

APRENDIZAJE SUPERVISADO



Dr. Gualberto Asencio Cortés

MÁSTER UNIVERSITARIO EN INTELIGENCIA ARTIFICIAL

Módulo de Aprendizaje automático



Universidad
Internacional
de Valencia



Universidad
Internacional
de Valencia

Este material es de uso exclusivo para los alumnos de la Universidad Internacional de Valencia. No está permitida la reproducción total o parcial de su contenido ni su tratamiento por cualquier método por aquellas personas que no acrediten su relación con la Universidad Internacional de Valencia, sin autorización expresa de la misma.

Edita
Universidad Internacional de Valencia

Máster Universitario en
Inteligencia Artificial

Aprendizaje Supervisado
Módulo de Aprendizaje automático
6 ECTS

Dr. Gualberto Asencio Cortés

Leyendas



Enlace de interés



Ejemplo



Importante



abc Los términos resaltados a lo largo del contenido en color naranja se recogen en el apartado GLOSARIO.

Índice

CAPÍTULO 1. APRENDIZAJE AUTOMÁTICO	7
1.1. Metodología	7
1.1.1. Introducción al aprendizaje automático.....	7
1.1.2. La metodología CRISP-DM	8
1.1.3. Roles en ciencia de datos.....	10
1.1.4. Comprensión del negocio	11
1.1.5. Comprensión de los datos.....	11
1.1.6. Preparación de los datos	12
1.1.7. Modelado	13
1.1.8. Evaluación de negocio	15
1.1.9. Despliegue.....	15
1.2. Tipos de aprendizaje	16
1.2.1. Aprendizaje supervisado.....	16
1.2.2. Aprendizaje no supervisado	17
1.2.3. Otros tipos de aprendizaje.....	17
1.3. Estructura de datos	18
1.3.1. Estructuras y tipos en ciencia de datos	18
1.3.2. Instancias, clase, atributos y modelos.....	19
1.3.3. Características de los conjuntos de datos	20
1.4. Limpieza de datos.....	20
1.4.1. Introducción a la limpieza de datos	20
1.4.2. Normalización y estandarización.....	21
1.4.3. Detección de outliers.....	25
1.4.4. Imputación de valores ausentes	27
1.4.5. Selección de atributos	29
 CAPÍTULO 2. VALIDACIÓN Y EVALUACIÓN.....	33
2.1. Validación hold-out	36
2.2. Validación cruzada.....	39
2.3. Ajuste de parámetros y validación anidada	43
2.4. Evaluación en regresión.....	45
2.5. Evaluación en clasificación.....	48
 CAPÍTULO 3. REGRESIÓN.....	54
3.1. Regresión lineal múltiple.....	54
3.2. Vecinos más cercanos	62

CAPÍTULO 4. CLASIFICACIÓN	75
4.1. Regresión logística.....	75
4.2. Árboles de decisión	83
GLOSARIO	96
ENLACES DE INTERÉS	100
BIBLIOGRAFÍA	101



Capítulo 1

Aprendizaje automático

1.1. Metodología

1.1.1. Introducción al aprendizaje automático

El aprendizaje supervisado es un tipo de aprendizaje concreto dentro de lo que se conoce como **aprendizaje automático** (*machine learning* en inglés). Por tanto, para comprender en qué consiste el aprendizaje supervisado, debemos introducir en primer lugar los conceptos fundamentales en los que se apoya el aprendizaje automático.

El aprendizaje automático es una rama de conocimiento en la que se abordan algoritmos que son capaces de crear modelos de conocimiento abstractos a partir de históricos de datos. Por consiguiente, estos modelos generados automáticamente son de naturaleza empírica, pues solo se basan en hechos representados por datos.

En contraposición, los modelos más utilizados en numerosos campos de conocimiento científico suelen ser teóricos, como sucede en la física, la química, la biología, la medicina, la agronomía, la economía, la psicología, etc. En todos estos campos, los modelos suelen proceder del estudio en condiciones ideales de objetos sujetos a determinadas condiciones experimentales. No obstante, dada la alta capacidad de procesamiento de información de los computadores actuales y el despliegue de aparatos de medición capaces de generar grandes volúmenes de datos, los modelos empíricos van cobrando cada vez mayor importancia.

Se espera de ellos que sean más precisos y ajustados a la realidad que los modelos teóricos y que superen sus condiciones ideales de aplicabilidad. En cualquier caso, ya sean empíricos o teóricos, los modelos nos permiten hacer inferencias y nos ayudan a comprender mejor la realidad.

El aprendizaje automático es una rama de conocimiento que pertenece al campo de la **minería de datos** (*data mining* en inglés). La minería de datos, a su vez, hace referencia al conjunto de procesos, métodos y técnicas que conducen a la extracción de conocimiento a partir de bases de datos. El proceso de minería de datos es amplio y complejo y, por consiguiente, han surgido diversas metodologías para llevar a cabo los proyectos de minería de datos de forma racional y estructurada.

Entre las metodologías de minería de datos propuestas, tanto en la literatura como por la industria, destacan el proceso KDD (*Knowledge Discovery from Databases*, descubrimiento de conocimiento a partir de bases de datos) y las metodologías CRISP-DM, SEMMA y ASUM-DM. Mientras que en el proceso KDD la minería de datos es solo una etapa, en las metodologías CRISP-DM, SEMMA y ASUM-DM, la minería de datos es, en sí misma, el proceso global llevado a cabo.

El proceso KDD fue introducido en 1996 por Fayyad, Piatetsky-Shapiro y Smith, y ha sido ampliamente utilizado en proyectos de minería de datos reales, especialmente en el ámbito académico. Sin embargo, esta metodología carece de proyección en el ámbito de la industria, pues están ausentes ciertas tareas críticas en los proyectos reales empresariales, tales como una fase de comprensión del problema de negocio que motiva el proyecto o el despliegue y la puesta en explotación de los modelos en un entorno empresarial.

Las metodologías CRISP-DM y ASUM-DM sí tienen en cuenta la proyección industrial de los proyectos de minería de datos e incluyen las tareas críticas que estos necesitan contemplar. ASUM-DM fue una evolución de CRISP-DM diseñada por IBM en 2015, pero aún no ha sido acogida por la industria. Entre todas las metodologías existentes, la más utilizada y reconocida actualmente es CRISP-DM, y es la que estudiaremos en este capítulo.

1.1.2. La metodología CRISP-DM

La **metodología CRISP-DM** (*Cross-Industry Standard Process for Data Mining* en inglés) fue concebida a finales de 1996 y se puso en marcha gracias a un proyecto de financiación europea en el año 1997. El proyecto CRISP-DM fue codirigido por cinco entidades: SPSS, Teradata, NCR, Daimler AG y Ohra. La primera versión de la metodología CRISP-DM se presentó en 1999; es la utilizada en la actualidad y la que abordamos en este capítulo.

CRISP-DM integra todas las tareas necesarias en los proyectos de minería de datos reales, desde la fase de comprensión del problema hasta la puesta en producción de sistemas automatizados analíticos, predictivos y/o prospectivos, incluyendo las tareas de adquisición y comprensión de los datos, la limpieza y la transformación, el análisis y la visualización, la creación de modelos y la extracción de patrones, la evaluación y la interpretación de resultados.

La metodología CRISP-DM se puede aplicar a un vasto y variado repertorio de problemas.

Por ejemplo, resulta útil para las siguientes tareas:

- Encontrar perfiles de clientes fraudulentos (evasión de pagos e impuestos).
- Descubrir relaciones implícitas entre síntomas y enfermedades de pacientes de un hospital.

- Determinar relaciones entre especificaciones técnicas de máquinas, archivos de registro y diagnóstico de errores en fábricas o centros de datos.
- Estimar la probabilidad de que los clientes de una empresa se vayan a la competencia.
- Determinar patrones de compra de clientes de un supermercado y recomendar productos atractivos a clientes en función de lo que compran.
- Segmentar clientes de forma automática para definir diferentes objetivos de mercado y dirigir campañas de marketing específicas a cada segmento.

CRISP-DM se compone de seis fases, las cuales dependen entre sí tanto en forma secuencial como cíclica, pudiendo existir iteraciones que permitan mejorar la aproximación obtenida en otras anteriores. Las seis fases de CRISP-DM son:

- Comprensión del negocio.
- Comprensión de los datos.
- Preparación de los datos.
- Modelado.
- Evaluación.
- Despliegue.



Enlaces de interés

En el siguiente enlace, se puede consultar un esquema de la metodología CRISP-DM para la realización de proyectos de minería de datos, incluyendo sus seis fases y las dependencias entre las mismas. La fase de comprensión del negocio (*business understanding* en inglés) es fundamental en la metodología, pues se recurre a esta desde varias otras fases, como la comprensión de los datos o la evaluación de los resultados.

<http://crisp-dm.eu/reference-model/>

Además, en el siguiente enlace se encuentra la web oficial de la metodología CRISP-DM (versión 1.0).

<https://www.sv-europe.com/crisp-dm-methodology/>



Descarga de archivo

En Campus virtual > Aula de la asignatura > Recursos y materiales podrás consultar la guía completa de la metodología CRISP-DM (versión 1.0) (archivo: [CRISP-DM](#)).

En los apartados 1.1.4 a 1.1.9 se estudian de forma resumida cada una de las seis fases de la metodología CRISP-DM. El objetivo de este apartado del capítulo es proporcionar un marco de referencia en el que situar, dentro de un proyecto real, los conceptos, los algoritmos y las técnicas que se verán a lo largo de la asignatura.

1.1.3. Roles en ciencia de datos

Aunque la metodología CRISP-DM no especifica nada al respecto, en esta asignatura introducimos cuatro de los roles principales que actualmente suelen asumir los miembros de un equipo de trabajo dentro de los proyectos reales de **ciencia de datos** (*data science* en inglés).

- **Director técnico de proyecto (*data scientist chief*)**. Es la persona responsable del proyecto, quien planifica el proyecto, asigna recursos, supervisa el trabajo del equipo, establece prioridades y realiza la validación última de los resultados del proyecto. Debe tanto reunir conocimientos de gestión de equipos y negocio, como poseer un perfil técnico.
- **Científico de datos (*data scientist*)**. Se encarga de analizar los problemas, comunicarse con expertos y personal técnico, diseñar e implementar las soluciones más apropiadas, diseñar y ejecutar la experimentación e interpretar los resultados. Debe poseer conocimientos completos y profundos de la metodología de minería de datos, matemáticas e ingeniería informática, así como habilidad para la comunicación. Se suele valorar el grado de doctor.
- **Analista de datos (*data analyst*)**. Es la persona que se encarga de realizar estudios de análisis de datos e inteligencia de negocio (*business intelligence* en inglés) con el objetivo de extraer conclusiones útiles a partir de los datos y elaborar informes y cuadros de mandos. Debe poseer altos conocimientos de estadística, así como de las tecnologías de inteligencia de negocio.
- **Ingeniero de datos (*data engineer*)**. Es la persona que maneja los datos con destreza utilizando herramientas de transformación de datos, sistemas gestores de bases de datos y lenguajes de programación para extraer, limpiar, transformar y cargar los datos de forma estructurada. Su objetivo es proporcionar datos limpios y organizados para los analistas y científicos de datos, así como automatizar los procesos de tratamiento de datos. Debe poseer altos conocimientos de ingeniería informática.

Estos cuatro roles no son los únicos que se emplean en los proyectos de minería de datos. También existen otros, como ingenieros de software, administradores de bases de datos y de sistemas, y diseñadores gráficos.



Enlaces de interés

En el siguiente enlace se encuentra un diagrama que muestra algunas de las habilidades habitualmente requeridas en equipos de proyectos de ciencia de datos. Como se puede observar, el rol de científico de datos es uno de los que más habilidades debe reunir dentro de un proyecto de ciencia de datos.

<https://www.atkearney.com/analytics/article?/a/it-s-challenge-bringing-structure-to-the-unstructured-world-of-big-data>

Se puede obtener más información sobre los roles mencionados y otros en este artículo publicado en el sitio KDnuggets, uno de los sitios web especializados en minería de datos de mayor reconocimiento.

<https://www.kdnuggets.com/2015/11/different-data-science-roles-industry.html>

1.1.4. Comprensión del negocio

Esta es la primera fase por la cual debe comenzar todo proyecto de minería de datos, por sencillo que sea. Además, debería ser una fase con actividad recurrente durante todo el proyecto, pues es importante que los desarrollos sean útiles y estén alineados con las necesidades que motivaron el proyecto.

En esta fase se establecen los requisitos y objetivos del proyecto desde una perspectiva empresarial para luego trasladarlos a objetivos técnicos y a un plan de proyecto. Para llevar a cabo esta fase, es necesario comprender en profundidad el problema que se quiere resolver. En concreto, esta fase se compone de las siguientes tareas genéricas:

- **Determinar los objetivos de negocio.** Se determina cuál es el problema que se pretende resolver y por qué se usa la minería de datos para dicho propósito. Se establecen también cuáles serán los criterios para medir el éxito en el proyecto, ya sean de tipo cualitativo o cuantitativo. Por ejemplo, si el problema es detectar fraude en el uso de la electricidad, el criterio de éxito podría ser cuantitativo: el número de detecciones de fraude.
- **Evaluar la situación actual.** Se determinan los antecedentes y requisitos del problema, tanto en términos de negocio como de minería de datos. Algunos de los aspectos que hay que tener en cuenta pueden ser el personal con conocimiento previo acerca del tema, la calidad y cantidad de los datos requeridos para resolver el problema, las ventajas competitivas de aplicar minería de datos al problema, entre otros.
- **Determinar los objetivos de la minería de datos.** El objetivo de esta tarea es corresponder los objetivos del negocio con metas del proyecto de minería de datos que es preciso alcanzar. Por ejemplo, si el objetivo del negocio fuera el desarrollo de una campaña publicitaria para incrementar la asignación de créditos hipotecarios, la meta de la minería de datos sería determinar el perfil de los clientes respecto a su capacidad de endeudamiento.
- **Producir un plan de proyecto.** La última tarea de esta fase tiene como objetivo desarrollar el plan de proyecto teniendo en cuenta qué pasos se deben seguir y qué procedimientos se emplearán para cada uno de ellos.

1.1.5. Comprensión de los datos

En esta fase se lleva a cabo la recolección y exploración inicial de los datos, con el objetivo de establecer un primer contacto con el problema. Esta fase suele ser crítica en el proyecto, pues el buen entendimiento de los datos tiene como consecuencia una importante reducción en el tiempo global del proyecto, además de incrementar las garantías de éxito.

Las tareas genéricas que se deben desarrollar en esta fase de comprensión de los datos son recolectar datos iniciales, describir los datos, explorar los datos y verificar la calidad de los datos.

- **Recolectar datos iniciales.** Tiene como objetivo principal la recolección de los datos iniciales y su adecuación para su posterior procesamiento. Se deben elaborar informes que incluyan una lista de los datos adquiridos, su localización, las técnicas utilizadas en su recolección, así como los problemas y las soluciones encontradas.

- **Describir los datos.** Se deben describir formalmente los datos obtenidos: el número de conjuntos de datos, sus filas y columnas, su identificación, el significado de cada columna y la descripción rigurosa del formato de los datos.
- **Explorar los datos.** El objetivo de esta tarea es descubrir la estructura y distribución general de los datos. Se aplican técnicas básicas de estadística descriptiva que revelan propiedades de los datos. Se crean tablas de frecuencia e histogramas, y se construyen gráficas de distribución. Se crea un informe de exploración de datos.
- **Verificar la calidad de los datos.** Se lleva a cabo una verificación de los datos para determinar su consistencia, la cantidad y distribución de los valores nulos o valores fuera de rango que puedan provocar ruido en el modelado posterior. El objetivo es detectar el grado de completitud, consistencia y corrección de los datos.

1.1.6. Preparación de los datos

En esta fase se trata de seleccionar, limpiar y generar conjuntos de datos correctos, organizados y preparados para la fase de modelado. Aunque ciertos paradigmas de técnicas de minería de datos requieran cierto formato específico en los datos, en la gran mayoría de los casos, en la fase de modelado se acepta una única estructura y formato de datos, la cual estudiaremos en el apartado 1.3 de este capítulo.



Esta es una fase sumamente crítica en un proyecto de minería de datos. Los errores en los datos que pasan inadvertidos y que no son resueltos en esta fase se trasladan hasta la fase de modelado, lo cual perjudica la exactitud de los modelos, cuya causa, además, suele ser muy difícil de encontrar. Incluso es posible entregar al cliente resultados basados en datos que aún contienen errores no detectados. Por esta razón, esta fase es crucial y generalmente demanda siempre el mayor esfuerzo y tiempo del proyecto.

Las tareas genéricas que se deben realizar en esta fase son las siguientes:

- **Seleccionar los datos.** Se selecciona un subconjunto de datos considerando la calidad y consistencia de los datos, así como las limitaciones en su cantidad o en los tipos de datos que están relacionados con las técnicas de minería de datos que se utilizarán en el modelado.
- **Limpiar los datos.** En esta tarea, es posible aplicar un extenso repertorio de técnicas con el fin de mejorar la calidad y consistencia de los datos para la fase de modelado. Algunas de estas técnicas pueden ser la selección de columnas, la reducción del volumen de filas de datos, la normalización de los datos, la discretización de columnas numéricas, la detección y corrección de valores anómalos (*outliers* en inglés) o la imputación de valores ausentes, entre otras.
- **Construir datos.** En esta tarea se extrae información de los conjuntos de datos originales y se crean nuevas filas y columnas con el objetivo de exponer conjuntos de datos con mayor representatividad para la fase de modelado.



Ejemplo

Durante la construcción de datos, la creación de nuevas columnas se puede basar en el cálculo mediante fórmulas que conjugan otras columnas.

Supongamos que en los datos originales existen columnas para la medición de los dos lados de la planta de una vivienda. Se puede crear una nueva columna con el producto de ambas, de forma que contenga la superficie de la vivienda. Esta nueva columna puede aportar información más relevante que las medidas de los lados de la planta para un modelo que pretende estimar el precio de una vivienda.

- **Integrar los datos.** Esta fase incluye todas las operaciones necesarias sobre los conjuntos de datos independientes para reunir su información de forma racional en un único conjunto de datos. El objetivo es crear conjuntos de datos fusionados y homogéneos. En esta fase se pueden fusionar filas, columnas o incluso tablas completas.
- **Formatear los datos.** Consiste en transformar la forma de los datos sin modificar su contenido, con el fin de habilitar el uso de determinadas técnicas de minería de datos o cumplir con ciertas interfaces funcionales, por ejemplo, con una **interfaz de programación de aplicaciones** (API, de *application programming interface* en inglés). Las operaciones concretas de esta tarea podrían ser, entre otras, las siguientes: sustituir tabuladores por comas, eliminar caracteres especiales, borrar espacios extra o acortar cadenas de caracteres.

Dada la importancia de esta fase y su impacto en los proyectos de minería de datos, estudiaremos varias de las técnicas empleadas en la tarea de limpieza de datos con mayor profundidad en el apartado 1.4 de este capítulo.

1.1.7. Modelado

En esta fase se lleva a cabo la creación de modelos de conocimiento a partir de los datos suministrados desde la fase anterior. Estos modelos de conocimiento pueden ser de diversa índole, en función de la tarea de minería de datos que se pretenda resolver. Por ejemplo, se pueden crear modelos de clasificación o regresión con el objetivo de estimar o inferir el valor de una determinada variable.



Esta es la fase central de todo proyecto de minería de datos y el foco principal de interés de la asignatura, en torno a la cual gira la actividad principal que se desarrolla en los proyectos. Dado que todos los proyectos de minería de datos se basan en el aprendizaje automático a partir de los datos, los distintos tipos de modelado que se pueden realizar en esta fase son, en esencia, distintos tipos de aprendizaje. En el apartado 1.2 estudiaremos los distintos tipos de aprendizaje que es posible utilizar.

Selección de los algoritmos de modelado

En esta tarea se aborda el problema de la selección de los algoritmos más adecuados para el conjunto de datos de estudio. Aunque los algoritmos más eficaces se descubren tras un análisis comparativo de resultados en conjuntos de datos de prueba, a veces es conveniente seleccionar un subconjunto de algoritmos candidatos y someterlos a prueba para reducir los tiempos de experimentación. Para ello, hay que comprobar que se cumplen los siguientes criterios:

- Los algoritmos son del tipo de aprendizaje apropiado para el problema.
- Se dispone de datos compatibles para los algoritmos (numéricos, categóricos, cadenas, fechas, valores ausentes, etc.).
- Se tiene conocimiento del funcionamiento de los algoritmos a aplicar, sus bondades y deficiencias.
- Los algoritmos son adecuados para la naturaleza de los datos (distribución y dependencias en los datos). Por ejemplo, ciertos algoritmos, como las redes bayesianas, asumen independencia en las variables.
- Los algoritmos tienen una complejidad computacional adecuada y su implementación arroja tiempos de ejecución aceptables en otros problemas en la literatura para datos de un volumen y características similares.
- Los modelos de conocimiento deben ser interpretables. En determinados problemas, se requiere que los modelos generados por los algoritmos puedan comprenderse con razonable facilidad. Estos modelos se denominan de *caja blanca*. A los que conllevan una interpretación sumamente compleja, se les denomina de *caja negra*. En determinadas ocasiones, aunque estos sean más precisos, se favorece la interpretabilidad y se opta por modelos de caja blanca.

Generar el plan de prueba

Esta tarea también se denomina *diseño experimental* en ciencia de datos y es muy importante llevarla a cabo adecuadamente. Se debe generar un plan para probar la eficacia y la validez de los modelos construidos sobre los datos del problema.

La gran mayoría de algoritmos de aprendizaje automático pueden ser configurados, pues poseen un conjunto de parámetros que determinan las características del modelo que se generará. En esta tarea de generación del plan de prueba, se escogen los valores de los parámetros que se usarán para los algoritmos. En muchas ocasiones, además, se crean listas de valores de parámetros con el fin de experimentar con cada uno de ellos y determinar cuál es la mejor parametrización en función de los resultados obtenidos.

Se determinan aquí también las métricas de evaluación que se calcularán para evaluar la bondad de los modelos y la forma de validación de los algoritmos. En el Capítulo 2 se estudiarán las métricas de evaluación y las formas de validación en aprendizaje supervisado.

Construir los modelos

Se ejecutan los algoritmos seleccionados sobre los datos preparados, utilizando la parametrización de estos y la forma de validación programados en el plan de prueba, con el fin de generar uno o más modelos y calcular las métricas de evaluación. Los resultados totales generados deben estar detallados, organizados y almacenados de forma adecuada para su posterior análisis e interpretación.

Evaluación de resultados

Se analizan las métricas de evaluación obtenidas con el fin de evaluar la bondad de los modelos y garantizar que cumplan con los criterios de éxito de minería de datos definidos al inicio del proyecto. Esta es una evaluación puramente técnica basada en los resultados de modelado. Se resumen los resultados obtenidos, se enumeran las cualidades de los modelos generados y se hace una comparativa y una ranking de eficacia y eficiencia de los algoritmos y los conjuntos de datos del problema.

1.1.8. Evaluación de negocio

Se evalúan los modelos obtenidos, teniendo en cuenta el cumplimiento de los criterios de éxito del problema de negocio planteado al inicio del proyecto y usando el conocimiento del dominio del problema. Además, se deben resumir los hallazgos o conclusiones más importantes que se pueden extraer de los resultados experimentales obtenidos en todas las fases anteriores, tanto en la comprensión y preparación de datos como en la comprensión del problema y el modelado.

Si los modelos y hallazgos obtenidos cumplen con las expectativas de negocio, se procede a la siguiente fase, la explotación del modelo. Si no, se evalúa en esta fase si proceder a iterar nuevamente sobre las fases anteriores con el objetivo de encontrar nuevos hallazgos y mejorar los resultados obtenidos por los modelos. Las tareas son las siguientes:

- **Evaluar los resultados.** En esta tarea se realiza la evaluación formal de los resultados obtenidos en las fases anteriores del proyecto, teniendo en cuenta los criterios de éxito de negocio definidos al inicio del proyecto. Se intentan explicar las causas que provocaron los grados de éxito alcanzados.
- **Revisar el proceso.** Se lleva a cabo en esta tarea una revisión de todo el proceso llevado a cabo en el proyecto, con el fin de encontrar si existen errores o riesgos que puedan afectar al éxito. Esta es una tarea de aseguramiento de la calidad.
- **Determinar siguientes pasos.** Según los resultados de la evaluación y la revisión del proceso, el equipo del proyecto decide cómo proceder a continuación. Las decisiones que toman incluyen si hay que finalizar el proyecto y pasar a la implantación, iniciar nuevas iteraciones en el proyecto o generar nuevos proyectos o subproyectos de minería de datos.

1.1.9. Despliegue

En esta fase el conocimiento obtenido se transforma en acciones dentro del proceso de negocio, empleando los modelos construidos y los hallazgos obtenidos en la actividad productiva empresarial. Las tareas que se realizan son planear la implementación, monitorizar y mantener, crear el informe final y revisar el proyecto.

- **Planear la implementación.** En esta tarea se toman los resultados de la evaluación de negocio y se extrae una estrategia para su implantación. Si se identifica un procedimiento general para crear los modelos, este debe estar documentado para su posterior automatización e implantación.
- **Monitorizar y mantener.** Se deben preparar estrategias para la monitorización y mantenimiento de los modelos puestos en explotación, con el fin de observar cualquier comportamiento anómalo del sistema y corregirlo de forma que no provoque deficiencias en el servicio al cliente.

- **Crear el informe final.** El informe final del proyecto debe ser un resumen de sus puntos principales, sus hallazgos, la experiencia y el conocimiento alcanzados, y los resultados logrados.
- **Revisar el proyecto.** Se realiza una revisión final de todo el proceso llevado a cabo en el proyecto, evaluando las acciones realizadas de forma correcta e incorrecta, con el fin de enumerar las lecciones aprendidas en el proyecto.

1.2. Tipos de aprendizaje

1.2.1. Aprendizaje supervisado

Tal como hemos estudiado en el apartado anterior, el aprendizaje automático, en esencia, consiste en que la máquina pueda aprender patrones en los datos con el fin de crear modelos abstractos que representen la realidad de dichos datos y con ellos resolver problemas. Pues bien, en este apartado vamos a ver qué tipos de problemas podemos resolver mediante el aprendizaje automático.

Fundamentalmente, se distinguen dos categorías de problemas que pueden resolverse mediante aprendizaje automático: aquellos problemas donde existe una variable especial de interés destacada en el conjunto de datos (variable de salida) y aquellos problemas donde no existe tal variable.

El aprendizaje supervisado es la parte del aprendizaje automático que se ocupa del primer tipo de problemas (en los que existe una variable de salida). Por el contrario, el aprendizaje no supervisado es la parte que se ocupa del segundo tipo de problemas (en los que no existe una variable de salida).

En la gran mayoría de los casos, en aprendizaje supervisado se pretende estimar (inferir o predecir) el valor de la variable de salida en función del resto de variables. Así pues, los problemas en aprendizaje supervisado suelen ser problemas de inferencia (o predicción).



En los problemas de inferencia con aprendizaje supervisado, es preciso que todas las filas de datos tengan un valor para la variable de salida. Si en algunas filas no hay valor en la variable de salida, no hay aprendizaje a partir de ellas, pues no se puede determinar la relación entre la variable de salida y el resto de las variables.

Existen dos grandes tipos de problemas de predicción en función del tipo de dato de la variable salida:

- **Regresión.** Si la variable de salida es numérica (número real).
- **Clasificación.** Si la variable es categórica (número discreto o etiqueta nominal).

En los capítulos 3 y 4 de la asignatura estudiaremos algunos de los algoritmos clásicos para problemas de regresión y clasificación, respectivamente, y resolveremos problemas de ambos tipos.

Aparte de los problemas de predicción (regresión o clasificación), en aprendizaje automático pueden resolverse otros tipos de problemas, tales como la discretización supervisada, la corrección de outliers mediante modelos de conocimiento, la selección de instancias o la selección de atributos, entre otros.

En el apartado 1.4.5 de este capítulo estudiaremos, de forma resumida, en qué consiste la selección de atributos y alguna de las técnicas supervisadas más utilizadas.

1.2.2. Aprendizaje no supervisado

El aprendizaje no supervisado, como hemos explicado en el apartado anterior, es la parte del aprendizaje automático que se ocupa de los problemas donde no hay una variable especial de interés destacada en el conjunto de datos (variable de salida).

Por este motivo, los problemas de aprendizaje no supervisado no son de inferencia o predicción, pues si hubiera inferencia habría una variable de interés que inferir. Los dos problemas habituales en aprendizaje no supervisado son los siguientes:

- **Clustering.** Creación de grupos de datos similares, de gran importancia para dividir los problemas, y analizar y modelar cada grupo de forma separada.
- **Reglas de asociación.** Creación de reglas que asocian las variables entre sí mediante relaciones causa-efecto (antecedente y consecuente), muy útil para estudiar las dependencias entre los datos y para los sistemas de recomendación.

Aparte de los dos problemas anteriores, existen numerosas técnicas no supervisadas de aprendizaje automático para otros problemas, entre ellas las de reducción de la dimensionalidad.

La **reducción de la dimensionalidad** consiste en reducir la cantidad de datos antes de la fase de modelado, con el objetivo de mejorar, o al menos mantener, la eficacia de los modelos, y al mismo tiempo incrementar su eficiencia (por la reducción del volumen de datos). Esta reducción de la dimensionalidad puede ser tanto supervisada como no supervisada y se puede llevar a cabo o bien mediante la eliminación de filas o columnas, o bien transformando el espacio de valores a otro con menor número de filas o columnas.

Como hemos comentado anteriormente, en el apartado 1.4.5 de este capítulo estudiaremos técnicas supervisadas para realizar selección de atributos, tarea que supone una reducción de la dimensionalidad de los datos por eliminación de columnas.

Por otra parte, en la asignatura de *Aprendizaje no supervisado* se estudiarán tanto los problemas de clustering y reglas de asociación como la reducción de la dimensionalidad mediante técnicas no supervisadas.

1.2.3. Otros tipos de aprendizaje

Aparte del aprendizaje supervisado y no supervisado, existen otros tipos de aprendizaje automático, tales como el aprendizaje semisupervisado y el aprendizaje por refuerzo.

Con relación al **aprendizaje semisupervisado**, conviene recordar los problemas de predicción del aprendizaje supervisado, en el cual es preciso que todas las filas tengan un valor para la variable de salida. En cambio, en aprendizaje semisupervisado, los problemas que se resuelven involucran una variable de salida, aunque no todas las filas de datos poseen un valor para la variable de salida.

Dado que el volumen de datos sin valor para la variable de salida puede ser elevado, en lugar de eliminar las filas sin valor en la variable de salida, las técnicas de aprendizaje semisupervisado pueden construir modelos que tengan en cuenta todas las filas de datos y aun así resuelvan problemas de predicción.

En la asignatura de *Aprendizaje no supervisado* se abordarán algunas de las técnicas clásicas de aprendizaje semisupervisado.

Por otra parte, el **aprendizaje por refuerzo** consiste en un aprendizaje en el que el algoritmo recibe algún tipo de valoración acerca de la idoneidad de la respuesta que produce el modelo aprendido. Cuando la respuesta es correcta, el aprendizaje por refuerzo se parece al aprendizaje supervisado. En ambos tipos de aprendizaje, el modelo (denominado habitualmente *aprendiz* en aprendizaje por refuerzo) recibe información con detalle de su acierto.

Sin embargo, ambos tipos de aprendizaje (aprendizaje supervisado y aprendizaje por refuerzo) difieren significativamente en las respuestas erróneas, es decir, cuando el aprendiz responde de forma inadecuada.

En estos casos, en aprendizaje supervisado el modelo recibe el valor exacto de la variable de salida, con lo cual pueden computarse exactamente medidas de error. Sin embargo, en aprendizaje por refuerzo solo se comunica al aprendiz que su comportamiento ha sido inadecuado y, en algunas ocasiones, una estimación de la cantidad de error cometido.

1.3. Estructura de datos

1.3.1. Estructuras y tipos en ciencia de datos

En ciencia de datos, la ciencia en la que se incluye la minería de datos y, por ende, el aprendizaje automático, existe una estructura de datos fundamental en la que se apoyan la gran mayoría de algoritmos: la **tabla de datos**, también conocida como *conjunto de datos* (*dataset* en inglés).

La tabla de datos es una estructura en dos dimensiones (con filas y columnas), al igual que las matrices en matemáticas. No obstante, a diferencia de las matrices, las tablas de datos pueden contener diferentes tipos de datos (números, cadenas, fechas...). A pesar de ello, cada columna de una tabla de datos debe contener solo datos del mismo tipo.

Aunque la tabla de datos es la estructura principal utilizada en aprendizaje automático, también se utilizan otras estructuras en contextos más particulares, como los grafos, las series temporales, los datos espacializados (o georeferenciados), las secuencias (como textos, secuencias biológicas, etc.), los datos jerarquizados (árboles), etc. Se recomienda consultar los capítulos 12 a 19 de Aggarwal (2015) para aprender sobre el manejo y las particularidades de estas y otras estructuras de datos.

Cada dato (o celda) de una tabla de datos puede ser de uno de los siguientes tipos de datos:

- **Numérico.** Números reales en matemáticas, también llamados valores continuos (por ejemplo, 5,18). En Python, se corresponde con los tipos float o double. Sus valores tienen una relación de orden implícita (por ejemplo, 3,67 es menor que 4,55).
- **Categórico.** Valores discretos, también llamados valores nominales (por ejemplo, “Rojo” o “Alto”). Pueden ser cadenas, booleanos o números enteros. Pueden tener o no relación de orden entre sus valores (por ejemplo, “Bajo” es menor que “Alto”, pero “Rojo” no es menor ni mayor que “Verde”).

1.3.2. Instancias, clase, atributos y modelos

Hasta ahora hemos dicho que los conjuntos de datos están compuestos de filas y columnas (variables) y que en aprendizaje supervisado existe una variable destacada: la variable de salida.

A partir de este punto en adelante, debemos cambiar esta nomenclatura e introducir la que realmente se utiliza en minería de datos. En concreto, llamaremos **instancias** (también ejemplo, muestra, observación, punto o prototipo) (*instance, data point, sample* en inglés) a cada fila de la tabla de datos.

Por otra parte, llamaremos **clase** (también etiqueta, variable objetivo, respuesta, efecto, consecuencia) (*class, target, response* en inglés) a la variable de salida utilizada en aprendizaje supervisado. Por último, llamaremos **atributo** (también característica, causa, variable de entrada, variable explicativa) (*attribute, feature, predictor, predictive variable* en inglés) a cada una de las demás variables de la tabla de datos, excepto la clase.

Por tanto, podemos formular el cometido de un algoritmo de predicción en aprendizaje supervisado como la extracción inteligente de un **modelo** (también modelo de conocimiento, patrón) (*knowledge model, pattern* en inglés) a partir de las relaciones encontradas entre los atributos y la clase en las instancias de la tabla de datos.

En la Tabla 1 se muestra la estructura de la tabla de datos que usaremos en la asignatura y su nomenclatura.

Tabla 1

Estructura de datos basada en instancias, atributos y clase para minería de datos

		Atributos				Clase
		x_1	...	x_p	y	
X	e_1	$x_{1,1}$...	$x_{1,l}$	y_1	
	
	e_n	$x_{n,1}$...	$x_{n,p}$	y_n	

La tabla de datos tiene p atributos y n instancias. X representa la matriz de valores de atributos, y representa el vector de valores de la clase, e_1, \dots, e_n son las instancias (vectores fila), x_1, \dots, x_p son los atributos (vectores columna) y x_{ij} son los valores de los atributos para cada instancia (con $i = 1, \dots, n$ y $j = 1, \dots, p$). Tal como mencionamos anteriormente, cada atributo x_j ($j = 1, \dots, p$) puede ser de un tipo diferente (numérico o categórico) y, por tanto, todos sus valores x_{ij} ($i = 1, \dots, n$) son de dicho tipo.

Con respecto a la clase, si esta es numérica (regresión), $y_i \in \mathbb{R}, i = 1, \dots, n$. Por el contrario, si la clase es categórica (clasificación), definimos su conjunto de valores (su dominio) como $y_i \in \{c_1, \dots, c_m\}, i = 1, \dots, n$. Cada valor c_j (con $j = 1, \dots, m$) del dominio puede ser un símbolo, una cadena o un número entero.

En programación, suele usarse una variable de tipo matriz o **data frame** (dos dimensiones), de nombre X , para almacenar los valores de los atributos de la tabla de datos. Asimismo, se suele emplear una variable de tipo vector (una dimensión), de nombre y , para almacenar los valores de la clase.

1.3.3. Características de los conjuntos de datos

Es habitual observar y anotar ciertas propiedades de los conjuntos de datos al inicio de los proyectos de minería de datos con el fin de catalogar o enfocar el tipo de problema que se pretende resolver.

Aunque es posible extraer un gran número de características de las tablas de datos, es habitual observar al menos las siguientes:

- Número de instancias.
- Número de atributos.
- Nombre, tipo y breve descripción de cada atributo.
- Tipo de datos de la clase, si tiene, y dominio de valores.
- Nombre y breve descripción de la clase, si tiene.
- Cantidad de valores ausentes.

Por ejemplo, el conjunto de datos de especies de flores iris (“iris” dataset) posee las características que se muestran en la Tabla 2.

Tabla 2

Características del conjunto de datos “iris”

Número de instancias	150
Número de atributos	4
Descripción de atributos	sepal-length. Numérico. Longitud del sépalo. sepal-width. Numérico. Anchura del sépalo. petal-length. Numérico. Longitud del pétalo. petal-width. Numérico. Anchura del pétalo.
Tipo de datos de la clase	Categórico: {iris-setosa, iris-versicolor, iris-virginica}
Descripción de la clase	Nombre: class. Descripción: Especie de la planta.
Valores ausentes	0



Enlace de interés

Pueden consultarse las características de varios datasets de ejemplo en la página de utilidades de carga de datasets de la librería scikit-learn de Python, más concretamente, en las secciones 5.13 a 5.18. https://scikit-learn.org/stable/datasets/toy_dataset.html

1.4. Limpieza de datos

1.4.1. Introducción a la limpieza de datos

La limpieza de datos es una tarea fundamental dentro de los proyectos de minería de datos y se encuentra enmarcada dentro de la fase de preparación de los datos de la metodología CRISP-DM (véase el apartado 1.1.6 de este capítulo).



La limpieza o preprocessado de datos consiste en la eliminación de defectos en los datos y la adaptación de estos de cara a la fase posterior de modelado, con el objetivo de producir la forma de los datos que mejor exponga sus patrones, de la forma más nítida posible. Al igual que el aprendizaje automático, las técnicas de limpieza de datos pueden ser tanto supervisadas como no supervisadas, dependiendo de si estas requieren o no de la presencia de una clase en el conjunto de datos.

Por otra parte, en función de la parte de la tabla de datos sobre la que actúe la técnica de limpieza de datos, distinguimos entre técnicas de filas, de columnas o de matriz completa. Por ejemplo, una técnica de limpieza que normaliza los valores de los atributos es una técnica de columna, pues su actuación se basa en todos los valores de una columna cada vez.

En este apartado 1.4 estudiaremos algunas de las tareas típicas dentro de la limpieza de datos, tales como la normalización y estandarización de atributos, la detección de outliers, la imputación de valores ausentes y la selección de atributos. Existen, no obstante, otras numerosas tareas de limpieza de datos, como la discretización, la generación de atributos o la selección de instancias, entre otras.

1.4.2. Normalización y estandarización

Normalización

La normalización de un conjunto de datos consiste en transformar los valores de sus atributos numéricos al rango continuo $[0, 1]$ mediante una operación matemática llamada **homotecia**. Esta transformación se lleva a cabo para cada columna (atributo) de forma independiente. En concreto, para un atributo x_i , su normalización x_i^N se define como:

$$x_i^N = \left\{ \frac{x_{i,j} - \min(x_i)}{\max(x_i) - \min(x_i)}, \forall j \in 1, \dots, n \right\}$$

Nótese que la normalización hace que a cada valor de un atributo se le reste el mínimo de dicho atributo (columna) y se divida por la diferencia entre máximo y mínimo. La operación de resta se denomina *centrado*, mientras que la división se denomina *escalado*. Por tanto, diremos que la normalización, al igual que otras transformaciones como la estandarización (que veremos a continuación), centra y escala los datos.

Una vez que un atributo está normalizado, suele interesar desnormalizar valores a su rango original. Para calcular el valor x desnormalizado a partir de un valor x^N normalizado sobre el atributo x_i , basta aplicar la operación en sentido contrario:

$$x = x^N \cdot (\max(x_i) - \min(x_i)) + \min(x_i)$$

La normalización es una transformación muy recomendable cuando se pretenden usar cálculos de distancias o productos escalares entre ejemplos. En caso de no aplicar normalización, la diferencia de escalas de valores entre los atributos puede hacer que uno tenga más influencia que otro (por ejemplo, dos atributos con saldos bancarios en diferentes monedas), cuando ambos atributos tienen realmente la misma importancia.

A continuación, veamos un ejemplo de aplicación de la normalización sobre una tabla de datos artificiales (también llamados *datos sintéticos*) muy sencilla.

</>

```
from sklearn import preprocessing
import numpy as np

# Carga de datos.
X_train = np.array([[ 1., -1.,  2.],
                   [ 2.,  0.,  0.],
                   [ 0.,  1., -1.]])  
  
# Limpieza de datos: normalización.
normalizer = preprocessing.MinMaxScaler()
X_train_norm = normalizer.fit_transform(X_train)  
  
# Salida de resultados.
print(X_train_norm)
```

Programa 1. Normalización de datos.

```
[[0.5      0.        1.        ]
 [1.        0.5       0.33333333]
 [0.        1.        0.        ]]
```

Salida 1. Salida del Programa 1.

En el Programa 1 se ha hecho uso del paquete **preprocessing** de la librería scikit-learn, el cual posee implementadas numerosas técnicas de limpieza de datos. Entre ellas, la normalización se encuentra tanto en la clase **Normalizer** como en **MinMaxScaler**. En el ejemplo, hemos utilizado esta última.

Tal como se aprecia en el Programa 1, hemos definido una matriz de números de tipo **np.array** y se ha creado un objeto llamado **normalizer** de la clase **MinMaxScaler**. Por último, el método **fit_transform()** nos permite crear un modelo de normalización y aplicarlo sobre dicha matriz de datos.

Nótese, por ejemplo, que el primer atributo del conjunto de datos (primera columna de la matriz) tiene como valor mínimo 0 y como máximo 2. Por tanto, en la matriz resultado dichos valores se convierten en 0 y 1, respectivamente. El valor 1, que se encuentra equidistante entre el mínimo y el máximo, se convierte, por tanto, en 0,5.



Enlace de interés

En la siguiente página web se puede consultar más información acerca de la normalización mediante la clase **MinMaxScaler**:

<https://scikit-learn.org/stable/modules/preprocessing.html#scaling-features-to-a-range>

Se puede consultar más información sobre la clase `MinMaxScaler` y sus métodos en la siguiente web de su documentación oficial:

<http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html>

La clase `Normalizer` también permite realizar la normalización de datos. Puede consultar la siguiente página web para ver algunos ejemplos. Si desea reproducir los mismos resultados vistos anteriormente en este apartado, debe usar los parámetros `axis=0` y `norm='max'` en el método `preprocessing.normalize()`.

<https://scikit-learn.org/stable/modules/preprocessing.html#normalization>

Estandarización

La estandarización consiste en transformar la distribución de los valores de los atributos para que estos posean una media y desviación típica determinadas. Esta transformación se lleva a cabo para cada atributo de forma independiente. En concreto, para un atributo x_i , su estandarización x_i^S se define como:

$$x_i^S = \left\{ \frac{x_{i,j} - \text{media}(x_i)}{\text{desv}(x_i)}, \forall j \in 1, \dots, n \right\}$$

A cada valor del atributo se le resta la media aritmética de los valores originales del atributo y se divide por la desviación típica de estos. Al igual que la normalización, la operación de resta es el centrado, que, en el caso de la estandarización, en lugar de centrarse en el mínimo (normalización), se centra en la media.

Análogamente, la división es el escalado. El escalado en normalización comprende el rango completo de valores originales (máximo menos mínimo).

El escalado en estandarización comprende tan solo la amplitud definida por la desviación típica. Por consiguiente, los valores estandarizados no están acotados en un rango definido (como sucedía con la normalización).

Una vez un atributo está estandarizado, suele interesar desestandarizar valores a su distribución original. Para calcular el valor x desestandarizado a partir de un valor x_i^S estandarizado sobre el atributo x_i , basta aplicar la operación en sentido contrario:

$$x = x_i^S \cdot \text{desv}(x_i) + \text{media}(x_i)$$

La estandarización, al igual que la normalización, es una transformación muy recomendable cuando se aplican modelos de aprendizaje automático, pues las diferencias en escalas y rangos de valores entre los atributos perjudica el entrenamiento de modelos. En concreto, aquellos atributos con mayores valores tienen una influencia mayor sobre la función objetivo en el aprendizaje, como veremos en los capítulos 3 y 4.

A continuación, en el Programa 2, vemos un ejemplo de aplicación de la estandarización sobre la misma tabla de datos artificiales del Programa 1.

```

from sklearn import preprocessing
import numpy as np

# Carga de datos.
X_train = np.array([[ 1., -1.,  2.],
                   [ 2.,  0.,  0.],
                   [ 0.,  1., -1.]])  
  

# Limpieza de datos: estandarización.
standardizer = preprocessing.StandardScaler()
X_train_std = standardizer.fit_transform(X_train)  
  

# Salida de resultados.
print(X_train_std)
  
```

Programa 2. Estandarización de datos.

```

[[ 0.          -1.22474487  1.33630621]
 [ 1.22474487  0.          -0.26726124]
 [-1.22474487  1.22474487 -1.06904497]]
  
```

Salida 2. Salida del Programa 2.

En este programa se ha vuelto a hacer uso del paquete `preprocessing` de la librería scikit-learn. La estandarización se encuentra en la clase `StandardScaler`.

Tal como se aprecia en el programa, tenemos la misma matriz de números vista anteriormente y se ha creado un objeto llamado `standardizer` de la clase `StandardScaler`. Por último, el método `fit_transform()` nos crea el modelo de estandarización y lo aplica sobre la matriz de datos.

Nótese, por ejemplo, que el primer atributo del conjunto de datos tiene media 1 y desviación típica 0,8165. Por tanto, en la matriz resultado el valor 1 se convierte en 0, pues se le resta la media. El valor 2, por ejemplo, lleva el siguiente cálculo:

$$\frac{2 - \text{media}(a_1)}{\text{desv}(a_1)} = \frac{2 - 1}{0,8165} = 1,2247$$



Enlaces de interés

En la siguiente página web se puede consultar más información acerca de la estandarización mediante la clase `StandardScaler`:

<https://scikit-learn.org/stable/modules/preprocessing.html#standardization-or-mean-removal-and-variance-scaling>

Se puede consultar más información sobre la clase `StandardScaler` y sus métodos en la siguiente web de su documentación oficial:

<http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html>

1.4.3. Detección de outliers

Uno de los problemas más habituales en los conjuntos de datos es la presencia de valores anómalos que se encuentran fuera de la distribución natural de los datos.

Se trata de los valores outliers, o simplemente outliers. Estos valores pueden reducir la eficacia de los modelos de aprendizaje automático y debemos limpiar los conjuntos de datos previamente para evitarlo.



Determinar qué valores se pueden considerar outliers dentro de un conjunto de datos no es un problema trivial, pues la frontera entre outliers y no outliers generalmente es difusa y, aun así, es preciso determinar un umbral a partir del cual considerar cada muestra como outlier o no.

Existen en la literatura numerosos algoritmos que permiten detectar la presencia de outliers en un conjunto de datos para marcar cada ejemplo como outlier o no outlier. Por razones de extensión, en esta asignatura veremos de forma resumida solo un algoritmo: el algoritmo de envolvente elíptica.

En el algoritmo de envolvente elíptica (*elliptic envelope* en inglés) se presupone que los datos siguen una distribución normal (distribución gaussiana). Si los datos no siguen dicha distribución, los resultados podrían no ser los deseados. Es decir, los outliers determinados por el algoritmo podrían no serlos e, igualmente, ejemplos no estimados como outliers sí que podrían serlo.

El algoritmo de envolvente elíptica crea superficies elípticas en torno a la nube de puntos con valores normales en el conjunto de datos. Conforme la superficie elíptica es mayor, mayor distancia tienen los puntos que pertenecen a esta con respecto a los puntos con valores normales y, por tanto, mayor probabilidad de que estos sean outliers (véase Figura 1).

Para construir las superficies elípticas, se centran y escalan los datos con respecto a su mediana y rango intercuartílico (IQR), respectivamente. Esto es, para cada atributo se resta el valor de su mediana (centrado) y se divide por el rango intercuartílico (escalado).

Nótese que el uso de la mediana evita la distorsión que los outliers introducen en la media aritmética. Una vez realizada esta operación, todo valor de dicho atributo que supere, por exceso o por defecto, la mediana más un determinado umbral (habitualmente, $1,5 \cdot \text{IQR}$ o $3 \cdot \text{IQR}$) será considerado como valor outlier.

A continuación, vemos un ejemplo de aplicación de la detección de outliers sobre el conjunto de datos de ejemplo “outliers.csv”, suministrado en la asignatura.



```
import pandas as pd
from scipy import stats
from sklearn.covariance import EllipticEnvelope
from limpieza_funciones import grafico_outliers
```

```
import pandas as pd
from scipy import stats
from sklearn.covariance import EllipticEnvelope
from limpieza_funciones import grafico_outliers

# Carga de datos.
df = pd.read_csv("outliers.csv")

# Limpieza de datos: detección de outliers.
outlier_method = EllipticEnvelope().fit(df)
scores_pred = outlier_method.decision_function(df)
threshold = stats.scoreatpercentile(scores_pred, 25)

# Dibujar gráfica de outliers.
grafico_outliers(df, outlier_method, 150, threshold, -7, 7)
```

Programa 3. Detección de outliers.

La gráfica generada por el Programa 3 es:

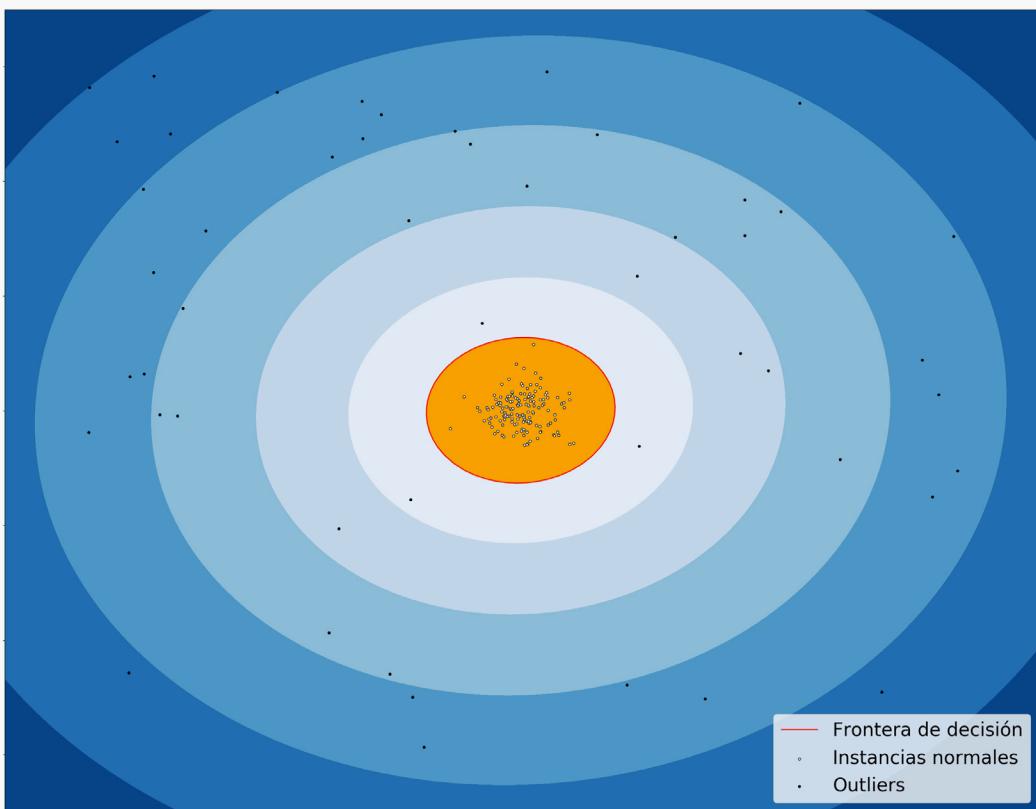


Figura 1. Gráfica de detección de outliers.

Para generar la gráfica se necesita importar la función `grafico_outliers()`, que se encuentra en el archivo “`limpieza_funciones.py`”, suministrado en la asignatura.

En el Programa 3, se importan, en primer lugar, los datos del archivo “outliers.csv”. Este es un conjunto de datos artificial que posee 2 atributos y 200 instancias, de las cuales las primeras 150 contienen valores normales y las 50 últimas son outliers. Los valores de ambos atributos se encuentran en el rango $[-7, 7]$.

A continuación, se ajusta un modelo de detección de outliers mediante la clase `EllipticEnvelope` de scikit-learn, la cual implementa el algoritmo de envolvente elíptica que hemos explicado anteriormente. Con el método `decision_function()` obtenemos la función que puntúa la normalidad de una instancia con respecto a la distribución centrada y escalada en la mediana e IQR, respectivamente.

Esta función se usa, a continuación, para establecer un umbral o frontera al percentil 25 %, el cual permite decidir si una muestra es o no outlier. Para ello, se usa la función `scoreatpercentile()`. Finalmente, se invoca la función `grafico_outliers()`, que genera la Figura 1.



Enlace de interés

Se pueden explorar otros algoritmos de detección de outliers en la siguiente página web:

http://scikit-learn.org/stable/modules/outlier_detection.html

En la siguiente web se encuentra un ejemplo de aplicación de varios algoritmos de detección de outliers de forma comparativa (incluye código Python):

https://scikit-learn.org/stable/auto_examples/miscellaneous/plot_anomaly_comparison.html

1.4.4. Imputación de valores ausentes

En numerosas ocasiones, los conjuntos de datos suelen estar incompletos, y varias de las instancias no poseen valores para todos sus atributos. Este es un fenómeno natural, pues la información no siempre puede recabarse, por algún motivo no estaba disponible o era errónea y se eliminó.

Los datos que faltan son los valores ausentes (*missing values* en inglés), cuyo valor en programación suele ser `NaN`, `NA`, `N/A` o `Null`, entre otros (según el lenguaje de programación). Por ejemplo, en Python, los valores ausentes se pueden representar con `np.nan` (NumPy not-a-number).

Hay muchas implementaciones de algoritmos de aprendizaje automático que no toleran los valores ausentes, como la gran mayoría en la librería scikit-learn. Por esta razón, se requiere limpiar el conjunto de datos previamente con el objetivo de hacer desaparecer dichos valores ausentes.

La opción más sencilla podría ser eliminar todas las instancias que contengan algún valore ausente, pero esto puede suponer una pérdida importante de información.

Por consiguiente, los valores ausentes deberían ser reemplazados por valores concretos. La pregunta es: ¿cuál sería el valor más adecuado para sustituir un valor ausente?

Existen numerosas técnicas que pretenden determinar el valor más adecuado para un valor ausente. Estas técnicas se denominan *imputación* (o *estimación*) de valores ausentes. Una de las técnicas más simples consiste en sustituir los valores ausentes por la media aritmética, mediana o moda del atributo en el que se encuentran. Si el atributo es numérico, entonces se suele usar la media aritmética o mediana. Si el atributo es categórico, se suele usar la moda. Otras técnicas más complejas plantean el problema como de aprendizaje supervisado e incluyen la creación de un modelo de conocimiento para la estimación de valores ausentes.

A continuación, mostramos en el Programa 4 un ejemplo de aplicación de la imputación de valores ausentes en una tabla simple de datos artificiales utilizando la técnica de sustitución por la media aritmética.

```
</>
import numpy as np
from sklearn.preprocessing import Imputer

# Carga de datos.
a = [[1, 2], [np.nan, 3], [7, 6]]

# Limpieza de datos: imputación valores ausentes (modelo).
imp = Imputer(missing_values='NaN', strategy='mean', axis=0)
imp.fit(a)

# Limpieza de datos: imputación valores ausentes (aplicación).
b = [[np.nan, 10], [6, np.nan], [8, 1]]
print(imp.transform(b))
```

Programa 4. Imputación de valores ausentes.

```
[[ 4.        10.       ]
 [ 6.        3.66666667]
 [ 8.        1.        ]]
```

Salida 4. Salida del Programa 4.

En el Programa 4 construimos una lista de listas con nombre `a`, que será el conjunto de datos sobre el que el estimador de valores ausentes calculará la media aritmética de cada atributo. Para ello, creamos un objeto `imp` de la clase `Imputer`, que configuramos para que reconozca los valores ausentes por `np.nan` (`NaN`), utilice la media aritmética para imputar y que aplique a cada atributo (`axis=0`). Ajustamos el modelo de estimación de valores ausentes `Imputer` al conjunto de datos a mediante el método `fit()`.

Aplicamos la imputación de valores ausentes sobre el conjunto de datos `b` mediante el método `transform()`. Los valores imputados obtenidos son 4 y 3,66666667, correspondientes a los valores `np.nan` del conjunto `b`. Nótese que 4 y 3,66666667 son las medias aritméticas de los atributos del conjunto `a`.



Enlace de interés

Se puede consultar más información sobre la clase `Imputer` y sus métodos en la siguiente web de su documentación oficial.

<https://scikit-learn.org/0.18/modules/generated/sklearn.preprocessing.Imputer.html>

1.4.5. Selección de atributos

La selección de atributos (*feature selection* en inglés) es una subtarea dentro de la tarea de reducción de dimensionalidad, la cual, a su vez, se incluye dentro de la limpieza de datos. Las técnicas de selección de atributos pueden ser tanto supervisadas como no supervisadas, en función de si estas consideran o no la existencia de una clase en el conjunto de datos.

Todas las técnicas de selección de atributos necesitan de un **algoritmo de evaluación de atributos** que les permita determinar la bondad de estos con el fin de seleccionar aquellos que puedan ser candidatos para ser eliminados. Existen numerosos algoritmos de evaluación de atributos, dependiendo de cómo calculen la métrica de bondad, pero existen dos categorías: algoritmos de evaluación de atributos individuales (o univariantes) y algoritmos de evaluación de conjuntos de atributos (o multivariantes).



Nótese que la relevancia, o poder predictivo, que un atributo puede tener de cara a la predicción de una clase (problema de inferencia mediante aprendizaje supervisado) puede depender de la existencia de otros atributos. Esto es, un atributo puede recibir una métrica baja de bondad, por su débil relación matemática con la variable clase y , sin embargo, ser de gran importancia cuando contribuye, junto a otros, a la predicción de esta. Por esta razón, los algoritmos de evaluación de conjuntos de atributos, aunque mucho más costosos computacionalmente que los de atributos individuales, suelen seleccionar atributos que, en conjunto, son más relevantes.

En el caso de la selección de atributos supervisada, el algoritmo de evaluación puede estar definido solo para clasificación o para regresión, o bien para ambos tipos de aprendizaje. Por ello, es importante tener en cuenta la aplicabilidad del algoritmo deseado al conjunto de datos y seleccionar uno que sea compatible.

Cuando se lleva a cabo una selección de atributos mediante un algoritmo de evaluación de conjuntos de atributos, se necesita también disponer de un **algoritmo de búsqueda**, debido a que el espacio de posibles subconjuntos que hay que explorar es exponencial con respecto al número de atributos inicial del conjunto de datos. Una exploración exhaustiva es inviable, pues el problema es NP-completo. En la asignatura *Algoritmos de optimización* se abordarán algunos de los algoritmos clásicos de búsqueda, los cuales pueden adaptarse para ser usados en selección de atributos multivariante.

Una de las técnicas no supervisadas más habituales de selección de atributos es la eliminación por baja varianza. La idea es eliminar aquellos atributos cuyos valores sean prácticamente constantes en el conjunto de datos, esto es, que no varíen significativamente. En algunos contextos, este tipo de atributos no suelen aportar ninguna información de cara a la predicción de la clase.

A continuación, mostramos en el Programa 5 un ejemplo de aplicación de la selección de atributos mediante eliminación por baja varianza sobre una tabla simple de datos artificiales.

```
from sklearn.feature_selection import VarianceThreshold

# Carga de datos.

X = [[0, 0, 1], [0, 1, 0], [1, 0, 0],
      [0, 1, 1], [0, 1, 0], [0, 1, 1]]

# Limpieza de datos: selección de atributos.

sel = VarianceThreshold(threshold = (.8 * (1 - .8)))

print(sel.fit_transform(X))
```

Programa 5. Selección de atributos no supervisada mediante eliminación por baja varianza.

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

Salida 5. Salida del Programa 5.

En el Programa 5 se construye una tabla de datos con tres atributos, de los cuales el primero es prácticamente constante (salvo un valor a 1, el resto —5 de 6 veces— vale 0). La clase que implementa este tipo de selección en scikit-learn es **VarianceThreshold**. El umbral utilizado en este ejemplo es 0,8, el cual deja pasar un atributo si varía al menos un 20 % a través de las instancias. Tras la aplicación de la selección, el conjunto de datos resultante solo contiene los dos últimos atributos (véase Salida 4).

Estudiemos a continuación dos algoritmos supervisados de evaluación univariante de atributos, basados en las métricas F-test e información mutua, respectivamente.

La métrica F-test mide el grado de dependencia lineal entre dos atributos y es útil para detectar atributos correlacionados linealmente. La métrica de información mutua mide cualquier tipo de dependencia estadística, lineal o no lineal, pero necesita un número mayor de instancias para que sus resultados sean fiables.

A continuación, mostramos en el Programa 6 una comparativa de la aplicación de la selección de atributos supervisada univariante mediante las métricas F-test e información mutua sobre una tabla de datos artificiales.

```
</>  
import numpy as np  
  
import matplotlib.pyplot as plt  
from sklearn.feature_selection import f_regression,  
    mutual_info_regression  
  
# Carga de datos.  
np.random.seed(0)  
X = np.random.rand(1000, 3)  
y = X[:, 0] + np.sin(6 * np.pi * X[:, 1]) +  
    0.1 * np.random.randn(1000)  
  
# Evaluación de atributos: F-Test.  
f_test, _ = f_regression(X, y)  
f_test /= np.max(f_test)  
  
# Evaluación de atributos: información mutua.  
mi = mutual_info_regression(X, y)  
mi /= np.max(mi)  
  
# Graficar distribución de los datos y evaluación de atributos.  
plt.figure(figsize=(15, 5))  
for i in range(3):  
    plt.subplot(1, 3, i + 1)  
    plt.scatter(X[:, i], y, edgecolor='black', s=20)  
    plt.xlabel("$x_{}$".format(i + 1), fontsize=14)  
    if i == 0:  
        plt.ylabel("$y$", fontsize=14)  
        plt.title("F-test={:.2f}, MI={:.2f}".format(f_test[i],  
            mi[i]), fontsize=16)  
    plt.show()
```

Programa 6. Selección de atributos supervisada univariante para regresión. Comparativa de algoritmos de evaluación F-test e Información Mutua.

Las gráficas generadas por el Programa 6 son:

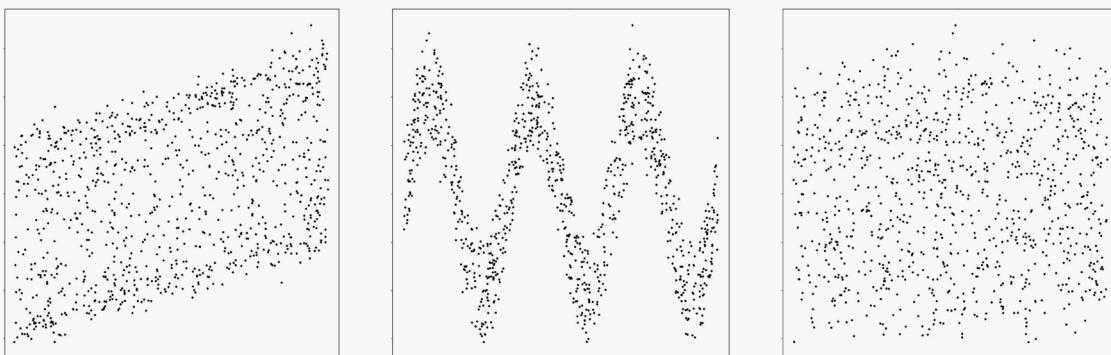


Figura 2. Gráficas generadas por el Programa 6.

En el Programa 6, creamos en la variable X una tabla de datos con 3 atributos y 1000 instancias de números reales aleatorios entre 0 y 1 (con distribución uniforme). También creamos un vector columna para la clase, llamado y , con valores que dependen de X según la siguiente ecuación:

$$y = x_1 + \sin(6\pi x_2) + 0,1 \cdot N(0,1)$$

Nótese que la clase (y) solo depende de los atributos x_1 (lineal) y x_2 (no lineal), no de x_3 (el atributo x_3 es irrelevante). A continuación, mediante las funciones `f_regression()` y `mutual_info_regression()`, se calculan las métricas F-test e información mutua para el conjunto de datos X . Cada métrica se divide por el máximo obtenido en cada atributo, de modo que los valores de ambas estén normalizados (entre 0 y 1) y puedan ser comparados.

En último lugar, se crean tres gráficas que muestran la relación de cada atributo (x_1, x_2, x_3) con la clase (y) (véase Figura 2), además de incluir en su título los valores de las métricas. Debido a que la métrica F-test detecta dependencias lineales, su valor es máximo (1) para el atributo x_1 . Sin embargo, su valor se reduce considerablemente para el atributo x_3 , debido a que la señal de dependencia lineal de este con la clase es baja.

Con respecto a la métrica de información mutua, su valor es máximo en el atributo x_2 , pues la dependencia no lineal (sinusoidal) está muy bien marcada. Finalmente, el atributo x_3 , que no contribuye en el valor de la clase, recibe los valores mínimos (0) de ambas métricas (nótese la nube de puntos sin patrón alguno de la tercera gráfica).

Para ampliar la información relacionada con los contenidos del Capítulo 1 (metodología, tipos de aprendizaje, estructura de datos y limpieza de datos), recomendamos consultar el capítulo 2 de James, Witten, Hastie y Tibshirani (2013), los capítulos 1 y 2 de Aggarwal (2015), los capítulos 1 y 2 de Mueller y Guido (2016), los capítulos 1, 3 y 4 de Sarkar, Bali y Sharma (2018), el capítulo 2 de Kirk (2017) y el capítulo 1 de Watt, Borhani y Katsaggelos (2016).



Capítulo 2

Validación y evaluación

Como hemos estudiado en el Capítulo 1, los algoritmos de aprendizaje automático aprenden un modelo de conocimiento a partir de datos históricos de un problema para llevar a cabo diferentes propósitos, fundamentalmente inferencia o predicción de una clase, pero también reducción de la dimensionalidad, imputación de valores ausentes, detección y corrección de outliers, agrupamiento de muestras o creación de reglas de asociación, entre muchos otros.

A partir de este capítulo en adelante, en esta asignatura nos centraremos únicamente en problemas de inferencia o predicción de una clase (siempre dentro del aprendizaje supervisado).

Para llevar a cabo esta tarea, los algoritmos deben crear un modelo de predicción a partir de un conjunto de datos que relacione los atributos de entrada con la clase de salida, modelo que permitirá posteriormente generar predicciones.

Existe una gran variedad y tipos de algoritmos de aprendizaje supervisado para problemas de inferencia. En esta asignatura estudiaremos cuatro de ellos: regresión lineal múltiple y vecinos más cercanos (para regresión); regresión logística y árboles de decisión (para clasificación).

No obstante, en otras asignaturas del máster se estudiarán también otros algoritmos, tales como las redes neuronales y las redes bayesianas.



Dado un mismo conjunto de datos, no todos los algoritmos de inferencia producen las mismas predicciones: unos algoritmos se comportan mejor con unos datos, mientras que otros algoritmos lo hacen mejor con otros datos. Se hace, por tanto, necesario comparar los resultados de predicción entre diferentes algoritmos aplicados sobre el mismo conjunto de datos. Gracias a la comparación de algoritmos sobre el mismo conjunto de datos, se puede hacer un ranking de bondad y escoger el mejor. Recuérdese que, durante la fase de explotación, se suele utilizar solo el mejor modelo conseguido en la fase de modelado.

Definimos como **conjunto de entrenamiento** (o simplemente entrenamiento, *training* en inglés) el conjunto de datos utilizado para que el algoritmo aprenda y genere su modelo de conocimiento. Una vez entrenado un algoritmo, el modelo debe probarse (para evaluar su bondad). Definimos como **conjunto de test** (también conjunto de prueba o simplemente test, *test* en inglés) el conjunto de datos utilizado para que el algoritmo realice sus predicciones utilizando el modelo aprendido con el conjunto de entrenamiento.

Una vez entrenado un algoritmo, el modelo debe probarse prediciendo la clase de otros ejemplos diferentes a los de entrenamiento. Es decir, el conjunto de test no debe contener ninguna instancia del conjunto de entrenamiento, sino que deben ser ejemplos diferentes. Este aspecto es fundamental para evaluar correctamente un aprendizaje.

Con el aprendizaje automático ocurre como con el aprendizaje de las personas: no podríamos asegurarnos de que un niño ha aprendido una lección si se le pregunta exactamente por los mismos ejemplos del libro que ha estudiado. Cualquiera que aprenda las cosas como un papagayo (aprenderse de memoria la lección) acertaría perfectamente todas estas preguntas, pero no habría aprehendido la esencia de la lección, los conceptos abstractos, y, por tanto, no sabría trasladarlos y aplicarlos en contextos diferentes.



Aprender las cosas de memoria es un problema de aprendizaje que llamaremos sobreajuste (sobreajuste de un modelo a los datos, *overfitting* en inglés). De forma análoga, aprender un concepto demasiado general de los datos también es otro problema de aprendizaje: como el niño que aprende la lección de forma tan esquemática que no profundiza en ningún aspecto. Llamamos este problema subajuste (*underfitting* en inglés).

Tanto el sobreajuste como el subajuste se detectan analizando la diferencia entre los errores cometidos por el modelo cuando predice los mismos ejemplos de entrenamiento (los llamamos **errores de entrenamiento**) con respecto a los errores cuando predice los ejemplos de un test completamente diferente (llamados **errores de generalización**). El sobreajuste de un modelo se caracteriza por arrojar errores de entrenamiento muy bajos, pero errores de generalización altos. Por contra, el subajuste se caracteriza por poseer errores de entrenamiento y generalización altos y muy similares. En la Figura 3 se ilustra este comportamiento:

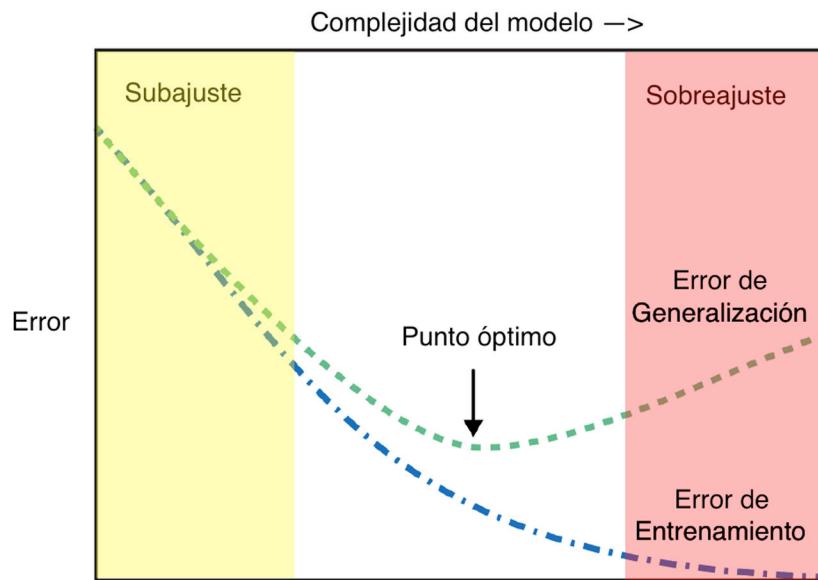


Figura 3. Subajuste y sobreajuste en aprendizaje supervisado.

Tal como se aprecia en la Figura 3, conforme aumentamos la complejidad del modelo, el error de entrenamiento puede reducirse hasta eliminarse por completo. No obstante, los modelos así sobreajustados cometen mayores errores cuando predicen ejemplos no vistos en el entrenamiento (error de generalización). El objetivo es encontrar el punto óptimo de complejidad del modelo en el que el error de generalización se minimiza.



Enlace de interés

En los dos siguientes enlaces se pueden encontrar ilustrados gráficamente el sobreajuste y el subajuste, mostrando la forma que suelen tener los modelos subajustados (*underfitting*), óptimos (*optimal*) y sobreajustados (*overfitting*). En el primer enlace se muestra para regresión, mientras que en el segundo para clasificación.

<https://pythonmachinelearning.pro/a-guide-to-improving-deep-learning-performance/>

<https://www.safaribooksonline.com/library/view/scala-and-spark/9781785280849/3c1c7845-81d-47b9-a54f-c2584fe930b3.xhtml>

Para poder evaluar la bondad de las predicciones y cuantificar los errores producidos por los modelos, debemos definir las métricas (o medidas) de evaluación. Estas métricas son diferentes si el problema es de regresión o de clasificación.

Aunque las diferentes métricas de evaluación más conocidas se verán en detalle en los apartados 2.4 y 2.5, necesitamos introducir aquí, en primer lugar, unas métricas básicas para evaluar predicciones: una para problemas de regresión y otra para clasificación.

La métrica que introducimos para regresión es el error absoluto medio (MAE, de *mean absolute error* en inglés) y se define según la siguiente ecuación:

$$\text{MAE} = \frac{1}{n} \cdot \sum_{i=1}^n |y_i - \hat{y}_i|$$

Donde n es el número de ejemplos del conjunto de test, $y_i \in \mathbb{R}$ es la clase del ejemplo i -ésimo e $\hat{y}_i \in \mathbb{R}$ la predicción del modelo para dicho ejemplo. Nótese, por tanto, que la métrica MAE, para la evaluación de regresión, es una media aritmética de las diferencias en valor absoluto entre los valores reales y predichos.

Por otra parte, la métrica que introducimos para clasificación es la exactitud (*accuracy* en inglés) y es el porcentaje de aciertos con respecto al total de predicciones realizadas. En concreto, se define tal como se muestra en la siguiente ecuación:

$$\text{Exactitud} = \frac{100}{n} \cdot \sum_{i=1}^n y_i = \hat{y}_i$$

En la ecuación de la métrica de exactitud, asumimos que el operador de igualdad ($=$) devuelve 1 si los valores comparados coinciden, y 0 en caso contrario. Por consiguiente, la exactitud mide la tasa porcentual de predicciones acertadas con respecto al total, n , de predicciones realizadas.

Dado que necesitamos dos conjuntos de datos para evaluar los modelos, uno de entrenamiento y otro de test, y que solo disponemos habitualmente de un único conjunto de datos para un problema, debemos crear subconjuntos para entrenamiento y test a partir del conjunto original de datos.

Llamaremos **validación** (*validation* o *resampling* en inglés) al proceso mediante el cual se divide el conjunto de datos en subconjuntos de entrenamiento y test con el objetivo de evaluar de forma adecuada la bondad de los algoritmos de aprendizaje supervisado. En los siguientes apartados 2.1, 2.2 y 2.3 veremos diferentes estrategias para llevar a cabo este proceso de validación.

2.1. Validación hold-out

La primera forma de validación que estudiaremos será la **validación hold-out** (mantener fuera). En ella, el conjunto de datos del problema se divide en dos partes: entrenamiento y test. Para definir esta división, se utiliza habitualmente un porcentaje que indica la proporción de muestras que se destinarán al conjunto de entrenamiento, siendo el resto de las muestras las destinadas para el conjunto de test (denominaremos este procedimiento como *validación hold-out mediante split*). Las proporciones más habituales suelen ser 66 % y 34 % (entrenamiento-test), o bien 70 % y 30 %.



Un aspecto importante a tener en cuenta es la aleatoriedad o no en la selección de las muestras para entrenamiento y test. Pueden existir muchas combinaciones de muestras posibles al dividir un conjunto, por ejemplo, 70 % y 30 %. Es posible hacer una validación hold-out no aleatoria, en la que se escoge el primer 70 % de las muestras para entrenamiento y el último 30 % de las muestras para test (preservando el orden original de las muestras). Este procedimiento es útil en conjuntos con dependencias entre las muestras; por ejemplo, en series temporales.

Sin embargo, también es posible hacer una validación hold-out aleatoria, en la que, en primer lugar, se reordenan aleatoriamente las muestras y, posteriormente, se escoge el primer 70 % de las muestras para entrenamiento y el último 30 % de las muestras para test.

Este procedimiento es más recomendable si no hay dependencias de orden que respetar entre las muestras.

Por otra parte, también es posible que la división entre entrenamiento y test se nos venga dada con el problema y, en tal caso, debemos respetar dicha división. Esto puede suceder debido a un especial interés en focalizar los esfuerzos en la predicción de determinado tipo de muestras de test, o para la comparación de resultados con otros algoritmos o metodologías existentes (en cuyo caso deben usarse los mismos conjuntos de entrenamiento y test). Esta forma de validación también se considera hold-out.

A continuación, mostramos en el Programa 7 un ejemplo de aplicación de la validación hold-out mediante split en un problema de clasificación utilizando el conjunto de datos “iris”, cuyas características se mostraron en la Tabla 2.

```
</>
from sklearn import datasets
from sklearn.dummy import DummyClassifier
from sklearn.model_selection import train_test_split

# Carga de datos.
iris = datasets.load_iris()

# Mostrar características de la tabla de datos.
print("Tabla de datos: %d instancias y %d atributos" % (iris.data.shape[0],
iris.data.shape[1]))
print("Valores de la clase:", set(iris.target))

# Validación: hold-out split 70-30%.
X_train, X_test, y_train, y_test = train_test_split(
    iris.data, iris.target, test_size=0.3, random_state=0)

# Mostrar características de los conjuntos de training y test.
print("Training: %d instancias y %d atributos" % (X_train.data.shape[0], X_
train.data.shape[1]))
print("Test: %d instancias y %d atributos" % (X_test.data.shape[0], X_test.
data.shape[1]))

# Construcción del objeto que contiene el algoritmo de aprendizaje.
clf = DummyClassifier()

# Entrenamiento del algoritmo de aprendizaje.
clf.fit(X_train, y_train)

# Evaluación del algoritmo de aprendizaje.
evaluacion = clf.score(X_test, y_test)
print("Exactitud:", evaluacion)
```

Programa 7. Validación hold-out mediante split en un problema de clasificación.

Tabla de datos: 150 instancias y 4 atributos

Valores de la clase: {0, 1, 2}

Training: 105 instancias y 4 atributos

Test: 45 instancias y 4 atributos

Exactitud: 0.3111111111111111

Salida 7. Salida del Programa 7.

En el Programa 7, se importa el conjunto de datos “iris” mediante la función `load_iris()` del paquete `datasets` de scikit-learn. Esta función devuelve un diccionario con varios campos, entre ellos: `data` y `target`, que contienen los atributos (`data`) y la clase (`target`) de las instancias de la tabla de datos.

A continuación, se muestran las características principales del conjunto de datos cargado. Mediante la propiedad `shape` (de tipo tupla) de la matriz `iris.data` de tipo `numpy.ndarray` es posible acceder al número de filas (posición 0 de la tupla `shape`) y al número de columnas (posición 1). Ambos valores se muestran mediante la función `print`.

Una vez tenemos el conjunto de datos, lo dividiremos en entrenamiento y test utilizando la validación hold-out mediante `split` que hemos aprendido. Este procedimiento puede llevarse a cabo a través de la función `train_test_split()` del paquete `model_selection` de scikit-learn.

Le indicamos a la función `train_test_split()` la matriz de atributo (`iris.data`), el vector de la clase (`iris.target`), el tamaño (en tanto por uno) que queremos que tenga el subconjunto de test (`test_size`) y la semilla aleatoria para la reordenación de muestras previa a la partición de datos (`random_state`). En este ejemplo se ha usado un 70 % y 30 % (`test_size=0.3`). Si se hubiese deseado omitir la reordenación aleatoria de las muestras, hubiese sido necesario indicarlo mediante el parámetro `shuffle=False` de la función.

La partición de datos mediante la función `train_test_split()` resulta en cuatro elementos: la matriz de atributos del subconjunto de entrenamiento (`X_train`), la matriz de atributos del subconjunto de test (`X_test`), el vector de la clase del subconjunto de entrenamiento (`y_train`) y el vector de la clase del subconjunto de test (`y_test`).

A continuación, se muestran las características de los subconjuntos de entrenamiento y test. Tal como se puede apreciar en la Salida 6, el 70 % de los datos para entrenamiento se traduce en 105 instancias, mientras que el 30 % supone 45 instancias para test.

Una vez disponemos de los conjuntos de entrenamiento y test, procedemos a entrenar el modelo mediante un algoritmo de clasificación. Recuérdese que el conjunto de datos “iris” contiene una clase categórica, con lo que el algoritmo de aprendizaje supervisado debe ser apto para clasificación.

Dado que aún no hemos estudiado ningún algoritmo de inferencia basado en aprendizaje supervisado (los estudiaremos en los capítulos 3 y 4), hemos utilizado en el ejemplo el algoritmo `DummyClassifier`, provisto en el paquete `dummy` de scikit-learn.

El algoritmo **DummyClassifier** es muy sencillo. Tan solo infiere como clase de un ejemplo de test aquel valor aleatorio, entre los valores de la clase del conjunto de entrenamiento, cuya probabilidad de ocurrencia es proporcional al número de ejemplos de cada clase.

Por ejemplo, en el conjunto “iris” hay 3 valores de la clase categórica (es decir, 3 clases) y 50 ejemplos de cada clase (en total son 150). Por tanto, en este caso, el algoritmo **DummyClassifier** predecirá la clase de un ejemplo de test con las siguientes probabilidades: (0,333; 0,333; 0,333). En este caso particular, la distribución es uniforme, pero, en general, depende de la distribución de clases y puede haber ciertas clases más probables que otras, debido a que hay más muestras de una clase que de otra en el conjunto de entrenamiento.

Para probar un algoritmo de aprendizaje supervisado sobre un conjunto de datos, debemos crear, en primer lugar, un objeto de la clase que implemente el algoritmo deseado. En el caso del ejemplo que nos ocupa, es la clase **DummyClassifier**. A continuación, entrenamos un modelo mediante el método **fit()**, al cual indicamos la matriz de atributos (**X_train**) y el vector de clases (**y_train**) de entrenamiento.

Por último, una vez entrenado el modelo, utilizamos el método **score()** para predecir los ejemplos de test y evaluar la bondad del algoritmo. Para ello, le indicamos al método la matriz de atributos del conjunto de test (**X_test**) para realizar las predicciones y el vector de clases del conjunto de test (**y_test**) para evaluar las predicciones.

El valor devuelto por defecto del método **score()** es la métrica de exactitud, que definimos al principio de este capítulo. Tal como apreciamos en la Salida 6, la exactitud conseguida (en tanto por uno) es aproximadamente del 0,333. Si se repite la ejecución del Programa 7, los resultados podrán variar, pues se realiza una reordenación aleatoria de las muestras durante el proceso de validación hold-out.

Nótese que la exactitud teórica es del 0,333, debido a que las predicciones de **DummyClassifier** son aleatorias con una probabilidad que se distribuye uniformemente en las tres clases del conjunto de datos.

2.2. Validación cruzada

La validación hold-out es sencilla de realizar, económica (en términos de tiempo computacional) y ampliamente utilizada. No obstante, presenta un grave inconveniente y es la escasa representatividad de los resultados que se obtienen. Esto es, los algoritmos de aprendizaje solo se evalúan utilizando un único conjunto de test. Dado un mismo conjunto de datos, la elección del conjunto de test puede (y, de hecho, suele) hacer variar significativamente las métricas de bondad.



Extraer un único subconjunto de test del conjunto de datos del problema y probar los algoritmos con él conduce habitualmente a resultados sesgados a las particularidades del conjunto de test elegido. Necesitamos, por tanto, esquemas de validación que proporcionen resultados de bondad más generales y que nos permitan escoger los mejores algoritmos con mayor confianza.

La técnica de validación más utilizada que posee las características de generalidad y representatividad de sus resultados es la **validación cruzada** (CV, de *cross validation* en inglés).

La validación cruzada consiste en partir, en primer lugar, el conjunto de datos original en K subconjuntos (también llamados *bolsas*, *folds* en inglés) de igual tamaño. Por ejemplo, si el conjunto de datos posee 150 ejemplos (como en el caso de “iris”) y $K = 10$, entonces las 10 bolsas tendrían 15 ejemplos cada una.

Una vez partido el conjunto de datos en K bolsas, se realizan K validaciones de tipo hold-out empleando, en cada validación, como test una bolsa distinta y como entrenamiento el resto de ejemplos de las demás bolsas. En la Tabla 3 se ilustra gráficamente el proceso de validación cruzada con $K = 5$ bolsas.

Tabla 3

Validación cruzada con $K = 5$ bolsas

		Bolsas de instancias				
		Bolsa 1	Bolsa 2	Bolsa 3	Bolsa 4	Bolsa 5
Iteraciones	1	Test	Train	Train	Train	Train
	2	Train	Test	Train	Train	Train
	3	Train	Train	Test	Train	Train
	4	Train	Train	Train	Test	Train
	5	Train	Train	Train	Train	Test

Tal como se puede apreciar en la Tabla 3, se realizan 5 iteraciones de validación cruzada. En cada iteración, la bolsa utilizada como test está marcada en rojo, mientras que el resto de bolsas (las marcadas en azul) se utilizan como entrenamiento (*train*).

A continuación, mostramos en el Programa 8 un ejemplo en el que se ilustran los ejemplos escogidos en cada iteración de una validación cruzada. Nótese que en este programa no se hacen predicciones ni se evalúan, sino que tan solo se muestra el índice (comenzando en cero) de los ejemplos de entrenamiento y test de cada iteración.

```
</>
from sklearn.model_selection import KFold

# Carga de datos.
X = ["a", "b", "c", "d", "e", "f", "g", "h", "i", "j"]

# Validación cruzada.
kf = KFold(n_splits = 3)
bolsas = kf.split(X)
```

```
# Mostrar ejemplos de cada iteración.
k = 1
for train, test in bolsas:
    print("Iteracion", k, ":")
    print(" - Entrenamiento: %s" % (train))
    print(" - Test: %s" % (test))
    k = k + 1
```

Programa 8. Visualización de ejemplos escogidos en cada iteración de una validación cruzada.

Iteracion 1 :

- Entrenamiento: [4 5 6 7 8 9]
- Test: [0 1 2 3]

Iteracion 2 :

- Entrenamiento: [0 1 2 3 7 8 9]
- Test: [4 5 6]

Iteracion 3 :

- Entrenamiento: [0 1 2 3 4 5 6]
- Test: [7 8 9]

Salida 8. Salida del Programa 8.

En el Programa 8 creamos un conjunto de datos de 10 instancias con un único atributo de tipo categórico (cadena de caracteres, en particular) y sin clase, que almacenamos en la variable X. A continuación, mediante la clase `KFold` y el método `split()` troceamos la tabla de datos en $K = 3$ pares de subconjuntos de entrenamiento/test, tal como se muestra en la Salida 7. Nótese que en la primera iteración se tienen 4 ejemplos en el test, a diferencia del resto de iteraciones, que tienen 3. Esto es debido a que el reparto de 10 ejemplos entre $K = 3$ no es exacto (división inexacta).

En el Programa 9 mostramos un ejemplo de aplicación de la validación cruzada en el problema de clasificación del conjunto “iris” mediante el clasificador `DummyClassifier` introducido anteriormente.

```
</>
from sklearn import datasets
from sklearn.dummy import DummyClassifier
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold

# Carga de datos.
iris = datasets.load_iris()
```

```

# Mostrar características de la tabla de datos.
print("Tabla de datos: %d instancias y %d atributos" % (iris.data.shape[0],
iris.data.shape[1]))
print("Valores de la clase:", set(iris.target))

# Construcción del objeto que contiene el algoritmo de aprendizaje.
clf = DummyClassifier()

# Validación, entrenamiento y evaluación del algoritmo de aprendizaje.
evaluacion = cross_val_score(clf, iris.data, iris.target, cv = KFold(n_
splits=10))
print("Exactitud de cada bolsa:", evaluacion)
print("Exactitud (media +/- desv.): %0.3f (+/- %0.3f)" % (evaluacion.mean(),
evaluacion.std()))

```

Programa 9. Validación cruzada en un problema de clasificación.

```

Tabla de datos: 150 instancias y 4 atributos

Valores de la clase: {0, 1, 2}

Exactitud de cada bolsa: [0.2 0.26666667 0.33333333 0.33333333 0.53333333
0.13333333 0.53333333 0.33333333 0.33333333 0.06666667]

Exactitud (media +/- desv.): 0.307 (+/- 0.144)

```

Salida 9. Salida del Programa 9.

Para realizar la validación cruzada de un algoritmo de aprendizaje supervisado, hemos usado en el Programa 9 la función `cross_val_score()` del paquete `model_selection` de scikit-learn. Para ello, le indicamos a la función el objeto que implementa el algoritmo de predicción (en este caso `DummyClassifier`), la matriz de atributos de entrenamiento (`iris.data`), el vector de clases de entrenamiento (`iris.target`) y el objeto que implementa el algoritmo de validación (`KFold`, el mismo que hemos usado anteriormente en el Programa 8). En este caso, hemos usado $K = 10$ bolsas para la validación cruzada.

Finalmente, el programa escribe por pantalla los valores de exactitud alcanzados por el algoritmo en cada bolsa de validación, así como el promedio y la desviación típica de dichos valores. Nótese que, a diferencia de la validación hold-out, donde solo había un valor de exactitud, ahora obtenemos K valores de exactitud, que, habitualmente, mostramos de forma resumida mediante el promedio y desviación estándar. Nótese, asimismo, que los valores de exactitud rondan al 0,333 teórico, al igual que ocurría con la validación hold-out.

Un caso particular de validación cruzada se produce cuando el número de bolsas K coincide con el número de instancias del conjunto de datos (n), en cuyo se trata de una **validación leave-one-out** (dejar uno fuera). La idea es predecir cada ejemplo del conjunto de datos usando como entrenamiento el resto. Como veremos a continuación, la validación leave-one-out posee una interesante propiedad que la validación cruzada con $K < n$ no posee.



Los resultados de bondad obtenidos por una validación cruzada con $K < n$ pueden depender, en gran medida, de la elección aleatoria de las bolsas, especialmente en conjuntos de datos con alta diversidad en los valores de sus atributos y/o clase. De hecho, se recomienda repetir la propia validación cruzada un número determinado de veces, con el objetivo de que los resultados sean estadísticamente significativos.

Mediante la validación leave-one-out (validación cruzada con $K = n$) el problema anterior desaparece, pues la elección de bolsas no es aleatoria, sino que existe una única posible. No obstante, la validación leave-one-out es más costosa en tiempo de computación. Nótese que requiere n entrenamientos de modelos y n predicciones, mientras que la validación cruzada con $K < n$ tan solo requiere K entrenamientos y n predicciones. Para conjuntos de datos extensos (con n igual a varios millones de instancias), la validación leave-one-out suele resultar impráctico y se recurre a una validación cruzada con $K = 10$ (o valores de K similares).



Enlace de interés

Otro tipo de validación muy interesante que también se utiliza en proyectos de clasificación es la **validación estratificada**. En esta validación, los conjuntos de test que se generan guardan aproximadamente la misma distribución de clases que los conjuntos de entrenamiento. De este modo, los algoritmos de aprendizaje suelen arrojar mejores resultados, pues aprenden modelos más adecuados a la naturaleza del test, lo que evita el conocido problema del desbalanceo de clases (distribución desigual de clases). Puede consultar más información sobre la validación estratificada y cómo utilizarla en Python con la librería scikit-learn en el siguiente enlace:

https://scikit-learn.org/stable/modules/cross_validation.html#cross-validation-iterators-with-stratification-based-on-class-labels

2.3. Ajuste de parámetros y validación anidada

Como veremos en los capítulos 3 y 4, los algoritmos de aprendizaje supervisado pueden poseer uno o varios **parámetros de configuración**, que regulan su funcionamiento y permiten adaptar el algoritmo a la naturaleza de los datos de entrenamiento. Los valores que se utilicen para los parámetros de los algoritmos afectan a la bondad de los resultados obtenidos, con lo cual estos deben ser cuidadosamente seleccionados.

La elección óptima de los parámetros de los algoritmos, aquella que maximiza la bondad de los resultados predictivos, se denomina **ajuste de parámetros** (*hyperparameter tuning* en inglés). Para llevar a cabo este proceso, generalmente se hace una búsqueda de los mejores valores de parámetros, de manera que se maximice una o varias métricas de evaluación (por ejemplo, la exactitud en clasificación o el MAE en regresión). Esta búsqueda puede hacerse de forma exhaustiva o mediante alguna heurística o metaheurística. En la asignatura *Algoritmos de optimización* se aborda en detalle en qué consisten este tipo de búsquedas, las cuales pueden aplicarse al problema del ajuste de parámetros.



El ajuste de los parámetros no debe hacerse maximizando la bondad de los resultados obtenidos en el conjunto de test, sino en el conjunto de entrenamiento. El conjunto de test debe mantenerse aislado durante toda la fase de entrenamiento y ajuste del modelo. Respetar esta norma es fundamental para poder evaluar correctamente la bondad de los modelos, perseguir que estos sean generales y no incurrir en el problema del sobreajuste, anteriormente explicado.

No debemos usar el conjunto de test para ajustar los parámetros de los algoritmos. Sin embargo, para ajustar dichos parámetros, es preciso evaluar la bondad de los resultados obtenidos tras entrenar un modelo con un set de parámetros. Por esta razón, habitualmente se crea para ello el conjunto de test de validación (*validation test* en inglés), el cual se extrae como parte del conjunto de entrenamiento, tal como ilustra la Figura 4.

En la Figura 4 se aprecia cómo se procede al ajuste y validación del modelo a partir de un conjunto de datos utilizando internamente tests de validación, lo cual habilita la posibilidad de llevar a cabo el ajuste de los parámetros de los algoritmos de aprendizaje.

Observe en la Figura 4 que cada modelo se entrena con el conjunto de entrenamiento y se prueba con el test de validación, de manera que las predicciones producidas son evaluadas, de cara a reajustar el modelo con nuevos parámetros y mejorar su bondad.

Finalmente, tras el proceso de ajuste y validación del modelo, este se prueba con el conjunto independiente de test para realizar la evaluación final.



Figura 4. Esquema de validación anidada con tests de validación. Adaptado de “Evaluation” por P. L. Lanzi, 2005, en *Machine Learning and Data Mining* (diapositivas en línea). Recuperado de <https://www.slideshare.net/pierluca.lanzi/machine-learning-and-data-mining-14-evaluation-and-credibility>

A pesar de que el ajuste de parámetros se haga en un conjunto de validación extraído a partir del conjunto de entrenamiento, este procedimiento suele causar también sobreajuste del modelo a las peculiaridades del conjunto de validación escogido. De esta forma, si la distribución y las relaciones entre los datos del conjunto de test independiente no se parecen a las del test de validación, los resultados del modelo en este último conjunto suelen ser peores que los obtenidos en el test de validación.

Debido al riesgo de sobreajuste del modelo al test de validación, se suelen utilizar no uno, sino varios tests de validación utilizando una validación cruzada sobre el conjunto de entrenamiento. Este procedimiento se denomina **validación anidada** (*nesting validation* o *double validation* en inglés), de forma que se tienen dos validaciones:

- Validación externa. La validación que hemos estudiado, en la cual se parte el conjunto de datos original en uno o varios pares de conjuntos de entrenamiento y test.
- Validación interna. Es una partición del conjunto de entrenamiento, a su vez, en subconjuntos de entrenamiento y test. Útil para el ajuste de parámetros de modelos de aprendizaje.

Utilizando la validación anidada, el ajuste de parámetros puede consistir en encontrar los valores óptimos para los mismos de forma que se maximice el promedio, obtenido para todos los tests de validación, de alguna de bondad. Una vez encontrado el set óptimo de parámetros, el modelo se entrena con el conjunto completo de entrenamiento usando dicho set y se evalúa su bondad sobre el test independiente.

2.4. Evaluación en regresión

En este apartado estudiaremos algunas de las métricas de evaluación más utilizadas en problemas de regresión. Al comienzo de este capítulo definimos la métrica MAE, que se basa en las diferencias en valor absoluto entre predicciones y valores reales. A continuación, definimos la métrica RMSE, que se basa en las diferencias al cuadrado entre predicciones y valores reales, tal como se muestra en la siguiente fórmula.

$$\text{RMSE} = \sqrt{\frac{1}{n} \cdot \sum_{i=1}^n (y_i - \hat{y}_i)^2}$$

Tal como se aprecia en la definición de RMSE, las diferencias entre predicciones y valores reales se encuentran elevadas al cuadrado, promediadas, y finalmente calculada su raíz cuadrada. Nótese que, a diferencia de la métrica MAE, en RMSE los mayores errores ($y_i - \hat{y}_i$) pesan mucho más en el cálculo de la métrica, pues están elevados al cuadrado.

Ambas métricas, MAE y RMSE, pertenecen al conjunto de **métricas de evaluación absolutas**, pues las desviaciones entre los valores reales y predichos tienen la misma unidad de magnitud que la variable de clase. Por ejemplo, si se tratara de un problema en el que hay que predecir el precio de una vivienda en dólares, aplicamos un algoritmo de aprendizaje de regresión y evaluamos los resultados; los valores de las métricas MAE y RMSE estarían también en dólares.

En determinadas ocasiones, necesitamos conocer la magnitud de los errores cometidos de forma relativa al valor real que se predice, esto es, disponer de métricas que nos revelen la proporción (porcentaje o tanto por uno) del valor desviado con respecto al valor real. Se trata de las **métricas de evaluación relativas**. Entre las más utilizadas, se encuentra la métrica MAPE, que definimos a continuación.

$$\text{MAPE} = \frac{100}{n} \cdot \sum_{i=1}^n \left| \frac{y_i - \hat{y}_i}{y_i} \right|$$

En el Programa 10 mostramos un ejemplo de aplicación de las métricas de evaluación de regresión que hemos estudiado. En este ejemplo se ha usado el conjunto de datos “Boston” de regresión, en el que el objetivo es predecir precios de viviendas en la ciudad de Boston. En la Tabla 4 se muestran las características del conjunto de datos.

Se ha utilizado un algoritmo de regresión muy simple llamado `DummyRegressor` (incluido en scikit-learn), que produce siempre como predicción la media aritmética de las clases del conjunto de entrenamiento. Por ejemplo, si el conjunto de entrenamiento tuviese tres instancias con clases 3, 4 y 5, este regresor siempre predeciría el valor 4.

Tabla 4*Características del conjunto de datos “Boston”*

Número de instancias	506
Número de atributos	13
Descripción de atributos	Por motivos de extensión no se ha incluido aquí, pero puede consultarse en el siguiente enlace: https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_boston.html
Tipo de datos de la clase	Numérico
Descripción de la clase	Nombre: MEDV. Descripción: valor medio de las viviendas ocupadas por propietarios.
Valores ausentes	0



```

import numpy as np
from math import sqrt
from pprint import pprint
from sklearn import datasets
from sklearn.dummy import DummyRegressor
from sklearn.model_selection import cross_validate
from sklearn.model_selection import KFold
from sklearn.metrics import make_scorer
from sklearn.metrics import mean_squared_error

# Carga de datos.
datos = datasets.load_boston()

# Algoritmo de aprendizaje.
reg = DummyRegressor()

```

```

# Métricas de evaluación.
metricas = {
    'MAE': 'neg_mean_absolute_error',
    'RMSE': make_scorer(
        lambda y, y_pred:
            sqrt(mean_squared_error(y, y_pred)),
            greater_is_better=False),
    'MAPE': make_scorer(
        lambda y, y_pred:
            np.mean(np.abs((y - y_pred) / y)) * 100,
            greater_is_better=False)}

# Validación y evaluación del modelo.
evaluacion = cross_validate(reg, datos.data, datos.target,
                            cv = KFold(n_splits=10), scoring = metricas)

# Presentación de Los resultados de La evaluación.
pprint(evaluacion)

```

Programa 10. Métricas de evaluación en regresión.

```

{'fit_time': array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]),

'score_time': array([0.001, 0.    , 0.    , 0.001, 0.001,
                    0.    , 0.    , 0.    , 0.    , 0.    ]),

'test_MAE': array([-5.352, -3.53 , -4.947, -10.762,
                   -6.44, -10.048, -3.174, -11.293, -10.234, -4.638]),

'test_MAPE': array([-29.251, -12.906, -30.399, -30.346,
                     -18.562, -27.268, -14.909, -98.893, -90.94, -33.262]),

'test_RMSE': array([-6.329, -5.549, -5.611, -13.845,
                     -10.196, -13.139, -4.072, -13.456, -10.809, -5.994]),

'train_MAE': array([-6.841, -6.961, -6.954, -6.065, -6.58,
                     -6.144, -7.043, -6.27, -6.525, -6.959]),

'train_MAPE': array([-37.737, -38.425, -38.389, -34.709,
                     -36.861, -35.016, -38.865, -31.041, -33.275, -37.789]),

'train_RMSE': array([-9.458, -9.512, -9.52 , -8.571, -9.089,
                     -8.697, -9.585, -8.612, -9.051, -9.483])}

```

Salida 10. Salida del Programa 10.

En el Programa 10 se carga, en primer lugar, el conjunto de datos “Boston”. A continuación, se construye un objeto de tipo `DummyRegressor`, que implementa el sencillo algoritmo de aprendizaje explicado anteriormente. En la variable `métricas` se construye un diccionario con las métricas que deseamos evaluar. Si bien la métrica MAE ya viene definida como `neg_mean_absolute_error` en scikit-learn, las métricas RMSE y MAPE debemos implementarlas.

Para definir las métricas RMSE y MAPE, hacemos uso de la función `make_scoring()` de scikit-learn. Con `make_scoring()` podemos convertir cualquier función en una métrica de evaluación de algoritmos de predicción, siempre que dicha función reciba dos vectores (un vector (`y`) de valores reales y otro (`y_pred`) con las predicciones) y devuelva un valor numérico, que será interpretado como valor de bondad.

Hay que tener en cuenta que scikit-learn asume por defecto que toda métrica de evaluación genera un número de manera que valores más altos son mejores valores.

Este criterio corresponde adecuadamente con métricas como la exactitud (si la exactitud es mayor, el resultado es mejor). Sin embargo, no corresponde con métricas como MAE, RMSE ni MAPE, pues miden errores y, cuanto mayores sean sus valores, el resultado será peor. Por esta razón, en estas métricas, establecemos el parámetro `greater_is_better=False`, diseñado para tal propósito.

En el Programa 10 se ha realizado una validación cruzada con 10 bolsas mediante la función `cross_valide-`
`date()`. Esta función, a diferencia de `cross_val_score()`, que hemos usado en ejemplos anteriores, nos permite indicar no una, sino varias métricas de evaluación. Para ello, usamos el argumento `scoring` de dicha función.

Finalmente, se ha usado la función `pprint()` del paquete `pprint`, que permite escribir por pantalla variables de tipo diccionario de un modo más legible que la función `print()`. Tal como se aprecia en el resultado del programa (Salida 9), el diccionario incluido en la variable evaluación contiene 8 entradas: `fit_time`, `score_time`, `test_MAE`, `test_MAPE`, `test_RMSE`, `train_MAE`, `train_MAPE` y `train_RMSE`.

Las dos primeras entradas del diccionario de evaluación almacenan los tiempos de ejecución empleado en el entrenamiento (`fit_time`) y evaluación (`score_time`) del modelo.

Las entradas con prefijo `train_` hacen referencia a métricas que se evalúan sobre el conjunto de entrenamiento. Esto es, se entrena el modelo con el conjunto de entrenamiento y se prueba el modelo prediciendo los mismos ejemplos de entrenamiento.

Como ya estudiamos al comienzo de este capítulo, estos valores obtenidos del entrenamiento son generalmente mejores que los obtenidos probando el modelo sobre el conjunto de test (entradas con el prefijo `test_`). No obstante, dada la naturaleza del regresor que hemos probado (`DummyRegressor`), y su escasa o nula capacidad predictiva, los resultados obtenidos son muy deficientes, tanto evaluados en entrenamiento como en test. En el Capítulo 3 estudiaremos algoritmos más eficaces.

2.5. Evaluación en clasificación

A diferencia de la regresión, una predicción en clasificación generalmente se evalúa teniendo en cuenta si se acierta o no se acierta la clase real, no con un grado de acierto.

Por ejemplo, supongamos un problema de regresión en el que se necesita dar respuesta a la pregunta “¿Cuántos litros por metro cuadrado lloverá la próxima semana?”. La evaluación de la predicción consistiría en medir el grado de acierto como la diferencia numérica entre el valor real y el predicho (por ejemplo, 12 l/m³ de error).

Por el contrario, supongamos un problema de clasificación en el que se necesita responder a la pregunta “¿Lloverá o no lloverá la próxima semana?”. En este caso, la evaluación de la predicción consistirá en determinar si el modelo ha acertado o ha fallado, sin tener en cuenta ningún grado numérico del error en una única predicción.

Para evaluar las predicciones en clasificación, la métrica de evaluación principal es la **matriz de confusión**. La matriz de confusión es una métrica que, a diferencia de las métricas estudiadas hasta ahora, es de tipo matriz y contiene el recuento de predicciones realizadas organizadas por su correspondencia con la clase real.

La matriz de confusión es una matriz cuadrada cuyo orden es el número de clases del conjunto de datos. Por ejemplo, en la Tabla 5 se muestra una matriz de confusión para conjuntos de datos con una clase de 4 valores (también se dice que tiene 4 clases).

Tabla 5
Matriz de confusión con 4 clases

		Clase predicha			
		C ₁	C ₂	C ₃	C ₄
Clase real	C ₁	P _{1,1}	P _{1,2}	P _{1,3}	P _{1,4}
	C ₂	P _{2,1}	P _{2,2}	P _{2,3}	P _{2,4}
	C ₃	P _{3,1}	P _{3,2}	P _{3,3}	P _{3,4}
	C ₄	P _{4,1}	P _{4,2}	P _{4,3}	P _{4,4}

El valor $P_{ij} \in \mathbb{N}$ con $i, j = 1, \dots, m$ (m es el número de clases) de la matriz confusión es el número de predicciones en las que el valor real de la clase es C_i y la predicción que arroja el modelo es C_j . Según esta definición de la matriz confusión, observamos que el número total de aciertos es $\sum_{i=1}^m P_{i,i} = P_{1,1} + P_{2,2} + P_{3,3} + P_{4,4}$ (suma de la diagonal principal de la matriz). Los fallos del modelo se encuentran repartidos en el resto de elementos de la matriz. Dado que la matriz de confusión contiene el recuento total, n , de predicciones realizadas, organizadas en m^2 casos posibles, donde el número de fallos del modelo es, por tanto, $n - \sum_{i=1}^m P_{i,i}$.

Según estas definiciones, la métrica de exactitud que hemos estudiado también puede expresarse del siguiente modo (donde m es el número de clases, n el número de predicciones y P_{ii} los elementos de la diagonal principal de la matriz de confusión):

$$\text{Exactitud} = \frac{100}{n} \cdot \sum_{i=1}^m P_{i,i}$$

Es especialmente relevante el caso particular binario, es decir, cuando en el conjunto de datos se tienen 2 clases ($m = 2$). Por ejemplo, en el problema “¿Lloverá o no lloverá la próxima semana?” o “¿Es un cliente propenso a irse a la competencia?”, o cualquier otro en el que la respuesta sea “sí” o “no”.

En estos casos, a las clases suelen asociarse los conceptos de positivo y negativo. De esta forma, tenemos $m^2 = 4$ casos posibles:

- El modelo dice “sí” y la realidad es “sí” (acierto positivo; true positive o TP en inglés).
- El modelo dice “no” y la realidad es “no” (acierto negativo; true negative o TN en inglés).
- El modelo dice “sí” y la realidad es “no” (falso positivo; false positive o FP en inglés).
- El modelo dice “no” y la realidad es “sí” (falso negativo; false negative o FN en inglés).

Por consiguiente, en el caso binario la matriz de confusión queda definida tal como se muestra en la Tabla 6, y la exactitud según la siguiente ecuación:

$$\text{Exactitud} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}$$

Tabla 6
Matriz de confusión binaria (2 clases)

		Clase predicha	
		Positiva	Negativa
Clase real	Positiva	TP	FN
	Negativa	FP	TN

Tanto la matriz de confusión binaria como las métricas de clasificación binaria TP, TN, FP y FN no se utilizan únicamente en problemas de clasificación binaria, sino también en problemas de clasificación multiclas ($m > 2$). En concreto, en cualquier problema de clasificación se puede evaluar la bondad de un modelo con respecto a cada clase. De este modo, se distinguen m evaluaciones de tipo binario. Esto es, m matrices de confusión, m tuplas $\langle \text{TP}, \text{TN}, \text{FP}, \text{FN} \rangle$, m valores de exactitud, etc., todas calculadas teniendo en cuenta los aciertos y los fallos para cada clase real de forma separada. La siguiente ecuación muestra la exactitud solo para la clase C_i .

$$\text{Exactitud}(C_i) = 100 \cdot \frac{P_{i,i}}{\sum_{j=1}^m P_{i,j}}$$

Definimos dos nuevas métricas de evaluación para clasificación binaria, pero que pueden usarse también en problemas multiclas, tal como hemos comentado.

La primera de ellas es la precisión (*precision* en inglés), la cual se centra en la que definamos como clase positiva y mide la tasa de aciertos positivos con respecto al total de predicciones positivas realizadas. Su fórmula es la siguiente:

$$\text{Precision} = 100 \cdot \frac{TP}{TP + FP}$$

Tal como se desprende de su definición, la precisión mide el grado en el que podemos confiar en las predicciones positivas de un modelo. Un modelo con una precisión del 100 % significa que, siempre que da una predicción positiva, acierta. Por ejemplo, si el modelo resuelve la pregunta “¿Lloverá o no lloverá la próxima semana?” y su predicción es “sí”, entonces acierta seguro. Nótese que esto no implica que cuando dice “no” también acierte siempre: es decir, FN puede ser mayor que cero.

Un modelo con una precisión del 100 % ($FP = 0$) es muy deseable, pero que pasaría si solo rara vez diese predicciones positivas. Es decir, un modelo, cuando da una predicción positiva, acierta, pero puede apenas dar predicciones positivas (casi siempre da negativas). Un modelo como este puede no ser de utilidad, pues escasas veces se puede contar con él para tomar decisiones. Necesitamos una forma de medir la cantidad de casos positivos que el modelo es capaz de cubrir con sus predicciones positivas acertadas. La sensibilidad o cobertura (*recall* o *coverage* en inglés) es la métrica que expresa dicha cantidad. En la siguiente ecuación se muestra su definición:

$$\text{Sensibilidad} = \frac{TP}{TP + FN}$$

Un modelo con una sensibilidad del 100 % ($FN = 0$) cubre con predicciones positivas todos los casos reales positivos, es decir, siempre que se ha dado un caso positivo en la realidad, el modelo lo ha predicho correctamente. Esto parece muy deseable, pero, de forma análoga a como hemos visto con la precisión, no debemos dar por bueno siempre un valor de 100 % de sensibilidad. ¿Qué pasaría si el modelo diera predicciones positivas siempre? Tendría un 100 % de sensibilidad y acertaría en todos los casos que son positivos, pero fallaría en todos los negativos.



Debemos encontrar modelos con valores altos tanto en sensibilidad como en precisión. De esta forma, tendremos un modelo útil que suele acertar cuando da una predicción positiva, y suele darla siempre que se dan casos positivos reales.

Sin embargo, en la práctica, la precisión y la sensibilidad de los modelos suelen estar reñidas: cuando conseguimos mejorar la precisión de un modelo, su sensibilidad empeora, y viceversa. Necesitamos, por tanto, optimizar ambas métricas (sensibilidad y precisión) en conjunto, no solo una de ellas. Dado que muchos algoritmos admiten una única métrica para optimizar, definimos una nueva métrica, capaz de resumir en un solo número la sensibilidad y la precisión de los modelos. Se trata de la métrica F-measure o F1, que se define como la media armónica entre sensibilidad y precisión, tal como se muestra en la siguiente ecuación:

$$F1 = 100 \cdot \frac{2 \cdot \text{Sensibilidad} \cdot \text{Precision}}{\text{Sensibilidad} + \text{Precision}}$$

En el Programa 11 mostramos un ejemplo en el que se calculan las métricas de evaluación estudiadas tras aplicar el algoritmo simple de clasificación **DummyClassifier**, analizado previamente, sobre el conjunto de datos “iris” empleando una validación cruzada de 10 bolsas.



```
from sklearn import datasets
from sklearn.dummy import DummyClassifier
from sklearn.model_selection import cross_val_predict
from sklearn.model_selection import KFold
import sklearn.metrics as metrics

# Carga de datos.
datos = datasets.load_iris()

# Algoritmo de aprendizaje.
clf = DummyClassifier()

# Validación y obtención de las predicciones del modelo.
y_pred = cross_val_predict(clf, datos.data, datos.target,
                           cv = KFold(n_splits=10))

# Presentación de los resultados de la evaluación.
print("Exactitud: %.3f\n" %
      (metrics.accuracy_score(datos.target, y_pred)))
print("Precisión: %.3f\n" %
      (metrics.precision_score(datos.target, y_pred, average="micro")))
print("Sensibilidad: %.3f\n" %
      (metrics.recall_score(datos.target, y_pred,
                            average="micro")))
print("F1: %.3f\n" %
      (metrics.f1_score(datos.target, y_pred, average="micro")))
print("Matriz de confusión:\n",
      metrics.confusion_matrix(datos.target, y_pred))
print("Tabla de métricas:\n",
      metrics.classification_report(datos.target, y_pred))
```

Programa 11. Métricas de evaluación en clasificación.

```
Exactitud: 0.287
Precisión: 0.287
Sensibilidad: 0.287
F1: 0.287
Matriz de confusión:
[[17 16 17]
 [15 16 19]
 [18 22 10]]
```

Tabla de métricas:

	precision	recall	f1-score	support
0	0.34	0.34	0.34	50
1	0.30	0.32	0.31	50
2	0.22	0.20	0.21	50
avg / total	0.28	0.29	0.29	150

Salida 11. Salida del Programa 11.

La función `cross_val_predict()` de scikit-learn permite realizar la validación cruzada y devolver un único vector que reúne las predicciones realizadas en todas las bolsas de la validación. Este vector, llamado `y_pred`, y el vector de las clases reales de los ejemplos, llamado `datos.target`, son usados para calcular las diferentes métricas.



Enlace de interés

En la siguiente página web del sitio oficial de la documentación de scikit-learn puede encontrar un estudio más detallado de las métricas de precisión y sensibilidad con ejemplos de aplicación en Python.

http://scikit-learn.org/stable/auto_examples/model_selection/plot_precision_recall.html

Para ampliar la información relacionada con los contenidos del Capítulo 2 (validación, ajuste de parámetros y evaluación), recomendamos consultar el capítulo 5 de James et al. (2013), el apartado 10.9 de Aggarwal (2015), el capítulo 6 de Mueller y Guido (2016), el capítulo 5 de Sarkar (2018) y los apartados 5.3 y 6.4 de Watt et al. (2016).



Capítulo 3

Regresión

3.1. Regresión lineal múltiple

La primera aproximación que estudiaremos para resolver problemas de regresión en aprendizaje supervisado es la regresión lineal. Dentro del paradigma de la regresión lineal se incluyen numerosos algoritmos de aprendizaje supervisado. Todos los algoritmos de regresión lineal se basan en encontrar relaciones lineales entre los atributos y la clase. Dado que la linealidad es una característica inherente a muchísimos problemas en la naturaleza, las técnicas de regresión lineal han sido utilizadas con éxito en numerosos problemas.

Se distinguen dos tipos fundamentales de regresión: simple y múltiple. La regresión simple únicamente utiliza un atributo de entrada, mientras que la regresión múltiple admite varios atributos. En este capítulo abordaremos la regresión lineal múltiple, como caso general.

El algoritmo clásico de regresión lineal, y el más utilizado, se denomina *ordinary least squares* (OLS). Es apto tanto para regresión simple como múltiple y es en el que nos vamos a centrar en este capítulo.

Como sabemos de capítulos anteriores, los modelos de conocimiento en aprendizaje supervisado pueden ser entendidos como funciones, no necesariamente matemáticas, que reciben una entrada (un vector con los valores de los atributos de un ejemplo de test) y devuelven una salida (un valor estimado para la clase del ejemplo de test).

El algoritmo OLS es capaz de generar un modelo que consta de una función matemática muy sencilla: una combinación lineal de los atributos. En concreto, el modelo que genera OLS puede expresarse matemáticamente del siguiente modo:

$$\hat{y} = w_0 + w_1 \cdot x_1 + \dots + w_p \cdot x_p$$

En la ecuación anterior, \hat{y} es el valor predicho por el algoritmo, x_1, \dots, x_p son los valores de los atributos, $w_1, \dots, w_p \in \mathbb{R}$ son los coeficientes del modelo y w_0 es el término independiente de la combinación lineal (*intercept* en inglés).

A continuación, vamos a describir cómo funciona el proceso de entrenamiento del modelo generado por el algoritmo OLS, para, posteriormente, explicar cómo se llevan a cabo las predicciones usando el modelo, una vez que ha sido entrenado.

El entrenamiento mediante OLS, al igual que con cualquier otro algoritmo, parte del conjunto de datos de entrenamiento, compuesto por X (matriz de atributos) e y (vector de clases). OLS comienza, entonces, un proceso iterativo de búsqueda de los valores óptimos para los coeficientes y el término independiente, w_0, \dots, w_p , del modelo.

Todo proceso de optimización iterativo requiere de una función objetivo (también función de bondad o de coste) para evaluar la bondad de una posible solución, y de un esquema de búsqueda que genere nuevos valores para evaluar en la siguiente iteración. En el caso de OLS, la función objetivo es la suma de errores al cuadrado (MSE de *mean squared error* en inglés) y se trata de encontrar los valores de w_0, \dots, w_p que minimizan su valor. En la siguiente ecuación se define la función objetivo que OLS trata de minimizar.

$$f(w_0, \dots, w_p) = \frac{1}{2n} \cdot \sum_{i=1}^n \left(\left(w_0 + \sum_{j=1}^p w_j \cdot x_{ij} \right) - y_i \right)^2$$

En la ecuación de la función objetivo f de OLS notamos, en primer lugar, que su valor depende únicamente de los coeficientes del modelo w_0, \dots, w_p , dado que tanto los atributos de entrenamiento x_{ij} (elementos de la matriz X) como la clase y_i (elementos del vector y) son inmutables.

La ecuación básicamente consiste en la suma de las diferencias al cuadrado entre la estimación del modelo con los coeficientes w_0, \dots, w_p y la clase real y_i . El factor $\frac{1}{2n}$ actúa como corrección al volumen de datos de entrenamiento, con el fin de normalizar la función objetivo.

Tal como se puede deducir, cuanto menor sea el valor de la función f , mejor será el ajuste del modelo a los datos de entrenamiento, y se espera que tendrá mayor capacidad predictiva en ejemplos de test.

En la Figura 5 se muestra el escenario de cálculo de la función objetivo de la regresión lineal para un conjunto de datos con un único atributo (x) y una clase (y) (regresión lineal simple).

En la figura, los puntos de color negro son los ejemplos de entrenamiento, mientras que la línea recta de color azul representa el modelo de regresión lineal con unos valores concretos para sus coeficientes. La función objetivo mide la suma de las longitudes de los segmentos de color rojo (errores) elevadas al cuadrado.

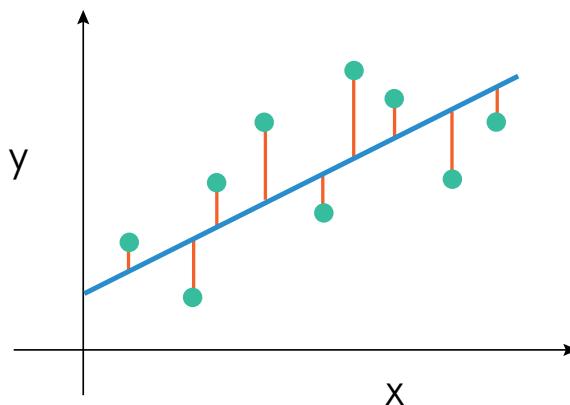


Figura 5. Escenario de cálculo de la función objetivo en regresión lineal.

En la Figura 6 se ilustra cómo pueden interpretarse los coeficientes del modelo en un conjunto de datos con un único atributo. Tal como se puede apreciar, el coeficiente w_1 es la pendiente de la recta de regresión, mientras que el término independiente w_0 del modelo es la ordenada en el origen ($x = 0$) de dicha recta.

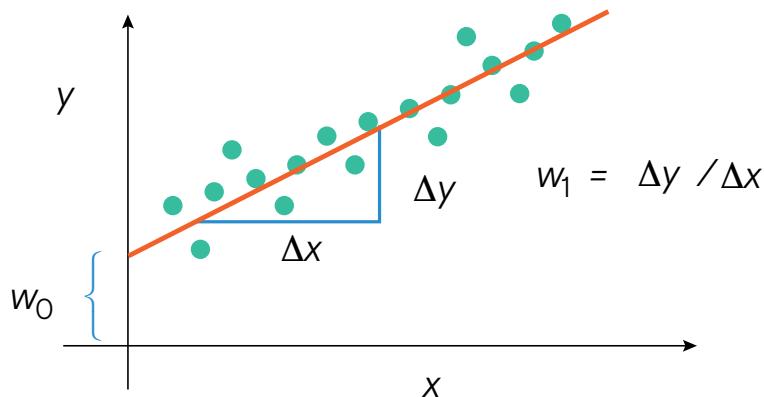


Figura 6. Interpretación geométrica de los coeficientes del modelo de regresión lineal.

Una vez definida la función objetivo de OLS, el algoritmo de búsqueda se encarga de generar valores para los coeficientes, de forma que en cada iteración se obtenga un valor de la función f menor. El proceso se repite hasta que se alcance una situación donde no se encuentra ningún cambio en la función f , o bien el cambio es muy pequeño. El algoritmo de búsqueda más habitual usado en OLS es el descenso de gradiente (*gradient descent optimization* en inglés).

Dado que no es el objetivo de esta asignatura profundizar en los algoritmos de búsqueda, tan solo indicaremos que el algoritmo del descenso de gradiente encuentra la dirección y magnitud del cambio de cada coeficiente que señala hacia la región de mayor descenso en el valor de la función objetivo. Para ello, se basa en un parámetro externo de tipo numérico: la tasa de aprendizaje (*learning rate* en inglés).



Enlace de interés

Un recurso muy interesante para aprender no solo aprendizaje supervisado en general, sino el funcionamiento del algoritmo del descenso de gradiente dentro de la regresión lineal y la tasa de aprendizaje es el Curso Intensivo de Machine Learning ofrecido por Google. Puede consultarse en el siguiente enlace el apartado del curso (denominado “Reducción de la pérdida”) que describe el funcionamiento de la regresión lineal y del descenso de gradiente.

<https://developers.google.com/machine-learning/crash-course/reducing-loss/gradient-descent>

Una vez definido el proceso de aprendizaje del algoritmo OLS, veamos cómo se aplica el modelo aprendido para producir predicciones en las instancias de test. Este proceso de predicción es muy sencillo. Basta sustituir los valores x_i de los atributos de la instancia de test en la ecuación del modelo y evaluarla para producir el valor de \hat{y} , el cual será la predicción final del modelo (los valores de w_0, \dots, w_p ya están definidos).

Definimos a continuación una nueva métrica de evaluación para regresión que debería usarse solo para evaluar modelos de lineales, como OLS. Por esta razón, la definimos en este apartado, en lugar de en el Capítulo 2.

Esta métrica es el coeficiente de determinación, o R^2 , y recoge la cantidad de variabilidad de la clase (con respecto a su media aritmética) que el modelo es capaz de predecir con respecto al total de variabilidad de la clase. Su definición se muestra en la siguiente ecuación.

$$R^2 = 1 - \frac{\sum_{i=1}^n (y_i - \hat{y}_i)^2}{\sum_{i=1}^n (y_i - \bar{y})^2}$$

En la anterior ecuación, \bar{y} es la media aritmética de los valores de la clase. Un valor de $R^2 = 1$ implica una predicción perfecta, pues se obtendría que $y_i = \hat{y}_i, \forall i$. Un valor de $R^2 = 0$ implica que el modelo tiene la misma eficacia que la del **DummyRegressor**, el cual, recordemos, producía como predicción siempre la media aritmética de la clase. Si el valor de R^2 fuera menor que 0, el modelo tendría peor eficacia que **DummyRegressor**.

Veamos, a continuación, en el Programa 12 un ejemplo de aplicación del algoritmo OLS, con el cual aplicamos regresión lineal múltiple sobre el conjunto de datos “Boston”, cuya descripción se mostró en la Tabla 4.



```
import numpy as np
from math import sqrt
from pprint import pprint
from sklearn import datasets
from sklearn.dummy import DummyRegressor
from sklearn.model_selection import cross_validate
from sklearn.model_selection import KFold
from sklearn.metrics import make_scorer
from sklearn.metrics import mean_squared_error
```

```
# Carga de datos.  
datos = datasets.load_boston()  
  
# Algoritmo de aprendizaje.  
reg = linear_model.LinearRegression()  
  
# Métricas de evaluación.  
metricas = {  
    'MAE': metrics.mean_absolute_error,  
    'RMSE': lambda y, y_pred:  
        sqrt(metrics.mean_squared_error(y, y_pred)),  
    'MAPE': lambda y, y_pred:  
        np.mean(np.abs((y - y_pred) / y)) * 100,  
    'R2': metrics.r2_score}  
  
# Validación y obtención de las predicciones del modelo.  
seed = 1  
y_pred = cross_val_predict(reg, datos.data, datos.target,  
                           cv = KFold(n_splits=10, random_state=seed))  
  
# Cálculo de las métricas de evaluación.  
MAE = metricas['MAE'](datos.target, y_pred)  
RMSE = metricas['RMSE'](datos.target, y_pred)  
MAPE = metricas['MAPE'](datos.target, y_pred)  
R2 = metricas['R2'](datos.target, y_pred)  
  
# Gráfica de realidad vs. predicción.  
fig, ax = plt.subplots()  
ax.scatter(datos.target, y_pred, edgecolors=(0, 0, 0))  
ax.plot([datos.target.min(), datos.target.max()],  
        [datos.target.min(), datos.target.max()], 'k--', lw=4)  
ax.set_xlabel('Valor real de la clase')  
ax.set_ylabel('Predicción')  
plt.title("MAE: %.3f RMSE: %.3f MAPE: %.3f R2: %.3f" %  
          (MAE, RMSE, MAPE, R2))  
plt.show()
```

Programa 12. Regresión lineal múltiple con el algoritmo OLS y el conjunto de datos “Boston”.

La gráfica generada por el Programa 12 es:

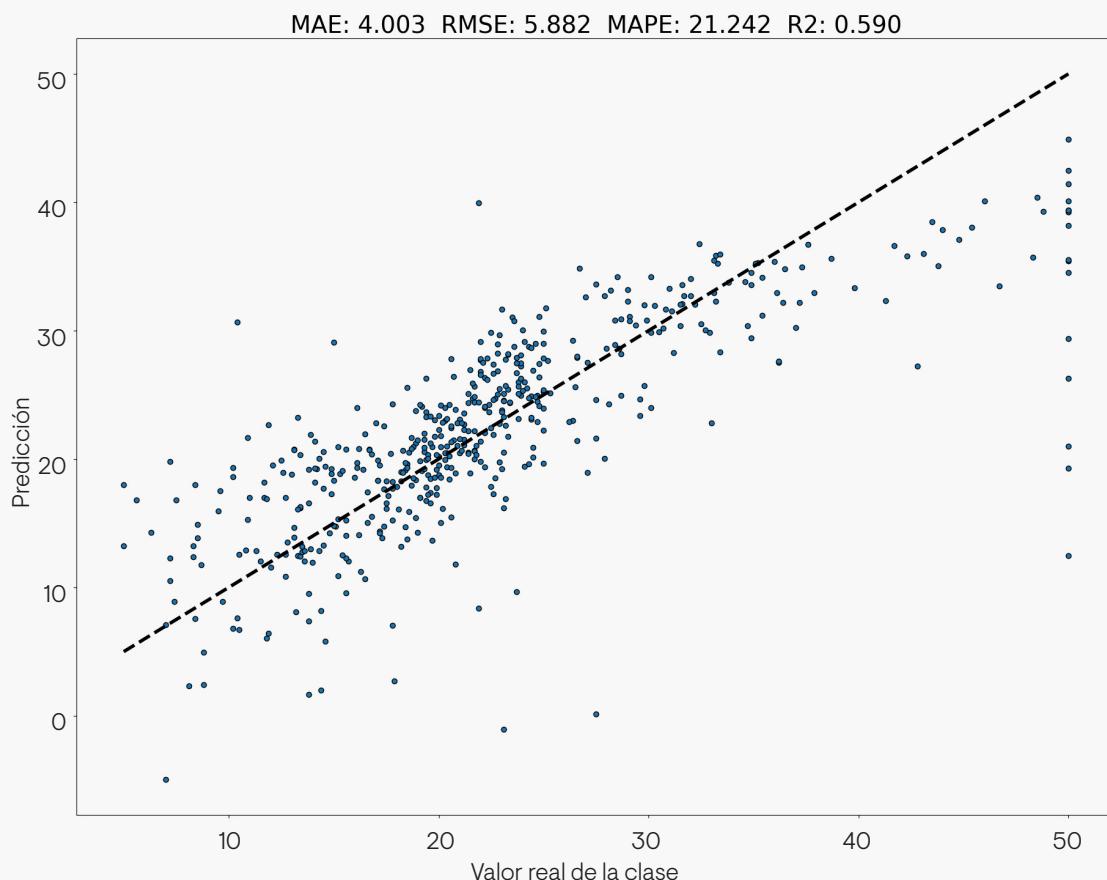


Figura 7. Gráfica generada por el Programa 12.

En el Programa 12, tras cargar los datos del conjunto “Boston”, construimos un objeto de tipo `LinearRegression`, en el que scikit-learn tiene implementado el algoritmo OLS. Definimos un diccionario de métricas cuya clave es el nombre de la métrica y cuyo valor es la función que implementa la métrica. Las métricas MAE y R^2 vienen implementadas en scikit-learn, pero RMSE y MAPE las definimos usando expresiones lambda.



Con la función `cross_val_predict()` extraemos las predicciones realizadas por el modelo tras llevar a cabo una validación cruzada con 10 bolsas. Nótese que, a diferencia de los ejercicios anteriores de la asignatura, hemos establecido una semilla, `seed`, para la generación de números aleatorios. Esta semilla afecta a las bolsas generadas durante el proceso de validación cruzada. Para que los resultados sean los mismos en cualquier ejecución del programa, basta con utilizar siempre la misma semilla. Dado que el algoritmo OLS no es estocástico (es decir, no lleva asociada aleatoriedad en ninguno de sus pasos), el resultado del Programa 12 debería ser el mismo en todas sus ejecuciones.

El cálculo de las métricas de evaluación puede hacerse de forma sencilla usando el diccionario de métricas. Tras acceder a un elemento del diccionario mediante su clave alfanumérica (nombre de la métrica), obtenemos la función de dicha métrica `y`, directamente, la invocamos pasándole como argumentos los vectores de la clase real (`datos.target`) y predicha (`y_pred`).

Una vez obtenidas las métricas, visualizamos los pares de valores <clase real, clase predicha> en una gráfica de tipo nube de puntos (scatter en inglés), la cual ilustra la relación realidad vs. predicción. Dicha gráfica puede apreciarse en la Figura 7. Los puntos de color azul representan las predicciones, mientras que la línea discontinua de color negro muestra la bisectriz del primer cuadrante (recta $y = x$).

La línea discontinua de color negro de la Figura 7 sirve como referencia para identificar la bondad de las predicciones. Cuanto más distantes estén los puntos de color azul (predicciones) de la línea discontinua, peores son las predicciones. Las predicciones pueden estar por encima de la línea, en cuyo caso diremos que las predicciones son **sobreestimaciones** de la realidad (el valor predicho es mayor que el real). Por el contrario, si las predicciones se encuentran por debajo de la línea, diremos que son **subestimaciones** (pues el valor predicho es menor que el real). Es muy importante no confundir esta línea con la recta de regresión, la cual no se muestra en la figura.

En el título de la gráfica (parte superior) se muestran los valores de las métricas de evaluación que hemos estudiado: MAE, RMSE, MAPE y R^2 . A la hora de interpretar los resultados obtenidos, y determinar si son buenos resultados o no, debemos tener en cuenta los criterios exigidos de calidad al inicio del proyecto real de minería de datos, tal como se explicó en el apartado 1.1.8 del Capítulo 1.



Esto quiere decir que los valores de las métricas de evaluación, por sí solas de manera aislada, no informan acerca de si el modelo es suficientemente bueno o si es preciso seguir mejorándolo. Para saber si el modelo es suficientemente bueno, es necesario comparar los resultados con los obtenidos por otros modelos, con los errores cometidos por operativa humana o con los resultados obtenidos en problemas similares.

La interpretación puramente técnica de las métricas de evaluación únicamente señala que el modelo obtenido mediante el algoritmo OLS sobre el conjunto “Boston” es mejor que una predicción obtenida por la esperanza matemática de la clase (media aritmética) (algoritmo `DummyRegressor`), pues $R^2 > 0$. La misma interpretación podemos extraer si observamos el valor de MAE (4,003) y la desviación típica de la clase ($\sigma_y = 9,188$), pues $1 - \frac{\text{MAE}}{\sigma_y} = 1 - \frac{4,003}{9,188} = 0,564 > 0$. Nótese en este último cálculo que, en lugar de operar con las diferencias al cuadrado (como ocurre con R^2), operamos con las diferencias en valor absoluto (MAE y desviación típica).

La regresión lineal posee una serie de ventajas e inconvenientes que es importante señalar. Las ventajas fundamentales que podríamos destacar son:

- La simplicidad del modelo. El modelo consta únicamente de una ecuación matemática basada en una combinación lineal de los atributos.

- La fácil interpretabilidad del modelo. El modelo es sencillo de interpretar, pues es fácil observar la importancia de cada atributo a partir de la magnitud de su coeficiente asociado. A mayor coeficiente, en valor absoluto, mayor importancia tiene el atributo. Además, por el signo del coeficiente sabemos si la relación de cada atributo con la clase es directa (signo positivo) o inversa (signo negativo).
- La gran eficacia en problemas de índole lineal. Cuando la naturaleza del problema de predicción es lineal, esto es, cuando hay relación lineal entre los atributos y la clase, los modelos de regresión lineal se ajustan muy bien a la realidad.
- El tiempo de ejecución del entrenamiento muy razonable. El entrenamiento consiste, como hemos estudiado, en el ajuste de los coeficientes de la combinación lineal de los atributos mediante un proceso iterativo de búsqueda (generalmente por descenso de gradiente). Dependiendo de la adecuación del parámetro tasa de aprendizaje, citado anteriormente, a la naturaleza de los datos, la convergencia hacia el óptimo puede llevar más o menos tiempo. Además, los tiempos de ejecución se pueden reducir tomando muestras del conjunto de datos para evaluar la función objetivo, en lugar de todos los datos. Esta última optimización del modelo lo convertiría en estocástico, pues los resultados pueden depender de la elección aleatoria de las muestras.
- El tiempo de ejecución de la predicción casi instantáneo. Una vez entrenado el modelo de regresión lineal, realizar las predicciones es trivial e inmediato, pues tan solo hay que evaluar la función matemática (combinación lineal de atributos) usando los atributos de los ejemplos de test.

Por otra parte, para que la regresión lineal funcione con eficacia, se asumen una serie de hipótesis sobre los datos:

- La independencia entre los atributos. Los modelos de regresión lineal asumen que los atributos no poseen una correlación significativa entre ellos, es decir, se requiere que sean variables estadísticamente independientes.
- La distribución normal de los datos. Para que los modelos de regresión lineal sean eficaces, los valores de los atributos deberían逼近arse a una distribución normal. Especialmente, la presencia de valores anómalos (outliers) en el conjunto de datos afecta muy negativamente a los modelos lineales.
- La relación lineal entre los atributos y la clase. Tal como hemos comentado anteriormente, si la relación de tipo lineal entre los atributos y la clase es escasa o nula, los modelos lineales serán ineficaces. En la práctica, la linealidad de estas relaciones se puede detectar tanto gráficamente (mediante gráficas de tipo scatter por pares entre cada atributo y la clase) como numéricamente (mediante matrices de correlación, por ejemplo).
- La homocedasticidad de los datos. Esta característica hace referencia a que los errores cometidos por los modelos tengan una varianza constante pese a la variación de los valores de los atributos. Esto es, los errores del modelo no dependen de los valores de los atributos.

Existen en la literatura varios algoritmos que pueden mejorar el algoritmo OLS en determinados escenarios. En concreto, las variantes **Ridge** y **Lasso** incorporan nuevos términos a la función objetivo utilizada para ajustar los coeficientes del modelo lineal. Estos nuevos términos permiten evitar el sobreajuste del modelo a los datos, especialmente cuando estos poseen valores anómalos.



Enlace de interés

En el siguiente artículo puede encontrar una figura (Figura 3) que muestra un ejemplo en el que se aprecia la distribución de los errores de un modelo entrenado con datos de un atributo que posee homocedasticidad (primera gráfica). Por el contrario, la heterocedasticidad se da cuando los errores cometidos por el modelo no se distribuyen de igual forma por los valores del atributo (segunda y tercera gráficas).

<https://pareonline.net/getvn.asp?v=8&n=2>

En el siguiente artículo de la documentación oficial de scikit-learn se describen y se ponen en práctica numerosos algoritmos de regresión lineal, incluyendo OLS, Ridge y Lasso.

http://scikit-learn.org/stable/modules/linear_model.html

3.2. Vecinos más cercanos

La aproximación que estudiamos a continuación recibe el nombre de vecinos más cercanos (KNN de *k nearest neighbors* en inglés). Esta aproximación se basa en la premisa de que ejemplos similares en sus atributos presentan también un comportamiento similar en el valor de sus clases.

Aunque nos centremos en este algoritmo de aprendizaje automático para problemas de regresión, también puede ser aplicado tanto en clasificación como en problemas no supervisados.

La idea fundamental de la técnica de los vecinos más cercanos para regresión consiste en producir como predicción el promedio de las clases de los ejemplos de entrenamiento más parecidos (vecinos) al ejemplo de test que hay que predecir. Definiremos en este capítulo qué debe considerarse más parecido, aspecto que, como veremos, afecta a la eficacia del algoritmo.



El proceso de entrenamiento de KNN es muy sencillo, pues tan solo consiste en almacenar el conjunto de datos de entrenamiento, para poder ser consultado posteriormente en la fase de predicción. KNN no elabora ningún modelo a partir de los datos. Por esta razón, el proceso de entrenamiento apenas consume tiempo de ejecución.

Para llevar a cabo las predicciones, el algoritmo KNN clásico se basa en los dos siguientes elementos configurables: el número $k \in \mathbb{N}$ de vecinos más cercanos y la función de distancia d (para poder evaluar la similitud entre los ejemplos).

Con respecto a esta última, existen numerosas funciones de distancia propuestas en la literatura. En este capítulo usaremos la distancia de Minkowski, la cual se define así:

$$\text{Minkowski}(e_i, e_j) = \left(\sum_{d=1}^p |x_{i,d} - x_{j,d}|^q \right)^{1/q}$$

Cuando $q = 2$, la distancia de Minkowski también se conoce como distancia euclídea, tal como se muestra en la siguiente ecuación. En la Figura 8 se muestran los lugares geométricos de los puntos equidistantes a un punto de origen utilizando la función de distancia de Minkowski con distintos valores de q .

$$\text{Euclidea}(e_i, e_j) = \sqrt{\sum_{d=1}^p |x_{i,d} - x_{j,d}|^2}$$

La función de distancia es un elemento crítico de KNN que afecta en gran medida a la bondad de las predicciones. Cada problema puede requerir una función de distancia más adecuada a la naturaleza y distribución de los datos. En la mayoría de los casos, la distancia más utilizada es la euclídea.

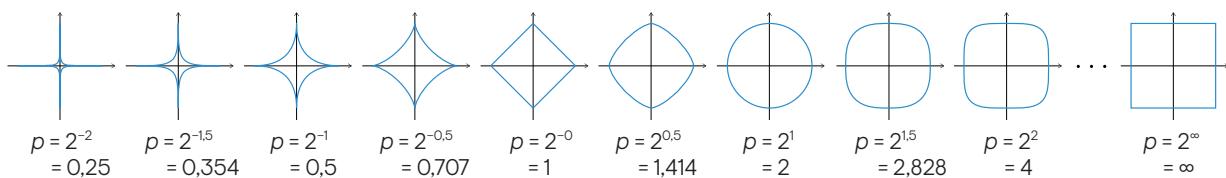


Figura 8. Distancia de Minkowski para diferentes valores de q . Por Waldir bajo licencia CC BY-SA 3.0. Recuperado de https://commons.wikimedia.org/wiki/File:2D_unit_balls.svg



Enlace de interés

En la clase `DistanceMetric` del paquete `sklearn.neighbors` se encuentran definidas una gran variedad de funciones de distancia. Además, los atributos categóricos (números discretos, cadenas o booleanos) requieren funciones de distancia específicas, también incluidas en la clase `DistanceMetric`.

<http://scikit-learn.org/stable/modules/generated/sklearn.neighbors.DistanceMetric.html>

Cualquier función de distancia que se pretenda usar debería cumplir las cuatro siguientes propiedades matemáticas:

- No negatividad: $d(e_i, e_j) \geq 0$
- Identidad: $d(e_i, e_j) = 0 \Leftrightarrow e_i = e_j$
- Simetría: $d(e_i, e_j) = d(e_j, e_i)$
- Desigualdad triangular: $d(e_i, e_r) + d(e_r, e_j) \geq d(e_i, e_j)$

La distancia de Minkowski solo cumple las cuatro anteriores propiedades si $q \geq 1$. Una vez introducidos los elementos sobre los que se apoya el algoritmo KNN (número k de vecinos y función de distancia), definimos a continuación los vecinos más cercanos v_1, \dots, v_k de un ejemplo e_i como los índices de aquellos ejemplos del conjunto de entrenamiento con las menores distancias a dicho ejemplo e_i .

Esto es, v_1, \dots, v_k son los vecinos más cercanos de e_i si cumplen la siguiente expresión:

$$d(e_i, e_{v_1}) \leq d(e_i, e_{v_2}) \leq \dots \leq d(e_i, e_{v_k}) \leq d(e_i, e_j)$$

$$1 \leq j \leq n \wedge j \neq v_l \wedge i \neq j \quad \forall l \in \{1, \dots, k\}$$

$$v_l \in \{1, \dots, n\} \quad v_l \neq i$$

Otra forma de indicar al algoritmo KNN cómo seleccionar los vecinos, también muy usada en la literatura, es mediante un radio r . Esto quiere decir que, en lugar de usar un número fijo k de vecinos más cercanos, para calcular los vecinos del ejemplo e_i se toman todos los ejemplos que se encuentren hasta una distancia r de dicho ejemplo. En este caso, v_1, \dots, v_k son los vecinos más cercanos de e_i si cumplen:

$$d(e_i, e_{v_j}) \leq r \quad \forall j \in \{1, \dots, k\}$$

$$v_j \in \{1, \dots, n\} \quad v_j \neq i$$

Una vez definidos todos estos elementos, la predicción de KNN se puede expresar mediante la siguiente ecuación (siendo v_1, \dots, v_k los vecinos más cercanos de e_i):

$$\hat{y}_i = \frac{1}{k} \cdot \sum_{j=1}^k y_{v_j}$$

Tal como se aprecia en la anterior ecuación, la predicción \hat{y} resulta del promedio de las clases de los k vecinos más cercanos a e_i . En la Figura 9 se muestra un ejemplo con predicciones de KNN (estrellas de color azul) para tres ejemplos de test (estrellas de color verde) usando $k = 3$ vecinos más cercanos. El conjunto de datos ilustrado en la Figura 9 solo tiene un atributo (x):

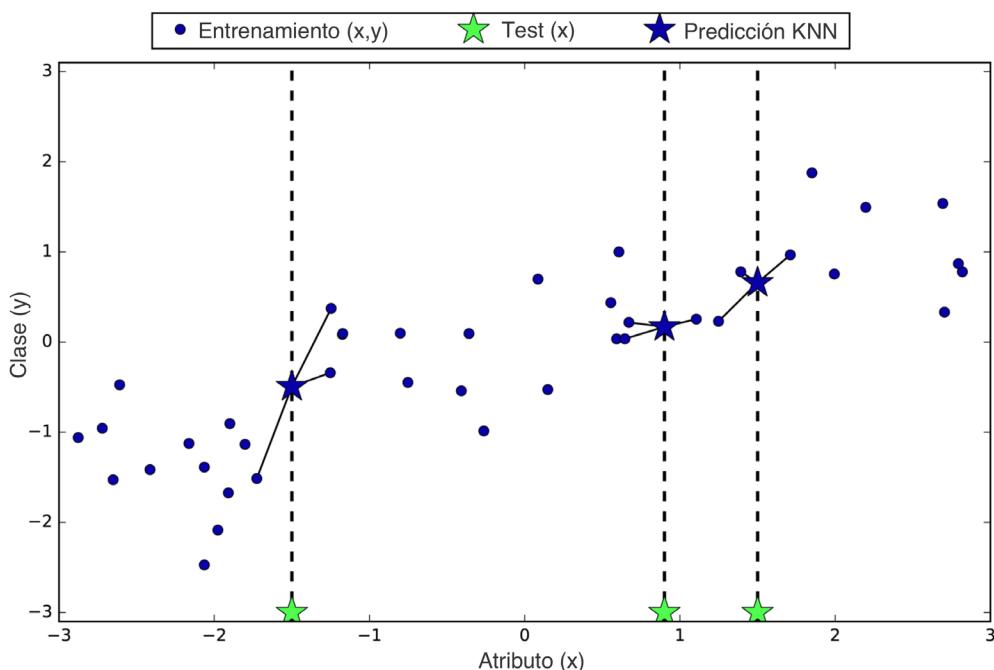


Figura 9. Predicción de KNN en regresión con un único atributo (x). Adaptado de *Introduction to Machine Learning with Python* (p. 42), por A. C. Mueller y S. Guido, 2016, Sebastopol: O'Reilly.

A continuación, mostramos en el Programa 13 un ejemplo de aplicación del algoritmo KNN sobre el conjunto de datos “Boston”, utilizando $k = 10$ vecinos más cercanos. Tal como hemos procedido en anteriores programas, se ha utilizado una validación cruzada con 10 bolsas y se han evaluado las métricas MAE, RMSE, MAPE y R^2 .

```
</>
import numpy as np
from math import sqrt
from sklearn import datasets
import sklearn.metrics as metrics
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_predict
from sklearn.neighbors import KNeighborsRegressor
from evaluacion_funciones import *

# Carga de datos.
datos = datasets.load_boston()

# Algoritmos de aprendizaje.
k = 10
reg = KNeighborsRegressor(n_neighbors = k)

# Métricas de evaluación.
metricas = {
    'MAE': metrics.mean_absolute_error,
    'RMSE': lambda y, y_pred:
        sqrt(metrics.mean_squared_error(y, y_pred)),
    'MAPE': lambda y, y_pred:
        np.mean(np.abs((y - y_pred) / y)) * 100,
    'R2': metrics.r2_score}

# Validación y obtención de las predicciones del modelo.
seed = 1
y_pred = cross_val_predict(reg, datos.data, datos.target,
                           cv = KFold(n_splits=10, random_state=seed))

# Evaluación y presentación de resultados.
eval = evaluacion(datos.target, y_pred, metricas)
grafica_real_vs_pred(datos.target, y_pred, eval, "KNN"+str(k))
```

Programa 13. Regresión con el algoritmo KNN y el conjunto de datos “Boston”.

La gráfica generada por el Programa 13 es:

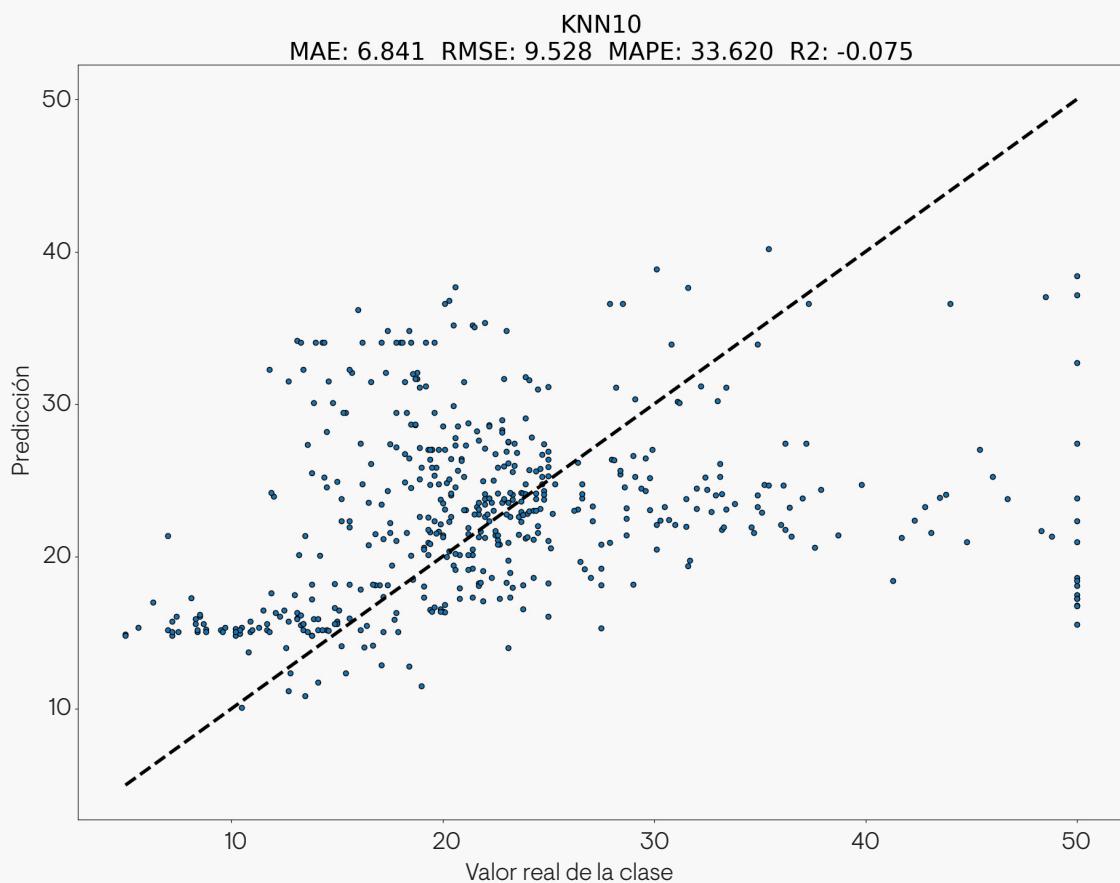


Figura 10. Gráfica generada por el Programa 13.

En el Programa 13 hemos usado una librería propia con funciones de evaluación que se encuentra en el archivo “evaluacion_funciones.py”. En concreto, se han usado las funciones `evaluacion()` y `grafica_real_vs_pred()` de dicha librería. Tras cargar el conjunto de datos, creamos un objeto de tipo `KNeighborsRegressor`, en el que tenemos implementado el algoritmo KNN. Además, le indicamos que deseamos usar 10 vecinos más cercanos, gracias al argumento `n_neighbors`.

A continuación, definimos en un diccionario las métricas que deseamos utilizar para evaluar los resultados. Continuamos utilizando las cuatro métricas que hemos estudiado: MAE, RMSE, MAPE y R^2 . Mediante la función `cross_val_predict()` realizamos la validación cruzada con 10 bolsas y obtenemos las predicciones en el vector `y_pred` de tipo `ndarray`.

Finalmente, calculamos las métricas de evaluación gracias a la función `evaluacion()`, desarrollada para la asignatura en el fichero “evaluacion_funciones.py”. Esta función nos devuelve un diccionario cuyas claves son los nombres de las métricas y sus valores las medidas calculadas. La llamada a la función `grafica_real_vs_pred()` genera la gráfica que se muestra en la Figura 10.

Esta función recibe el vector de clases reales, el de clases predichas, el diccionario de métricas evaluadas y el nombre de la técnica utilizada en la predicción, para incluirla como texto en el título de la gráfica.

Tal como se puede apreciar en la Figura 10, las medidas de evaluación de KNN en el conjunto “Boston” son pésimas, mucho peores que las conseguidas por la regresión lineal con el algoritmo OLS (véase Figura 7). En concreto, la métrica $R^2 = -0,075$ está cercana a cero, lo cual significa que KNN tiene aproximadamente la misma eficacia que `DummyRegressor`, es decir, da siempre como predicción la media aritmética de las clases de los ejemplos de entrenamiento.

Además, el valor del error $MAE = 6,841$, alcanzado por KNN, es mucho más alto (peor) al conseguido por OLS ($MAE = 4,003$).

Como veremos a continuación, el algoritmo KNN se apoya en ciertos aspectos que afectan enormemente a su funcionamiento. En concreto, la eficacia predictiva del algoritmo KNN depende en gran medida de los siguientes elementos:

- La escala de valores de los atributos. Los atributos cuyos valores son más altos contribuyen más al cálculo de los vecinos que los atributos con valores más bajos. Es necesario igualar las escalas de los atributos antes de utilizar el algoritmo KNN, para que las diferencias entre los valores de los atributos sean equiparables. Esto se consigue mediante la normalización o la estandarización de los datos, tal como vimos en el Capítulo 1 (apartado 1.4.2).



Ejemplo

Por ejemplo, un atributo que recoge el peso de una persona (en kilogramos) contribuiría más que un atributo para la altura de la persona (en metros). Un conjunto de datos de personas puede tener un valor promedio del atributo peso de 80 (kg) y un valor promedio del atributo altura de 1,7 (m). Una diferencia de una unidad en el atributo peso no debería ser equivalente a la del atributo altura.

- La medida de similitud entre ejemplos. La forma de cuantificar numéricamente el parecido entre los ejemplos del conjunto de datos debe ser acorde a la naturaleza del problema y al significado de los atributos.
 - **La función de distancia.** La función de distancia es la que determina el valor numérico que mide la diferencia entre dos ejemplos cualesquiera. Tal como hemos indicado anteriormente, existen numerosas funciones de distancia disponibles en la literatura. Debe tenerse en cuenta la aplicabilidad de la función de distancia al tipo de datos de los atributos.
 - **La ponderación de atributos.** Hasta ahora hemos supuesto que todos los atributos tienen la misma importancia en su relación con la clase. Sin embargo, esto generalmente no es así. Es posible calibrar la importancia que queremos dar a los atributos dentro de la función de distancia, como veremos a continuación.

- La selección de vecinos más cercanos. Una vez que se han evaluado las distancias desde un ejemplo e_i hasta todos los ejemplos del conjunto de entrenamiento, la forma de seleccionar los vecinos más cercanos de e_i es importante y afecta al resultado de la predicción. Las estrategias que hemos estudiado consisten en seleccionar los k ejemplos con las menores distancias, o bien en seleccionar todos los ejemplos cuya distancia con e_i es menor o igual a un determinado radio r . No obstante, también es posible diseñar otras nuevas estrategias con el fin de mejorar la eficacia de las predicciones.
- El cálculo de la predicción a partir de los vecinos más cercanos. Una vez que están seleccionados los vecinos más cercanos, es preciso extraer de ellos la información necesaria para construir el valor de predicción.
 - **Función de resumen.** La función de resumen toma los valores de las clases de los vecinos más cercanos y realiza con ellos un cálculo que produce un valor representativo de estos. La función de resumen que hemos estudiado es la media aritmética, pero podría utilizarse cualquier otra función que estimemos conveniente.
 - **Ponderación de vecinos.** Hemos supuesto que todos los vecinos contribuyen por igual al cálculo de la predicción. Sin embargo, es posible que algunos vecinos tengan más relación con el ejemplo e_i que hay que predecir que otros.

La ponderación de atributos puede aplicarse o bien transformando el conjunto de datos (multiplicando los valores de los atributos por sus pesos correspondientes), o bien introduciendo los pesos dentro de la función de distancia (y manteniendo el conjunto de datos intacto). Siguiendo esta última forma, en la siguiente ecuación se muestra la distancia de Minkowski modificada para introducir las ponderaciones w en los atributos.

$$\text{Minkowski}(e_i, e_j) = \left(\frac{1}{\sum w} \cdot \sum_{d=1}^p w_d |x_{i,d} - x_{j,d}|^q \right)^{1/q}$$

Por otra parte, la ponderación de vecinos se aplica en la función de cálculo de la predicción de KNN que estudiamos anteriormente. En concreto, el valor predicho por KNN, teniendo en cuenta ponderaciones (μ) en los vecinos, puede expresarse del siguiente modo:

$$\hat{y}_i = \frac{1}{\sum \mu} \cdot \sum_{j=1}^k \mu_j \cdot y_{v_j}$$

Teniendo en cuenta los elementos anteriores, los cuales afectan a la eficacia de KNN, mostramos en el Programa 14 un ejemplo de aplicación de KNN en validación cruzada de 10 bolsas con cuatro diferentes configuraciones:

- Algoritmo KNN ($k = 10$) aplicado al conjunto de datos original. El resultado obtenido con esta configuración es el mismo que el del Programa 13 y se mostró en la Figura 10.
- Estandarización de los datos + KNN ($k = 10$). Antes de utilizar KNN, los datos fueron estandarizados utilizando la clase StandardScaler, tal como se estudió en el apartado 1.4.2 del Capítulo 1. El resultado con esta configuración se muestra en la Figura 11.
- Estandarización de los datos + Selección de atributos + KNN ($k = 10$). En esta configuración, tras estandarizar los datos, se ha realizado una selección de atributos univariante mediante la métrica f_regression, tomando el 10 % de los atributos más relevantes. El resultado se muestra en la Figura 12.

- Estandarización de los datos + Selección de atributos + KNN ($k = 10, p = 1$). En esta configuración se han realizado los mismos pasos que en la anterior y se ha utilizado la distancia de Minkowski con $q = 1$ (también llamada distancia de Manhattan en este caso). El resultado se muestra en la Figura 13.

```
</>  
import numpy as np  
from math import sqrt  
from sklearn import datasets  
import sklearn.metrics as metrics  
from sklearn.pipeline import Pipeline  
from sklearn.neighbors import KNeighborsRegressor  
from sklearn.model_selection import cross_val_predict  
from sklearn.model_selection import KFold  
from sklearn.preprocessing import StandardScaler  
from sklearn.feature_selection import f_regression  
from sklearn.feature_selection import SelectPercentile  
from evaluacion_funciones import *  
  
# Carga de datos.  
datos = datasets.load_boston()  
  
# Construcción de Los algoritmos de aprendizaje.  
k = 10  
base = 'KNN' + str(k)  
algoritmos = []  
  
# Algoritmo 1: KNN  
algoritmos[base] = KNeighborsRegressor(n_neighbors = k)  
  
# Algoritmo 2: Estandarización + KNN  
pasos = [('estandarizacion', StandardScaler()),  
          ('reg', algoritmos['KNN'+str(k)])]  
algoritmos['Est+'+base] = Pipeline(pasos)  
  
# Algoritmo 3: Estandarización + Selección atributos + KNN  
pasos = [('estandarizacion', StandardScaler()),  
          ('selatr', SelectPercentile(score_func=f_regression,  
                                      percentile=10)),  
          ('reg', algoritmos['KNN'+str(k)])]  
algoritmos['Est+SelAtr+'+base] = Pipeline(pasos)
```

```

# Algoritmo 4: Estandarización + Selecc. atributos + KNN(p=1)
pasos = [('estandarizacion', StandardScaler()),
          ('selatr', SelectPercentile(score_func=f_regression,
                                         percentile=10)),
          ('reg', KNeighborsRegressor(n_neighbors = k, p=1))]
algoritmos['Est+SelAtr+'+'base+'+'(p=1)'] = Pipeline(pasos)

# Métricas de evaluación.
metricas = {
    'MAE': metrics.mean_absolute_error,
    'RMSE': lambda y, y_pred:
        sqrt(metrics.mean_squared_error(y, y_pred)),
    'MAPE': lambda y, y_pred:
        np.mean(np.abs((y - y_pred) / y)) * 100,
    'R2': metrics.r2_score}

# Validación y obtención de las predicciones del modelo.
seed = 1
y_pred = {}
for nombre, alg in algoritmos.items():
    y_pred[nombre] = cross_val_predict(alg, datos.data,
                                       datos.target, cv=KFold(n_splits=10, random_state=seed))

# Evaluación y presentación de resultados.
for nombre, alg in algoritmos.items():
    eval = evaluacion(datos.target, y_pred[nombre], metricas)
    grafica_real_vs_pred(datos.target, y_pred[nombre],
                          eval, nombre)
  
```

Programa 14. Regresión con KNN usando diferentes configuraciones.

Tal como se puede apreciar en los resultados obtenidos por el Programa 14, la estandarización previa de los datos hace disparar las métricas de evaluación alcanzadas por KNN, que superan incluso las obtenidas por el algoritmo OLS.

En concreto, se alcanza un $R^2 = 0,64$, frente al $R^2 = -0,075$ obtenido sin estandarizar los datos. Comprobamos, por tanto, la importancia de equiparar la escala de valores de los atributos.

Este aspecto es crítico y debe ser tenido en cuenta cuando vayan a usarse posteriormente algoritmos de modelado basados en distancias.

Otro aspecto fundamental que hemos señalado anteriormente es la ponderación de los atributos. En el Programa 14, al hacer una selección de atributos clásica, las ponderaciones de atributos utilizadas son binarias, es decir, o se incluye el atributo en el cálculo de la función de distancia ($w = 01$) o no se incluye ($w = 0$). En otros programas se podrían utilizar valores de ponderación en un rango continuo, con el objetivo de afinar los resultados. El resultado obtenido mejoró algo el de la configuración anterior, alcanzando un $R^2 = 0,727$.

En la última configuración probada, observamos que, al utilizar la distancia de Manhattan en lugar de la euclídea, los valores de las métricas mejoran un poco más, alcanzando un $R^2 = -0,727$ y un MAE = 3,27. Frente al $R^2 = 0,59$ y al MAE = 4,003 que obtuvimos con OLS, los resultados con KNN son significativamente mejores.

En el Programa 14 hemos automatizado aún más el proceso de modelado mediante el uso de diccionarios de algoritmos y pipelines. De este modo, el programa puede ser fácilmente reutilizado para incorporar nuevos algoritmos y configuraciones.

Los pipelines son un versátil mecanismo implementado en scikit-learn que nos permite diseñar una cadena de pasos para automatizar un proceso complejo, lo cual produce un objeto de tipo pipeline que, a todos los efectos, puede actuar como un nuevo regresor o clasificador.

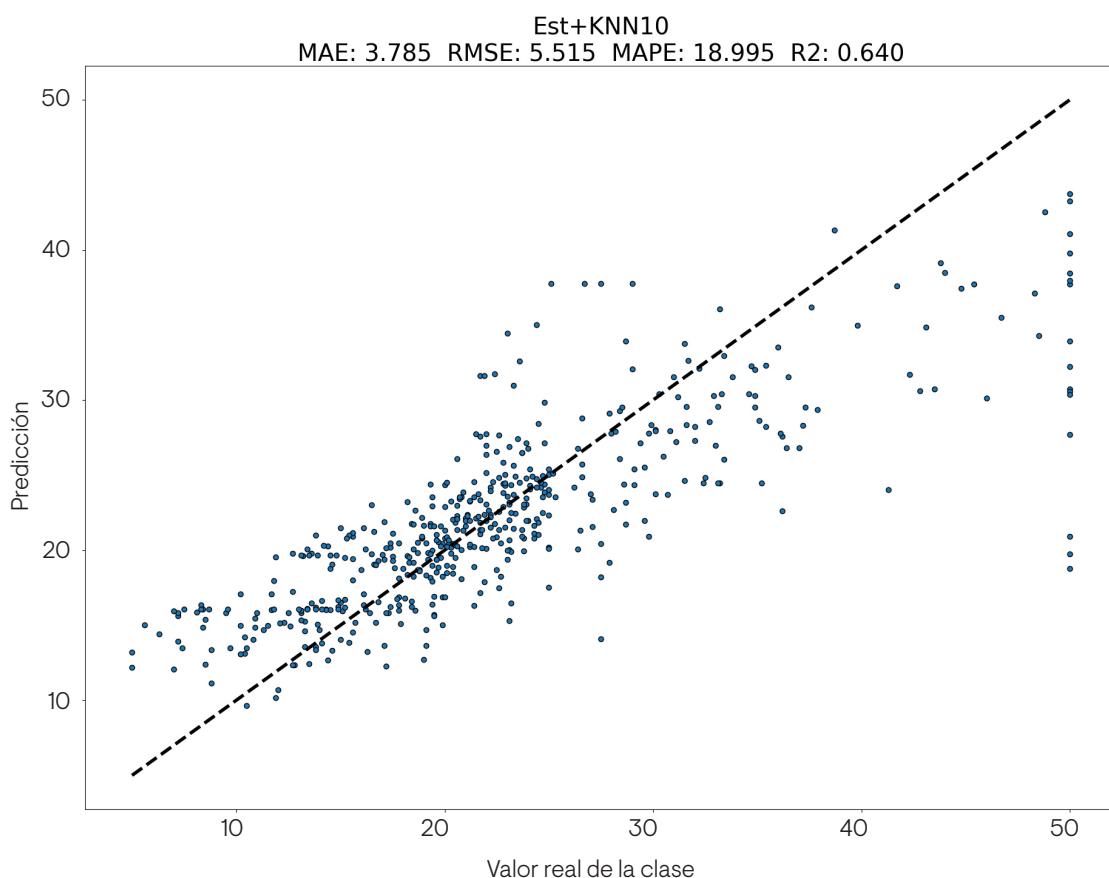


Figura 11. Gráfica producida por el Programa 14. Pipeline con estandarización y KNN.

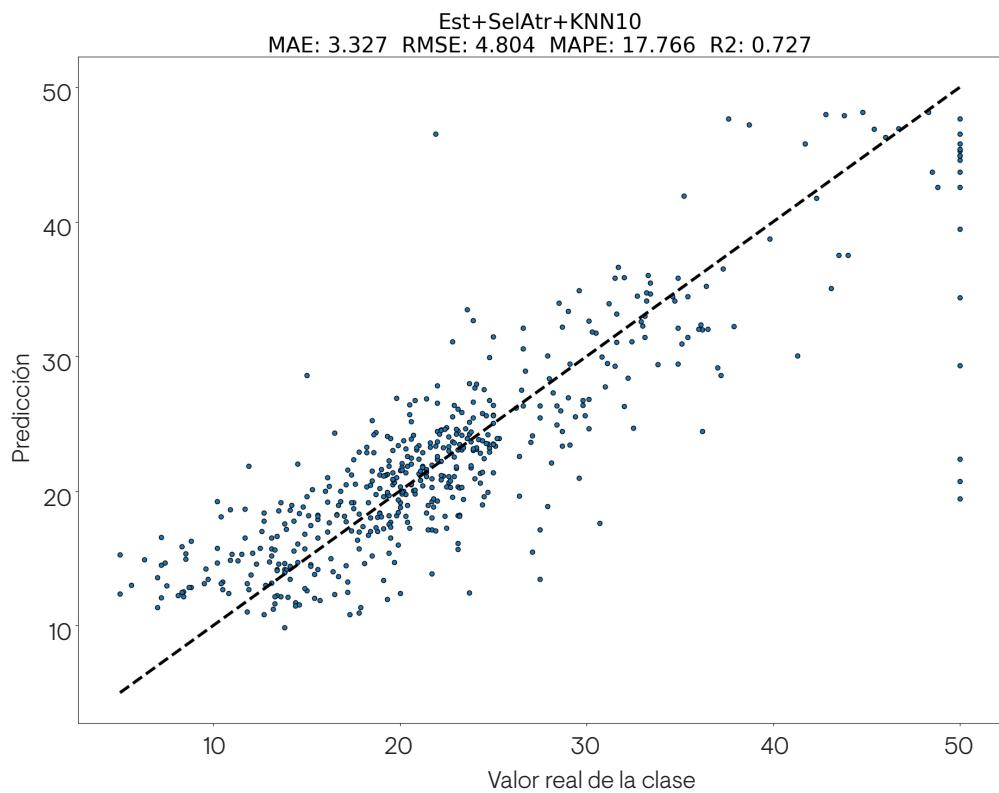


Figura 12. Gráfica producida por el Programa 14. Pipeline con estandarización, selección de atributos y KNN.

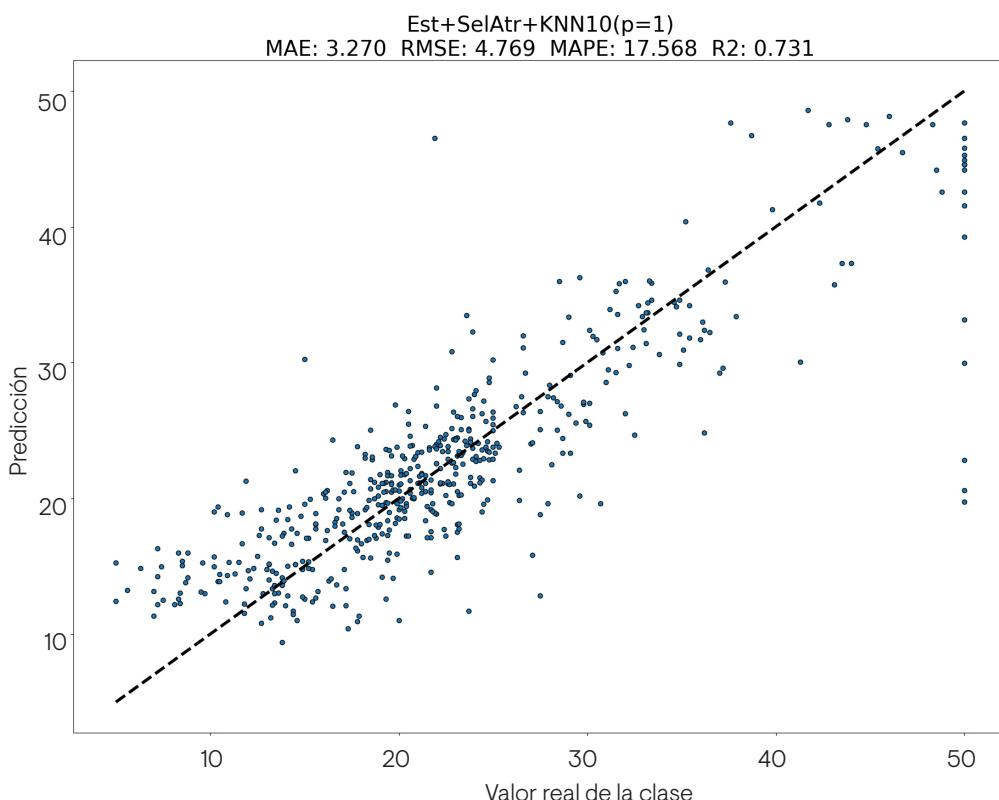


Figura 13. Gráfica producida por el Programa 14. Pipeline con estandarización, selección de atributos y KNN con distancia Minkowski ($p = 1$).

El algoritmo KNN de vecinos más cercanos posee una serie de ventajas e inconvenientes que es importante señalar. Las ventajas fundamentales que podríamos destacar son:

- La simplicidad del modelo. El modelo es tan simple que realmente no tiene modelo propio, tan solo los datos de entrenamiento.
- La fácil interpretabilidad de las predicciones. Las predicciones realizadas por el algoritmo KNN pueden ser fácilmente interpretadas si mostramos un resumen de los vecinos utilizados en la predicción.



Ejemplo

Por ejemplo, un modelo KNN que aprende de un histórico de datos de campañas agronómicas de cultivos de sandías puede predecir que la cosecha en la próxima campaña será aproximadamente de 12,5 toneladas. Y es posible extraer del modelo que es debido a que el clima en los últimos meses y otros condicionantes previos se parecen mucho a los de las campañas de 2002 y 2006 (observando los atributos de los vecinos más cercanos, por ejemplo, con $k = 2$).

Esta explicación, junto con el clima, los condicionantes y las cosechas de dichas campañas, puede ayudar a entender por qué el modelo dio 12,5 toneladas como predicción.

- El tiempo de ejecución del entrenamiento prácticamente nulo. El tiempo empleado en el entrenamiento consiste únicamente en almacenar una referencia en los datos de entrenamiento para el cálculo de distancias durante la fase de predicción. No hay construcción de ningún modelo a partir de los datos.

Por otra parte, el algoritmo KNN posee ciertos inconvenientes:

- El tiempo de ejecución alto en la fase de predicción. Como hemos comentado anteriormente, para realizar la predicción a partir de un ejemplo de test, es preciso calcular las distancias a todos los ejemplos de entrenamiento, lo cual es el proceso más costoso del algoritmo. Este puede ser un importante hándicap cuando el conjunto de datos es enorme, con varios millones de instancias.
- La falta de generalización. Al no crearse ningún modelo a partir de los datos, si estos poseen anomalías, la eficacia del algoritmo puede verse afectada, especialmente con valores bajos de k . Además, si los datos están muy dispersos en el espacio de atributos o bien no son representativos, los errores de generalización del modelo pueden ser elevados.
- La necesidad de atributos relevantes y en igual escala. Tal como hemos comprobado en los ejemplos de este capítulo, la presencia de atributos en diferentes escalas puede afectar negativamente a los resultados. Además, los datos con una alta dimensionalidad (gran cantidad de atributos) deberían ser reducidos mediante una selección de los atributos más relevantes.



Enlace de interés

La búsqueda de los vecinos más cercanos a un ejemplo e_p , necesaria para llevar a cabo el algoritmo KNN, tiene un coste lineal con respecto al número n de instancias del conjunto de entrenamiento. Existen algoritmos y estructuras de datos más eficientes para la búsqueda de los vecinos más cercanos, como **KDTree** y **BallTree**. En la siguiente página web es posible documentarse acerca de estas técnicas y aprender cómo utilizarlas usando las librerías de scikit-learn.

<https://scikit-learn.org/stable/modules/neighbors.html#nearest-neighbor-algorithms>

Para ampliar la información relacionada con los contenidos del Capítulo 3 (regresión), recomendamos consultar el capítulo 3 de James et al. (2013), el apartado 11.5 de Aggarwal (2015), el capítulo 2 de Mueller y Guido (2016), el capítulo 6 de Sarkar (2018), el capítulo 3 de Kirk (2017) y los capítulos 3 y 5 de Watt et al. (2016).



Capítulo 4

Clasificación

Como hemos estudiado en capítulos anteriores, la clasificación es una tarea del aprendizaje supervisado en la que se dispone de una clase de tipo categórico que se precisa estimar. En este capítulo estudiaremos dos de las técnicas más utilizadas, tanto en la literatura como en la industria, como son la regresión logística y los árboles de decisión. Comenzamos, en el primer apartado del capítulo, con el algoritmo de regresión logística.

4.1. Regresión logística

La regresión logística (LOGR, de *logistic regression* en inglés) es también conocida como regresión logit, clasificación por máxima entropía o clasificación log-lineal. A pesar de que en su nombre aparezca la palabra *regresión*, no tiene que ver con la regresión, sino que es un algoritmo para realizar clasificación.

La forma que LOGR tiene para hacer la clasificación se basa en la idea de encontrar una frontera lineal entre dos clases a partir de la transformación exponencial de una combinación lineal de los atributos. Como veremos, esta transformación exponencial permite definir una frontera suave entre las dos clases.

Esto crea una distribución de probabilidad continua entre las dos clases con una forma muy característica, denominada *función logística*, que observamos en la Figura 14:

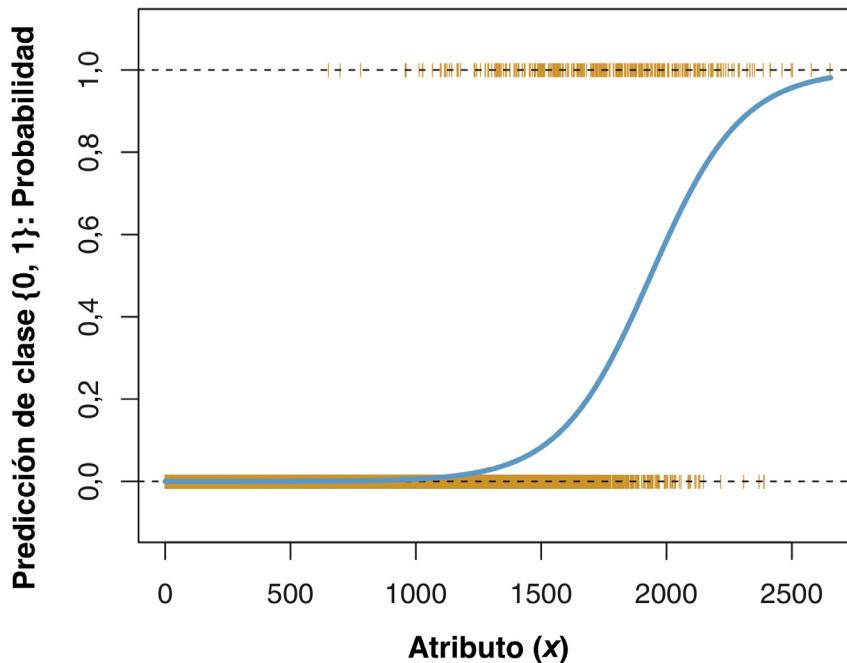


Figura 14. Distribución de los valores de predicción de LOGR según su función logística. Adaptado de *An introduction to statistical learning with applications in R* (p. 131), por G. James, D. Witten, T. Hastie y R. Tibshirani, 2013, Nueva York: Springer.

En la Figura 14 se muestra un conjunto de datos con un único atributo (x) y una clase binaria con valores $\{0, 1\}$. Los puntos del conjunto de datos están representados por pequeños segmentos verticales de color naranja, unos en la parte inferior de la figura (ejemplos de clase 0) y otros en la parte superior (ejemplos de clase 1). La predicción de LOGR se muestra en color azul de forma continua para todo el rango de valores del atributo x , con el objetivo de mostrar cómo varía su predicción en función del valor de x .



El objetivo del clasificador LOGR no es predecir si la clase de un ejemplo e_i es 0 o 1, sino predecir la probabilidad P_i de que su clase sea 1. Una vez se obtenga dicha probabilidad, es posible extraer una predicción binaria siguiendo esta simple regla: si $P_i < 0,5$, entonces la predicción de la clase se considera 1; si $P_i \leq 0,5$, entonces se considera 0.

Aunque hemos indicado 0,5 como valor umbral para la decisión de clase, es posible usar otros umbrales incluso más eficaces. De hecho, se podría encontrar de forma automática cuál es el umbral óptimo con el que se maximiza la bondad de las predicciones del clasificador.

Una de las métricas más utilizadas para medir la bondad de los clasificadores que son capaces de estimar la probabilidad de una clase binaria es la métrica AUC (de *area under curve* en inglés), que es el área bajo la curva ROC. La curva ROC (de *receiver operating characteristic* en inglés) mide la relación entre las tasas de aciertos y de falsos positivos en función del umbral de clasificación. Cuanto mayor sea el área bajo la curva ROC, mejor será el clasificador. Un clasificador perfecto tendría un valor de AUC = 1.



Enlace de interés

En los siguientes enlaces se puede encontrar más documentación y poner en práctica los conocimientos usando la librería scikit-learn y los ejemplos ya resueltos.

https://en.wikipedia.org/wiki/Receiver_operating_characteristic

http://scikit-learn.org/stable/auto_examples/model_selection/plot_roc.html

http://scikit-learn.org/stable/auto_examples/model_selection/plot_roc_crossval.html

Como hemos podido observar, el algoritmo LOGR está diseñado para clasificación binaria. Es decir, la clase solo toma dos valores (por ejemplo, {0, 1}). Sin embargo, con LOGR es posible también resolver un problema de clasificación multinomial (también llamada *multiclas*), esto es, cuando la clase posee más de dos valores. Un ejemplo de clasificación multiclas podría ser el problema de predicción de un riesgo de incendio en bajo, medio o alto.



Para resolver un problema de clasificación multinomial con $m > 2$ clases mediante un algoritmo de clasificación binaria, como LOGR, se lleva cabo la descomposición del problema original en m subproblemas de clasificación binaria de tipo One-vs-Rest (OvR, una contra el resto). En cada subproblema, las instancias son las mismas que las del conjunto original, con los mismos atributos, pero con una clase distinta, de tipo binario.

Por ejemplo, si $m = 3$, el primer subproblema binario se formaría con la clase 1 contra el resto. En este subproblema, la clase {0, 1} significaría que, si la clase es 1, la instancia es de clase 1, pero si la clase fuera 0, significaría que la clase es 2 o bien 3 (no es de clase 1). Por ejemplo, en el subproblema binario para la clase 2, la clase {0, 1} significaría que, si la clase es 1, la instancia es de clase 2, pero si la clase fuera 0, significaría que la clase es 1 o bien 3 (no es de clase 2). Y de forma análoga para el tercer subproblema.

La división del problema multiclas en subproblemas binarios de tipo OvR permite realizar una clasificación multinomial usando clasificadores binarios. Cada clasificador binario, una vez entrenado, aprendería una frontera lineal de decisión que enfrenta cada clase con las restantes, tal como se muestra en la Figura 15 para un problema con 3 clases:

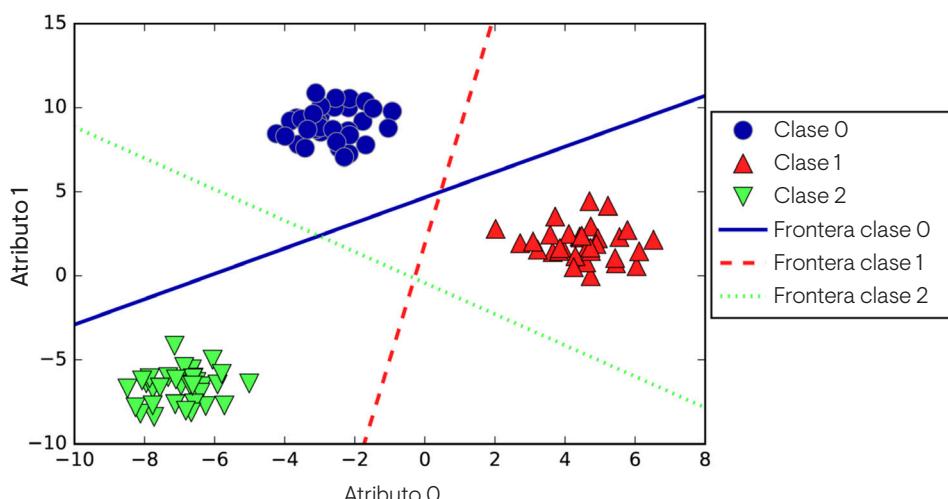


Figura 15. Tres clasificadores binarios para resolver un problema multiclas con 3 clases. Adaptado de *Introduction to Machine Learning with Python* (p. 66), por A. C. Mueller y S. Guido, 2016, Sebastopol: O'Reilly.

Una vez que se obtienen las fronteras de los clasificadores binarios, se pueden crear las regiones de decisión multinomial, tal como se muestra en la Figura 16. Para ello, se calculan las líneas bisectrices a partir de los puntos de corte de las fronteras de decisión binarias.

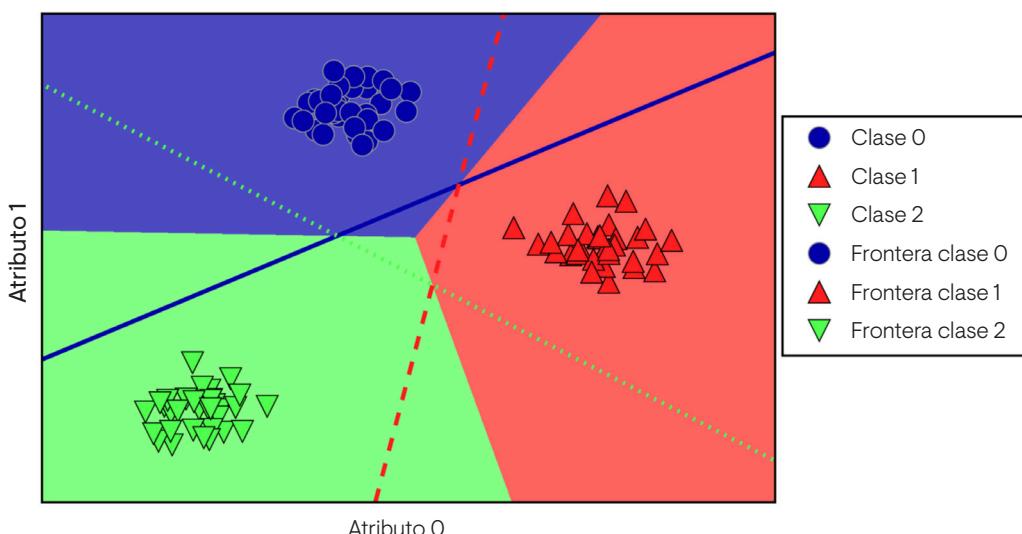


Figura 16. Regiones de clasificación multiclas con 3 clases. Adaptado de *Introduction to Machine Learning with Python* (p. 67), por A. C. Mueller y S. Guido, 2016, Sebastopol: O'Reilly.

Para comprender el funcionamiento del algoritmo LOGR, debemos definir, en primer lugar, en qué consiste su proceso de entrenamiento, para luego explicar cómo funciona la fase de predicción, al igual que hemos estudiado los algoritmos del Capítulo 3.

El proceso de entrenamiento de LOGR consiste en ajustar los coeficientes de su modelo matemático, de forma similar al modelo del algoritmo OLS de regresión lineal. El modelo de LOGR es una función matemática que permite devolver predicciones a partir de la transformación exponencial de una combinación lineal de los atributos. Su expresión es la siguiente:

$$\hat{y} = \frac{1}{1 + e^{-w_0 - w_1 \cdot x_1 - \dots - w_p \cdot x_p}}$$

El valor de predicción devuelto por la anterior expresión es un número real entre 0 y 1, es decir, $0 \leq \hat{y} \leq 1$. El entrenamiento de LOGR consiste en encontrar los valores de w_0, \dots, w_p óptimos a partir del conjunto de datos de entrenamiento. Los valores óptimos son aquellos que maximizan la siguiente función objetivo:

$$f(w_0, \dots, w_p) = \sum_{i=1}^n y_i \cdot \ln(\hat{y}_i) + (1 - y_i) \cdot \ln(1 - \hat{y}_i)$$

Nótese que en la función objetivo contabilizan solo los aciertos y, además, cuentan más los aciertos donde el modelo ha dado un valor de probabilidad lo más próximo a 0, o a 1, posible. Esto es, dado que $y_i \in \mathbb{N}\{0,1\}$ y $\hat{y}_i \in \mathbb{R}[0,1]$, el primer sumando del sumatorio de la función objetivo, $y_i \cdot \ln(\hat{y}_i)$, es distinto de 0 cuando la clase $y_i = 1$, y su valor es más alto cuanto mayor sea el valor predicho \hat{y}_i . Por otra parte, el segundo sumando del sumatorio, $(1 - y_i) \cdot \ln(1 - \hat{y}_i)$, es distinto de 0 cuando la clase $y_i = 0$, y su valor es más alto cuanto menor sea el valor predicho \hat{y}_i . Esta función objetivo también se denomina *función de verosimilitud (likelihood function en inglés)*.

La frontera de decisión que el modelo de clasificación binaria LOGR construye es el lugar geométrico de los puntos que satisfacen la condición $w_0 + w_1 \cdot x_1 + \dots + w_p \cdot x_p = 0$. En el caso de que el conjunto de datos posea un único atributo, la frontera de decisión sería un punto (\mathbb{R}). En el caso de que tuviera dos atributos, la frontera sería una recta (\mathbb{R}^2), y así sucesivamente. Nótese que, cuando $w_0 + w_1 \cdot x_1 + \dots + w_p \cdot x_p = 0$, el valor de $\hat{y} = 1/(1+e^0) = 1/2 = 0,5$, que es valor de probabilidad umbral para decidir si la predicción es de la clase 0 o de la clase 1.

En la Tabla 7 se muestran los valores extremos posibles del modelo LOGR. Como se puede apreciar, los valores exactos 0 y 1 realmente no se alcanzan en el modelo LOGR, pues tanto w como x son valores finitos.

Tabla 7
Valores extremos del modelo LOGR

	Mínimo	Mitad	Máximo
\hat{y}	0	0,5	1
$w_0 + w_1 \cdot x_1 + \dots + w_p \cdot x_p = 0$	$-\infty$	0	$+\infty$

Para encontrar los valores óptimos de los coeficientes del modelo, se pueden emplear diferentes estrategias de búsqueda. La clase **LogisticRegression** de la librería scikit-learn implementa el algoritmo LOGR y utiliza, entre otros, el algoritmo coordinate descent (CD) para optimizar los coeficientes del modelo.

Del mismo modo que comentamos en regresión lineal, es posible evitar el sobreajuste del modelo LOGR a los datos de entrenamiento mediante los esquemas de regularización de tipo L1 (**Lasso**) y L2 (**Ridge**). Los métodos de regularización consisten en añadir un término a la función objetivo de forma que los coeficientes del modelo reduzcan sus valores y se aproximen a 0.

En el Programa 15 mostramos un ejemplo de aplicación del algoritmo LOGR al conjunto de datos “iris”. Se ha llevado a cabo una validación cruzada con 10 bolsas y se han obtenido las métricas de clasificación estudiadas: exactitud (ACC), precisión (PREC), sensibilidad (RECALL) y F1.

```
</>
import sklearn.metrics as metrics
from sklearn.datasets import load_iris
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_predict
from sklearn.model_selection import KFold
from evaluacion_funciones import *

# Carga de datos.
datos = load_iris()
datos.data = datos.data[:,2:] # Tomamos solo 2 atributos.
```

```

# Construcción del algoritmo de aprendizaje.
seed = 1
algoritmos = {'LOGR': LogisticRegression(solver='sag',
                                         max_iter=1000, random_state=seed,
                                         multi_class='ovr')}

# Métricas de evaluación.
metricas = {
    'ACC': metrics.accuracy_score,
    'PREC': lambda y_true, y_pred:
        metrics.precision_score(y_true, y_pred,
                               average='micro'),
    'RECALL': lambda y_true, y_pred:
        metrics.recall_score(y_true, y_pred,
                             average='micro'),
    'F1': lambda y_true, y_pred:
        metrics.f1_score(y_true, y_pred, average='micro')}

# Validación y obtención de las predicciones del modelo.
seed = 1
y_pred = {}
for nombre, alg in algoritmos.items():
    y_pred[nombre] = cross_val_predict(alg, datos.data, datos.target,
                                       cv = KFold(n_splits=10, random_state=seed))

# Evaluación y presentación de resultados.
for nombre, alg in algoritmos.items():
    eval = evaluacion(datos.target, y_pred[nombre], metricas)
    modelo_completo = alg.fit(datos.data, datos.target)
    mapa_modelo_clasif_2d(datos.data, datos.target,
                           modelo_completo, eval, nombre)
    print("Matriz de confusión (%s):\n%s" % (nombre,
                                                metrics.confusion_matrix(datos.target, y_pred[nombre])))
    print("Tabla de métricas (%s):\n%s" % (nombre,
                                              metrics.classification_report(datos.target,
                                                y_pred[nombre], digits=3)))

```

Programa 15. Clasificación con LOGR y visualización de un mapa con las regiones del modelo.

Matriz de confusión (LOGR):

```

[[50  0  0]
 [ 1 40  9]
 [ 0  4 46]]

```

Tabla de métricas (LOGR):

	precision	recall	f1-score	support
0	0.980	1.000	0.990	50
1	0.909	0.800	0.851	50
2	0.836	0.920	0.876	50
avg / total	0.909	0.907	0.906	150

Salida 15. Salida del Programa 15.

Con el objetivo de visualizar en dos dimensiones las regiones de clasificación del modelo LOGR, hemos tomado únicamente los dos atributos del pétalo de la flor de “iris” (atributos tercero y cuarto). Se crea, a continuación, un objeto de tipo **LogisticRegression**, cuyo algoritmo de búsqueda ha sido configurado con **solver='sag'**. El algoritmo ‘**sag**’ utiliza la optimización stochastic average gradient descent y tiene la ventaja de ser muy eficiente en tiempo de ejecución.

La clasificación multinomial con 3 clases (recordemos que “iris” posee 3 clases) se lleva a cabo mediante tres clasificadores internos binarios LOGR de tipo OvR, lo cual indicamos con el argumento **multi_class='ovr'** del constructor de la clase **LogisticRegression**. Establecemos el número de iteraciones de LOGR a 1000, que es número suficiente para garantizar la convergencia de la solución.

Gracias a la función **mapa_modelo_clasif_2d()** del módulo “evaluacion_funciones.py”, suministrado en la asignatura, podemos crear una gráfica como la que se muestra en la Figura 17:

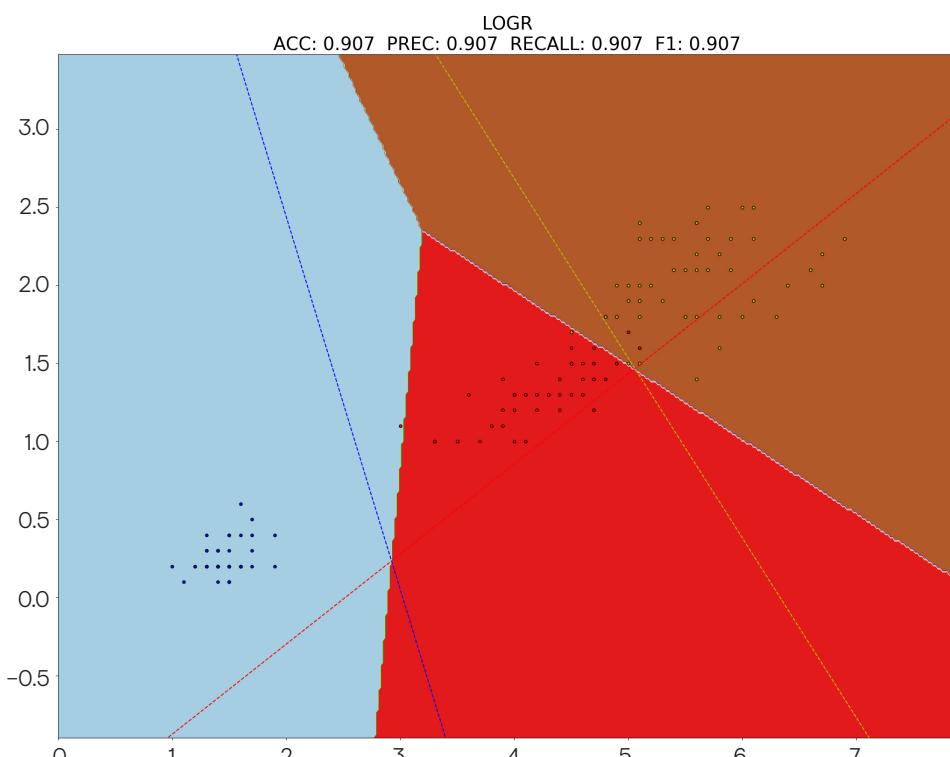


Figura 17. Gráfica del Programa 15: Mapa de regiones de clasificación del modelo LOGR.

En esta gráfica se muestran las regiones que el clasificador ha aprendido en su modelo. Cualquier punto de test futuro será clasificado según la región en la que se encuentre en dicho mapa, y la probabilidad asignada por el clasificador a la clase predicha será mayor conforme dicho punto se aleje de las fronteras de decisión (marcadas con líneas coloreadas discontinuas en la Figura 17).

Es importante apreciar en la Figura 17 que las fronteras de decisión marcadas con líneas discontinuas fueron obtenidas para cada submodelo binario LOGR entrenado con clases de tipo OvR. Por ejemplo, la frontera de decisión de la clase de color azul (línea discontinua azul) fue ajustada por el submodelo binario LOGR especializado únicamente en predecir si la clase es azul o no es azul.

Dicho submodelo se generó a partir del conjunto de datos “iris”, cuya clase $\{0, 1, 2\}$ original fue sustituida por $\{0, 1\}$, donde 1 representa la clase azul y 0 las clases roja y amarilla (el resto de clases menos la azul). De forma análoga se construyeron las otras dos fronteras de decisión.

A tenor de los resultados numéricos obtenidos, mostrados en la Salida 14, observamos una mejora importante con respecto al clasificador básico de prueba **DummyClassifier** (véase Salida 10). Era de esperar: el clasificador **DummyClassifier** daba siempre como predicción la clase más frecuente del conjunto de entrenamiento. En concreto, si observamos la matriz de confusión, vemos que LOGR ha cometido tan solo $1 + 4 + 9 = 14$ errores de clasificación.

El algoritmo LOGR posee una serie de ventajas e inconvenientes que es importante señalar. Las ventajas fundamentales que podríamos destacar son:

- La simplicidad del modelo. El modelo de LOGR es muy parecido al de OLS y, por tanto, comparte su simplicidad. Se trata de un modelo basado en una ecuación matemática simple, cuyo ajuste y evaluación son sencillos.
- La fácil interpretabilidad de las predicciones. Al igual que OLS, el modelo es sencillo de interpretar observando la importancia de cada atributo según la magnitud de su coeficiente asociado. A mayor coeficiente, en valor absoluto, mayor importancia tiene el atributo. Al igual que OLS, por el signo del coeficiente sabemos si la relación de cada atributo con la clase es directa (signo positivo) o inversa (signo negativo).
- El tiempo de ejecución del entrenamiento muy razonable. Al igual que OLS, el entrenamiento consiste en el ajuste de los coeficientes de la combinación lineal de los atributos mediante un proceso iterativo de búsqueda y optimización. Los tiempos de entrenamiento de LOGR son similares a los de OLS, aunque algo superiores, debido al cálculo de exponentes y logaritmos en la función objetivo.
- El tiempo de ejecución de la predicción casi instantáneo. Una vez entrenado el modelo LOGR, realizar las predicciones es inmediato. Tan solo hay que evaluar la función matemática del modelo usando los atributos de los ejemplos de test.

Por otra parte, el algoritmo LOGR posee ciertos inconvenientes:

- La independencia entre los atributos. Los modelos de regresión lineal, incluida la regresión logística, asumen que los atributos no poseen una correlación significativa entre ellos, es decir, se requiere que sean variables estadísticamente independientes.

- La excesiva simplicidad del modelo. El modelo LOGR solo es capaz de crear una única frontera de decisión lineal entre dos clases. En muchos problemas, esto puede ser insuficiente, pues las regiones de decisión son mucho más complejas y se requieren varias fronteras de decisión, incluso no lineales.

4.2. Árboles de decisión

El último tipo de técnicas de aprendizaje supervisado que estudiaremos en la asignatura son los árboles de decisión (*decision trees* en inglés). Un árbol de decisión es un tipo de modelo de conocimiento que puede generarse a partir de datos y que puede usarse tanto para clasificación como regresión.

En este último caso, se denomina árbol de regresión, aunque su construcción es análoga a la del árbol de decisión.



La idea conceptual de los árboles de decisión es dividir el conjunto de datos de forma jerárquica en trozos cada vez menores hasta llegar a trozos donde los ejemplos son todos, o prácticamente todos, de la misma clase. La forma de dividir los datos en trozos se hace a partir de los valores de los atributos.

En concreto, cada vez que se divide un conjunto de datos en dos o más trozos (y, así, de forma recursiva), suele ser un único atributo el que se usa. Dependiendo del valor del atributo, se aplican filtros a los datos para generar trozos de datos menores cuyas filas han sido filtradas para que tengan únicamente ciertos valores del atributo en cuestión.



Ejemplo

Para comprender en qué consisten los árboles de decisión y cómo se pueden utilizar para hacer predicciones, veamos un ejemplo de árbol de decisión construido a partir de un conjunto de datos sencillo, que llamaremos “jugar-tenis”, el cual se muestra en la Tabla 8. Como se puede apreciar, es un conjunto de datos de 14 instancias con 5 atributos (**Day**, **Outlook**, **Temp**, **Humidity** y **Wind**) y una clase (**Play**). Imaginemos que tenemos un proyecto de minería de datos con el conjunto “jugar-tenis”, cuyo objetivo es determinar si en un día determinado se jugará o no al tenis en función de las condiciones meteorológicas.

Dado que es un problema de clasificación (la clase es categórica), aplicamos un algoritmo de árboles de decisión, que nos produce automáticamente el modelo en forma de árbol. Este modelo nos permitirá realizar predicciones de jugar o no jugar al tenis en función de las condiciones meteorológicas que tenga la instancia de test a predecir.

>>>

>>>

Tabla 8

Conjunto de datos “jugar-tenis”

Day	Outlook	Temp	Humidity	Wind	Play
D1	Sunny	Hot	High	Weak	No
D2	Sunny	Hot	High	Strong	No
D3	Overcast	Hot	High	Weak	Yes
D4	Rain	Mild	High	Weak	Yes
D5	Rain	Cool	Normal	Weak	Yes
D6	Rain	Cool	Normal	Strong	No
D7	Overcast	Cool	Normal	Strong	Yes
D8	Sunny	Mild	High	Weak	No
D9	Sunny	Cool	Normal	Weak	Yes
D10	Rain	Mild	Normal	Weak	Yes
D11	Sunny	Mild	Normal	Strong	Yes
D12	Overcast	Mild	High	Strong	Yes
D13	Overcast	Hot	Normal	Weak	Yes
D14	Rain	Mild	High	Strong	No



Enlace de interés

En el siguiente enlace puede consultar el modelo de árbol generado a partir del conjunto de datos “jugar-tenis”.

<https://nullpointerexception1.wordpress.com/2017/12/16/a-tutorial-to-understand-decision-tree-id3-learning-algorithm/>

Supongamos, por ejemplo, que nos interesa saber si se jugará al tenis en un día con el cielo (*outlook* en inglés) soleado (*sunny* en inglés), humedad normal, temperatura alta (*hot* en inglés) y viento (*wind* en inglés) fuerte (*strong* en inglés). Para saberlo, vamos a utilizar el árbol que se puede encontrar en el último enlace de interés, tal como describimos a continuación:

En primer lugar, debemos definir un árbol como una estructura de datos constituida por nodos y aristas que unen nodos, de manera que no se forman ciclos y en la que existe un primer nodo destacado, llamado *raíz del árbol*. La estructura de árbol es ampliamente utilizada en informática para representar una gran variedad de objetos y conceptos.

Comenzamos leyendo el árbol que se puede encontrar en el último enlace de interés por la parte superior, en la que nos encontramos su nodo raíz. Cada nodo de un árbol de decisión contiene el nombre de un atributo. El nodo raíz contiene el atributo “Outlook”, el cual nos informa que el primer atributo que debemos observar de la instancia de test, para poder clasificarla, es el atributo “Outlook”.

>>>

>>>

Observamos que, en el día especificado por nuestra instancia de test, el cielo es soleado. Por tanto, seguiremos leyendo del árbol tomando como camino la arista etiquetada con “Sunny” que parte del nodo raíz. Esta arista nos lleva a un nuevo nodo que contiene el atributo “Humidity”. Por ello, observamos el valor que tiene este atributo en nuestra instancia de test: “Normal” (humedad normal).

De forma similar a como hemos procedido antes, tomamos el camino definido por la arista etiquetada con “Normal” que parte del nodo “Humidity”. Esta nos lleva a un último nodo especial que no contiene más aristas a partir de él. Este tipo de nodos se llama *nodo hoja* y nos da una predicción directa para nuestra instancia de test (no hay que observar más atributos). L

a predicción que nos da es “Yes”, lo cual nos indica que el modelo entiende que se jugará al tenis.

Nótese que el hecho de que el día fuese de temperatura alta y viento fuerte ha sido irrelevante para el modelo. Si observamos los datos, apreciamos que, en todos los días en los que el cielo estaba soleado y había una humedad normal, se ha jugado al tenis (días D9 y D11), con independencia del resto de factores.

Pongamos otro ejemplo: supongamos que el día está nublado (“Outlook” = “Overcast”), hay una temperatura alta, humedad alta y viento fuerte. Comenzamos a leer nuevamente nuestro árbol desde la raíz. El nodo raíz nos indica observar el atributo “Outlook” y este tiene “Overcast” como valor en nuestro día. Por tanto, tomamos la arista etiquetada con “Overcast”. En este caso, dicha arista nos lleva a un nodo hoja, el cual nos da una predicción directamente: “Yes” (sí se juega al tenis).

Tal como acabamos de ver, el hecho de que el día fuera de temperatura alta, humedad alta y viento fuerte ha sido completamente irrelevante para el modelo. Si observamos el conjunto de datos, podemos apreciar que todos los días nublados (“Outlook” = “Overcast”) se ha jugado al tenis (días D3, D7, D12 y D13), con independencia del resto de condiciones meteorológicas.

Si hiciéramos predicciones con todos los días del conjunto de datos usando el modelo de árbol visto anteriormente, obtendríamos una exactitud, precisión, sensibilidad y F1 del 100 %. Sin embargo, como ya sabemos, existe el riesgo de tener un modelo sobreajustado. Para validar adecuadamente el modelo, tendríamos que aplicar alguno de los métodos de validación que hemos estudiado en el Capítulo 2.



Para construir un árbol de decisión, el algoritmo de construcción debe asignar a cada nodo del árbol el atributo más apropiado en cada caso. El atributo más apropiado debería ser aquel que, mediante sus valores posibles, mejor distingue las instancias de una clase o varias clases con respecto a las demás, esto es, el atributo que mejor discrimine las clases de los ejemplos.

Existen varios algoritmos en la literatura científica que permiten crear árboles de decisión a partir de conjuntos de datos. El primer algoritmo clásico que se encuentra en la literatura se denomina ID3, propuesto por Ross Quinlan, que data de 1986. Este algoritmo es capaz de generar árboles n -arios, es decir, árboles que pueden tener más de dos aristas por cada nodo.

Para cada nodo, ID3 selecciona el mejor atributo en función de la métrica de ganancia de información (*information gain* en inglés). ID3 tiene la ventaja de que, una vez construido el árbol, lo refina eliminando los nodos que pudieran causar sobreajuste del modelo. Este procedimiento se llama *poda del árbol* (*tree pruning* en inglés).

ID3 tiene el inconveniente de que solo puede utilizarse con conjuntos de datos en los que todos los atributos son categóricos. No admite atributos numéricos, pues no existe un conjunto finito de valores posibles a partir del cual crear aristas en los nodos.

El siguiente algoritmo clásico de interés en la literatura se denomina C4.5 y es el sucesor de ID3, diseñado por el mismo autor de ID3. C4.5 admite atributos numéricos en el conjunto de datos. Para crear las aristas en un nodo que contiene un atributo numérico, C4.5 crea automáticamente una serie de puntos de corte en el rango de valores numéricos del atributo. C4.5 divide entonces el conjunto de datos en aquellos trozos cuyos valores en el atributo se encuentran en cada intervalo resultante de la división por dichos puntos de corte.

Una vez construido el árbol, C4.5 extrae de él todas las reglas de decisión posibles del tipo “si..., entonces...”, esto es, todos los caminos posibles que llevan desde el nodo raíz hasta cada uno de los nodos hoja del árbol. Cada camino expresa una serie de condiciones que deben darse para obtener una predicción concreta.

Continuando con el ejemplo del conjunto de datos “jugar-tenis”, una regla de tipo “si..., entonces...” del árbol puede ser la que se tomó para predecir la primera de las dos instancias de test que pusimos como ejemplo. En concreto, la regla sería: si “Outlook” = “Sunny” y “Humidity” = “Normal”, entonces “Yes” (se juega al tenis).

C4.5 extrae todas las reglas posibles del árbol construido y las ordena por su eficacia, desde la más eficaz a la menor, con el objetivo de, finalmente, podar aquellas reglas que empeoran la eficacia global del árbol de decisión.

Otro algoritmo de aprendizaje de árboles de decisión muy similar a C4.5 es el denominado CART (*classification and regression trees* en inglés). Como su propio nombre indica, permite realizar tanto clasificación como regresión usando árboles como modelo. A diferencia de ID3 y C4.5, CART produce únicamente árboles binarios; esto es, desde cada nodo parten a lo sumo dos aristas. La librería scikit-learn implementa una versión optimizada del algoritmo CART, pero no admite atributos categóricos, tan solo numéricos.



Enlaces de interés

En el siguiente artículo se analizan las métricas que se utilizan para seleccionar los atributos más prometedores en cada nodo de un árbol de decisión, tales como Gini o la entropía, incluyendo ejemplos numéricos para su comprensión. Además, se avanza en dicho artículo desde los árboles de decisión hacia los algoritmos de grupos (*ensemble* en inglés) de árboles, tales como los conocidos algoritmos Random Forest, Gradient Boosting Machines o XGBoost, que permiten combinar varios árboles de decisión entre sí en un único modelo de predicción.

<https://www.analyticsvidhya.com/blog/2016/04/complete-tutorial-tree-based-modeling-scratch-in-python/>

Si desea ver ilustrado el proceso de construcción de un árbol de decisión de una forma gráfica, muy visual e interactiva, puede consultar la siguiente web, dedicada al aprendizaje visual de algoritmos de aprendizaje automático:

<http://www.r2d3.us/una-introduccion-visual-al-machine-learning-1/>

En el Programa 16, se muestra un ejemplo de aplicación del algoritmo CART de la librería scikit-learn para la construcción de árboles de decisión mediante validación cruzada con 10 bolsas. El algoritmo se ha aplicado a tres conjuntos de datos distintos: “iris”, “wine” y “breast-cancer”.

En la Tabla 9 y en la Tabla 10 mostramos las características de los conjuntos de datos “wine” y “breast-cancer”, respectivamente.

Tabla 9

Características del conjunto de datos “wine”

Número de instancias	178
Número de atributos	13
Descripción de atributos	Por motivos de extensión no se ha incluido aquí, pero puede consultarse en el siguiente enlace: https://archive.ics.uci.edu/ml/datasets/wine
Tipo de datos de la clase	Categórico: {class-0, class-1, class-2}
Descripción de la clase	Nombre: class. Descripción: tipo de vino.
Valores ausentes	0

Tabla 10

Características del conjunto de datos “breast-cancer”

Número de instancias	569
Número de atributos	30
Descripción de atributos	Por motivos de extensión no se ha incluido aquí, pero puede consultarse en el siguiente enlace: https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic)
Tipo de datos de la clase	Categórico: {malignant, benign}
Descripción de la clase	Nombre: class. Descripción: tipo de tumor.
Valores ausentes	0

En el Programa 16 hemos aumentado nuevamente el nivel de automatización, de manera que tanto los conjuntos de datos, como los algoritmos, métricas e, incluso, experimentos se encuentran definidos al comienzo del programa en forma de diccionarios.

```
</>

from sklearn import tree
from sklearn import datasets
import sklearn.metrics as metrics
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_predict
from evaluacion_funciones import *

seed = 1

#####
# PASO 1. CARGA DE CONJUNTOS DE DATOS.

datos = {
    'IRIS': datasets.load_iris(),
    'WINE': datasets.load_wine(),
    'BREAST-CANCER': datasets.load_breast_cancer()
}

#####
# PASO 2. CONSTRUCCIÓN DE ALGORITMOS DE APRENDIZAJE.

algoritmos = {
    'DT': tree.DecisionTreeClassifier()
}

#####
# PASO 3. CONSTRUCCIÓN DE EXPERIMENTOS.

experimentos = {
    'DT -> IRIS': (algoritmos['DT'], datos['IRIS']),
    'DT -> WINE': (algoritmos['DT'], datos['WINE']),
    'DT -> BREAST-CANCER': (algoritmos['DT'],
                             datos['BREAST-CANCER'])
}

#####
# PASO 4. DEFINICIÓN DE MÉTRICAS DE EVALUACIÓN.

metricas = {
    'ACC':     metrics.accuracy_score,
```

```

'PREC': lambda y_true, y_pred:
    metrics.precision_score(y_true, y_pred,
                            average='micro'),
'RECALL': lambda y_true, y_pred:
    metrics.recall_score(y_true, y_pred,
                          average='micro'),
'F1': lambda y_true, y_pred:
    metrics.f1_score(y_true, y_pred, average='micro')
}

#####
# PASO 5. VALIDACIÓN Y OBTENCIÓN DE LAS PREDICCIONES.

y_pred = {}
for nombre, exp in experimentos.items():
    y_pred[nombre] = cross_val_predict(exp[0], exp[1].data,
                                       exp[1].target, cv=KFold(n_splits=10, random_state=seed))

#####
# PASO 6. EVALUACIÓN Y PRESENTACIÓN DE RESULTADOS.

for nombre, exp in experimentos.items():

    # 6.1. Generación de métricas de evaluación
    #       (para generar tablas numéricas).
    eval = evaluacion(exp[1].target, y_pred[nombre], metricas)
    print("Matriz de confusión (%s):\n%s" % (nombre,
                                                metrics.confusion_matrix(exp[1].target, y_pred[nombre])))
    print("Tabla de métricas (%s):\n%s" % (nombre,
                                              metrics.classification_report(exp[1].target,
                                                                             y_pred[nombre], digits=3)))
    # 6.2. Creación de modelo de 2 atributos
    #       (para visualización 2D).
    modelo_completo_2d = exp[0].fit(exp[1].data[:, :2],
                                      exp[1].target)
    mapa_modelo_clasif_2d(exp[1].data[:, :2], exp[1].target,
                           modelo_completo_2d, eval, nombre)

    # 6.3. Creación de modelo con todos los datos
    #       (para visualización árbol).
    if (nombre.startswith('DT')):
        modelo_completo = exp[0].fit(exp[1].data,
                                      exp[1].target)
        mostrar_modelo_arbol(modelo_completo, exp[1],
                              nombre_archivo = nombre.replace('>', ''))
```

Programa 16. Clasificación de varios conjuntos de datos con CART. Visualización de mapas de regiones de clasificación y de los árboles de decisión generados.

En concreto, en el Programa 16 se han definido tres experimentos, en los que se aplica el algoritmo CART (implementado en la clase `DecisionTreeClassifier` del paquete `tree` de la librería scikit-learn) a los tres conjuntos de datos antes mencionados: “iris”, “wine” y “breast-cancer”. Una vez definidos los experimentos, en el paso 5 del Programa 16 se realiza la validación cruzada mediante la función `cross_val_predict()` de scikit-learn, tal como hemos procedido en anteriores programas.

En el paso 6.1, se evalúan, para cada experimento, las métricas de clasificación a partir de las predicciones obtenidas en el paso anterior (función `evaluacion()` del módulo “evaluación_funciones.py” suministrado en la asignatura). Se muestra la matriz de confusión y la tabla de métricas del experimento, gracias a las funciones `confusion_matrix()` y `classification_report()` de scikit-learn.

En el paso 6.2 se crea una figura que contiene el mapa de regiones de clasificación del modelo generado en el experimento (para todos los experimentos). Dado que el mapa de regiones es una figura en dos dimensiones, para poder mostrarlo se entrena un modelo únicamente con los dos primeros atributos de todo el conjunto de datos del experimento (llamado `modelo_completo_2d`).

Finalmente, en el paso 6.3, se crea una figura que contiene una representación visual del árbol de decisión generado a partir del conjunto de datos completo del experimento. Este tipo de figura solo aplica a modelos de tipo árbol de decisión, no a otros algoritmos como la regresión logística, por ejemplo. Por ello, se ha usado la clave `DT` (decision tree) en el programa para producir esta figura solo si procede.

Nótese que, tanto para el paso 6.2 como para el 6.3, se generan modelos completos con todo el conjunto de datos. Esto se debe a que la validación cruzada, necesaria para producir las métricas de evaluación, produjeron internamente 10 modelos (uno por cada bolsa de validación). Por motivos de simplicidad, mostramos únicamente un árbol con todos los datos de cada conjunto (“iris”, “wine” y “breast-cancer”), en lugar de $3 \cdot 10 = 30$ árboles de decisión.

Los resultados producidos por el Programa 16 son los siguientes:

- Mapa de regiones de clasificación del modelo generado en el experimento con el conjunto de datos “iris” (Figura 18).
- Mapa de regiones de clasificación del modelo generado en el experimento con el conjunto de datos “wine” (Figura 19).
- Mapa de regiones de clasificación del modelo generado en el experimento con el conjunto de datos “breast-cancer” (Figura 20).
- Árbol de decisión generado por CART para el conjunto “iris” (Figura 21).
- Árbol de decisión generado por CART para el conjunto “wine” (Figura 22).
- Árbol de decisión generado por CART para el conjunto “breast-cancer” (Figura 23).
- Matrices de confusión y tablas de métricas de los 3 experimentos: Salida 15.

Observando las métricas de evaluación arrojadas por el algoritmo CART para el conjunto de datos “iris”, notamos cierta mejora con respecto a LOGR ($F1 = 0,933$ de CART frente a $F1 = 0,907$ de LOGR).

En los mapas de regiones de clasificación de CART apreciamos que existen múltiples fronteras de decisión, a diferencia de en los mapas que se obtuvieron con LOGR, algoritmo el cual solo es capaz de producir una única frontera de decisión por cada clase. Además, podemos apreciar que las fronteras de decisión generadas por CART son todas ortogonales con respecto a los ejes definidos por los atributos. Esto se debe a los puntos de corte introducidos por el algoritmo en los atributos numéricos.

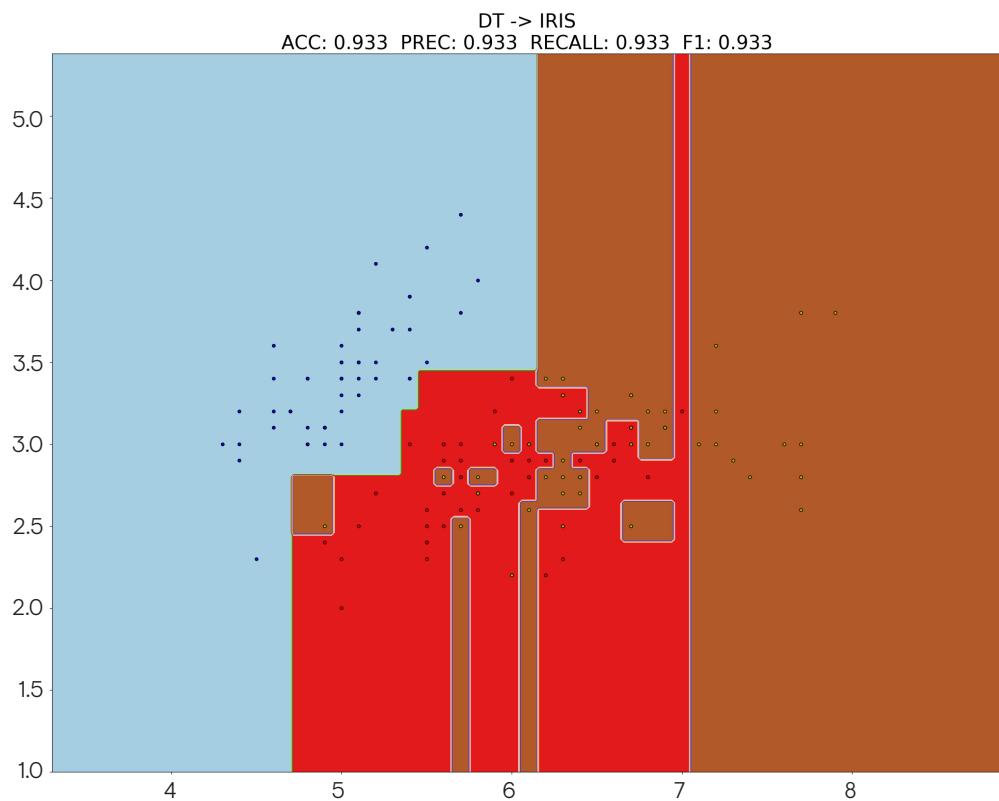


Figura 18. Mapa de regiones de clasificación del modelo CART para el conjunto “iris” producido por el Programa 16.

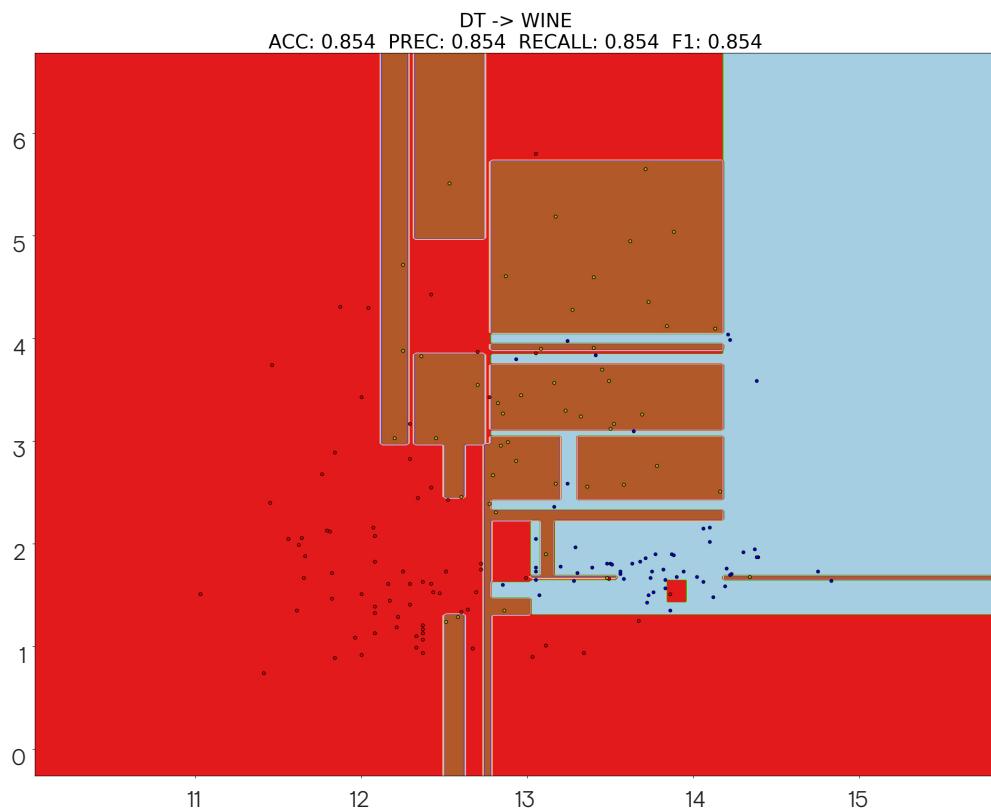


Figura 19. Mapa de regiones de clasificación del modelo CART para el conjunto “wine” producido por el Programa 16.

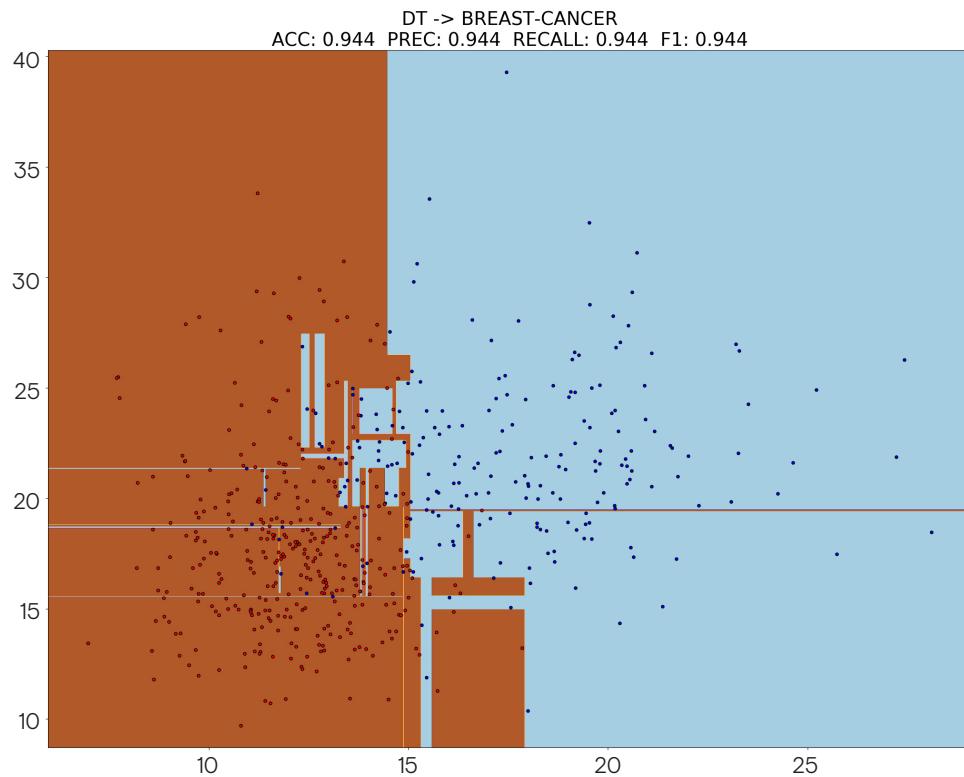


Figura 20. Mapa de regiones de clasificación del modelo CART para el conjunto “breast-cancer” producido por el Programa 16.

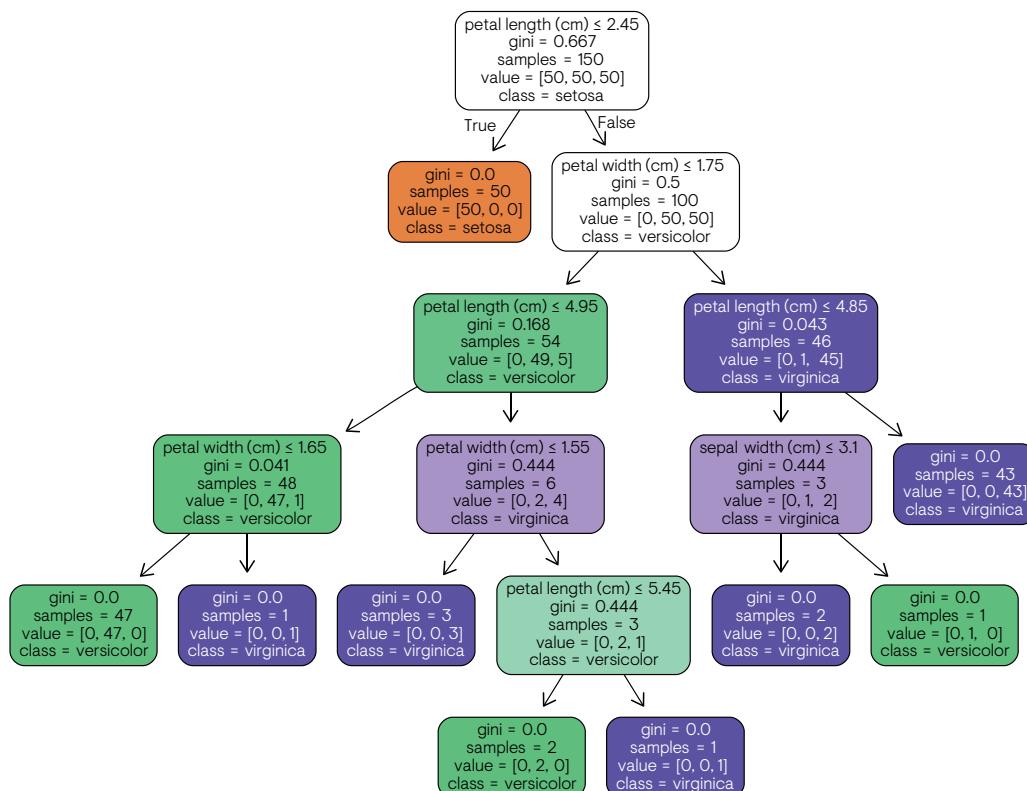


Figura 21. Árbol de decisión generado por CART con el conjunto “iris”.

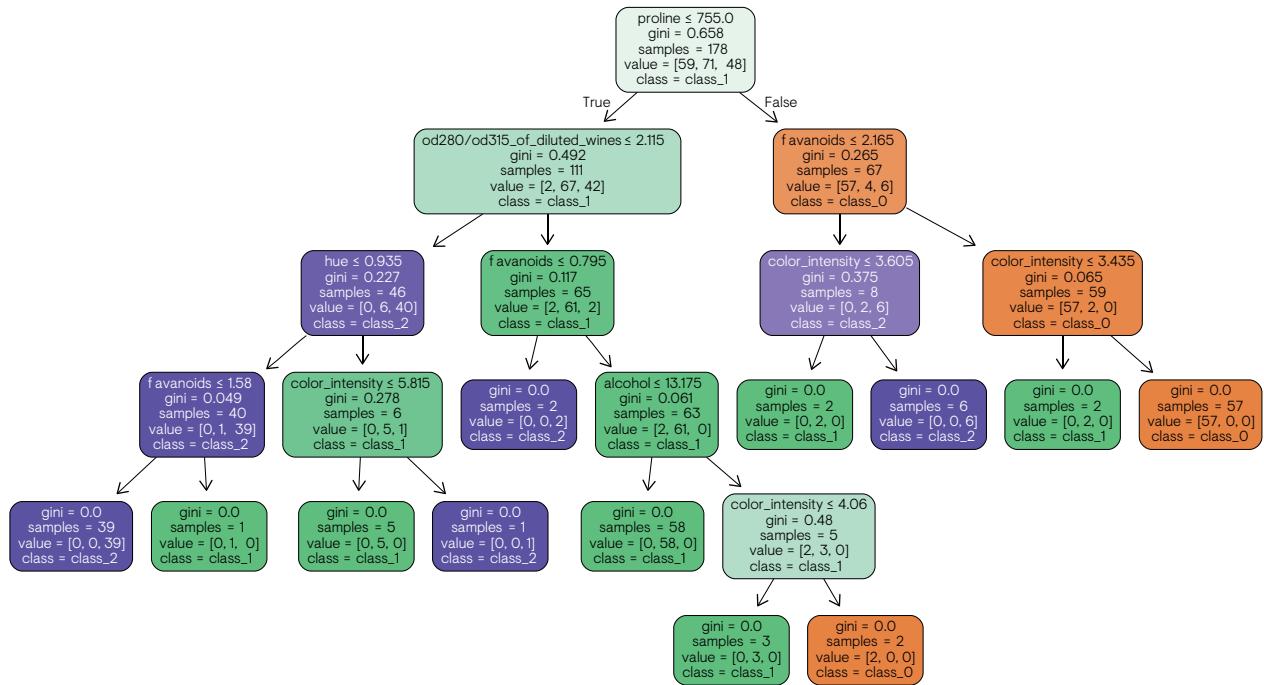


Figura 22. Árbol de decisión generado por CART con el conjunto “wine”.

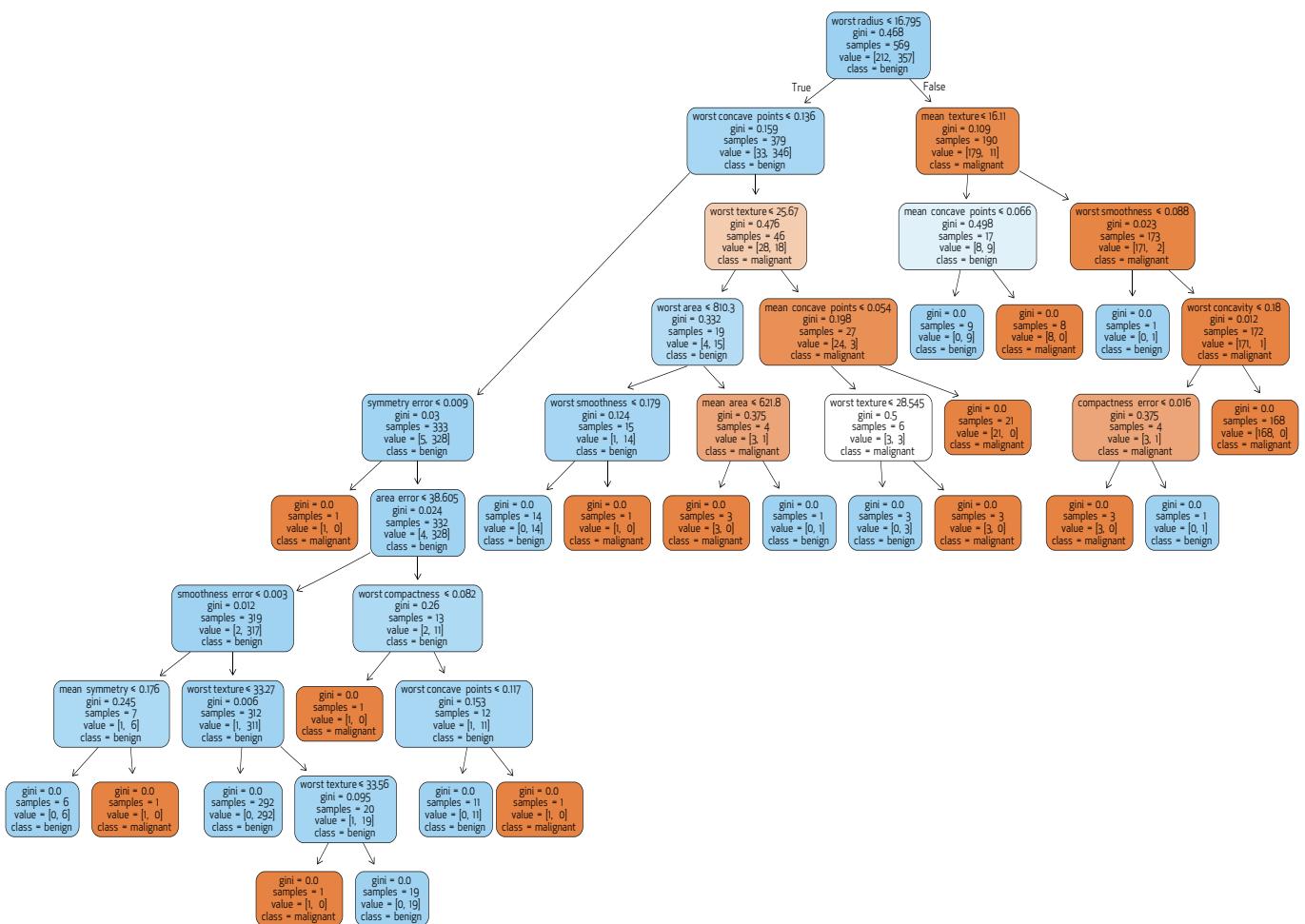


Figura 23. Árbol de decisión generado por CART con el conjunto “breast-cancer”.



Matriz de confusión (DT -> IRIS):

```
[[50  0  0]
 [ 0 46  4]
 [ 0  6 44]]
```

Tabla de métricas (DT -> IRIS):

	precision	recall	f1-score	support
0	1.000	1.000	1.000	50
1	0.885	0.920	0.902	50
2	0.917	0.880	0.898	50
avg / total	0.934	0.933	0.933	150

Matriz de confusión (DT -> WINE):

```
[[54  5  0]
 [ 6 59  6]
 [ 2  6 40]]
```

Tabla de métricas (DT -> WINE):

	precision	recall	f1-score	support
0	0.871	0.915	0.893	59
1	0.843	0.831	0.837	71
2	0.870	0.833	0.851	48
avg / total	0.859	0.860	0.859	178

Matriz de confusión (DT -> BREAST-CANCER):

```
[[195 17]
 [ 22 335]]
```

Tabla de métricas (DT -> BREAST-CANCER):

	precision	recall	f1-score	support
0	0.899	0.920	0.909	212
1	0.952	0.938	0.945	357
avg / total	0.932	0.931	0.932	569

Salida 16. Salida del Programa 16: Matrices de confusión y tablas de métricas por clase.

El algoritmo CART posee una serie de ventajas e inconvenientes que es importante señalar. Las ventajas fundamentales que podríamos destacar son:

- La simplicidad del modelo. Los modelos de árbol de decisión son fácilmente legibles, pues es posible observar de forma jerárquica cómo intervienen los atributos en la toma de decisión de cara a producir una predicción. Además, los atributos situados en la parte superior del árbol poseen mayor relevancia para el modelo que los situados en la parte inferior, lo cual ayuda a entender cuáles son las variables críticas del problema.

- La fácil interpretabilidad de las predicciones. Las predicciones llevadas a cabo por un modelo de árbol de decisión pueden ser fácilmente explicadas trazando el camino que lleva desde el nodo raíz de árbol hasta el nodo hoja que produjo la predicción. De esta forma, es posible conocer la causa que produjo la predicción, pues basta con extraer la regla “si..., entonces...” a partir del camino antes mencionado.
- El tiempo de ejecución del entrenamiento razonable. El entrenamiento de un árbol de decisión consiste en evaluar los atributos más prometedores para cada nodo, así como los puntos de corte óptimos para los valores de aquellos que son numéricos. En la mayoría de los casos, el algoritmo CART ofrece unos tiempos muy razonables de entrenamiento, incluso para conjuntos grandes de datos.
- El tiempo de ejecución de la predicción muy rápido. Una vez entrenado un modelo de árbol de decisión, realizar las predicciones es muy rápido, pues tan solo es necesario seguir el camino desde el nodo raíz hacia el nodo hoja al que se llegue tras cumplir las condiciones que satisfaga la instancia de test. Este tiempo es logarítmico con respecto al número de instancias de entrenamiento, con lo cual es muy eficiente.
- La gran flexibilidad a las características de los datos. Los algoritmos de árboles de decisión son robustos frente a outliers, valores ausentes y atributos en diferentes escalas de valores. La eficacia del modelo suele verse poco afectada por estos factores. Esto hace que la preparación de datos pueda ser mucho más liviana cuando usamos árboles de decisión.

Por otra parte, los modelos de árbol de decisión poseen ciertos inconvenientes:

- El riesgo de sobreajuste. Los algoritmos de árboles de decisión pueden llegar a generar árboles muy grandes, con muchísimos nodos y aristas, que se especializan demasiado en los datos del conjunto de entrenamiento y no generalizan adecuadamente. Este riesgo de sobreajuste aumenta conforme el conjunto de datos posee mayor cantidad de atributos y menor número de instancias. No obstante, existen mecanismos para tratar de evitar el sobreajuste, tales como podar el árbol (eliminar nodos), establecer un umbral mínimo de ejemplos cubiertos por un nodo hoja o limitar la profundidad máxima (número de niveles) del árbol.
- La sensibilidad al desbalanceo de clases. Al igual que la mayoría de los clasificadores no especializados en desbalanceo de clases, la eficacia del modelo queda muy mermada por el hecho de que en el conjunto de datos haya muchas más instancias de una clase que de otras, esto es, si la proporción de ejemplos de cada clase no está equilibrada.

Para ampliar la información relacionada con los contenidos del Capítulo 4 (clasificación), recomendamos consultar los capítulos 4 y 8 de James et al. (2013), el capítulo 10 de Aggarwal (2015), el capítulo 2 de Mueller y Guido (2016), el capítulo 9 de Sarkar (2018), el capítulo 5 de Kirk (2017) y los capítulos 4 y 6 de Watt et al. (2016).



Glosario

Ajuste de parámetros

Proceso de elección óptima de los parámetros de algoritmos de aprendizaje automático, de manera que se maximice la bondad de los resultados predictivos.

Algoritmo de búsqueda

Un algoritmo de búsqueda trata de encontrar las soluciones más prometedoras a un problema de forma inteligente de manera que se optimice (maximice o minimice) una determinada función objetivo.

Algoritmo de evaluación de atributos

Un algoritmo de evaluación de atributos, en aprendizaje supervisado, permite determinar la bondad de los atributos con el fin de seleccionar aquellos más relevantes de cara a su relación con la clase. Existen dos categorías: algoritmos de evaluación de atributos individuales (también llamados *univariantes*) y algoritmos de evaluación de conjuntos de atributos (*multivariantes*).

Aprendizaje automático

El aprendizaje automático es una rama de conocimiento en la que se abordan algoritmos que son capaces de crear modelos de conocimiento abstractos a partir de históricos de datos.

Atributo

Un atributo es una variable (columna) de una tabla de datos que actúa como variable de entrada en el aprendizaje automático.

Ciencia de datos

Es una rama de la ciencia que estudia las técnicas informáticas para la adquisición, el procesamiento y el análisis de la información. La ciencia de datos es el marco de conocimiento fundamental para las disciplinas de la minería de datos y el aprendizaje automático.

Clase

Es la variable (columna) de una tabla de datos que actúa como variable de salida en el aprendizaje supervisado.

Conjunto de entrenamiento

Es el conjunto de datos utilizado para que un algoritmo de aprendizaje automático aprenda y genere su modelo de conocimiento.

Conjunto de test

Es el conjunto de datos utilizado para que un algoritmo de aprendizaje automático realice sus predicciones utilizando el modelo aprendido con el conjunto de entrenamiento.

Errores de entrenamiento

Son los errores cometidos por un modelo cuando predice los mismos ejemplos de entrenamiento con los que fue entrenado.

Errores de generalización

Son los errores cometidos por un modelo cuando predice los ejemplos de un conjunto de test diferente al de entrenamiento.

Homotecia

La homotecia es una operación matemática que permite cambiar de escala los valores de una variable. Suele emplearse para normalizar o estandarizar los datos.

Instancia

Una instancia es una fila de una tabla de datos. También se llama *ejemplo, muestra, observación, punto o prototipo (instance, data point, sample)*.

Interfaz de programación de aplicaciones

Una interfaz de programación de aplicaciones (API, de *application programming interface* en inglés) es un repertorio de funciones definidas mediante sus parámetros de entrada y de salida, expuestas y preparadas para ser utilizadas por otras aplicaciones informáticas.

Matriz de confusión

La matriz de confusión es una métrica de tipo matriz que contiene el recuento de predicciones realizadas por un algoritmo de aprendizaje automático organizadas por su correspondencia con la clase real. Es una matriz cuadrada cuyo orden es el número de clases del conjunto de datos.

Metodología CRISP-DM

La metodología CRISP-DM (de *cross-industry standard process for data mining* en inglés) fue concebida a finales de 1996 e integra todas las tareas necesarias en los proyectos de minería de datos reales, desde la fase de comprensión del problema hasta la puesta en producción de sistemas automatizados analíticos, predictivos y/o prospectivos, incluyendo las tareas de adquisición y comprensión de los datos, limpieza y transformación, análisis y visualización, creación de modelos y extracción de patrones, evaluación e interpretación de resultados.

Métricas de evaluación absolutas

Son medidas para evaluar la bondad de los algoritmos de aprendizaje supervisado en las que las desviaciones entre los valores reales y los predichos tienen la misma unidad de magnitud que la variable de clase.

Métricas de evaluación relativas

Son medidas para evaluar la bondad de los algoritmos de aprendizaje supervisado que revelan la proporción (porcentaje o tanto por uno) del valor desviado con respecto al valor real.

Minería de datos

La minería de datos hace referencia al conjunto de procesos, métodos y técnicas que conducen a la extracción de conocimiento a partir de bases de datos.

Modelo

Un modelo, en aprendizaje supervisado, representa el conocimiento adquirido por un algoritmo de aprendizaje que es extraído de forma inteligente a partir de las relaciones encontradas entre los atributos y la clase en las instancias de una tabla de datos.

Reducción de la dimensionalidad

La reducción de la dimensionalidad consiste en reducir la cantidad de datos antes de la fase de modelado, con el objetivo de mejorar, o al menos mantener, la eficacia de los modelos, y al mismo tiempo incrementar su eficiencia (por la reducción del volumen de datos).

Sobreajuste

El sobreajuste de un modelo a los datos es un fenómeno indeseado que sucede cuando un algoritmo de aprendizaje tiende a construir modelos de complejidad elevada que minimizan en gran medida los errores de entrenamiento a costa de aumentar los errores de generalización.

Sobreestimación

En aprendizaje supervisado, una sobreestimación es una predicción cuyo valor es superior al valor real de la clase.

Subajuste

El subajuste de un modelo a los datos es un fenómeno indeseado que sucede cuando un algoritmo de aprendizaje tiende a construir modelos de escasa complejidad que son incapaces de reflejar las relaciones importantes del conjunto de datos. Es el problema contrario al sobreajuste.

Subestimación

En aprendizaje supervisado, una subestimación es una predicción cuyo valor es inferior al valor real de la clase.

Tabla de datos

La tabla de datos es una estructura en dos dimensiones (con filas y columnas). A diferencia de las matrices en matemáticas, las tablas de datos pueden contener diferentes tipos de datos (números, cadenas, fechas...) en su interior. Cada columna de una tabla de datos debe contener solo datos del mismo tipo. Las tablas de datos son la estructura principal utilizada en el aprendizaje automático.

Validación

Proceso mediante el cual se divide el conjunto de datos en subconjuntos de entrenamiento y test con el objetivo de evaluar de forma adecuada la bondad de los algoritmos de aprendizaje supervisado.

Validación anidada

Tipo de validación en la que tienen lugar dos validaciones en cascada: una externa y otra interna. La validación externa puede coincidir con una validación cruzada clásica, mientras que la interna se aplica sobre cada bolsa de entrenamiento de la externa, lo cual permite que se utilicen varios tests de validación.

Validación cruzada

La validación cruzada consiste en partir el conjunto de datos original en subconjuntos (también llamados *bolsas*, *folds* en inglés) de igual tamaño. Una vez partido el conjunto de datos en bolsas, se realizan validaciones de tipo hold-out empleando, en cada validación, como test una bolsa distinta y como entrenamiento el resto de los ejemplos de las demás bolsas.

Validación estratificada

Tipo de validación en la que los conjuntos de test que se generan guardan aproximadamente la misma distribución de clases que los conjuntos de entrenamiento.

Validación hold-out

Tipo de validación en la que el conjunto de datos del problema se divide una única vez en dos partes: entrenamiento y test. Se utiliza habitualmente un porcentaje, que indica la proporción de muestras que se destinarán al conjunto de entrenamiento, siendo el resto de las muestras las destinadas para el conjunto de test.

Validación leave-one-out

Caso particular de validación cruzada en la que el número de bolsas coincide con el número de instancias del conjunto de datos.



Enlaces de interés

KDnuggets

Sitio web de noticias y artículos de interés de reconocido prestigio dentro del área de la ciencia de datos y el aprendizaje automático. Contiene información de actualidad sobre algoritmos de aprendizaje supervisado, así como las metodologías, técnicas, herramientas, frameworks y librerías software para el análisis inteligente de la información.

<https://www.kdnuggets.com>

Aprendizaje automático visual

Sitio web visual e interactivo dedicado al funcionamiento de algoritmos de aprendizaje automático. Contiene un artículo que explica el proceso de construcción de los árboles de decisión y se ilustra el concepto de sobreajuste de los modelos.

<http://www.r2d3.us>

Curso de Machine Learning de Google

Curso en línea muy completo para aprender aprendizaje supervisado ofrecido por Google. Posee una amplia cobertura de conceptos, métodos y algoritmos dentro del aprendizaje automático.

<https://developers.google.com/machine-learning/crash-course>

Sitio web de scikit-learn

Sitio web oficial de la librería scikit-learn, que proporciona un framework para el desarrollo, la utilización y la experimentación de algoritmos de aprendizaje automático usando el lenguaje de programación Python.

<http://scikit-learn.org/stable/>

Canal de Youtube sobre Machine Learning de Google

Canal de vídeos de YouTube sobre aprendizaje automático ofrecidos por Google. Incluye tanto vídeos introductorios como avanzados para ampliar en temas de deep learning y tratamiento de imágenes.

https://www.youtube.com/playlist?list=PLOU2XLYxmslluiBfYad6rFYQU_jL2ryal

Blog de Analytics Vidhya

Sitio web de noticias y artículos sobre ciencia de datos y aprendizaje automático. Contiene información de actualidad sobre algoritmos de aprendizaje supervisado y su aplicación en problemas del mundo real.

<https://www.analyticsvidhya.com/blog/>

Bibliografía



- Aggarwal, C. C. (2015). *Data Mining: The Textbook*. New York: Springer.
- James, G., Witten, D., Hastie, T., y Tibshirani, R. (2013). *An introduction to statistical learning*. New York: Springer.
- Kirk, M. (2017). *Thoughtful Machine Learning with Python*. Sebastopol: O'Reilly.
- Mueller, A. C., y Guido, S. (2016). *Introduction to Machine Learning with Python*. Sebastopol: O'Reilly.
- Sarkar, D., Bali, R., y Sharma, T. (2018). *Machine Learning with Python*. New York: Apress.
- Watt, J., Borhani, R., y Katsaggelos, A. K. (2016). *Machine Learning Refined. Foundations, Algorithms, and Applications*. Cambridge: Cambridge University Press.



Autor
Dr. Gualberto Asencio Cortés