72)$$

que constituye una medida de la variabilidad de los valores de $x(t)$ en torno a su valor medio \bar{x} . En general, el momento de orden p de la distribución ρ_x vendrá dado por

$$E[x^p] = \int_{-\infty}^{\infty} x^p \rho_x(x) dx. \quad (73)$$

Aparte de la media y la varianza, dos de los momentos más relevantes son la *asimetría*, definida como el momento central (momento de la variable $x - \bar{x}$) de tercer orden

$$E[(x - \bar{x})^3] = \int_{-\infty}^{\infty} (x - \bar{x})^3 \rho_x(x) dx, \quad (74)$$

y la *curtosis*, definida como el cociente entre los momentos de tercer y cuarto orden con una corrección para que una distribución gaussiana tenga curtosis nula.

$$K = \frac{E[x^3]}{E[x^4]} - 3. \quad (75)$$

La curtosis permite cuantificar la medida en que una distribución de probabilidad se desvía de un patrón gaussiano, y es uno de los estadísticos que se utilizan en el cálculo de las componentes independientes del método ICA. El código 6.1 explica cómo calcular los momentos estadísticos de un conjunto de datos. En este ejemplo se utilizan datos gaussianos de media $\mu = 10$ y desviación típica $\sigma = 5,5$ (varianza es $\sigma^2 = 30,25$). Al final del código se muestran los valores muestrales estimados, que coinciden con los valores poblacionales correspondientes.

Código 6.1: Cálculo de los momentos estadísticos de un conjunto de datos

```

1 from scipy import stats
2 import numpy as np
3
4 # datos gaussianos:
5 mu, sigma = 10, 5.5
6 y = mu + sigma*np.random.randn(10000)
7
8 # Estimación de momentos:
9 # momento de primer orden: media

```

```

10 media = y.mean()
11
12 # momento segundo orden: varianza
13 var = y.var()
14
15 # momento tercer orden: asimetría (skweness)
16 skew = stats.skew(y)
17
18 # momento cuarto orden: Curtosis
19 kurt = stats.kurtosis(y)
20
21 >>> print media, var, skew, kurt
22 10.0575611617 30.9623694469 0.0126290768091 0.025294666173

```

En caso de tener que calcular los momentos de la distribución a partir de un conjunto discreto de valores de la señal $\{x_1, x_2, \dots, x_n\}$, procederemos a realizar una *estimación muestral*. Dos posibles estimadores muestrales (aunque no los únicos) de la media $\bar{x} = E[x]$ y la varianza $\sigma = \sqrt{\text{Var}[x]}$ vienen dados, respectivamente, por

$$\bar{x} \approx \frac{1}{n} \sum_{i=1}^n x_i \quad (76)$$

$$\sigma \approx \frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2. \quad (77)$$

Los estimadores muestrales presentan un error respecto a los valores reales de la distribución (valores poblacionales), puesto que para cada conjunto de valores $\{x_1, x_2, \dots, x_n\}$ se obtiene un valor estimado diferente. El error de estimación consta típicamente de dos partes, una debida a la desviación en media del estimador respecto al valor real conocida como *sesgo*, y otra parte que da cuenta de la dispersión de los valores estimados a partir de diferentes conjuntos de valores conocida como *varianza*. Dependiendo del estimador muestral escogido, tendremos una cierta ponderación de error debido a sesgo y debido a varianza. Por ejemplo, es fácil demostrar que mientras que $\frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2$ estima la desviación típica σ de forma sesgada, el estimador

$$\sigma \approx \frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})^2. \quad (78)$$

es un estimador no sesgado de la desviación típica de una distribución.

Actividades

1. Visualizad un diagrama de dispersión de las puntuaciones de películas de usuarios con correlaciones muy bajas (cerca a -1), nulas y muy altas (cerca a 1). Determinar aproximadamente cuál sería la línea de ajuste en cada caso.
2. Relacionad las puntuaciones de películas en *Movielens* con la edad: ¿se parecen más las puntuaciones de edades cercanas que las puntuaciones generales entre sí?
3. Completad la aplicación de recomendación de películas. La aplicación deberá pedir opinión sobre 5 películas al usuario y a partir de esos datos sugerirle películas que podrían gustarle.
4. Tomad una imagen con manchas de puntos (dibujadas con cualquier programa de dibujo) y obtened los grupos correspondientes utilizando k-medios y c-medios difuso.
5. Tomad una fotografía e intentad delimitar zonas similares utilizando c-medios difuso.
6. Agrupad las películas de la base de datos *Movielens* por sus valoraciones. Analizad si hay correlación con el género de la película (fichero *u.genre*).
7. Utilizad el núcleo gaussiano en el programa de agrupamiento espectral al calcular la matriz de distancia. Analizad la influencia del parámetro σ en el número de grupos obtenido.
8. Sustituid la laplaciana no normalizada por la laplaciana de recorrido aleatorio (*random walk*) en el programa 2.8 y comparad los resultados obtenidos.
9. Modificad la implementación (programa 4.1) del Naïve Bayes (subapartado 4.2.1) aplicando la distribución gaussiana para poder tratar atributos numéricos y aplicadlo al conjunto de datos de las flores (subapartado 4.3.1).
10. Modificad la implementación (programa 4.2) del kNN (subapartado 4.3.1) para que vote teniendo en cuenta la ponderación de ejemplos en función de su cercanía al ejemplo de test.
11. Modificad la implementación (programa 4.2) del kNN (subapartado 4.3.1) para que utilice la distancia de Hamming y aplicadlo al conjunto de datos de las setas (subapartado 4.2.1).
12. Modificad la implementación (programa 4.3) del clasificador lineal basado en distancias (subapartado 4.3.2) para que utilice la distancia de Hamming y aplicadlo al conjunto de datos de las setas (subapartado 4.2.1).
13. Modificad la implementación (programa 4.4) del clasificador basado en *clustering* (subapartado 4.3.3) para que utilice la distancia de Hamming y aplicadlo al conjunto de datos de las setas (subapartado 4.2.1).
14. Modificad la implementación (programa 4.4) del clasificador basado en *clustering* (subapartado 4.3.3) cambiando el método de *clustering* (ver subapartado 2.3).
15. Modificad la implementación (programa 4.5) de los árboles de decisión (subapartado 4.4.1) para que incluya el tratamiento de los puntos de corte de atributos numéricos y aplicadlo al conjunto de datos de las flores (subapartado 4.3.1).
16. Implementad el algoritmo ID3 (subapartado 4.4.1) partiendo del programa 4.5 y aplicadlo al conjunto de datos de las flores (subapartado 4.3.1).
17. Añadid la funcionalidad de la poda de ramas al programa realizado en el ejercicio anterior.
18. Diseñad las reglas débiles para tratar el problema de las flores (subapartado 4.3.1) y modificad el programa 4.6 para poder distinguir la clase *iris-setosa* de las demás.
19. Añadid la técnica del *uno contra todos** (descrito en el subapartado 4.5.4) al programa 4.6 y aplicadlo al conjunto de datos de las flores (subapartado 4.3.1).
20. Partiendo de los dos ejercicios anteriores, mirad cuál es mejor sobre el conjunto de datos y si la diferencia es estadísticamente significativa.
21. Modificad el programa 4.7 para que calcule los dos algoritmos del subapartado 4.5.1.
22. Añadid el kernel polinómico $(\|x-z\|^2 + 1)^d$ con d equivalente al grado del polinomio) al programa 4.8.

* En inglés, *one vs all*.

23. Procesad los datos del segundo conjunto de categorización descrito en el subapartado 4.1.1 para eliminar las palabras sin contenido semántico utilizando la lista de *stop words* del inglés descrita en el mismo subapartado y comparad el primer algoritmo del subapartado 4.5.3 con los datos sin procesar.
24. Procesad los datos del segundo conjunto de categorización descrito en el subapartado 4.1.1 para eliminar las palabras que no aparecen más de dos veces en el corpus de entrenamiento y comparad el primer algoritmo del subapartado 4.5.3 con los datos sin procesar, con los del problema anterior y utilizando los dos procesos.
25. Realizad los dos ejercicios anteriores para el tercer conjunto de datos del subapartado 4.1.1 y aplicadlo con el segundo programa del subapartado 4.5.3.
26. Ampliad el programa 4.10 con el kernel para normalizar descrito en el subapartado 4.5.3 y comparadlo al programa sin normalizar.
27. Probad los diferentes tipos de kernels de las *svm^{light}* modificando el programa 4.11.
28. Probad los diferentes tipos de kernels de las *libsvm* modificando el programa 4.12.
29. Ampliad el programa 4.13 con el kernel para la ponderación de términos con la información mútua (subapartado 3.1.3).
30. Ampliad el programa 4.13 con el kernel para la ponderación de términos con la entropía (subapartado 4.4.1).
31. Ampliad el programa 4.13 con el kernel para la ponderación de términos del *tf-idf* descrita en el subapartado 4.5.3.
32. Ampliad el ejercicio anterior para que incluya el kernel para la semejanza entre términos y así completar el *tf-idf*.
33. Ampliad los programas 4.2 y 4.3 con la validación cruzada y comparad cuál de los dos va mejor para el conjunto de datos de las flores (subapartado 4.3.1) y si la diferencia es significativa utilizando el test de Student.
34. Repetid el ejercicio anterior utilizando el test del Wilcoxon en lugar del de Student.
35. Buscad ocho conjuntos de datos en repositorio de la UCI, añadirlos al de las flores y setas y utilizad el test de Friedman y su mejora para comparar los algoritmos: Naïve Bayes, kNN, lineal, árboles de decisión, AdaBoost y SVMs (códigos 4.1, 4.2, 4.3, 4.5, 4.6 y 4.12).
36. Utilizad el estadístico kappa dos a dos sobre los algoritmos: Naïve Bayes, kNN, lineal, árboles de decisión, AdaBoost y SVMs (códigos 4.1, 4.2, 4.3, 4.5, 4.6 y 4.13) para crear una matriz de valores sobre el tercer conjunto de datos del subapartado 4.1.1 y cread un dendrograma para comparar los diferentes algoritmos.
37. Resolved cada uno de los problemas de ejemplo de los métodos metaheurísticos con los restantes métodos. Comparad los resultados obtenidos y el tiempo de computación requerido en cada caso.
38. Utilizad una imagen en escala de grises a modo de mapa de alturas, donde los pixels más claros representan puntos más elevados en un mapa. Encontrad el camino entre dos puntos del mapa que cumpla que el desnivel acumulado (total de subida y bajada) sea mínimo utilizando el algoritmo de la colonia de hormigas (subapartado 5.5). Nótese que previamente a aplicar el algoritmo hay que transformar el mapa de alturas en un grafo.
39. Escribid un programa que resuelva un sudoku utilizando el método de la búsqueda tabú visto en el subapartado 5.7. Para que la función objetivo cumpla las propiedades vistas en el subapartado 5.1 tiene que admitir sudokus con errores e ir eliminándolos progresivamente.
40. En un instituto con 4 grupos de alumnos y 5 profesores hay que encontrar una distribución de horarios compatible (ningún profesor puede dar clase en dos grupos a la vez y ningún grupo puede recibir dos clases a la vez). Cada grupo tiene 6 clases cada día durante cinco días a la semana, que se dividen en 10 asignaturas con 3 horas semanales cada una. En un mismo día no pueden tener más de una hora de la misma asignatura. Cada profesor imparte dos asignaturas en su totalidad, por lo que da un total de 24 horas de clase a la semana. Hay que obtener un horario válido utilizando alguno de los métodos vistos en este apartado.

Bibliografía

Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. USA: Springer.

Duda, R. O.; Hart, P. E.; Stork, D. G. (2001). *Pattern Classification* (2.^a ed.). USA: John Wiley and Sons, Inc.

Frank, A.; Asuncion, A. (2010). *UCI Machine Learning Repository* (Disponible online: <http://archive.ics.uci.edu/ml>). USA: University of California, School of Information and Computer Science.

Mitchell, T. M. (1997). *Machine Learning*. USA: McGraw-Hill.

Segaran, T. (2007). *Programming Collective Intelligence*. USA: O'Reilly.

Shawe-Taylor, J.; Cristianini, N. (2004). *Kernel Methods for Pattern Analysis*. UK: Cambridge University Press.

