Aprendizaje Computacional

Mario Graff

Tabla de contenidos

Prefacio **8**
Notación . . . 8
Licencia . . . 9

1 Tipos de Aprendizaje Computacional **10**
1.1 Introducción . . . 10
1.1.1 Aplicaciones de Aprendizaje Computacional . . . 11
1.2 Metodología General en Aprendizaje Supervisado y No Supervisado . . . 11
1.3 Aprendizaje No Supervisado . . . 12
1.4 Aprendizaje Supervisado . . . 12
1.5 Definiciones de Aprendizaje Supervisado . . . 17
1.5.1 Características de la hipótesis . . . 19
1.5.2 Estilos de Aprendizaje . . . 22
1.5.3 Sobre-aprendizaje . . . 22
1.5.4 Sub-aprendizaje . . . 23

2 Teoría de Decisión Bayesiana **25**
Paquetes usados . . . 25
2.1 Introducción . . . 25
2.2 Probabilidad . . . 26
2.2.1 Ejemplos . . . 26
2.3 Teorema de Bayes . . . 26
2.3.1 Problema Sintético . . . 27
2.3.2 Predicción . . . 29
2.4 Error de Clasificación . . . 31
2.5 Riesgo . . . 33
2.5.1 Acción nula . . . 34
2.6 Seleccionando la acción . . . 35

3 Métodos Paramétricos **36**
Paquetes usados . . . 36
3.1 Introducción . . . 36
3.2 Metodología . . . 37
3.3 Estimación de Parámetros . . . 37
3.3.1 Verosimilitud . . . 37

3.3.2 Distribución de Bernoulli . . . 38
3.3.3 Ejemplo: Distribución Gausiana . . . 38
3.4 Metodología de Clasificación . . . 40
3.5 Conjunto de Entrenamiento y Prueba . . . 41
3.5.1 Modelo . . . 41
3.5.2 Estimación de Parámetros . . . 41
3.5.3 Predicción . . . 42
3.5.4 Rendimiento . . . 43
3.6 Clasificador Bayesiano Ingenuo . . . 43
3.7 Ejemplo: Breast Cancer Wisconsin . . . 44
3.7.1 Entrenamiento . . . 45
3.7.2 Predicción . . . 45
3.7.3 Rendimiento . . . 45
3.8 Diferencias en Rendimiento . . . 46
3.9 Regresión . . . 48
3.9.1 Ejemplo: Diabetes . . . 49

4 Rendimiento **53**
Paquetes usados . . . 53
4.1 Introducción . . . 53
4.2 Clasificación . . . 54
4.2.1 Error . . . 55
4.2.2 Exactitud (*Accuracy*) . . . 55
4.2.3 Recall . . . 55
4.2.4 Precisión (*Precision*) . . . 56
4.2.5 F_β . . . 56
4.2.6 Medidas Macro . . . 56
4.2.7 Entropía Cruzada . . . 57
4.2.8 Área Bajo la Curva *ROC* . . . 57
4.2.9 Ejemplo . . . 57
4.3 Regresión . . . 59
4.3.1 Ejemplo . . . 60
4.4 Conjunto de Validación y Validación Cruzada . . . 61
4.4.1 k-Iteraciones de Validación Cruzada . . . 63

5 Reducción de Dimensión **65**
Paquetes usados . . . 65
5.1 Introducción . . . 66
5.2 Selección de Variables basadas en Estadísticas . . . 66
5.2.1 Ventajas y Limitaciones . . . 68
5.3 Selección hacia Adelante . . . 69
5.3.1 Ventajas y Limitaciones . . . 71
5.4 Selección mediante Modelo . . . 71

5.5 Análisis de Componentes Principales 72
5.5.1 Ejemplo - Visualización 74
5.6 UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction 76

6 Agrupamiento **79**
6.1 Paquetes usados 79
6.2 Introducción 79
6.3 K-medias 80
6.3.1 μ_i 80
6.3.2 G_i 80
6.3.3 Algoritmo 82
6.4 Ejemplo: Iris 84
6.5 Rendimiento 85
6.6 Número de Grupos 87

7 Métodos No Paramétricos **89**
7.1 Paquetes usados 89
7.2 Introducción 89
7.3 Histogramas 90
7.3.1 Selección del tamaño del bin 90
7.4 Estimador de Densidad por Kernel 92
7.4.1 Caso multidimensional 94
7.5 Estimador de Densidad por Vecinos Cercanos 95
7.6 Clasificador de vecinos cercanos 95
7.6.1 Implementación 97
7.7 Regresión 99

8 Árboles de Decisión **101**
8.1 Paquetes usados 101
8.2 Introducción 101
8.3 Clasificación 101
8.3.1 Predicción 104
8.3.2 Entrenamiento 104
8.3.3 Ejemplo: Breast Cancer Wisconsin 107
8.4 Regresión 108
8.4.1 Predicción 109
8.4.2 Entrenamiento 109

9 Discriminantes Lineales **113**
9.1 Paquetes usados 113
9.2 Introducción 113

9.3 Función Discriminante . . . 113
9.3.1 Clasificación Binaria . . . 114
9.3.2 Geometría de la Función de Decisión . . . 115
9.3.3 Múltiples Clases . . . 119
9.4 Máquinas de Soporte Vectorial . . . 120
9.4.1 Optimización . . . 121
9.4.2 Kernel . . . 123
9.5 Regresión Logística . . . 124
9.5.1 Optimización . . . 125
9.6 Comparación . . . 126

10 Optimización **128**
10.1 Paquetes usados . . . 128
10.2 Introducción . . . 128
10.3 Descenso por Gradiente . . . 129
10.3.1 Ejemplo - Regresión Lineal . . . 129
10.4 Diferenciación Automática . . . 132
10.4.1 Una Variable . . . 132
10.4.2 Dos Variables . . . 133
10.4.3 Visualización . . . 133
10.4.4 Regresión Lineal . . . 133
10.5 Regresión Logística . . . 135
10.6 Actualización de Parámetros . . . 138

11 Redes Neuronales **140**
11.1 Paquetes usados . . . 140
11.2 Introducción . . . 140
11.3 Regresión Logística Multinomial . . . 140
11.3.1 Optimización . . . 142
11.3.2 Optimización Método Adam . . . 143
11.3.3 Comparación entre Optimizadores . . . 144
11.4 Perceptrón . . . 144
11.4.1 Composición de Perceptrones Lineales . . . 146
11.5 Perceptrón Multicapa . . . 146
11.5.1 Desvanecimiento del Gradiente . . . 147
11.6 Ejemplo: Dígitos . . . 148

12 Ensambles **152**
12.1 Paquetes usados . . . 152
12.2 Introducción . . . 152
12.3 Fundamentos . . . 153
12.4 Bagging . . . 155
12.4.1 Ejemplo: Dígitos . . . 155

12.4.2 Ejemplo: Diabetes . . . 158
12.5 Stack Generalization . . . 161
12.5.1 Ejemplo: Diabetes . . . 161

13 Comparación de Algoritmos **163**
13.1 Paquetes usados . . . 163
13.2 Introducción . . . 163
13.3 Intervalos de confianza . . . 163
13.3.1 Método: Distribución Normal . . . 164
13.3.2 Ejemplo: Exactitud . . . 164
13.3.3 Método: Bootstrap del error estándar . . . 166
13.3.4 Método: Percentil . . . 167
13.3.5 Ejemplo: macro-recall . . . 168
13.4 Comparación de Algoritmos . . . 169
13.4.1 Método: Distribución t de Student . . . 169
13.4.2 Método: Bootstrap en diferencias . . . 170

Referencias **173**

Apéndices **174**

A Estadística **174**
Paquetes usados . . . 174
A.1 Error estándar . . . 174
A.1.1 Media . . . 174
A.1.2 Ejemplo: Media . . . 175
A.1.3 Ejemplo: Coeficientes OLS . . . 177
A.2 Bootstrap . . . 177
A.2.1 Ejemplo . . . 178

B Código **180**
Paquetes usados . . . 180
B.1 Clasificador Bayesiano Gausiano . . . 180
B.1.1 Estimación de Parámetros . . . 181
B.1.2 Predicción . . . 182
B.1.3 Uso . . . 182

C Conjunto de Datos **183**
Paquetes usados . . . 183
C.1 Problemas Sintéticos . . . 183
C.2 Mezcla de Clases . . . 183
C.2.1 Clases Separadas . . . 184

C.3 Problemas de Clasificación 185
C.3.1 Breast Cancer Wisconsin 186
C.3.2 Iris 186
C.3.3 Números 186
C.3.4 Vino 186
C.4 Problemas de Regresión 186
C.4.1 Problema Sintético 187
C.4.2 Diabetes 188

Prefacio

Este curso ha evolucionado de las clases de Aprendizaje Computacional impartidas en la Maestría en Ciencia de Datos e Información (MCDI) de INFOTEC y de Aprendizaje Computacional en la Maestría en Métodos para el Análisis de Políticas Públicas del CIDE.

En la MCDI compartí el curso con la Dra. Claudia N. Sánchez y parte de este material, en particular algunas figuras, fueron generadas por la Dra. Sánchez.

El curso trata de ser auto-contenido, es decir, no debería de ser necesario leer otras fuentes para poder entenderlo y realizar las actividades. De cualquier manera es importante comentar que el curso está basado en los siguientes libros de texto:

- Introduction to machine learning, Third Edition. Ethem Alpaydin. MIT Press.
- Probabilistic Machine Learning: An Introduction. Kevin Patrick Murphy. MIT Press.
- An Introduction to Statistical Learning with Applications in R. Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. Springer Texts in Statistics.
- All of Statistics. A Concise Course in Statistical Inference. Larry Wasserman. MIT Press.
- An Introduction to the Bootstrap. Bradley Efron and Robert J. Tibshirani. Monographs on Statistics and Applied Probability 57. Springer-Science+Business Media.
- Understanding Machine Learning: From Theory to Algorithms. Shai Shalev-Shwartz and Shai Ben-David. Cambridge University Press.

Notación

La Tabla 1 muestra la notación que se seguirá en este documento.

Tabla 1: Notación

Símbolo	Significado
x	Variable usada comunmente como entrada
y	Variable usada comunmente como salida
$\mathbb{R}$	Números reales
$\mathbf{x}$	Vector Columna $\mathbf{x} \in \mathbb{R}^d$
$\|\mathbf{x}\|$	Norma Euclideana
d	Dimensión

Símbolo	Significado
$\mathbf{w} \cdot \mathbf{x}$	Producto punto donde $\mathbf{w}$ y $\mathbf{x} \in \mathbb{R}^d$
$\mathcal{D}$	Conjunto de datos
$\mathcal{T}$	Conjunto de entrenamiento
$\mathcal{V}$	Conjunto de validación
$\mathcal{G}$	Conjunto de prueba
N	Número de ejemplos
K	Número de clases
$\mathbb{P}(\cdot)$	Probabilidad
$\mathcal{X}, \mathcal{Y}$	Variables aleatorías
$\mathcal{N}(\mu, \sigma^2)$	Distribución Normal con parámetros μ y σ^2
$f_{\mathcal{X}}$	Función de densidad de probabilidad de $\mathcal{X}$
$\mathbb{1}(e)$	Función para indicar; 1 si e es verdadero
Ω	Espacio de búsqueda
$\mathbb{V}$	Varianza
$\mathbb{E}$	Esperanza

Licencia

1 Tipos de Aprendizaje Computacional

El **objetivo** de la unidad es conocer los diferentes tipos de aprendizaje computacional y los conceptos globales propios a este campo de estudio.

1.1 Introducción

Aprendizaje computacional es una rama de la IA que estudia los algoritmos que son capaces de aprender a partir de una serie de ejemplos o con alguna guía. Existen diferentes tipos de aprendizaje computacional, los mas comunes son: aprendizaje supervisado, aprendizaje no-supervisado y aprendizaje por refuerzo.

En **aprendizaje supervisado** se crean modelos partiendo de un conjunto de pares, entrada y salida, donde el objetivo es encontrar un modelo que logra aprender esta relación y predecir ejemplos no vistos en el proceso, en particular a esto se le conoce como *inductive learning*. Complementando este tipo de aprendizaje supervisado se tiene lo que se conoce como *transductive learning*, en el cual se cuenta con un conjunto de pares y solamente se requiere conocer la salida en otro conjunto de datos. En este segundo tipo de aprendizaje todos los datos son conocidos en el proceso de aprendizaje.

Aprendizaje no-supervisado es aquel donde se tiene un conjunto de entradas y se busca aprender alguna relación de estas entradas, por ejemplo, generando grupos o utilizando estas entradas para hacer una transformación o encontrar un patrón.

Finalmente **aprendizaje por refuerzo** es aquel donde se tiene un agente que tiene que aprender como interactuar con un ambiente. La interacción es tomando una acción en cada diferente estado del ambiente. Por ejemplo, el agente puede ser un jugador virtual en un juego de ajedrez entonces la acción es identificar y mover una pieza en el tablero, el objetivo de ganar la partida. La característica de aprendizaje por refuerzo es que el agente va a recibir una recompensa al final de la interacción con el ambiente, e.g., final del juego y el objetivo es optimizar las acciones para que la recompensa sea la mayor posible.

1.1.1 Aplicaciones de Aprendizaje Computacional

Es importante mencionar que no todo lo que se hace en aprendizaje automático está relacionado con juegos clásicos y que existen avances importantes en otros dominios. Como por ejemplo, recientemente en el área de cardiología se presenta el siguiente artículo https://www.nature.com/articles/s41591-018-0268-3, o en dermatología para detección de cancer https://www.nature.com/articles/nature21056 solo por mostrar algunos otros ejemplos.

Cabe mencionar que los tres tipos de aprendizaje no son excluyentes uno del otro, comúnmente para resolver un problema complejo se combinan diferentes tipos de aprendizaje y otras tecnologías de IA para encontrar una solución aceptable. Probablemente una de las pruebas más significativas de lo que puede realizarse con aprendizaje automático es lo realizado por AlphaGo, leer el resumen del siguiente artículo https://www.nature.com/articles/nature16961 y recientemente se quita una de las restricciones originales, la cual consiste en contar con un conjunto de jugadas realizadas por expertos, publicando este descubrimiento en https://www.nature.com/articles/nature24270.

En el área de aprendizaje, hay una tendencia de utilizar plataformas donde diferentes empresas u organismos gubernamentales o sin fines de lucro, ponen un problema e incentivan al publico en general a resolver este problema. La plataforma sirve de mediador en este proceso. Ver por ejemplo https://www.kaggle.com.

En el ámbito científico también se han generado este tipo de plataformas aunque su objetivo es ligeramente diferente, lo que se busca es tener una medida objetiva de diferentes soluciones y en algunos casos facilitar la reproducibilidad de las soluciones. Ver por ejemplo http://codalab.org.

1.2 Metodología General en Aprendizaje Supervisado y No Supervisado

Antes de continuar con la descripción de los diferentes tipos de aprendizaje es importante mencionar la metodología que se sigue en los problemas de aprendizaje supervisado y no supervidado

1. Todo empieza con un conjunto de datos $\mathcal{D}$ que tiene la información del fenómeno de interés.
2. Se selecciona el conjunto de entrenamiento $\mathcal{T} \subseteq \mathcal{D}$.
3. Se diseña un algoritmo, f, utilizando $\mathcal{T}$.
4. Se utiliza f para estimar las características modeladas.
5. Se mide el rendimiento de f.

1.3 Aprendizaje No Supervisado

Iniciamos la descripción de los diferentes tipos de aprendizaje computacional con **aprendizaje no-supervisado**; el cual inicia con un conjunto de elementos. Estos tradicionalmente se puede transformar en conjunto de vectores, i.e. $\mathcal{D} = \{x_1, \dots, x_N\}$, donde $x_i \in \mathbb{R}^d$. Durante este curso asumiremos que esta transformación existe y en algunos casos se hará explícito el algoritmo de transformación.

El **objetivo** en aprendizaje no supervisado es desarrollar algoritmos capaces de encontrar patrones en los datos, es decir, en $\mathcal{D}$. Existen diferentes tareas que se pueden considerar dentro de este tipo de aprendizaje. Por ejemplo, el agrupamiento puede servir para segmentar clientes o productos, en otra línea también cabría el análisis del carrito de compras (Market Basket Analysis); donde el objetivo es encontrar la co-ocurrencias de productos, es decir, se quiere estimar la probabilidad de que habiendo comprado un determinado artículo también se compre otro artículo. Con esta descripción ya se podrá estar imaginando la cantidad de aplicaciones en las que este tipo de algoritmos es utilizado en la actualidad.

Regresando a la representación vectorial, existen casos donde se pueden visualizar los elementos de $\mathcal{D}$, lo cuales están representados como puntos que se muestran en la Figura 1.1. Claramente esto solo es posible si $x_i \in \mathbb{R}^2$ o si se hace algún tipo de transformación $f : \mathbb{R}^d \to \mathbb{R}^2$, como se realizó en la figura.

En la Figura 1.1 se pueden observar dos o tres grupos de puntos, entonces el objetivo sería crear el algoritmo que dado $\mathcal{D}$ regrese un identificador por cada elemento, dicho identificador representa el grupo al que pertenece el elemento en cuestión. Esta tarea se le conoce como agrupamiento (Clustering). Asumiendo que se aplica un algoritmo de agrupamiento a los datos anteriores; entonces, dado que podemos visualizar los datos, es factible representar el resultado del algoritmo si a cada punto se le asigna un color dependiendo de la clase a la que pertenece. La Figura 1.2 muestra el resultado de este procedimiento.

Se puede observar en la figura anterior, el algoritmo de agrupamiento separa los puntos en tres grupos, representados por los colores azul, naranja y verde. Cabe mencionar que utilizando algún otro criterio de optimización se hubiera podido encontrar dos grupos, el primero de ellos sería el grupo de los puntos de color verde y el segundo sería el grupo formado por los puntos azules y naranjas. Es importante recalcar que no es necesario visualizar los datos para aplicar un algoritmo de agrupamiento. En particular el ejercicio de visualización de datos y del resultado de agrupamiento que se muestra en la figuras anteriores tiene el objetivo de generar una intuición de lo que está haciendo un algoritmo de agrupamiento.

1.4 Aprendizaje Supervisado

Aprendizaje supervisado es un problema donde el componente inicial es un conjunto de pares, entrada y salida. Es decir se cuenta con $\mathcal{D} = \{(x_1, y_1), \dots, (x_N, y_N)\}$, donde $x_i \in \mathbb{R}^d$ corres-

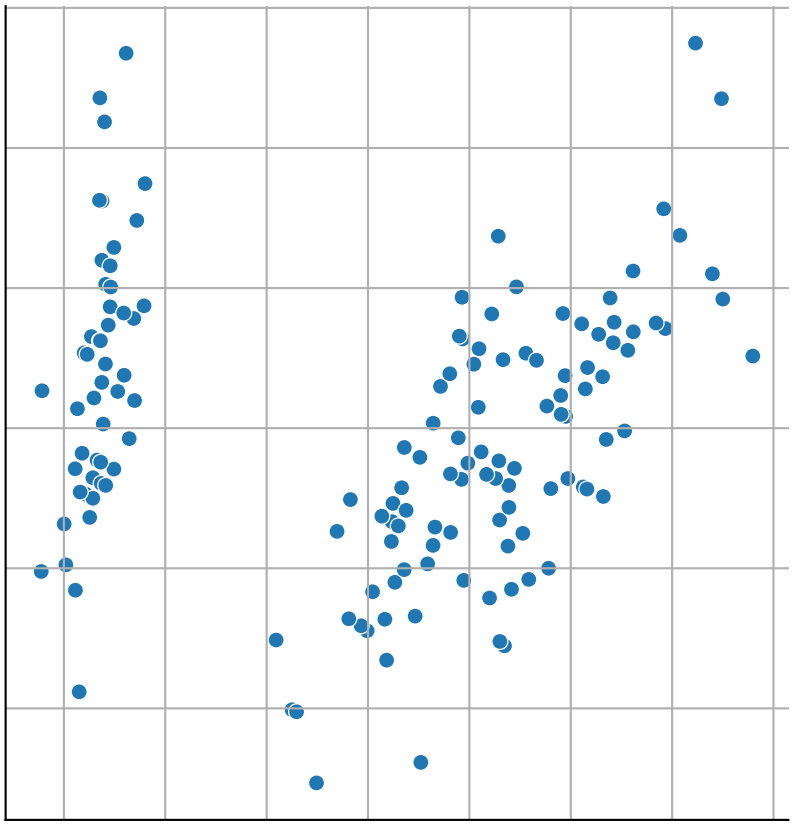


Figura 1.1: Proyección de los datos del Iris en dos dimensiones

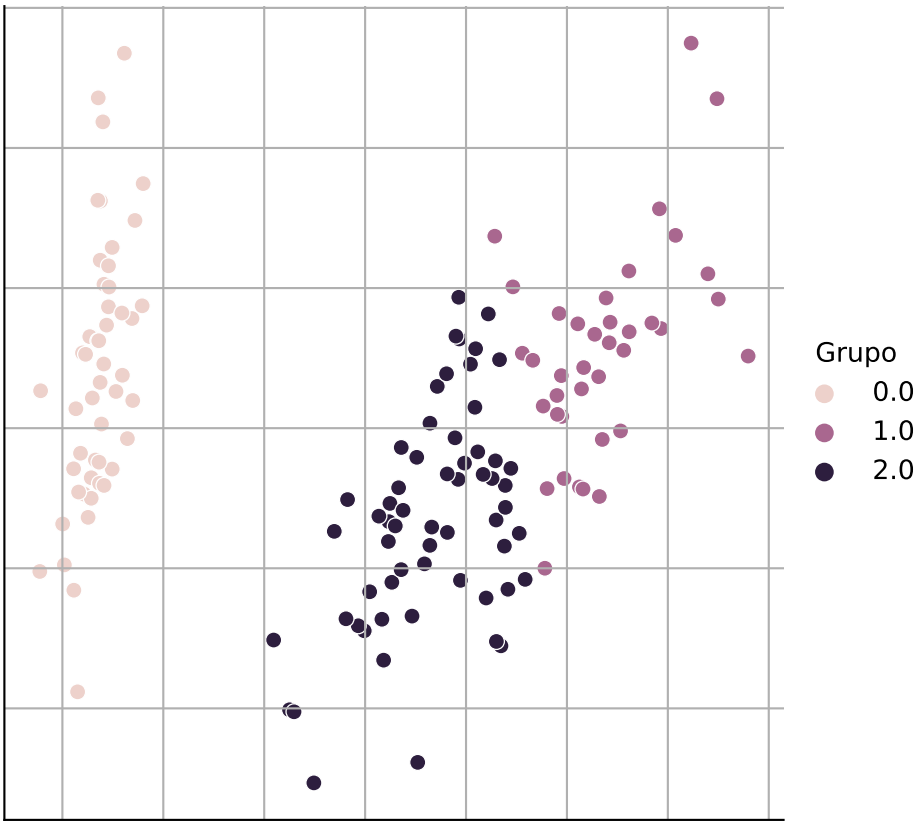


Figura 1.2: Proyección de agrupar los datos del Iris usando K-medias

ponde a la i-ésima entrada y y_i es la salida asociada a esa entrada. Tomando en cuenta estas condiciones iniciales podemos definir el *objetivo* de aprendizaje supervisado como encontrar un algoritmo capaz de regresar la salida y dada una entrada x.

Existe una gran variedad de problemas que se puede categorizar como tareas de aprendizaje supervisado, solamente hay que recordar que en todos los casos se inicia con un conjunto $\mathcal{D}$ de pares entrada y salida. En ocasiones la construcción del conjunto es directa, por ejemplo en el caso de que se quiera identificar si una persona será sujeta a un crédito, entonces el conjunto a crear esta compuesto por las características de las personas que se les ha otorgado un crédito y el estado final de crédito, es decir, si el crédito fue pagado o no fue pagado. En otro ejemplo, suponiendo que se quiere crear un algoritmo capaz de identificar si un texto dado tiene una polaridad positiva o negativa, entonces el conjunto se crea recolectando textos y a cada texto un conjunto de personas decide si el texto dado es positivo o negativo y la polaridad final es el consenso de varias opiniones; a este problema en general se le conoce como análisis de sentimientos.

La cantidad de problema que se pueden poner en términos de aprendizaje supervisado es amplia, un problema tangible en esta época y relacionado a la pandemia del COVID-19 sería el crear un algoritmo que pudiera predecir cuántos serán los casos positivos el día de mañana dando como entradas las restricciones en las actividades; por ejemplo escuelas cerradas, restaurantes al 30% de capacidad entre otras.

Los ejemplos anteriores corresponden a dos de las clases de problemas que se resuelven en aprendizaje supervisado estas son problemas de clasificación y regresión. Definamos de manera formal estos dos problemas. Cuando $y \in \{0, 1\}$ se dice que es un problema de **clasificación binaria**, por otro lado cuando $y \in \{0, 1\}^K$ se encuentra uno en clasificación **multi-clase** o **multi-etiqueta** y finalmente si $y \in \mathbb{R}$ entonces es un problema de **regresión**.

Haciendo la liga entre los ejemplos y las definiciones anteriores, podemos observar que el asignar un crédito o la polaridad a un texto es un problema de clasificación binaria, dado que se puede asociar 0 y 1 a la clase positivo y negativo; y en el otro caso a pagar o no pagar el crédito. Si el problema tiene mas categorías, supongamos que se desea identificar positivo, negativo o neutro, entonces se estaría en el problema de clasificación multi-clase. Por otro lado el problema de predecir el número de casos positivos se puede considerar como un problema de regresión, dado que el valor a predecir difícilmente se podría considerar como una categoría.

Al igual que en aprendizaje no supervisado, en algunos casos es posible visualizar los elementos de $\mathcal{D}$, el detalle adicional es que cada objeto tiene asociado una clase, entonces se selecciona un color para representar cada clase. En la Figura 1.3 se muestra el resultado donde los elementos de $\mathcal{D}$ se encuentra en $\mathbb{R}^2$ y el color representa cada una de la clases de este problema de clasificación binaria.

Usando esta representación es sencillo imaginar que el problema de clasificación se trata en encontrar una función que separe los puntos naranjas de los puntos azules, como se pueden imagina una simple línea recta podría separar estos puntos. La Figura 1.4 muestra un ejemplo

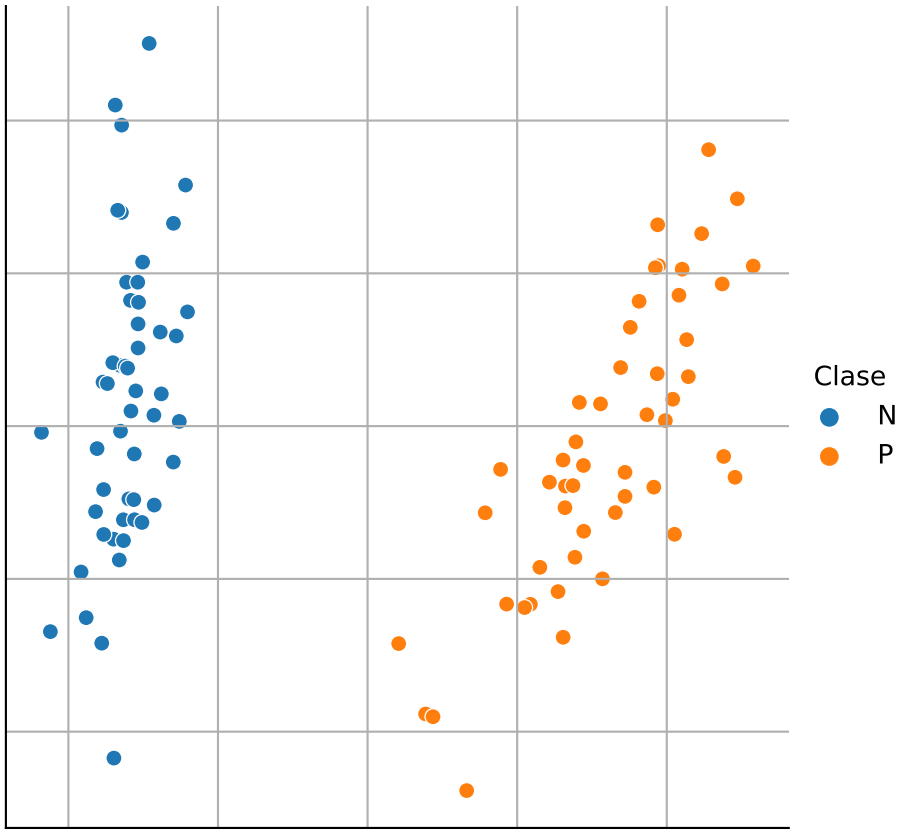


Figura 1.3: Proyección del Iris mostrando dos clases linealmente separables

de lo que haría un clasificador representado por la línea; la clase es dada por el signo de $ax + by + c$, donde a, b y c son parámetros identificados a partir de $\mathcal{D}$.

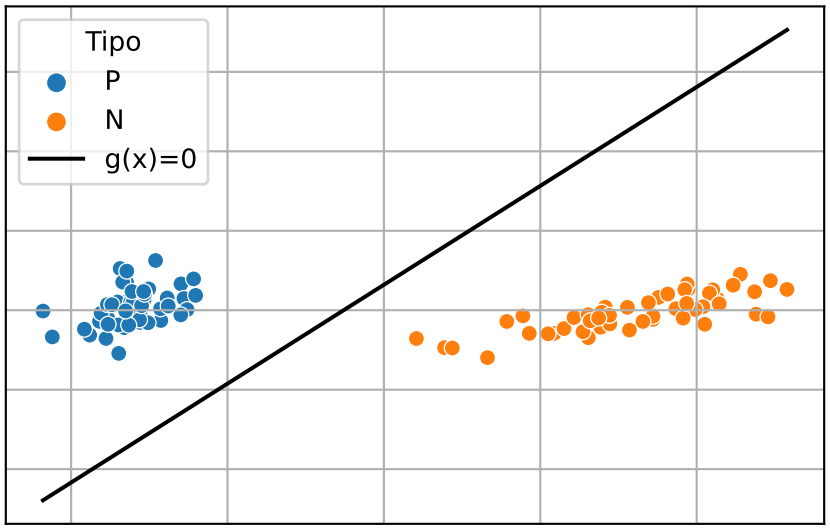


Figura 1.4: Proyección del Iris mostrando dos clases linealmente separables

Siguiendo en esta misma línea, también es posible observar los puntos en un problema de regresión, solamente que en este caso un eje corresponde a las entradas, i.e. x, y el otro eje es la salida, i.e. y. La Figura 1.5 muestra un ejemplo de regresión, donde se puede observar que la idea es una encontrar una función que pueda seguir de manera adecuada los puntos datos.

El problema de regresión es muy conocido y seguramente ya se imaginaron que la respuesta sería encontrar los parámetros de una parábola. La Figura 1.6 muestra una visualización del regresor, mostrado en color negro y los datos de entrenamiento en color azul.

Al igual que en aprendizaje no supervisado, este ejercicio de visualización no es posible en todos los problemas de aprendizaje supervisado, pero si permite ganar intuición sobre la forma en que trabajan estos algoritmos.

1.5 Definiciones de Aprendizaje Supervisado

El primer paso es empezar a definir los diferentes conjuntos con los que se trabaja en aprendizeje computacional. Todo inicia con el **conjunto de entrenamiento** identificado en este documento como $\mathcal{T}$. Este conjunto se utiliza para estimar los parámetros o en general buscar un algoritmo que tenga el comportamiento esperado.

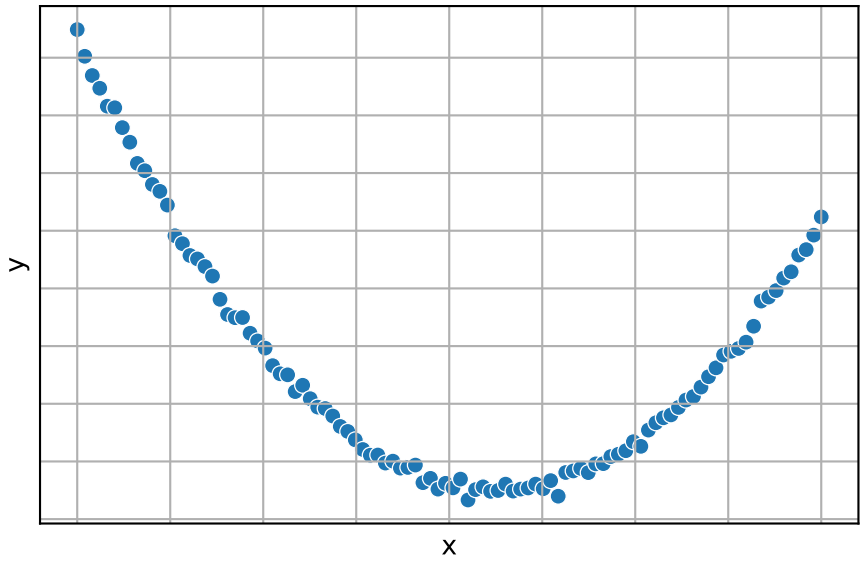


Figura 1.5: Problema de regresión

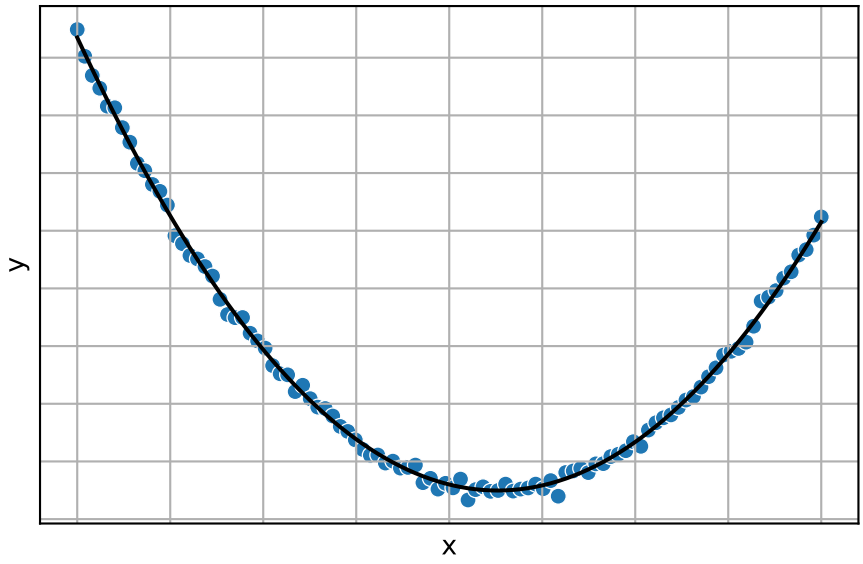


Figura 1.6: Problema de regresión con función con función estimada

Se puede asumir que existe una función f que genera la relación entrada salida mostrada en $\mathcal{T}$, es decir, idealmente se tiene que $\forall_{(x,y)\in\mathcal{D}} f(x) = y$. En este contexto, aprendizaje supervisado se entiende como el proceso de encontrar una función h^* que se comporta similar a f.

Para encontrar h^*, se utiliza $\mathcal{T}$; el conjunto de hipótesis (funciones), $\mathcal{H}$, que se considera puede aproximar f; una función de error, L; y el error empírico $E(h \mid \mathcal{T}) = \sum_{(x,y)\in\mathcal{T}} L(y, h(x))$. Utilizando estos elementos la función buscada es: $h^* = \mathsf{argmin}_{h\in\mathcal{H}} E(h \mid \mathcal{T})$.

El encontrar la función h^* no resuelve el problema de aprendizaje en su totalidad, además se busca una función que sea capaz de generalizar, es decir, que pueda predecir correctamente instancias no vistas. Considerando que se tiene un **conjunto de prueba**, $\mathcal{G} = \{(x_i, y_i)\}$ para $i = 1 \ldots M$, donde $\mathcal{T} \cap \mathcal{G} = \emptyset$ y $\mathcal{T} \cup \mathcal{G} = \mathcal{D}$. La idea es que el error empírico sea similar en el conjunto de entrenamiento y prueba. Es decir $E(h^* \mid \mathcal{T}) \approx E(h^* \mid \mathcal{G})$.

1.5.1 Características de la hipótesis

Continuando con algunas definiciones, en la búsqueda de encontrar h^* uno puede elegir por aquella que captura todos los datos de entrenamiento siendo muy específica. Para poder explicar mejor este concepto usemos la siguiente figura que representa un ejemplo sintético.

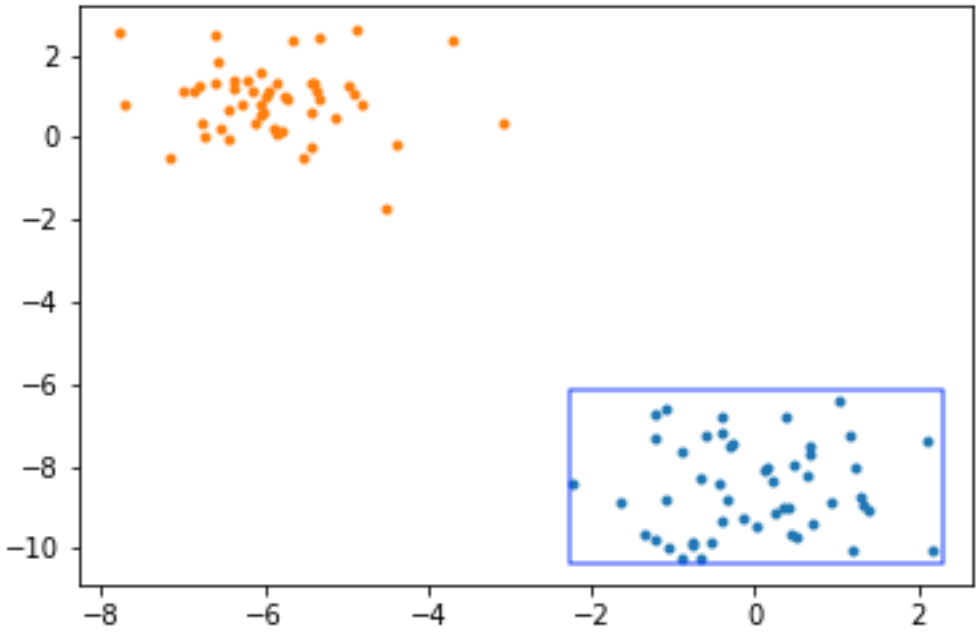


Figura 1.7: Hipótesis mas específica

Los puntos naranja representan una clase y los puntos azules son la otra clase, el clasificador es mostrado en el rectángulo. Para completar el funcionamiento de este clasificador falta mencionar que cualquier nuevo punto que esté dentro del rectángulo será calificado como clase azul y naranja si se encuentra fuera del rectángulo. Como se puede observar, el rectángulo contiene todos los elementos de una clase y en alguno de sus lados toca con uno de los puntos del conjunto de entrenamiento de la clase data.

En este momento, es posible visualizar que el complemento de ser muy específico es ser lo mas general posible. La siguiente siguiente figura muestra un clasificador que es lo mas general. Se observa que el rectángulo casi toca uno de los puntos del conjunto de entrenamiento de la clase contraria, esto lo hace del lado exterior y el procedimiento para clasificar continua siendo el que se mencionó anteriormente.

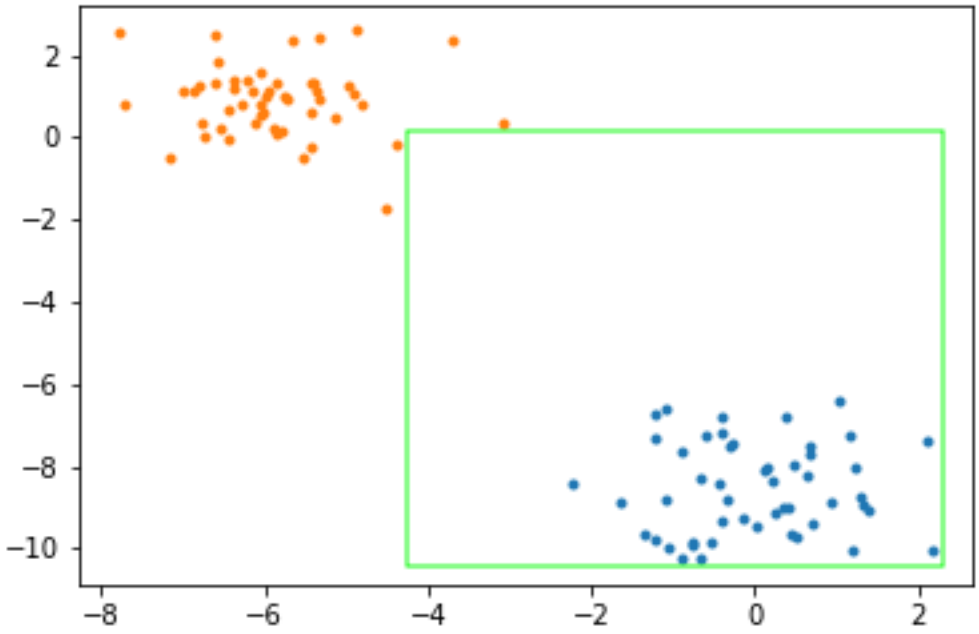


Figura 1.8: Hipótesis más general

Uno puede elegir una hipótesis que se encuentre entre la hipótesis mas general y la más específica, esto también se puede visualizar en la siguiente figura. Donde todas las hipótesis se encuentran representadas en gris.

Finalmente, para poder describir mejor el comportamiento de un clasificador se hace uso de la distancia que hay entre la hipótesis mas general y específica y la hipótesis utilizada. El margen

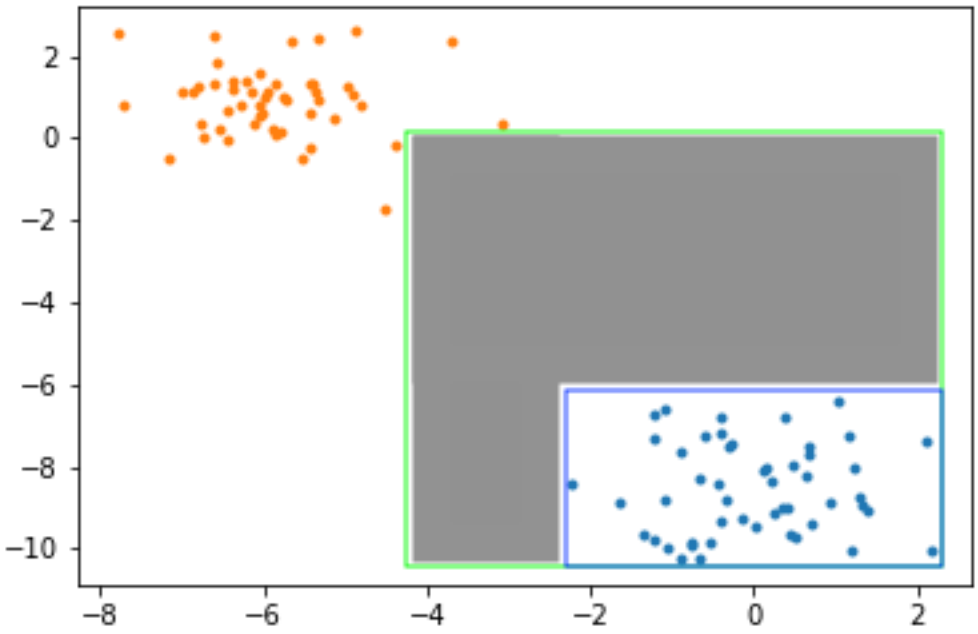


Figura 1.9: Clase de hipótesis

se puede visualizar en la siguiente figura, donde la hipótesis más general es mostrada en verde, la más específica en morada y la hipótesis utilizada en negro.

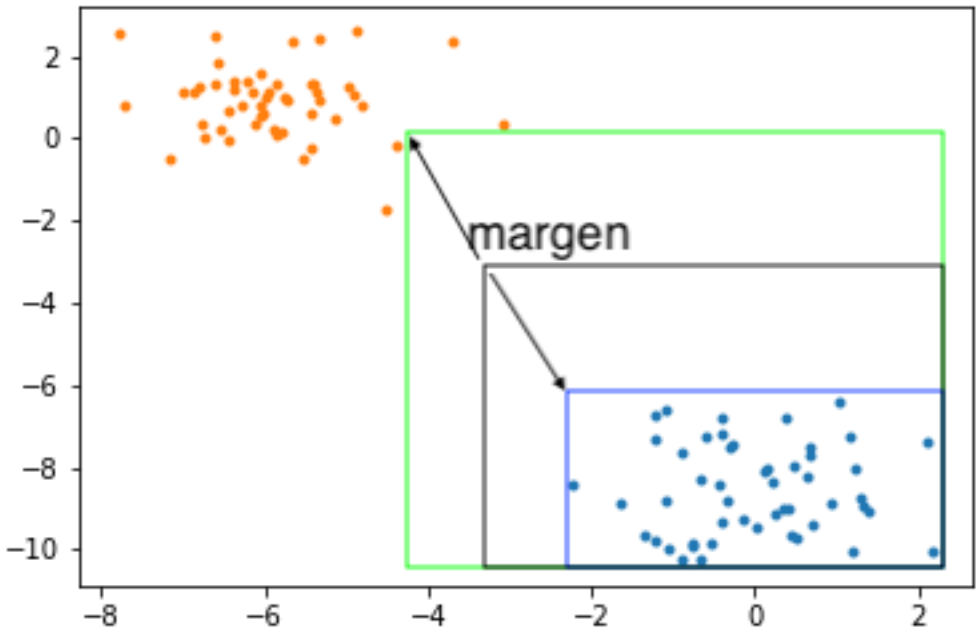


Figura 1.10: Margen

1.5.2 Estilos de Aprendizaje

Utilizando $\mathcal{T}$ y $\mathcal{G}$ podemos definir **inductive learning** como el proceso de aprendizaje en donde solamente se utiliza $\mathcal{T}$ y el algoritmo debe de ser capaz de predecir cualquier instancia. Por otro lado, **transductive learning** es el proceso de aprendizaje donde se utilizar $\mathcal{T} \cup \{x \mid (x, y) \in \mathcal{G}\}$ para aprender y solamente es de interés el conocer la clase o variable dependiente del conjunto $\mathcal{G}$.

1.5.3 Sobre-aprendizaje

Existen clases de algoritmos, $\mathcal{H}$, que tienen un mayor grado de libertad el cual se ve reflejado en una capacidad superior para aprender, pero por otro lado, existen problemas donde no se requiere tanta libertad, esta combinación se traduce en que el algoritmo no es capaz de generalizar y cuantitativamente se ve como $E(h^* \mid \mathcal{T}) \ll E(h^* \mid \mathcal{G})$.

Para mostrar este caso hay que imaginar que se tiene un algoritmo que guarda el conjunto de entrenamiento y responde lo siguiente:

$$h^*(x) = \begin{cases} y & \text{si}(x, y) \in \mathcal{T} \\ 0 \end{cases}$$

Es fácil observar que este algoritmo tiene $E(h^* \mid \mathcal{T}) = 0$ dado que se aprende todo el conjunto de entrenamiento.

La Figura 1.11 muestra el comportamiento de un algoritmo que sobre-aprende, el algoritmo se muestra en la línea naranja, la línea azul corresponde a una parábola (cuyos parámetros son identificados con los datos de entrenamiento) y los datos de entrenamiento no se muestran; pero se pueden visualizar dado que son datos generados por una parábola mas un error gaussiano. Entonces podemos ver que la línea naranja pasa de manera exacta por todos los datos de entrenamiento y da como resultado la línea naranja que claramente tiene un comportamiento mas complejo que el comportamiento de la parábola que generó los datos.

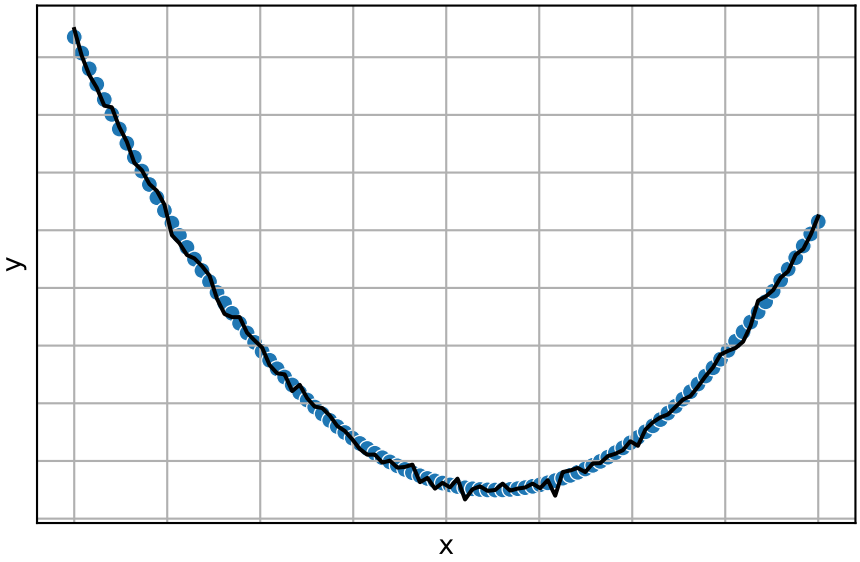


Figura 1.11: Sobre-aprendizaje en un problema de regresión

1.5.4 Sub-aprendizaje

Por otro lado existen problemas donde el conjunto de algoritmos $\mathcal{H}$ no tienen los grados de libertad necesarios para aprender, dependiendo de la medida de error esto se refleja como $E(h^* \mid \mathcal{T}) \gg 0$. La Figura 1.12 muestra un problema de regresión donde el algoritmo de aprendizaje presenta el problema de sub-aprendizaje.

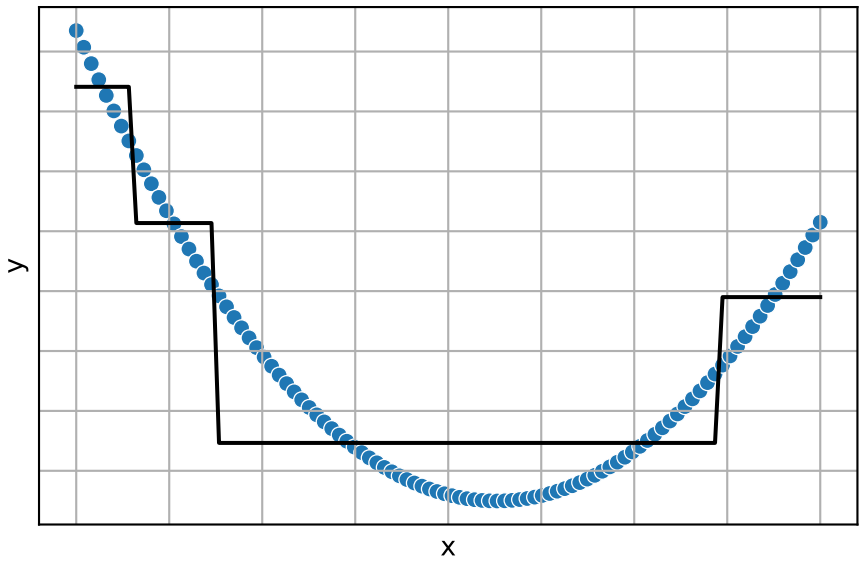


Figura 1.12: Sub-aprendizaje en un problema de regresión

2 Teoría de Decisión Bayesiana

El **objetivo** de la unidad es analizar el uso de la teoría de la probabilidad para la toma de decisiones. En particular el uso del teorema de Bayes para resolver problemas de clasificación y su uso para tomar la decisión que reduzca el riesgo.

Paquetes usados

```
from scipy.stats import multivariate_normal
from matplotlib import pylab as plt
import numpy as np
import pandas as pd
import seaborn as sns
```

2.1 Introducción

Al diseñar una solución a un problema particular lo mejor que uno puede esperar es tener una certeza absoluta sobre la respuesta dada. Por ejemplo, si uno diseña un algoritmo que ordene un conjunto de números uno esperan que ese algoritmo siempre regrese el orden correcto independientemente de la entrada dada, es mas un algoritmo de ordenamiento que en ocasiones se equivoca se consideraría de manera estricta erróneo.

Sin embargo, existen problemas cuyas características, como incertidumbre en la captura de los datos, variables que no se pueden medir, entre otros factores hacen que lo mejor que se puede esperar es un algoritmo exacto y preciso. Todos los problemas que trataremos en Aprendizaje Computacional caen dentro del segundo escenario. El lenguaje que nos permite describir de manera adecuada este tipo de ambiente, que se caracteriza por variables aleatorios es el de la probabilidad.

2.2 Probabilidad

En Sección 1.4 se describió que el punto de inicio de aprendizaje supervisado es el conjunto $\mathcal{D} = \{(x_1, y_1), \ldots, (x_N, y_N)\}$, donde $x_i \in \mathbb{R}^d$ corresponde a la i-ésima entrada y y_i es la salida asociada a esa entrada; este conjunto tiene el objetivo de guiar un proceso de búsqueda para encontrar una método que capture de la relación entre x y y.

Se pueden tratar la variables x y y de $\mathcal{D}$ como dos variables aleatorías $\mathcal{X}$ y $\mathcal{Y}$, respectivamente; en este dominio el problema de identificar la respuesta ($\mathcal{Y}$) dada la entrada ($\mathcal{X}$) se puede describir como encontrar la probabilidad de observar $\mathcal{Y}$ habiendo visto $\mathcal{X}$, es decir, $\mathbb{P}(\mathcal{Y} \mid \mathcal{X})$.

2.2.1 Ejemplos

Por ejemplo, en un problema de clasificación binaria se tiene que la respuesta tiene dos posibles valores, e.g., $\mathcal{Y} = \{0, 1\}$. Entonces el problema es saber si dada una entrada x el valor de la respuesta es 1 o 0. Utilizando probabilidad la pregunta quedaría como conocer la probabilidad de que $\mathcal{Y} = 1$ o $\mathcal{Y} = 0$ cuando se observa $\mathcal{X} = x$, es decir, encontrar $\mathbb{P}(\mathcal{Y} = 1 \mid \mathcal{X} = x)$ y compararlo contra $\mathbb{P}(\mathcal{Y} = 0 \mid \mathcal{X} = x)$. Tomando en cuenta estos valores de probabilidad se puede concluir el valor de la salida dado que $\mathcal{X} = x$. También está el caso que las probabilidades sean iguales, e.g., si $\mathbb{P}(\mathcal{Y} = 1 \mid \mathcal{X} = x) = \mathbb{P}(\mathcal{Y} = 0 \mid \mathcal{X} = x) = 0.5$ o que su diferencia sea muy pequeña y entonces se toma la decisión de desconocer el valor de la salida.

Para el caso de regresión ($y \in \mathbb{R}$), el problema se puede plantear asumiendo que $\mathcal{Y}$ proviene de una distribución particular cuyos parámetros están dados por la entrada $\mathcal{X}$. Por ejemplo, en regresión lineal se asume que $\mathcal{Y}$ proviene de una distribución Gausiana con parámetros dados por $\mathcal{X}$, es decir, $\mathbb{P}(\mathcal{Y} \mid \mathcal{X} = x) = \mathcal{N}(g(x) + \epsilon, \sigma^2)$, donde los parámetros de la función g son identificados mediante $\mathcal{X}$ y $\epsilon \sim \mathcal{N}(0, \sigma^2)$ es el error con media cero y desviación estándar σ. Con estas condiciones la salida y es $\mathbb{E}[\mathcal{Y} \mid \mathcal{X} = x]$; asumiendo que se esa variable se distribuye como una normal entonces $\mathbb{E}[\mathcal{Y} \mid \mathcal{X} = x] = \mathbb{E}[g(x) + \epsilon] = g(x) + \mathbb{E}[\epsilon] = g(x)$.

2.3 Teorema de Bayes

El problema se convierte en cómo calcular $\mathbb{P}(\mathcal{Y} \mid \mathcal{X})$, lo cual se puede realizar mediante el Teorema de Bayes el cual se deriva a continuación.

La probabilidad conjunta se puede expresar como $\mathbb{P}(\mathcal{X}, \mathcal{Y})$, esta probabilidad es conmutativa por lo que $\mathbb{P}(\mathcal{X}, \mathcal{Y}) = \mathbb{P}(\mathcal{Y}, \mathcal{X})$. En este momento se puede utilizar la definición de **probabilidad condicional** que es $\mathbb{P}(\mathcal{Y}, \mathcal{X}) = \mathbb{P}(\mathcal{Y} \mid \mathcal{X})\mathbb{P}(\mathcal{X})$. Utilizando estas ecuaciones el **Teorema de Bayes** queda como

$$\mathbb{P}(\mathcal{Y} \mid \mathcal{X}) = \frac{\mathbb{P}(\mathcal{X} \mid \mathcal{Y})\mathbb{P}(\mathcal{Y})}{\mathbb{P}(\mathcal{X})}, \tag{2.1}$$

donde al término $\mathbb{P}(\mathcal{X} \mid \mathcal{Y})$ se le conoce como **verosimilitud**, $\mathbb{P}(\mathcal{Y})$ es la probabilidad **a priori** y $\mathbb{P}(\mathcal{X})$ es la **evidencia**.

Es importante mencionar que la evidencia se puede calcular mediante la probabilidad total, es decir:

$$\mathbb{P}(\mathcal{X}) = \sum_{y \in \mathcal{Y}} \mathbb{P}(\mathcal{X} \mid \mathcal{Y} = y)\mathbb{P}(\mathcal{Y} = y). \tag{2.2}$$

2.3.1 Problema Sintético

Con el objetivo de entender el funcionamiento del Teorema de Bayes, esta sección presenta un problema sintético. El procedimiento es el siguiente, primero se generarán los datos, los cuales van a ser tres nubes de puntos generadas mediantes tres distribuciones gausianas multivariadas. Con estas tres nubes de puntos, se utilizará el Teorema de Bayes (Ecuación 2.1) para clasificar todos los puntos generados.

El primer paso es definir las tres distribuciones gausianas multivariadas, para este objetivo se usa la clase `multivariate_normal` como se muestra a continuación.

```
p1 = multivariate_normal(mean=[5, 5],
                         cov=[[4, 0], [0, 2]])
p2 = multivariate_normal(mean=[1.5, -1.5],
                         cov=[[2, 1], [1, 3]])
p3 = multivariate_normal(mean=[12.5, -3.5],
                         cov=[[2, 3], [3, 7]])
```

Los parámetros de la distribución son el vector de medías y la matriz de covarianza, para la primera distribución estos corresponden a $= [5, 5]^\top$ y

$$\Sigma = \begin{pmatrix} 4 & 0 \\ 0 & 2 \end{pmatrix}.$$

Una vez definidas las distribuciones podemos generar números aleatoreos de las mismas, en el siguiente código se generar 1000 vectores aleatorios de las tres distribuciones.

```
X_1 = p1.rvs(size=1000)
X_2 = p2.rvs(size=1000)
X_3 = p3.rvs(size=1000)
```

Para graficar estas tres nubes de puntos se puede hacer uso del siguiente código, donde se hace uso de la librería `pandas` y `seaborn` para la generar la gráfica.

```
D = np.concatenate((X_1, X_2, X_3))
clase = [1] * 1000 + [2] * 1000 + [3] * 1000
D = np.concatenate((D, np.atleast_2d(clase).T), axis=1)
df = pd.DataFrame(D, columns=['x', 'y', 'clase'])
sns.set_style('whitegrid')
sns.relplot(data=df, kind='scatter', x='x',
            y='y', hue='clase')
```

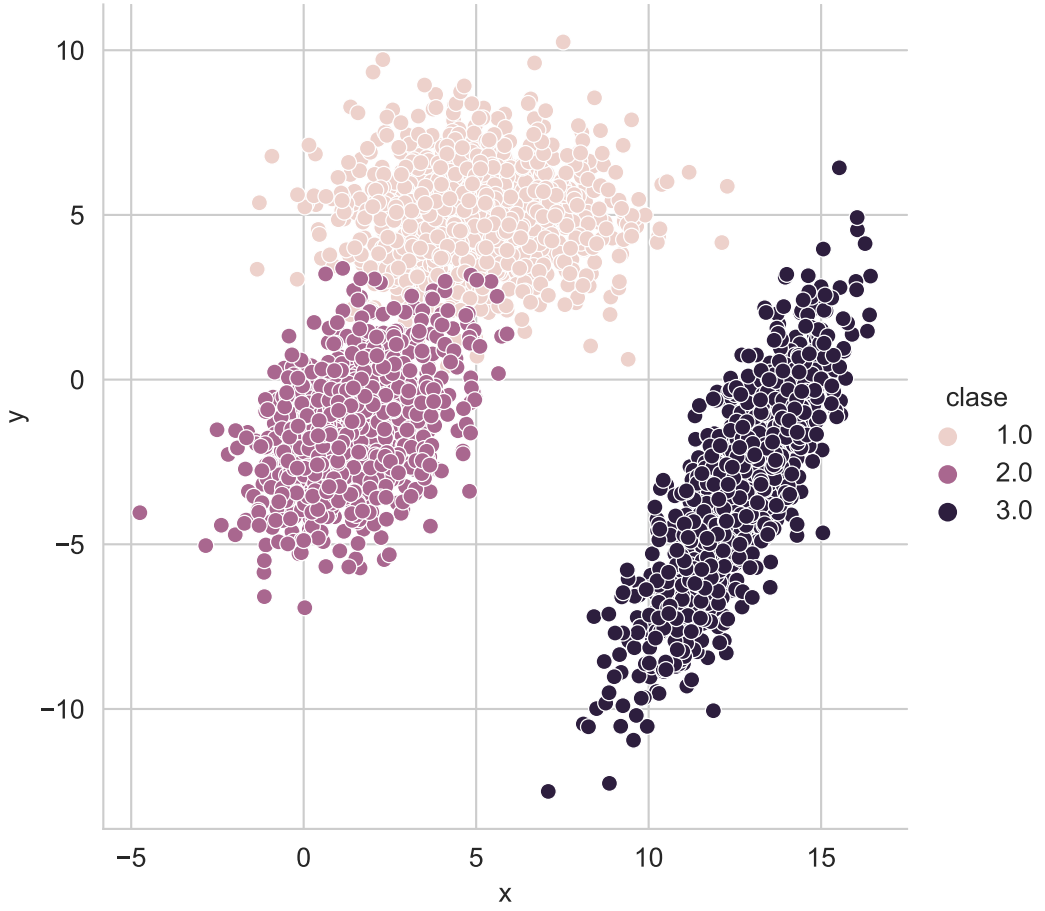


Figura 2.1: Muestras de 3 distribuciones gausianas

El resultado del código anterior se muestra en la Figura 2.1, donde se puede visualizar las tres nubes de puntos, donde el color indica la clase.

2.3.2 Predicción

En esta sección se describe el primer ejemplo del paso 4 de la metodología general (ver Sección 1.2) de los algoritmos de aprendizaje supervisado. El algoritmo f mencionado en la metodología corresponde en este ejemplo al uso del Teorema de Bayes y las distribuciones `p1`, `p2` y `p3` y los correspondientes priors.

Quitando la evidencia del Teorema de Bayes (Ecuación 2.1) se observa que $\mathbb{P}(\mathcal{Y} \mid \mathcal{X}) \propto \mathbb{P}(\mathcal{X} \mid \mathcal{Y})\mathbb{P}(\mathcal{Y})$. En el ejemplo creado se observa que $\mathbb{P}(\mathcal{Y} = 1) = \frac{1000}{3000}$, las otras probabilidades a priori tienen el mismo valor, es decir, $\mathbb{P}(\mathcal{Y} = 2) = \mathbb{P}(\mathcal{Y} = 3) = \frac{1}{3}$.

La verosimilitud está definida en las variables `p1`, `p2` y `p3`; en particular en la función `pdf`, es decir, $\mathbb{P}(\mathcal{X} \mid \mathcal{Y} = 1)$ es `p1.pdf`, $\mathbb{P}(\mathcal{X} \mid \mathcal{Y} = 2)$ corresponde a `p2.pdf` y equivalentemente `p3.pdf` es la verosimilitud cuando $\mathcal{Y} = 3$.

Utilizando esta información $\mathbb{P}(\mathcal{X} \mid \mathcal{Y})\mathbb{P}(\mathcal{Y})$ se calcula de la siguiente manera.

```
X = np.concatenate((X_1, X_2, X_3))
posterior = (np.vstack([p1.pdf(X),
                        p2.pdf(X),
                        p3.pdf(X)]) * 1 / 3).T
```

La evidencia (Ecuación 2.2) es un factor normalizador que hace que las probabilidad sume a uno, el siguiente código calcula la evidencia, $\mathbb{P}(\mathcal{X})$

```
evidencia = posterior.sum(axis=1)
```

Finalmente, $\mathbb{P}(\mathcal{Y} \mid \mathcal{X})$ se obtiene normalizando $\mathbb{P}(\mathcal{X} \mid \mathcal{Y})\mathbb{P}(\mathcal{Y})$ que se puede realizar de la siguiente manera.

```
posterior = posterior / np.atleast_2d(evidencia).T
```

La clase corresponde a la probabilidad máxima, en este caso se compara la probabilidad de $\mathbb{P}(\mathcal{Y} = 1 \mid \mathcal{X})$, $\mathbb{P}(\mathcal{Y} = 2 \mid \mathcal{X})$ y $\mathbb{P}(\mathcal{Y} = 3 \mid \mathcal{X})$; y la clase es aquella que tenga mayor probabilidad. El siguiente código muestra este procedimiento, donde el primer paso es crear un arreglo para mapear el índice a la clase. El segundo paso es seleccionar la probabilidad máxima y después transformar el índice de la probabilidad máxima a la clase.

```
clase = np.array([1, 2, 3])
indice = posterior.argmax(axis=1)
prediccion = clase[indice]
```

En la variable `prediccion` se tienen las predicciones de las clases, ahora se analizará si estas predicciones corresponden con la clase original que fue generada. Por la forma en que se generó `X` se sabe que los primero 1000 elementos pertenecen a la clase 1, los siguientes 1000 a la clase 2 y los restantes a la clase 3. A continuación se muestra el arreglo `y` que tiene esta estructura.

```
y = np.array([1] * 1000 + [2] * 1000 + [3] * 1000)
```

Teniendo las predicciones y los valores de reales de las clases, lo que se busca es visualizar los ejemplos que no fueron clasificados de manera correcta, el siguiente código muestra este procedimiento.

```
_ = [dict(x=x, y=y, error=error)
     for (x, y), error in zip(X, y != prediccion)]
df_error = pd.DataFrame(_)
sns.set_style('whitegrid')
sns.relplot(data=df_error, kind='scatter',
            x='x', y='y', hue='error')
```

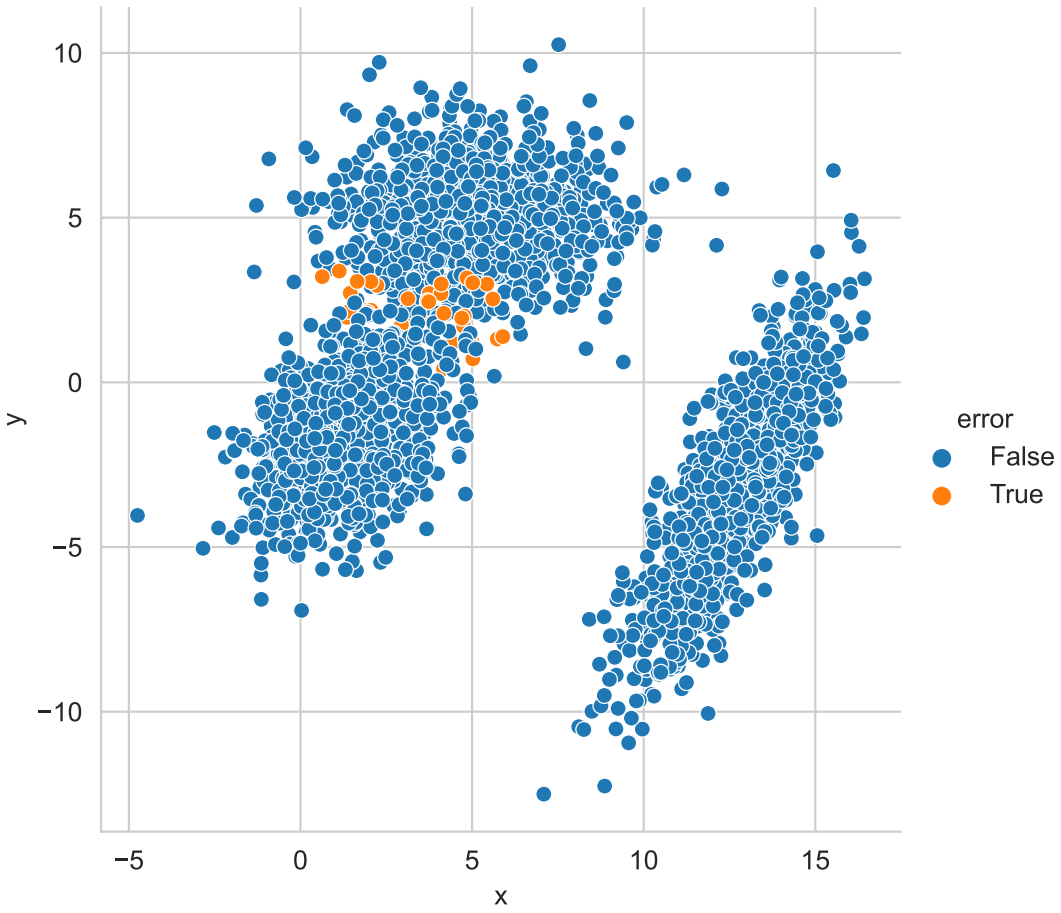


Figura 2.2: Error en problema de clasificación

La Figura 2.2 muestra todos los datos generados, en color azul se muestran aquellos datos que fueron correctamente clasificados y en color naranja (error igual a True) se muestran aquellos ejemplos donde el proceso de clasificación cometió un error.

2.4 Error de Clasificación

Este ejemplo ayuda a ilustrar el caso donde, aun teniendo el modelo perfecto, este produce errores al momento de usarlo para clasificar. Se podía asumir que este error en clasificación iba a ocurrir desde el momento que las nubes de puntos de la clase 1 y 2 se traslapan como se observa en la Figura 2.1.

El ejemplo sirve también para ilustrar el 5 paso de la metodología general (ver Sección 1.2) de los algoritmos de aprendizaje supervisado que corresponde a medir el rendimiento de un

modelo. Primero se empieza por medir el error promedio utilizando el siguiente código; donde el `error` es 0.0133.

```
error = (y != prediccion).mean()
```

La siguiente siguiente pregunta es conocer cuánto varia este error si se vuelve a realizar el muestreo de las distribuciones `p1`, `p2` y `p3`. Una manera de conocer esta variabilidad de la medición del error es calculando su **error estándar**.

El error estándar (ver Sección A.1) está definido como $\sqrt{\mathbb{V}(\hat{\theta})}$ donde $\hat{\theta}$ es el valor estimado, en este caso el `error`. El error es una variable aleatoria que sigue una distribución de Bernoulli, dado que para cada ejemplo tiene dos valores 1 que indica que en ese ejemplo el clasificador se equivocó y 0 cuando se predice la clase correcta. El parámetro de la distribución Bernoulli, p, se estima como la media entonces el error estandar de p corresponde al error estándar de la media (ver Sección A.1.1), i.e., $\sqrt{\mathbb{V}(\hat{p})} = \sqrt{\frac{\hat{p}(1-\hat{p})}{N}}$, dado que la varianza σ^2 de una distribución Bernoulli con parámetro p es $p(1-p)$. Para el ejemplo analizado el error estándar se calcula con la siguiente instrucción; teniendo un valor de 0.0021.

```
se_formula = np.sqrt(error * (1 - error) / 3000)
```

Aunque el error estándar del parámetro p de la distribución Bernoulli si se puede calcular analíticamente, se usará la técnica de Bootstrap (ver Sección A.2) para ejemplificar aquellas estadísticas donde no se puede. Esta técnica requiere generar B muestras de N elementos con remplazo de los datos. En este caso los datos son los errores entre `y` y `prediccion`. Siguiendo el método presentado en la Sección A.2 se generan los índices para generar la muestra como se observa en la primera línea del siguiente código. En la segunda línea se hacen las B repeticiones las cuales consisten en calcular $\hat{p}$. Se puede observar como se usa directamente `y` y `prediccion` junto con el arreglo de índices `s` para calcular la media del error. Finalmente se calcula la desviación estándar de `B` (tercera línea) y ese valor corresponde al error estándar.

```
S = np.random.randint(y.shape[0], size=(500, y.shape[0]))
B = [(y[s] != prediccion[s]).mean() for s in S]
se = np.std(B)
```

El error estándar, `se`, calculado es 0.0022.

El error estándar corresponde a la distribución que tiene la estimación del parámetro de interés, mediante Boostrap se simulo está distribución y con el siguiente código se puede observar su histograma, donde los datos estimados se encuentran en la lista `B`.

```
sns.set_style('whitegrid')
sns.displot(B, kde=True)
```

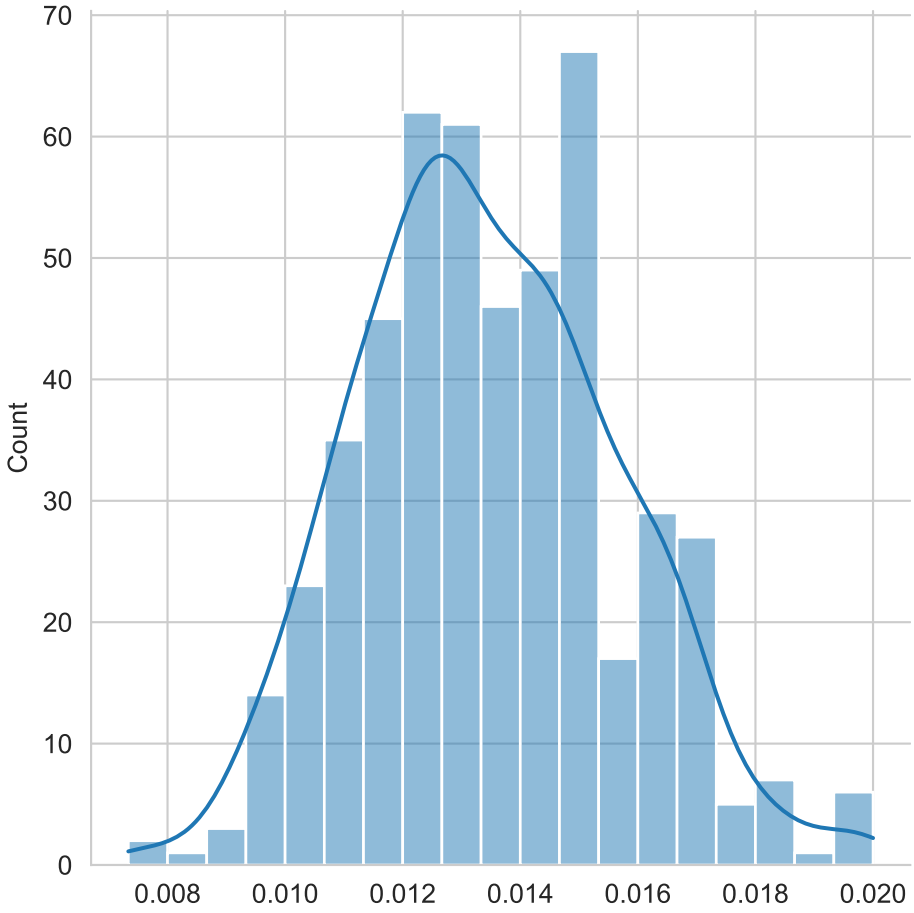


Figura 2.3: Distribución del error de clasificación

La Figura 2.3 muestra el histograma de la estimación del error en el ejemplo analizado.

2.5 Riesgo

Como es de esperarse, existen aplicaciones donde el dar un resultado equivocado tiene un mayor impacto dependiendo de la clase. Por ejemplo, en un sistema de autentificación, equivocarse dándole acceso a una persona que no tiene permisos, es mucho mas grave que no dejar entrar a una persona con los privilegios adecuados.

Una manera de incorporar el costo de equivocarse en el proceso de selección de la clase es modelarlo como una función de riesgo, es decir, seleccionar la clase que tenga el menor riesgo. Para realizar este procedimiento es necesario definir α_i como la acción que se toma al seleccionar la clase $\mathcal{Y} = i$. Entonces el riesgo esperado por tomar la acción α_i está definido por:

$$R(\alpha_i \mid x) = \sum_k \lambda_{ik} \mathbb{P}(\mathcal{Y} = k \mid \mathcal{X} = x),$$

donde λ_{ik} es el costo de tomar la acción i en la clase k.

Suponiendo una función de costo 0/1, donde el escoger la clase correcta tiene un costo 0 y el equivocarse en cualquier caso tiene un costo 1 se define como:

$$\lambda_{ik} = \begin{cases} 0 \text{ si } i = k \\ 1 \text{ de lo contrario} \end{cases}.$$

Usando la función de costo 0/1 el riesgo se define de la siguiente manera:

$$\begin{aligned} R(\alpha_i \mid x) &= \sum_k \lambda_{ik} \mathbb{P}(\mathcal{Y} = k \mid \mathcal{X} = x) \\ &= \sum_{k \neq i} \mathbb{P}(\mathcal{Y} = k \mid \mathcal{X} = x) \\ &= 1 - \mathbb{P}(\mathcal{Y}_i \mid \mathcal{X} = x). \end{aligned}$$

Recordando que $\sum_k \mathbb{P}(\mathcal{Y} = k \mid \mathcal{X} = x) = 1$. Por lo tanto en el caso de costo 0/1 se puede observar que mínimo riesgo corresponde a la clase más probable.

2.5.1 Acción nula

En algunas ocasiones es importante diseñar un procedimiento donde la acción a tomar sea el avisar que no se puede tomar una acción de manera automática y que se requiere una intervención manual.

La primera idea podría ser incrementar el número de clases y asociar una clase a la intervención manual, sin embargo en este procedimiento estaríamos incrementando la complejidad del problema. Un procedimiento mas adecuado sería incrementar el número de acciones, α de tal manera que la acción α_{K+1} corresponda a la intervención esperada, esto para cualquier problema de K clases.

La extensión del costo 0/1 para este caso estaría definida como:

$$\lambda_{ik} = \begin{cases} 0 \text{ si } i = k \\ \lambda \text{ si } i = K+1 \\ 1 \text{ de lo contrario} \end{cases},$$

donde $0 < \lambda < 1$.

Usando la definición de riesgo, el riesgo de tomar la acción α_{K+1} es

$$\begin{aligned} R(\alpha_{K+1} \mid x) &= \sum_{k}^{K} \lambda_{(K+1)k} \mathbb{P}(\mathcal{Y} = k \mid \mathcal{X} = x) \\ &= \sum_{k}^{K} \lambda \mathbb{P}(\mathcal{Y} = k \mid \mathcal{X} = x) \\ &= \lambda \sum_{k}^{K} \mathbb{P}(\mathcal{Y} = k \mid \mathcal{X} = x) = \lambda. \end{aligned}$$

2.6 Seleccionando la acción

Tomando en cuenta lo que hemos visto hasta el momento y usando como base el costo 0/1 que incluye la acción nula, se puede observar que el riesgo de seleccionar una clase está dado por $R(\alpha_i \mid x) = 1 - \mathbb{P}(\mathcal{Y} = i \mid \mathcal{X} = x)$ y el riesgo de la acción nula es $R(\alpha_{K+1} \mid x) = \lambda$.

En esta circunstancias se selecciona la clase $\widehat{y}$ si es la clase con la probabilidad máxima (i.e., $\widehat{y} = \arg\max_k \mathbb{P}(\mathcal{Y} = k \mid \mathcal{X} = x)$) y además $\mathbb{P}(\mathcal{Y} = \widehat{y} \mid \mathcal{X} = x) > 1 - \lambda$.

3 Métodos Paramétricos

El **objetivo** de la unidad es conocer las características de los modelos paramétricos y aplicar máxima verosimilitud para estimar los parámetros del modelo paramétrico en problemas de regresión y clasificación.

Paquetes usados

```
from EvoMSA.model import GaussianBayes
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_breast_cancer,\
                             load_diabetes
from scipy.stats import norm, multivariate_normal
from scipy.special import logsumexp
from matplotlib import pylab as plt
import numpy as np
import pandas as pd
import seaborn as sns
```

3.1 Introducción

Existen diferentes tipos de algoritmos que se puede utilizar para resolver problemas de aprendizaje supervisado y no supervisado. En particular, esta unidad se enfoca en presentar las técnicas que se pueden caracterizar como métodos paramétricos.

Los métodos paramétricos se identifican por asumir que los datos provienen de una distribución de la cual se desconocen los parámetros y el procedimiento es encontrar los parámetros de la distribución que mejor modelen los datos. Una vez obtenidos los parámetros se cuenta con todos los elementos para utilizar el modelo y predecir la característica para la cual fue entrenada.

3.2 Metodología

Hasta el momento se han presentado ejemplos de los pasos 4 y 5 de la metodología general (ver Sección 1.2); esto fue en la Sección 2.3.2 y en la Sección 2.4. Esta sección complementa los ejemplos anteriores al utilizar todos pasos de la metodología general de aprendizaje supervisado (ver Sección 1.2). En particular se enfoca al paso 3 que corresponde al diseño del algoritmo f que modela el fenómeno de interés utilizando los datos $\mathcal{T} \subset \mathcal{D}$.

El algoritmo f corresponde a asumir que los datos $\mathcal{D}$ provienen de una distribución F la cual tiene una serie de parámetros θ que son identificados con $\mathcal{T}$.

3.3 Estimación de Parámetros

Se inicia la descripción de métodos paramétricos presentando el procedimiento general para estimar los parámetros de una distribución. Se cuenta con un conjunto $\mathcal{D}$ donde los elementos $x \in \mathcal{D}$ son $x \in \mathbb{R}^d$. Los elementos $x \in \mathcal{D}$ tienen un distribución F, i.e., $x \sim F$, son independientes y F está definida por la función de densidad de probabilidad f_θ, que a su vez está definida por θ parámetros. Utilizando $\mathcal{D}$ el objetivo es identificar los parámetros θ que hacen observar a $\mathcal{D}$ lo más probable.

3.3.1 Verosimilitud

Una solución para maximizar el observar $\mathcal{D}$ es maximizando la verosimilitud. La verosimilitud es la función distribución conjunta de los elementos en $\mathcal{D}$, i.e., $f_\theta(x_1, x_2, \dots, x_N)$. Considerando que la muestras son independientes entonces $f_\theta(x_1, x_2, \dots, x_N) = \prod_{x \in \mathcal{D}} f_\theta(x)$. La función de verosimilitud considera la ecuación anterior como una función de los parámetros θ, es decir,

$$\mathcal{L}(\theta) = \prod_{x \in \mathcal{D}} f_\theta(x),$$

siendo el logaritmo de la verosimilitud

$$\ell(\theta) = \log \mathcal{L}(\theta) = \sum_{x \in \mathcal{D}} \log f_\theta(x).$$

3.3.2 Distribución de Bernoulli

La verosimilitud se ejemplifica con la identificación del parámetro p de una distribución Bernoulli. Una distribución Bernoulli modela dos estados, por un lado se tiene la clase negativa identificada por 0; identificando la clase positiva como 1. Entonces, la probabilidad de observar 1 es $\mathbb{P}(X = 1) = p$ y $\mathbb{P}(X = 0) = 1 - p$. Estas ecuaciones se pueden combinar para definir $f_\theta(x) = p^x(1-p)^{1-x}$.

Utilizando el logaritmo de la verosimilitud se tiene:

$$\ell(p) = \sum_{i=1}^{N} \log p^{x_i}(1-p)^{1-x_i} = \sum_{i=1}^{N} x_i \log p + (1-x_i)\log(1-p).$$

Recordando que el máximo de $\ell(p)$ se obtiene cuando $\frac{d}{dp}\ell(p) = 0$, entonces estimar p corresponde a resolver lo siguiente:

$$\begin{aligned}
\frac{d}{dp}\ell(p) &= 0 \\
\frac{d}{dp}[\sum_{i=1}^{N} x_i \log p + (1-x_i)\log(1-p)] &= 0 \\
\frac{d}{dp}[\sum_{i=1}^{N} x_i \log p + \log(1-p)(N - \sum_{i=1}^{N} x_i)] &= 0 \\
\sum_{i=1}^{N} x_i \frac{d}{dp}\log p + (N - \sum_{i=1}^{N} x_i)\frac{d}{dp}\log(1-p) &= 0 \\
\sum_{i=1}^{N} x_i \frac{1}{p} + (N - \sum_{i=1}^{N} x_i)\frac{-1}{(1-p)} &= 0
\end{aligned}$$

Realizando algunas operaciones algebraicas se obtiene:

$$\hat{p} = \frac{1}{N}\sum_{i=1}^{N} x_i.$$

3.3.3 Ejemplo: Distribución Gausiana

Esta sección sigue un camino práctico presentando el código para estimar los parámetros de una distribución Gausiana donde se conocen todos los parámetros. La distribución se usa para generar 1000 muestras y después de esta población se estiman los parámetros; de estas

manera se tienen todos los elementos para comparar los parámetros reales θ de los parámetros estimados $\hat{\theta}$.

La distribución que se usará se utilizó para generar un problema sintético (ver Sección 2.3.1) de tres clases. Los parámetros de la distribución son: $= [5, 5]^\top$ y $\Sigma = \begin{pmatrix} 4 & 0 \\ 0 & 2 \end{pmatrix}$. La siguiente instrucción se puede utilizar para generar 1000 muestras de esa distribución.

```
D = multivariate_normal(mean=[5, 5],
                        cov=[[4, 0],
                             [0, 2]]).rvs(size=1000)
```

La media estimada de los datos en `D` se calcula usando la función **`np.mean`** de la siguiente manera

```
mu = np.mean(D, axis=0)
```

donde el eje de operación es el primero que corresponde al índice 0. La media estimada es: $\hat{\mu} = [5.0836, 5.0294]^\top$ con un error estándar (`se`) de $[0.0639, 0.0456]^\top$ que se calcula con el siguiente código.

```
se = np.std(D, axis=0) / np.sqrt(1000)
```

Hasta el momento se ha estimado μ, falta por estimar Σ, que se puede realizar con la siguiente instrucción

```
cov = np.cov(D, rowvar=False)
```

donde el parámetro **`rowvar`** indica la forma en que están proporcionados los datos. La estimación da los siguientes valores $\hat{\Sigma} = \begin{pmatrix} 4.0880 & -0.0272 \\ -0.0272 & 2.0799 \end{pmatrix}$; se puede observar que son similares al parámetro con que se simularon los datos.

Siguiendo con la inercia de presentar el error estándar de cada estimación, en las siguientes instrucciones se presenta el error estándar de $\hat{\Sigma}$, el cual se calcula utilizando la técnica de bootstrap (ver Sección A.2) implementada en el siguiente código.

```
S = np.random.randint(D.shape[0],
                      size=(500, D.shape[0]))
B = [np.cov(D[s], rowvar=False) for s in S]
se = np.std(B, axis=0)
```

Se puede observar que la función `np.cov` se ejecuta utilizando la muestra indicada en la variable `s`. El error estándar (`se`) de $\hat{\Sigma}$ corresponde a $\begin{pmatrix} 0.1702 & 0.1011 \\ 0.1011 & 0.0970 \end{pmatrix}$. Se observa que los elementos fuera de la diagonal tienen un error estándar tal que el cero se encuentra en el intervalo $\hat{\Sigma} \pm se$; lo cual indica que el cero es un valor factible. Lo anterior se puede verificar tomando en cuenta que se conoce Σ y que el parámetro real es 0 para aquellos elementos fuera de la diagonal.

3.4 Metodología de Clasificación

Habiendo descrito el proceso para estimar los parámetros de una distribución, por un lado se presentó de manera teórica con la distribución Bernoulli (ver Sección **??**) y de manera práctica con una distribución Gausiana (ver Sección 3.3.3), se está en la posición de usar todos estos elementos para presentar el proceso completo de clasificación. La metodología general de aprendizaje supervisado (ver Sección 1.2) está definida por cinco pasos, estos pasos se especializan para el problema de clasificación y regresión, utilizando modelos paramétricos, de la siguiente manera.

1. Todo empieza con un conjunto de datos $\mathcal{D}$ que tiene la información del fenómeno de interés.
2. Se selecciona el conjunto $\mathcal{T} \subset \mathcal{D}$, el procedimiento se describe en la Sección 3.5.
3. Se diseña un algoritmo, f, el cual se basa en un modelo (ver Sección 3.5.1) y la estimación de sus parámetros (ver Sección 3.5.2) utilizando $\mathcal{T}$.
4. En la Sección 3.5.3 se describe el uso de f para predecir.
5. La Sección 3.5.4 muestra el procedimiento para medir el rendimiento utilizando un conjunto de prueba (ver Sección 3.5).

La metodología de clasificación se ilustra utilizando el problema sintético (ver Sección 2.3.1) de tres clases que se presentó en el Capítulo 2. Específicamente las entradas que definían a cada clase estaban en la variables `X_1`, `X_2` y `X_3`. Entonces las clases se pueden colocar en la variable `y` tal como se indica a continuación.

```
X = np.concatenate((X_1, X_2, X_3))
y = np.array([1] * 1000 + [2] * 1000 + [3] * 1000)
```

Las variables `X` y `y` contiene la información del conjunto $\mathcal{D} = (\mathcal{X}, \mathcal{Y})$ donde cada renglón de `X` es una realización de la variable aleatoria $\mathcal{X}$ y equivalentemente cada elemento en `y` es una realización de $\mathcal{Y}$.

3.5 Conjunto de Entrenamiento y Prueba

En la Sección 3.3.3 se había utilizado a $\mathcal{D}$ en el procedimiento de maximizar la verosimilitud, esto porque el objetivo en ese procedimiento era estimar los parámetros de la distribución. Pero el objetivo en aprendizaje supervisado es diseñar un algoritmo (función en este caso) que modele la relación entre $\mathcal{X}$ y $\mathcal{Y}$. Para conocer esto es necesario medir el rendimiento del algoritmo en instancias que no han sido vistas en el entrenamiento.

En consecuencia, se requieren contar con datos para medir el rendimiento, a este conjunto de datos se le conoce como el conjunto de prueba, $\mathcal{G}$. $\mathcal{G}$ se crea a partir de $\mathcal{D}$ de tal manera que $\mathcal{G} \cap \mathcal{T} = \emptyset$ y $\mathcal{D} = \mathcal{G} \cup \mathcal{T}$. La siguiente instrucción se puede utilizar para dividir la generación de estos conjuntos a partir de $\mathcal{D}$.

```
T, G, y_t, y_g = train_test_split(X, y, test_size=0.2)
```

El parámetro `test_size` indica la proporción del tamaño de conjunto $\mathcal{G}$ en relación con el conjunto $\mathcal{D}$.

3.5.1 Modelo

El inicio de métodos paramétricos es el Teorema de Bayes (Ecuación 2.1) $\mathbb{P}(\mathcal{Y} \mid \mathcal{X}) = \frac{\mathbb{P}(\mathcal{X}|\mathcal{Y})\mathbb{P}(\mathcal{Y})}{\mathbb{P}(\mathcal{X})}$ donde se usa la verosimilitud $\mathbb{P}(\mathcal{X} \mid \mathcal{Y})$ y el prior $\mathbb{P}(\mathcal{Y})$ para definir la probabilidad a posteriori $\mathbb{P}(\mathcal{Y} \mid \mathcal{X})$. En métodos paramétricos se asume que se puede modelar la verosimilitud con una distribución particular, que por lo generar es una distribución Gausiana multivariada. Es decir, la variable aleatoria $\mathcal{X}$ dado $\mathcal{Y}$ ($\mathcal{X}_{|\mathcal{Y}}$) es $\mathcal{X}_{|\mathcal{Y}} \sim \mathcal{N}(\mu_y, \Sigma_y)$, donde se observa que los parámetros de la distribución Gausiana dependen de la variable aleatoria $\mathcal{Y}$ y estos pueden ser identificados cuando $\mathcal{Y}$ tiene un valor específico.

3.5.2 Estimación de Parámetros

Dado que por definición del problema (ver Sección 2.3.1) se conoce que la verosimilitud para cada clase proviene de una Gausiana, i.e., $\mathcal{X}_{|\mathcal{Y}} \sim \mathcal{N}(\mu_y, \Sigma_y)$, en esta sección se estimarán los parámetros utilizando este conocimiento.

El primer paso en la estimación de parámetros es calcular el prior $\mathbb{P}(\mathcal{Y})$, el cual corresponde a clasificar el evento sin observar el valor de $\mathcal{X}$. Esto se puede modelar mediante una distribución Categórica con parámetros p_i donde $\sum_i^K p_i = 1$. Estos parámetros se pueden estimar utilizando la función `np.unique` de la siguiente manera

```
labels, counts = np.unique(y_t, return_counts=True)
prior = counts / counts.sum()
```

La variable `prior` contiene en el primer elemento $\mathbb{P}(\mathcal{Y} = 1) = 0.3292$, en el segundo $\mathbb{P}(\mathcal{Y} = 2) = 0.3412$ y en el tercero $\mathbb{P}(\mathcal{Y} = 3) = 0.3296$ que es aproximadamente $\frac{1}{3}$ el cual es el valor real del prior.

Siguiendo los pasos en estimación de parámetros de una Gausiana(Sección 3.3.3) se pueden estimar los parámetros para cada Gausiana dada la clase. Es decir, se tiene que estimar los parámetros μ y Σ para la clase 1, 2 y 3. Esto se puede realizar iterando por las etiquetas contenidas en la variable `labels` y seleccionando los datos en `T` que corresponden a la clase analizada, ver el uso de la variable `mask` en el slice de la línea 4 y 5. Después se inicializa una instancia de la clase `multivariate_normal` para ser utilizada en el cómputo de la función de densidad de probabilidad. El paso final es guardar las instancias de las distribuciones en la lista `likelihood`.

```
likelihood = []
for k in labels:
    mask = y_t == k
    mu = np.mean(T[mask], axis=0)
    cov = np.cov(T[mask], rowvar=False)
    likelihood_k = multivariate_normal(mean=mu, cov=cov)
    likelihood.append(likelihood_k)
```

Los valores estimados para la media, en cada clase son: $\hat{\mu}_1 = [4.9973, 5.0072]^\top$, $\hat{\mu}_2 = [1.4635, -1.4699]^\top$ y $\hat{\mu}_3 = [12.4715, -3.5220]^\top$. Para las matrices de covarianza, los valores estimados corresponden a $\hat{\Sigma}_1 = \begin{pmatrix} 3.8630 & -0.1489 \\ -0.1489 & 1.9239 \end{pmatrix}$, $\hat{\Sigma}_2 = \begin{pmatrix} 2.1296 & 1.0982 \\ 1.0982 & 3.1945 \end{pmatrix}$ y $\hat{\Sigma}_3 = \begin{pmatrix} 2.1219 & 3.2834 \\ 3.2834 & 7.5732 \end{pmatrix}$.

Estas estimaciones se pueden comparar con los parámetros reales (Sección 2.3.1). También se puede calcular su error estándar para identificar si el parámetro real, θ, se encuentra en el intervalo definido por $\hat{\theta} - 2\hat{se} \leq \hat{\theta} \leq \hat{\theta} + 2\hat{se}$ que corresponde aproximadamente al 95% de confianza asumiendo que la distribución de la estimación del parámetro es Gausiana.

3.5.3 Predicción

Una vez que se tiene la función que modela los datos, se está en condiciones de utilizarla para predecir(ver Sección 2.3.2) nuevos datos.

En esta ocasión se organiza el procedimiento de predicción en diferentes funciones, la primera función recibe los datos a predecir `X` y los componentes del modelo, que son la verosimilitud (`likelihood`) y el `prior`. La función calcula $\mathbb{P}(\mathcal{Y} = y \mid \mathcal{X} = x)$ que es la probabilidad de cada clase dada la entrada x. Se puede observar en la primera línea que se usa la función de densidad

de probabilidad (`pdf`) para cada clase y esta se multiplica por el `prior` y en la tercera línea se calcula la evidencia. Finalmente, se regresa el a posteriori.

```
def predict_prob(X, likelihood, prior):
    likelihood = [m.pdf(X) for m in likelihood]
    posterior = np.vstack(likelihood).T * prior
    evidence = posterior.sum(axis=1)
    return posterior / np.atleast_2d(evidence).T
```

La función `predict_proba` se utiliza como base para predecir la clase, para la cual se requiere el mapa entre índices y clases que se encuentra en la variable `labels`. Se observa que se llama a la función `predict_proba` y después se calcula el argumento que tiene la máxima probabilidad regresando la etiqueta asociada.

```
def predict(X, likelihood, prior, labels):
    _ = predict_prob(X, likelihood, prior)
    return labels[np.argmax(_, axis=1)]
```

3.5.4 Rendimiento

El rendimiento del algoritmo se mide en el conjunto de prueba `G`, utilizando como medida el error de clasificación (Sección 2.4). El primer paso es predecir las clases de los elementos en `G`, utilizando la función `predict` que fue diseñada anteriormente. Después se mide el error, con la instrucción de la segunda línea.

```
hy = predict(G, likelihood, prior, labels)
error = (y_g != hy).mean()
```

El error que tiene el algoritmo en el conjunto de prueba es 0.02, el cual es ligeramente superior al encontrado con el modelo ideal.

El error estándar se calcula con la siguiente instrucción el cual tiene un valor de 0.0055.

```
se_formula = np.sqrt(error * (1 - error) / y_g.shape[0])
```

3.6 Clasificador Bayesiano Ingenuo

Uno de los clasificadores mas utilizados, sencillo de implementar y competitivo, es el clasificador Bayesiano Ingenuo. En la Sección 3.5.1 se asumió que la variable aleatoria $\mathcal{X} = (\mathcal{X}_1, \mathcal{X}_2, \ldots, \mathcal{X}_d)$

dado y ($\mathcal{X}_{|y}$) es $\mathcal{X}_{|y} \sim \mathcal{N}(\mu_y, \Sigma_y)$, donde $\mu_y \in \mathbb{R}^d$, $\Sigma_y \in \mathbb{R}^{d\times d}$ y $f(\mathcal{X}_1, \mathcal{X}_2, \ldots, \mathcal{X}_d)$ es la función de densidad de probabilidad conjunta.

En el clasificador Bayesiano Ingenuo se asume que las variables $\mathcal{X}_i$ y $\mathcal{X}_j$ para $i \neq j$ son independientes, esto trae como consecuencia que $f(\mathcal{X}_1, \mathcal{X}_2, \ldots, \mathcal{X}_d) = \prod_i^d f(\mathcal{X}_i)$. Esto quiere decir que cada variable está definida como una Gausina donde se tiene que identificar μ y σ^2.

La estimación de los parámetros de estas distribuciones se puede realizar utilizando un código similar siendo la única diferencia que en se calcula σ^2 de cada variable en lugar de la covarianza Σ, esto se puede observar en la quinta línea donde se usa la función `np.var` en el primer eje. El resto del código es equivalente al usado en la Sección 3.5.2.

```
likelihood = []
for k in labels:
    mask = y_t == k
    mu = np.mean(T[mask], axis=0)
    var = np.var(T[mask], axis=0, ddof=1)
    likelihood_k = multivariate_normal(mean=mu, cov=var)
    likelihood.append(likelihood_k)
```

Los parámetros estimados en la versión ingenua son equivalentes con respecto a las medias, i.e., $\hat{\mu}_1 = [4.9973, 5.0072]^\top$, $\hat{\mu}_2 = [1.4635, -1.4699]^\top$ y $\hat{\mu}_3 = [12.4715, -3.5220]^\top$. La diferencia se puede observar en las varianzas, que a continuación se muestran como matrices de covarianza para resaltar la diferencia, i.e., $\hat{\Sigma}_1 = \begin{pmatrix} 3.8630 & 0.0000 \\ 0.0000 & 1.9239 \end{pmatrix}$, $\hat{\Sigma}_2 = \begin{pmatrix} 2.1296 & 0.0000 \\ 0.0000 & 3.1945 \end{pmatrix}$ y $\hat{\Sigma}_3 = \begin{pmatrix} 2.1219 & 0.0000 \\ 0.0000 & 7.5732 \end{pmatrix}$ se observa como los elementos fuera de la diagonal son ceros, lo cual indica la independencia entra las variables de entrada.

Finalmente, el código para predecir se utiliza el código descrito en la Sección 3.5.3 dado que el modelo está dado en las variables `likelihood` y `prior`.

El `error` del clasificador Bayesiano Ingenuo, en el conjunto de prueba, es de 0.02 y su error estándar (`se_formula`) es 0.0052.

3.7 Ejemplo: Breast Cancer Wisconsin

Esta sección ilustra el uso del clasificador Bayesiano al generar dos modelos (Clasificador Bayesiano y Bayesiano Ingenuo) del conjunto de datos de *Breast Cancer Wisconsin.* Estos datos se pueden obtener utilizando la función `load_breast_cancer` tal y como se muestra a continuación.

```
X, y = load_breast_cancer(return_X_y=True)
```

El primer paso es contar con los conjuntos de entrenamiento y prueba para poder realizar de manera completa la evaluación del proceso de clasificación. Esto se realiza ejecutando la siguiente instrucción.

```
T, G, y_t, y_g = train_test_split(X, y, test_size=0.2)
```

3.7.1 Entrenamiento

Los dos modelos que se utilizarán será el clasificador Bayesiano Gausiano y Bayesiano Ingenuo, utilizando la clase `GaussianBayes` que se explica en el Apéndice B. Las siguientes dos instrucciones inicializan estos dos clasificadores, la única diferencia es el parámetro `naive` que indica si el clasificador es ingenuo.

```
gaussian = GaussianBayes().fit(T, y_t)
naive = GaussianBayes(naive=True).fit(T, y_t)
```

3.7.2 Predicción

Habiendo definido los dos clasificadores, las predicciones del conjunto de prueba se realiza de la siguiente manera.

```
hy_gaussian = gaussian.predict(G)
hy_naive = naive.predict(G)
```

3.7.3 Rendimiento

El rendimiento de ambos clasificadores se calcula de la siguiente manera

```
error_gaussian = (y_g != hy_gaussian).mean()
error_naive = (y_g != hy_naive).mean()
```

El clasificador Bayesiano Gausiano tiene un error de 0.0614 y el error de Bayesiano Ingenuo es 0.0877. Se ha visto que el error es una variable aleatoria, entonces la pregunta es saber si esta diferencia en rendimiento es significativa o es una diferencia que proviene de la aleatoriedad de los datos.

3.8 Diferencias en Rendimiento

Una manera de ver si existe una diferencia en rendimiento es calcular la diferencia entre los dos errores de clasificación, esto es

```
naive = (y_g != hy_naive).mean()
completo = (y_g != hy_gaussian).mean()
if naive > completo:
    diff = naive - completo
else:
    diff = completo - naive
```

que tiene un valor de 0.0175. De la misma manera que se ha utilizado la técnica de bootstrap (Sección A.2) para calcular el error estándar de la media, se puede usar para estimar el error estándar de la diferencia en rendimiento. El siguiente código muestra el procedimiento para estimar este error estándar.

```
S = np.random.randint(y_g.shape[0],
                      size=(500, y_g.shape[0]))
if naive > completo:
    diff_f = lambda s: (y_g[s] != hy_naive[s]).mean() -\
                       (y_g[s] != hy_gaussian[s]).mean()
else:
    diff_f = lambda s: (y_g[s] != hy_gaussian[s]).mean() -\
                       (y_g[s] != hy_naive[s]).mean()
B = [diff_f(s) for s in S]
se = np.std(B, axis=0)
```

El error estándar de la diferencia de rendimiento es de 0.0240, una procedimiento simple para saber si la diferencia observada es significativa, es dividir la diferencia entre su error estándar dando un valor de 0.7320. En el caso que el valor absoluto fuera igual o superior a 2 se sabría que la diferencia es significativa con una confianza de al menos 95%, esto asumiendo que la diferencia se comporta como una distribución Gausiana.

El histograma de los datos que se tienen en la variable `B` se observa en la Figura 3.1. Se puede ver que la forma del histograma asemeja una distribución Gausiana y que el cero esta en el cuerpo de la Gausiana, tal y como lo confirmó el cociente que se calculado.

Se puede conocer la probabilidad de manera exacta calculando el área bajo la curva a la izquierda del cero, este sería el valor p, si este es menor a 0.05 quiere decir que se tiene una confianza mayor del 95% de que los rendimientos son diferentes. Para este ejemplo, el área se calcula con el siguiente código

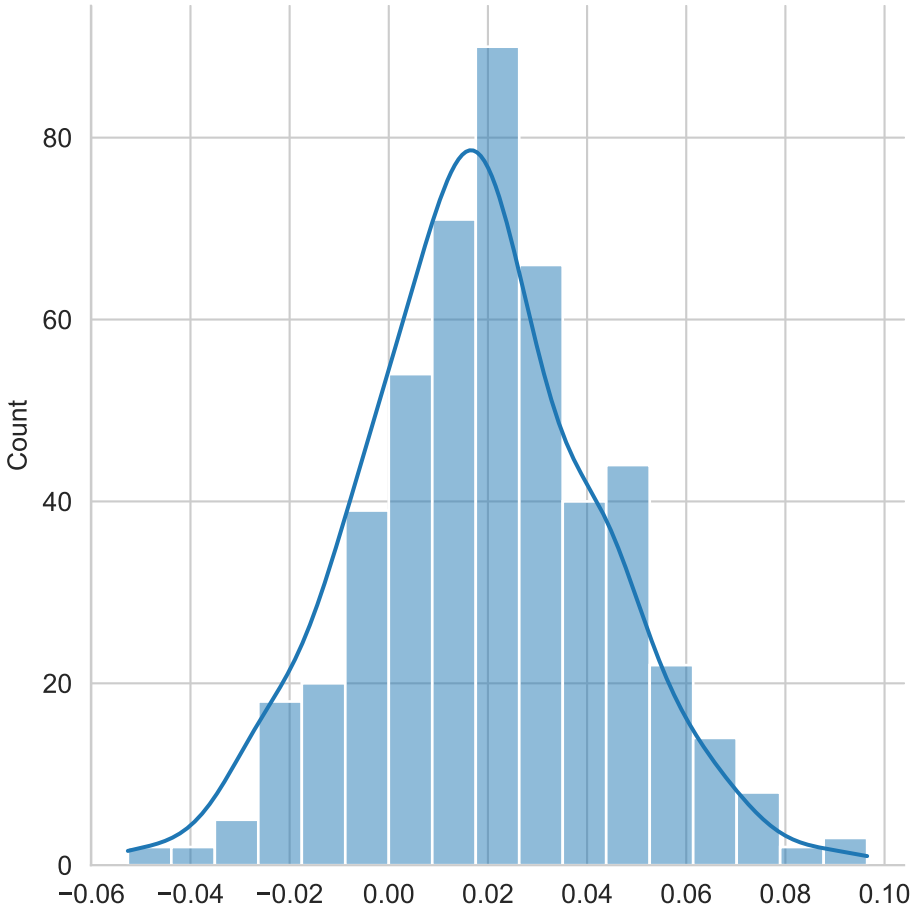


Figura 3.1: Diferencia entre Clasificadores Bayesianos

```
dist = norm(loc=diff, scale=se)
p_value = dist.cdf(0)
```

teniendo el valor de 0.2321, lo que significa que se tiene una confianza del 76% de que los dos algoritmos son diferentes considerando el error de clasificación como medida de rendimiento.

3.9 Regresión

Hasta este momento se han revisado métodos paramétricos en clasificación, ahora es el turno de abordar el problema de regresión. La diferencia entre clasificación y regresión como se describió en la Sección 1.4 es que en regresión $y \in \mathbb{R}$.

El procedimiento de regresión que se describe en esta sección es regresión de **Mínimos Cuadrados Ordinaria** (OLS -*Ordinary Least Squares*-), en el cual se asume que $y \sim \mathcal{N}(\mathbf{w} \cdot \mathbf{x} + \epsilon, \sigma^2)$, de tal manera que $y = \mathbb{E}[\mathcal{N}(\mathbf{w} \cdot \mathbf{x} + \epsilon, \sigma^2)]$.

Trabajando con $y = \mathbb{E}[\mathcal{N}(\mathbf{w} \cdot \mathbf{x} + \epsilon, \sigma^2)]$, se considera lo siguiente $y = \mathbb{E}[\mathcal{N}(\mathbf{w} \cdot \mathbf{x}, 0) + \mathcal{N}(0, \sigma^2)]$ que implica que el error ϵ es independiente de $\mathbf{x}$, lo cual se transforma en $y = \mathbf{w} \cdot \mathbf{x} + \mathbb{E}[\epsilon]$, donde $\mathbb{E}[\epsilon] = 0$. Por lo tanto $y = \mathbf{w} \cdot \mathbf{x}$.

La función de densidad de probabilidad de una Gausiana corresponde a

$$f(\alpha) = \frac{1}{\sigma\sqrt{2\pi}} \exp -\frac{1}{2}(\frac{\alpha - \mu}{\sigma})^2,$$

donde α, en el caso de regresión, corresponde a $\mathbf{w} \cdot \mathbf{x}$ (i.e., $\alpha = \mathbf{w} \cdot \mathbf{x}$).

Utilizando el método de verosimilitud el cual corresponde a maximizar

$$\begin{aligned}
\mathcal{L}(\mathbf{w}, \sigma) &= \prod_{(\mathbf{x},y)\in\mathcal{D}} f(\mathbf{w} \cdot \mathbf{x}) \\
&= \prod_{(\mathbf{x},y)\in\mathcal{D}} \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{1}{2}(\frac{\mathbf{w} \cdot \mathbf{x} - y}{\sigma})^2\right) \\
\ell(\mathbf{w}, \sigma) &= \sum_{(\mathbf{x},y)\in\mathcal{D}} \log \frac{1}{\sigma\sqrt{2\pi}} - \frac{1}{2}(\frac{\mathbf{w} \cdot \mathbf{x} - y}{\sigma})^2 \\
&= -\frac{1}{2\sigma^2} \sum_{(\mathbf{x},y)\in\mathcal{D}} (\mathbf{w} \cdot \mathbf{x} - y)^2 - N \log \frac{1}{\sigma\sqrt{2\pi}}.
\end{aligned}$$

El valor de cada parámetro se obtiene al calcular la derivada parcial con respecto al parámetro de interés, entonces se resuelven d derivadas parciales para cada uno de los coeficientes $\mathbf{w}$.

En este proceso se observar que el término $N \log \frac{1}{\sigma\sqrt{2\pi}}$ no depende de $\mathbf{w}$ entonces no afecta el máximo siendo una constante en el proceso de derivación y por lo tanto se desprecia. Lo mismo pasa para la constante $\frac{1}{2\sigma^2}$. Una vez obtenidos los parámetros w se obtiene el valor σ.

Una manera equivalente de plantear este problema es como un problema de algebra lineal, donde se tiene una matriz de observaciones X que se construyen con las variables $\mathbf{x}$ de $\mathcal{X}$, donde cada renglón de X es una observación, y el vector dependiente $\mathbf{y}$ donde cada elemento es la respuesta correspondiente a la observación.

Viéndolo como un problema de algebra lineal lo que se tiene es

$$X\mathbf{w} = \mathbf{y},$$

donde para identificar $\mathbf{w}$ se pueden realizar lo siguiente

$$X^\top X\mathbf{w} = X^\top \mathbf{y}.$$

Despejando $\mathbf{w}$ se tiene

$$\mathbf{w} = (X^\top X)^{-1} X^\top \mathbf{y}.$$

Previamente se ha presentado el error estándar de cada parámetro que se ha estimado, en caso de la regresión el error estándar (Sección A.1.3) de w_j es $\sigma\sqrt{(X^\top X)^{-1}_{jj}}$.

3.9.1 Ejemplo: Diabetes

Esta sección ilustra el proceso de resolver un problema de regresión utilizando OLS. El problema a resolver se obtiene mediante la función `load_diabetes` de la siguiente manera

```
X, y = load_diabetes(return_X_y=True)
```

El siguiente paso es generar los conjuntos de entrenamiento y prueba (Sección 3.5)

```
T, G, y_t, y_g = train_test_split(X, y, test_size=0.2)
```

Con el conjunto de entrenamiento `T` y `y_t` se estiman los parámetros de la regresión lineal tal y como se muestra a continuación

```
m = LinearRegression().fit(T, y_t)
```

Los primeros tres coeficientes de la regresión lineal son $\mathbf{w} = [29.24, -222.33, 512.92, ...]$ y $w_0 = 150.50$ lo cual se encuentran en las siguientes variables

```
w = m.coef_
w_0 = m.intercept_
```

La pregunta es si estos coeficientes son estadísticamente diferentes de cero, esto se puede conocer calculando el error estándar de cada coeficiente. Para lo cual se requiere estimar σ que corresponde a la desviación estándar del error tal y como se muestra en las siguientes instrucciones.

```
error = y_t - m.predict(T)
std_error = np.std(error)
```

El error estándar de $\mathbf{w}$ es

```
diag = np.arange(T.shape[1])
_ = np.sqrt((np.dot(T.T, T)**(-1))[diag, diag])
se = std_error * _
```

y para saber si los coeficientes son significativamente diferente de cero se calcula el cociente `m.coef_` entre `se`; teniendo los siguientes valores $[0.51, -3.89, 9.34, ...]$, para las tres primeras componentes. Se observa que hay varios coeficientes con valor absoluto menor que 2, lo cual significa que esas variables tiene un coeficiente que estadísticamente no es diferente de cero.

La predicción del conjunto de prueba se puede realizar con la siguiente instrucción

```
hy = m.predict(G)
```

Finalmente, la Figura 3.2 muestra las predicciones contra las mediciones reales. También se incluye la línea que ilustra el modelo ideal.

Complementando el ejemplo anterior, se realiza un modelo que primero elimina las variables que no son estadísticamente diferentes de cero (primera línea) y después crea nuevas variables al incluir el cuadrado, ver las líneas dos y tres del siguiente código.

```
mask = np.fabs(m.coef_ / se) >= 2
T = np.concatenate((T[:, mask], T[:, mask]**2), axis=1)
G = np.concatenate((G[:, mask], G[:, mask]**2), axis=1)
```

Se observa que la identificación de los coeficientes $\mathbf{w}$ sigue siendo lineal aun y cuando la representación ya no es lineal por incluir el cuadrado. Siguiendo los pasos descritos previamente, se inicializa el modelo y después se realiza la predicción.

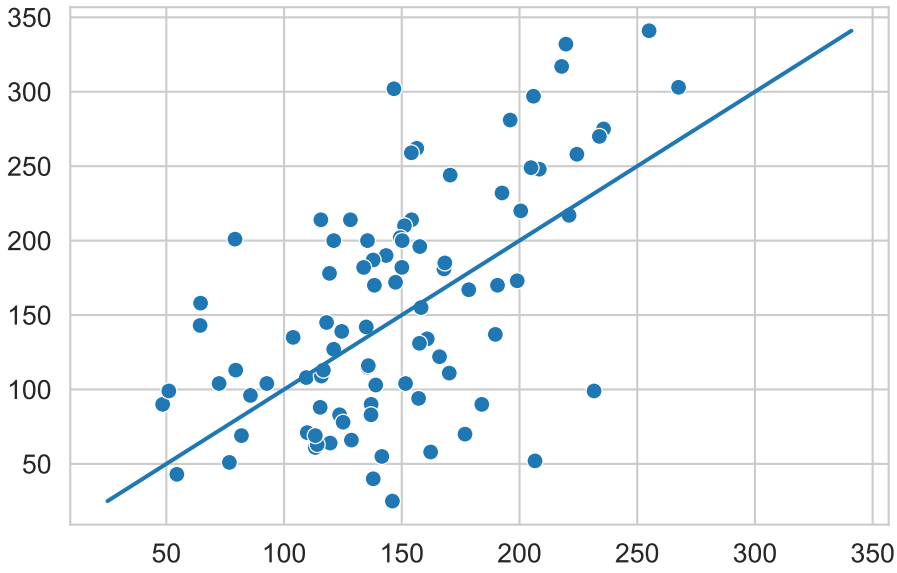


Figura 3.2: Regresión Lineal

```
m2 = LinearRegression().fit(T, y_t)
hy2 = m2.predict(G)
```

En este momento se compara si la diferencia entre el error cuadrático medio, del primer y segundo modelo, la diferencia es -13.9616 indicando que el primer modelo es mejor.

```
diff = ((y_g - hy2)**2).mean() -  ((y_g - hy)**2).mean()
```

Para comprobar si esta diferencia es significativa se calcula el error estándar, utilizando bootstrap (Sección A.2) tal y como se muestra a continuación.

```
S = np.random.randint(y_g.shape[0],
                      size=(500, y_g.shape[0]))
B = [((y_g[s] - hy2[s])**2).mean() -
      ((y_g[s] - hy[s])**2).mean()
     for s in S]
se = np.std(B, axis=0)
```

Finalmente, se calcula el área bajo la curva a la izquierda del cero, teniendo un valor de 0.4999 lo cual indica que los dos modelos son similares. En este caso se prefiere el modelo más simple porque se observar que incluir el cuadrado de las variables no contribuye a generar un mejor model. El área bajo la curva se calcula con el siguiente código.

```
dist = norm(loc=diff, scale=se)
p_value = dist.cdf(0)
```

4 Rendimiento

El **objetivo** es contrastar las características de diferentes medidas de rendimiento en aprendizaje supervisado así como simular un procedimiento de aprendizaje supervisado.

Paquetes usados

```
from EvoMSA.model import GaussianBayes
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_breast_cancer,\
                             load_diabetes
from sklearn.metrics import recall_score
from sklearn.model_selection import train_test_split, KFold
from sklearn import metrics
import numpy as np
import pandas as pd
from matplotlib import pylab as plt
import seaborn as sns
```

4.1 Introducción

Es importante conocer el rendimiento del algoritmo de aprendizaje computacional desarrollado. En aprendizaje supervisado la medición se hace mediante el conjunto de prueba, $\mathcal{G}$, mientras que en aprendizaje no supervisado es posible utilizar el conjunto de entrenamiento $\mathcal{T}$ o utilizar un conjunto de prueba. Es importante notar que aunque en el proceso de entrenamiento puede usar una función de rendimiento para estimar o encontrar el algoritmo que modela los datos, es importante complementar esta medición con otras funciones de rendimiento. Esta unidad describe algunas de las medidas más utilizadas para medir el rendimiento de algoritmos de clasificación y regresión.

4.2 Clasificación

En clasificación existen diferentes medidas de rendimiento, algunas de ellas son exactitud (*accuracy*), precisión (*precision*), recall, y F_1, entre otras. Sebastiani (2015) describe de manera axiomática algunas de estas medidas y se dan recomendaciones en general sobre medidas de rendimiento para clasificadores.

Varias de las medidas de rendimiento toman como insume la **Tabla de Confusión** (Tabla 4.1), la cual contiene la información del proceso de clasificación. La siguiente tabla muestra la estructura de esta tabla para un problema binario, donde se tiene una clase positiva identificada con p y una clase negativa (n). La variable y indica las clases reales y la variable $\hat{y}$ representa la estimación (predicción) hecha por el clasificador. Adicionalmente, la tabla se puede extender a K clases siguiendo la misma estructura; la diagonal contienen los elementos correctamente identificados y los elementos fuera de la diagonal muestra los errores.

Tabla 4.1: Tabla de Confusión

	$\hat{y} = p$	$\hat{y} = n$
$y = p$	Verdaderos Pos.	Falsos Neg.
$y = n$	Falsos Pos.	Verdaderos Neg.

La tabla se puede ver como valores nominales, es decir contar el número de ejemplos clasificados como verdaderos positivos o como proporción de tal manera que las cuatro celdas sumen 1. En esta descripción se asume que son proporcionen, esto porque se seguirá una interpretación probabilística descrita en este artículo para presentar las diferentes medidas de rendimiento.

Viendo la Tabla 4.1 como una proporción y combinando con la interpretación probabilística la tabla quedaría de la siguiente manera.

Tabla 4.2: Tabla de Confusión como Proporción

	$\hat{y} = p$	$\hat{y} = n$
$y = p$	$\mathbb{P}(y = p, \hat{y} = p)$	$\mathbb{P}(y = p, \hat{y} = n)$
$y = n$	$\mathbb{P}(y = n, \hat{y} = p)$	$\mathbb{P}(y = n, \hat{y} = n)$

Partiendo de la Tabla 4.2 se puede calcular la probabilidad marginal de cualquier variable y también las probabilidades condicionales, por ejemplo $\mathbb{P}(y = p) = \sum_k \mathbb{P}(y = p, \hat{y} = k)$ que es la suma de los elementos del primer renglón de la tabla anterior.

4.2.1 Error

Se empieza la descripción con el error de clasificación (Sección 2.4) el cual es la proporción de errores y se puede definir como

$$\mathsf{error}(y, \hat{y}) = 1 - \mathsf{accuracy}(y, \hat{y}).$$

4.2.2 Exactitud (*Accuracy*)

El error se define mediante la exactitud. La exactitud es la proporción de ejemplos correctamente clasificados, utilizando la notación de la tabla de confusión quedaría como:

$$\mathsf{accuracy}(y, \hat{y}) = \mathbb{P}(y = p, \hat{y} = p) + \mathbb{P}(y = n, \hat{y} = n).$$

Una manera equivalente de ver la exactitud es utilizando la probabilidad condicional, es decir,

$$\begin{aligned}\mathsf{accuracy}(y, \hat{y}) = \mathbb{P}(\hat{y} = p \mid y = p)\mathbb{P}(y = p) \\ + \mathbb{P}(\hat{y} = n \mid y = n)\mathbb{P}(y = n).\end{aligned}$$

Esta manera ayuda a entender el caso cuando se tiene una clase con muchos ejemplos, e.g., $\mathbb{P}(y = p) \gg \mathbb{P}(y = n)$, en ese caso se ve que la exactitud está dominado por el primer término, i.e., $\mathbb{P}(\hat{y} = p \mid y = p)\mathbb{P}(y = p)$. En este caso, la manera trivial de optimizar la exactitud es crear un clasificador que siempre regrese la clase p. Por esta razón la exactitud no es una medida adecuada cuando las clases son desbalanciadas, es buena medida cuando $\mathbb{P}(y = p) \approx \mathbb{P}(y = n)$.

4.2.3 Recall

La siguiente medida de rendimiento es el recall, este calcula la probabilidad de ejemplos correctamente clasificados como p dados todos los ejemplos que se tienen de la clase p. En base a esta ecuación se puede observar que un algoritmo trivial con el máximo valor de recall solamente tiene que predecir como clase p todos los elementos.

La segunda ecuación ayuda a medir en base de la tabla de confusión.

$$\begin{aligned}\mathsf{recall}_p(y, \hat{y}) &= \mathbb{P}(\hat{y} = p \mid y = p) \\ &= \frac{\mathbb{P}(\hat{y} = p, y = p)}{\mathbb{P}(y = p)}\end{aligned} \tag{4.1}$$

4.2.4 Precisión (*Precision*)

La precisión complementa el recall, al calcular la probabilidad de los ejemplos correctamente clasificados como p dadas las predicciones de los ejemplos. Es decir, en la probabilidad condicional se observa que se conocen las predicciones positivas y de esas predicciones se mide si estas son correctamente clasificadas. Basándose en esto, se puede ver que una manera de generar un algoritmo competitivo en esta media corresponde a predecir la clase solo cuando exista una gran seguridad de la clase.

$$\begin{aligned} \mathsf{precision}_p(y, \hat{y}) &= \mathbb{P}(y = p \mid \hat{y} = p) \\ &= \frac{\mathbb{P}(y = p, \hat{y} = p)}{\mathbb{P}(\hat{y} = p)} \end{aligned} \tag{4.2}$$

4.2.5 F_β

Finalmente, una manera de combinar el recall (Ecuación 4.1) con la precisión (Ecuación 4.2) es la medida F_β, es probable que esta medida se reconozca más cuando $\beta = 1$. La idea de β es ponderar el peso que se le quiere dar a la precisión con respecto al recall.

$$F_\beta^p(y, \hat{y}) = (1 + \beta^2) \frac{\mathsf{precision}_p(y, \hat{y}) \cdot \mathsf{recall}_p(y, \hat{y})}{\beta^2 \cdot \mathsf{precision}_p(y, \hat{y}) + \mathsf{recall}_p(y, \hat{y})} \tag{4.3}$$

4.2.6 Medidas Macro

En las definiciones de precisión (Ecuación 4.2), recall (Ecuación 4.1) y F_β (Ecuación 4.3) se ha usado un subíndice y superíndice con la letra p esto es para indicar que la medida se está realizando con respecto a la clase p. Esto ayuda también a ilustrar que en un problema de K clases se tendrán K diferentes medidas de precisión, recall y F_β; cada una de esas medidas corresponde a cada clase.

En ocasiones es importante tener solamente una medida que englobe el rendimiento en el caso de los tres rendimientos que se han mencionado, se puede calcular su versión macro que es la media de la medida. Esto es para un problema de K clases la precisión, recall y F_β se definen de la siguiente manera.

$$\mathsf{macro\text{-}precision}(y, \hat{y}) = \frac{1}{K} \sum_k \mathsf{precision}_k(y, \hat{y}),$$

$$\mathsf{macro\text{-}recall}(y, \hat{y}) = \frac{1}{K} \sum_k \mathsf{recall}_k(y, \hat{y}),$$

$$\text{macro-}F_\beta(y, \hat{y}) = \frac{1}{K} \sum_k F_\beta^k(y, \hat{y}).$$

4.2.7 Entropía Cruzada

Una función de costo que ha sido muy utilizada en redes neuronales y en particular en aprendizaje profundo es la **Entropía Cruzada** (Cross Entropy) que para una distribución discreta se define como: $H(P, Q) = -\sum_x P(x) \log Q(x)$.

Para cada ejemplo x se tiene $\mathbb{P}(\mathcal{Y} = k \mid \mathcal{X} = x)$ y el clasificador predice $\hat{\mathbb{P}}(\mathcal{Y} = k \mid \mathcal{X} = x)$. Utilizando estas definiciones se puede decir que $P = \mathbb{P}$ y $Q = \hat{\mathbb{P}}$ en la definición de entropía cruzada; entonces

$$H(\mathbb{P}(\mathcal{Y} \mid \mathcal{X} = x), \hat{\mathbb{P}}(\mathcal{Y} \mid \mathcal{X} = x)) = \\ -\sum_k^K \mathbb{P}(\mathcal{Y} = k \mid \mathcal{X} = x) \log \hat{\mathbb{P}}(\mathcal{Y} = k \mid \mathcal{X} = x).$$

Finalmente la medida de rendimiento quedaría como $\sum_x H(\mathbb{P}(\mathcal{Y} \mid \mathcal{X} = x), \hat{\mathbb{P}}(\mathcal{Y} \mid \mathcal{X} = x))$.

4.2.8 Área Bajo la Curva *ROC*

El área bajo la curva *ROC* (*Relative Operating Characteristic*) es una medida de rendimiento que también está pasada en la probabilidad a posteriori $\mathbb{P}(\mathcal{Y} \mid \mathcal{X})$ con la característica de que la clase se selecciona en base a un umbral ρ. Es decir, dado un ejemplo x, este ejemplo pertenece a la clase p si $\mathbb{P}(\mathcal{Y} = p \mid \mathcal{X} = x) \geq \rho$.

Se observa que modificando el umbral ρ se tienen diferentes tablas de confusión, para cada tabla de confusión posible se calcula la tasa de verdaderos positivos (TPR) que corresponde al recall (Sección 4.2.3), i.e., $\mathbb{P}(\hat{\mathcal{Y}} = p \mid \mathcal{Y} = p)$, y la tasa de falsos positivos (FPR) que es $\mathbb{P}(\hat{\mathcal{Y}} = p \mid \mathcal{Y} = n)$. Cada par de TPR y FPR representan un punto de la curva *ROC*. El rendimiento corresponde al área debajo de la curva delimitada por los pares TPR y FPR.

4.2.9 Ejemplo

El ejemplo de Breast Cancer Wisconsin (Sección 3.7) se utiliza para ilustrar el uso de la medidas de rendimiento presentadas hasta el momento.

```
D, y = load_breast_cancer(return_X_y=True)
T, G, y_t, y_g = train_test_split(D, y,
                                  random_state=0,
                                  test_size=0.2)
gaussian = GaussianBayes().fit(T, y_t)
hy_gaussian = gaussian.predict(G)
```

El clasificador Gausiano tiene un `accuracy` de 0.8947, el cual se puede calcular con el siguiente código.

```
accuracy = metrics.accuracy_score(y_g, hy_gaussian)
```

Las medidas de `recall`, `precision` y `f1` se presentan en la Tabla 4.3, en la última columna se presenta el macro de cada una de las medidas.

```
recall = metrics.recall_score(y_g, hy_gaussian,
                              average=None)
precision = metrics.precision_score(y_g, hy_gaussian,
                                    average=None)
f1 = metrics.f1_score(y_g, hy_gaussian,
                      average=None)
```

Tabla 4.3: Rendimiento

	$y = 0$	$y = 1$	Macro
`recall`	0.8298	0.9403	0.8850
`precision`	0.9070	0.8873	0.8972
`f1`	0.8667	0.9130	0.8899

Por otro lado la `entropia` cruzada es 2.1071 que se puede calcular con el siguiente código.

```
prob = gaussian.predict_proba(G)
entropia = metrics.log_loss(y_g, prob)
```

Complementando la información de las medidas que se calculan mediante la posteriori se encuentra la curva ROC, la cual se puede calcular con el siguiente código y se muestra en la Figura 4.1

```
fpr, tpr, thresholds = metrics.roc_curve(y_g, prob[:, 1])
df = pd.DataFrame(dict(FPR=fpr, TPR=tpr))
sns.set_style('whitegrid')
fig = sns.lineplot(df, x='FPR', y='TPR')
```

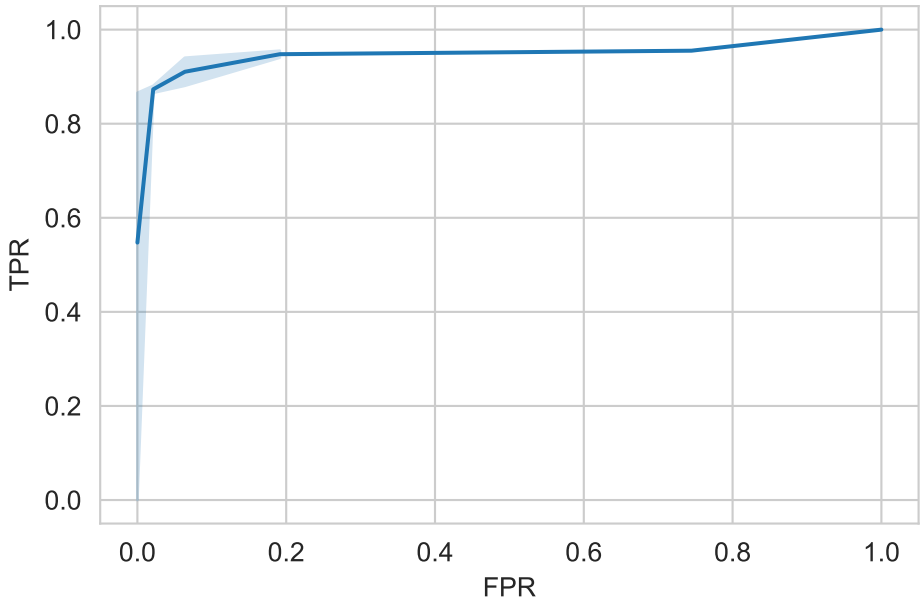


Figura 4.1: ROC

Teniendo un valor de área bajo la curva (`auc_score`) de 0.9540 que se obtuvo de la siguiente manera.

```
auc_score = metrics.roc_auc_score(y_g, prob[:, 1])
```

4.3 Regresión

Con respecto a regresión las siguientes funciones son utilizadas como medidas de rendimiento.

Error cuadrático medio (Mean Square Error):

$$\mathsf{mse}(y, \hat{y}) = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2. \tag{4.4}$$

Error absoluto medio (Mean Absolute Error):

$$\mathsf{mae}(y, \hat{y}) = \frac{1}{N} \sum_{i=1}^{N} \mid y_i - \hat{y}_i \mid . \tag{4.5}$$

Media del porcentaje de error absoluto:

$$\mathsf{mape}(y, \hat{y}) = \frac{1}{N} \sum_{i=1}^{N} \mid \frac{y_i - \hat{y}_i}{y_i} \mid . \tag{4.6}$$

La proporción de la varianza explicada por el modelo:

$$R^2(y, \hat{y}) = 1 - \frac{\sum_{i=1}^{N}(y_i - \hat{y}_i)^2)}{\sum_{i=1}^{N}(y_i - \bar{y}_i)^2)}. \tag{4.7}$$

4.3.1 Ejemplo

Las medidas anteriores se ejemplifican utilizando el ejemplo de diabetes(#sec-diabetes) que se puede descargar y modelar mediante OLS (Sección 3.9) de la siguiente manera.

```
X, y = load_diabetes(return_X_y=True)
T, G, y_t, y_g = train_test_split(X, y,
                                  random_state=0,
                                  test_size=0.2)
m = LinearRegression().fit(T, y_t)
```

La predicción en el conjunto de prueba sería:

```
hy = m.predict(G)
```

Las diferentes medidas de rendimiento para problemas de regresión se puede calcular de la siguiente manera.

El error cuadrático medio (Ecuación 4.4), `mse` corresponde a

```
mse = metrics.mean_squared_error(y_g, hy)
```

y tienen un valor de 3424.2593.

El error absoluto medio (Ecuación 4.5), `mae`, tiene un valor de 46.1736 calculado de la siguiente manera

```
mae = metrics.mean_absolute_error(y_g, hy)
```

La media del porcentaje de error absoluto (Ecuación 4.6), `mape`, es 0.3805 obtenido con el siguiente código

```
mape = metrics.mean_absolute_percentage_error(y_g, hy)
```

Finalmente, la varianza explicada por el modelo R^2 (Ecuación 4.7), `r2`, es 0.3322

```
r2 = metrics.r2_score(y_g, hy)
```

4.4 Conjunto de Validación y Validación Cruzada

Antes de inicia la descripción de otro algoritmo para la selección de características es necesario describir otro conjunto que se utiliza para optimizar los hiperparámetros del algoritmo de aprendizaje. Previamente se describieron los conjuntos de Entrenamiento y Prueba (Sección 3.5), i.e., $\mathcal{T}$ y $\mathcal{G}$. En particular estos conjuntos se definieron utilizando todos los datos $\mathcal{D}$ con lo que se especifica el problema.

La mayoría de algoritmos de aprendizaje tiene hiperparámetros que pueden ser ajustados para optimizar su comportamiento al conjunto de datos que se está analizando. Estos hiperparámetros pueden estar dentro del algoritmo o pueden ser modificaciones al conjunto de datos para adecuarlos al algoritmo. El segundo caso es el que se analizará en esta unidad. Es decir, se seleccionarán las variables que facilitan el aprendizaje.

Para optimizar los parámetros es necesario medir el rendimiento del algoritmo, es decir, observar como se comporta el algoritmo en el proceso de predicción. La manera trivial sería utilizar el conjunto de prueba $\mathcal{G}$ para medir el rendimiento. Pero es necesario recordar que este conjunto no debe ser visto durante el aprendizaje y la optimización de los parámetros es parte de ese proceso. Si se usara $\mathcal{G}$ entonces dejaría de ser el conjunto de prueba y se tendría que seleccionar otro conjunto de prueba.

Entonces para optimizar los parámetros del algoritmo se selecciona del conjunto de entrenamiento, i.e., $\mathcal{T}$, el **conjunto de validación**, $\mathcal{V}$. Este conjunto tiene la característica que $\mathcal{T} \cap \mathcal{V} \cap \mathcal{G} = \emptyset$ y $\mathcal{T} \cup \mathcal{V} \cup \mathcal{G} = \mathcal{D}$. Una manera de realizar estos es seleccionar primeramente el conjunto de prueba $\mathcal{G}$ y de los datos restantes generar los conjuntos de entrenamiento $\mathcal{T}$ y validación $\mathcal{V}$.

Para ejemplificar esta idea se utiliza el ejemplo de Breast Cancer Wisconsin (Sección 3.7) utilizando un Clasificador Bayesiano donde el hiperparámetro es si se utilizar un Bayesiano Ingenuo o se estima la matriz de covarianza.

El primer paso es obtener los datos del problema, lo cual se muestra en la siguiente instrucción.

```
D, y = load_breast_cancer(return_X_y=True)
```

Con los datos $\mathcal{D}$ se genera el conjunto de prueba $\mathcal{G}$ y los datos para estimar los parámetros y optimizar los hiperparámetros del algoritmo. En la variable `T` se tiene los datos para encontrar el algoritmo y en `G` se tiene el conjunto de prueba.

```
T, G, y_t, y_g = train_test_split(D, y,
                                  random_state=0,
                                  test_size=0.2)
```

Los datos de entrenamiento y validación se generan de manera equivalente tal como se muestra en la siguiente instrucción. El conjunto de validación ($\mathcal{V}$) se encuentra en la variable `V` y la variable dependiente en `y_v`.

```
T, V, y_t, y_v = train_test_split(T, y_t,
                                  random_state=0,
                                  test_size=0.3)
```

En este momento ya se tienen todos los elementos para medir el rendimiento de cada hiperparámetro. Empezando por el clasificador con la matriz de covarianza completa. El recall en ambas clases es [0.9615, 0.8824].

```
gaussian = GaussianBayes().fit(T, y_t)
hy_gaussian = gaussian.predict(V)
recall = recall_score(y_v, hy_gaussian, average=None)
```

La segunda opción es utilizar un clasificador Bayesiano Ingenuo, el cual se especifica con el parámetro `naive` tal y como se muestra en las siguientes instrucciones. El recall en las dos clases es [0.8654, 0.9765].

```
ingenuo = GaussianBayes(naive=True).fit(T, y_t)
hy_ingenuo = ingenuo.predict(V)
score = recall_score(y_v, hy_ingenuo, average=None)
```

Comparando el rendimiento de los dos hiperparámetros se observa cual de los dos modelos obtiene el mejor rendimiento. Con el fin de completar el ejemplo se describe calcular el rendimiento en $\mathcal{G}$ del algoritmo con la matriz de covarianza completa. Este algoritmo tiene un rendimiento de [0.8298, 0.9403] que se puede calcular con el siguiente código.

```
gaussian = GaussianBayes().fit(np.concatenate((T, V)),
                               np.concatenate((y_t, y_v)))
hy_gaussian = gaussian.predict(G)
score = recall_score(y_g, hy_gaussian, average=None)
```

4.4.1 k-Iteraciones de Validación Cruzada

Cuando se cuenta con pocos datos para medir el rendimiento del algoritmo es común utilizar la técnica de *k-fold cross-validation* la cual consiste en partir k veces el conjunto de entrenamiento para generar k conjuntos de entrenamiento y validación.

La idea se ilustra con la siguiente tabla, donde se asume que los datos son divididos en 5 bloques ($k = 5$), cada columna de la tabla ilustra los datos de ese bloque. Si los datos se dividen en $k = 5$ bloques, entonces existen k iteraciones que son representadas por cada renglón de la siguiente tabla, quitando el encabezado de la misma. La letra en cada celda identifica el uso que se le dará a esos datos en la respectiva iteración, es decir, $\mathcal{T}$ representa que se usará como conjunto de entrenamiento y $\mathcal{V}$ se usa para identificar aquellos datos que se usarán como conjunto de validación.

La idea es entrenar y probar el rendimiento del algoritmo k veces usando las particiones en cada renglón. Es decir, la primera vez se usan los datos de la primera columna como el conjunto de validación, y el resto de columnas, $[2, 3, 4, 5]$, como conjunto de entrenamiento para estimar los parámetros del algoritmo. En la segunda iteración se usan los datos del segundo renglón donde se observa que los datos en la cuarta columna corresponden al conjunto de validación y los datos en las columnas $[1, 2, 3, 5]$ son usados como conjunto de prueba. Las iteraciones siguen hasta que todos los datos fueron utilizados en una ocasión como conjunto de validación.

1	2	3	4	5
$\mathcal{V}$	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$
$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{V}$	$\mathcal{T}$
$\mathcal{T}$	$\mathcal{T}$	$\mathcal{V}$	$\mathcal{T}$	$\mathcal{T}$
$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{V}$
$\mathcal{T}$	$\mathcal{V}$	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$

Se utiliza el mismo problema para medir el rendimiento del hiperparámetro del clasificador Gausiano. Lo primero es seleccionar el conjunto de prueba ($\mathcal{G}$) que se realiza con el siguiente código.

```
T, G, y_t, y_g = train_test_split(D, y,
                                  random_state=0,
                                  test_size=0.2)
```

La validación cruzada con k-iteraciones se puede realizar con la clase `KFold` de la siguiente manera. La primera línea crear una variable para guardar el rendimiento. En la segunda línea se inicializa el procedimiento indicando que los datos sean tomados al azar. Después se realiza el ciclo con las k iteraciones, para cada iteración se genera un índice `ts` que indica cuales son los datos del conjunto de entrenamiento y `vs` que corresponde a los datos de validación. Se estiman los parámetros usando `ts` tal y como se observa en la cuarta línea. Habiendo estimado los parámetros se predicen los datos del conjunto de validación (5 línea), se mide el recall en todas las clases y se guarda en la lista `perf`. Al final se calcula la media de los k rendimientos medidos, teniendo un valor de [0.8903, 0.9174].

```
perf = []
kfold = KFold(shuffle=True, random_state=0)
for ts, vs in kfold.split(T):
    gaussian = GaussianBayes().fit(T[ts], y_t[ts])
    hy_gaussian = gaussian.predict(T[vs])
    _ = recall_score(y_t[vs], hy_gaussian, average=None)
    perf.append(_)
perf = np.mean(perf, axis=0)
```

Un procedimiento equivalente se realiza para el caso del clasificador Bayesiano Ingenuo tal y como se muestra a continuación. La media del recall en las clases es [0.8260, 0.9785] Se observa que el clasificador Bayesiano con la matriz de covarianza tiene un mejor rendimiento en validación que el clasificador Bayesiano Ingenuo. El último paso sería calcular el rendimiento en el conjunto $\mathcal{G}$ lo cual fue presentado anteriormente.

```
perf = []
kfold = KFold(shuffle=True)
for ts, vs in kfold.split(T):
    gaussian = GaussianBayes(naive=True).fit(T[ts], y_t[ts])
    hy_gaussian = gaussian.predict(T[vs])
    _ = recall_score(y_t[vs], hy_gaussian, average=None)
    perf.append(_)
perf = np.mean(perf, axis=0)
```

5 Reducción de Dimensión

El **objetivo** de la unidad es aplicar técnicas de reducción de dimensionalidad, para mejorar el aprendizaje y para visualizar los datos

Paquetes usados

```
from scipy.stats import multivariate_normal, norm, kruskal
from sklearn.datasets import load_diabetes,\
                             load_breast_cancer,\
                             load_iris,\
                             load_wine,\
                             load_digits
from sklearn.feature_selection import f_regression,\
                                      SelectKBest,\
                                      SelectFromModel,\
                                      SequentialFeatureSelector
from sklearn.model_selection import KFold, train_test_split
from sklearn.metrics import recall_score, make_scorer, r2_score
from sklearn.linear_model import LinearRegression
from sklearn.naive_bayes import GaussianNB
from sklearn.preprocessing import StandardScaler
from sklearn import decomposition
from EvoMSA.model import GaussianBayes
import umap
import numpy as np
import pandas as pd
from matplotlib import pylab as plt
import matplotlib as mpl
import seaborn as sns
```

5.1 Introducción

Habiendo descrito problemas de clasificación y regresión, podemos imaginar que existen ocasiones donde las variables que describen al problema no contribuyen dentro de la solución, o que su aporte está dado por otras componentes dentro de la descripción. Esto trae como consecuencia, en el mejor de los caso, que el algoritmo tenga un mayor costo computacional o en un caso menos afortunado que el algoritmo tenga un rendimiento menor al que se hubiera obtenido seleccionado las variables. Es pertinente mencionar que el caso contrario correspondiente al incremento del número de variables es también un escenario factible y se abordará en otra ocasión.

Existen diferentes maneras para reducir la dimensión de un problema, es decir, transformar la representación original $x \in \mathbb{R}^d$ a una representación $\hat{x} \in \mathbb{R}^m$ donde $m < d$. El objetivo es que la nueva representación $\hat{x}$ contenga la información necesaria para realizar la tarea de clasificación o regresión. También otro objetivo sería reducir a $\mathbb{R}^2$ o $\mathbb{R}^3$ de tal manera que se pueda visualizar el problema. En este último caso el objetivo es que se mantengan las características de los datos en $\mathbb{R}^d$ en la reducción.

Esta descripción inicia con una metodología de selección basada en calcular estadísticas de los datos y descartar aquellas que no proporcionan información de acuerdo a la estadística.

5.2 Selección de Variables basadas en Estadísticas

Se utilizará el problema sintético (Sección 2.3.1) de tres clases para describir el algoritmo de selección. Este problema está definido por tres Distribuciones Gausianas donde se generan tres muestras de 1000 elementos cada una utilizando el siguiente código.

```
p1 = multivariate_normal(mean=[5, 5],
                         cov=[[4, 0], [0, 2]])
p2 = multivariate_normal(mean=[1.5, -1.5],
                         cov=[[2, 1], [1, 3]])
p3 = multivariate_normal(mean=[12.5, -3.5],
                         cov=[[2, 3], [3, 7]])
X_1 = p1.rvs(size=1000)
X_2 = p2.rvs(size=1000)
X_3 = p3.rvs(size=1000)
```

Estas tres distribuciones representan el problema de clasificación para tres clases. El siguiente código une las tres matrices `X_1`, `X_2` y `X_3` y genera un arreglo `y` que representa la clase.

```
D = np.concatenate((X_1, X_2, X_3), axis=0)
y = np.array(['a'] * X_1.shape[0] +
             ['b'] * X_2.shape[0] +
             ['c'] * X_3.shape[0])
```

Por construcción el problema está en $\mathbb{R}^2$ y se sabe que las dos componentes contribuyen a la solución del mismo, es decir, imagine que una de las variables se pierde, con la información restante se desarrollaría un algoritmo de clasificación con un rendimiento mucho menor a aquel que tenga toda la información.

Continuando con el problema sintético, en está ocasión lo que se realiza es incluir en el problema una variable que no tiene relación con la clase, para esto se añade una variable aleatoria con una distribución Gausiana con $\mu = 2$ y $\sigma = 3$ tal como se muestra en el siguiente código.

```
N = norm.rvs(loc=2, scale=3, size=3000)
D = np.concatenate((D, np.atleast_2d(N).T), axis=1)
```

El objetivo es encontrar la variable que no está relacionada con la salida. Una manera de realizar esto es imaginar que si la media en las diferentes variables es la misma en todas las clases entonces esa variable no contribuye a discriminar la clase. En Sección 3.5.2 se presentó el procedimiento para obtener las medias que definen $\mathbb{P}(\mathcal{X} \mid \mathcal{Y})$ para cada clase. El siguiente código muestra el procedimiento para calcular las medías que son $\mu_1 = [4.9991, 5.0004, 2.1551]^\top$, $\mu_2 = [1.4479, -1.5558, 2.0507]^\top$ y $\mu_3 = [12.4621, -3.5588, 1.9571]^\top$.

```
labels = np.unique(y)
mus = [np.mean(D[y==i], axis=0) for i in labels]
```

Se observa que la media de la tercera variable es aproximadamente igual para las tres clases, teniendo un valor cercano a 2 tal y como fue generada. Entonces lo que se busca es un procedimiento que permita identificar que las muestras en cada grupo (clase) hayan sido originadas por la misma distribución. Es decir se busca una prueba que indique que las primeras dos variables provienen de diferentes distribuciones y que la tercera provienen de la misma distribución. Es pertinente comentar que este procedimiento no es aplicable para problemas de regresión.

Si se puede suponer que los datos provienen de una Distribución Gausiana entonces la prueba a realizar es ANOVA, en caso contrario se puede utilizar su equivalente método no paramétrico como es la prueba Kruskal-Wallis. Considerando que de manera general se desconoce la distribución que genera los datos, entonces se presenta el uso de la segunda prueba.

La prueba Kruskal-Wallis identifica si un conjunto de muestras independientes provienen de la misma distribución. La hipótesis nula es que las muestras provienen de la misma distribución.

La función `kruskal` implementa esta prueba y recibe tantas muestras como argumentos. En el siguiente código ilustra su uso, se observa que se llama a la función `kruskal` para cada columna en `D` y se calcula su valor p. Los valores p obtenidos son: $[0.0000, 0.0000, 0.5128]$ lo cual indica que para las primeras dos variables la hipótesis nula se puede rechazar y por el otro lado la hipótesis nula es factible para la tercera variable con un valor $p = 0.5128$.

```
res = [kruskal(*[D[y==l, i] for l in labels]).pvalue
       for i in range(D.shape[1])]
```

En lugar de discriminar aquellas características que no aportan a modelar los datos, es más común seleccionar las mejores características. Este procedimiento se puede realizar utilizando los valores p de la prueba estadística o cualquier otra función que ordene la importancia de las características.

El procedimiento equivalente a la estadística de Kruskal-Wallis en regresión es calcular la estadística F cuya hipótesis nula es asumir que el coeficiente obtenido en una regresión lineal entre las variables independientes y la dependiente es zero. Esta estadística se encuentra implementada en la función `f_regression`. El siguiente código muestra su uso en el conjunto de datos de diabetes; el cual tiene 10 variables independientes. En la variable `p_values` se tienen los valores p se puede observar que el valor p correspondiente a la segunda variable tiene un valor de 0.3664, lo cual hace que esa variable no sea representativa para el problema que se está resolviendo.

```
X, y = load_diabetes(return_X_y=True)
f_statistics, p_values = f_regression(X, y)
```

Un ejemplo que involucra la selección de las variables más representativas mediante una calificación que ordenan la importancia de las mismas se muestra en el siguiente código. Se puede observar que las nueve variables seleccionadas son: [0, 2, 3, 4, 5, 6, 7, 8, 9] descartando la segunda variable que tiene el máximo valor p.

```
sel = SelectKBest(score_func=f_regression, k=9).fit(X, y)
vars = sel.get_support(indices=True)
```

5.2.1 Ventajas y Limitaciones

Las técnicas vistas hasta este momento requieren de pocos recursos computacionales para su cálculo. Además están basadas en estadísticas que permite saber cuales son las razones de funcionamiento. Estas fortalezas también son origen a sus debilidades, estas técnicas observan en cada paso las variables independientes de manera aislada y no consideran que estas variables pueden interactuar. Por otro lado, la selección es agnóstica del algoritmo de aprendizaje utilizado.

5.3 Selección hacia Adelante

Un procedimiento que pone en el centro del proceso de selección al algoritmo de aprendizaje utilizado es **Selección hacia Adelante** y su complemento que sería **Selección hacia Atrás.** El algoritmo de selección hacia adelante es un procedimiento iterativo que selecciona una variable a la vez guiada por el rendimiento de esta variable cuando es usada en el algoritmo de aprendizaje. Al igual que los procedimientos anteriores este no modifica las características del problema, solamente selecciona las que se consideran relevantes.

En selección hacia adelante y hacia atrás se inicia con el conjunto de entrenamiento $\mathcal{T} = \{(x_i, y_i)\}$, con una función L que mide el rendimiento del algoritmo de aprendizaje $\mathcal{H}$. La idea es ir seleccionando de manera iterativa aquellas variables que generan un modelo con mejores capacidades de generalización. Para medir la generalización del algoritmo se pueden realizar de diferentes maneras, una es mediante la división de $\mathcal{X}$ en dos conjuntos: entrenamiento y validación; y la segunda manera corresponde a utilizar k-iteraciones de validación cruzada.

Suponga un conjunto $\pi \subseteq \{1, 2, \ldots, d\}$ de tal manera que $\mathcal{T}_\pi$ solamente cuenta con las variables identificadas en el conjunto π. Utilizando está notación el algoritmo se puede definir de la siguiente manera. Inicialmente $\$ \hat{}0 = \$$, en el siguiente paso $\pi^{j+1} \leftarrow \pi^j \cup \arg\max_{\{i|i\in\{1,2,\ldots,d\}, i\notin\pi^j\}} \mathcal{P}(\mathcal{H}, \mathcal{T}_{\pi\cup i}, L)$, donde $\mathcal{P}$ representa el rendimiento del algoritmo $\mathcal{H}$ en el subconjunto $\pi \cup i$ usando la función de rendimiento L. Este proceso continua si $\mathcal{P}^{j+1} > \mathcal{P}^j$ donde $\mathcal{P}^0 = 0$.

Es importante mencionar que el algoritmo antes descrito es un algoritmo voraz y que el encontrar el óptimo de este problema de optimización no se garantiza con este tipo de algoritmos. Lo que quiere decir es que el algoritmo encontrará un óptimo local.

Ilustrando estos pasos en el conjunto de Breast Cancer Wisconsin (Sección 3.7).

```
D, y = load_breast_cancer(return_X_y=True)
T, G, y_t, y_g = train_test_split(D, y, test_size=0.2)
```

El primer paso es medir el rendimiento cuando solamente una variable interviene en el proceso. Como inicialmente $\$ \hat{}0 = \$$ entonces solo es necesario generar un clasificador cuando sola una variable está involucrada. El rendimiento de cada variable se guarda en la variable `perf`, se puede observar que el primer ciclo (línea 3) itera por todas las variables en la representación, para cada una se seleccionan los datos solo con esa variable `T1 = T[:, np.array([var])]` después se hacen k-iteraciones de validación cruzada y finalmente se guarda el rendimiento. El rendimiento que se está calculando es macro-recall.

```
perf = []
kfold = KFold(shuffle=True)
for var in range(T.shape[1]):
    T1 = T[:, np.array([var])]
```

```
    perf_inner = []
    for ts, vs in kfold.split(T1):
        gaussian = GaussianNB().fit(T1[ts], y_t[ts])
        hy_gaussian = gaussian.predict(T1[vs])
        _ = recall_score(y_t[vs], hy_gaussian,
                         average='macro')
        perf_inner.append(_)
    perf.append(np.mean(perf_inner))
```

La Figura 5.1 muestra el rendimiento de las 30 variables, se observa como una gran parte de las variables proporcionan un rendimiento superior al 0.8 y la variable que tiene el mejor rendimiento es la que corresponde al índice 7 y valor 0.9115.

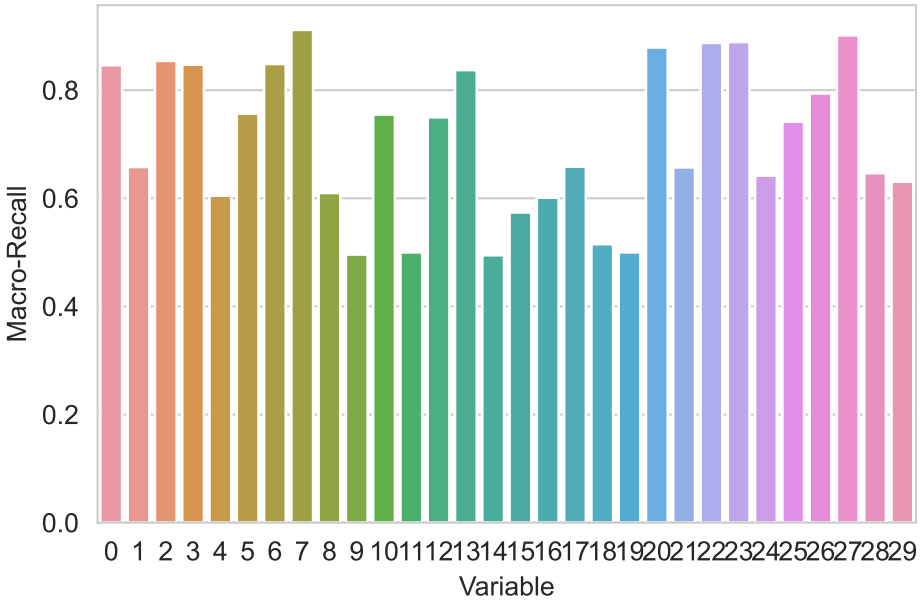


Figura 5.1: Rendimiento de las variables en la primera iteración del selección hacia adelante

El algoritmo de selección hacia atrás y adelante se implementa en la clase `SequentialFeatureSelector` y su uso se observa en las siguientes instrucciones.

```
kfolds = list(KFold(shuffle=True).split(T))
scoring = make_scorer(lambda y, hy: recall_score(y, hy,
                                        average='macro'))
seq = SequentialFeatureSelector(estimator=GaussianNB(),
                                scoring=scoring,
```

```
                                    n_features_to_select='auto',
                                    cv=kfolds).fit(T, y_t)
```

Al igual que en el algoritmo de `SelectKBest` las variables seleccionadas se pueden observar con la función `get_support`. En este caso las variables seleccionadas son: $[1, 2, 4, 7, 9, 11, 15, 19, 20, 21, 22, 23, 24, 27, 28]$.

Utilizando el parámetro `direction='backward'` para utilizar selección hacia atrás en el mismo conjunto de datos, da como resultado la siguiente selección $[0, 1, 2, 9, 11, 13, 16, 18, 20, 21, 22, 24, 26, 27, 28]$. Se puede observar que las variables seleccionadas por los métodos son diferentes. Esto es factible porque los algoritmos solo aseguran llegar a un máximo local y no está garantizado que el máximo local corresponda al máximo global.

5.3.1 Ventajas y Limitaciones

Una de las ventajas de la selección hacia atrás y adelante es que el algoritmo termina a lo más cuando se han analizado todas las variables, esto eso para un problema en $\mathbb{R}^d$ se analizarán un máximo de d variables. La principal desventaja es que estos algoritmos son voraces, es decir, toman la mejor decisión en el momento, lo cual tiene como consecuencia que sean capaces de garantizar llegar a un máximo local y en ocasiones este máximo local no corresponde al máximo global. Con el fin de complementar esta idea, en un problema $\mathbb{R}^d$ se tiene un espacio de búsqueda de $2^d - 1$, es decir, se tiene esa cantidad de diferentes configuraciones que se pueden explorar. En los algoritmos de vistos se observa un máximo de d elementos de ese total.

5.4 Selección mediante Modelo

Existen algoritmos de aprendizaje supervisado donde sus parámetros indican la importancia que tiene cada característica en el problema. Para ejemplificar esto se utilizará el problema de Diabetes que fue utilizado en la Sección 3.9.1. Los datos se obtienen con la siguiente instrucción, adicionalmente se normalizan las características para tener una media 0 y desviación estándar 1.

```
D, y = load_diabetes(return_X_y=True)
D = StandardScaler().fit_transform(D)
```

El siguiente paso se estiman los parámetros de una regresión lineal (Sección 3.9), también se muestran los valores de los primeros tres coeficientes de la regresión. Se puede observar que el valor absoluto de estos coeficientes está en diferentes rangos, el primer coeficiente es cercano a cero, el segundo está alrededor de 10 y el tercero es superior a 20. Considerando que los datos están normalizados, entonces el valor absoluto indica el efecto que tiene esa variable en

el resultado final, en estos tres datos es puede concluir que el tercer coeficiente tiene una mayor contribución que los otros dos coeficientes.

```
reg = LinearRegression().fit(D, y)
reg.coef_[:3]
```

```
array([ -0.47612079, -11.40686692,  24.72654886])
```

La clase `SelectFromModel` permite seleccionar las d características que el modelo considera más importante. En el siguiente ejemplo se selecciona las $d = 4$ características más importantes; dentro de estas características se encuentra la tercera (i.e., índice 2) cuyo coeficiente se había presentado previamente.

```
sel = SelectFromModel(reg, max_features=4,
                      threshold=-np.inf, prefit=True)
np.where(sel.get_support())[0]
```

```
array([2, 4, 5, 8])
```

Habiendo descrito el procedimiento para seleccionar aquellas características más importantes, es necesario encontrar cuál es el número de características que producen el mejor rendimiento. Para realizar esto, se realiza k-iteraciones de validación cruzada (Sección 4.4.1), donde se varía el número de características y en cada caso se mide el rendimiento utilizando la medida R^2 (Sección 4.3).

La Figura 5.2 muestra el rendimiento (R^2) para cada modelo generado, empezando cuando se usan las 2 características más importantes y terminando con el modelo que tiene todas las características. En la figura se puede observar que el rendimiento mejora considerablemente de 2 a 3 características, este sigue mejorando y llega un punto donde incluir más características tiene un impacto negativo en el rendimiento.

5.5 Análisis de Componentes Principales

Los algoritmos de selección hacia atrás y adelante tiene la característica de requerir un conjunto de entrenamiento de aprendizaje supervisado, por lo que no podrían ser utilizados en problemas de aprendizaje no-supervisado. En esta sección se revisará el uso de Análisis de Componentes Principales (Principal Components Analysis - PCA) para la reducción de dimensión. PCA tiene la firma: $f : \mathbb{R}^d \to \mathbb{R}^m$ donde $m < d$

La idea de PCA es buscar una matriz de proyección $W^\top \in \mathbb{R}^{m \times d}$ tal que los elementos de $\mathcal{D} = \{x_i\}$ sea transformados utilizando $z = W^\top x$ donde $z \in \mathbb{R}^m$. El objetivo es que la muestra

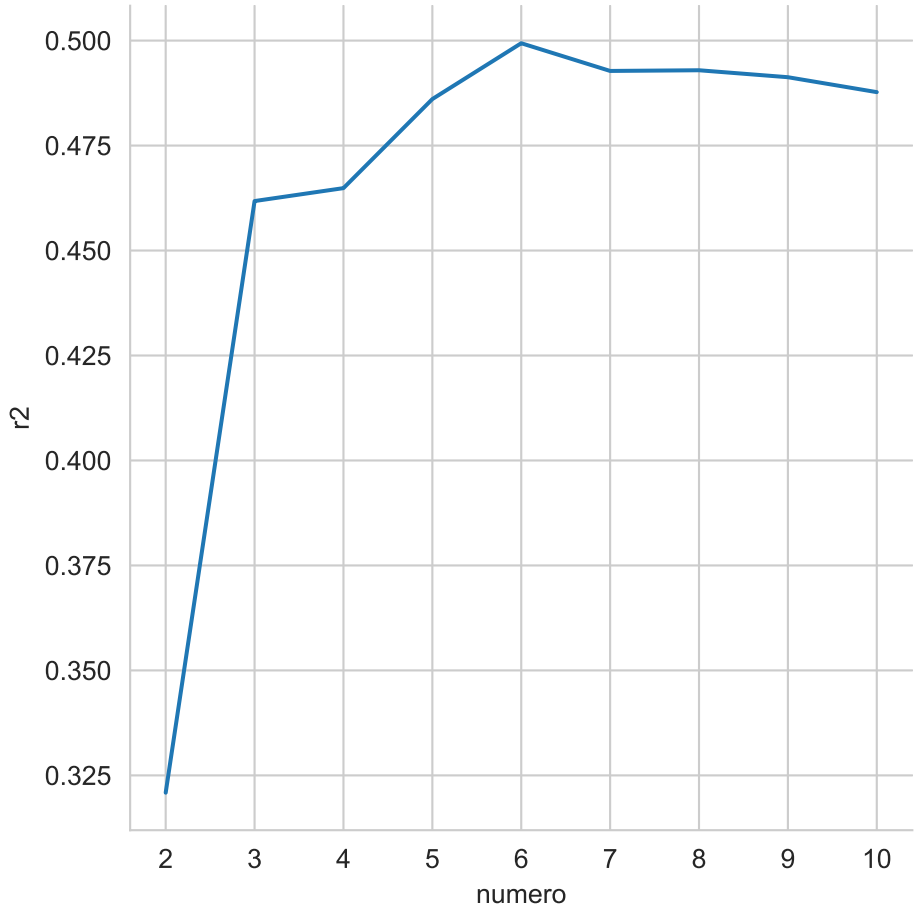


Figura 5.2: Rendimiento variando el número de características

z_1 tenga la mayor variación posible. Es decir, se quiere observar en la primera característica de los elementos transformados la mayor variación; esto se puede lograr de la siguiente manera.

Si suponemos que $\mathbf{x} \sim \mathcal{N}_d(\mu, \Sigma)$ y $\mathbf{w} \in \mathbb{R}^d$ entonces $\mathbf{w} \cdot \mathbf{x} \sim \mathcal{N}(\mathbf{w} \cdot \mu, \mathbf{w}^\top \Sigma \mathbf{w})$ y por lo tanto $\mathsf{Var}(\mathbf{w} \cdot \mathbf{x}) = \mathbf{w}^\top \Sigma \mathbf{w}$.

Utilizando esta información se puede describir el problema como encontrar $\mathbf{w}_1$ tal que $\mathsf{Var}(\mathbf{z}_1)$ sea máxima, donde $\mathsf{Var}(\mathbf{z}_1) = \mathbf{w}_1^\top \Sigma \mathbf{w}_1$. Dado que en este problema de optimización tiene multiples soluciones, se busca además maximizar bajo la restricción de $|| \mathbf{w}_1 ||= 1$. Escribiéndolo como un problema de Lagrange quedaría como: $\max_{\mathbf{w}_1} \mathbf{w}_1^\top \Sigma \mathbf{w}_1 - \alpha(\mathbf{w}_1 \cdot \mathbf{w}_1 - 1)$. Derivando con respecto a $\mathbf{w}_1$ se tiene que la solución es: $\Sigma \mathbf{w}_i = \alpha \mathbf{w}_i$ donde esto se cumple solo si $\mathbf{w}_1$ es un eigenvector de Σ y α es el eigenvalor correspondiente. Para encontrar $\mathbf{w}_2$ se requiere $|| \mathbf{w}_2 ||= 1$ y que los vectores sean ortogonales, es decir, $\mathbf{w}_2 \cdot \mathbf{w}_1 = 0$. Realizando las operaciones necesarias se encuentra que $\mathbf{w}_2$ corresponde al segundo eigenvector y así sucesivamente.

5.5.1 Ejemplo - Visualización

Supongamos que deseamos visualizar los ejemplos del problema del iris. Los ejemplos se encuetran en $\mathbb{R}^4$ entonces para poderlos graficar en $\mathbb{R}^2$ se requiere realizar una transformación como podría ser Análisis de Componentes Principales.

Empezamos por cargar los datos del problema tal y como se muestra en la siguiente instrucción, en la segunda linea se normalizan los datos para tener una media 0 y desviación estándar 1, este procedimiento es necesario cuando los valores de las características se encuentran en diferentes escalas.

```
D, y = load_iris(return_X_y=True)
D = StandardScaler().fit_transform(D)
```

Habiendo importado los datos el siguiente paso es inicializar la clase de PCA, para esto requerimos especificar el parámetro que indica el número de componentes deseado, dado que el objetivo es representar en $\mathbb{R}^2$ los datos, entonces el ocupamos dos componentes. La primera línea inicializa la clase de PCA, después, en la segunda línea se hace la proyección.

```
pca = decomposition.PCA(n_components=2).fit(D)
Xn = pca.transform(D)
```

El siguiente código se utiliza para visualizar los datos. El resultado se muestra en la Figura 5.3, donde se observa en diferente color cada una de las clases.

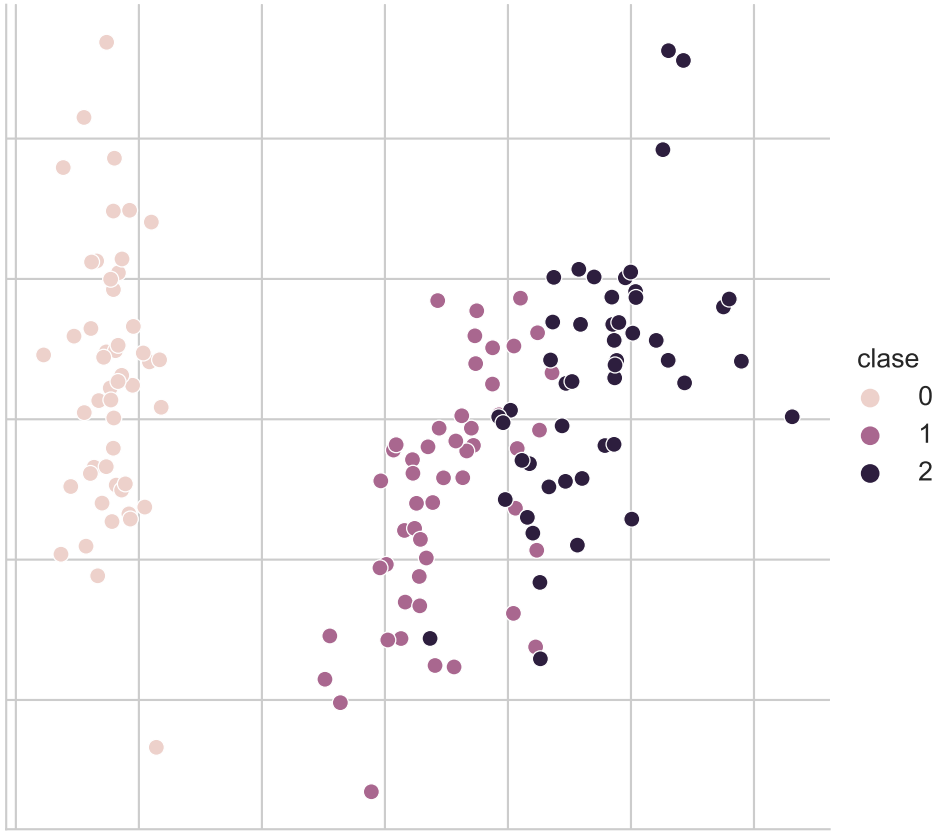


Figura 5.3: Proyección mediante PCA del problema del Iris

5.6 UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction

En esta sección se presenta el uso de UMAP propuesto por McInnes, Healy, y Melville (2020). UMAP es una técnica no lineal para reducir la dimensión y que con dimensiones bajas ($d = 2$) permite tener una visualización de los datos. El ejemplo que se realizará es visualizar mediante UMAP los datos del problema de Dígitos, estos datos se pueden obtener con la siguiente instrucción; en estas instrucciones también se normalizan los datos para tener una media 0 y desviación estándar 1.

```
D, y = load_digits(return_X_y=True)
D = StandardScaler().fit_transform(D)
```

Con el objetivo de ilustrar la diferencia entre una técnica de proyección lineal como PCA con respecto a una técnica no lineal, la Figura 5.4 presenta la proyección del conjunto de Dígitos utilizando PCA. Como se puede observar en la figura, esta proyección no da una separación clara entre grupos, aunque si se puede identificar una estructura, donde algunos números se encuentran en la periferia de la figura.

UMAP genera una proyección que intuitivamente trata de preservar la distancia que existe entre los K vecinos cercanos en la dimensión original como en la dimensión destino. Por este motivo un parámetro importante de UMAP es la cantidad de vecinos, que se tiene el efecto de priorizar la estructura global o local de los datos. En el siguiente código se realiza una exploración de este parámetro generando una visualización para $K = [2, 4, 8, 16]$. El código que realiza está exploración se muestra en las siguientes instrucciones.

```
df = pd.DataFrame()
for K in [2, 4, 8, 16]:
    reducer = umap.UMAP(n_neighbors=K)
    low_dim = reducer.fit_transform(D)
    _ = pd.DataFrame(low_dim, columns=['x', 'y'])
    _['Clase'] = y
    _['K'] = K
    df = pd.concat((df, _))
```

La Figura 5.5 muestra las diferentes visualizaciones cuando el número de vecinos de cambia en UMAP, se puede observar como para $K = 2$ la visualización no muestra ninguna estructura, pero partiendo de $K = 4$ se puede visualizar grupos que corresponden a los diferentes dígitos.

Con el objetivo de mostrar con mayor detalle los grupos encontrados, la Figura 5.6 muestra la visualización de $K = 8$ en ella se puede observar como los ejemplos se agrupan de acuerdo al dígito que representan, también se puede ver la similitud entre los diferentes dígitos y como algunos ejemplos están muy cerca a un grupo al cual no pertenecen.

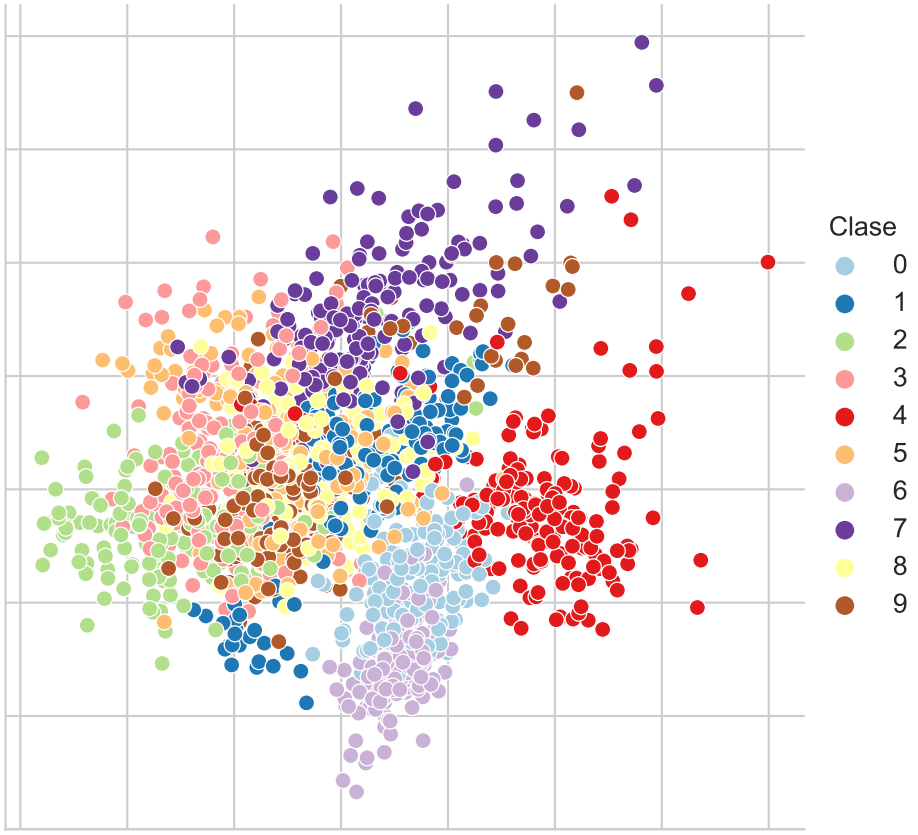


Figura 5.4: Proyección mediante PCA del problema de Dígitos

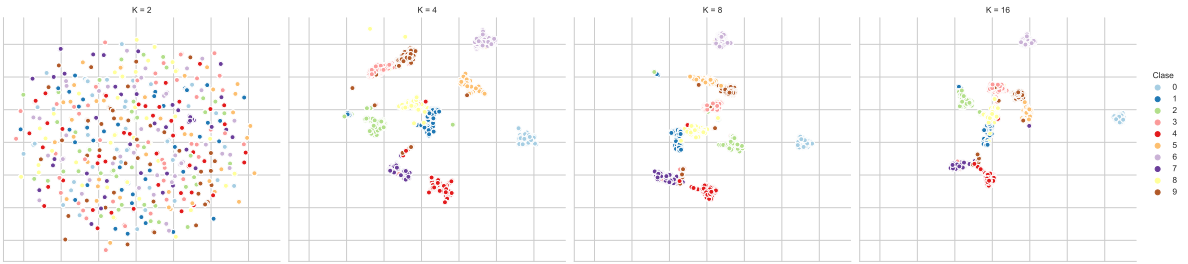


Figura 5.5: Proyección mediante UMAP del problema de Dígitos variando el número de vecinos

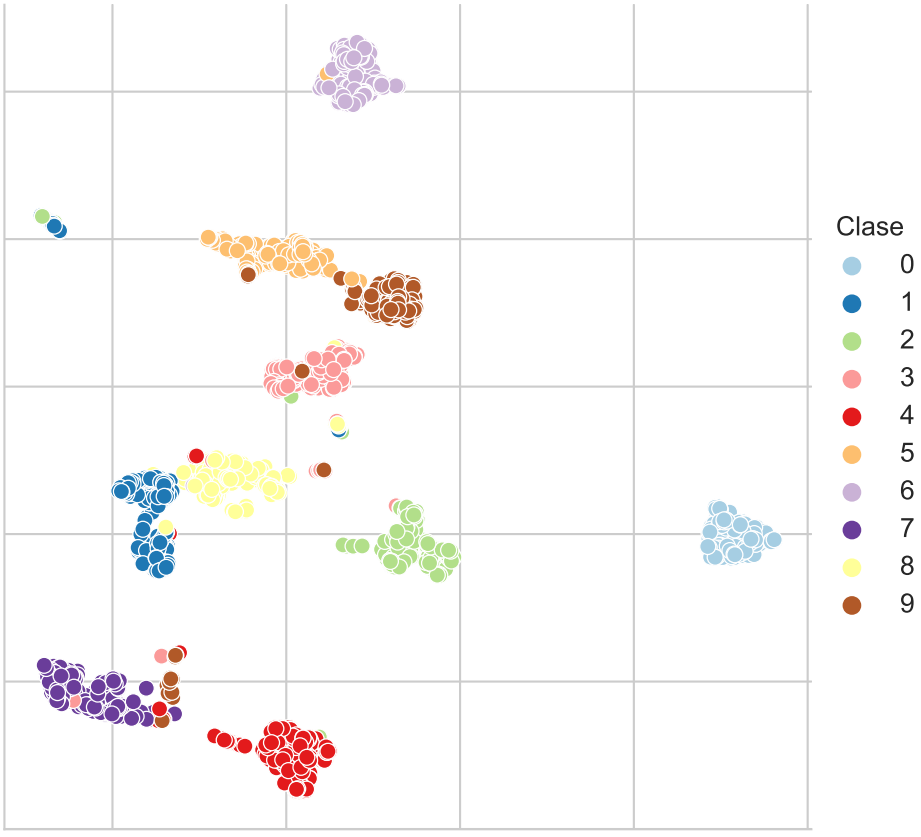


Figura 5.6: Proyección mediante UMAP del problema de dígitos con ocho vecinos

6 Agrupamiento

El **objetivo** de la unidad es conocer y aplicar el algoritmo de agrupamiento k-medias

6.1 Paquetes usados

```
from sklearn.cluster import KMeans
from sklearn.datasets import load_iris
from sklearn import decomposition
from sklearn import metrics
from scipy import linalg
from itertools import permutations
from matplotlib import pylab as plt
import numpy as np
import pandas as pd
import seaborn as sns
```

6.2 Introducción

Esta unidad trata el problema de agrupamiento, el cual es un problema de aprendizaje no supervisado (Sección 1.3), en el cual se cuenta con un conjunto $\mathcal{D} = \{x_i \mid i = 1, \dots, N\}$ donde $x_i \in \mathbb{R}^d$. El objetivo de agrupamiento es separar los elementos de $\mathcal{D}$ en K grupos. Es decir asociar a $x_i \in \mathcal{D}$ a un grupo $x_i \in G_j$ donde $\cup_j^K G_j = \mathcal{D}$ y $G_j \cap G_i = \emptyset$ para todo $i \neq j$.

Por supuesto existen diferentes algoritmos que se han desarrollado para generar esta partición, en particular, todos de ellos encuentran la participación optimizando una función objetivo que se considera adecuada para el problema que se está analizando. En particular, esta unidad se enfoca a describir uno de los algoritmos de agrupamiento más utilizados que es K-medias.

6.3 K-medias

De manera formal el objetivo de K-medias es encontrar la partición $G = \{G_1, G_2, \dots, G_K\}$ que corresponda al $\min \sum_{i=1}^{K} \sum_{x \in G_i} \mid\mid x - \mu_i \mid\mid$, donde μ_i es la media de todos los elementos que pertenecen a G_i.

Para comprender la función objetivo ($\min \sum_{i=1}^{K} \sum_{x \in G_i} \mid\mid x - \mu_i \mid\mid$) de k-medias se explican los dos componentes principales que son las medias μ_i y los grupos G_i.

Para ilustrar tanto a μ_i como a G_i se utiliza el problema del iris (Sección 5.5.1) cuyos datos se pueden obtener de la siguiente manera.

```
D, y = load_iris(return_X_y=True)
```

6.3.1 μ_i

Como se describió, μ_i es la media de los elementos que corresponden al grupo G_i. Asumiendo que el grupo 1 (G_1) tiene 10 elementos seleccionados de manera aleatoria de $\mathcal{D}$ como se muestra a continuación.

```
index = np.arange(len(D))
np.random.shuffle(index)
sel = index[:10]
G_1 = D[sel]
```

La variable `G_1` tiene los 10 elementos considerados como miembros de G_1 entonces μ_1 se calcula como la media de cada componente, lo cual se puede calcular con el siguiente código.

```
mu_1 = G_1.mean(axis=0)
```

La Figura 6.1 muestra los elementos seleccionados ($x \in G_1$) y la media (μ_1) del grupo. Los elementos se encuentran en $\mathbb{R}^4$ y para visualizarlos se transformaron usando PCA descrito en la Sección 5.5.1.

6.3.2 G_i

El complemento del procedimiento anterior es encontrar los elementos de G_i dando la μ_i. El ejemplo consiste en generar dos medias, es decir, $K = 2$ y encontrar los elementos que corresponden a las medias generadas. Se puede utilizar cualquier procedimiento para generar dos vectores de manera aleatoria, pero en este ejemplo se asume que estos vectores corresponden

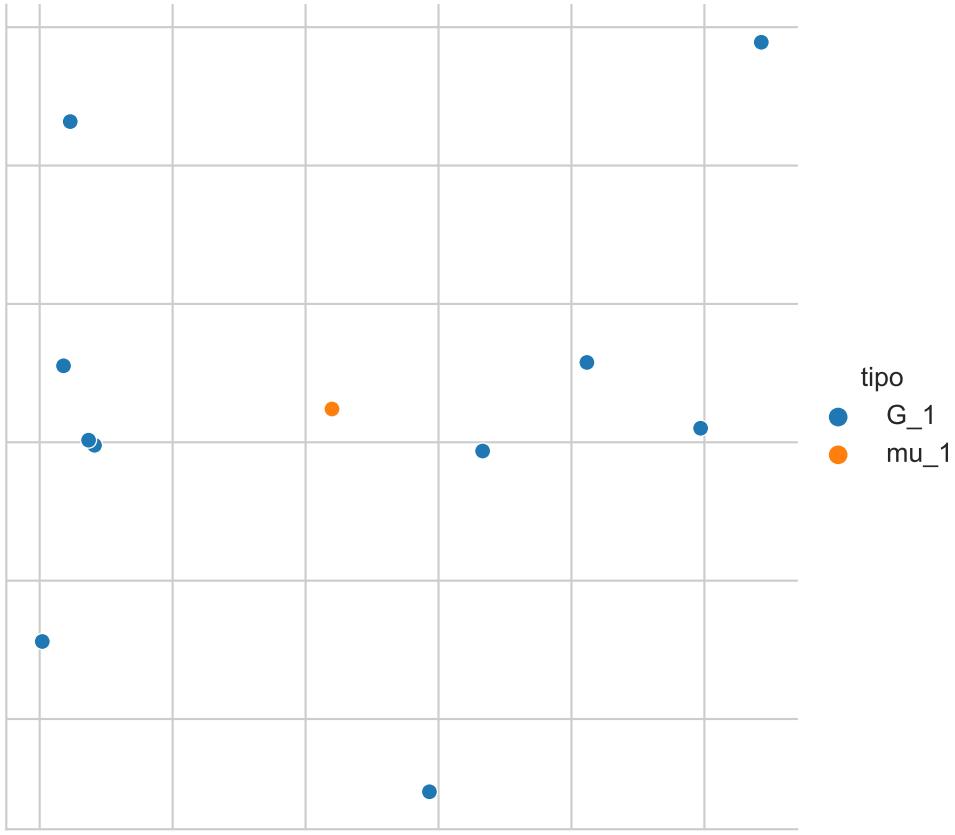


Figura 6.1: Elementos del primer grupo y su centro

a dos elementos de $\mathcal{D}$. Estos elementos son los que se encuentran en los indices 50 y 100 tal y como se muestra en las siguientes instrucciones.

```
mu_1 = D[50]
mu_2 = D[100]
```

El elemento x pertenece al grupo G_i si el valor $|| x - \mu_i ||$ corresponde al $\min_j || x - \mu_j ||$. Entonces se requiere calcular $|| x - \mu_i ||$ para cada elemento $x \in \mathcal{D}$ y para cada una de las medias μ_i. Esto se puede realizar con la siguiente instrucción

```
dis = np.array([linalg.norm((D - np.atleast_2d(mu)),
                            axis=1)
                for mu in [mu_1, mu_2]]).T
```

donde se puede observar que el ciclo itera por cada una de las medias, i.e., `mu_1` y `mu_2`. Después se calcula la norma utilizando la función `linalg.norm` y finalmente se regresa la transpuesta para tener una matriz de 150 renglones y dos columnas que corresponden al número de ejemplos en $\mathcal{D}$ y a las dos medias. Los valores de `dis[50]` y `dis[100]` son array([0. , 1.8439]) y array([1.8439, 0.]) respectivamente. Tal y como se espera porque μ_1 corresponde al índice 50 y μ_2 es el índice 100. Estos dos ejemplos, array([0. , 1.8439]) y array([1.8439, 0.]), permiten observar que el argumento mínimo de `dis` identifica al grupo del elemento, haciendo la consideración que el índice 0 representa G_1 y el índice 1 es G_2. La siguiente instrucción muestra como se realiza esta asignación.

```
G = dis.argmin(axis=1)
```

La Figura 6.2 muestra los grupos formados, el primer grupo `G_1` se encuentra en azul y el segundo en naranja, también muestra los elementos que fueron usados como medias de cada grupo; estos elementos se observan en color verde.

6.3.3 Algoritmo

Habiendo explicado μ_i y G_i se procede a describir el procedimiento para calcular los grupos utilizado por k-medias. Este es un procedimiento iterativo que consta de los siguientes pasos.

1. Se generar K medias de manera aleatoria, donde μ_i corresponde a G_i.
2. Para cada media, μ_i, se seleccionan los elementos más cercanos, esto es, $x \in G_i$ si $|| x - \mu_i ||$ corresponde al $\min_j || x - \mu_j ||$.
3. Se actualizan las μ_i con los elementos de G_i.
4. Se regresa al paso 2.

El procedimiento termina cuando se llega a un número máximo de iteraciones o que la variación de los μ_i es mínima, es decir, que los grupos no cambian.

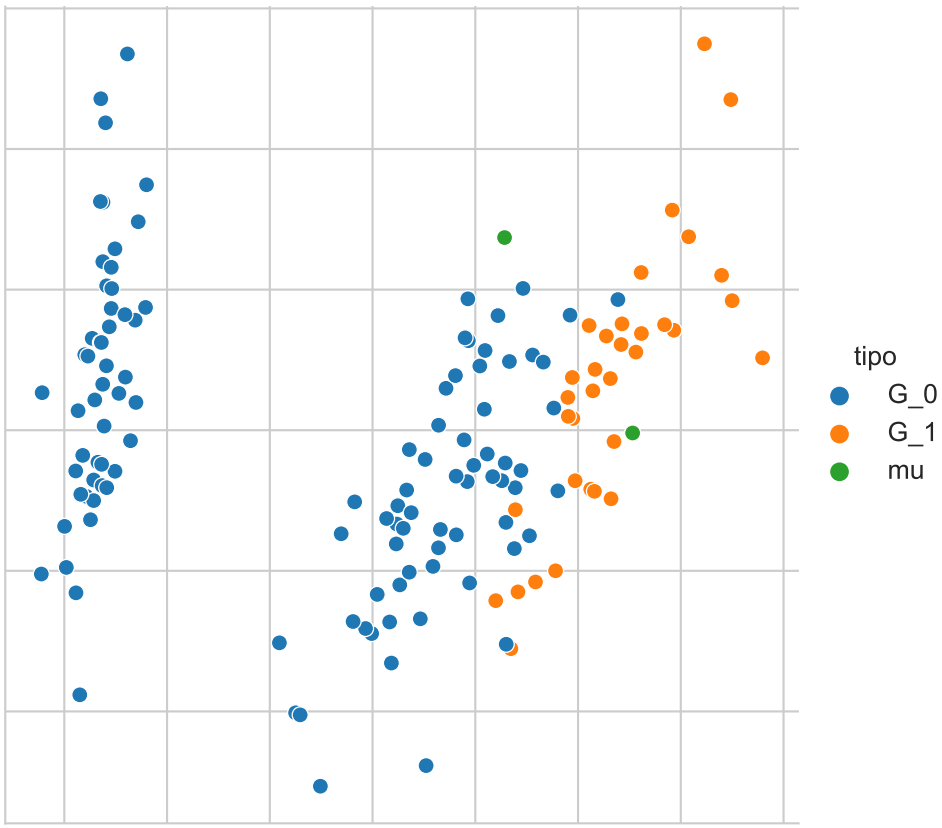


Figura 6.2: Proyección de los elementos del conjunto y sus centros

6.4 Ejemplo: Iris

En el siguiente ejemplo se usará K-medias para encontrar 2 y 3 grupos en el conjunto del iris. La clase se inicializa primero con 2 grupos (primera línea). En la segunda instrucción se predice los grupos para todo el conjunto de datos. Las medias para cada grupo se encuentran en el atributo `cluster_centers_`.

```
m = KMeans(n_clusters=2, n_init='auto').fit(D)
cl = m.predict(D)
```

La Figura 6.3 muestra el resultado del algoritmo k-means en el conjunto del Iris, se muestran los dos grupos G_1 y G_2 y en color verde μ_1 y μ_2.

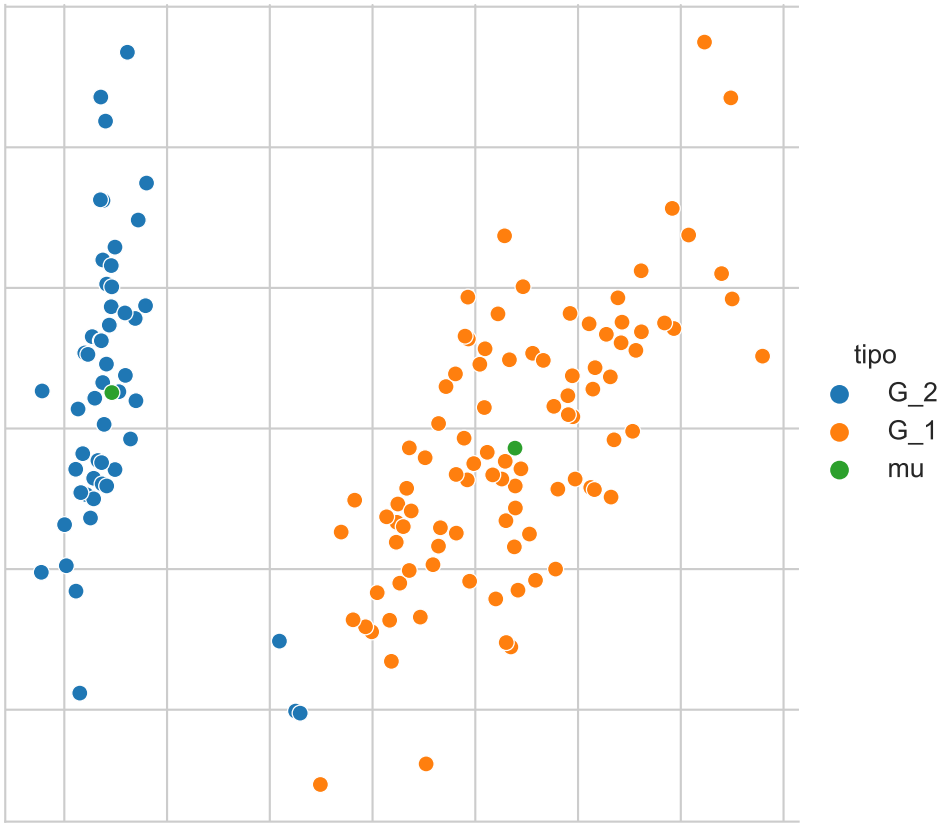


Figura 6.3: Proyección de k-means usando dos grupos

Un procedimiento equivalente se puede realizar para generar tres grupos, el único cambio es el parámetro `n_clusters` en la clase `KMeans` de la siguiente manera.

```
m = KMeans(n_clusters=3, n_init='auto').fit(D)
cl = m.predict(D)
```

La Figura 6.4 muestra los tres grupos y con sus tres respectivas medias en color rojo.

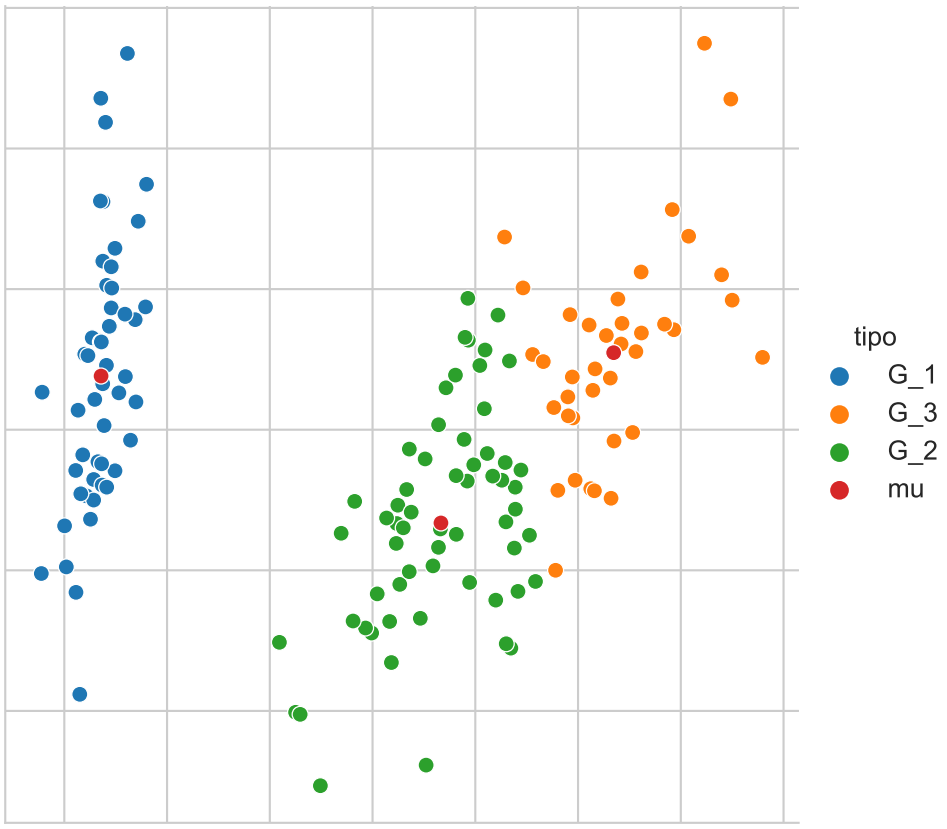


Figura 6.4: Proyección de k-means usando tres grupos

6.5 Rendimiento

Recordando que en aprendizaje no supervisado no se tiene una variable dependiente que predecir. En este caso particular se utilizó un problema de clasificación para ilustrar el procedimiento de k-medias, entonces se cuenta con una clase para cada elemento $x \in \mathcal{D}$. Además se sabe que el problema del iris tiene tres clases, entonces utilizando los tres grupos obtenidos previamente podemos medir que tanto se parecen estos tres grupos a las clases del iris. Es decir, se puede saber si el algoritmo de k-medias agrupa los elementos de tal manera que cada grupo corresponda a una clase del iris.

Los grupos generados se encuentran en la lista `cl` y las clases se encuentran en `y`. La lista `y` tiene organizada las clases de la siguiente manera: los primeros 50 elementos son la clase 0, los siguientes 50 son clase 1 y los últimos son la clase 2. Dado que K-medias no conoce los clases y genera los grupos empezando de manera aleatoria, entonces es probable que los grupos sigan una numeración diferente al problema del iris. Los grupos en `cl` están organizados de la siguiente manera aproximadamente los 50 primeros elementos son del grupo 1, los siguientes son grupo 0 y finalmente los últimos son grupo 2. Entonces se puede hacer una transformación usando la variable `perm` con la siguiente información array([0, 1, 2]).

Utilizando `perm` se calcula la exactitud (Sección 4.2.2) utilizando la siguiente instrucción. Se obtiene una exactitud de 0.8867 que significa que la mayoría de los datos se agrupan en un conjunto que corresponde a la clase del conjunto del iris.

```
acc = metrics.accuracy_score(y, perm[cl])
```

En general en agrupamiento no se cuenta con la composición de los grupos, es más, se desconocen cuántos grupos modela el problema. Para estas ocasiones es imposible medir el accuracy o cualquier otra medida de agrupamiento que requiera la composición de real de los grupos.

Una medida que no requiere conocer los grupos es el **Silhouette Coefficient**; el cual mide la calidad de los grupos, mientras mayor sea el valor significa una mejor calidad en los grupos. Este coeficiente se basa en la siguiente función:

$$s = \frac{b - a}{\max(a, b)},$$

donde a corresponde a la distancia media entre un elemento y todos las objetos del mismo grupo; y b es la distancia media entre la muestra y todos los elementos del grupo más cercano.

Por ejemplo, en el problema del Iris s tiene un valor de 0.5512 calculado con la siguiente instrucción.

```
sil = metrics.silhouette_score(D, cl, metric='euclidean')
```

Otra medida de la calidad de los grupos es índice de **Calinski-Harabasz** que mide la dispersión entre grupos y dentro del grupo, al igual que Silhouette, mientras mayor sea la estadística mejor es el agrupamiento. Para el problema del Iris el índice de Calinski-Harabasz tiene un valor de 561.5937 obtenido con la siguiente instrucción.

```
cal_har = metrics.calinski_harabasz_score(D, cl)
```

6.6 Número de Grupos

Utilizando una medida de rendimiento de agrupamiento se analizar cual sería el número adecuado de grupos para un problema dado. El procedimiento es variar el número de grupos y medir el rendimiento para cada grupo y quedarse con aquel que da el mejor rendimiento considerando también el número de grupos.

Por ejemplo, el siguiente instrucción calcula el coeficiente de Silhouette y el índice de Calinski-Harabasz en el problema del Iris.

```
S1 = []
S2 = []
for k in range(2, 11):
    m = KMeans(n_clusters=k, n_init='auto').fit(D)
    cl = m.predict(D)
    _ = metrics.silhouette_score(D, cl, metric='euclidean')
    S1.append(_)
    _ = metrics.calinski_harabasz_score(D, cl)
    S2.append(_)
```

Estas dos estadísticas se pueden observar en la Figura 6.5. En color azul se observa el coeficiente de Silhouette; donde el mejor resultado es cuando $K = 2$. En color naranja se muestra el índice Calinski-Harabasz done el mejor resultado se tiene cuando $K = 3$. Considerando que se está trabajando con el problema del Iris se conoce que la mejor agrupación es para $K = 3$ dado que son tres clases.

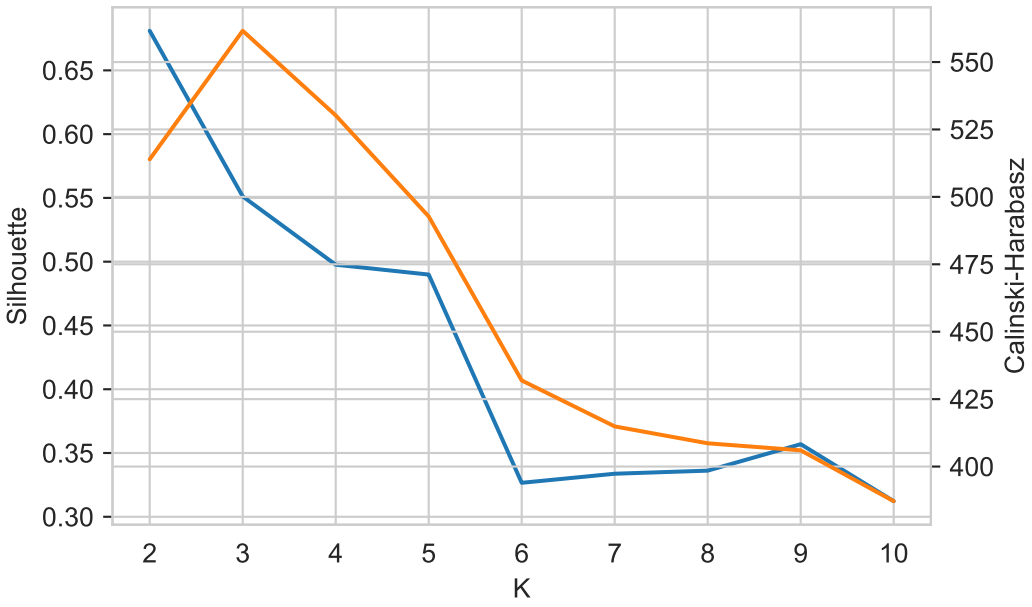


Figura 6.5: Rendimiento variando el número de grupos

7 Métodos No Paramétricos

El **objetivo** de la unidad es conocer las características de diferentes métodos no paramétricos y aplicarlos para resolver problemas de regresión y clasificación.

7.1 Paquetes usados

```
from sklearn.neighbors import NearestNeighbors,\
                              KNeighborsClassifier,\
                              KNeighborsRegressor
from sklearn.metrics import pairwise_distances
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_digits, load_diabetes
from scipy.stats import norm
from collections import Counter
from matplotlib import pylab as plt
import numpy as np
import pandas as pd
import seaborn as sns
```

7.2 Introducción

Los métodos paramétricos asumen que los datos provienen de un modelo común, esto da la ventaja de que el problema de estimar el modelo se limita a encontrar los parámetros del mismo, por ejemplo los parámetros de una distribución Gausiana. Por otro lado en los métodos no paramétricos asumen que datos similares se comportan de manera similar, estos algoritmos también se les conoces como algoritmos de memoria o basados en instancias.

7.3 Histogramas

El primer problema que estudiaremos será la estimación no paramétrica de una función de densidad, f, recordando que se cuenta con un conjunto $\mathcal{D} = \{x_i\}$ que es tomado de f y el objetivo es usar $\mathcal{D}$ para estimar la función de densidad $\hat{f}$.

El **histograma** es una manera para estimar la función de densidad. Para formar un histograma se divide la línea en h segmentos disjuntos, los cuales se denominan *bins*. El histograma corresponde a una función constante por partes, donde la altura es la proporción de elementos de $\mathcal{D}$ que caen en el bin analizado.

Suponiendo que todos los valores en $\mathcal{D}$ están en el rango $[0, 1]$, los bins se pueden definir como:

$$B_1 = [0, \frac{1}{m}), B_2 = [\frac{1}{m}, \frac{2}{m}), \dots, B_m = [\frac{m-1}{m}, 1],$$

donde m es el número de bins y $h = \frac{1}{m}$. Se puede definir a $\hat{p}_j = \frac{1}{N} \sum_{x \in \mathcal{D}} 1(x \in B_j)$ y $p_j = \int_{B_j} f(u)du$, donde p_j es la probabilidad del j-ésimo bin y $\hat{p}_j$ es su estimación. Usando está definición se puede definir la estimación de f como:

$$\hat{f}(x) = \sum_{j=1}^{N} \frac{\hat{p}_j}{h} 1(x \in B_j).$$

Con esta formulación se puede ver la motivación de usar histogramas como estimador de f véase:

$$\mathbb{E}(\hat{f}(x)) = \frac{\mathbb{E}(\hat{p}_j)}{h} = \frac{p_j}{h} = \frac{\int_{B_j} f(u)du}{h} \approx \frac{hf(x)}{h} = f(x).$$

7.3.1 Selección del tamaño del bin

Una parte crítica para usar un histograma es la selección de h o equivalente el número de bins del estimador. Utilizando el método descrito en Wasserman (2004), el cual se basa en minimizar el riesgo haciendo una validación cruzada, obteniendo la siguiente ecuación:

$$\hat{J}(h) = \frac{2}{(N-1)h} - \frac{N+1}{(N-1)h} \sum_{j=1}^{N} \hat{p}_j^2.$$

Para ilustrar el uso de la ecuación de minimización del riesgo se utilizará en el ejemplo utilizado en Wasserman (2004). Los datos se pueden descargar de http://www.stat.cmu.edu/~larry/all-of-statistics/=Rprograms/a1882_25.dat.

El primer paso es leer el conjunto de datos, dentro del ejemplo usado en Wasserman (2004) se eliminaron todos los datos menores a 0.2, esto se refleja en la última línea.

```
D = np.r_[[list(map(float, x.strip().split()))
          for x in open("a1882_25.dat").readlines()]]
D = D[:, 2]
D = D[D <= 0.2]
```

Haciendo un paréntesis en el ejemplo, para poder calcular $\hat{p}_j$ es necesario calcular el histograma; dado que los valores están normalizados podemos realizar el histograma utilizando algunas funciones de `numpy` y librerías tradicionales.

Para el ilustrar el método para generar el histograma se genera un histograma con 100 bins (primera línea). El siguiente paso (segunda línea) es encontrar los límites de los bins, para este proceso se usa la función `np.linspace`. En la tercera línea se encuentra el bin de cada elemento, con la característica que `np.searchsorted` regresa 0 si el valor es menor que el límite inferior y el tamaño del arreglo si es mayor. Entonces las líneas 4 y 5 se encargan de arreglar estas dos características. Finalmente se cuenta el número de elementos que pertenecen a cada bin con la ayuda de la clase `Counter`.

```
m = 100
limits = np.linspace(D.min(), D.max(), m + 1)
_ = np.searchsorted(limits, D, side='right')
_[_ == 0] = 1
_[_ == m + 1] = m
p_j = Counter(_)
```

Realizando el procedimiento anterior se obtiene el histograma presentado en la Figura 7.1

Uniendo estos elementos se puede definir una función de riesgo de la siguiente manera

```
def riesgo(D, m=10):
    """Riesgo de validación cruzada de histograma"""
    N = D.shape[0]
    limits = np.linspace(D.min(), D.max(), m + 1)
    h = limits[1] - limits[0]
    _ = np.searchsorted(limits, D, side='right')
    _[_ == 0] = 1
    _[_ == m + 1] = m
```

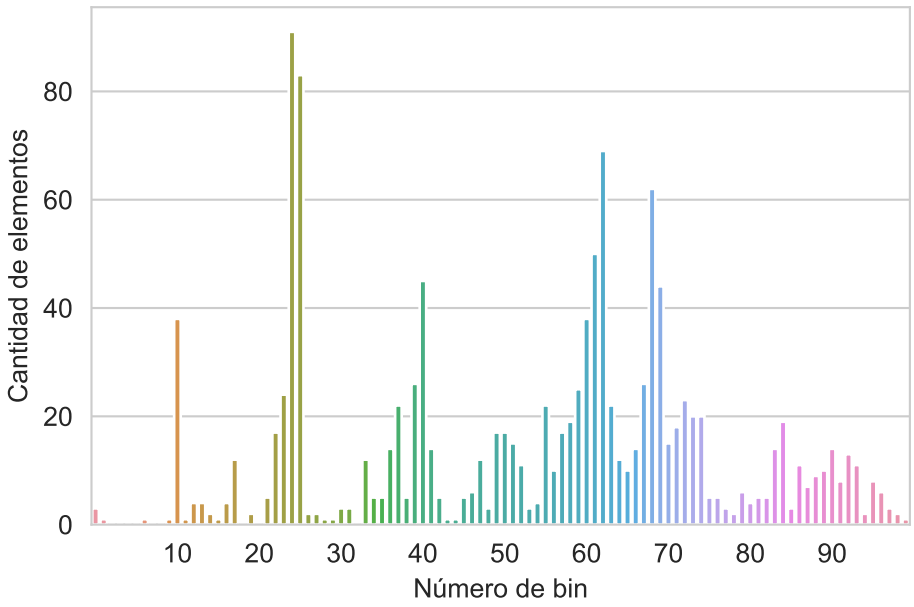


Figura 7.1: Histograma

```
    p_j = Counter(_)
    cuadrado = sum([(x / N)**2 for x in p_j.values()])
    return (2 / ((N - 1) * h)) - ((N + 1) * cuadrado / ((N - 1) * h))
```

donde las partes que no han sido descritas solamente implementan la ecuación $\hat{J}(h)$.

Finalmente se busca el valor h que minimiza la ecuación, iterando por diferentes valores de m se obtiene la Figura 7.2 donde se observa la variación del riesgo con diferentes niveles de bin.

7.4 Estimador de Densidad por Kernel

Como se puede observar el histograma es un estimador discreto, otro estimador muy utilizado que cuenta con la característica de ser suave es el estimador de densidad por kernel, K, el cual está definido de la siguiente manera.

$$\hat{f}(x) = \frac{1}{hN} \sum_{w \in \mathcal{D}} K(\frac{x - w}{h}),$$

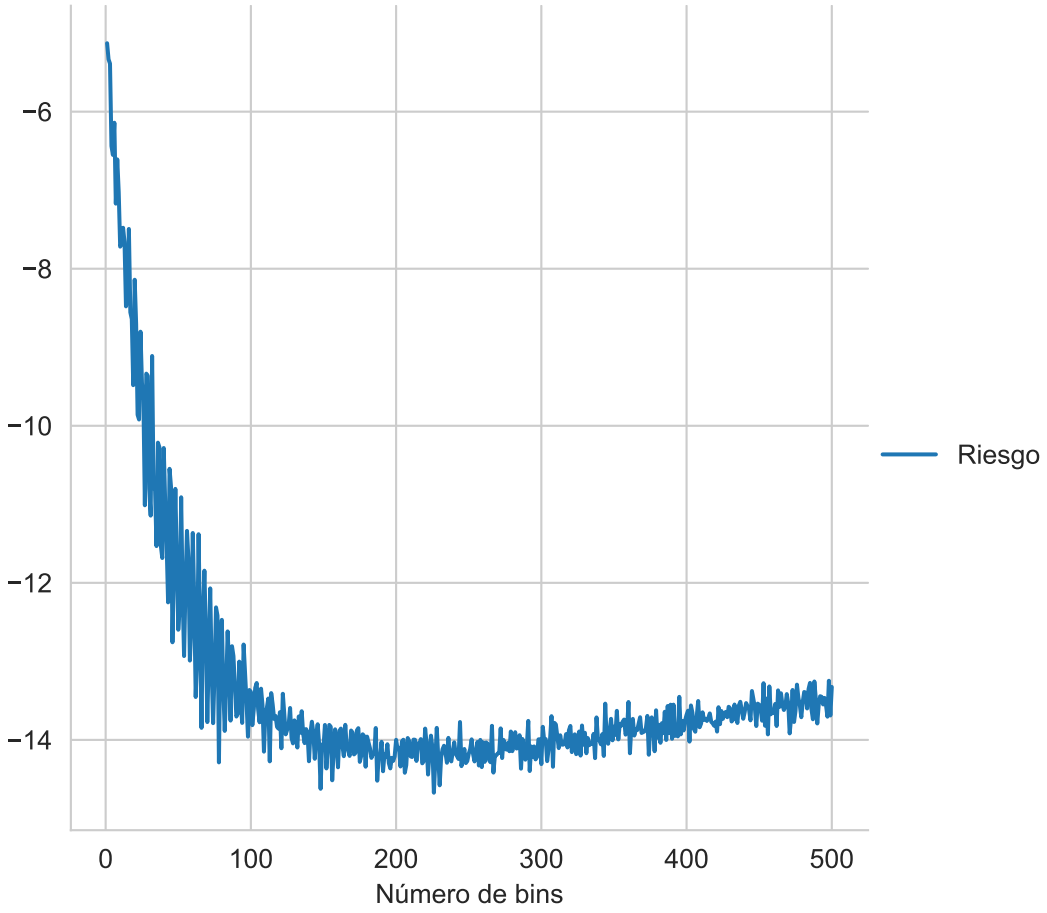


Figura 7.2: Riesgo

donde el kernel K podría ser $K(x) = \frac{1}{\sqrt{2\pi}} \exp[-\frac{x^2}{2}]$, con parámetros $\mu = 0$ y $\sigma = 1$. La Figura 7.3 muestra la estimación obtenida, con $h = 0.003$, en los datos utilizados en el ejemplo del histograma.

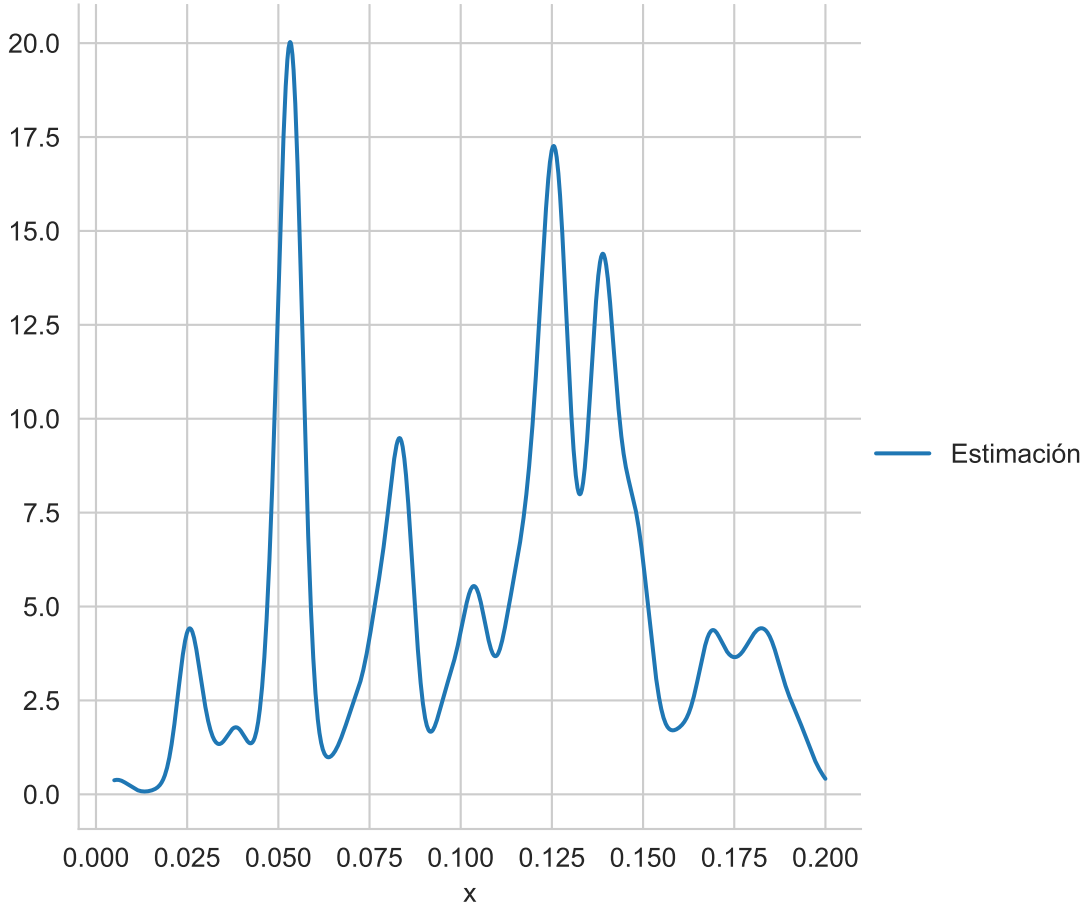


Figura 7.3: Estimación por Kernel

7.4.1 Caso multidimensional

Para el caso multidimensional el estimador quedaría como

$$\hat{f}(\mathbf{x}) = \frac{1}{h^d N} \sum_{\mathbf{w} \in \mathcal{D}} K(\frac{\mathbf{x} - \mathbf{w}}{h}),$$

donde d corresponde al número de dimensiones. Un kernel utilizado es:

$$K(\mathbf{x}) = (\frac{1}{\sqrt{2\pi}})^d \exp[-\frac{|| \mathbf{x} ||^2}{2}].$$

7.5 Estimador de Densidad por Vecinos Cercanos

Dado un conjunto $\mathcal{D} = (x_1, \ldots, x_N)$, es decir, donde se conoce la posición de x en $\mathcal{D}$ y una medida de distancia d, los k vecinos cercanos a x, $\mathsf{kNN}(x)$, se puede calcular ordenando $\mathcal{D}$ de la siguiente manera. Sea $(\pi_1, \pi_2, \ldots, \pi_N)$ la permutación tal que $x_{\pi_1} = \mathsf{arg\ min}_{w \in \mathcal{D}} d(x, w)$, donde $w_{\pi_1} \in \mathcal{D}$, π_2 corresponde al segundo índice menor, y así sucesivamente. Usando esta notación los k vecinos corresponden a $\mathsf{kNN}(x) = (x_{\pi_1}, x_{\pi_2}, \ldots, x_{\pi_k})$.

Una maneara intuitiva de definir h sería en lugar de pensar en un valor constante para toda la función, utilizar la distancia que existe con el k vecino mas cercano, es decir, $h = d(w_{\pi_k}, x) = d_k(x)$, donde $w_{\pi_k} \in \mathcal{D}$. Remplazando esto en el estimado de densidad por kernel se obtiene:

$$\hat{f}(x) = \frac{1}{d_k(x) N} \sum_{w \in \mathcal{D}} K(\frac{x - w}{d_k(x)}).$$

Utilizando los datos anteriores el estimador por vecinos cercanos, con $k = 50$, quedaría como:

7.6 Clasificador de vecinos cercanos

El clasificador de vecinos cercanos es un clasificador simple de entender, la idea es utilizar el conjunto de entrenamiento y una función de distancia para asignar la clase de acuerdo a los k-vecinos más cercanos al objeto deseado.

Utilizando la notación $kNN(x)$ se define el volumen de $kNN(x)$ como $V(x)$ y $N_c(x) = \sum_{x_\pi \in \mathsf{kNN}(x)} 1(y_\pi = c)$ donde y_π es la salida asociada a x_π. $N_c(x)$ corresponde al número de vecinos de x que pertenecen a la clase c. Con esta notación se define la verosimilitud como:

$$\mathbb{P}(\mathcal{X} = x \mid \mathcal{Y} = c) = \frac{N_c(x)}{N_c V(x)},$$

donde N_c es el número de elementos en $\mathcal{D}$ de la clase c.

Utilizando el Teorema de Bayes y sabiendo que $\mathcal{P}(Y = c) = \frac{N_c}{N}$ la probabilidad a posteriori queda como:

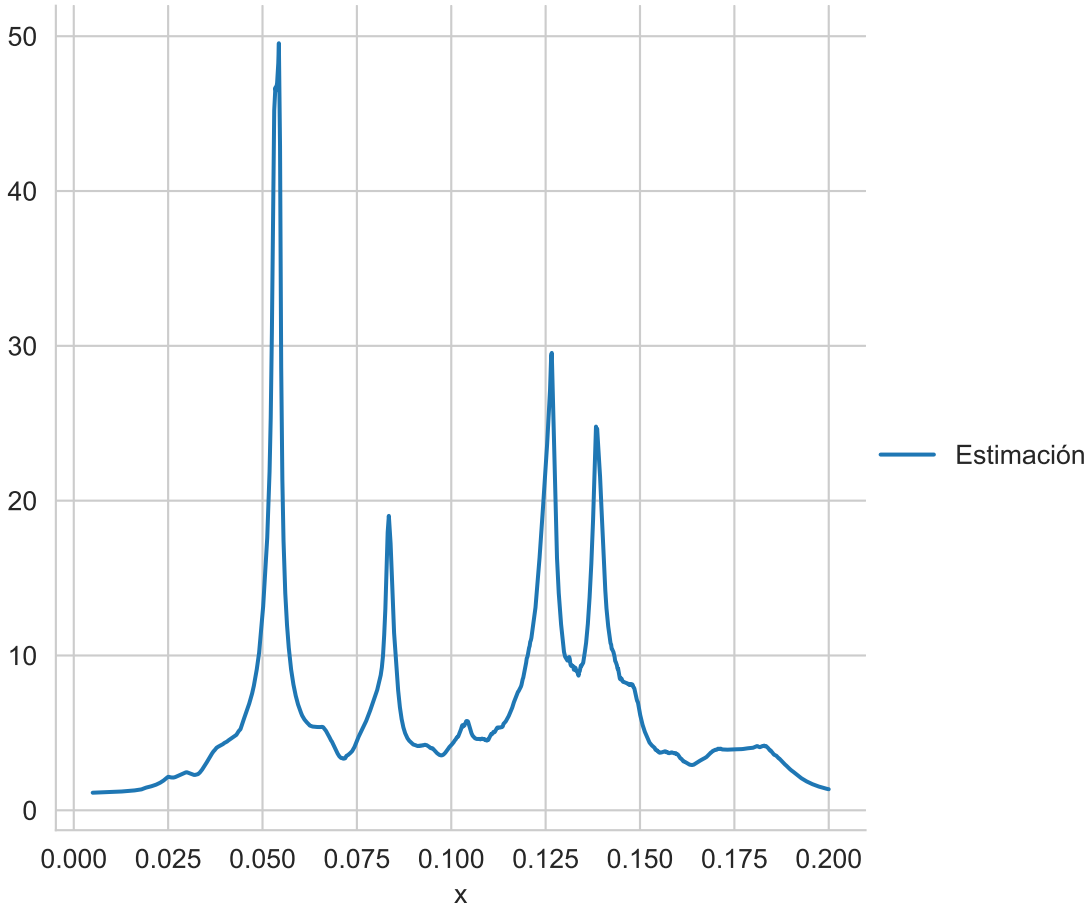


Figura 7.4: Estimación por Vecinos Cercanos

$$\begin{aligned}\mathcal{P}(\mathcal{Y} = c \mid \mathcal{X} = x) &= \frac{\frac{N_c(x)}{N_c V(x)} \frac{N_c}{N}}{\sum_u \frac{N_u(x)}{N_u V(x)} \frac{N_u}{N}} \\ &= \frac{N_c(x)}{\sum_u N_u(x)} \\ &= \frac{N_c(x)}{k},\end{aligned}$$

donde $\sum_u N_u(x) = k$ porque $N_u(x)$ corresponde al número de elementos de $\mathsf{kNN}(x)$ que pertenecen a la clase u y en total se seleccionan k elementos.

7.6.1 Implementación

El clasificador de vecinos cercanos tiene una implementación directa, aunque ineficiente, cuando el número de ejemplos en el conjunto de entrenamiento es grande. Esta implementación se ejemplifica con los datos de dígitos que se cargan y se dividen en el conjunto de entrenamiento y prueba de la siguiente manera.

```
X, y = load_digits(return_X_y=True)
T, G, y_t, y_g = train_test_split(X, y, test_size=0.2)
```

7.6.1.1 kNN

Lo primero que se realiza es la función para calcular los $\mathsf{kNN}(x)$ esto se puede generando una función `kNN` que recibe de parámetros x, el conjunto $\mathcal{D}$, la cantidad de vecinos (k) y la distancia.

El código de la función `kNN` se muestra a continuación, donde en la primera línea se convierte a x en un arreglo de dos dimensiones. Esto tiene el objetivo de generar un código que pueda buscar los k vecinos cercanos de un conjunto de puntos. Por ejemplo, se podría calcular los vecinos cercanos de todo el conjunto $\mathcal{G}$.

La segunda línea calcula los vecinos cercanos usando la función `argsort` lo único que se tiene que conocer es el eje donde se va a realizar la operación que en este caso el 0. La transpuesta es para regresar el índice de los vecinos en cada renglón.

```
def kNN(x, D, k=1, d=lambda x, y: pairwise_distances(x, y)):
    x = np.atleast_2d(x)
    return (d(D, x).argsort(axis=0))[:k].T
```

En este momento es importante mencionar que el problema de los k vecinos cercanos tiene muchas aplicaciones además del los algoritmos de aprendizaje supervisado que se verán en esta unidad. Por ejemplo, cuando uno tiene una colección de objetos que podrían ser documentos, videos, fotografías o cualquier objeto, este problema permite encontrar los objetos más cercanos a un objeto dado. Lo que se desea es que el algoritmo regrese el resultado lo antes posible y por ese motivo no se puede utilizar el algoritmo que se acaba de mencionar dado que compara x contra todos los elementos de $\mathcal{D}$. El área que estudia este tipo de problemas es el área de Recuperación de Información.

Por ejemplo, el siguiente código calcula los cinco vecinos más cercanos de los tres primeros elementos de $\mathcal{G}$.

```
kNN(G[:3], T, k=5)
```

```
array([[1264, 1399,  316,  234,   65],
       [ 659,  348,  319,  875, 1155],
       [ 937,   81,  110, 1382,    7]])
```

La manera más sencilla de crear el clasificador de vecinos cercanos es utilizando un método exhaustivo en el cálculo de distancia. Como se comentó, existen métodos más eficientes y la clase `NearestNeighbors` implementa dos de ellos adicionales al método exhaustivo. Por ejemplo, el siguiente código realiza el procedimiento equivalente al ejemplo visto previamente.

```
knn = NearestNeighbors(n_neighbors=5).fit(T)
knn.kneighbors(G[:3], return_distance=False)
```

```
array([[1264, 1399,  316,  234,   65],
       [ 659,  348,  319,  875, 1155],
       [ 937,   81,  110, 1382,    7]])
```

7.6.1.2 $N_c(x)$

El clasificador se basa en la función $N_c(x)$, esta función se implementa conociendo las etiquetas y $\mathsf{kNN}(x)$. Aunque $N_c(x)$ requiere el parámetro de la clase, la función calculará $N_c(x)$ para todas las clases. La función `N_c` recibe de parámetros todos los parámetros de `kNN` y además requiere la clases de cada elemento de $\mathcal{D}$ estas clases se dan como un arreglo adicional. El siguiente código muestra la función, donde en la primera línea se calcula los k vecinos y después se transforman los índices a las clases correspondientes, el resultado es guardado en la variable `knn`. La segunda línea usa la clase `Counter` para contar la frecuencia de cada clase en cada ejemplo dado en `x`.

```
def N_c(x, D, clases, k=1,
        d=lambda x, y: pairwise_distances(x, y)):
    knn = clases[kNN(x, D, k=k, d=d)]
    return [Counter(x) for x in knn]
```

Por ejemplo, la siguiente instrucción calcula $N_c(x)$ para todos los datos en $\mathcal{G}$ usando $k = 5$.

```
nc = N_c(G, T, y_t, k=5)
```

El elemento en el índice 100, tiene el siguiente resultado Counter({1: 5}), que indica que la clase 1, fue vista 5. El error de este algoritmo en el conjunto de prueba es 0.0139, calculado con las siguientes instrucciones. Se observa que la primera línea genera las predicciones usando la función `most_common` y a continuación se calcula el error.

```
hy = np.array([x.most_common(n=1)[0][0] for x in nc])
error = (y_g != hy).mean()
```

Una implementación del clasificador de vecinos cercanos usando métodos eficientes para calcular kNN se encuentra en la clase `KNeighborsClassifier` la cual se puede utilizar de la siguiente manera.

```
kcl = KNeighborsClassifier().fit(T, y_t)
hy = kcl.predict(G)
```

7.7 Regresión

La idea de utilizar vecinos cercanos no es solamente para problemas de clasificación, en problemas de regresión se puede seguir un razonamiento equivalente, el único cambio es en la función $N_c(x)$ donde en lugar de calcular la frecuencia de las clases de los vecinos cercanos a x se hace un promedio (pesado) de la respuesta de cada uno de los vecinos cercanos.

Para ilustrar esta adecuación en problemas de regresión se utiliza el conjunto de datos de diabetes, estos datos y los conjuntos se obtienen con las siguientes instrucciones.

```
X, y = load_diabetes(return_X_y=True)
T, G, y_t, y_g = train_test_split(X, y, test_size=0.2)
```

Se puede observar en la función `regresion` que la diferencia con clasificación es que se calcula el promedio, en lugar de contar la frecuencia de las clases.

```
def regresion(x, D, respuesta, k=1,
              d=lambda x, y: pairwise_distances(x, y)):
    knn = respuesta[kNN(x, D, k=k, d=d)]
    return knn.mean(axis=1)
```

La media del error absoluto en el conjunto $\mathcal{G}$ es 51.1888 calculado con las siguientes instrucciones.

```
hy = regresion(G, T, y_t, k=5)
error = np.fabs(y_g - hy).mean()
```

La clase equivalente a `KNeighborsClassifier` para regresión es `KNeighborsRegressor` la cual se puede utilizar asi.

```
krg = KNeighborsRegressor().fit(T, y_t)
hy = krg.predict(G)
```

8 Árboles de Decisión

El **objetivo** de la unidad es conocer y aplicar árboles de decisión a problemas de clasificación y regresión.

8.1 Paquetes usados

```
from sklearn import tree
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_breast_cancer, load_diabetes
from sklearn.inspection import DecisionBoundaryDisplay
from scipy.stats import multivariate_normal
from matplotlib import pylab as plt
import numpy as np
import pandas as pd
import seaborn as sns
```

8.2 Introducción

Los árboles de decisión son una estructura de datos jerárquica, la cual se construye utilizando una estrategia de divide y vencerás. Los árboles son un método no paramétrico diseñado para problemas de regresión y clasificación.

El árbol se camina desde la raíz hacia las hojas; en cada nodo se tiene una regla que muestra el camino de acuerdo a la entrada y la hoja indica la clase o respuesta que corresponde a la entrada.

8.3 Clasificación

Utilizando el procedimiento para generar tres Distribuciones Gausianas (Sección 2.3.1) se generan las siguientes poblaciones (Figura 8.1) con medias $\mu_1 = [5, 5]^\top$, $\mu_2 = [-5, -10]^\top$ y $\mu_3 = [15, -6]^\top$; utilizando las matrices de covarianza originales.

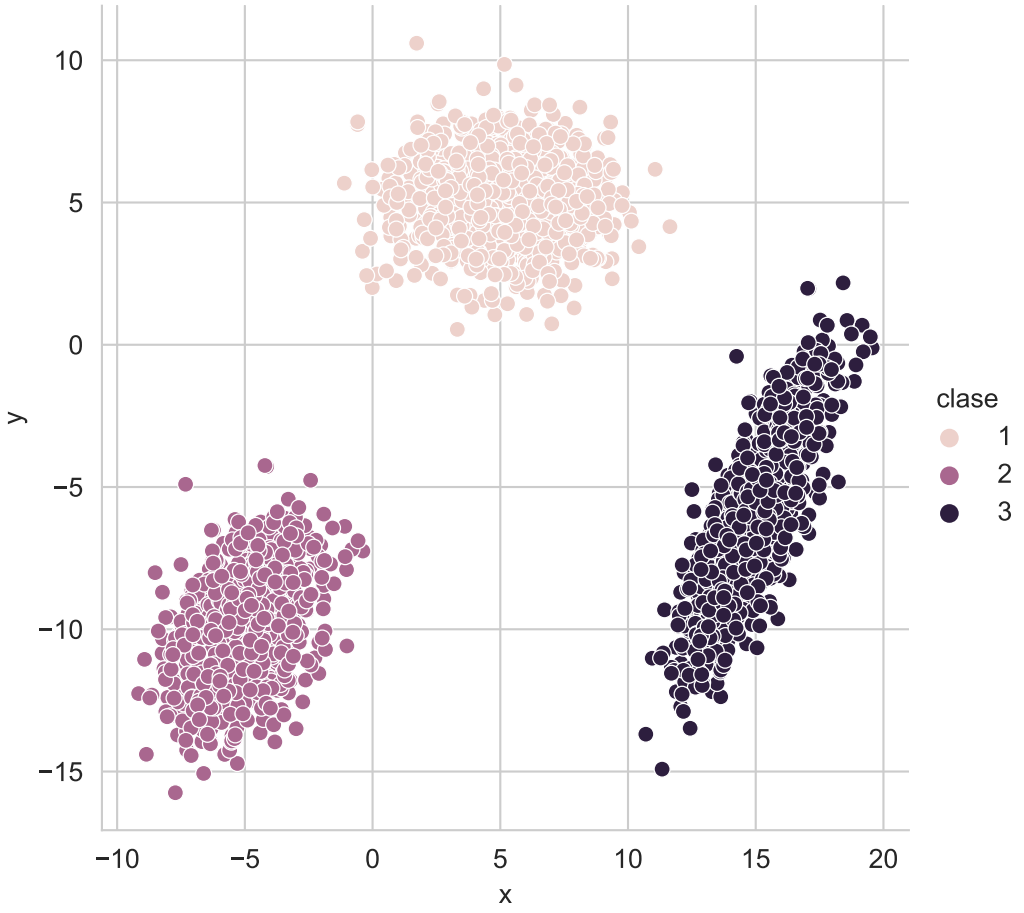


Figura 8.1: Tres distribuciones Gausianas

Con estas tres poblaciones, donde cada distribución genera una clase se crea un árbol de decisión. El árbol se muestra en la Figura 8.2, donde se observa, en cada nodo interno, la siguiente información. La primera línea muestra el identificador del nodo, la segunda corresponde a la función de corte, la tercera línea es la entropía ($H(\mathcal{Y}) = -\sum_{y \in \mathcal{Y}} \mathbb{P}(\mathcal{Y} = y) \log_2 \mathbb{P}(\mathcal{Y} = y)$), la cuarta es el número de elementos que llegaron al nodo y la última la frecuencia de cada clase en ese nodo.

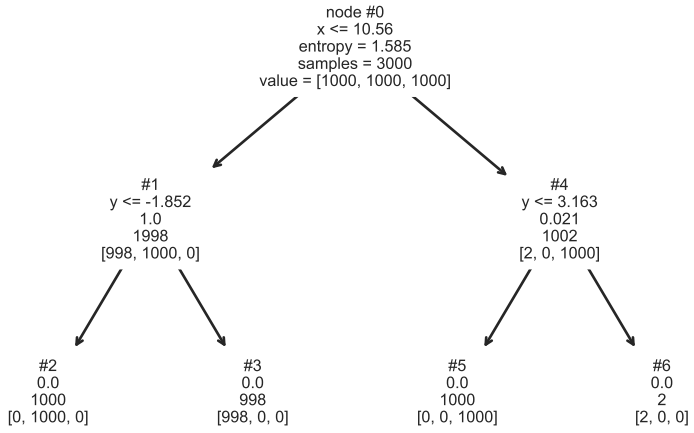


Figura 8.2: Árbol de decisión

Por ejemplo el nodo raíz del árbol tiene una entropía de 1.5850, la función de decisión es $x \leq 10.5605$ que indica que todos los elementos con un valor en x menor o igual del valor calculado están del lado izquierdo. Los hojas (nodos #2, #3, #5, y #6) no cuentan con una función de corte, dado que son la parte final del árbol. En el árbol mostrado se observa que la entropía en todos los casos es 0, lo cual indica que todos los elementos que llegaron a ese nodo son de la misma clase. No en todos los casos las hojas tienen entropía cero y existen parámetros en la creación del árbol que permiten crear árboles más simples. Por ejemplo, hay hojas que tienen muy pocos ejemplos, uno se podría preguntar ¿qué pasaría si esas hojas se eliminan? para tener un árbol más simple.

La siguiente Figura 8.3 muestra el árbol generado cuando se remueven el nodo #6. Se observa un árbol con menos nodos, aunque la entropía en es diferente de cero en algunas hojas. La segunda parte de la figura muestra la función de decisión que genera el árbol de decisión. Se observa que cada regla divide el espacio en dos usando la información que se muestra en cada nodo.

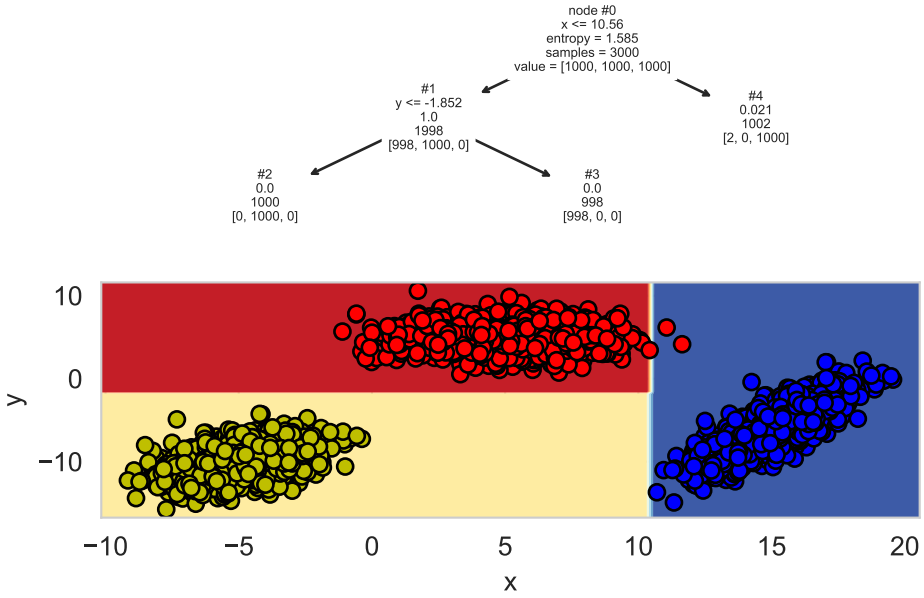


Figura 8.3: Árbol de decisión y función

8.3.1 Predicción

Utilizando el árbol y la función de decisión mostrada en Figura 8.3, se puede explicar el proceso de clasificar un nuevo elemento. Por ejemplo, el elemento $\mathbf{u} = (x = -3, y = 0.5)$ pasaría por los nodos #0, #1 y #3 para llegar a la clase correspondiente.

8.3.2 Entrenamiento

Existen diferentes sistemas para la generación de un árbol de decisión (e.g., Quinlan (1986)) la mayoría de ellos comparten las siguiente estructura general. La construcción un árbol se realiza mediante un procedimiento recursivo en donde se aplica la función de corte $f_m(\mathbf{x}) = x_i \leq a$ en el nodo m, donde el parámetro a y la componente x_i se identifican utilizando los datos que llegan al nodo m de tal manera que se maximice una función de costo.

Una función de costo podría estar basada en la entropía, es decir, para cada posible corte se mide la entropía en los nodos generados y se calcula la esperanza de la entropía de la siguiente manera.

$$L(x_i, a) = \sum_h \frac{\mid \mathcal{D}_h \mid}{\mid \mathcal{D}_m \mid} H(\mathcal{D}_h),$$

donde $H(\mathcal{D}_h)$ es la entropía de las etiquetas del conjunto $\mathcal{D}_h$, la entropía se puede calcular con la siguiente función. La función recibe un arreglo con las clases, está protegida para calcular $0 \log 0 = 0$ y finalmente regresa la entropía de `arr`.

```
def H(arr):
    a, b = np.unique(arr, return_counts=True)
    b = b / b.sum()
    return - (b * np.log2(b, where=b != 0)).sum()
```

La función que optimiza $L(x_i, a)$, para encontrar a se implementa en el procedimiento `corte_var`. Este procedimiento asume que las etiquetas (`labels`) están ordenadas por la variable x_i, es decir la primera etiqueta corresponde al valor mínimo de x_i y la última al valor máximo. Considerando esto, el valor de a es el índice con el menor costo. En la primera línea se inicializa la variable `mejor` para guardar el valor de a con mejor costo. La segunda línea corresponde a $\mid \mathcal{D}_m \mid$, en la tercera línea se identifican los diferentes valores de a que se tiene que probar, solo se tienen que probar aquellos puntos donde cuando la clase cambia con respecto al elemento adyacente, esto se calcula con la función `np.diff`; dado que está quita el primer elemento entonces es necesario incrementar 1. El ciclo es por todos los puntos de corte, se calculan el costo para los elementos que están a la izquierda y derecha del corte y se compara el resultado con el costo con menor valor encontrado hasta el momento. La última línea regresa el costo mejor así como el índice donde se encontró.

```
def corte_var(labels):
    mejor = (np.inf, None)
    D_m = labels.shape[0]
    corte = np.where(np.diff(labels))[0] + 1
    for j in corte:
        izq = labels[:j]
        der = labels[j:]
        a = (izq.shape[0] / D_m) * H(izq)
        b = (der.shape[0] / D_m) * H(der)
        perf = a + b
        if perf < mejor[0]:
          mejor = (perf, j)
    return mejor
```

En el siguiente ejemplo se usa la función `corte_var`; la función regresa un costo de 0.4591 y el punto de corte es el elemento 3, se puede observar que es el mejor punto de corte en el arreglo dado.

```
costo, indice = corte_var(np.array([0, 0, 1, 0, 0, 0]))
```

Con la función `corte_var` se optimiza el valor a de $L(x_i, a)$, ahora es el turno de optimizar x_i con respecto a la función de costo. El procedimiento `corte` encuentra el mínimo con respecto de x_i, está función recibe los índices (`idx`) donde se buscará estos valores, en un inicio `idx` es un arreglo de 0 al número de elemento del conjunto $\mathcal{D}$ menos uno. La primera línea define la variable donde se guarda el menor costo, en la segunda línea se ordenan las variables, la tercera línea se obtienen las etiquetas involucradas. El ciclo va por todas las variables x_i. Dentro del ciclo se llama a la función `corte_var` donde se observa como las etiquetas van ordenadas de acuerdo a la variable que se está analizando; la función regresa el corte con menor costo y se compara con el menor costo obtenido hasta el momento, si es menor se guarda en `mejor`. Finalmente, se regresa `mejor` y los índices ordenados para poder identificar los elementos del hijo izquierdo y derecho.

```
def corte(idx):
    mejor = (np.inf, None, None)
    orden = np.argsort(X[idx], axis=0)
    labels = y[idx]
    for i, x in enumerate(orden.T):
        comp = corte_var(labels[x])
        if comp[0] < mejor[0]:
            mejor = (comp[0], i, comp[1])
    return mejor, idx[orden[:, mejor[1]]]
```

Con la función `corte` se puede encontrar los parámetros de la función de corte $f_m(\mathbf{x}) = x_i \leq a$ para cada nodo del árbol completo del ejemplo anterior. Por ejemplo, los parámetros de la función de decisión para la raíz (#0) que se observa en la Figura 8.3 se puede obtener con el siguiente código.

```
best, orden = corte(np.arange(X.shape[0]))
perf, i, j = best
(X[orden[j], i] + X[orden[j-1], i]) / 2
```

```
10.560464864725406
```

La variable `orden` tiene la información para dividir el conjunto dado, lo cual se realiza en las siguientes instrucciones, donde `idx_i` corresponde a los elementos a la izquierda y `idx_d` son los de la derecha.

```
idx_i = orden[:j]
idx_d = orden[j:]
```

Teniendo los elementos a la izquierda y derecha, se puede calcular los parámetros de la función de corte del nodo #1 los cuales se pueden calcular con las siguientes instrucciones.

```
best, orden = corte(idx_i)
perf, i, j = best
(X[orden[j], i] + X[orden[j-1], i]) / 2
```

```
-1.8516821607950367
```

> **Nota**
>
> La función `corte` no verifica que se esté en una hoja, entonces si se hace el corte en una hora regresará `(np.inf, none, None)`

8.3.3 Ejemplo: Breast Cancer Wisconsin

Se utiliza el conjunto de datos de Breast Cancer Wisconsin para ejemplificar el algoritmo de Árboles de Decisión. Las siguientes instrucciones se descargan los datos y se dividen en los conjuntos de entrenamiento y prueba.

```
X, y = load_breast_cancer(return_X_y=True)
T, G, y_t, y_g = train_test_split(X, y, test_size=0.2)
```

La siguiente instrucción entrena un árbol de decisión utilizando como función de costo la entropía. En la librería se encuentran implementadas otras funciones como el coeficiente Gini y Entropía Cruzada Sección 4.2.7 (*Log-loss*).

```
arbol = tree.DecisionTreeClassifier(criterion='entropy').fit(T, y_t)
```

Como es de esperarse la predicción se realiza con el método `predict` como se ve a continuación.

```
hy = arbol.predict(G)
```

El error en el conjunto de prueba $\mathcal{G}$ es 0.1316, se puede comparar este error con otros algoritmos utilizados en este conjunto como clasificadores paramétricos basados en distribuciones Gausianas (Sección 3.7.3). La siguiente instrucción muestra el cálculo del error.

```
error = (y_g != hy).mean()
```

Un dato interesante, considerando los parámetros con los que se inicializó el árbol, entonces este hizo que todas las hojas fueran puras, es decir, con entropía cero. Por lo tanto el error de clasificación en el conjunto de entrenamiento $\mathcal{T}$ es cero, como se puede verificar con el siguiente código.

```
(y_t != arbol.predict(T)).mean()
```

```
0.0
```

8.4 Regresión

Los árboles de decisión aplicados a problemas de regresión siguen una idea equivalente a los desarrollados en problemas de clasificación. Para ejemplificar las diferencias se utiliza el siguiente problema sintético; el cual corresponde a la suma de un seno y un coseno como se muestra a continuación.

```
X = np.linspace(-5, 5, 100)
y = np.sin(X) + 0.3 * np.cos(X * 3.)
```

Con este problema se genera un árbol de decisión utilizando la siguiente instrucción. El método `fit` espera recibir un arreglo en dos dimensiones por eso se usa la función `np.atleast_2d` y se calcula la transpuesta siguiendo el formato esperado. Se observa el uso del parámetro `max_depth` para limitar la profundidad del árbol de decisión.

```
arbol = tree.DecisionTreeRegressor(max_depth=3).fit(np.atleast_2d(X).T, y)
```

El árbol de decisión obtenido se muestra en la Figura 8.4. La información que se muestra en cada nodo interno es equivalente a la mostrada en los árboles de clasificación. La diferencia es que en los árboles de regresión se muestra el promedio (`value`) de las salidas que llegan a ese nodo y en regresión es la frecuencia de clases. Se observa que si la entrada es $x = -4.5$ entonces la respuesta la da el nodo #4 con un valor de 1.088.

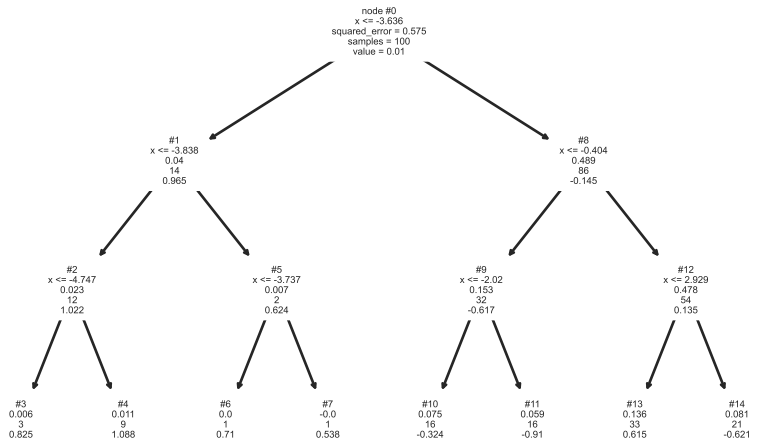


Figura 8.4: Árbol de Regresión

8.4.1 Predicción

El árbol anterior se usa para predecir todos los puntos del conjunto de entrenamiento, el resultado se muestra en la Figura 8.5. Se observa que la predicción es discreta, son escalones y esto es porque las hojas predicen el promedio de los valores que llegaron hasta ahí, en este caso el árbol tiene 8 hojas entonces a lo más ese árbol puede predecir 8 valores distintos.

8.4.2 Entrenamiento

Con respecto al proceso de entrenamiento la diferencia entre clasificación y regresión se encuentra en la función de costo que guía el proceso de optimización. En el caso de clasificación la función de costo era la esperanza de la entropía. Por otro lado, en regresión una función de costo utilizada es la varianza que es el error cuadrático que se muestra en los nodos. Para ejemplificar el uso de esta función de costo se utilizan los datos de Diabetes tal y como se muestran en las siguientes instrucciones.

```
X, y = load_diabetes(return_X_y=True)
T, G, y_t, y_g = train_test_split(X, y, test_size=0.2)
```

Con los datos de entrenamiento se genera el siguiente árbol de decisión para regresión. Solamente se muestran la información de la raíz y sus dos hijos. En la raíz se observa los parámetros de la función de corte, se selecciona la variable con índice 8 y se envían 235 elementos al hijo izquierdo y el resto al hijo derecho.

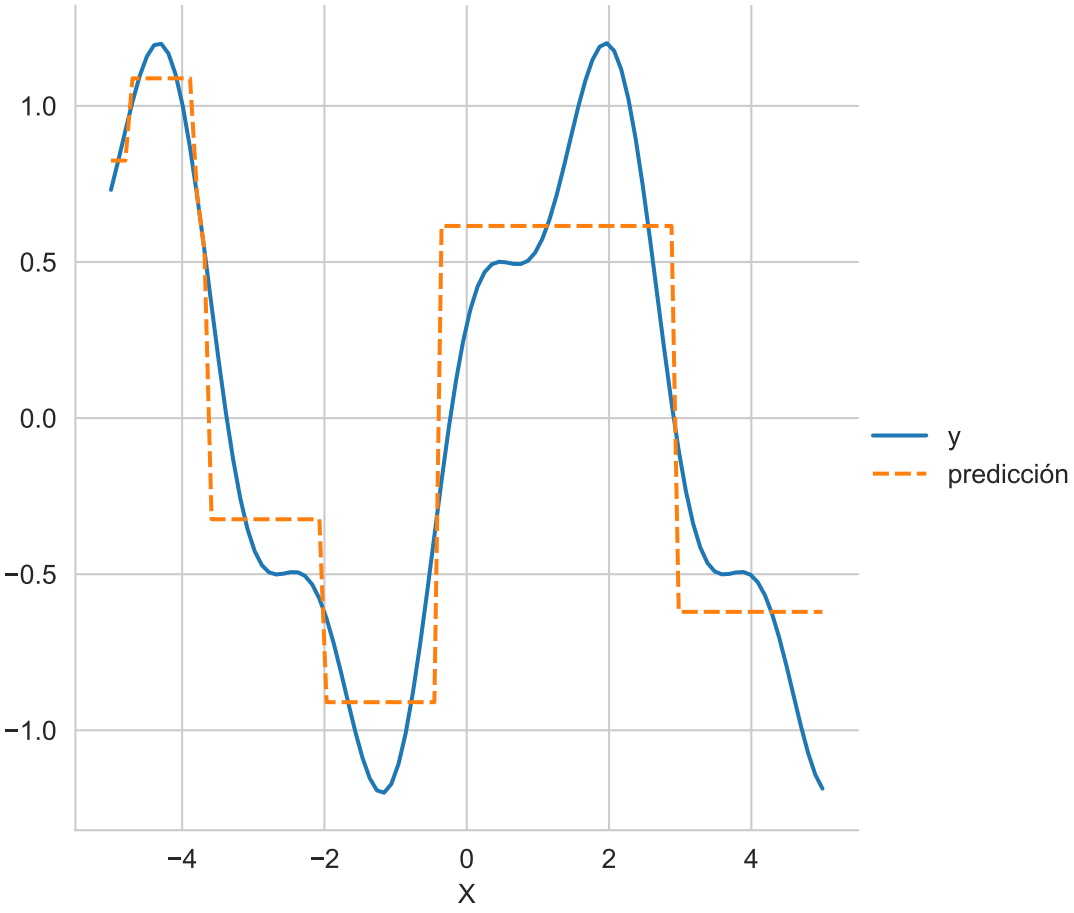
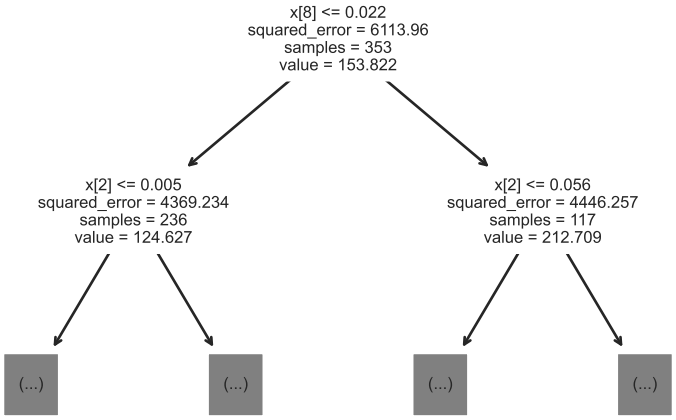


Figura 8.5: Problema de regresión

```
arbol = tree.DecisionTreeRegressor().fit(T, y_t)
_ = tree.plot_tree(arbol, max_depth=1)
```



El siguiente método implementa la función de corte para regresión se puede observar que la única diferente con la función `corte_var` definida en clasificación (Sección 8.3) es que la entropía `H` se cambia por la varianza `np.var`.

```
def corte_var(response):
    mejor = (np.inf, None)
    D_m = response.shape[0]
    corte = np.where(np.diff(response))[0] + 1
    for j in corte:
        izq = response[:j]
        der = response[j:]
        a = (izq.shape[0] / D_m) * np.var(izq)
        b = (der.shape[0] / D_m) * np.var(der)
        perf = a + b
        if perf < mejor[0]:
          mejor = (perf, j)
    return mejor
```

La función `corte_var` de regresión se utiliza para encontrar el punto de corte en los datos del conjunto de entrenamiento de la siguiente manera. En la primera línea se ordenan las variables independientes y en la segunda línea se itera por todas las variables independientes para calcular el corte con costo mínimo.

```
orden = T.argsort(axis=0)
res = [corte_var(y_t[orden[:, x]]) for x in range(10)]
res
```

```
[(5880.0170201244655, 184),
 (6005.50718971202, 31),
 (4432.754306919076, 211),
 (5160.660935428141, 234),
 (5720.628732790819, 196),
 (5825.595871884889, 204),
 (5240.6614709402975, 148),
 (4988.638691735953, 141),
 (4394.76290679546, 236),
 (5308.701388573202, 280)]
```

El resultado de ejecutar el código anterior se muestra a continuación; donde se observa que el costo mínimo corresponde a la variable con índice 8 tal y como se muestra en la figura anterior nodo derecho de la raíz.

9 Discriminantes Lineales

El **objetivo** de la unidad es conocer y aplicar diferentes métodos lineales de discriminación para atacar problemas de clasificación.

9.1 Paquetes usados

```
from sklearn.svm import LinearSVC, SVC
from sklearn.linear_model import LogisticRegression
from sklearn.decomposition import PCA
from sklearn.model_selection import train_test_split
from sklearn.metrics import f1_score, recall_score, precision_score
from scipy.stats import multivariate_normal
from sklearn.datasets import load_iris
from matplotlib import pylab as plt
import numpy as np
import pandas as pd
import seaborn as sns
```

9.2 Introducción

En unidades anteriores se han visto diferentes técnicas para discriminar entre clases; en particular se ha descrito el uso de la probabilidad $\mathbb{P}(\mathcal{Y} \mid \mathcal{X})$ para encontrar la clase más probable. Los parámetros de $\mathbb{P}(\mathcal{Y} \mid \mathcal{X})$ se han estimado utilizando métodos paramétricos y no paramétricos. En está unidad se describe el uso de funciones discriminantes para la clasificación y su similitud con el uso de $\mathbb{P}(\mathcal{Y} \mid \mathcal{X})$.

9.3 Función Discriminante

En la unidad de Teoría de Decisión Bayesiana (Capítulo 2) se describió el uso de $\mathbb{P}(\mathcal{Y} \mid \mathcal{X})$ para clasificar, se mencionó que la clase a la que pertenece $\mathcal{X} = x$ es la de mayor probabilidad, es decir,

$$C(x) = \text{argmax}_{k=1}^{K} \mathbb{P}(\mathcal{Y} = k \mid \mathcal{X} = x),$$

donde K es el número de clases y $\mathcal{Y} = k$ representa la k-ésima clase. Considerando que la evidencia es un factor que normaliza, entonces, $C(x)$ se puede definir de la siguiente manera.

$$C(x) = \text{argmax}_{k=1}^{K} \mathbb{P}(\mathcal{X} = x \mid \mathcal{Y} = k)\mathbb{P}(\mathcal{Y} = k).$$

Agrupando la probabilidad a priori y verosimilitud en una función g_k, es decir, $g_k(x) = P(\mathcal{X} = x \mid \mathcal{Y} = k)\mathbb{P}(\mathcal{Y} = k)$, hace que $C(x)$ se sea:

$$C(x) = \text{argmax}_{k=1}^{K} g_k(x).$$

Observando $C(x)$ y olvidando los pasos utilizados para derivarla, uno se puede imaginar que lo único necesario para generar un clasificador de K clases es definir un conjunto de functions g_k que separen las clases correctamente. En esta unidad se presentan diferentes maneras para definir g_k con la característica de que todas ellas son lineales, e.g., $g_k(\mathbf{x}) = \mathbf{w}_k \cdot \mathbf{x} + w_{k_0}$.

9.3.1 Clasificación Binaria

La descripción de discriminantes lineales empieza con el caso particular de dos clases, i.e., $K = 2$. En este caso $C(\mathbf{x})$ es encontrar el máximo de las dos funciones g_1 y g_2. Una manear equivalente sería definir a $C(\mathbf{x})$ como

$$C(\mathbf{x}) = \text{sign}(g_1(\mathbf{x}) - g_2(\mathbf{x})),$$

donde sign es la función que regresa el signo, entonces solo queda asociar el signo positivo a la clase 1 y el negativo a la clase 2. Utilizando esta definición se observa lo siguiente

$$\begin{aligned} g_1(\mathbf{x}) - g_2(\mathbf{x}) &= (\mathbf{w}_1 \cdot \mathbf{x} + w_{1_0}) - (\mathbf{w}_2 \cdot \mathbf{x} + w_{2_0}) \\ &= (\mathbf{w}_1 + \mathbf{w}_2) \cdot \mathbf{x} + (w_{1_0} - w_{2_0}) \quad , \\ &= \mathbf{w} \cdot \mathbf{x} + w_0 \end{aligned}$$

donde se concluye que para el caso binario es necesario definir solamente una función discriminante y que los parámetros de esta función son $\mathbf{w}$ y $\mathbf{w}_0$. Otra característica que se ilustra es que el parámetro $\mathbf{w}_0$ está actuando como un umbral, es decir, $\mathbf{x}$ corresponde a la clase positiva si $\mathbf{w} \cdot \mathbf{x} > -w_0$.

En la Figura 9.1 se observa el plano (linea) que divide las dos clases, este plano representa los puntos que satisfacen $g(\mathbf{x}) = 0$.

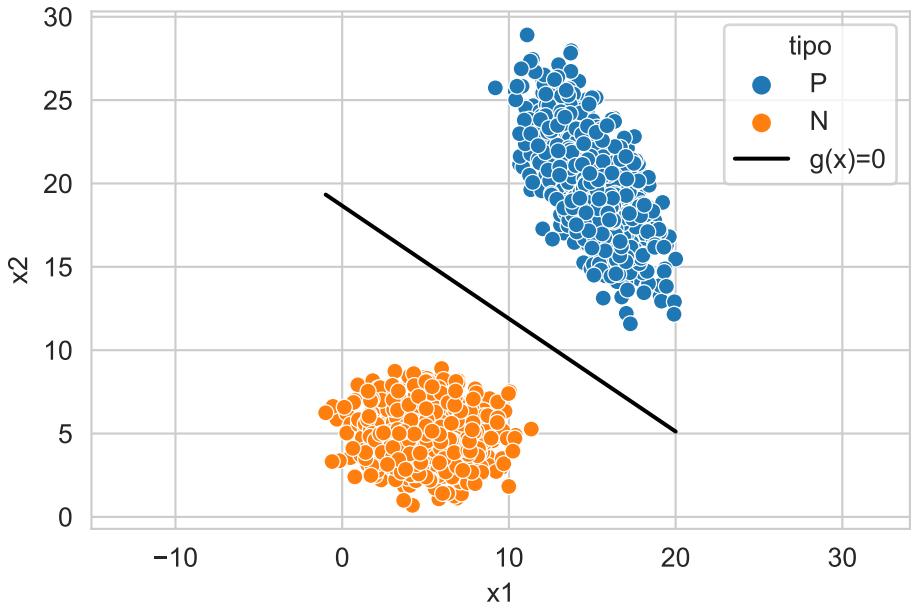


Figura 9.1: Función Discriminante

9.3.2 Geometría de la Función de Decisión

La función discriminante $g(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} + w_0$ tiene una representación gráfica. Lo primero que se observa es que los parámetros $\mathbf{w}$ viven en al mismo espacio que los datos, tal y como se puede observar en la Figura 9.2.

Siguiendo con la descripción, los parámetros $\mathbf{w}$ y la función $g(\mathbf{x})$ son ortogonales, tal y como se muestra en la Figura 9.3. Analiticamente la ortogonalidad se define de la siguiente manera. Sea $\mathbf{x}_a$ y $\mathbf{x}_b$ dos puntos en $g(\mathbf{x}) = 0$, es decir,

$$\begin{aligned} g(\mathbf{x}_a) &= g(\mathbf{x}_b) \\ \mathbf{w} \cdot \mathbf{x}_a + w_0 &= \mathbf{w} \cdot \mathbf{x}_b + w_0 \\ \mathbf{w} \cdot (\mathbf{x}_a - \mathbf{x}_b) &= 0, \end{aligned}$$

donde el vector $\mathbf{x}_a - \mathbf{x}_b$ es paralelo a $g(\mathbf{x}) = 0$, ortogonal a $\mathbf{w}$ y el sub-espacio generado por $\mathbf{w} \cdot (\mathbf{x}_a - \mathbf{x}_b) = 0$ pasa por el origen.

En la figura anterior, $\ell\mathbf{w}$ corresponde al vector $\mathbf{w}$ multiplicado por un factor ℓ de tal manera que intersecte con $g(\mathbf{x}) = 0$. El factor ℓ corresponde a la distancia que hay del origen a $g(\mathbf{x}) = 0$ la cual es $\ell = \frac{w_0}{||\mathbf{w}||}$. El signo de ℓ indica el lado donde se encuentra el origen con respecto a $g(\mathbf{x}) = 0$.

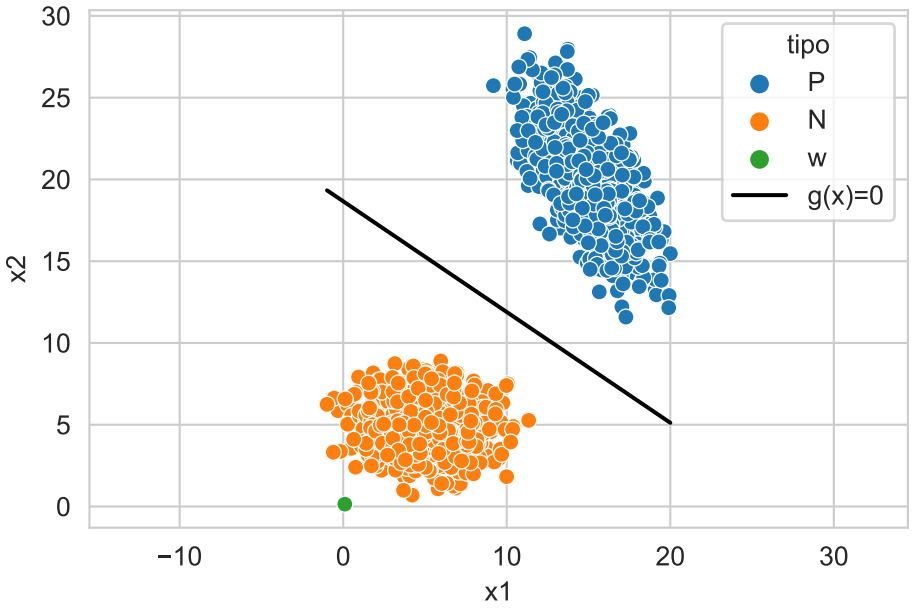


Figura 9.2: Función discriminante

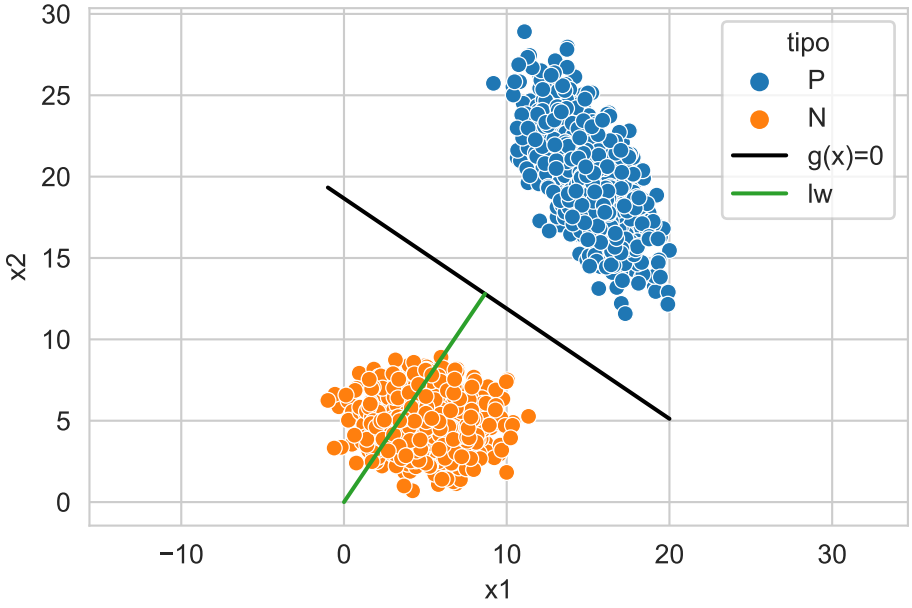


Figura 9.3: Visualizando que **w** y la función discriminante son ortogonales.

La Figura 9.4 muestra en rojo la línea generada por $\mathbf{w} \cdot \mathbf{x} = 0$, la función discriminante $g(\mathbf{x}) = 0$ (negro), la línea puntuada muestra la distancia entre ellas, que corresponde a ℓ y el vector $\mathbf{w}$. Visualmente, se observa que $\mathbf{w}$ está pegado a la línea roja, pero esto solo es un efecto de la resolución y estos elementos no se tocan.

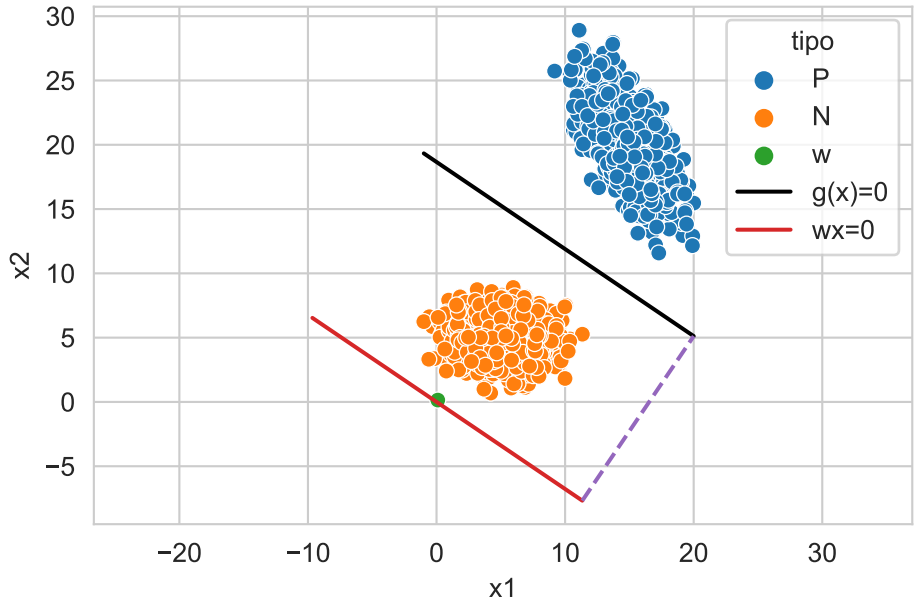


Figura 9.4: Geometría de la función discriminante.

Finalmente, será de utilidad representar a cada punto en $\mathcal{D}$ de la siguiente manera

$$\mathbf{x} = \mathbf{x}_g + \ell \frac{\mathbf{w}}{|| \mathbf{w} ||},$$

donde $\mathbf{x}_g$ corresponde a la proyección en el hiperplano ($g(\mathbf{x}) = 0$) de $\mathbf{x}$ y ℓ es la distancia que hay del hiperplano a $\mathbf{x}$. Utilizando esta representación se puede derivar la distancia ℓ de $\mathbf{x}$ con el siguiente procedimiento.

$$
\begin{aligned}
g(\mathbf{x}) &= g(\mathbf{x}_g + \ell \frac{\mathbf{w}}{||\ \mathbf{w}\ ||}) \\
&= \mathbf{w} \cdot (\mathbf{x}_g + \ell \frac{\mathbf{w}}{||\ \mathbf{w}\ ||}) + w_0 \\
&= \mathbf{w} \cdot (\mathbf{x}_g + \ell \frac{\mathbf{w}}{||\ \mathbf{w}\ ||}) \\
&= \mathbf{w} \cdot \mathbf{x}_g + \ell \mathbf{w} \cdot \frac{\mathbf{w}}{||\ \mathbf{w}\ ||} \\
&= \ell \mathbf{w} \cdot \frac{\mathbf{w}}{||\ \mathbf{w}\ ||} \\
&= \ell\ ||\ \mathbf{w}\ || \\
\ell &= \frac{g(\mathbf{x})}{||\ \mathbf{w}\ ||}
\end{aligned}
\tag{9.1}
$$

Como ya se había visto la distancia del origen al hiperplano está dada por $\ell_0 = \frac{w_0}{||\mathbf{w}||}$ y de cualquier elemento por $\ell_{\mathbf{x}} = \frac{g(\mathbf{x})}{||\mathbf{w}||}$. La Figura 9.5 muestra la $\ell_{\mathbf{x}}$ en un elemento de la clase negativa. Se puede observar el punto $\mathbf{x}_g$ que es donde intersecta la línea con el hiperplano.

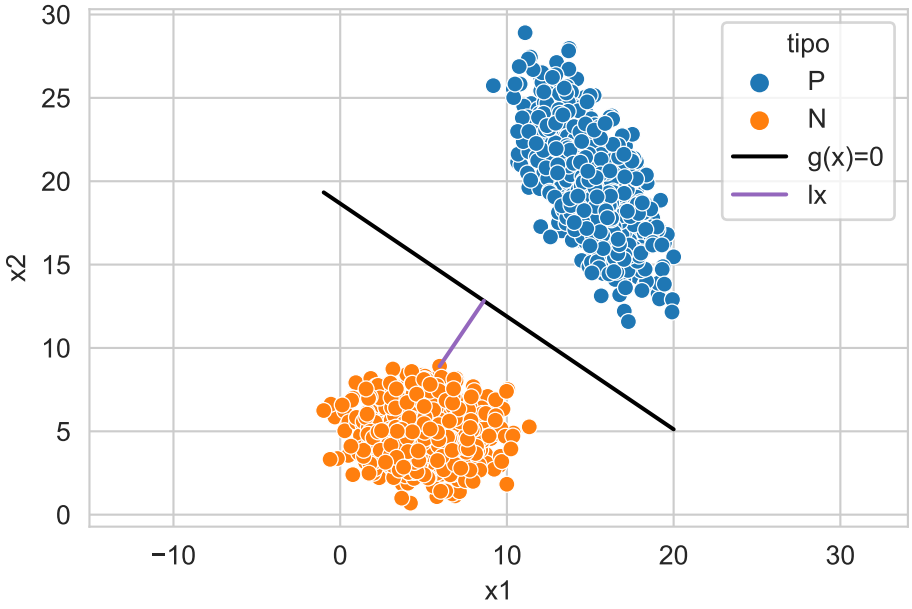


Figura 9.5: Distancia ($\ell_{\mathbf{x}} = x$) de un elemento al hiperplano

Considerando que el problema mostrado en la figura anterior está en $\mathbb{R}^2$, entonces $\mathbf{x}_g$ está dado por

$$\mathbf{x}_g = \frac{\mathbf{x} \cdot \mathbf{x}_0}{\mathbf{x}_0 \cdot \mathbf{x}_0}\mathbf{x}_0 - \ell_0 \frac{\mathbf{w}}{|| \mathbf{w} ||},$$

donde ℓ_0 es la distancia del origen al hiperplano y $\mathbf{x}_0$ es cualquier vector que está en $\mathbf{x}_0 \cdot \mathbf{w} = 0$. Para dimensiones mayores el término $\frac{\mathbf{x} \cdot \mathbf{x}_0}{\mathbf{x}_0 \cdot \mathbf{x}_0}$ es la proyección al hiperplano A tal que $A\mathbf{w} = 0$.

9.3.3 Múltiples Clases

Una manera de tratar un problema de K clases, es convertirlo en K problemas de clasificación binarios, a este procedimiento se le conoce como *Uno vs Resto*. La idea es entrenar K clasificadores donde la clase positiva corresponde a cada una de las clases y la clase de negativa se construye con todas las clases que no son la clase positiva en esa iteración. Finalmente, la clase predicha corresponde al clasificador que tiene el valor máximo en la función discriminante.

La Figura 9.6 ejemplifica el comportamiento de esta técnica en un problema de tres clases y utilizando un clasificador con discrimitante lineal. En la figura se muestra las tres funciones discriminantes $g_k(\mathbf{x}) = 0$, los parámetros escalados de esas funciones, i.e., $\ell_k \mathbf{w}_k$ y los datos. Por ejemplo se observa como la clase 1 mostrada en azul, se separa de las otras dos clases con la función $g_1(\mathbf{x}) = 0$, es decir, para $g_1(\mathbf{x}) = 0$ la clase positiva es 1 y la clase negativa corresponde a los elementos que corresponde a las clases 2 y 3.

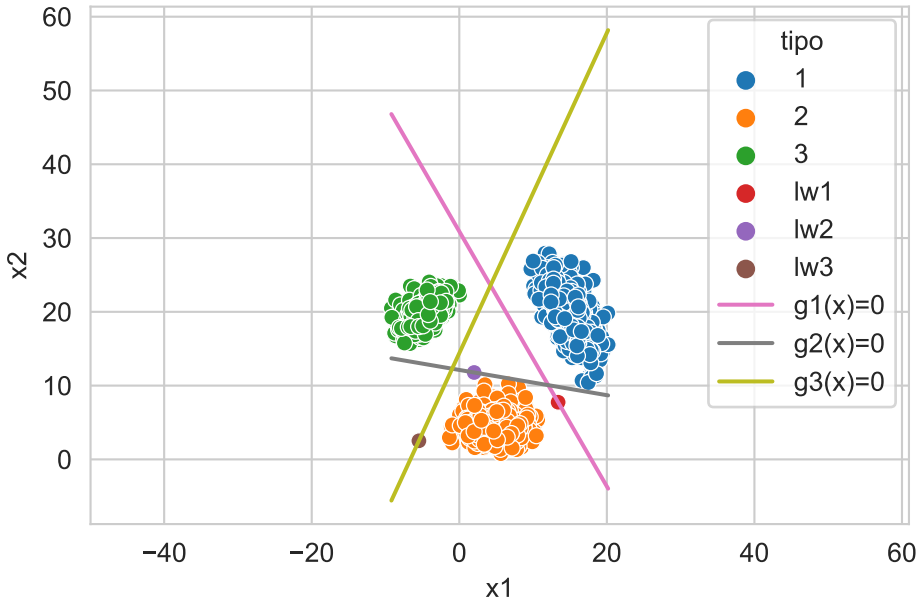


Figura 9.6: Problema multiclase

9.4 Máquinas de Soporte Vectorial

Es momento de describir algunos algoritmos para estimar los parámetros $\mathbf{w}$, y w_0 empezando por las máquinas de soporte vectorial. En este clasificador se asume un problema binario y las clases están representadas por -1 y 1, es decir, $y \in \{-1, 1\}$. Entonces, las máquinas de soporte vectorial tratan de encontrar una función con las siguientes características.

Sea $\mathbf{x}_i$ un ejemplo que corresponde a la clase 1 entonces se busca $\mathbf{w}$ tal que

$$\mathbf{w} \cdot \mathbf{x}_i + w_0 \geq +1.$$

En el caso contrario, es decir, $\mathbf{x}_i$ un ejemplo de la clase -1, entonces

$$\mathbf{w} \cdot \mathbf{x}_i + w_0 \leq -1.$$

Estas ecuaciones se pueden escribir como

$$(\mathbf{w} \cdot \mathbf{x}_i + w_0) y_i \geq +1,$$

donde $(\mathbf{x}_i, y_i) \in \mathcal{D}$.

La función discriminante es $g(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} + w_0$ y la distancia (Ecuación 9.1) que existe entre cualquier punto $\mathbf{x}_i$ al discriminante está dada por

$$\frac{g(\mathbf{x}_i)}{|| \mathbf{w} ||} y_i.$$

Entonces, se puede ver que lo que se busca es encontrar $\mathbf{w}$ de tal manera que cualquier punto $\mathbf{x}_i$ esté lo mas alejada posible del discriminante, esto se logra minimizando $\mathbf{w}$, es decir, resolviendo el siguiente problema de optimización:

$$\min \frac{1}{2} || \mathbf{w} ||$$

sujeto a $(\mathbf{w} \cdot \mathbf{x}_i + w_0) y_i \geq +1, \forall (\mathbf{x}_i, y_i) \in \mathcal{D}$.

9.4.1 Optimización

Este es un problema de optimización que se puede resolver utilizando multiplicadores de Lagrange lo cual quedaría como

$$f_p = \frac{1}{2} \mid\mid \mathbf{w} \mid\mid - \sum_{i}^{N} \alpha_i((\mathbf{w} \cdot \mathbf{x}_i + w_0)y_i - 1),$$

donde el mínimo corresponde a maximizar con respecto a $\alpha_i \geq 0$ y minimizar con respecto a $\mathbf{w}$ y w_0. En esta formulación existe el problema para aquellos problemas donde no es posible encontrar un hiperplano que separa las dos clases. Para estos casos donde no es posible encontrar una separación perfecta se propone utilizar

$$(\mathbf{w} \cdot \mathbf{x}_i + w_0)y_i \geq 1 - \xi_i,$$

donde ξ captura los errores empezando por aquellos elementos que están del lado correcto del hiperplano, pero que no son mayores a 1. La Figura 9.7 muestra un ejemplo donde existe un elemento negativo que se encuentra entre la función de decisión y el hiperplano de margen, i.e., el que corresponde a la restricción $\mathbf{w} \cdot \mathbf{x}_i + w_0 \geq 1$, es decir ese punto tiene un $0 < \xi < 1$. También se observa un elemento positivo que está muy cerca a $g(\mathbf{x}) = 1$.

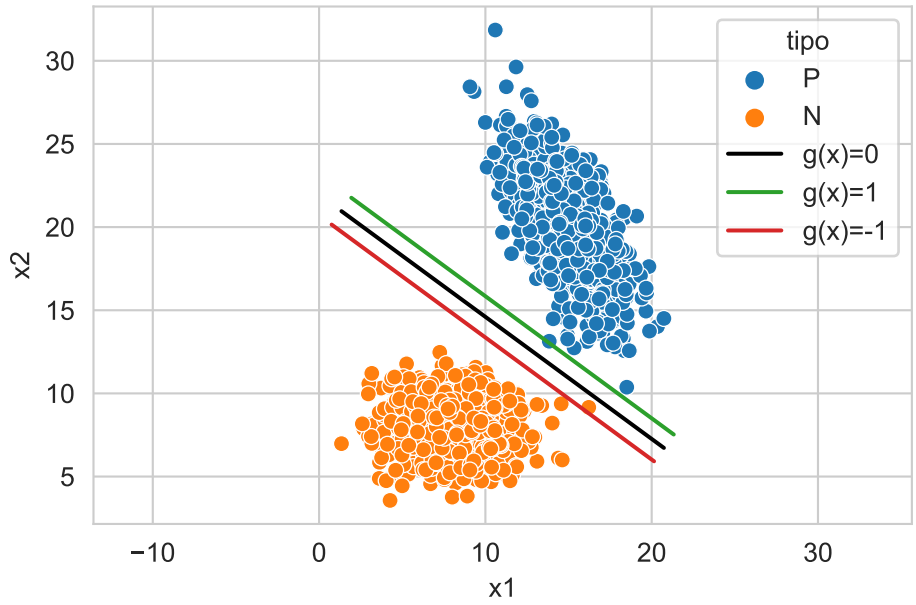


Figura 9.7: Hiperplanos

Continuando con el problema de optimización, en las condiciones anteriores la función a optimizar es $\min \frac{1}{2} \mid\mid \mathbf{w} \mid\mid + C \sum_i^N \xi_i$, utilizando multiplicadores de Lagrange queda como

$$f_p = \frac{1}{2} \mid\mid \mathbf{w} \mid\mid - \sum_i^N \alpha_i((\mathbf{w} \cdot \mathbf{x}_i + w_0)y_i - 1 + \xi_i) - \sum_i^N \beta_i \xi_i.$$

Se observa que el parámetro C controla la penalización que se hace a los elementos que se encuentran en el lado incorrecto del hiperplano o dentro del margen. La Figura 9.8 muestra el hiperplano generado utilizando $C = 1$ y $C = 0.01$. Se observa como el elemento que está correctamente clasificado en $C = 1$ pasa al lado incorrecto del hiperplano, ademas se ve como la función de decisión rota cuando el valor cambia.

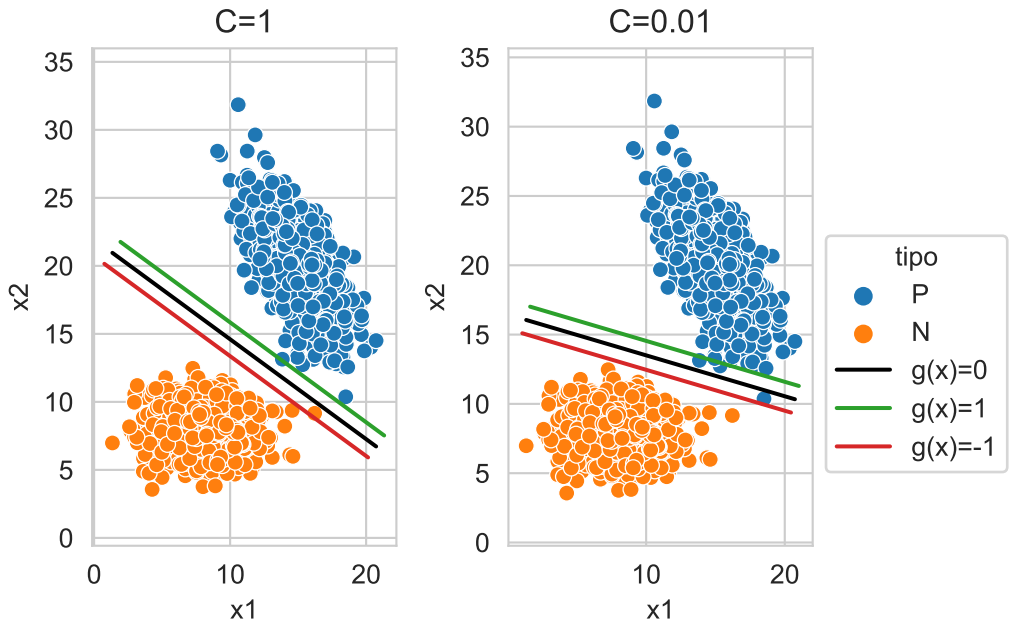


Figura 9.8: Hiperplanos para diferentes valores de C. Se observa que en $C = 0.01$ se clasifica incorrectamente un elemento positivo.

Este problema de optimización cumple con todas las características para poder encontrar su solución optimizando el problema dual. El problema dual corresponde a maximizar f_p con respecto a α_i, sujeto a que las restricciones de que el gradiente de f_p con respecto a $\mid\mid \mathbf{w} \mid\mid$, w_0 y ξ_i sean cero. Utilizando estas características el problema dual corresponde a

$$f_d = \sum_i^N \alpha_i - \frac{1}{2} \sum_i^N \sum_j^N \alpha_i \alpha_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j,$$

sujeto a las restricciones $\sum_i^N \alpha_i y_i = 0$ y $0 \leq \alpha_i \leq C$.

El problema de optimización dual tiene unas características que lo hacen deseable en ciertos casos, por ejemplo, el problema depende del número de ejemplos (N) en lugar de la dimensión. Entonces en problemas donde $d > N$ es más conveniente utilizar el dual.

9.4.2 Kernel

La otra característica del problema dual es que permite visualizar lo siguiente. Suponiendo que se usa una función $\phi : \mathbb{R}^d \leftarrow \mathbf{R}^{\hat{d}}$, de tal manera, que en el espacio ϕ se puede encontrar un hiperplano que separa las clases. Incorporando la función ϕ produce la siguiente función a optimizar

$$f_d = \sum_i^N \alpha_i - \frac{1}{2} \sum_i^N \sum_j^N \alpha_i \alpha_j y_i y_j \phi(\mathbf{x}_i) \cdot \phi(\mathbf{x}_j),$$

donde primero se transforman todos los datos al espacio generado por ϕ y después se calcula el producto punto. El producto punto se puede cambiar por una función **Kernel**, i.e., $K(\mathbf{x}_i, \mathbf{x}_j) = \phi(\mathbf{x}_i) \cdot \phi(\mathbf{x}_j)$ lo cual hace que innecesaria la transformación al espacio ϕ. Utilizando la función de kernel, el problema de optimización dual queda como:

$$f_d = \sum_i^N \alpha_i - \frac{1}{2} \sum_i^N \sum_j^N \alpha_i \alpha_j y_i y_j K(\mathbf{x}_i, \mathbf{x}_j).$$

La función discriminante está dada por $g(\mathbf{x}) = \sum_i^N \alpha_i y_i K(\mathbf{x}_i, \mathbf{x})$, donde aquellos elementos donde $\alpha \neq 0$ se les conoce como los vectores de soporte. Estos elementos son los que se encuentran en el margen, dentro del margen y en el lado incorrecto de la función discriminante.

La Figura 9.9 muestra los datos del iris (proyectados con Análisis de Componentes Principales Sección 5.5), las clases se encuentran en color azul, naranja y verde; en color rojo se muestran los vectores de soporte. La figura derecha muestra en color negro aquellos vectores de soporte que se encuentran en el lado incorrecto del hiperplano. Por otro lado se puede observar como los vectores de soporte separan las clases, del lado izquierdo se encuentran todos los elementos de la clase 0, después se observan las clases 1 y del lado derecho las clases 2. Los vectores de soporte están en la frontera de las clases y los errores se encuentran entre las clases 1 y 2 que corresponden a las que no son linealmente separables.

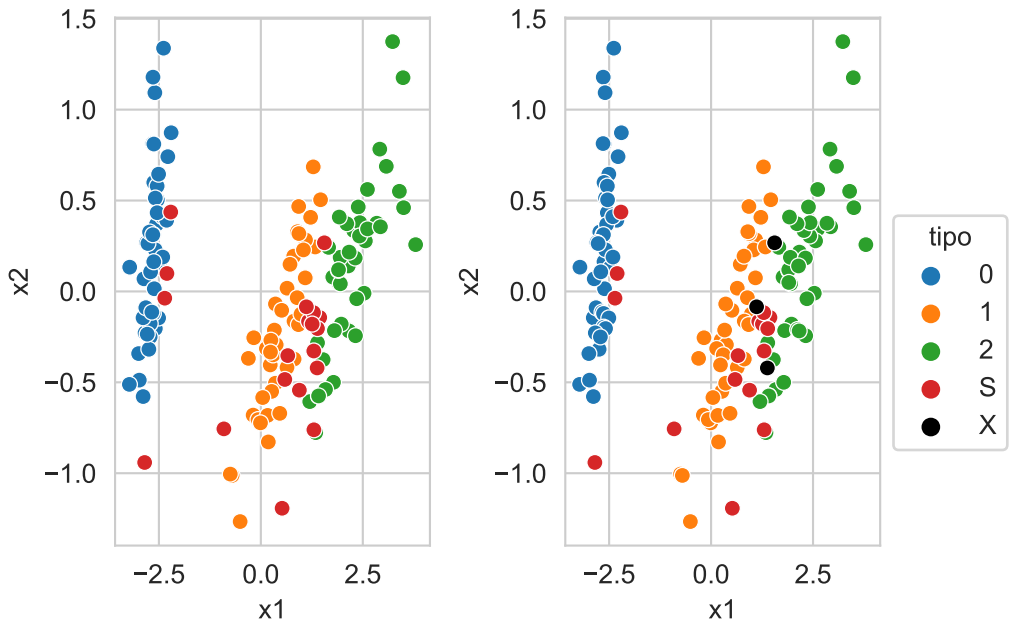


Figura 9.9: Visualización de los vectores de soporte usando PCA.

9.5 Regresión Logística

En clasificación binaria (Sección 9.3.1) se describió que la función discriminante se puede definir como la resta, i.e., $g_1(\mathbf{x}) - g_2(\mathbf{x})$; equivalentemente se pudo haber seleccionado la división $(\frac{g_1(\mathbf{x})}{g_2(\mathbf{x})})$ para generar la función discriminante o el logaritmo de la división, i.e., $\log \frac{g_1(\mathbf{x})}{g_2(\mathbf{x})}$. Esta última ecuación en el caso de $g_i(\mathbf{x}) = \mathbb{P}(\mathcal{Y} = i \mid \mathcal{X} = \mathbf{x})$ corresponde a la función logit, tal y como se muestra a continuación.

$$\begin{aligned}\log \frac{\mathbb{P}(\mathcal{Y} = 1 \mid \mathcal{X} = \mathbf{x})}{\mathbb{P}(\mathcal{Y} = 2 \mid \mathcal{X} = \mathbf{x})} &= \frac{\mathbb{P}(\mathcal{Y} = 1 \mid \mathcal{X} = \mathbf{x})}{1 - \mathbb{P}(\mathcal{Y} = 1 \mid \mathcal{X} = \mathbf{x})} \\ &= \mathsf{logit}(\mathbb{P}(\mathcal{Y} = 1 \mid \mathcal{X} = \mathbf{x})),\end{aligned}$$

donde la inversa del logit es la función sigmoide, $\mathsf{sigmoid}(x) = \frac{1}{1+\exp(-x)}$, es decir $\mathsf{sigmoid}(\mathsf{logit}(y)) = y$.

Trabajando un poco con el logit se puede observar que para el caso de dos clases está función queda como

$$\begin{aligned}\mathsf{logit}(\mathbb{P}(\mathcal{Y}=1 \mid \mathcal{X}=\mathbf{x})) &= \log \frac{\mathbb{P}(\mathcal{Y}=1 \mid \mathcal{X}=\mathbf{x})}{\mathbb{P}(\mathcal{Y}=2 \mid \mathcal{X}=\mathbf{x})} \\ &= \log \frac{\mathbb{P}(\mathcal{X}=\mathbf{x} \mid \mathcal{Y}=1)\mathbb{P}(\mathcal{Y}=1)}{\mathbb{P}(\mathcal{X}=\mathbf{x} \mid \mathcal{Y}=2)\mathbb{P}(\mathcal{Y}=2)} \\ &= \log \frac{\mathbb{P}(\mathcal{X}=\mathbf{x} \mid \mathcal{Y}=1)}{\mathbb{P}(\mathcal{X}=\mathbf{x} \mid \mathcal{Y}=2)} + \log \frac{\mathbb{P}(\mathcal{Y}=1)}{\mathbb{P}(\mathcal{Y}=2)}\end{aligned},$$

asumiendo que la matriz de covarianza (Σ) es compartida entre las dos clases la ecuación anterior quedaría como:

$$\begin{aligned}\mathsf{logit}(\mathbb{P}(\mathcal{Y}=1 \mid \mathcal{X}=\mathbf{x})) &= \log \frac{(2\pi)^{-\frac{d}{2}} \mid \Sigma \mid^{-\frac{1}{2}} \exp\left(-\frac{1}{2}(\mathbf{x}- {}_1)^\top \Sigma^{-1}(\mathbf{x}- {}_1)\right)}{(2\pi)^{-\frac{d}{2}} \mid \Sigma \mid^{-\frac{1}{2}} \exp\left(-\frac{1}{2}(\mathbf{x}- {}_2)^\top \Sigma^{-1}(\mathbf{x}- {}_2)\right)} \\ &\quad + \log \frac{\mathbb{P}(\mathcal{Y}=1)}{\mathbb{P}(\mathcal{Y}=2)} \\ &= \mathbf{w} \cdot \mathbf{x} + w_0\end{aligned}$$

donde $\mathbf{w} = \Sigma^{-1}({}_1 - {}_2)$ y $w_0 = -\frac{1}{2}({}_1 + {}_2)^\top \Sigma^{-1}({}_1 + {}_2) + \log \frac{\mathbb{P}(\mathcal{Y}=1)}{\mathbb{P}(\mathcal{Y}=2)}$.

En el caso de regresión logística, se asume que $\mathsf{logit}(\mathbb{P}(\mathcal{Y}=1 \mid \mathcal{X}=\mathbf{x})) = \mathbf{w} \cdot \mathbf{x} + w_0$ y se realiza ninguna asunción sobre la distribución que tienen los datos. Equivalentemente, se puede asumir que $\log \frac{\mathbb{P}(\mathcal{Y}=1 \mid \mathcal{X}=\mathbf{x})}{\mathbb{P}(\mathcal{Y}=2 \mid \mathcal{X}=\mathbf{x})} = \mathbf{w} \cdot \mathbf{x} + w_0^0$, realizando algunas substituciones se puede ver que $w_0 = w_0^0 + \log \frac{\mathbb{P}(\mathcal{Y}=1)}{\mathbb{P}(\mathcal{Y}=2)}$.

9.5.1 Optimización

Se puede asumir que $\mathcal{Y} \mid \mathcal{X}$ sigue una distribución Bernoulli en el caso de dos clases, entonces el logaritmo de la verosimilitud (Sección 3.3.1) quedaría como:

$$\ell(\mathbf{w}, w_0 \mid \mathcal{D}) = \prod_{(\mathbf{x},y)\in\mathcal{D}} (C(\mathbf{x}))^y (1 - C(\mathbf{x})))^{1-y},$$

donde $C(\mathbf{x})$ es la clase estimada por el clasificador.

Siempre que se tiene que obtener el máximo de una función esta se puede transformar a un problema de minimización, por ejemplo, para el caso anterior definiendo como $E = -\log \ell$, utilizando esta transformación el problema sería minimizar la siguiente función:

$$E(\mathbf{w}, w_0 \mid \mathcal{D}) = -\sum_{(\mathbf{x},y)\in\mathcal{D}} y \log C(x) + (1-y)\log(1-C(x)). \tag{9.2}$$

Es importante notar que la ecuación anterior corresponde a Entropía cruzada (Sección 4.2.7), donde $y = \mathbb{P}(\mathcal{Y} = y \mid \mathcal{X} = \mathbf{x}))$ y $C(\mathbf{x}) = \hat{\mathbb{P}}(\mathcal{Y} = y \mid \mathcal{X} = \mathbf{x})$ y los términos $1-y$ y $1-C(\mathbf{x})$ corresponde a la otra clase.

Otra característica de $E(\mathbf{w}, w_0 \mid \mathcal{D})$ es que no tiene una solución cerrada y por lo tanto es necesario utilizar un método de optimización (Capítulo 10) para encontrar los parámetros $\mathbf{w}$ y w_0.

9.6 Comparación

Es momento de comparar el comportamiento de los dos métodos de discriminantes lineales visto en la unidad, estos son, Máquinas de Soporte Vectorial (MSV) y Regresión Logística (RL). La Figura 9.10 muestra el hiperplano generado por MSV y RL, además se puede observar los valores de los pesos $\mathbf{w}$ para cada uno de los algoritmos.

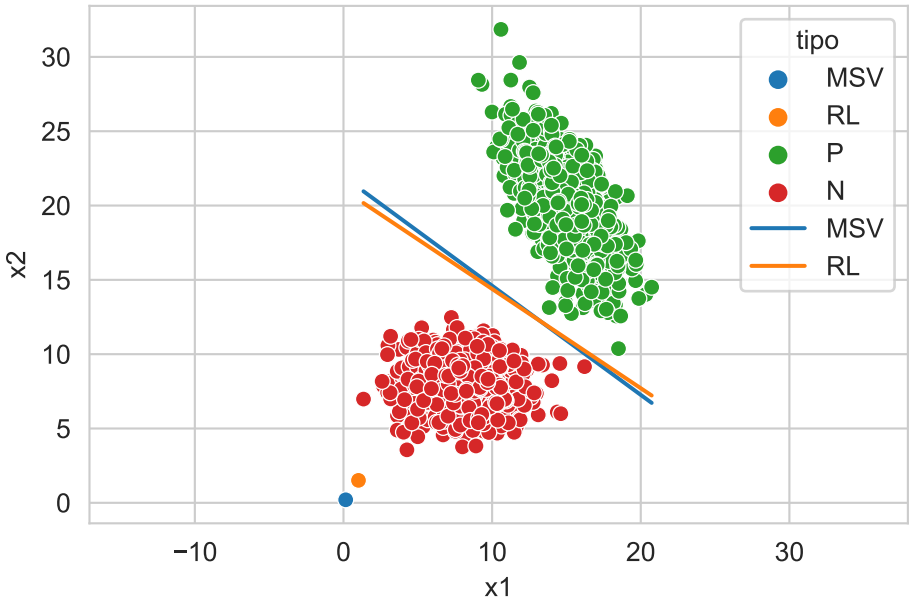


Figura 9.10: Comparación de dos métodos lineales

Complementando la comparación anterior con los datos del iris que se pueden obtener con las siguientes dos instrucciones.

```
D, y = load_iris(return_X_y=True)
T, G, y_t, y_g = train_test_split(D, y,
                                  test_size=0.4,
                                  random_state=3)
```

Los clasificadores a comparar son una máquina de soporte vectorial lineal, una máquina de soporte vectorial usando un kernel polinomial de grado 1 y una regresión logística, tal y como se muestra en el siguiente código.

```
svm = LinearSVC(dual=False).fit(T, y_t)
svm_k = SVC(kernel='poly', degree=1).fit(T, y_t)
lr = LogisticRegression().fit(T, y_t)
```

La Tabla 9.1 muestra el rendimiento (en medidas macro) de estos algoritmos en el conjunto de prueba, se puede observar que estos algoritmos tienen rendimientos diferentes para esta selección del conjunto de entrenamiento y prueba. También en esta ocasión la regresión lineal es la que presenta el mejor rendimiento. Aunque es importante aclarar que este rendimiento es resultado del proceso aleatorio de selección del conjunto de entrenamiento y prueba.

Tabla 9.1: Rendimiento de clasificadores lineales

Clasificador	Precisión	Recall	F_1
MSV - Lineal	0.9524	0.9500	0.9473
MSV - Kernel	0.9474	0.9481	0.9473
RL	0.9667	0.9667	0.9649

10 Optimización

El **objetivo** de la unidad es conocer y aplicar el método de **Descenso de Gradiente** y **Propagación hacia Atrás** par estimar los parámetros de modelos de clasificación y regresión.

10.1 Paquetes usados

```
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler, minmax_scale
from scipy.stats import multivariate_normal
from jax import grad, jit, value_and_grad, random
from jax.scipy.optimize import minimize
import jax.lax as lax
import jax.numpy as jnp
from matplotlib import pylab as plt
import numpy as np
import pandas as pd
import seaborn as sns
```

10.2 Introducción

Existen diferentes modelos de clasificación y regresión donde no es posible encontrar una solución analítica para estimar los parámetros, por ejemplo en Regresión Logística (Sección 9.5). Es en este escenario donde se voltea a métodos de optimización iterativos para calcular los parámetros.

En esta unidad se describe posiblemente el método de optimización más conocido que es **Descenso de Gradiente**. Este método como su nombre lo indica utiliza el gradiente como su ingrediente principal; se describirá como se puede calcular el gradiente utilizando un método gráfico y como este método naturalmente realiza **Propagación hacia Atrás**.

10.3 Descenso por Gradiente

En un modelo de clasificación y regresión interesa encontrar un vector de parámetros, $\mathbf{w}^*$, que minimicen una función de error, E, de la siguiente manera:

$$\mathbf{w}^* = \mathsf{argmin}_{\mathbf{w}} E(\mathbf{w} \mid \mathcal{D}).$$

En el caso de que $E(\mathbf{w} \mid \mathcal{D})$ sea una función diferenciable, el gradiente está dado por:

$$\nabla_{\mathbf{w}} E(\mathbf{w} \mid \mathcal{D}) = [\frac{\partial E}{\partial \mathbf{w}_1}, \frac{\partial E}{\partial \mathbf{w}_2}, ...]^\top.$$

La idea general es tomar la dirección opuesta al gradiente para encontrar el mínimo de la función. Entonces el cambio de parámetro está dado por

$$\begin{aligned} \Delta \mathbf{w} &= -\eta \nabla_{\mathbf{w}} E \\ \mathbf{w}^{t+1} &= \mathbf{w}^{t-1} + \Delta \mathbf{w} \\ &= \mathbf{w}^{t-1} - \eta \nabla_{\mathbf{w}} E \end{aligned}$$

10.3.1 Ejemplo - Regresión Lineal

Suponiendo que se quieren estimar los parámetros de la siguiente ecuación lineal: $f(x) = ax+b$, para lo cual se tiene un conjunto de entrenamiento en el intervalo $x = [-10, 10]$, generado con los parámetros $a = 2.3$ y $b = -3$. Importante no olvidar que los parámetros $a = 2.3$ y $b = -3$ son desconocidos y se quieren estimar usando **Descenso por Gradiente** y también es importante mencionar que para este problema en particular es posible tener una solución analítica para estimar los parámetros.

El primer paso es definir la función de error $E(a, b \mid \mathcal{D})$, en problemas de regresión una función de error viable es: $E(a, b \mid \mathcal{D}) = \sum_{(x,y)\in\mathcal{D}} (y - f(x))^2$.

La regla para actualizar los valores iniciales es: $w = w - \eta \nabla_w E$; por lo que se procede a calcular $\nabla_w E$ donde w corresponde a los parámetros a y b.

$$\begin{aligned} \frac{\partial E}{\partial w} &= \frac{\partial}{\partial w} \sum (y - f(x))^2 \\ &= 2 \sum (y - f(x)) \frac{\partial}{\partial w} (y - f(x)) \\ &= -2 \sum (y - f(x)) \frac{\partial}{\partial w} f(x) \end{aligned}$$

donde $\frac{\partial}{\partial a} f(x) = x$ y $\frac{\partial}{\partial b} f(x) = 1.0$ Las ecuaciones para actualizar a y b serían:

$$
\begin{aligned}
e(y, x) &= y - f(x) \\
a &= a + 2\eta \sum_{(x,y)\in\mathcal{D}} e(y, x)x \\
b &= b + 2\eta \sum_{(x,y)\in\mathcal{D}} e(y, x)
\end{aligned}
$$

Con el objetivo de visualizar descenso por gradiente y completar el ejemplo anterior, el siguiente código implementa el proceso de optimización mencionado. Lo primero es generar el conjunto de entrenamiento.

```
x = np.linspace(-10, 10, 50)
y = 2.3 * x - 3
```

El proceso inicia con valores aleatorios de a y b, estos valores podrían ser 5.3 y -5.1, además se utilizará una $\eta = 0.0001$, se guardarán todos los puntos visitados en la lista `D`. Las variables y valores iniciales quedarían como:

```
a = 5.3
b = -5.1
delta = np.inf
eta = 0.0001
D = [(a, b)]
```

El siguiente ciclo realiza la iteración del proceso de optimización y se detienen cuando los valores estimados varían poco entre dos iteraciones consecutivas, en particular cuando en promedio el cambio en las constantes sea menor a 0.0001.

```
while delta > 0.0001:
    hy = a * x + b
    e = (y - hy)
    a = a + 2 * eta * (e * x).sum()
    b = b + 2 * eta * e.sum()
    D.append((a, b))
    delta = np.fabs(np.array(D[-1]) - np.array(D[-2])).mean()
```

En la Figura 10.1 se muestra el camino que siguieron los parámetros hasta llegar a los parámetros que generaron el problema. Los parámetros que generaron el problema se encuentran marcados en negro.

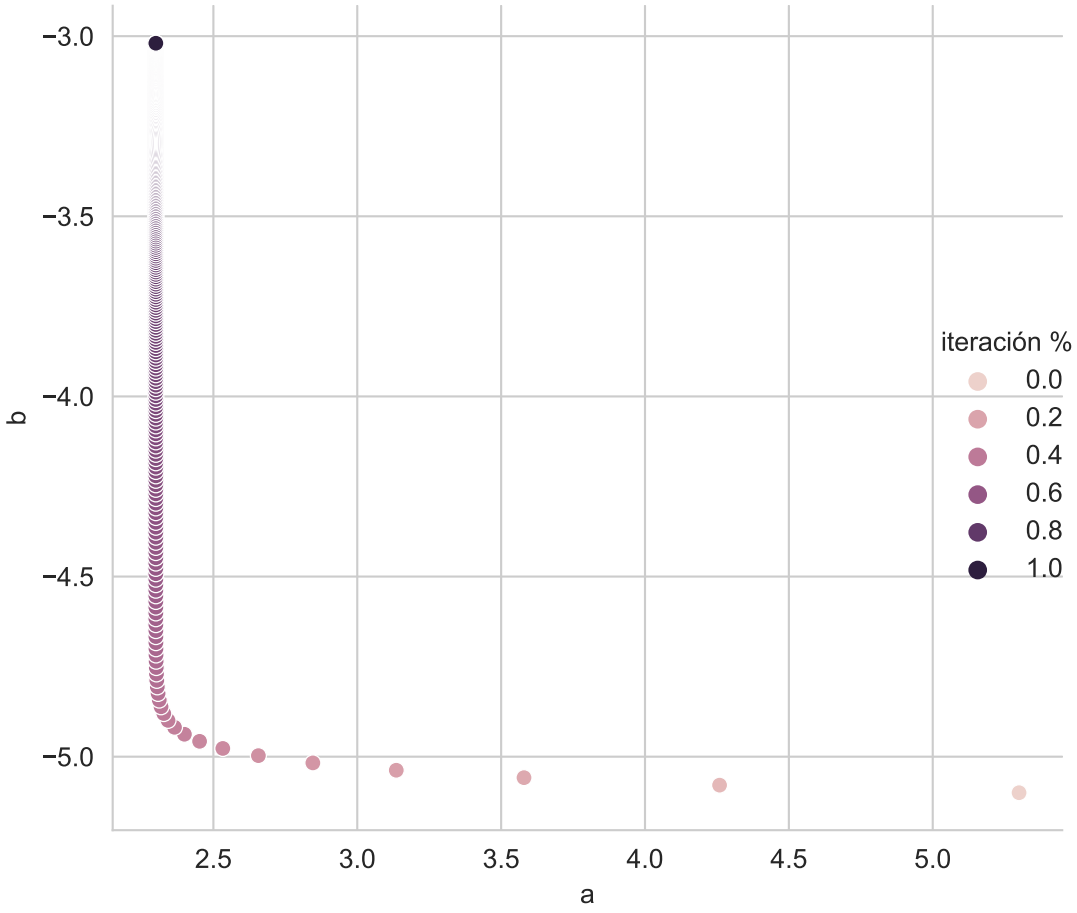


Figura 10.1: Camino de los parámetros en el proceso de optimización.

10.4 Diferenciación Automática

No en todos los casos es posible encontrar una ecuación cerrada para actualizar los parámetros; también el trabajo manual que se requiere para encontrar la solución analítica es considerable y depende de la complejidad de la función objetivo que se quiere derivar. Observando el procedimiento para actualizar los parámetros, i.e., $\mathbf{w}^{t+1} = w^{t-1} - \eta \nabla_{\mathbf{w}} E$, se concluye que se requiere evaluar $\nabla_{\mathbf{w}} E$ en un punto en particular, entonces, para solucionar el problema, es suficiente contar con un procedimiento que permita conocer el valor de $\nabla_{\mathbf{w}} E$ en cualquier punto deseado, y en particular no es necesario conocer la solución analítica de $\nabla_{\mathbf{w}} E$. Al procedimiento que encuentra el valor de la derivada de cualquier función en un punto de manera automática se le conoce como diferenciación automática.

10.4.1 Una Variable

Se puede entender el proceso de diferenciación automática siguiendo un par de ejemplos. Sea $f(x) = x^2$, en este caso $f'(x) = 2x$. Ahora considerando que se quiere evaluar $f'(2.5)$, se sabe que $f'(2.5) = 5$, pero también se puede generar un programa donde en la primera fase se calcule $f(2.5) = (2.5)^2$ y al momento de calcular f se guarde en un espacio asociado a f el valor de $f'(2.5)$. es decir 5. Se puede observar que este procedimiento soluciona este problema simple para cualquier función de una variable.

La librería JAX permite hacer diferenciación automática en Python. Siguiendo el ejemplo anterior, se genera la función $f(x) = x^2$ con el siguiente código

```
def sq(x):
    return x**2
```

Esta función se puede evaluar como `sq(2.5)`; lo interesante es que `sq` se puede componer con la función `grad` para calcular la derivada de la siguiente manera.

```
res = grad(sq)(2.5)
```

Ejecutando el código anterior se obtiene 5, `grad` internamente guarda 5, `grad` asociado a `sq`.

Ahora en el caso de una composición, por ejemplo, sea $g(x) = \sin(x)$ y se desea conocer $\frac{d}{dx} g(f(x))$, siguiendo el mismo principio primero se evalúa $f(2.5)$ y se guarda asociado a f el valor de 5 que corresponde a f', después se evalúa $g(5)$ y se guarda asociado a g el valor $\cos(2.5^2)$. Ahora para conocer el valor de $\frac{d}{dx} g(f(2.5))$ se camina en el sentido inverso multiplicando todos los valores guardados, es decir, se multiplica $5 \times \cos(2.5^2)$. Implementando este ejemplo en Python quedaría como:

```
def sin_sq(x):
    return jnp.sin(sq(x))
```

donde la función `sin_sq` tiene la composición de $g \circ f$. Ahora la derivada de esta composición en 2.5 se obtiene como `grad(sin_sq)(2.5)` obteniendo un valor de 4.9972.

10.4.2 Dos Variables

El ejemplo anterior se puede extender para cualquier composición de funciones, pero ese ejemplo no deja claro lo que pasaría con una función de dos o más variables, como por ejemplo, $h(x, y) = x \times y$. En este caso, el objetivo es calcular tanto $\frac{d}{dx}h$ como $\frac{d}{dy}h$.

El procedimiento es similar, con el ingrediente extra de que se tiene que recordar la función y la posición del argumento, es decir, para $h(2, 5)$ se asocia el valor $\frac{d}{dx}h$ con el primer argumento de la función h y $\frac{d}{dx}h$ con el segundo argumento de la función h. Para $h'(2, 5)$ se asocia el valor de 5 con el primer argumento de h y 2 con el segundo argumento de h. Se puede observar que los valores guardados corresponden a $\frac{d}{dx}h(2, 5) = y = 5$ y $\frac{d}{dy}h(2, 5) = x = 2$. Escribiendo este ejemplo en Python quedaría como

```
def prod(x):
    return x[0] * x[1]
```

calculando la derivada como `grad(prod)(jnp.array([2., 5.]))` se obtiene `[5., 2.]` que corresponde a la derivada parcial en cada argumento.

10.4.3 Visualización

Una manera de visualizar este proceso de diferenciación automática es poniéndolo en un árbol de expresión, en la Figura 10.2 se muestra la ecuación $ax^2 + bx + c$. Dentro de cada nodo se observa el valor que se tiene que guardar para cualquier valor de a, b, c y x.

10.4.4 Regresión Lineal

En esta sección se realiza utilizando diferenciación automática el ejemplo de regresión lineal (Sección 10.3.1). Lo primero que se tiene que hacer es definir la función para la cual se quieren optimizar los parámetros. Esta función es $ax + b$ la cual se define con el siguiente código. El primer argumento de la función son los parámetros y después viene los argumentos de la función que en este caso son los valores de x.

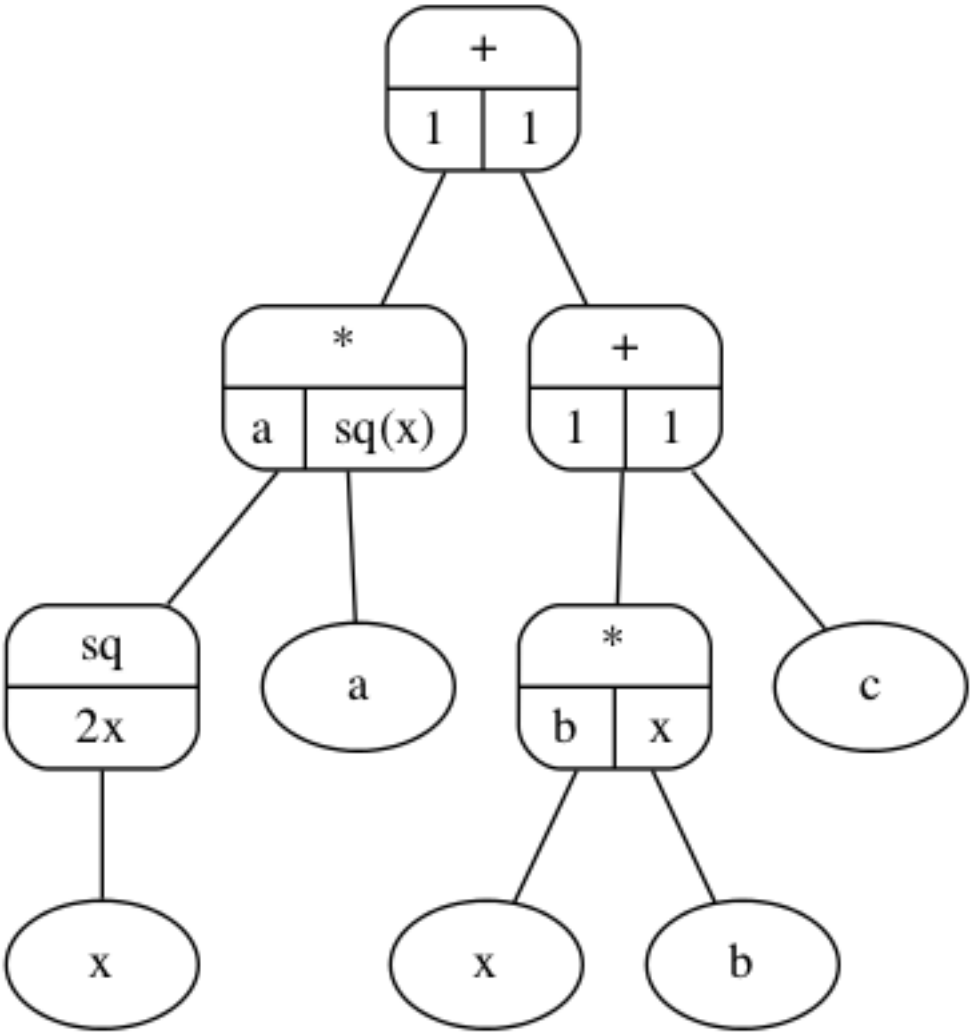


Figura 10.2: Árbol de expresión

```
def func(params, x):
    a, b = params
    return a * x + b
```

El siguiente paso es definir la función de error, en este caso la función de error definida previamente es $\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$ tal y como se muestra en la siguiente función.

```
def error(params, x, y):
    hy = func(params, x)
    return lax.fori_loop(0, x.shape[0],
                         lambda i, acc: acc + (y[i] - hy[i])**2,
                         0)
```

Finalmente, se actualizan los parámetros siguiendo un procedimiento equivalente al mostrado anteriormente. La primera linea define el valor de η, el valor de la variable `delta` indica el término del ciclo, la tercera línea define los parámetros iniciales (`params`), la siguiente guarda los todos los puntos visitados (cuarta línea) y la quinta línea deriva la función de `error` para calcular el gradiente. La primera instrucción dentro del ciclo actualiza los parámetros.

```
eta = 0.0001
delta = np.inf
params = jnp.array([5.3, -5.1])
x = jnp.array(x)
y = jnp.array(y)
D = [params]
error_grad = grad(error)
while delta > 0.0001:
    params -= eta * error_grad(params, x, y)
    D.append(params)
    delta = jnp.fabs(D[-1] - D[-2]).mean()
```

10.5 Regresión Logística

En esta sección, se presenta la implementación de Regresión Logística (Sección 9.5) utilizando diferenciación automática.

El método se probará en los datos generados por dos normales en $\mathbb{R}^2$ que se generan con las siguientes instrucciones.

```
X_1 = multivariate_normal(mean=[15, 20], cov=[[3, -3], [-3, 8]]).rvs(1000)
X_2 = multivariate_normal(mean=[8, 8], cov=[[4, 0], [0, 2]]).rvs(1000)
T = np.concatenate((X_1, X_2))
normalize = StandardScaler().fit(T)
T = jnp.array(normalize.transform(T))
y_t = jnp.array([1] * X_1.shape[0] + [0] * X_2.shape[0])
```

Siguiendo los pasos utilizados en el ejemplo anterior, la primera función a implementar es el model, es decir la función sigmoide, i.e., $\mathsf{sigmoid}(\mathbf{x}) = \frac{1}{1+\exp(-(\mathbf{w}\cdot\mathbf{x}+w_0))}$.

```
def modelo(params, x):
    _ = jnp.exp(-(jnp.dot(x, params[:2]) + params[-1]))
    return 1 / (1 + _)
```

El siguiente paso es implementar la función de error que guía el proceso de optimización, en el caso de regresión logística de dos clases la función de error corresponde a la media de la Entropía Cruzada (Ecuación 9.2). Esta función se implementa en dos pasos, primero se calcula la entropía cruzada usando el siguiente procedimiento. Donde la primera línea verifica selecciona si se trata de la clase positiva (1) o de la clase negativa 0 y calcula el $\log \hat{y}$ o $\log(1-\hat{y})$ respectivamente.

```
def entropia_cruzada(y, hy):
    _ = lax.cond(y == 1, lambda w: jnp.log(w), lambda w: jnp.log(1 - w), hy)
    return lax.cond(_ == -jnp.inf, lambda w: jnp.log(1e-6), lambda w: w, _)
```

El segundo paso es calcular la media de la entropía cruzada de cada elemento de $\mathcal{D}$, esto se puede realizar con la siguiente función.

```
def suma_entropia_cruzada(params, x, y):
    hy = modelo(params, x)
    return - lax.fori_loop(0, y.shape[0],
                           lambda i, x: x + entropia_cruzada(y[i], hy[i]),
                           1) / y.shape[0]
```

El paso final es actualizar los parámetros, en esta ocasión se utilizará la función `value_and_grad` que regresa la evaluación de la función así como la derivada, esto para poder desplegar como el error disminuye con el paso de las iteraciones. El código es similar al mostrado anteriormente. La diferencias son la forma en generar los parámetros iniciales, primeras tres líneas, el parámetro η, que se itera por $n = 5000$ iteraciones y la función `value_and_grad`.

```
key = random.PRNGKey(0)
key, subkey = random.split(key)
params = random.normal(subkey, (3,)) * jnp.sqrt(2 / 2)
eta = 0.1
error = []
error_grad  = value_and_grad(suma_entropia_cruzada)
for i in range(5000):
    _, g = error_grad(params, T, y_t)
    error.append(_)
    params -= eta * g
```

La Figura 10.3 muestra como se reduce la media de la entropía cruzada con respecto al número de iteraciones, este error sigue bajando aunque la velocidad disminuye drasticamente.

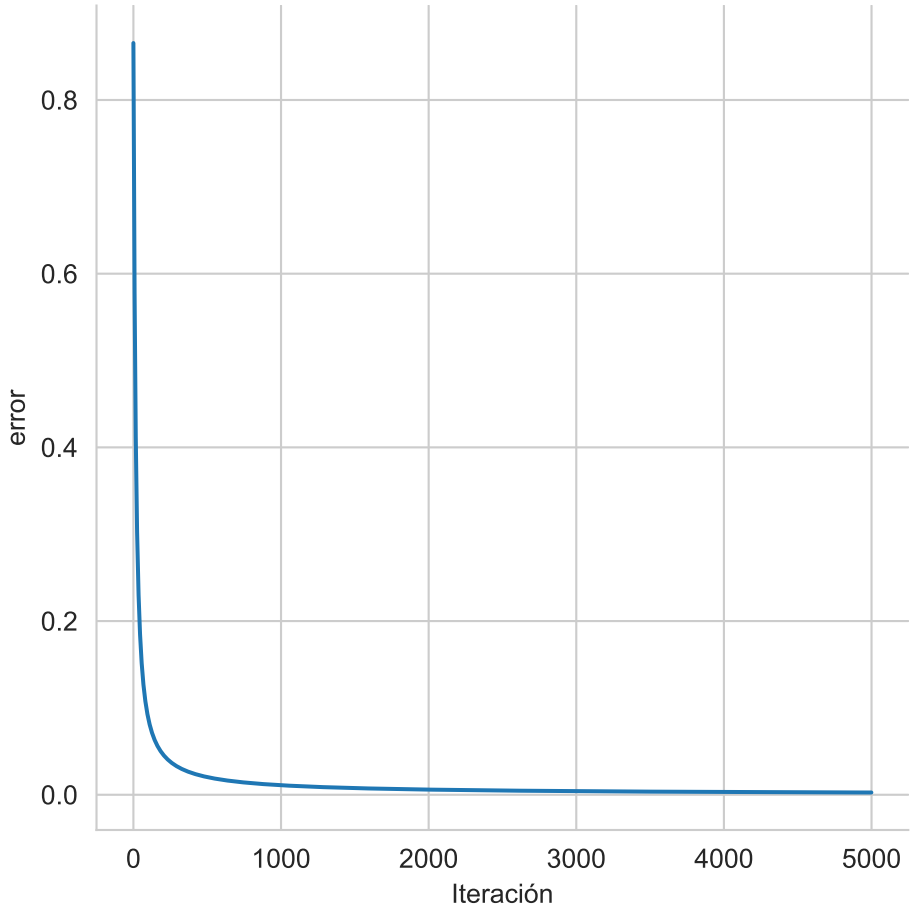


Figura 10.3: Descenso de Entropía Cruzada

Optimizando los parámetros de la regresión logística utilizando el procedimiento de diferenciación automática se obtiene [4.0584, 4.1883, -0.3099].

En Discriminantes Lineales (Capítulo 9) se presento el procedimiento para entrenar un clasificador logístico usando la siguiente instrucción.

```
lr = LogisticRegression().fit(T, y_t)
```

Los parámetros son [3.3654, 4.3295, -0.1182]. El error en el conjunto se entrenamiento es se puede calcular de la siguiente manera. En comparación el error obtenido por diferenciación automática es 0.0026, por supuesto este puede variar cambiando el número de iteraciones.

```
Array(0.00318785, dtype=float32)
```

10.6 Actualización de Parámetros

Por supuesto la regla $\mathbf{w}^{t+1} = \mathbf{w}^{t-1} - \eta \nabla_{\mathbf{w}} E$ para actualizar los parámetros $\mathbf{w}$ no es única y existe una gama de métodos que se pueden seleccionar dependiendo de las características del problema. En particular regresión logística es una problema de optimización convexo, en este tipo de problemas un algoritmos para encontrar los parámetros es el `BFGS`. Por ejemplo el siguiente código utiliza este algoritmo para encontrar los parámetros de la regresión logística.

```
p = random.normal(subkey, (3,)) * jnp.sqrt(2 / 2)
res = minimize(suma_entropia_cruzada, p, args=(T, y_t), method='BFGS')
```

Se observa como se usa la misma función de error (`suma_entropia_cruzada`) y los mismos parámetros iniciales. Los parámetros que se encuentran con este método son [18.4026, 27.2847, 3.2166] y tiene un error -0.0005.

La Figura 10.4 muestra la función discriminantes obtenida por cada uno de los métodos, el método de diferenciación automática (JAX), el método de la librearía `sklearn` y el algoritmo `BFGS`. Se puede observar que el plano de `sklearn` y `BFGS` son más similares, esto no es sorpresa porque los dos métodos implementan el mismo método de optimización, es decir, `BFGS`. Por supuesto la implementaciones tiene algunas diferencias, que pueden ir desde el número de iteraciones y la forma de generar los parámetros iniciales. Si se escalan los parámetros $\mathbf{w}$ para que tengan una longitud de 1 se observaría que `sklearn` y `BFGS` se tocan.

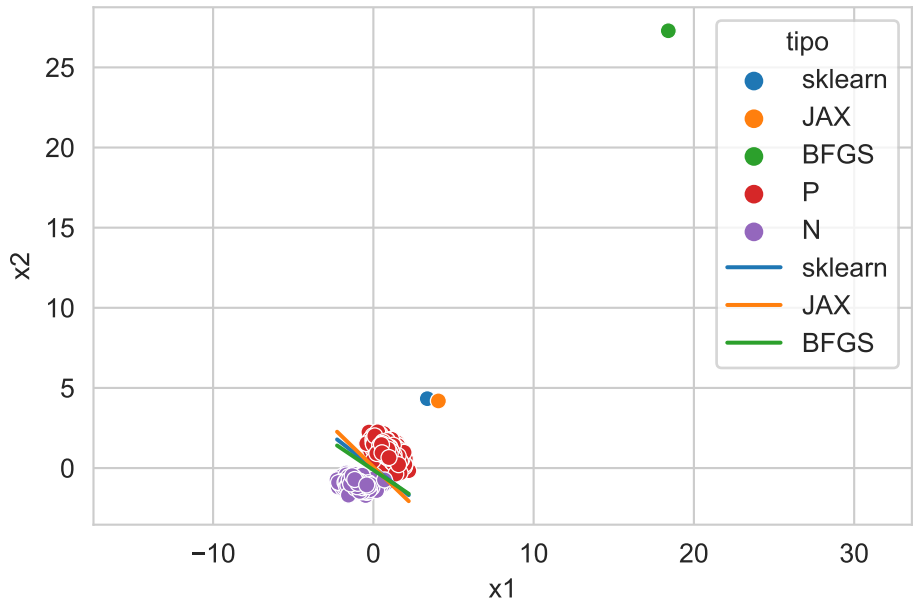


Figura 10.4: Comparación de modelos lineales con diferentes optimizadores

11 Redes Neuronales

El **objetivo** de la unidad es conocer, diseñar y aplicar redes neuronales artificiales para problemas de regresión y clasificación.

11.1 Paquetes usados

```
import jax
import jax.lax as lax
import jax.numpy as jnp
import optax
from sklearn.datasets import load_iris, load_digits
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
import numpy as np
import pandas as pd
import seaborn as sns
```

11.2 Introducción

Las redes neuronales son sin duda uno de los algoritmos de aprendizaje supervisado que mas han tomado auge en los últimos tiempos. Para iniciar la descripción de redes neuronales se toma de base el algoritmo de Regresión Logística (Sección 10.5) pero en el caso de múltiples clases, es decir, Regresión Logística Multinomial.

11.3 Regresión Logística Multinomial

La idea de regresión logística es modelar $\mathbb{P}(\mathcal{Y} = y \mid \mathcal{X} = \mathbf{x}) = \mathsf{Ber}(y \mid \mathsf{sigmoid}(\mathbf{w} \cdot \mathbf{x} + w_0))$, es decir, que la clase y esta modelada como una distribución Bernoulli con parámetro $\mathsf{sigmoid}(\mathbf{w} \cdot \mathbf{x} + w_0)$. Siguiendo está definición que es equivalente a la mostrada anteriormente, se puede modelar a un problema de multiples clases como $\mathbb{P}(\mathcal{Y} = y \mid \mathcal{X} = \mathbf{x}) = \mathsf{Cat}(y \mid \mathsf{softmax}(W\mathbf{x} + \mathbf{w}_0))$, es

decir, que la clase proviene de una distribución Categórica con parámetros $\mathsf{softmax}(W\mathbf{x}+\mathbf{w}_0)$, donde $W \in \mathbb{R}^{K\times d}$, $\mathbf{x} \in \mathbb{R}^d$ y $\mathbf{w}_0 \in \mathbb{R}^d$.

La función $\mathsf{softmax}(\mathbf{v})$, donde $\mathbf{v} = W\mathbf{x} + \mathbf{w}_0$ está definida como:

$$\mathbf{v}_i = \frac{\exp \mathbf{v}_i}{\sum_{j=1}^{K} \exp \mathbf{v}_j}.$$

La función `jax.nn.softmax` implementa softmax; en el siguiente ejemplo se calcula para el vector `[2, 1, 3]`

```
jax.nn.softmax(np.array([2, 1, 3]))
```

```
Array([0.24472848, 0.09003057, 0.66524094], dtype=float32)
```

Se puede observar que softmax transforma los valores del vector $\mathbf{v}$ en probabilidades.

Para seguir explicando este tipo de regresión logística se utilizará el problema del Iris, el cual se obtiene de la siguiente manera, es importante notar que las entradas están normalizadas para tener media cero y desviación estándar uno.

```
D, y = load_iris(return_X_y=True)
normalize = StandardScaler().fit(D)
D = normalize.transform(D)
```

El siguiente paso es generar el modelo de la Regresión Logística Multinomial, el cual depende de una matriz de coeficientes $W \in \mathbb{R}^{K\times d}$ y $\mathbf{w}_0 \in \mathbb{R}^d$. Los parámetros iniciales se puede generar con la función `parametros_iniciales` tal y como se muestra a continuación.

```
n_labels = np.unique(y).shape[0]
def parametros_iniciales(key=0):
    key = jax.random.PRNGKey(key)
    d = D.shape[1]
    params = []
    normal = jax.random.normal
    sqrt = jnp.sqrt
    for _ in range(n_labels):
        key, subkey = jax.random.split(key)
        _ = dict(w=normal(subkey, (d, )) * sqrt(2 / d),
                 w0=jnp.ones(1))
        params.append(_)
    return params
```

Utilizando los parámetros en el formato anterior, hace que el modelo se pueda implementar con las siguientes instrucciones. Donde el ciclo es por cada uno de los parámetros de las K clases y la última línea calcula el softmax.

```
@jax.jit
def modelo(params, X):
    o = []
    for p in params:
        o.append(X @ p['w'] + p['w0'])
    return jax.nn.softmax(jnp.array(o), axis=0).T
```

Una característica importante es que la función de perdida, en este caso, a la Entropía Cruzada (Sección 4.2.7), requiere codificada la probabilidad de cada clase en un vector, donde el índice con probabilidad 1 corresponde a la clase, esto se puede realizar con el siguiente código.

```
y_oh = jax.nn.one_hot(y, n_labels)
```

Ahora se cuenta con todos los elementos para implementar la función de Entropía Cruzada para múltiples clases, la cual se muestra en el siguiente fragmento.

```
@jax.jit
def media_entropia_cruzada(params, X, y_oh):
    hy = modelo(params, X)
    return - (y_oh * jnp.log(hy)).sum(axis=1).mean()
```

11.3.1 Optimización

El siguiente paso es encontrar los parámetros del modelo, para esto se utiliza el método de optimización visto en Regresión Logística (Sección 10.5) con algunos ajustes. Lo primero es que se desarrolla todo en una función `fit` que recibe el parámetro η, los parámetros a identificar y el número de épocas, es decir, el número de iteraciones que se va a realizar el procedimiento.

Dentro de la función `fit`se observa la función `update` que calcula los nuevos parámetros, también regresa el valor del error, esto para poder visualizar como va aprendiendo el modelo. La primera linea después de la función `update` genera la función que calculará el valor y el gradiente de la función `media_entropia_cruzada`. Finalmente viene el ciclo donde se realizan las actualizaciones de los parámetros y se guarda el error calculado en cada época.

```
def fit(eta, params, epocas=500):
    @jax.jit
    def update(params, eta, X, y_oh):
        _ , gs = error_grad(params, X, y_oh)
        return _, jax.tree_map(lambda p, g: p - eta * g,
                               params, gs)

    error_grad  = jax.value_and_grad(media_entropia_cruzada)
    error = []
    for i in range(epocas):
        _, params = update(params, eta, D, y_oh)
        error.append(_)
    return params, error
```

11.3.2 Optimización Método Adam

Como se había visto en la Sección 10.6 existen diferentes métodos para encontrar los parámetros, en particular en esta sección se utilizará el método Adam (implementado en la librería optax) para encontrar los parámetros de la Regresión Logística Multinomial. Se decide utilizar este método dado que su uso es frecuente en la identificación de parámetros de redes neuronales.

Siguiendo la misma estructura que la función `fit`, la función `adam` recibe tres parámetros el primer es la instancia de optimizador, la segunda son los parámetros y finalmente el número de épocas que se va a ejecutar. La primera línea de la función `update` (que se encuentra en `adam`) calcula el valor de la función de error y su gradiente, estos son utilizados por `optimizer.update` para calcular la actualización de parámetros así como el nuevo estado del optimizador, los nuevos parámetros son calculados en la tercera línea y la función regresa los nuevos parámetros, el estado del optimizador y el error en esa iteración. La primera línea después de `update` inicializa el optimizador, después se general la función que calculará el valor y gradiente de la función `media_entropia_cruzada`. El ciclo llama la función `update` y guarda el error encontrado en cada época.

```
def adam(optimizer, params, epocas=500):
    @jax.jit
    def update(params, opt_state, X, y_oh):
        loss_value, grads = error_grad(params,
                                       X,
                                       y_oh)
        updates, opt_state = optimizer.update(grads,
                                              opt_state,
                                              params)
```

```
        params = optax.apply_updates(params, updates)
        return params, opt_state, loss_value

    opt_state = optimizer.init(params)
    error_grad  = jax.value_and_grad(media_entropia_cruzada)
    error = []
    for i in range(epocas):
        params, opt_state, loss_value = update(params,
                                               opt_state,
                                               D,
                                               y_oh)
        error.append(loss_value)
    return params, error
```

11.3.3 Comparación entre Optimizadores

Los optimizadores descritos anteriormente se pueden utilizar con el siguiente código, donde la primera línea calcula los parámetros iniciales, después se llama a la función `fit` para encontrar los parámetros con el primer método. La tercera línea genera una instancia del optimizador Adam; el cual se pasa a la función `adam` para encontrar los parámetros con este método.

```
params = parametros_iniciales()
p1, error1 = fit(1e-2, params)
optimizer = optax.adam(learning_rate=1e-2)
p2, error2 = adam(optimizer, params)
```

La Figura 11.1 muestra cómo la media de la Entropía Cruzada se minimiza con respecto a las épocas para los dos métodos. Se puede observar como el método `adam` converge más rápido y llega a un valor menor de Entropía Cruzada.

Finalmente la exactitud (Sección 4.2.2) en el conjunto de entrenamiento del modelo estimado con `fit` es 0.8867 (i.e., `(y == modelo(p1, D).argmax(axis=1)).mean()`) y del estimado con `adam` es 0.9667.

11.4 Perceptrón

La unidad básica de procesamiento en una red neuronal es el perceptrón, el cual es un viejo conocido de Discriminantes Lineales (Capítulo 9), es decir, $g(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} + \mathbf{w}_0$. En problemas de clasificación binaria se encuentran los parámetros de $g(\mathbf{x})$ de tal manera que genera un hiperplano y se clasifican los elementos de acuerdo al lado positivo o negativo del hiperplano.

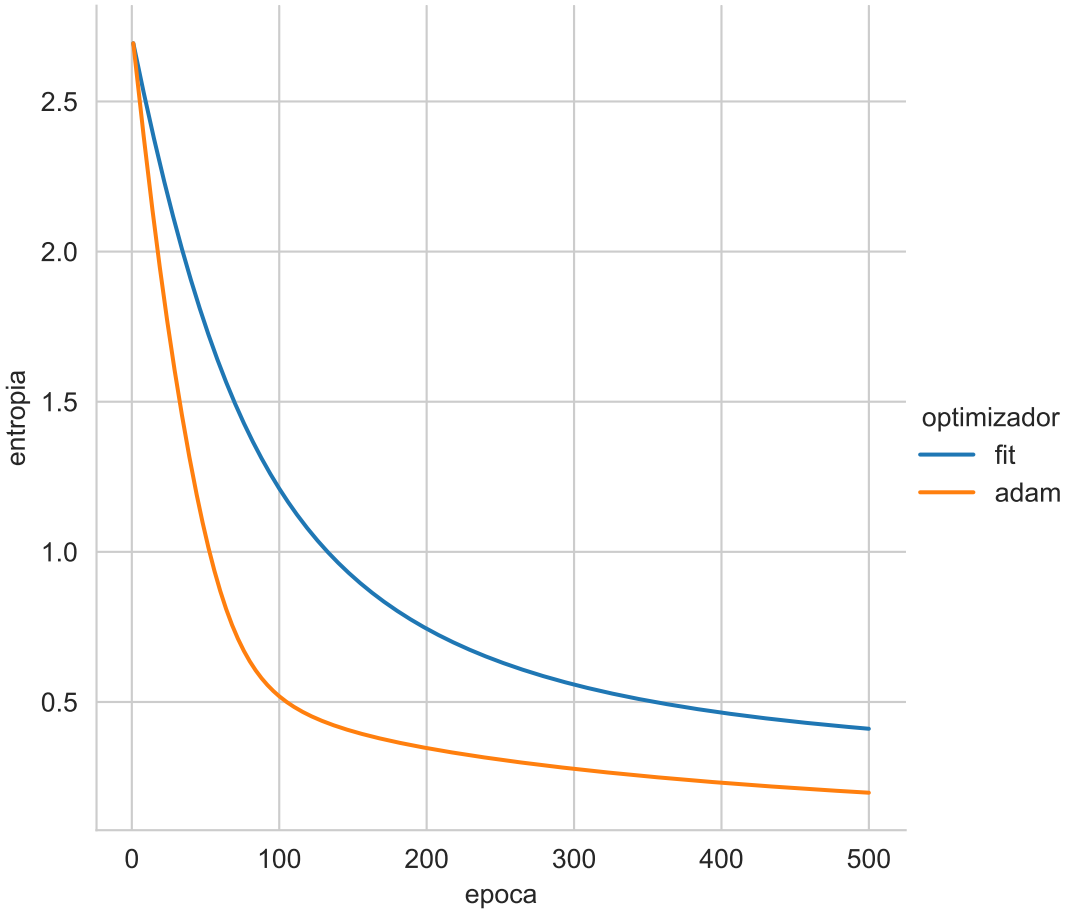


Figura 11.1: Comparación de diferentes optimizadores.

En problemas de Regresión (Sección 3.9) los parámetros de $g(\mathbf{x})$ se encuentran utilizando mínimos cuadrados.

En el case de tener $K > 2$ clases entonces el problema se puede afrontar entrenando $g_k(\mathbf{x})$ perceptrones $(k = 1, \dots, K)$ tal y como se realizó Discriminantes Lineales (Sección 9.3.3). De manera concisa se puede definir a $g : \mathbb{R}^d \to \mathbb{R}^K$, es decir, $g(\mathbf{x}) = W\mathbf{x} + \mathbf{w}_0$ tal y como se realizó en Regresión Logística Multinomial (Sección 11.3). En el caso de desear conocer la probabilidad de pertenencia a una clase, en el caso binario se utilizó $g(\mathbf{x}) = \mathsf{sigmoid}(\mathbf{w} \cdot \mathbf{x} + \mathbf{w}_0)$ y en el caso multiclase $g(\mathbf{x}) = \mathsf{softmax}(W\mathbf{x} + \mathbf{w}_0)$.

11.4.1 Composición de Perceptrones Lineales

Se puede realizar una composición de perceptrones de la siguiente manera, sea $g_1 : \mathbb{R}^d \to \mathbb{R}^{d'}$ y $g_2 : \mathbb{R}^{d'} \to \mathbb{R}^K$, es decir $g = g_2 \circ g_1$. Realizando está composición en las ecuaciones descritas anteriormente se tiene

$$\begin{aligned} \hat{\mathbf{y}}_1 &= g_1(\mathbf{x}) \\ \hat{\mathbf{y}} &= g_2(\hat{\mathbf{y}}_1) \end{aligned}$$

Expandiendo las ecuaciones anteriores se tiene

$$\begin{aligned} \hat{\mathbf{y}}_1 &= W_1\mathbf{x} + \mathbf{w}_{1_0} \\ \hat{\mathbf{y}} &= W_2\hat{\mathbf{y}}_1 + \mathbf{w}_{2_0} \\ &= W_2(W_1\mathbf{x} + \mathbf{w}_{1_0}) + \mathbf{w}_{2_0} \\ &= W_2W_1\mathbf{x} + W_2\mathbf{w}_{1_0} + \mathbf{w}_{2_0} \\ &= W\mathbf{x} + \mathbf{w}_0 \end{aligned}$$

como se puede observar la composición realizada da como resultado una red donde se tienen que identificar $W \in \mathbb{R}^{K \times d}$ y $\mathbf{w}_0 \in \mathbb{R}^d$, es decir, son K perceptrones equivalentes al modelado de Regresión Logística Multinomial. Esto es porque la composición fue con funciones lineales.

11.5 Perceptrón Multicapa

Para evitar que la composición de perceptrones colapsen a una función equivalente, es necesario incluir una función no lineal, sea ϕ esta función no lineal, (a esta función se le conoce como **función de activación**) entonces se puede observar que la composición $g = g_2 \circ g_1$ donde $g_1 = \phi(W_1\mathbf{x} + \mathbf{w}_{1_0})$ resulta en

$$\begin{aligned}\hat{\mathbf{y}}_1 &= \phi(W_1\mathbf{x} + \mathbf{w}_{1_0}) \\ \hat{\mathbf{y}} &= W_2\hat{\mathbf{y}}_1 + \mathbf{w}_{2_0}\end{aligned}.$$

A la estructura anterior se le conoce como una red neuronal de una capa oculta, la salida de la capa oculta está en $\hat{\mathbf{y}}_1$ y la salida de la red es $\hat{\mathbf{y}}$. Siguiendo la notación anterior se puede definir una red neuronal con dos capas ocultas de la siguiente manera

$$\begin{aligned}\hat{\mathbf{y}}_1 &= \phi(W_1\mathbf{x} + \mathbf{w}_{1_0}) \\ \hat{\mathbf{y}}_2 &= \phi_2(W_2\hat{\mathbf{y}}_1 + \mathbf{w}_{2_0}), \\ \hat{\mathbf{y}} &= (W_3\hat{\mathbf{y}}_2 + \mathbf{w}_{3_0})\end{aligned}$$

donde la salida de la primera capa oculta ($\hat{\mathbf{y}}_1$) es la entrada de la segunda capa oculta y su salida ($\hat{\mathbf{y}}_2$) se convierte en la entrada de la capa de salida.

11.5.1 Desvanecimiento del Gradiente

Por lo expuesto hasta el momento se podría pensar que una candidata para ser la función ϕ es la función sigmoid, aunque esto es factible, esta presenta el problema de desvanecimiento del gradiente. Para ejemplificar este problema, se utilizan dos funciones $g_1(x) = w_1x + 1.0$ y $g_1(x) = w_2x + 1.0$; haciéndose la composición de estas dos funciones $g = g_2 \circ g_1$.

Las siguientes instrucciones implementan las funciones anteriores utilizando la librería JAX.

```
@jax.jit
def g_1(params, x):
    return jax.nn.sigmoid(params['w1'] * x + 1.0)

@jax.jit
def g_2(params, x):
    return jax.nn.sigmoid(params['w2'] * x + 1.0)

@jax.jit
def g(params, x):
    return g_2(params, g_1(params, x))
```

Utilizando unos parámetros aleatorios generados con el siguiente código

```
key = jax.random.PRNGKey(0)
key, subkey1, subkey2 = jax.random.split(key, num=3)
params = dict(w1=jax.random.normal(subkey1),
              w2=jax.random.normal(subkey2))
```

se tiene que la derivada de g_1 con respecto a w_1 (i.e., `jax.grad(g_1)(params, 1.0)`) y g_2 con respecto a w_2 (i.e., `jax.grad(g_2)(params, 1.0)`) es 0.1418 y 0.1171, respectivamente. El problema viene cuando se calcula $\frac{\partial g}{\partial w_1}$ en este caso se obsevar que el gradiente es pequeño comparado con el gradiente obtenido en $\frac{dg}{dw_1}$, i.e., `jax.grad(g)(params, 1.0)`, donde se observa que el gradiente para w_1 corresponde a 0.0157, el gradiente de w_2 sigue en la misma mágnitud teniendo un valor de 0.1077.

El problema de desvanecimiento de gradiente, hace que el gradiente disminuya de manera exponencial, entonces los pesos asociados a las capas alejadas de la capa de salida reciben un gradiente equivalente a cero y no se cuenta con información para actualizar sus pesos. Por este motivo es recomendable utilizar funciones de activación que no presenten esta característica, una muy utilizada es $\mathsf{ReLU}(x) = \max(0, x)$.

11.6 Ejemplo: Dígitos

Para ejemplificar el uso de una red neuronal en un poblema de clasificación se utilizarán los datos de Dígitos, los cuales se pueden obtener con las siguientes instrucciones.

```
X, y = load_digits(return_X_y=True)
T, G, y_t, y_g = train_test_split(X, y, test_size=0.2)
```

Un procedimiento necesario en redes neuronales es que los datos estén normalizados, tradicionalmente esto se realiza haciendo que los datos tengan media cero y desviación estandar uno. En las siguientes lineas se normalizan los datos usando la clase `StandardScaler` se puede observar que los parámetros para la normalización son encontrados en el conjunto de entrenamiento (`T`) y aplicados tanto al conjunto de entrenamiento como el conjunto de prueba `G`.

```
normalize = StandardScaler().fit(T)
T = normalize.transform(T)
G = normalize.transform(G)
```

Como se realizó previamente es necesario convertir las clases de salida para que cada ejemplo sea un vector unitario donde el índice con el valor 1 representa la clase, esto se realiza con las siguientes instrucciones.

```
n_labels = np.unique(y).shape[0]
yt_oh = jax.nn.one_hot(y_t, n_labels)
```

Es momento de decidir la estructura de la red neuronal, las únicas dos restricciones es que la primera capa tiene que tener la dimensión del vector de entrada, en esta caso corresponde a 64 (`T.shape[1]`) y la última capa tiene que tener de salida el número de clases, en este caso 10 (`n_labels`), el resto de las capas ocultas pueden tener cualquier valor solamente es necesario que la dimensiones sean coherentes con la operación que se va a realizar.

La red que se va a implementar es la siguiente, como super-índice se encuentran las dimensiones para que sea más fácil seguir la estructura de la red.

$$\begin{aligned}
\hat{\mathbf{y}}_1^{32} &= \phi(W^{32\times 64}\mathbf{x}^{64} + \mathbf{w}_{1_0}^{32}) \\
\hat{\mathbf{y}}_2^{16} &= \phi(W^{16\times 32}\hat{\mathbf{y}}_1^{32} + \mathbf{w}_{2_0}^{16}) \\
\hat{\mathbf{y}}^{10} &= W^{10\times 16}\hat{\mathbf{y}}_2^{16} + \mathbf{w}_{3_0}^{10}
\end{aligned}$$

Considerando que las entradas se encuentran en una matrix $X^{N\times 64}$, entonces se puede definir esta estructura en términos de múltiplicación de matrices, lo cual queda como

$$\begin{aligned}
\hat{Y}_1^{N\times 32} &= \phi(X^{N\times 64}W^{64\times 32} + \mathbf{w}_{1_0}^{32}) \\
\hat{Y}_2^{N\times 16} &= \phi(\hat{Y}_1^{N\times 32}W^{32\times 16} + \mathbf{w}_{2_0}^{16}), \\
\hat{Y}^{N\times 10} &= \hat{Y}_2^{N\times 16}W^{16\times 10} + \mathbf{w}_{3_0}^{10}
\end{aligned}$$

donde la suma con el término $\mathbf{w}_0$ se realiza en la dimensión que corresponde y se replica tantas veces para cumplir con la otra dimensión. Esta configuración se puede expresar en un lista como la que se muestra a continuación.

```
d = [64, 32, 16, 10]
```

Utilizando esta notación los parámetros iniciales de la red se pueden generar con la siguiente función, se puede observar como el ciclo está iterando por los elementos de `d` creando pares, para generar las dimensiones adecuadas para las matrices W.

```
def parametros_iniciales(d, key=0):
    key = jax.random.PRNGKey(key)
    params = []
    for init, end in zip(d, d[1:]):
```

```
        key, subkey1, subkey2 = jax.random.split(key, num=3)
        _ = dict(w=jax.random.normal(subkey1, (init, end)) * jnp.sqrt(2 / (init * end)),
                 w0=jax.random.normal(subkey2, (end, )) * jnp.sqrt(2 / end))
        params.append(_)
    return params
```

Habiendo generado los parámetros iniciales de la red, es momento para implmentar la red, la siguiente función implementa la red, se puede observar como es realizan las operaciones matriciales tal y como se mostraron en las ecuaciones anteriores. La función de activiación ϕ seleccionada fue ReLU que está implementada en la función `jax.nn.relu`.

```
@jax.jit
def ann(params, D):
    y1 = jax.nn.relu(D @ params[0]['w'] + params[0]['w0'])
    y2 = jax.nn.relu(y1 @ params[1]['w'] + params[1]['w0'])
    return y2 @ params[2]['w'] + params[2]['w0']
```

Como medida de error se usa la Entropía Cruzada (Sección 4.2.7) tal y como se implementa a continuación. El caso $0 \log 0$ corresponde a un valor no definido lo cual genera que el valor de la función tampoco este definido, para proteger la función en ese caso se usa `jnp.nansum` que trata los valores no definidos como ceros.

```
@jax.jit
def media_entropia_cruzada(params, D, y_oh):
    hy = jax.nn.softmax(ann(params, D), axis=1)
    return - jnp.nansum(y_oh * jnp.log(hy), axis=1).mean()
```

En esta ocasión se utiliza el optimizador Adam, tal y como se muestra en la siguiente función. La única diferencia con respecto al visto previamente es la función `update_finite` que actualiza los parámetros siempre y cuando el nuevo valor sea un valor numérico, de los contrario se queda con el valor anterior.

```
def adam(optimizer, params, epocas=500):
    @jax.jit
    def update_finite(a, b):
        m = jnp.isfinite(b)
        return jnp.where(m, b, a)

    @jax.jit
    def update(params, opt_state, X, y_oh):
        loss_value, grads = error_grad(params, X, y_oh)
```

```
        updates, opt_state = optimizer.update(grads, opt_state, params)
        params = optax.apply_updates(params, updates)
        return params, opt_state, loss_value

    opt_state = optimizer.init(params)
    error_grad  = jax.value_and_grad(media_entropia_cruzada)
    error = []
    for i in range(epocas):
        p, opt_state, loss_value = update(params, opt_state, T, yt_oh)
        params = jax.tree_map(update_finite, params, p)
        error.append(loss_value)
    return params, error
```

Finalmente, la red creada se entrana utilizando las siguientes instrucciones, donde la primera linea genera los parámetros iniciales, después se inicializa el optimizador y en la línea final se llama al optimizador con los parámetros y número de épocas.

```
params = parametros_iniciales(d)
optimizer = optax.adam(learning_rate=1e-2)
p, error = adam(optimizer, params, epocas=500)
```

El resultado de esta red, es que el error en el conjunto de prueba es 0.0167 calculado con la siguiente instrucción `(ann(p, G).argmax(axis=1) != y_g).mean()`.

12 Ensambles

El **objetivo** de la unidad es conocer y aplicar diferentes técnicas para realizar un ensamble de clasificadores o regresores.

12.1 Paquetes usados

```
from scipy.stats import binom
from sklearn.datasets import load_diabetes, load_digits
from sklearn.model_selection import train_test_split
from sklearn.svm import LinearSVC, LinearSVR, SVR
from sklearn.tree import DecisionTreeClassifier, DecisionTreeRegressor
from sklearn.ensemble import BaggingRegressor
from sklearn.neighbors import KNeighborsRegressor
from sklearn.metrics import recall_score, mean_absolute_percentage_error
from collections import Counter
from matplotlib import pylab as plt
import numpy as np
import pandas as pd
import seaborn as sns
```

12.2 Introducción

Como se ha visto hasta el momento, cada algoritmo de clasificación y regresión tiene un sesgo, este puede provenir de los supuestos que se asumieron cuando se entrenó o diseño; por ejemplo, asumir que los datos provienen de una distribución gausiana multivariada o que se pueden separar los ejemplos mediante un hiperplano, entre otros. Dado un problema se desea seleccionar aquel algoritmo que tiene el mejor rendimiento, visto de otra manera, se selecciona el algoritmo cuyo sesgo este mejor alineado al problema. Una manera complementaria sería utilizar varios algoritmos y tratar de predecir basados en las predicciones individuales de cada algoritmo. En esta unidad se explicarán diferentes metodologías que permiten combinar predicciones de algoritmos de clasificación y regresión.

12.3 Fundamentos

La descripción de ensambles se empieza observando el siguiente comportamiento. Suponiendo que se cuenta con M algoritmos de clasificación binaria cada uno tiene una exactitud de $p = 0.51$ y estos son completamente independientes. El proceso de clasificar un elemento corresponde a preguntar la clase a los M clasificadores y la clase que se recibe mayor votos es la clase seleccionada, esta votación se comporta como una variable aleatoria que tiene una distribución Binomial. Suponiendo con la clase del elemento es 1, en esta condición la función cumulativa de distribución (cdf) con parámetros $k = \lfloor \frac{M}{2} \rfloor$, $n = M$ y $p = 0.51$ indica seleccionar la clase 0 y $1 - \text{cdf}$ corresponde a la probabilidad de seleccionar la clase 1.

La Figura 12.1 muestra como cambia la exactitud, cuando el número de clasificadores se incrementa, cada uno de esos clasificadores son independientes y tiene una exactitud de $p = 0.51$, se puede observar que cuando $M = 501$ el rendimiento es 0.673 y con $9,999$ clasificadores se tiene un valor de 0.977.

En el caso de regresión, en particular cuando se usa como función de error el cuadrado del error, i.e., $(\hat{y}-y)^2$ se tiene el intercambio entre varianza y sesgo, el cual se deriva de la siguiente manera.

$$
\begin{aligned}
\mathbb{E}[(\hat{y} - y)^2] = & \\
& = \mathbb{E}[(\hat{y} - \mathbb{E}[\hat{y}] + \mathbb{E}[\hat{y}] - y)^2] \\
& = \underbrace{\mathbb{E}[(\hat{y} - \mathbb{E}[\hat{y}])^2]}_{\mathbb{V}(\hat{y})} + \mathbb{E}[(\mathbb{E}[\hat{y}] - y)^2] \\
& + 2\mathbb{E}[(\hat{y} - \mathbb{E}[\hat{y}])(\mathbb{E}[\hat{y}] - y)] \\
& = \mathbb{V}(\hat{y}) + (\underbrace{\mathbb{E}[\hat{y}] - y}_{\text{sesgo}})^2 + 2\underbrace{\mathbb{E}[(\hat{y} - \mathbb{E}[\hat{y}])]}_{\mathbb{E}[\hat{y}]-\mathbb{E}[\hat{y}]=0}(\mathbb{E}[\hat{y}] - y) \\
& = \mathbb{V}(\hat{y}) + (\mathbb{E}[\hat{y}] - y)^2
\end{aligned}
$$

Se observa que el cuadrado del error está definido por la varianza de $\hat{y}$ (i.e., $\mathbb{V}(\hat{y})$), la cual es independiente de la salida y y el sesgo al cuadrado del algoritmo (i.e., $(\mathbb{E}[\hat{y}] - y)^2$).

En el contexto de ensamble, asumiendo que se tienen M regresores independientes donde la predicción está dada por $\bar{y} = \frac{1}{M}\sum_{i=1}^{M} \hat{y}^i$, se tiene que el sesgo de cada predictor individual es igual al sesgo de su promedio (i.e., $(\mathbb{E}[\bar{y}] - y) = (\mathbb{E}[\hat{y}^i] - y)$) como se puede observar a continuación.

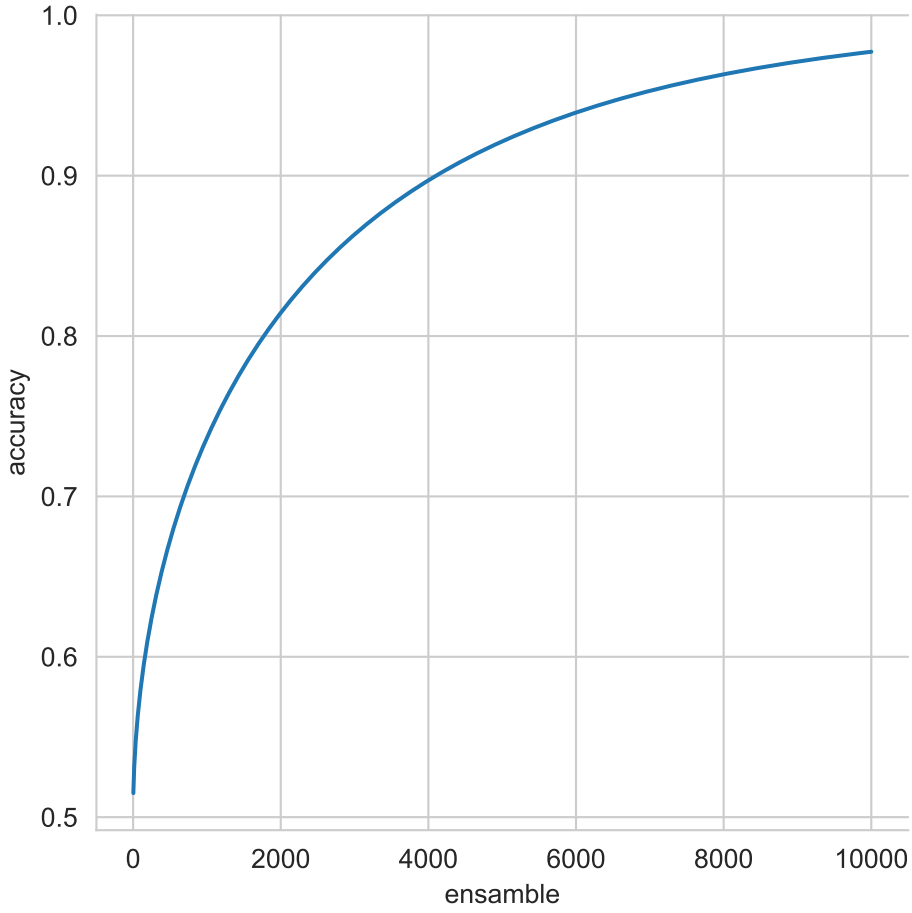


Figura 12.1: Rendimiento cuando el número de clasificadores se incrementa

$$\mathbb{E}[\bar{y}] = \mathbb{E}[\frac{1}{M}\sum_{i=1}^{M}\hat{y}^i]$$
$$= \frac{1}{M}\sum_{i=1}^{M}\underbrace{\mathbb{E}[\hat{y}^i]}_{\mathbb{E}[\hat{y}]} = \frac{1}{M}M\mathbb{E}[\hat{y}] = \mathbb{E}[\hat{y}]$$

Por otro lado la varianza del promedio (i.e., $\mathbb{V}(\bar{y})$) está dada por $\mathbb{V}(\bar{y}) = \frac{1}{M}\mathbb{V}(\hat{y})$, que se deriva siguiendo los pasos del error estándar de la media (Sección A.1.1).

Esto quiere decir que si se tienen M regresores independientes, entonces el error cuadrado de su promedio es menor que el error de cada regresor individual, esto es porque su la varianza se reduce tal y como se mostró.

Tanto en el caso de clasificación como en el caso del error cuadrado, es poco probable contar con clasificadores y regresores que sean completamente independientes, entonces sus predicciones van a estar relacionadas en algún grado y no se podrá llegar a las reducciones obtenidas en el procedimiento presentado.

12.4 Bagging

Siguiendo con la idea de combinar M instancias independientes de un tipo de algoritmo, en esta sección se presenta el algoritmo Bagging (Bootstrap Aggregation) el cual como su nombre lo indica se basa la técnica de Bootstrap (Sección A.2) para generar M instancias del algoritmo y la combinación es mediante votación o el promedio en caso de regresión o que se cuente con la probabilidad de cada clase.

12.4.1 Ejemplo: Dígitos

Para ejemplificar el uso del algoritmo de Bagging se utilizará el conjunto de datos de Dígitos. Estos datos se pueden obtener y generar el conjunto de entrenamiento ($\mathcal{T}$) y prueba ($\mathcal{G}$) con las siguientes instrucciones.

```
X, y = load_digits(return_X_y=True)
T, G, y_t, y_g = train_test_split(X, y,
                                  test_size=0.2,
                                  random_state=0)
```

Los algoritmos que se utilizarán de base son Máquinas de Soporte Vectorial Lineal (Sección 9.4) y Árboles de Decisión (Capítulo 8). Lo primero que se realizará es entrenar una instancia de estos algoritmos para poder comparar su rendimiento en el conjunto de prueba contra Bagging.

La siguientes instrucciones entrenan una máquina de soporte vectorial, calculando en la segunda línea el macro-recall (Sección 4.2.6). El rendimiento se presenta en una tabla para facilitar la comparación.

```
svc = LinearSVC(dual=False).fit(T, y_t)
svc_recall = recall_score(y_g, svc.predict(G),
                          average="macro")
```

Complementando las instrucciones anteriores, en el siguiente código se entrena un Árbol de Decisión.

```
tree = DecisionTreeClassifier(criterion='entropy',
                              min_samples_split=9).fit(T, y_t)
tree_recall = recall_score(y_g, tree.predict(G),
                           average="macro")
```

El algoritmo de Bootstrap inicia generando las muestras tal y como se realizó en el ejemplo del error estándar de la media (Sección A.2.1); el siguiente código genera las muestras, en particular el ensamble sería de $M = 11$ elementos.

```
B = np.random.randint(T.shape[0],
                      size=(11, T.shape[0]))
```

Empezando con Bagging usando como clasificador base la Máquina de Soporte Vectorial Lineal. La primera línea de las siguientes instrucciones, entra los máquinas de soporte, después se realizan las predicciones. En la tercera línea se calcula la clase que tuvo la mayor cantidad de votos y finalmente se calcula el error en términos de macro-recall.

```
svc_ins = [LinearSVC(dual=False).fit(T[b], y_t[b])
           for b in B]
hys = np.array([m.predict(G) for m in svc_ins])
hy = np.array([Counter(x).most_common(n=1)[0][0]
               for x in hys.T])
bsvc_recall = recall_score(y_g, hy, average="macro")
```

El siguiente algoritmo son los Árboles de Decisión; la única diferencia con respecto a las instrucciones anteriores es la primera línea donde se entrenan los árboles.

```
tree_ins = [DecisionTreeClassifier(criterion='entropy',
                                   min_samples_split=9).fit(T[b], y_t[b])
            for b in B]
hys = np.array([m.predict(G) for m in tree_ins])
hy = np.array([Counter(x).most_common(n=1)[0][0]
               for x in hys.T])
btree_recall = recall_score(y_g, hy,
                            average="macro")
```

Como se mencionó, la predicción final se puede realizar de dos manera en clasificación una es usando votación, como se vió en los códigos anteriores y la segunda es utilizando el promedio de las probabilidades. En el caso de las Máquinas de Soporte Vectorial, estas no calculas las probabilidad de cada clase, pero se cuenta con el valor de la función de decisión, en el siguiente código se usa está información, la segunda y tercera línea normaliza los valores para que ningún valor sea mayor que 1 y menor que -1 y finalmente se calcula la suma para después seleccionar la clase que corresponde al argumento máximo.

```
hys = np.array([m.decision_function(G) for m in svc_ins])
hys = np.where(hys > 1, 1, hys)
hys = np.where(hys < -1, -1, hys)
hys = hys.sum(axis=0)
csvc_recall = recall_score(y_g, hys.argmax(axis=1),
                           average="macro")
```

El procedimiento anterior se puede adaptar a los Árboles de Decisión utilizando el siguiente código.

```
hys = np.array([m.predict_proba(G)
                for m in tree_ins])
ctree_recall = recall_score(y_g,
                            hys.sum(axis=0).argmax(axis=1),
                            average="macro")
```

Finalmente, la Tabla 12.1 muestra el rendimiento de las diferentes combinaciones, se puede observar el valor tanto de las Máquinas de Soporte Vectorial (M.S.V) Lineal y de los Árboles de decisión cuando se utilizaron fuera del ensamble; en el segundo renglón se muestra el rendimiento cuando la predicción del ensamble se hizo mediante votación y el último renglón presenta el rendimiento cuando se hace la suma.

Comparando los diferentes rendimientos, se puede observar que no existe mucha diferencia en rendimiento en las M.S.V Lineal y que la mayor mejora se presentó en los Árboles de Decisión.

Este comportamiento es esperado dado que para que Bagging funciones adecuadamente requiere algoritmos inestables, es decir, algoritmos cuyo comportamiento cambia considerablemente con un cambio pequeño en el conjunto de entrenamiento, este es el caso de los Árboles. Por otro lado las M.S.V son algoritmos estables y un cambio pequeño en su conjunto de entrenamiento no tendrá una repercusión considerable en el comportamiento del algoritmo.

Tabla 12.1: Rendimiento (macro-recall) de bagging y estimadores base.

	M.S.V. Lineal	Árboles de Decisión
Único	0.9425	0.8288
Votación ($M = 11$)	0.9467	0.9366
Suma ($M = 11$)	0.9494	0.9368

12.4.2 Ejemplo: Diabetes

Ahora toca el turno de atacar un problema de regresión mediante Bagging, el problema que se utilizará es el de Diabetes. Las instrucciones para obtener el problema y generar los conjuntos de entrenamiento ($\mathcal{T}$) y prueba ($\mathcal{G}$) se muestra a continuación.

```
X, y = load_diabetes(return_X_y=True)
T, G, y_t, y_g = train_test_split(X, y,
                                  random_state=0,
                                  test_size=0.2)
```

Tomando el caso de Dígitos como base, el primer algoritmo a entrenar es la M.S.V. Lineal y se usa como mediada de rendimiento el porcentaje del error absoluto (Ecuación 4.6); tal y como se muestran en las siguientes instrucciones.

```
svr = LinearSVR(dual='auto').fit(T, y_t)
svr_mape = mean_absolute_percentage_error(y_g,
                                          svr.predict(G))
```

El árbol de decisión y su rendimiento se implementa con el siguiente código.

```
tree = DecisionTreeRegressor(min_samples_split=9).fit(T,
                                                      y_t)
tree_mape = mean_absolute_percentage_error(y_g,
                                           tree.predict(G))
```

Al igual que en el caso de clasificación, la siguiente instrucción genera los índices para generar las muestras. Se hace un ensamble de $M = 11$ elementos.

```
B = np.random.randint(T.shape[0], size=(11, T.shape[0]))
```

En el caso de regresión, la predicción final corresponde al promedio de las predicciones individuales, la primera línea de las siguientes instrucciones se entrena las M.S.V Lineal, en la segunda instrucción se hacen las predicciones y se en la tercera se realiza el promedio y se mide el rendimiento.

```
svr_ins = [LinearSVR(dual='auto').fit(T[b], y_t[b])
           for b in B]
hys = np.array([m.predict(G) for m in svr_ins])
bsvr_mape = mean_absolute_percentage_error(y_g,
                                           hys.mean(axis=0))
```

De manera equivalente se entrenan los Árboles de Decisión, como se muestra a continuación.

```
tree_ins = [DecisionTreeRegressor(min_samples_split=9).fit(T[b], y_t[b])
            for b in B]
hys = np.array([m.predict(G) for m in tree_ins])
btree_mape = mean_absolute_percentage_error(y_g,
                                            hys.mean(axis=0))
```

La Tabla 12.2 muestra el rendimiento de los algoritmos de regresión utilizados, al igual que en el caso de clasificación, la M.S.V. no se ve beneficiada con el uso de Bagging. Por otro lado los Árboles de Decisión tienen un incremento en rendimiento considerable al usar Bagging.

Tabla 12.2: Rendimiento (MAPE) de bagging y estimadores base.

	M.S.V. Lineal	Árboles de Decisión
Único	0.4131	0.5381
Promedio ($M = 11$)	0.4131	0.4263

Hasta este momento los ensambles han sido de $M = 11$ elementos, queda la duda como varía el rendimiento con respecto al tamaño del ensamble. La Figura 12.2 muestra el rendimiento de Bagging utilizando Árboles de Decisión, cuando el ensamble cambia $M = 2, \ldots, 500$. Se observa que alrededor que hay un decremento importante cuando el ensamble es pequeño, después el error se incrementa y vuelve a bajar alrededor de $M = 100$. Finalmente se ve que el rendimiento es estable cuando $M > 200$.

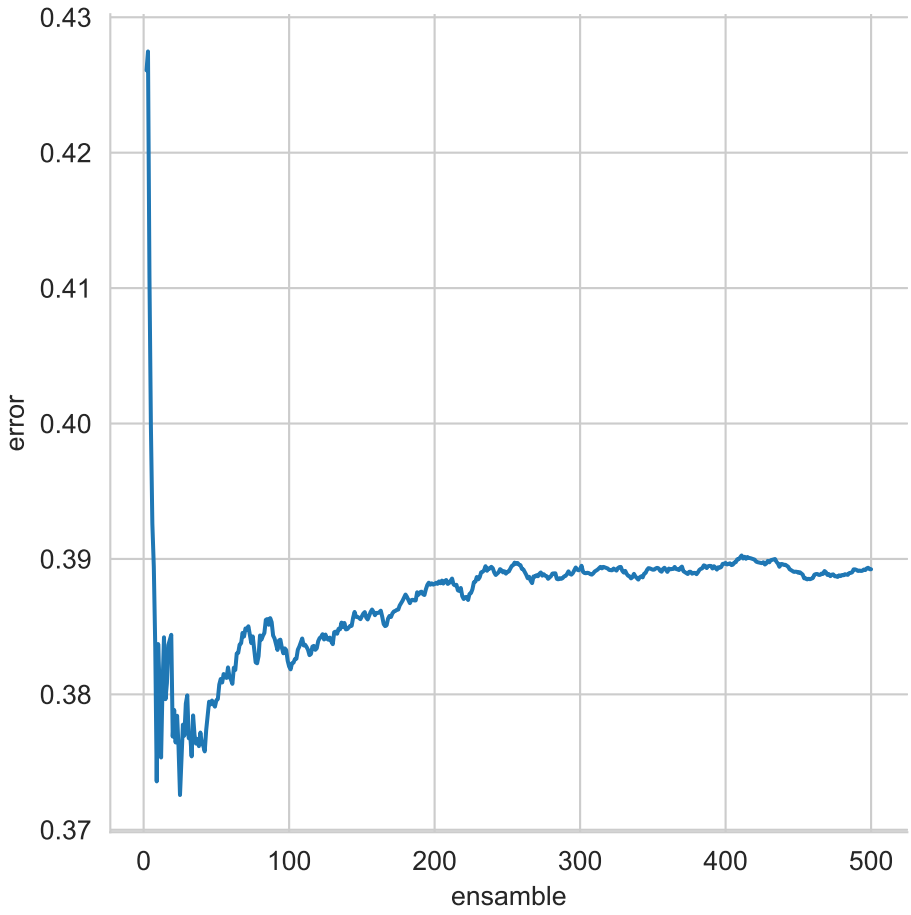


Figura 12.2: Variación del rendimiento con respecto al tamaño del ensamble (M).

12.5 Stack Generalization

Continuando con la descripción de ensambles, se puede observar que Bagging en el caso de la media se puede representar como $\sum_i^M \frac{1}{M}\hat{y}^i$, donde el factor $\frac{1}{M}$ se puede observar como un parámetro a identificar. Entonces la idea siguiente sería como se podría estimar parámetros para cada uno de los estimadores bases, e.g., $\sum_i^M w_i\hat{y}^i$ donde w_i para bagging corresponde a $\frac{1}{M}$. Se podría ir más allá y pensar que las predicciones $\mathbf{y} = (\hat{y}^1, \ldots, \hat{y}^M)$ podrían ser la entrada a otro estimador.

Esa es la idea detrás de Stack Generalization, la idea es utilizar las predicciones de los estimadores bases como las entradas de otro estimador. Para poder llevar este proceso es necesario contar con un conjunto independiente del conjunto de entrenamiento para encontrar los parámetros del estimador que combina las predicciones.

12.5.1 Ejemplo: Diabetes

Para ejemplificar el uso de Stack Generalization, se usa el conjunto de datos de Diabetes. Como se acaba de describir es necesario contar con un conjunto independiente para estimar los parámetros del estimador del stack. Entonces el primer paso es dividir el conjunto de entrenamiento en un conjunto de entrenamiento y validación ($\mathcal{V}$,) tal y como se muestra en la siguiente instrucción.

```
T1, V, y_t1, y_v = train_test_split(T, y_t,
                                    test_size=0.3)
```

Para este ejemplo se usará como regresores bases el algoritmo de Vecinos Cercanos (Sección 7.7) con diferentes parámetros (primera línea), después se usan los modelos para predecir el conjunto de validación ($\mathcal{V}$) y prueba ($\mathcal{G}$), esto se observa en la segunda y tercera línea del siguiente código.

```
models = [KNeighborsRegressor(n_neighbors=n).fit(T1, y_t1)
          for n in [7, 9]]
V_stack = np.array([m.predict(V) for m in models]).T
G_stack = np.array([m.predict(G) for m in models]).T
```

El porcentaje del error absoluto (Ecuación 4.6) de los estimadores bases en el conjunto de prueba se puede calcular con el siguiente código

```
mape_test = []
for hy in G_stack.T:
  mape = mean_absolute_percentage_error(y_g, hy)
  mape_test.append(mape)
```

teniendo los siguientes valores 0.3556 y 0.3545, respectivamente.

Finalmente, es momento de entrenar el regresor que combinará las salidas de los estimadores bases, i.e., Vecinos Cercanos. Se decidió utilizar una Máquina de Soporte Vectorial (Sección 9.4) con kernel polinomial de grado 2. Los parámetros de la máquina se estiman en la primera línea, y después se predicen los datos del conjunto de prueba.

```
stacking = SVR(kernel='poly', degree=2).fit(V_stack, y_v)
hy = stacking.predict(np.vstack(G_stack))
```

El error (Ecuación 4.6) obtenido por este procedimiento es de 0.3669; cabe mencionar que no en todos los casos el procedimiento de stacking consigue un mejor rendimiento que los estimadores bases, por ejemplo, en las siguientes instrucciones se entrena un Bagging con Árboles de Decisión para ser utilizado en lugar de la Máquina de Soporte Vectorial.

```
st_trees = BaggingRegressor(estimator=DecisionTreeRegressor(min_samples_split=9),
                            n_estimators=200).fit(V_stack, y_v)
hy = st_trees.predict(np.vstack(G_stack))
mape = mean_absolute_percentage_error(y_g, hy)
```

El rendimiento de este cambio es 0.4547 lo cual es mayor que el error obtenido por los estimadores base.

13 Comparación de Algoritmos

El **objetivo** de la unidad es conocer y aplicar diferentes procedimientos estadísticos para comparar y analizar el rendimiento de algoritmos.

13.1 Paquetes usados

```
from IngeoML import CI, SE
from scipy.stats import norm, wilcoxon
from sklearn.datasets import load_iris, load_breast_cancer
from sklearn.naive_bayes import GaussianNB
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split, StratifiedKFold
from sklearn.metrics import recall_score
import numpy as np
import seaborn as sns
```

13.2 Introducción

Hasta el momento se han descrito diferentes algoritmos de clasificación y regresión; se han presentado diferentes medidas para conocer su rendimiento, pero se ha dejado de lado el conocer la distribución de estas medidas para poder tener mayor información sobre el rendimiento del algoritmo y también poder comparar y seleccionar el algoritmo que tenga las mejores prestaciones ya sea en rendimiento o en complejidad.

13.3 Intervalos de confianza

El análisis del rendimiento se inicia partiendo de que el rendimiento se puede estimar a partir del conjunto de prueba, $\mathcal{G}$; el valor obtenido estima el rendimiento real, θ, el cual se considera una constante. Una manera de conocer el rango de valores donde se puede encontrar θ es generando su intervalo de confianza. El intervalo de confianza de θ está dado por $C = (a(\mathcal{G}), b(\mathcal{G}))$,

de tal manera que $P_\theta(\theta \in C) \geq 1 - \alpha$. Es importante mencionar que el intervalo no mide la probabilidad de θ dado que θ es una constante, en su lugar mide de que el valor estimado esté dentro de esos límites con esa probabilidad. Por otro lado se utiliza la notación $a(\mathcal{G})$ y $b(\mathcal{G})$ para hacer explicito que en este caso los límites del intervalo son obtenidos utilizando el conjunto de prueba. Una manera de entender el intervalo de confianza de cualquier parámetro es suponer que si el parámetro se estima 100 veces con el mismo procedimiento, en diferentes muestras, un intervalo del 95% de confianza dice que 95 de las veces la estimación del parámetro estará en el intervalo calculado.

13.3.1 Método: Distribución Normal

Existen diferentes procedimientos para generar intervalos de confianza, uno de ellos es asumir que la estimación de θ, i.e., $\hat{\theta}$ se distribuye como una normal, i.e., $\hat{\theta} \sim \mathcal{N}(\mu, \sigma^2)$, donde $\sigma = \mathsf{se} = \sqrt{\mathbb{V}(\hat{\theta})}$ corresponde al error estándar (Sección A.1) de la estimación $\hat{\theta}$. En estas condiciones el intervalo está dado por:

$$C = (\hat{\theta} - z_{\frac{\alpha}{2}}\mathsf{se}, \hat{\theta} + z_{\frac{\alpha}{2}}\mathsf{se}),$$

donde $z_{\frac{\alpha}{2}} = \Phi^{-1}(1 - \frac{\alpha}{2})$ y Φ es la función de distribución acumulada de una normal.

13.3.2 Ejemplo: Exactitud

Recordado que dado una entrada el clasificador puede acertar la clase a la que pertenece esa entrada, entonces el resultado se puede representar como 1 si la respuesta es correcta y 0 de lo contrario. En este caso la respuesta es una variable aleatoria con una distribución de Bernoulli. Recordando que la distribución Bernoulli está definida por un parámetro p, estimado como $\hat{p} = \frac{1}{N}\sum_{i=1}^{N} \mathcal{X}_i$ donde $\mathcal{X}_i$ corresponde al resultado del algoritmo en el i-ésimo ejemplo. La varianza de una distribución Bernoulli es $p(1-p)$ por lo que el error estándar es: $se = \sqrt{\frac{p(1-p)}{N}}$ dando como resultado el siguiente intervalo:

$$C = (\hat{p}_N - z_{\frac{\alpha}{2}}\sqrt{\frac{p(1-p)}{N}}, \hat{p}_N + z_{\frac{\alpha}{2}}\sqrt{\frac{p(1-p)}{N}}).$$

Suponiendo $N = 100$ y $p = 0.85$ el siguiente código calcula el intervalo usando $\alpha = 0.05$

```
alpha = 0.05
z = norm().ppf(1 - alpha / 2)
p = 0.85
N = 100
```

```
Cn = (p - z * np.sqrt(p * (1 - p) / N),
      p + z * np.sqrt(p * (1 - p) / N))
```

dando como resultado el siguiente intervalo, $C = (0.78, 0.92)$.

En el caso anterior se supuso que se contaba con los resultados de un algoritmo de clasificación, con el objetivo de completar este ejemplo a continuación se presenta el análisis con un Naive Bayes en el problema del Iris.

Lo primero que se realiza es cargar los datos y dividir en el conjunto de entrenamiento ($\mathcal{T}$) y prueba ($\mathcal{G}$) como se muestra a continuación.

```
X, y = load_iris(return_X_y=True)
T, G, y_t, y_g = train_test_split(X, y,
                                  random_state=1,
                                  test_size=0.3)
```

El siguiente paso es entrenar el algoritmo y realizar las predicciones en el conjunto de prueba ($\mathcal{G}$) tal y como se muestra en las siguientes instrucciones.

```
model = GaussianNB().fit(T, y_t)
hy = model.predict(G)
```

Con las predicciones se estima la exactitud y se siguen los pasos para calcular el intervalo de confianza como se ilustra en el siguiente código.

```
_ = np.where(y_g == hy, 1, 0)
p = _.mean()
N = _.shape[0]
C = (p - z * np.sqrt(p * (1 - p) / N), p + z * np.sqrt(p * (1 - p) / N))
```

El intervalo de confianza obtenido es $C = (0.86, 1.01)$. se puede observar que el límite superior es mayor que 1 lo cual no es posible dado que el máximo valor del accuracy es 1, esto es resultado de generar el intervalo de confianza asumiendo una distribución normal.

Cuando se cuenta con conjuntos de datos pequeños y además no se ha definido un conjunto de prueba, se puede obtener las predicciones del algoritmo de clasificación mediante el uso de validación cruzada usando K-fold. En el siguiente código se muestra su uso, el cambio solamente es en el procedimiento para obtener las predicciones.

```
kf = StratifiedKFold(n_splits=10,
                     random_state=0,
                     shuffle=True)
hy = np.empty_like(y)
for tr, ts in kf.split(X, y):
    model = GaussianNB().fit(X[tr], y[tr])
    hy[ts] = model.predict(X[ts])
```

El resto del código es equivalente al usado previamente obteniendo el siguiente intervalo de confianza $C = (0.92, 0.99)$.

13.3.3 Método: Bootstrap del error estándar

Existen ocasiones donde no es sencillo identificar el error estándar (se) y por lo mismo no se puede calcular el intervalo de confianza. En estos casos se emplea la técnica de Bootstrap (Sección A.2) para estimar $\mathbb{V}(\hat{\theta})$. Un ejemplo donde no es sencillo encontrar analíticamente el error estándar es en el *recall* (Sección 4.2.3).

Es más sencillo entender este método mediante un ejemplo. Usando el ejercicio de $N = 100$ y $p = 0.85$ y $\alpha = 0.05$ descrito previamente, el siguiente código primero construye las variables aleatorias de tal manera que den $p = 0.85$

```
alpha = 0.05
N = 100
z = norm().ppf(1 - alpha / 2)
X = np.zeros(N)
X[:85] = 1
```

`X` es una arreglo que podrían provenir de la evaluación de un clasificador usando alguna medida de similitud entre predicción y valor medido. El siguiente paso es generar seleccionar con remplazo y obtener $\hat{\theta}$ para cada muestra, en este caso $\hat{\theta}$ corresponde a la media. El resultado se guarda en una lista B y se repite el experimento 500 veces.

```
S = np.random.randint(X.shape[0],
                      size=(500, X.shape[0]))
B = [X[s].mean() for s in S]
```

El error estándar es y el intervalo de confianza se calcula con las siguientes instrucciones

```
se = np.sqrt(np.var(B))
C = (p - z * se, p + z * se)
```

el intervalo de confianza corresponde a $C = (0.88, 1.03)$.

Continuando con el mismo ejemplo pero ahora analizando Naive Bayes en el problema del Iris. El primer paso es obtener evaluar las predicciones que se puede observar en el siguiente código (previamente descrito.)

```
X, y = load_iris(return_X_y=True)
kf = StratifiedKFold(n_splits=10,
                     random_state=0,
                     shuffle=True)

hy = np.empty_like(y)
for tr, ts in kf.split(X, y):
    model = GaussianNB().fit(X[tr], y[tr])
    hy[ts] = model.predict(X[ts])
X = np.where(y == hy, 1, 0)
```

Realizando la selección con remplazo se obtiene el intervalo con las siguientes instrucciones

```
B = [X[s].mean() for s in S]
se = np.sqrt(np.var(B))
C = (p - z * se, p + z * se)
```

teniendo un valor de $C = (0.92, 0.99)$.

13.3.4 Método: Percentil

Existe otra manera de calcular los intervalos de confianza y es mediante el uso del percentil, utilizando directamente las estimaciones realizadas a $\hat{\theta}$ en la selección. El siguiente código muestra este método usando el ejemplo anterior,

```
alpha = 0.05 / 2
C = (np.percentile(B, alpha * 100),
     np.percentile(B, (1 - alpha) * 100))
```

obteniendo un intervalo de $C = (0.92, 0.99)$.

13.3.5 Ejemplo: macro-recall

Hasta el momento se ha usado una medida de rendimiento para la cual se puede conocer su varianza de manera analítica. Existen problemas donde esta medida no es recomendada, en el siguiente ejemplo utilizaremos macro-recall para medir el rendimiento de Naive Bayes en el problema del Iris. El primer paso es realizar las predicciones del algoritmo usando validación cruzada y hacer la muestra con reemplazo B.

```
alpha = 0.05
z = norm().ppf(1 - alpha / 2)

X, y = load_iris(return_X_y=True)
kf = StratifiedKFold(n_splits=10,
                     random_state=0,
                     shuffle=True)

hy = np.empty_like(y)
for tr, ts in kf.split(X, y):
    model = GaussianNB().fit(X[tr], y[tr])
    hy[ts] = model.predict(X[ts])

S = np.random.randint(hy.shape[0],
                      size=(500, hy.shape[0]))
B = [recall_score(y[s], hy[s], average="macro")
     for s in S]
```

El siguiente paso es calcular el intervalo asumiendo que este se comporta como una normal tal y como se muestra en las siguientes instrucciones;

```
p = np.mean(B)
se = np.sqrt(np.var(B))
C = (p - z * se, p + z * se)
```

obteniendo un intervalo de $C = (0.92, 0.98)$ Completando el ejercicio, el intervalo se puede calcular directamente usando el percentil, estimando un intervalo de $C = (0.92, 0.98)$

> **Nota**
>
> El método de bootstrap para calcular el error estándar y el método de percentil para calcular el intervalo de confianza están implementados en la clase `SE` y `CI` respectivamente. Las siguientes instrucciones se pueden utilizar para calcular el error estándar.

```
recall = lambda y, hy: recall_score(y, hy,
                                    average="macro")
se = SE(statistic=recall)
se(y, hy)
```

```
0.0170479222841964
```

Complementando el intervalo de confianza con el método de percentil se implementa en el siguiente código.

```
recall = lambda y, hy: recall_score(y, hy,
                                    average="macro")
ci = CI(statistic=recall)
ci(y, hy)
```

```
(0.9201651126651126, 0.9804857784370467)
```

13.4 Comparación de Algoritmos

Se han descrito varios procedimientos para conocer los intervalos de confianza de un algoritmos de aprendizaje. Es momento para describir la metodología para conocer si dos algoritmos se comportan similar en un problema dado.

13.4.1 Método: Distribución t de Student

Suponiendo que se tienen las medidas de rendimiento de dos algoritmos mediante validación cruzada de K-fold, es decir, se tiene el rendimiento del primer algoritmo como p_i^1 y del segundo como p_i^2 en la i-ésima instancia. Suponiendo que el rendimiento es una normal, entonces la resta, i.e., $p_i = p_i^1 - p_i^2$ también sería normal. Dado que se está comparando los algoritmos en los mismos datos, se puede utilizar la prueba t de Student de muestras dependientes. La estadística de la prueba está dada por $\frac{\sqrt{K}m}{S} \sim t_{K-1}$, donde m y S^2 es la media varianza estimada.

En el siguiente ejemplo se compara el rendimiento de Árboles Aleatorios y Naive Bayes en el problema de Breast Cancer. El primer paso es cargar las librerías así como obtener las predicciones de los algoritmos.

```
K = 30
kf = StratifiedKFold(n_splits=K,
```

```
                      random_state=0,
                      shuffle=True)
X, y = load_breast_cancer(return_X_y=True)

P = []
for tr, ts in kf.split(X, y):
    forest = RandomForestClassifier().fit(X[tr], y[tr]).predict(X[ts])
    naive = GaussianNB().fit(X[tr], y[tr]).predict(X[ts])
    P.append([recall_score(y[ts], hy, average="macro") for hy in [forest, naive]])
P = np.array(P)
```

Como se puede observar la medida de rendimiento es macro-recall. Continuando con el procedimiento para obtener la estadística t_{K-1}

```
p = P[:, 0] - P[:, 1]
t = np.sqrt(K) * np.mean(p) / np.std(p)
```

donde el valor de la estadística es 3.5981, si el valor está fuera del siguiente intervalo $(-2.045, 2.045)$ se rechaza la hipótesis nula de que los dos algoritmos se comportan similar.

En caso de que la medida de rendimiento no esté normalmente distribuido, la prueba no-parametrica equivalente corresponde a Wilcoxon. La instrucción `wilcoxon(P[:, 0], P[:, 1])` se puede utilizar para calcularla, dando un p_{value} de 0.0017. En ambos casos podemos concluir que los algoritmos Árboles Aleatorios y Naive Bayes son estadisticamente diferentes con una confianza del 95% en el problema de Breast Cancer.

13.4.2 Método: Bootstrap en diferencias

Un método para comparar el rendimiento de dos algoritmo que no asume ningún tipo de distribución se puede realizar mediante la técnica de Bootstrap. Nava-Muñoz, Graff Guerrero, y Escalante (2023) utilizan esta idea para comparar diferentes algoritmos en el esquema de una competencia de aprendizaje supervisado. La idea es calcular las predicciones de los algoritmos y realizar la muestra calculando en cada una la diferencia del rendimiento. Este se procedimiento se explicará mediante un ejemplo.

El primer paso es calcular las predicciones de los algoritmos, en este caso se realizar una validación cruzada, tal y como se muestra a continuación.

```
forest = np.empty_like(y)
naive = np.empty_like(y)
for tr, ts in kf.split(X, y):
```

```
    forest[ts] = RandomForestClassifier().fit(X[tr], y[tr]).predict(X[ts])
    naive[ts] = GaussianNB().fit(X[tr], y[tr]).predict(X[ts])
```

El macro-recall para los Bosques Aleatorios es 0.95 y para el Naive Bayes es 0.93. Lo que se observa es que los bosques tienen un mejor rendimiento, entonces la distribución de la diferencia del rendimiento entre bosques y Naive Bayes no debería de incluir al cero, si lo incluye la masa que está al lado izquierdo del cero debe de ser menor, esa mas corresponde al valor p.

Las muestras de la diferencia de rendimiento se pueden calcular de las siguientes instrucciones.

```
S = np.random.randint(y.shape[0],
                      size=(500, y.shape[0]))
r = recall_score
diff = lambda y, hy1, hy2: r(y, hy1, average="macro") -\
                           r(y, hy2, average="macro")
B = [diff(y[s], forest[s], naive[s])
     for s in S]
```

Finalmente, el p_{value} corresponde a la proporción de elementos que son menores que cero, i.e., `(np.array(B) < 0).mean()`, es decir, aquellas muestras donde Naive Bayes tiene un mejor desempeño que los bosques. En este caso el p_{value} tiene un valor de 0.0100. Dado que el valor es menor que 0.05 se puede rechazar la hipótesis nula con una confianza superior al 95% y concluir que existe una diferencia estadísticamente significativa en el rendimiento entre los dos algoritmos. La Figura 13.1 muestra la distribución de la diferencia de rendimiento, en esta se puede observar como la mayor parte de la masa se encuentra del lado positivo y que muy poca masa es menor que cero.

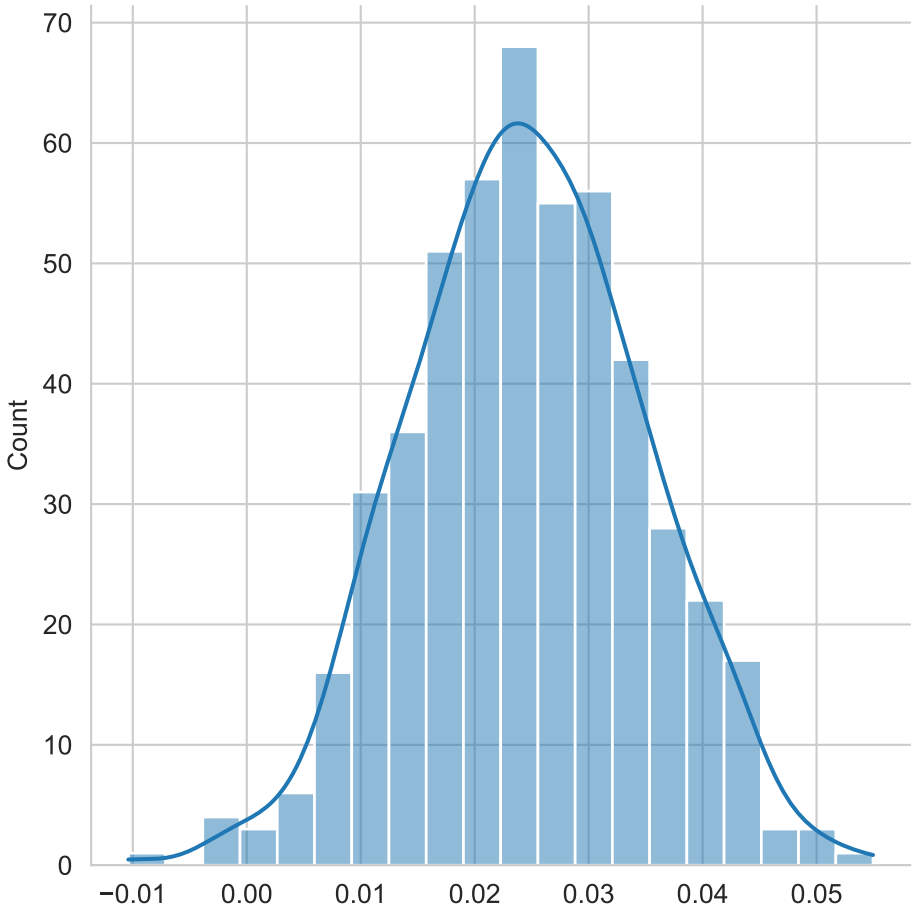


Figura 13.1: Distribución de la diferencia de rendimiento

Referencias

McInnes, Leland, John Healy, y James Melville. 2020. «UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction». https://arxiv.org/abs/1802.03426.

Nava-Muñoz, Sergio, Mario Graff Guerrero, y Hugo Jair Escalante. 2023. «Comparison of Classifiers in Challenge Scheme». En *Pattern Recognition*, editado por Ansel Yoan Rodríguez-González, Humberto Pérez-Espinosa, José Francisco Martínez-Trinidad, Jesús Ariel Carrasco-Ochoa, y José Arturo Olvera-López, 89-98. Cham: Springer Nature Switzerland. https://doi.org/10.1007/978-3-031-33783-3_9.

Quinlan, J. R. 1986. «Induction of decision trees». *Machine Learning 1986 1:1* 1 (marzo): 81-106. https://doi.org/10.1007/BF00116251.

Sebastiani, Fabrizio. 2015. «An Axiomatically Derived Measure for the Evaluation of Classification Algorithms». En *Proceedings of the 2015 International Conference on The Theory of Information Retrieval*, 11-20. ICTIR '15. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/2808194.2809449.

Wasserman, Larry. 2004. *All of Statistics : A Concise Course in Statistical Inference*. Springer. https://doi.org/10.1007/978-0-387-21736-9.

A Estadística

El **objetivo** de este apéndice es complementar la información de algunos procedimientos estadísticos usados en el curso.

Paquetes usados

```
from scipy.stats import norm
from matplotlib import pylab as plt
import numpy as np
import pandas as pd
import seaborn as sns
```

A.1 Error estándar

El **error estándar** está definido como $\sqrt{\mathbb{V}(\hat{\theta})}$ donde $\hat{\theta}$ es el valor estimado. No para todas las estadísticas es posible tener una ecuación analítica para calcular $\sqrt{\mathbb{V}(\hat{\theta})}$ y en los casos donde el valor analítico no se puede calcular se puede usar la técnica de Bootstrap.

A.1.1 Media

Una de las estadísticas donde si se puede calcular analíticamente $\sqrt{\mathbb{V}(\hat{\theta})}$ es la media, es decir, se tiene una muestra $\mathcal{D}$ con N elementos independientes y idénticamente distribuidos, entonces la media corresponde a

$$\bar{x} = \frac{1}{N} \sum_{x \in \mathcal{D}} x.$$

El error estándar de $\bar{x}$ es $\sqrt{\mathbb{V}(\bar{x})}$. Para derivar el valor analítico de este error estándar es necesario utilizar la siguiente propiedad de la varianza:

$$\mathbb{V}(\sum_i a_i \mathcal{X}_i) = \sum_i a_i^2 \mathbb{V}(\mathcal{X}_i),$$

donde a_i representa una constante y las variables aleatorias $\mathcal{X}$ son independientes. En lugar de utilizar la definición asumiendo la realización de las variables aleatorias, esto es, cuando $\mathcal{D}$ tiene valores, se define la media con respecto a N variables aleatorias, i.e., $\bar{\mathcal{X}} = \frac{1}{N}\sum_i^N \mathcal{X}_i$. En estas condiciones se observa que para el caso del error estándar de la media la constante es $\frac{1}{N}$ y las variables son independientes entonces

$$\begin{aligned}
\sqrt{\mathbb{V}(\bar{\mathcal{X}})} &= \sqrt{\mathbb{V}(\frac{1}{N}\sum_i^N \mathcal{X}_i)} \\
&= \sqrt{\sum_i^N \frac{1}{N^2}\mathbb{V}(\mathcal{X}_i)} \\
&= \sqrt{\frac{1}{N^2}\sum_i^N \sigma^2} \\
&= \sqrt{\frac{N}{N^2}\sigma^2} \\
&= \sqrt{\frac{\sigma^2}{N}} = \frac{\sigma}{\sqrt{N}},
\end{aligned}$$

donde σ^2 es la varianza de la distribución. Es importante notar que $\mathbb{V}(X_i)$ es independiente de i dado $\mathcal{X}_i \sim F$ para cualquier i, donde la distribución F tiene una varianza σ^2 por lo tanto $\mathbb{V}(X_i) = \sigma^2$.

A.1.2 Ejemplo: Media

El siguiente ejemplo complementa la información al presentar el error estándar de la media cuando los datos vienen de una distribución Gausiana. Suponiendo que se tiene 1000 muestras de una distribución Gausiana $\mathcal{N}(1, 4)$, i.e., $\mu = 1$ y $\sigma = 2$. La error estándar de estimar la media con esos datos está dado por $\mathbb{V}(\widehat{\mu}) = \frac{\sigma}{\sqrt{N}} = \frac{2}{\sqrt{1000}} = 0.0632$.

Continuado con el ejemplo, se simula la generación de esta población de 1000 elementos. El primer paso es iniciar la clase `norm` (que implementa una distribución Gausiana) para que se simule $\mathcal{N}(1, 4)$. Es importante notar que el parámetro `scale` de `norm` corresponde a la desviación estándar σ.

```
p1 = norm(loc=1, scale=2)
```

Usando `p1` se simulan 500 poblaciones de 1000 elementos cada una, y para cada una de esas poblaciones se calcula su media. La primera linea crea la muestra $\mathcal{D}$ y a continuación se calcula la media por cada población, renglón de `D`.

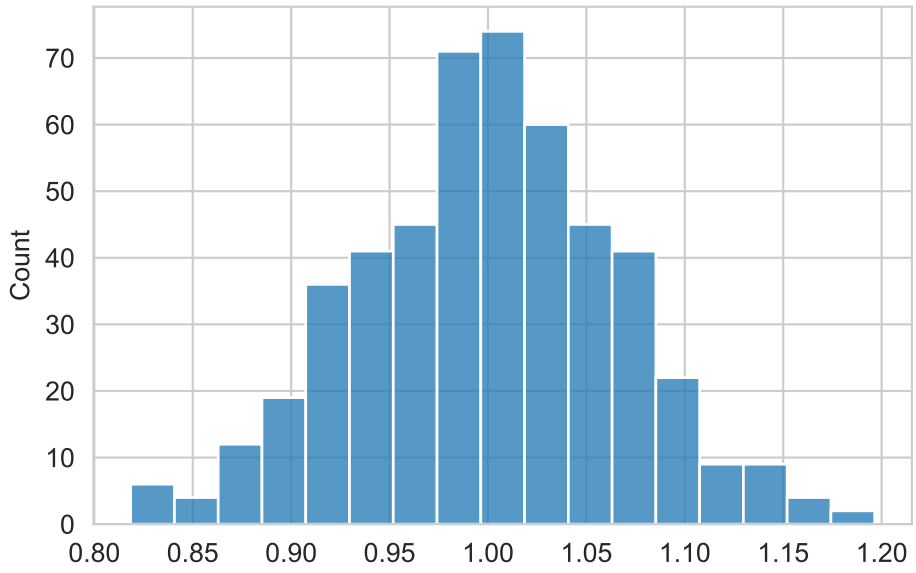
```
D = p1.rvs(size=(500, 1000))
mu = [x.mean() for x in D]
```

El error estándar es la desviación estándar de `mu`, el cual se puede calcular con la siguiente instrucción. `se` tiene un valor de 0.0663, que es similar al obtenido mediante $\mathbb{V}(\hat{\mu})$.

```
se = np.std(mu)
```

Para complementar la información se presenta el histograma de `mu` donde se puede observar la distribución de estimar la media de una población.

```
fig = sns.histplot(mu)
```



A.1.3 Ejemplo: Coeficientes OLS

El error estándar de los coeficientes estimados con mínimos cuadrados se puede calcular de la siguiente forma. El primer paso es utilizar la siguiente identidad

$$\mathbb{V}(A\mathcal{Y}) = A\Sigma A^\top,$$

donde A es una matriz y $\mathcal{Y}$ es un vector de variables aleatorias. La matriz A en OLS es

$$A = (X^\top X)^{-1} X^\top.$$

quedando la varianza como

$$\mathbb{V}(\mathbf{w}) = A\Sigma A^\top,$$

donde Σ es la covarianza de $\mathcal{Y}$. Dado que $\mathcal{Y}$ tiene una varianza constante entonces $\Sigma = \sigma^2 I$. Usando esta información se puede derivar la varianza de $\mathbf{w}$ de la siguiente manera

$$\begin{aligned}
\mathbb{V}(\mathbf{w}) &= A\sigma^2 I A^\top \\
&= \sigma^2 A A^\top \\
&= \sigma^2 (X^\top X)^{-1} X^\top ((X^\top X)^{-1} X^\top)^\top \\
&= \sigma^2 (X^\top X)^{-1} X^\top X (X^\top X)^{-1} \\
&= \sigma^2 (X^\top X)^{-1}
\end{aligned}$$

Por lo tanto el error estándar es: $\mathsf{se}(\mathbf{w}) = \sigma\sqrt{(X^\top X)^{-1}}$.

A.2 Bootstrap

Existen ocasiones donde no se cuenta con una ecuación cerrada para $\mathbb{V}(\hat{\theta})$, un ejemplo sería la mediana. En aquellas estadísticas donde no se tenga el error estándar no se pueda calcular analíticamente se puede utilizar Bootstrap.

Bootstrap es un procedimiento que permite calcular el error estándar. Suponiendo que se la estadística se calcula mediante la realización de N variables aleatorias, $\mathcal{X} \sim F$, es decir, $\theta = g(\mathcal{X}_1, \mathcal{X}_2, \dots, \mathcal{X}_N)$. Por ejemplo, la media es $g(\mathcal{X}_1, \mathcal{X}_2, \dots, \mathcal{X}_N) = \frac{1}{N}\sum_i^N \mathcal{X}_i$.

El Bootstrap simula la distribución θ mediante la selección con remplazo de N elementos del conjunto de la muestra $\mathcal{D} = (x_1, x_2, \dots, x_n)$ donde $x_i \sim F$. Es decir, se genera un nuevo

conjunto $\mathcal{D}_j = (x_{j_1}, x_{j_2}, \ldots, x_{j_n})$ donde x_{j_1} podría ser x_4 de $\mathcal{D}$ y dado que se selecciona con reemplazo entonces x_4 podría aparecer otra vez en $\mathcal{D}_j$. Utilizando $\mathcal{D}_j$ se puede estimar la estadística $\hat{\theta}_j = g(x_{j_1}, x_{j_2}, \ldots, x_{j_n})$. Bootstrap repite el proceso anterior B donde en cada iteración se selecciona con reemplazo N veces $\mathcal{D}$. Utilizando las θ_j calculadas se estima $\mathbb{V}(\theta)$ con la siguiente ecuación

$$\mathbb{V}(\hat{\theta}) = \frac{1}{B}\sum_{j=1}^{B}(\hat{\theta}_j - \frac{1}{B}\sum_{k}^{B}\hat{\theta}_k)^2.$$

A.2.1 Ejemplo

Utilizando los datos generados en la Sección A.1.2 se puede calcular el error estándar de la mediana. El primer paso es tener el conjunto $\mathcal{D}$ el cual puede ser cualquier renglón de los 500 de la variable `D`, por simplicidad se toma el primero como se muestra a continuación.

```
D_mediana = D[0]
```

La variable `D_mediana` tiene la muestra $\mathcal{D}$ con la que se trabajará para estimar el error estándar de la mediana. Se tienen que generar B repeticiones de muestrear $\mathcal{D}$ N veces con reemplazo. Esto se puede implementar usando índices y números aleatorios de $[0, N)$ considerando que en Python el primer índice es 0. Este procedimiento se muestra en la primera linea del siguiente código. El arreglo `S` contiene los índices para realizar las B muestras, en la segunda linea se itera por los renglones de `S` y en cada operación se calcula la mediana. Es decir, en cada iteración se está calculando un $\hat{\theta}_i$ que corresponde a la mediana. Finalmente, se calcula la desviación estándar de la lista `B` y ese valor corresponde a error estándar de la mediana.

```
S = np.random.randint(D_mediana.shape[0],
                      size=(500, D_mediana.shape[0]))
B = [np.median(D_mediana[s]) for s in S]
se = np.std(B)
```

`se` tiene un valor de 0.0836.

Considerando que se generaron 500 poblaciones de 1000 elementos que se encuentra en la variable `D` se puede visualizar el histograma de las medianas calculadas con `D` y aquellas obtenidas con Bootstrap. Se guardan estos dos valores en un `DataFrame` para posteriormente graficar el histograma, como se muestra en el siguiente código y en la Figura A.1.

```
df = pd.DataFrame(dict(Bootstrap=B,
                       Muestra=[np.median(x) for x in D]))
fig = sns.histplot(df)
```

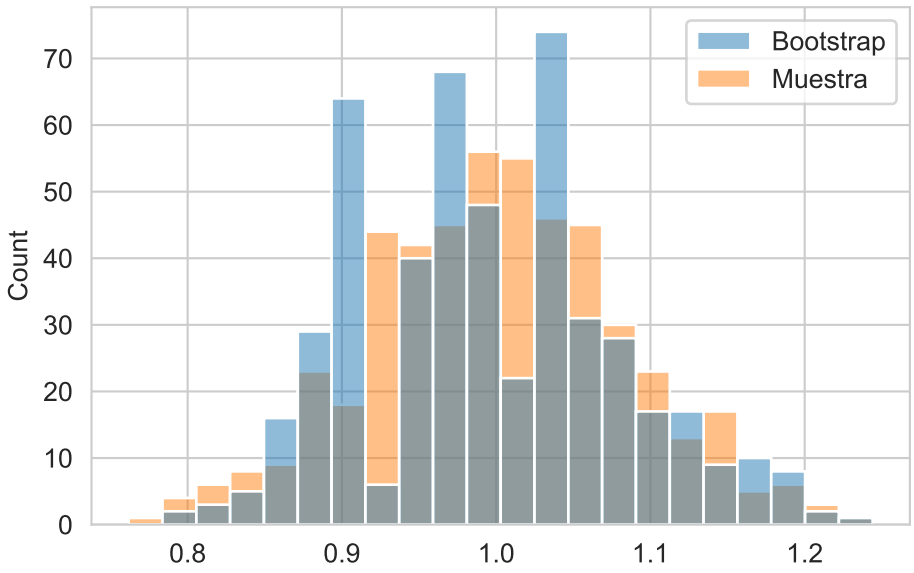


Figura A.1: Histograma de la mediana

B Código

El **objetivo** de este apéndice es describir algunas de las clases y métodos de Python utilizadas en el curso.

Paquetes usados

```
from scipy.stats import multivariate_normal
from scipy.special import logsumexp
import numpy as np
```

B.1 Clasificador Bayesiano Gausiano

```
class GaussianBayes(object):
    def __init__(self, naive=False) -> None:
        self._naive = naive

    @property
    def naive(self):
        return self._naive

    @property
    def labels(self):
        return self._labels

    @labels.setter
    def labels(self, labels):
        self._labels = labels
```

B.1.1 Estimación de Parámetros

```
    def fit(self, X, y):
        self.prior = y
        self.likelihood = (X, y)
        return self
```

```
    @property
    def prior(self):
        return self._prior

    @prior.setter
    def prior(self, y):
        labels, counts = np.unique(y, return_counts=True)
        prior = counts / counts.sum()
        self.labels = labels
        self._prior = np.log(prior)
```

```
    @property
    def likelihood(self):
        return self._likelihood

    @likelihood.setter
    def likelihood(self, D):
        X, y = D
        likelihood = []
        for k in self.labels:
            mask = y == k
            mu = np.mean(X[mask], axis=0)
            if self.naive:
                cov = np.var(X[mask], axis=0, ddof=1)
            else:
                cov = np.cov(X[mask], rowvar=False)
            _ = multivariate_normal(mean=mu,
                                    cov=cov,
                                    allow_singular=True)
            likelihood.append(_)
        self._likelihood = likelihood
```

B.1.2 Predicción

```
    def predict(self, X):
        hy = self.predict_log_proba(X)
        _ = np.argmax(hy, axis=1)
        return self.labels[_]
```

```
    def predict_proba(self, X):
        _ = self.predict_log_proba(X)
        return np.exp(_)
```

```
    def predict_log_proba(self, X):
        log_ll = np.vstack([m.logpdf(X)
                            for m in self.likelihood]).T
        prior = self.prior
        posterior = log_ll + prior
        evidence = np.atleast_2d(logsumexp(posterior,
                                           axis=1)).T
        return posterior - evidence
```

B.1.3 Uso

```
bayes = GaussianBayes().fit(T, y_t)
hy = bayes.predict(G)
```

C Conjunto de Datos

El **objetivo** de este apéndice es listar los conjuntos de datos utilizados en el curso.

Paquetes usados

```
from sklearn.datasets import load_breast_cancer,\
                             load_diabetes,\
                             load_digits, load_iris,\
                             load_wine
from scipy.stats import multivariate_normal
from matplotlib import pylab as plt
import numpy as np
import pandas as pd
import seaborn as sns
```

C.1 Problemas Sintéticos

En esta sección se presentan los problemas sintéticos que corresponden aquellos problemas en los que se conocen todos los parámetros y se usan para mostrar algunas características de los algoritmos.

C.2 Mezcla de Clases

```
p1 = multivariate_normal(mean=[5, 5],
                         cov=[[4, 0], [0, 2]])
X_1 = p1.rvs(size=1000)
p2 = multivariate_normal(mean=[1.5, -1.5],
                         cov=[[2, 1], [1, 3]])
X_2 = p2.rvs(size=1000)
```

```
p3 = multivariate_normal(mean=[12.5, -3.5],
                         cov=[[2, 3], [3, 7]])
X_3 = p3.rvs(size=1000)
```

Figura C.1 muestra estas tres distribuciones.

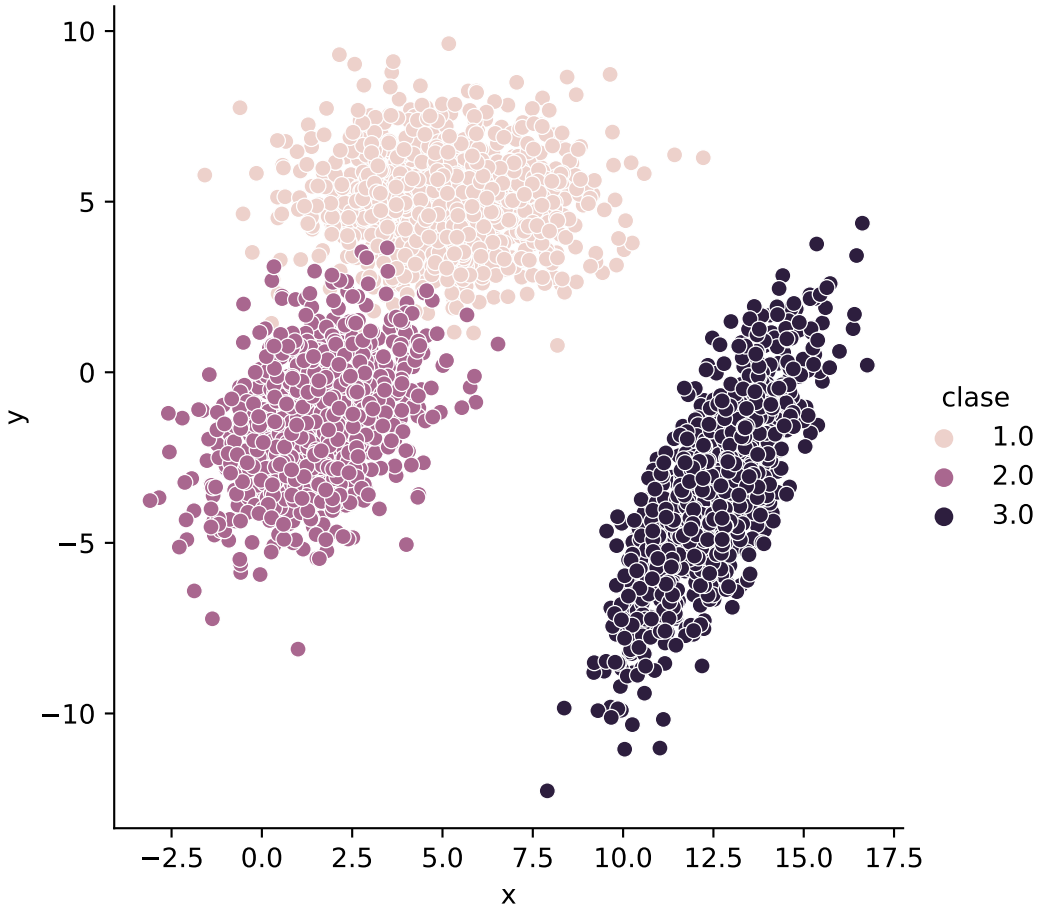


Figura C.1: Muestras de 3 distribuciones gausianas

C.2.1 Clases Separadas

```
X_1 = multivariate_normal(mean=[5, 5],
                          cov=[[4, 0], [0, 2]]).rvs(1000)
X_2 = multivariate_normal(mean=[-5, -10],
```

```
                              cov=[[2, 1], [1, 3]]).rvs(1000)
X_3 = multivariate_normal(mean=[15, -6],
                              cov=[[2, 3], [3, 7]]).rvs(1000)
```

Este problema se muestra en la Figura C.2.

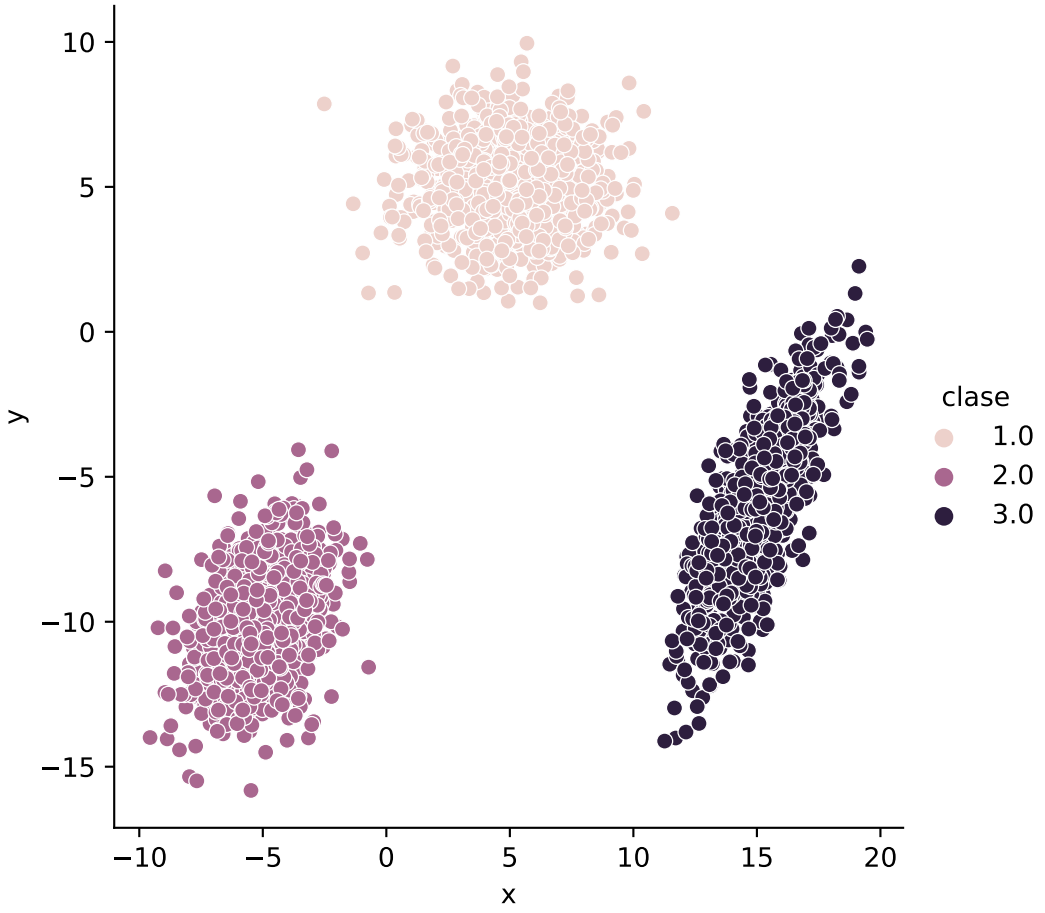


Figura C.2: Muestras de 3 distribuciones gausianas

C.3 Problemas de Clasificación

En esta sección se listan los problemas de clasificación utilizados durante el curso.

C.3.1 Breast Cancer Wisconsin

El conjunto de datos de Breast Cancer Wisconsin se obtiene con el siguiente código.

```
D, y = load_breast_cancer(return_X_y=True)
```

C.3.2 Iris

Un conjunto clásico en problemas de clasificación es el problema del Iris que se encuentra con las siguientes instrucciones.

```
D, y = load_iris(return_X_y=True)
```

C.3.3 Números

El conjunto de Digits es un conjunto de clasificación donde se trata de identificar el número escrito en una imagen; este conjunto de datos se descarga utilizando las siguientes instrucciones.

```
D, y = load_digits(return_X_y=True)
```

C.3.4 Vino

El conjunto de vino es un problema que tiene 3 clases, 178 ejemplos y se encuentra representado en $\mathbb{R}^{13}$; este problema se obtiene con las siguientes instrucciones.

```
D, y = load_wine(return_X_y=True)
```

C.4 Problemas de Regresión

En esta sección se listan los problemas de regresión utilizados para ejemplificar los algoritmos y su rendimiento.

C.4.1 Problema Sintético

El siguiente ejemplo es un problema de regresión sintético que se forma de la suma de dos funciones trascendentales como se muestra en el siguiente código.

```
X = np.linspace(-5, 5, 100)
y = np.sin(X) + 0.3 * np.cos(X * 3.)
```

La Figura C.3 muestra este problema sintético.

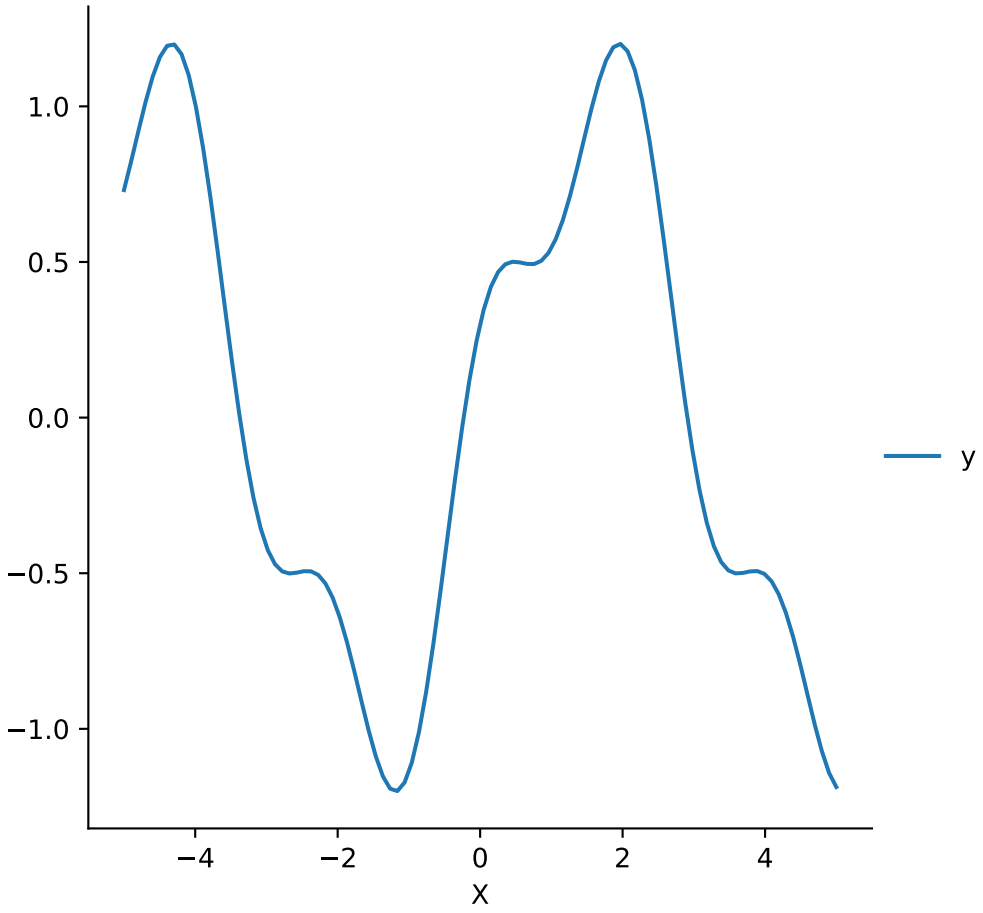


Figura C.3: Problema de Regresión

C.4.2 Diabetes

El conjunto de datos Diabetes es un problema que se puederecuperar usando el siguiente código.

```
D, y = load_diabetes(return_X_y=True)
```