FACULTAD DE MATEMÁTICAS

DEPARTAMENTO DE ESTADÍSTICA E INVESTIGACIÓN OPERATIVA

APRENDIZAJE SEMISUPERVISADO

Ignacio Roldán Bocanegra

APRENDIZAJE SEMISUPERVISADO

Memoria realizada por Ignacio Roldán Bocanegra

Dirigido por:

V^{o}B^{o}

Dr. Rafael Blanquero Bravo

Resumen

La presente memoria tiene como objetivo estudiar los conceptos fundamentales del aprendizaje semisupervisado, además de exponer algunas hipótesis necesarias para el correcto funcionamiento de algunos de sus métodos, que también explicaremos. Para ello, en el primer capítulo partiremos de las nociones básicas del aprendizaje automático, hasta llegar a las principales del aprendizaje semisupervisado, junto con las suposiciones anteriormente mencionadas. Más adelante, en el segundo capítulo expondremos algunas técnicas de validación que usaremos en los algoritmos propios de dicho aprendizaje, que explicaremos en el tercero, y con su implementación en `R` en el cuarto; para finalizar, en el quinto capítulo se presentará alguna aplicación con datos reales.

Abstract

The objective of this work is to study the essential concepts for semi-supervised learning, as well as to explain some assumptions that are necessary for the proper operation of some methods, which will also be explained. To achieve this goal, in the first chapter we will start from the basic notions of machine learning, to the ones of semi-supervised learning, along with the above assumptions. Later, in the second chapter we will explain some validation techniques which we will use in the algorithms of this type of learning, which we will explain in the third one, and with its implementation in `R` in the fourth one; finally, in the fifth chapter some applications based on real data will be presented.

Índice general

Introducción al aprendizaje automático

En los últimos años, el aprendizaje automático ha cobrado una gran importancia en el análisis de cualquier conjunto de datos. Tradicionalmente, se han considerado dos tipos distintos de aprendizaje automático [1].

El primero de ellos es el **aprendizaje no supervisado**. Sea $X = \{x_i\}_{i \in I}$ un conjunto con n instancias, donde x_i es un vector con d componentes $\forall i \in I \coloneqq \{1, \ldots, n\}$.

A veces es conveniente definir la matriz $(n \times d)-$dimensional, $\mathbf{X} = \left(x_i^T\right)_{i \in I}$, donde cada fila corresponde a una instancia.

El objetivo del aprendizaje no supervisado es el de encontrar una estructura que permita estudiar con más facilidad el conjunto X. Aunque se suele identificar este problema como el de la estimación de la densidad generada por X, también hay otros objetivos del aprendizaje no supervisado, como, por ejemplo, la detección de *outliers* [2].

Por otro lado, tenemos el **aprendizaje supervisado**. El objetivo es encontrar una relación entre x e y, a partir de un conjunto de entrenamiento con pares $\{(X, Y)\} = \{(x_i, y_i)\}$ dado, $\forall i \in I$. En este caso, y es lo que se conoce como la variable objetivo de las variables predictoras x, donde x es un vector con d componentes.

Cuando $y \in \mathbb{R}$, el problema del aprendizaje supervisado se conoce como **regresión**. Sin embargo, en esta memoria nos centraremos en problemas de **clasificación**, es decir, donde la variable y toma valores discretos.

En este último tipo de aprendizaje automático, los algoritmos se dividen en dos tipos: por un lado están los **métodos generativos**, que tratan de modelar cómo los datos son generados; y por otro lado tenemos los **métodos discriminativos**, que buscan predecir la variable y en función de x, es decir, intentan predecir $p(y|x)$.

Sin embargo, a medida que se fue profundizando en el estudio de las técnicas de aprendizaje automático, se comenzó a investigar un tercer tipo, conocido como **aprendizaje semisupervisado** [3].

El aprendizaje semisupervisado se encuentra a medio camino entre los dos tipos anteriormente descritos. Trata con conjuntos de datos donde se proporciona una información extra de la variable objetivo, como en el caso supervisado, pero no necesariamente para todas las instancias.

En este caso, el conjunto $X = \{x_i\}_{i \in I}$ se puede dividir en dos: $X_l \coloneqq \{x_1, \ldots, x_l\}$, para los que los valores $Y_l \coloneqq \{y_1, \ldots, y_l\}$ son conocidos, y $X_u \coloneqq \{x_{l+1}, \ldots, x_{l+u}\}$, donde los valores de la variable y son desconocidos.

En esta memoria, nos centraremos en el estudio de este tipo de aprendizaje automático, ya que muchas veces es fácil obtener una base de datos donde no todas las instancias tienen etiqueta. De hecho, es menos común encontrar una base de datos con todos los ejemplos etiquetados, ya que el proceso de etiquetado puede ser costoso o puede plantear dificultades técnicas.

Por otro lado, aunque podría plantearse realizar el estudio ignorando los ejemplos no etiquetados, éstos pueden proporcionar información útil para mejorar el algoritmo de aprendizaje automático a utilizar.

Es por ello que en capítulos posteriores expondremos los fundamentos para entender este tipo de aprendizaje automático, dando primero unas nociones básicas del aprendizaje automático en general y del aprendizaje semisupervisado, en particular, además de proporcionar algunos algoritmos útiles para conjuntos de datos con las características descritas.

Capítulo 1

Nociones básicas

En este capítulo, nos centraremos en el aprendizaje semisupervisado, además de dar alguna noción general de conceptos necesarios para la validación de algoritmos.

1.1. Tipos según el objetivo

El aprendizaje semisupervisado, debido a su estructura, se puede dividir principalmente en dos tipos [4], según el objetivo del análisis que se busque hacer a la base de datos.

Por un lado, tenemos el **aprendizaje transductivo**, introducido por Vapnik [5], que separa el conjunto dado en conjunto de entrenamiento (donde la variable objetivo es conocida), y conjunto test donde ésta no es dada. El objetivo de este tipo es tratar de predecir los valores de la variable objetivo del conjunto test.

Por otro lado, el **aprendizaje inductivo** trata de obtener una función de predicción usando todo el conjunto $\{(X, Y)\}$, es decir, usando tanto los datos donde la variable objetivo es conocida como aquellos en los que no.

En esta memoria nos centraremos en los modelos transductivos.

1.2. Tipos de datos no etiquetados

Según algunos autores como Xu [6], los ejemplos no etiquetados pueden dividirse en una serie de categorías en función de su calidad. Dichos tipos son los siguientes:

- Tipo 1: Las instancias no etiquetadas son generadas por la misma distribución que los ejemplos etiquetados, por lo que se puede usar el mismo conjunto de etiquetas que el de las instancias etiquetadas.

- Tipo 2: Las instancias no etiquetadas son generadas por una distribución distinta que los ejemplos etiquetados, pero puede usarse el mismo conjunto de etiquetas que el de las instancias etiquetadas.

- Tipo 3: Los ejemplos no etiquetados no pueden usar el conjunto de etiquetas, pero comparten cierta estructura con las instancias etiquetadas.

- Tipo 4: Las instancias no etiquetadas corresponden a información irrelevante o que tiene poca relación con los ejemplos etiquetados.

- Tipo 5: Los ejemplos no etiquetados corresponden a dos o tres tipos de datos distintos.

En función del tipo, el aprendizaje semisupervisado será más útil o no. Esto depende de unas suposiciones que se detallarán en la siguiente sección.

1.3. Suposiciones necesarias

Como hemos descrito anteriormente, el aprendizaje semisupervisado es un tipo de aprendizaje automático, creado para el estudio de conjuntos de datos menos restrictivos.

Pero, ¿es realmente interesante este estudio? Es decir, ¿proporciona mejores resultados que si consideramos únicamente los otros tipos de aprendizaje automático?

Parece obvio decir que sí lo es. Sin embargo, hay que aclarar algo importante y es que, para que la información extra que proporciona este tipo de aprendizaje ayude a la obtención de mejores resultados, es necesario que dicha información sea relevante para el problema de clasificación.

En caso contrario, el aprendizaje semisupervisado no tiene por qué mejorar los resultados del aprendizaje supervisado.

Por otro lado, para el correcto desarrollo de este tipo de técnicas, es necesario definir ciertas hipótesis que deben cumplirse.

1.3.1. *The Smoothness Assumption*

La primera suposición, conocida como ***The Smoothness Assumption*** (que en español sería 'la suposición de suavidad'), trata de adecuarse a la suposición que se suele hacer en el aprendizaje supervisado sobre la distancia entre instancias.

The Smoothness Assumption: Si dos puntos x_1, x_2 de una región de alta densidad se encuentran próximos entre sí, entonces lo mismo debe ocurrir con los correspondientes valores de la variable respuesta.

Esta suposición implica que si dos puntos están conectados de alguna forma (por ejemplo, pertenecen al mismo grupo), entonces sus resultados en la variable objetivo deben estar próximos también.

Es importante tener en cuenta que esta suposición se aplica tanto a problemas de clasificación como de regresión.

1.3.2. *The Cluster Assumption*

Supongamos que sabemos que los puntos de la misma clase tienden a formar un *cluster* o grupo en el espacio determinado por las variables predictoras. Entonces, los ejemplos que no están etiquetados pueden ayudar a encontrar con mayor facilidad esta relación, para su correcta clasificación.

En otras palabras, se podría llegar a crear un algoritmo sobre los *clusters* y usar las instancias etiquetadas para asociar cada clase a cada *cluster*.

De esta forma se establece ***The Cluster Assumption***, o en español, 'la suposición de los grupos', que puede formularse como sigue:

The Cluster Assumption: Si dos instancias están en el mismo *cluster*, éstos probablemente pertenecerán a la misma clase también.

Es importante destacar que esta suposición no implica que cada clase forme un único y compacto *cluster*, sino que usualmente no solemos observar dos clases distintas en el mismo *cluster*. Sin embargo, si podría ocurrir que una misma clase se divida en dos *clusters* distintos.

Además, esta suposición puede verse como un caso especial de la suposición anterior.

1.3.3. *The Manifold Assumption*

Por último, tenemos ***The Manifold Assumption***, que en español se podría traducir como 'el supuesto de las variedades'. Esta suposición cons-

tituye la base de muchos métodos de aprendizaje semisupervisado y puede formularse de la siguiente forma:

The Manifold Assumption: Los datos (que son de gran dimensión) se encuentran (aproximadamente) en variedades de dimensiones mucho más bajas.

Esta suposición es importante ya que permite enfrentarse a *the curse of dimensionality* o la maldición de la dimensión, que básicamente relaciona el hecho de que la complejidad crece de forma exponencial con el número de dimensiones y, por tanto, es necesario también un crecimiento exponencial del número de instancias dadas.

Sin embargo, el uso tanto de los ejemplos etiquetados como no etiquetados puede evitar este problema, ya que se dispone de información extra.

1.4. Medidas de validación de un modelo

Antes de comenzar con la descripción de los distintos algoritmos de aprendizaje semisupervisado, vamos a introducir una serie de conceptos necesarios para la evaluación del rendimiento de un modelo. Recordemos que en esta memoria nos centramos en modelos de clasificación bajo un enfoque transductivo.

Como hemos dicho anteriormente, los modelos transductivos separan el conjunto dado en **conjunto de entrenamiento**, donde los valores de la variable objetivo son conocidos, y en **conjunto test** donde éstos no son conocidos. Sin embargo, en el aprendizaje semisupervisado esto no es exactamente así.

En los algoritmos de aprendizaje semisupervisado, el conjunto de entrenamiento se compone de un subconjunto de los ejemplos etiquetados (que constituyen la base del algoritmo) y el conjunto de ejemplos no etiquetados (que completan el algoritmo), cosa que no ocurre en el aprendizaje supervisado, y el conjunto test consta de los ejemplos etiquetados restantes del conjunto de entrenamiento. Esto hace que se elaboren algoritmos distintos que en el aprendizaje supervisado, ya que normalmente se trabaja con un número pequeño de instancias etiquetadas.

Además, las instancias etiquetadas del conjunto test permiten evaluar el rendimiento del modelo, ya que al predecir los valores de la variable respuesta del conjunto test, conocemos tanto los valores reales de los ejemplos de este conjunto, como los pronosticados.

A continuación, mostraremos las distintas medidas que se pueden obtener una vez empleado el modelo de aprendizaje semisupervisado al conjunto test.

1.4.1. Matriz de confusión

Supongamos que tenemos una variable objetivo cualitativa con c modalidades, que divide al conjunto de datos en c clases. Como hemos dicho anteriormente, una vez aplicado el modelo al conjunto test, tendremos instancias donde sabemos tanto el valor real de su variable objetivo, como el valor de dicha variable predicho por el algoritmo. Es aquí donde entra en juego la denominada **matriz de confusión**.

La matriz de confusión es una matriz $c \times c$, donde las filas y las columnas toman los nombres de las c clases. Dicha matriz toma en cada una de sus celdas el número de instancias de la clase de la fila correspondiente que han sido clasificadas en la clase de su columna. De forma general, tiene el siguiente aspecto:

		PREDICCIÓN			
		1	...	c	
CLASE REAL	**1**	m_{11}	...	m_{1c}	$m_{1.}$
	⋮	⋮	⋱	⋮	⋮
	c	m_{c1}	...	m_{cc}	$m_{c.}$
		$m_{.1}$	...	$m_{.c}$	$m_{..}$

De forma más concreta, vamos a definir los siguientes conceptos relativos a la matriz de confusión:

- m_{ij} es el número de instancias de la clase i clasificadas en la clase j, $\forall i, j = 1, \ldots, c$.
- $m_{i.}$ es el número total de individuos pertenecientes a la clase i, $\forall i = 1, \ldots, c$.

  Es decir, corresponden a la suma de los valores i-ésima fila:

  $m_{i.} = \sum_{j=1}^{c} m_{ij}, \forall i = 1, \ldots, c$.

- $m_{.j}$ es el número total de individuos clasificados en la clase j,

  $\forall j = 1, \ldots, c.$

  Es decir, corresponden a la suma de los valores de la j-ésima columna: $m_{.j} = \sum_{i=1}^{c} m_{ij}, \forall j = 1, \ldots, c.$

- $m_{..}$ es el número total de individuos en la muestra test. Se verifica:

  $m_{..} = \sum_{j=1}^{c} m_{.j} = \sum_{i=1}^{c} m_{i.} = \sum_{i,j=1}^{c} m_{ij}$

A continuación se presenta un ejemplo sobre la construcción de la matriz de confusión:

- Supongamos que tenemos un conjunto de datos formado por 20 individuos, cuya variable objetivo es el color. De esos 20 individuos, 10 son de color azul, 4 de color rojo y 6 de color verde. Después de haber aplicado el algoritmo de clasificación, obtenemos que, de los 10 individuos azules, se predicen 9 como color azul y 1 como color verde; de los 4 de color rojo se predicen 2 como color rojo, 1 como color azul y 1 como color verde; y, de los 6 de color verde, se predicen los 6 de color verde. La matriz de confusión resultante sería la siguiente:

		PREDICCIÓN		
		Azul	**Rojo**	**Verde**
CLASE REAL	**Azul**	9	0	1
	Rojo	1	2	1
	Verde	0	0	6

Como vemos, la suma de cada fila da el número total real de cada una de las clases. Además, vemos como los elementos de la diagonal son las instancias que se han clasificado de forma correcta, y los demás no.

Sin embargo, la matriz de confusión no proporciona una medida de validación concreta, sino que a partir de ella se pueden definir las siguientes medidas [7].

1.4.2. Tasa de acierto

Por un lado tenemos la **tasa de acierto total** (en inglés *accuracy*) que es la fracción de observaciones que han sido clasificadas correctamente. A veces se suele dar en forma de porcentaje multiplicando por 100.

$$\text{Tasa de acierto total} = \sum_{i=1}^{c} \frac{m_{ii}}{m_{..}}.$$

También se puede definir la tasa de acierto en cada clase de la siguiente forma:

$$\text{Tasa de acierto clase } i\text{-ésima} = \frac{m_{ii}}{m_{i.}}, \forall i = 1, \ldots, c.$$

1.4.3. Error de clasificación

El **error de clasificación global** es la fracción de observaciones que se han clasificado de forma incorrecta. A veces se suele dar en forma de porcentaje multiplicando por 100.

$$\text{Error de clasificación global} = \sum_{\substack{i,j=1 \\ i \neq j}}^{c} \frac{m_{ij}}{m_{..}} = 1 - \text{accuracy}.$$

Al igual que antes, podemos definir el error de clasificación en cada clase de la siguiente forma:

$$\text{Error de clasificación clase } i\text{-ésima} = \sum_{\substack{j=1 \\ j \neq i}}^{c} \frac{m_{ij}}{m_{i.}}, \forall i = 1, \ldots, c.$$

1.4.4. Índice kappa

El **índice kappa** (κ) es una medida de validación que tiene en cuenta las posibles concordancias debidas al azar, y las compara con las observadas por el método. Los valores que puede tomar son los siguientes:

- Valor 1: Concordancia perfecta.

- Valor positivo (entre 0 y 1): Concordancia mejor que la que se obtendría al azar.
- Valor 0: Concordancia debida al azar.
- Valor negativo: Concordancia menor que la que se esperaría por azar.

El índice kappa se define como sigue:

$$\kappa = \frac{\sum_{i=1}^{c} m_{ii} - m_{..}/c}{m_{..} - m_{..}/c}.$$

Por ejemplo, en el ejemplo anterior tendríamos un índice kappa de:

$$\kappa = \frac{(9+2+6) - 20/3}{20 - 20/3} = \frac{18 - 20/3}{20 - 20/3} = 0{,}85.$$

1.4.5. Caso particular de dos clases

En el caso en que el número de clases sea dos (es decir, la variable respuesta sea binaria), las medidas de validación referidas a cada clase de forma individual reciben nombres específicos que suelen usarse con mayor frecuencia en la práctica cuando contamos con datos de este tipo.

Es preciso mencionar que, debido a que la variable es binaria, siempre puede identificarse un valor de la clase como positivo y el otro como negativo, en función del estudio que quiera realizarse. Es por esto que a continuación hablaremos de clase positiva y clase negativa.

En este caso, la matriz de confusión adoptará la siguiente expresión:

		PREDICCIÓN	
		Neg	**Pos**
CLASE REAL	**Neg**	m_{11}	m_{12}
	Pos	m_{21}	m_{22}

Especificidad

La especificidad o tasa de verdaderos negativos es la tasa de acierto en la clase negativa, es decir:

$$\text{Especificidad} = \frac{m_{11}}{m_{11} + m_{12}}.$$

Sensibilidad

La sensibilidad o tasa de verdaderos positivos (en inglés denominado también *Recall*) es la tasa de acierto en la clase positiva, es decir:

$$\text{Sensibilidad} = \frac{m_{22}}{m_{21} + m_{22}}.$$

Tasa de Falsos Positivos

La tasa de falsos positivos es el error de clasificación en la clase negativa, es decir:

$$\text{Tasa de Falsos Positivos} = \frac{m_{12}}{m_{11} + m_{12}}.$$

Tasa de Falsos Negativos

La tasa de falsos negativos es el error de clasificación en la clase positiva, es decir:

$$\text{Tasa de Falsos Negativos} = \frac{m_{21}}{m_{21} + m_{22}}.$$

Valor predictivo positivo

El valor predictivo positivo o **precisión** es la fracción de observaciones clasificadas como positivas de forma correcta, resultado de aplicar el teorema de Bayes.

$$\text{Precisión} = \frac{m_{22}}{m_{12} + m_{22}}.$$

Este valor es una estimación de la probabilidad de que un individuo clasificado como positivo realmente lo sea.

Valor predictivo negativo

El valor predictivo negativo (en inglés *negative predictive value*) es la fracción de observaciones clasificadas como negativas de forma correcta. De forma análoga a la precisión, es el resultado de aplicar el teorema de Bayes y se calcula:

$$\text{Valor predictivo negativo} = \frac{m_{11}}{m_{11} + m_{21}}.$$

Este valor es una estimación de la probabilidad de que un individuo clasificado como negativo realmente lo sea.

Medida-F

La Medida-F, más conocida como *F-measure*, es la media armónica entre la precisión (P) y la *Recall* (R):

$$\text{F-measure} = \frac{2}{\frac{1}{P} + \frac{1}{R}} = \frac{2m_{11}}{2m_{11} + m_{12} + m_{21}}.$$

Curva ROC y AUC

Supongamos que tenemos un clasificador f que depende de un valor umbral h de forma que:

- Si $f(x) < h$ clasificamos en la clase positiva.
- Si $f(x) \geq h$ clasificamos en la clase negativa.

La curva ROC (*Receiver Operating Characteristic*) es una curva que va tomando los puntos (1 - especificidad, sensibilidad), para los distintos valores de h.

- En el punto (0,0) todas las instancias de la clase negativa serán clasificadas de forma correcta, y todas las de la clase positiva de forma incorrecta.
- En el punto (1,1) todas las instancias de la clase positiva serán clasificadas de forma correcta, y todas las de la clase negativa de forma incorrecta.

- En el punto (0,1) todas las instancias son clasificadas de forma correcta.

Una vez obtenemos la curva ROC, podemos calcular la AUC, que es el área bajo dicha curva. Este área se usa como medida de validación, siendo mejor el modelo mientras mayor sea ésta.

Capítulo 2

Técnicas de validación

Además de las medidas de validación de un modelo, existen técnicas de validación que sirven para obtener dichas medidas de forma más precisa y fiable. A continuación detallaremos las más usuales.

2.1. Validación simple

La técnica más básica es la conocida como validación simple o *train-test*. Esta técnica separa el conjunto de instancias etiquetadas, de forma que se elabora el modelo al conjunto de entrenamiento (*train*) y, una vez desarrollado, se aplica al conjunto test.

La proporción usual es 2/3 para el conjunto de entrenamiento y 1/3 para el conjunto test. Además, la elección aleatoria se suele hacer estratificada, es decir, respetando la estructura de clases: si una clase tiene un peso en el conjunto de instancias etiquetadas del 60 %, suelen elegirse los conjunto de entrenamiento y test respetando este porcentaje.

Por otro lado, cuando el modelo depende de parámetros y no conocemos cuáles son los óptimos, se suele dividir el conjunto en tres partes: *train-validation-test*. Con los conjuntos *train-validation* se determinan dichos parámetros óptimos, y con el conjunto test se obtienen las medidas de efectividad.

2.2. Validación cruzada con k-pliegues

En la sección anterior hemos explicado la forma de selección del subconjunto de datos etiquetados para poder valorar el rendimiento de un modelo mediante el cálculo de las medidas de validación.

Sin embargo, esta selección puede producir errores que si se seleccionara otro subconjunto distinto podrían no ocurrir. Estadísticamente hablando, tendríamos un estimador de la medida de efectividad sobre la población con alta variabilidad. Es por ello, que introducimos la **validación cruzada en k-pliegues** [8], ya que se reduce el sesgo de dicho estimador.

La validación cruzada en k-pliegues consiste en una técnica donde se divide el conjunto de datos en k subconjuntos disjuntos, y a cada uno de ellos se le aplica el método y se calculan las medidas de efectividad. Ésto sirve para evitar que el conjunto seleccionado, en caso de no ser un subconjunto representativo de la población, provoque que el modelo realice malas estimaciones.

Para ello, en el caso particular del aprendizaje semisupervisado y de los algoritmos que se expondrán a continuación, la validación cruzada en k-pliegues consistirá en lo siguiente:

1. Dividimos los datos etiquetados en k subconjuntos independientes entre sí, haciendo una división estratificada manteniendo la representatividad de las clases en cada pliegue.

2. Cada uno de los pliegues servirá como conjunto test; por tanto, los $k-1$ restantes servirán como conjunto de entrenamiento para la creación del modelo, usando el algoritmo correspondiente.

3. Una vez creado el modelo con los $k-1$ pliegues, evaluaremos dicho modelo con el pliegue del conjunto test y sacaremos sus medidas de validación.

Así, no tendremos un único valor de cada medida de efectividad, sino que tendremos k valores de cada una de dichas medidas. Por ejemplo, no tendremos una única especificidad, sino que tendremos k especificidad, una por cada uno de los pliegues, y tomaremos como especificidad del modelo completo la media de todas estas.

2.3. Validación cruzada Monte Carlo

Otro tipo de técnica de validación es la validación cruzada de Monte Carlo.

Con la validación cruzada de Monte Carlo se realiza el modelo elegido un número de veces elevado (de cientos o de miles [9]), realizando en cada una de ellas el *train-test*, y promediando sus medidas de validación al final del proceso.

La diferencia fundamental de esta técnica con la validación cruzada en k-pliegues, es que los conjuntos test no tienen por qué ser disjuntos.

Por otro lado, como hemos dicho anteriormente, es necesario un gran número de realizaciones para obtener resultados fiables y concluyentes.

2.4. *Bootstrap*

Por último, tenemos la técnica de validación conocida como *Bootstrap* [10].

El *Bootstrap* es una técnica de remuestreo que se usa para obtener mejores estimaciones y medidas de efectividad más fiables, reduciendo el sesgo y la varianza del modelo.

Esta técnica consiste en seleccionar del conjunto de instancias etiquetadas una muestra con repetición de su tamaño, es decir, de tamaño l, y usar dicha muestra, junto con todas las instancias no etiquetadas, como conjunto de entrenamiento, y las instancias etiquetadas restantes como conjunto test. Al igual que en las demás técnicas, se calculan sus medidas de efectividad y luego se calcula un promedio de cada una de ellas por separado.

Para asegurar su correcto funcionamiento, se recomienda realizar dicho procedimiento un número elevado de veces. Dependiendo de la base de datos, se toman desde unas 200 iteraciones si dicha base de datos consta de muchas instancias, hasta unas 2000 veces, si la base de datos no dispone de muchos ejemplos.

Capítulo 3

Algoritmos

En este capítulo, se presentan los algoritmos más comunes y útiles en conjuntos de datos propios del aprendizaje semisupervisado.

3.1. *Self-training*

Comenzaremos analizando el algoritmo de aprendizaje semisupervisado denominado *Self-training* [11]. Este algoritmo consta de los siguientes pasos:

1. Se construye un clasificador con los ejemplos etiquetados.
2. Mediante dicho clasificador, se clasifican todos los ejemplos no etiquetados.
3. Se añaden al conjunto de ejemplos etiquetados los nuevos ejemplos clasificados que tengan un mayor índice de confianza.
4. Se repite el procedimiento hasta que sólo quede un cierto porcentaje de ejemplos no etiquetados, o se llegue al número máximo de iteraciones.

El porcentaje descrito anteriormente se calcula teniendo en cuenta el cardinal del conjunto de las instancias etiquetadas, y se suele tomar el 70 %, es decir, cuando el 70 % de las instancias no etiquetadas han sido clasificadas por el algoritmo, éste se detiene.

Este método es uno de los más usados debido a su facilidad de comprensión y de programación.

A continuación se detalla paso a paso el algoritmo:

Algoritmo *Self-training*:

Data: $Z_l = \{(X_l, Y_l)\}$; ejemplos etiquetados.
X_u; ejemplos no etiquetados.
$Z_u = \emptyset$; conjunto auxiliar.
ρ; umbral de confianza.
$iter = 0$; número de iteraciones.
$Iter$; número máximo de iteraciones.
$perc$; porcentaje de parada. $\rightarrow P = perc \cdot |X_u|$.

begin
  while $iter \leq Iter$ *and* $|Z_u| \leq P$ **do**
    Construir un clasificador C con Z_l;
    Clasificar X_u con $C; \Rightarrow Y_u$;
    Separar los elementos de Y_u con alto nivel de confianza ($> \rho$), que llamaremos Y'_u, con sus respectivos valores de X_u, que llamaremos $X'_u; \Rightarrow Z_u = Z_u \cup \{(X'_u, Y'_u)\}$;
    Modificar $Z_l = Z_l \cup \{(X'_u, Y'_u)\}; X_u = X_u \setminus X'_u; iter = iter + 1$;
  end
end

Result: $Z_l = Z_l \cup \{(X_u, Y_u^*)\}$, donde Y_u^* son las clases pronosticadas por el modelo para los valores restantes de X_u.

Veamos a continuación de forma más detallada algunos pasos del algoritmo:

Para empezar, se define $Z_u = \emptyset$ como conjunto auxiliar, que será el que incluya todos los ejemplos que no estaban etiquetados, y que tras cada iteración, sí lo están. En el momento en que su cardinal llega a cierto porcentaje del número total de instancias no etiquetadas inicial, el bucle finaliza.

Luego, en cada iteración se modifican los valores de X_u y Z_l. Este bucle parará si X_u disminuye mucho su número de elementos, ya que los que salen de X_u, lo llevamos a Z_u.

Por último, cuando se llega al test de parada, el modelo pronosticará los valores que no han tenido un alto nivel de confianza, y se incluirán en el resultado final. Es por ello que después se incluye en Z_l los valores de $\{(X_u, Y_u^*)\}$, que tras la última iteración no se encuentran todavía en el conjunto de instancias etiquetadas. Es decir, el resultado final obtenido son todos los valores de Z_l y los valores pronosticados restantes de X_u.

3.2. *Co-training*

Además de *Self-training*, el algoritmo más usado para el aprendizaje semisupervisado es el denominado *Co-training*, que puede describirse de forma global como sigue [12]:

1. Se construyen dos clasificadores con los ejemplos etiquetados.

2. Mediante dichos clasificadores, se clasifican todos los ejemplos no etiquetados.

3. Se añaden los nuevos ejemplos etiquetados que los dos clasificadores hayan clasificado de forma idéntica, o, en su defecto, los nuevos ejemplos etiquetados para los que al menos uno de los clasificadores proporcione un gran índice de confianza.

4. Se repite el procedimiento hasta que no haya ejemplos no etiquetados, o no se añada ningún nuevo ejemplo etiquetado al conjunto de datos etiquetados.

Este método es parecido al *Self-training*, con la diferencia de que en este caso hay dos clasificadores a tener en cuenta, y el umbral de confianza se suele tomar mayor (en torno al 95 %).

A continuación se describe el algoritmo conocido como *Democratic Co-learning*, que es una extensión del algoritmo *Co-training*, pero con más clasificadores. En este caso, no se exigirá que todos los clasificadores pronostiquen en la misma clase a un ejemplo (paso 3), sino que se modifica este paso, quedando como sigue:

Se añadirán los nuevos ejemplos etiquetados con la clase que los clasificadores hayan pronosticado con mayoría absoluta, y que la suma de los índices de confianza en dicha clase sea mayor a la suma de los índices de confianza en las restantes clases.

De esta forma, se evita la situación problemática de clasificar un nuevo ejemplo si dos clasificadores lo pronostican en la misma clase pero con un índice de confianza pequeño en ambos casos, cosa que podría ocurrir en el *Co-training*.

A continuación se detalla paso a paso el algoritmo:

Algoritmo *Democratic Co-learning*:

Data: $Z_l = \{(X_l, Y_l)\}$; ejemplos etiquetados.
X_u; ejemplos no etiquetados.
$Z_u = \{((0, \overset{d}{\cdots}, 0), 0)\}$; conjunto auxiliar.

begin

  while $X_u \neq \emptyset$ *and* $Z_u \neq \emptyset$ **do**

    Construir k clasificadores distintos $C_1, \ldots, C_k$ con Z_l;
    Clasificar X_u con $C_1, \ldots, C_k$; $\Rightarrow Y_u^1, \ldots, Y_u^k$;
    Separar los elementos de $Y_u^1, \ldots, Y_u^k$ que se hayan clasificado con mayoría absoluta en la misma clase, y que además la suma de los índices de confianza de dicha clase sea mayor que la suma de los de las restantes clases, que llamaremos Y'_u (que toman dicha clase como valor); con sus respectivos valores de X_u, que llamaremos X'_u; $\Rightarrow Z_u = \{(X'_u, Y'_u)\}$;
    Modificar $Z_l = Z_l \cup Z_u$; $X_u = X_u \setminus X'_u$;

  end

end

Result: $Z_l = Z_l \cup (X_u, Y_u^*)$, donde Y_u^* son los elementos resultantes de $Y_u^1, \ldots, Y_u^k$ que no tienen una mayoría absoluta de una única clase. Estos elementos son pronosticados con un regla de selección, como por ejemplo *weighted majority voting* [13].

Es importante resaltar que, en este algoritmo, el conjunto auxiliar Z_u va siendo reemplazado durante la ejecución del algoritmo, al contrario que en el caso de *Self-training*, donde se incrementa en cada iteración.

De hecho, el bucle se detiene si dicho conjunto es $\emptyset$ en alguna iteración. Si en alguna iteración Z_u es $\emptyset$, significa que ningún valor nuevo se añade al conjunto de ejemplos etiquetados, y esto es una condición de parada.

Si se cumple esta condición, las instancias no etiquetadas restantes son clasificadas por una regla de selección, como la ya mencionada *weigthed majority voting*. Esta regla consiste en clasificar cada instancia no etiquetada, usando las probabilidades de pertenencia a cada clase proporcionadas por los clasificadores, ponderando cada clasificador por un valor real $\nu_i, \forall i \in \{1, \ldots, k\}$, donde ν_i es el **peso** del i-ésimo clasificador, y son calculados en función del rendimiento de los clasificadores. Por ejemplo, una forma de calcularlos sería tomar la tasa de acierto total.

Por otro lado, la otra condición para finalizar el bucle se presenta cuando

todos los elementos no etiquetados ya han sido clasificados.

3.3. Métodos basados en grafos

Los métodos basados en grafos [14], se basan en la construcción de un grafo a partir de la base de datos dada, donde los vértices son tanto las instancias etiquetadas X_l, como las no etiquetadas X_u. Sobre este grafo, que en general será de gran tamaño, se tratará de asignar valores de la variable respuesta y a los vértices.

Para ello, se toman aristas que unan las instancias etiquetadas con las no etiquetadas. Usualmente suelen ser grafos no dirigidos. A cada arista se le asigna un peso w_{ij} que representa la similaridad entre los vértices x_i y x_j que definen dicha arista.

Si un peso w_{ij} es alto, se espera que las clases y_i e y_j sean la misma. Dichos pesos suelen calcularse de una de las siguientes formas:

- Se considera un grafo completo, donde el peso de cada arista disminuye a medida que la distancia euclídea entre los vértices que une, $\|x_i - x_j\|$, aumenta. Una definición usual suele ser la siguiente:

$$w_{ij} = \exp\left(-\frac{\|x_i - x_j\|^2}{2\sigma^2}\right),$$

  donde σ es un parámetro (*bandwidth*), que controla la velocidad de decrecimiento del peso. Por tanto, el peso es 1 si $x_i = x_j$, y 0 cuando $\|x_i - x_j\|$ tiende a infinito.

- Grafo kNN. Cada vértice se conecta con sus k vecinos más próximos según la distancia euclídea. Es importante resaltar que si x_i tiene como uno de sus k vecinos más próximos a x_j, no implica que se dé el recíproco. Una arista conecta a x_i con x_j si al menos uno de los dos tiene al otro como uno de sus k vecinos más próximos. Por tanto, un vértice puede tener más de k aristas. Si x_i y x_j están conectados, su peso será 1, en cuyo caso se dice que el grafo es no ponderado, o se calcula el peso con una fórmula como la anterior. Por último, si x_i y x_j no están conectados, $w_{ij} = 0$.

- Grafo εNN. Una arista conecta a x_i y x_j, si $\left\|x_i - x_j\right\| \leq \varepsilon$. Los pesos se calculan como en el apartado anterior. Este método es más fácil de construir que el kNN.

Por supuesto, si se conocen más detalles de una base de datos particular, se pueden crear mejores grafos para dicha base de datos.

Por otro lado, es importante resaltar que si se dibuja el grafo, la distancia entre los vértices no es relevante, sino que dos vértices son similares si están conectados por una arista.

A continuación se detalla uno de los métodos basados en grafos más usados. Sin pérdida de generalidad, consideraremos que la variable objetivo es binaria y que toma los valores $\{-1, 1\}$.

3.3.1. *Mincut*

El algoritmo *Mincut* consiste en encontrar un conjunto mínimo de aristas del grafo de forma que, al suprimirlas, los vértices etiquetados con la clase 1, y los etiquetados con la clase -1, queden completamente disconexos. A esto se le conoce como corte, que da lugar a una partición del grafo en dos conjuntos de vértices.

Para ello, si llamamos 'tamaño del corte' a la suma de los pesos de las aristas que son suprimidas, entonces buscamos minimizar el tamaño del corte.

Matemáticamente, buscamos una función $f(x) \in \{-1, 1\}$, donde $f(x_i) = y_i$ en los vértices etiquetados, y de tal modo que se minimice el tamaño del corte, es decir, el objetivo es minimizar:

$$\sum_{i,j:f(x_i)\neq f(x_j)} w_{ij}.$$

La expresión anterior proporciona el tamaño del corte, ya que si una arista es suprimida, entonces $f(x_i) \neq f(x_j)$.

Para lograr esto, se plantea el algoritmo *Mincut* como un problema de minimización, con función de perdida y regularizador apropiados.

Como buscamos que $f(x_i) = y_i$ para todos los vértices etiquetados, definimos la función de pérdida como:

$$c(x, y, f(x)) = \infty \cdot (y - f(x))^2,$$

donde definimos $\infty \cdot 0 = 0$. Esto hace que la función de pérdida valga 0 cuando $f(x_i) = y_i$, e infinito en otro caso, por lo que la pérdida será mínima cuando $f(x_i)$ coincida con la clase y_i en todos los nodos etiquetados.

Por otro lado, teniendo en cuenta que $f(x) \in \{-1, 1\}$, el tamaño del corte puede reescribirse como

$$\sum_{i,j=1}^{l+u} w_{ij}(f(x_i) - f(x_j))^2/4,$$

donde la suma ahora es en todos los vértices. Si x_i y x_j no están conectados, entonces $w_{ij} = 0$, mientras que si la arista existe y no forma parte del corte, entonces $f(x_i) - f(x_j) = 0$. Para las aristas del corte, cada sumando vale w_{ij}. Por tanto, esta función está bien definida para cada par de vértices y coincide con el tamaño del corte.

Por consiguiente, el problema de minimización del *Mincut* queda como sigue:

$$\min_{f:f(x)\in\{-1,1\}} \infty \sum_{i=1}^{l}(y_i - f(x_i))^2 + \sum_{i,j=1}^{l+u} w_{ij}(f(x_i) - f(x_j))^2,$$

donde el término $1/4$ se omite, ya que resulta un problema equivalente. Esto es un problema en números enteros, ya que f toma valores en $\{-1, 1\}$. Sin embargo, existen algoritmos que resuelven este tipo de problemas en tiempo polinomial.

Sin embargo, no existe una única solución, ya que se pueden dar casos donde vértices no etiquetados se puedan etiquetar de igual forma en la misma clase. Este inconveniente puede resolverse considerando problemas más complejos, donde además se tenga en cuenta la **confianza** de cada vértice etiquetado.

Capítulo 4

Paquete ssc de R

En este capítulo se describirá la funcionalidad del paquete `ssc` de `R`, que implementa los algoritmos anteriormente descritos [15].

El paquete `ssc` tiene como título *Semi-Supervised Classification Methods*. Dicho paquete se encuentra actualmente en la versión 2.1-0 y requiere para su ejecución una versión de `R` igual o superior a la 3.2.3.

A continuación, detallaremos las funciones del paquete `ssc` relacionadas con los algoritmos descritos en el capítulo anterior.

4.1. selfTraining

La función `selfTraining` es una función basada en el algoritmo *Self-training*.

El propio autor lo describe como un método simple y efectivo, habitualmente usado en bases de datos propias del aprendizaje semisupervisado.

Uso:

```
selfTraining(x, y, x.inst = TRUE, learner, learner.pars = NULL,
pred = "predict", pred.pars = NULL, max.iter = 50, perc.full = 0.7,
thr.conf = 0.5)
```

Argumentos:

`x`	Objeto que puede ser representado como matriz. Este objeto tiene dos posibles interpretaciones en función del valor de `x.inst`: una matriz con los ejemplos de entrada donde cada fila corresponde a un ejemplo o una matriz simétrica $n \times n$ precalculada donde sus celdas representan cierta comparación (distancia o núcleo) entre los ejemplos de entrenamiento.
`y`	Vector que contiene las etiquetas de los ejemplos de entrada. En este vector las instancias no etiquetadas se especifican con el valor `NA`.
`x.inst`	Valor booleano que indica si la matriz `x` contiene las instancias o no. Por defecto es `TRUE`.
`learner`	Función o cadena que indica el nombre de la función para entrenar, usando una matriz de instancias (o una matriz de distancias) y sus correspondientes clases.
`learner.pars`	Lista con los parámetros de la función `learner`, si fuera necesario. Por defecto es `NULL`.
`pred`	Función o cadena que indica la función que pronostica las probabilidades de cada clase, usando el clasificador entrenado en `learner`. Por defecto es `"predict"`.
`pred.pars`	Lista con los parámetros adicionales para la función `pred`, si fuera necesario. Por defecto es `NULL`.
`max.iter`	Número máximo de iteraciones para la ejecución del algoritmo *Self-training*. Por defecto es 50.
`perc.full`	Valor entre 0 y 1. Si la fracción de los nuevos ejemplos etiquetados alcanza este valor, el algoritmo *Self-training* se para. Por defecto es 0.7.
`thr.conf`	Valor entre 0 y 1 que indica el límite de confianza. En cada iteración, sólo los nuevos ejemplos etiquetados cuyo valor de confianza supere este valor (`thr.conf`) son añadidos al conjunto de entrenamiento.

Valores:

Una vez concluida la ejecución de la función `selfTraining`, se obtiene el siguiente conjunto de valores de salida:

`model`	Clasificador final que permite pronosticar nuevos valores.
`instances.index`	Índices de los ejemplos de entrenamiento usados en `model`. Estos índices incluyen los ejemplos iniciales y las instancias incluidas durante el proceso. Estos índices corresponden al objeto `x`.
`classes`	Valores de la variable respuesta `y`.
`pred`	Función especificada en el argumento `pred`.
`pred.pars`	Lista de parámetros indicada en el argumento `pred.pars`.

4.2. democratic

La función `democratic` está basada en el algoritmo *Democratic Co-learning*.

Como hemos mencionado en el capítulo anterior, *Democratic Co-learning* es un algoritmo basado en técnicas de *Co-training*, que hace uso de k clasificadores.

Uso:

```
democratic(x, y, x.inst = TRUE, learners, learners.pars = NULL,
preds = rep("predict", length(learners)), preds.pars = NULL)
```

Argumentos:

`x`	Objeto que puede ser representado como matriz. Este objeto tiene dos posibles interpretaciones en función del valor de `x.inst`: una matriz con los ejemplos de entrada donde cada fila corresponde a un ejemplo o una matriz simétrica $n \times n$ precalculada donde sus celdas representan cierta comparación (distancia o núcleo) entre los ejemplos de entrenamiento.
`y`	Vector que contiene las etiquetas de los ejemplos de entrenamiento. En este vector las instancias no etiquetadas se especifican con el valor `NA`.
`x.inst`	Valor booleano que indica si la matriz `x` contiene las instancias o no. Por defecto es `TRUE`.
`learners`	Lista de funciones o cadenas que indican el nombre de las funciones adicionales que se utilizarán como clasificadores supervisados.
`learners.pars`	Lista con el conjunto de los parámetros de cada función de `learners`, si fuera necesario. Por defecto es `NULL`.
`preds`	Lista de funciones o cadenas que indican la función que pronostica las probabilidades de pertenencia a cada clase, usando los clasificadores entrenados en `learners`. Por defecto es `"predict"` para cada una de las funciones de `learners`.
`preds.pars`	Lista con el conjunto de parámetros adicionales para cada función de `preds`, si fuera necesario. Por defecto es `NULL`.

Valores:

La función `democratic` devuelve tras su ejecución una lista que contiene los siguientes elementos:

`W`	Vector que contiene los votos ponderados asignados a los distintos clasificadores.
`model`	Lista con los k clasificadores finales que permite pronosticar nuevos valores.
`model.index`	Lista de k vectores que contienen los índices de las instancias de entrenamiento usadas por cada clasificador. Estos índices son relativos al argumento `y`.
`instances.index`	Índices de todos los ejemplos de entrenamiento usados para entrenar los k modelos. Estos índices incluyen los ejemplos iniciales y las instancias incluidas durante el proceso. Estos índices corresponden al argumento `y`.
`model.index.map`	Lista de k vectores con la misma información que `model.index`, pero cada índice corresponde al vector `instances.index`.
`classes`	Valores de la variable respuesta `y`.
`preds`	Funciones especificadas en el argumento `preds`.
`preds.pars`	Conjunto de listas proporcionado en el argumento `preds.pars`.
`x.inst`	Valor dado en el argumento `x.inst`.

Capítulo 5

Aplicación

Por último, en este capítulo vamos a aplicar lo desarrollado en esta memoria a una base de datos semisupervisada.

La base de datos elegida ha sido la base de datos ***magic***, disponible en https://sci2s.ugr.es/keel/dataset_smja.php?cod=1370.

Dicha base de datos consta del registro de partículas gamma de alta energía, detectado en el telescopio gamma Cherenkov atmosférico terrestre, usando técnicas de imagen.

El objetivo es discriminar estadísticamente imágenes generadas por los gammas primarios (señal, con clase g), de las imágenes de lluvias hadrónicas iniciadas por rayos cósmicos en la atmósfera superior (fondo, con clase h).

Por otro lado, la base de datos consta de los siguientes atributos:

- FLength: Variable continua. Mayor eje de la elipse, en milímetros.
- FWidth: Variable continua. Menor eje de la elipse, en milímetros.
- FSize: Variable continua. Logaritmo en base 10 de la suma de los contenidos de todos los píxeles, en pulgadas.
- FConc_two: Variable continua. Relación de la suma entre los dos píxeles más altos sobre FSize.
- FConc_one: Variable continua. Relación del píxel mayor sobre FSize.
- FAsym: Variable continua. Distancia entre el píxel mayor y el centro, proyectado en el eje mayor, en milímetros.

- FMLong: Variable continua. Raíz cúbica del tercer momento a lo largo del eje mayor.
- FMTrans: Variable continua. Raíz cúbica del tercer momento a lo largo del eje menor.
- FAlpha: Variable continua. Ángulo del eje mayor con el vector de origen.
- FDist: Variable continua. Distancia del origen al centro de la elipse.
- Class: Variable categórica. Es la variable objetivo, y toma como clase negativa g y como positiva h.

Para mostrar finalmente la tabla con los resultados para los métodos *Self-training* y *Democratic Co-learning*, hemos usado para el primero, el clasificador `naive_bayes` del paquete `naivebayes`, y para el segundo, el clasificador anterior y el clasificador `knn3` del paquete `caret`.

Antes de dar paso al código, cabe destacar que si tenemos variables de tipo factor como una de las variables predictoras, es necesario, debido a la naturaleza del paquete `ssc` que sólo admite atributos numéricos, modificar estas variables y transformarlas en variables *dummy*.

Una de las posibles codificaciones consiste en crear tantas variables binarias como número de niveles tenga la variable de tipo factor, donde cada variable binaria se asocia a un único nivel del factor. Así, cada una de estas nuevas variables tomará los valores 0 y 1; el valor 1 lo tomará cuando se presente un determinado nivel del factor, y 0 en otro caso. También existe otra posible codificación similar a esta, donde se crean tantas variables como número de niveles tenga la variable de tipo factor al que se le aplica dicha codificación, menos uno.

Esto permite transformar una variable de tipo factor en varios atributos numérica, ya que muchos procedimientos sólo permiten trabajar con valores numéricos. Además, dicha transformación debe realizarse para cada variable de tipo factor de la base de datos original.

Aunque nuestra base de datos no dispone de variables de tipo factor, cabe destacar que esta transformación puede hacerse por medio de la función `dummy.data.frame` del paquete `dummies`.

A continuación se muestra el código fuente en lenguaje `R` [16], donde primero vamos a separar las variables predictoras de la variable objetivo para transformar las base de datos en una base de datos semisupervisada, y a definir los atributos de las funciones `selfTraining` y `democratic`.

Luego, vamos a aplicar cada una de las técnicas de validación vistas en el capítulo 2. A cada una de ellas y a cada uno de los métodos mencionados, se les calcularán las medidas de validación mencionadas en esta memoria.

Sin embargo, en el caso del AUC, debido a que las funciones no devuelven las probabilidades de pertenencia a cada clase, sino que devuelven las predicciones, no es posible calcularlas de forma directa. Por ello, en el caso de *Self-training*, se crea el modelo resultante y se calcula a dicho modelo la curva ROC y el AUC con ayuda de la librería `pROC`. Debido a esto, no es hacer lo mismo para el caso de *Democratic Co-learning*, ya que usa más de un clasificador.

Por tanto, además de las medidas de validación mencionadas anteriormente, en el caso de *Self-training* se calcula además el AUC y se muestra además una gráfica de la curva ROC correspondiente a la muestra test que indique el título de dicha gráfica.

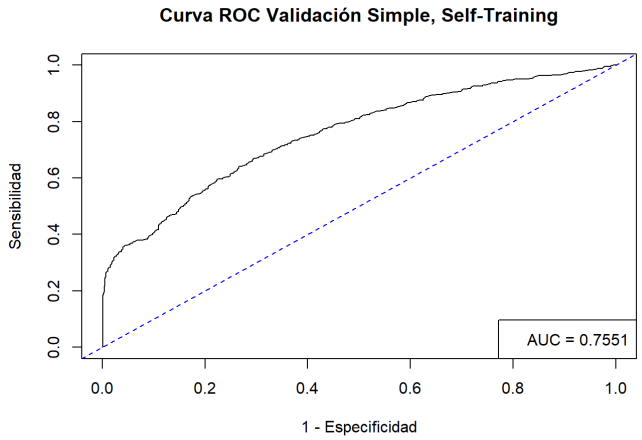
Por último, se mostrará una tabla con los resultados para verificar la efectividad en cada uno de los casos.

```
# Librerías usadas.
library(ssc)
library(caret)
library(naivebayes)
library(ROCR)

# Cargamos los datos, y nombramos sus atributos.
datos <- read.table("magic.dat", skip = 15, sep = ",")
colnames(datos) <- c("FLength", "FWidth", "FSize",
   "FConc_two", "FConc_one", "FAsym", "FMLong",
   "FMTrans", "FAlpha", "FDist","Class")

# Separamos las variables predictoras (y las
   estandarizamos) de la variable respuesta.
x <- datos[, -11]
x <- scale(x)
y <- datos[, 11]

# Ponemos una semilla para poder reproducir el código.
set.seed(11)

# Transformamos la base de datos en una base de datos
```

```
   semisupervisada, con en torno al 67.5% de instancias
   no etiquetadas.
aleatorio <- sort(sample(19020, size = 12838))
y[aleatorio] <- NA

# Definimos algunos valores para cálculos posteriores.
l <- length(which(!is.na(y)))
c <- length(levels(y))

# Para la elección estratificada, vamos a separar los
   niveles de y.
ceros <- which(y == "g")
unos <- which(y == "h")

# Para el Bootstrap, es necesario también lo siguiente.
unir <- c(ceros, unos)
nas <- which(is.na(y))


### Atributos del modelo selfTraining.

# Usamos como clasificador la función naive_bayes de la
   librería naivebayes.
learner_s <- function(x, y) naivebayes::naive_bayes(x,
   y)
learner_pars_s <- NULL
predict_s <- predict
predict_pars_s <- list(type = "prob")


### Atributos del modelo democratic.

# Usamos como clasificadores la función anterior, y la
   función knn3 de la librería caret.
learner_d_uno <- function(x, y)
   naivebayes::naive_bayes(x, y)
learner_pars_d_uno <- NULL
predict_d_uno <- predict
predict_pars_d_uno <- list(type = "prob")
learner_d_dos <- knn3
```

```
learner_pars_d_dos <- list(k = 3)
predict_d_dos <- predict
predict_pars_d_dos <- NULL
learner_d <- list(learner_d_uno, learner_d_dos)
learner_pars_d <- list(learner_pars_d_uno,
   learner_pars_d_dos)
predict_d <- list(predict_d_uno, predict_d_dos)
predict_pars_d <- list(predict_pars_d_uno,
   predict_pars_d_dos)


### Validación simple.

# Selección estratificada del conjunto test.
elec_ceros <- sample(ceros, size = 1330)
elec_unos <- sample(unos, size = 731)
xtrain <- x[-c(elec_ceros, elec_unos), ]
ytrain <- y[-c(elec_ceros, elec_unos)]
xtest <- x[c(elec_ceros, elec_unos), ]
ytest <- y[c(elec_ceros, elec_unos)]
m <- length(ytest)

# Creamos el modelo Self-training, y calculamos las
   predicciones y las medidas de efectividad.
modelo_s <- selfTraining(x = xtrain, y = ytrain,
   learner = learner_s, learner.pars = learner_pars_s,
   pred = predict_s, pred.pars = predict_pars_s)
pred_s <- predict(modelo_s, xtest)
tabla_s <- table(pred_s, ytest)
acierto_global_VS_s <- (tabla_s[1, 1] + tabla_s[2, 2])
   / m
error_global_VS_s <- (tabla_s[1, 2] + tabla_s[2, 1]) / m
Kappa_VS_s <- (tabla_s[1, 1] + tabla_s[2, 2] - m / c) /
   (m - m / c)
especificidad_VS_s <- tabla_s[1, 1] / (tabla_s[1, 1] +
   tabla_s[1, 2])
sensibilidad_VS_s <- tabla_s[2, 2] / (tabla_s[2, 1] +
   tabla_s[2, 2])
FP_VS_s <- tabla_s[1, 2] / (tabla_s[1, 1] + tabla_s[1,
   2])
```

```
FN_VS_s <- tabla_s[2, 1] / (tabla_s[2, 1] + tabla_s[2,
   2])
precision_VS_s <- tabla_s[2, 2] / (tabla_s[2, 2] +
   tabla_s[1, 2])
VPN_VS_s <- tabla_s[1, 1] / (tabla_s[1, 1] + tabla_s[2,
   1])
medida_F_VS_s <- 2 * tabla_s[1, 1] / (2 * tabla_s[1, 1]
   + tabla_s[1, 2] + tabla_s[2, 1])

# Para calcular el AUC por separado.
fx <- modelo_s$model$data$x
fy <- modelo_s$model$data$y
modelo_roc <- naive_bayes(fx, fy)
pred_s <- predict(modelo_roc, xtest, type = "prob")[, 2]
roc_s <- prediction(pred_s, ytest)

# Mostramos la curva ROC y calculamos el AUC.
plot(performance(roc_s, "tpr", "fpr"), main = "Curva
   ROC Validación Simple, Self-Training", xlab = "1 -
   Especificidad", ylab = "Sensibilidad")
abline(a = 0, b = 1, col = "blue", lty = 2)
auc_VS_s <- as.numeric(performance(roc_s,
   "auc")@y.values)
legend("bottomright", legend = paste("AUC =",
   round(auc_VS_s, 4)))
```



```
# Guardamos todas las medidas de validación.
datos_VS_s <- c(acierto_global_VS_s, error_global_VS_s,
   Kappa_VS_s, especificidad_VS_s, sensibilidad_VS_s,
   FP_VS_s, FN_VS_s, precision_VS_s, VPN_VS_s,
   medida_F_VS_s, auc_VS_s)

# Creamos el modelo Democratic Co-learning, y
   calculamos las predicciones y las medidas de
   efectividad.
modelo_d <- democratic(x = xtrain, y = ytrain, learners
   = learner_d, learners.pars = learner_pars_d, preds =
   predict_d, preds.pars = predict_pars_d)
pred_d <- predict(modelo_d, xtest)
tabla_d <- table(pred_d, ytest)
acierto_global_VS_d <- (tabla_d[1, 1] + tabla_d[2, 2])
   / m
error_global_VS_d <- (tabla_d[1, 2] + tabla_d[2, 1]) / m
Kappa_VS_d <- (tabla_d[1, 1] + tabla_d[2, 2] - m / c) /
   (m - m / c)
especificidad_VS_d <- tabla_d[1, 1] / (tabla_d[1, 1] +
   tabla_d[1, 2])
sensibilidad_VS_d <- tabla_d[2, 2] / (tabla_d[2, 1] +
   tabla_d[2, 2])
FP_VS_d <- tabla_d[1, 2] / (tabla_d[1, 1] + tabla_d[1,
   2])
FN_VS_d <- tabla_d[2, 1] / (tabla_d[2, 1] + tabla_d[2,
   2])
precision_VS_d <- tabla_d[2, 2] / (tabla_d[2, 2] +
   tabla_d[1, 2])
VPN_VS_d <- tabla_d[1, 1] / (tabla_d[1, 1] + tabla_d[2,
   1])
medida_F_VS_d <- 2 * tabla_d[1, 1] / (2 * tabla_d[1, 1]
   + tabla_d[1, 2] + tabla_d[2, 1])

# Guardamos todas las medidas de validación.
datos_VS_d <- c(acierto_global_VS_d, error_global_VS_d,
   Kappa_VS_d, especificidad_VS_d, sensibilidad_VS_d,
   FP_VS_d, FN_VS_d, precision_VS_d, VPN_VS_d,
   medida_F_VS_d)
```

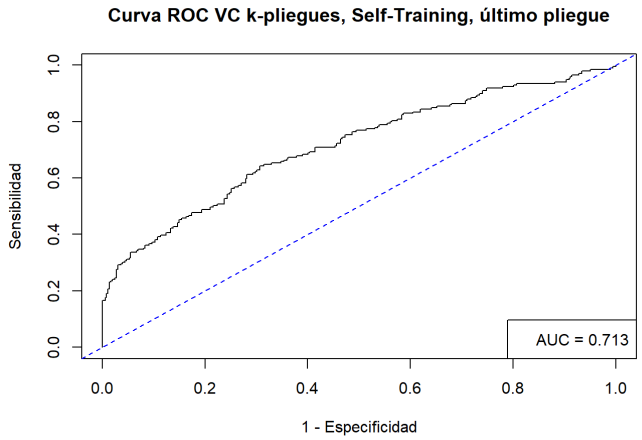
```
### Validación Cruzada k-pliegues.

# Usamos 11 pliegues, ya que es múltiplo del número de
   instancias etiquetadas.
k <- 11

# Sin embargo, para la selección estratificada, el
   número de instancias de cada clase no es múltiplo de
   11. Para equilibrar esto, separamos 4 pligues con 200
   instancias cuya variable objetivo valga "h", y 7
   pliegues con 199 instancias cuya variable objetivo
   vale "h".
suerte <- c(rep(200, 4), rep(199, 7))

# Para calcular los pliegues de forma aleatoria.
aleat_ceros <- sample(ceros)
aleat_unos <- sample(unos)
suma_ceros <- 0
suma_unos <- 0

# Definimos un vector por cada medida de validación de
   longitud 11.
acierto_global_s <- numeric(k)
error_global_s <- numeric(k)
Kappa_s <- numeric(k)
especificidad_s <- numeric(k)
sensibilidad_s <- numeric(k)
FP_s <- numeric(k)
FN_s <- numeric(k)
precision_s <- numeric(k)
VPN_s <- numeric(k)
medida_F_s <- numeric(k)
auc_s <- numeric(k)
acierto_global_d <- numeric(k)
error_global_d <- numeric(k)
Kappa_d <- numeric(k)
especificidad_d <- numeric(k)
sensibilidad_d <- numeric(k)
```

```
FP_d <- numeric(k)
FN_d <- numeric(k)
precision_d <- numeric(k)
VPN_d <- numeric(k)
medida_F_d <- numeric(k)

# Creamos un bucle de 11 iteraciones.
for (i in 1:k){

  # De forma aleatoria, o seleccionamos 199 instancias
     de clase "h", o 200.
  cuanto_unos <- sample(suerte, size = 1)

  # Como cada pliegue tiene 562 elementos, el número de
     instancias de la clase "g" es el siguiente.
  cuanto_ceros <- 562 - cuanto_unos

  # Aquí se seleccionan los elementos de cada pliegue.
     El primer pliegue tendrá los primeros elementos
     del conjunto de instancias desordenado de forma
     aleatoria anteriormente. En la siguiente
     iteración, tendrá los siguientes elementos (de ahí
     los valores de suma_ceros y suma_unos).
  valores_test <- sort(c(aleat_ceros[(suma_ceros +
     1):(suma_ceros + cuanto_ceros)],
     aleat_unos[(suma_unos + 1):(suma_unos +
     cuanto_unos)]))

  # Selección de conjunto de entrenamiento y test en
     cada pliegue.
  xtrain <- x[-valores_test, ]
  ytrain <- y[-valores_test]
  xtest <- x[valores_test, ]
  ytest <- y[valores_test]
  m <- length(ytest)

  # Creamos el modelo Self-training, y calculamos las
     predicciones y las medidas de efectividad.
  modelo_s <- selfTraining(x = xtrain, y = ytrain,
     learner = learner_s, learner.pars =
```

```
    learner_pars_s, pred = predict_s, pred.pars =
    predict_pars_s)
pred_s <- predict(modelo_s, xtest)
tabla_s <- table(pred_s, ytest)
acierto_global_s[i] <- (tabla_s[1, 1] + tabla_s[2,
    2]) / m
error_global_s[i] <- (tabla_s[1, 2] + tabla_s[2, 1])
    / m
Kappa_s[i] <- (tabla_s[1, 1] + tabla_s[2, 2] - m / c)
    / (m - m / c)
especificidad_s[i] <- tabla_s[1, 1] / (tabla_s[1, 1]
    + tabla_s[1, 2])
sensibilidad_s[i] <- tabla_s[2, 2] / (tabla_s[2, 1] +
    tabla_s[2, 2])
FP_s[i] <- tabla_s[1, 2] / (tabla_s[1, 1] +
    tabla_s[1, 2])
FN_s[i] <- tabla_s[2, 1] / (tabla_s[2, 1] +
    tabla_s[2, 2])
precision_s[i] <- tabla_s[2, 2] / (tabla_s[2, 2] +
    tabla_s[1, 2])
VPN_s[i] <- tabla_s[1, 1] / (tabla_s[1, 1] +
    tabla_s[2, 1])
medida_F_s[i] <- 2 * tabla_s[1, 1] / (2 * tabla_s[1,
    1] + tabla_s[1, 2] + tabla_s[2, 1])

# Para calcular el AUC por separado.
fx <- modelo_s$model$data$x
fy <- modelo_s$model$data$y
modelo_roc <- naive_bayes(fx, fy)
pred_s <- predict(modelo_roc, xtest, type = "prob")[,
    2]
roc_s <- prediction(pred_s, ytest)
auc_s[i] <- as.numeric(performance(roc_s,
    "auc")@y.values)

# Creamos el modelo Democratic Co-learning, y
    calculamos las predicciones y las medidas de
    efectividad.
modelo_d <- democratic(x = xtrain, y = ytrain,
    learners = learner_d, learners.pars =
```

```
    learner_pars_d, preds = predict_d, preds.pars =
    predict_pars_d)
  pred_d <- predict(modelo_d, xtest)
  tabla_d <- table(pred_d, ytest)
  acierto_global_d[i] <- (tabla_d[1, 1] + tabla_d[2,
    2]) / m
  error_global_d[i] <- (tabla_d[1, 2] + tabla_d[2, 1])
    / m
  Kappa_d[i] <- (tabla_d[1, 1] + tabla_d[2, 2] - m / c)
    / (m - m / c)
  especificidad_d[i] <- tabla_d[1, 1] / (tabla_d[1, 1]
    + tabla_d[1, 2])
  sensibilidad_d[i] <- tabla_d[2, 2] / (tabla_d[2, 1] +
    tabla_d[2, 2])
  FP_d[i] <- tabla_d[1, 2] / (tabla_d[1, 1] +
    tabla_d[1, 2])
  FN_d[i] <- tabla_d[2, 1] / (tabla_d[2, 1] +
    tabla_d[2, 2])
  precision_d[i] <- tabla_d[2, 2] / (tabla_d[2, 2] +
    tabla_d[1, 2])
  VPN_d[i] <- tabla_d[1, 1] / (tabla_d[1, 1] +
    tabla_d[2, 1])
  medida_F_d[i] <- 2 * tabla_d[1, 1] / (2 * tabla_d[1,
    1] + tabla_d[1, 2] + tabla_d[2, 1])

  # En cada iteración, quitamos o 199 o 200.
  suerte <- suerte[-cuanto_ceros]

  # Valores a partir de los cuales comenzar a
    seleccionar elementos del conjunto de instancias
    etiquetadas ordenador de forma aleatoria.
  suma_ceros <- suma_ceros + cuanto_ceros
  suma_unos <- suma_unos + cuanto_unos

}

# Mostramos la curva ROC relativa al último pliegue
   calculado.
plot(performance(roc_s, "tpr", "fpr"), main = "Curva
   ROC VC k-pliegues, Self-Training, último pliegue",
```

```
   xlab = "1 - Especificidad", ylab = "Sensibilidad")
abline(a = 0, b = 1, col = "blue", lty = 2)
legend("bottomright", legend = paste("AUC =",
   round(auc_s[i], 4)))
```



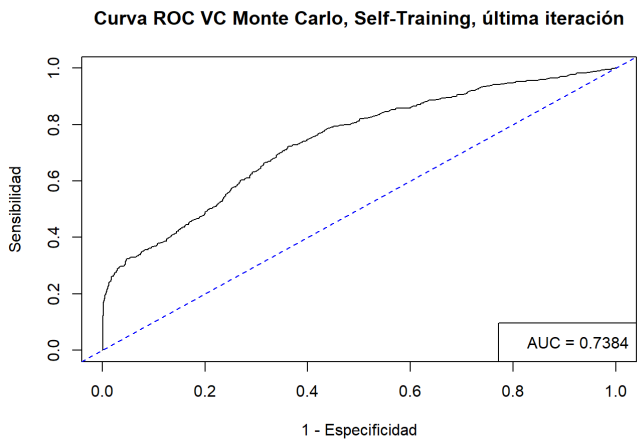

```
# Calculamos las medidas de efectividad, que serán la
   media de las correspondientes de cada pliegue.
acierto_global_VC_K_s <- mean(acierto_global_s)
error_global_VC_K_s <- mean(error_global_s)
Kappa_VC_K_s <- mean(Kappa_s)
especificidad_VC_K_s <- mean(especificidad_s)
sensibilidad_VC_K_s <- mean(sensibilidad_s)
FP_VC_K_s <- mean(FP_s)
FN_VC_K_s <- mean(FN_s)
precision_VC_K_s <- mean(precision_s)
VPN_VC_K_s <- mean(VPN_s)
medida_F_VC_K_s <- mean(medida_F_s)
auc_VC_K_s <- mean(auc_s)

# Guardamos todas las medidas de validación.
datos_VC_K_s <- c(acierto_global_VC_K_s,
   error_global_VC_K_s, Kappa_VC_K_s,
   especificidad_VC_K_s, sensibilidad_VC_K_s,
   FP_VC_K_s, FN_VC_K_s, precision_VC_K_s, VPN_VC_K_s,
   medida_F_VC_K_s, auc_VC_K_s)
```

```
# Calculamos las medidas de efectividad, que serán la
   media de las correspondientes de cada pliegue.
acierto_global_VC_K_d <- mean(acierto_global_d)
error_global_VC_K_d <- mean(error_global_d)
Kappa_VC_K_d <- mean(Kappa_d)
especificidad_VC_K_d <- mean(especificidad_d)
sensibilidad_VC_K_d <- mean(sensibilidad_d)
FP_VC_K_d <- mean(FP_d)
FN_VC_K_d <- mean(FN_d)
precision_VC_K_d <- mean(precision_d)
VPN_VC_K_d <- mean(VPN_d)
medida_F_VC_K_d <- mean(medida_F_d)

# Guardamos todas las medidas de validación.
datos_VC_K_d <- c(acierto_global_VC_K_d,
   error_global_VC_K_d, Kappa_VC_K_d,
   especificidad_VC_K_d, sensibilidad_VC_K_d,
   FP_VC_K_d, FN_VC_K_d, precision_VC_K_d, VPN_VC_K_d,
   medida_F_VC_K_d)


### Validación cruzada Monte Carlo.

# Número de iteraciones.
B <- 200

# Definimos un vector por cada medida de validación de
   longitud 200.
acierto_global_s <- numeric(B)
error_global_s <- numeric(B)
Kappa_s <- numeric(B)
especificidad_s <- numeric(B)
sensibilidad_s <- numeric(B)
FP_s <- numeric(B)
FN_s <- numeric(B)
precision_s <- numeric(B)
VPN_s <- numeric(B)
medida_F_s <- numeric(B)
auc_s <- numeric(B)
```

```
acierto_global_d <- numeric(B)
error_global_d <- numeric(B)
Kappa_d <- numeric(B)
especificidad_d <- numeric(B)
sensibilidad_d <- numeric(B)
FP_d <- numeric(B)
FN_d <- numeric(B)
precision_d <- numeric(B)
VPN_d <- numeric(B)
medida_F_d <- numeric(B)

# Creamos un bucle de 200 iteraciones.
for (i in 1:B){

  # En cada iteración, realizamos la validación simple.

  # Selección estratificada del conjunto test.
  elec_ceros <- sample(ceros, size = 1330)
  elec_unos <- sample(unos, size = 731)
  xtrain <- x[-c(elec_ceros, elec_unos), ]
  ytrain <- y[-c(elec_ceros, elec_unos)]
  xtest <- x[c(elec_ceros, elec_unos), ]
  ytest <- y[c(elec_ceros, elec_unos)]
  m <- length(ytest)

  # Creamos el modelo Self-trainig, y calculamos las
     predicciones y las medidas de efectividad.
  modelo_s <- selfTraining(x = xtrain, y = ytrain,
     learner = learner_s, learner.pars =
     learner_pars_s, pred = predict_s, pred.pars =
     predict_pars_s)
  pred_s <- predict(modelo_s, xtest)
  tabla_s <- table(pred_s, ytest)
  acierto_global_s[i] <- (tabla_s[1, 1] + tabla_s[2,
     2]) / m
  error_global_s[i] <- (tabla_s[1, 2] + tabla_s[2, 1])
     / m
  Kappa_s[i] <- (tabla_s[1, 1] + tabla_s[2, 2] - m / c)
     / (m - m / c)
  especificidad_s[i] <- tabla_s[1, 1] / (tabla_s[1, 1]
```

```
   + tabla_s[1, 2])
sensibilidad_s[i] <- tabla_s[2, 2] / (tabla_s[2, 1] +
   tabla_s[2, 2])
FP_s[i] <- tabla_s[1, 2] / (tabla_s[1, 1] +
   tabla_s[1, 2])
FN_s[i] <- tabla_s[2, 1] / (tabla_s[2, 1] +
   tabla_s[2, 2])
precision_s[i] <- tabla_s[2, 2] / (tabla_s[2, 2] +
   tabla_s[1, 2])
VPN_s[i] <- tabla_s[1, 1] / (tabla_s[1, 1] +
   tabla_s[2, 1])
medida_F_s[i] <- 2 * tabla_s[1, 1] / (2 * tabla_s[1,
   1] + tabla_s[1, 2] + tabla_s[2, 1])

# Para calcular el AUC por separado.
fx <- modelo_s$model$data$x
fy <- modelo_s$model$data$y
modelo_roc <- naive_bayes(fx, fy)
pred_s <- predict(modelo_roc, xtest, type = "prob")[,
   2]
roc_s <- prediction(pred_s, ytest)
auc_s[i] <- as.numeric(performance(roc_s,
   "auc")@y.values)

# Creamos el modelo Democratic Co-learning, y
   calculamos las predicciones y las medidas de
   efectividad.
modelo_d <- democratic(x = xtrain, y = ytrain,
   learners = learner_d, learners.pars =
   learner_pars_d, preds = predict_d, preds.pars =
   predict_pars_d)
pred_d <- predict(modelo_d, xtest)
tabla_d <- table(pred_d, ytest)
acierto_global_d[i] <- (tabla_d[1, 1] + tabla_d[2,
   2]) / m
error_global_d[i] <- (tabla_d[1, 2] + tabla_d[2, 1])
   / m
Kappa_d[i] <- (tabla_d[1, 1] + tabla_d[2, 2] - m / c)
   / (m - m / c)
especificidad_d[i] <- tabla_d[1, 1] / (tabla_d[1, 1]
```

```
    + tabla_d[1, 2])
  sensibilidad_d[i] <- tabla_d[2, 2] / (tabla_d[2, 1] +
    tabla_d[2, 2])
  FP_d[i] <- tabla_d[1, 2] / (tabla_d[1, 1] +
    tabla_d[1, 2])
  FN_d[i] <- tabla_d[2, 1] / (tabla_d[2, 1] +
    tabla_d[2, 2])
  precision_d[i] <- tabla_d[2, 2] / (tabla_d[2, 2] +
    tabla_d[1, 2])
  VPN_d[i] <- tabla_d[1, 1] / (tabla_d[1, 1] +
    tabla_d[2, 1])
  medida_F_d[i] <- 2 * tabla_d[1, 1] / (2 * tabla_d[1,
    1] + tabla_d[1, 2] + tabla_d[2, 1])

}

# Mostramos la curva ROC relativa a la última iteración
  calculada.
plot(performance(roc_s, "tpr", "fpr"), main = "Curva
  ROC VC Monte Carlo, Self-Training, última
  iteración", xlab = "1 - Especificidad", ylab =
  "Sensibilidad")
abline(a = 0, b = 1, col = "blue", lty = 2)
legend("bottomright", legend = paste("AUC =",
  round(auc_s[i], 4)))
```



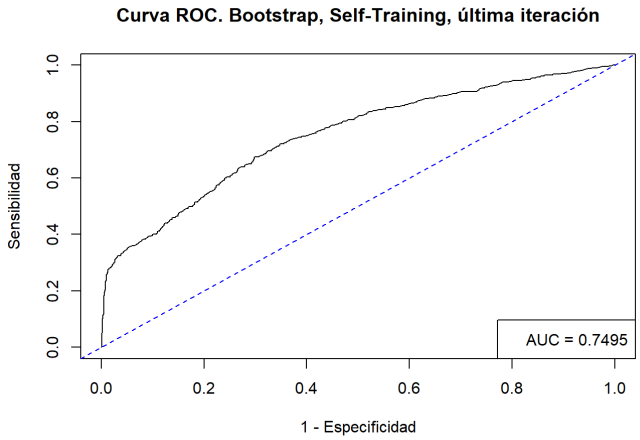
```
# Calculamos las medidad de efectividad, que serán la
   media de las correspondientes de cada iteración.
acierto_global_VC_MC_s <- mean(acierto_global_s)
error_global_VC_MC_s <- mean(error_global_s)
Kappa_VC_MC_s <- mean(Kappa_s)
especificidad_VC_MC_s <- mean(especificidad_s)
sensibilidad_VC_MC_s <- mean(sensibilidad_s)
FP_VC_MC_s <- mean(FP_s)
FN_VC_MC_s <- mean(FN_s)
precision_VC_MC_s <- mean(precision_s)
VPN_VC_MC_s <- mean(VPN_s)
medida_F_VC_MC_s <- mean(medida_F_s)
auc_VC_MC_s <- mean(auc_s)

# Guardamos todas las medidas de validación.
datos_VC_MC_s <- c(acierto_global_VC_MC_s,
   error_global_VC_MC_s, Kappa_VC_MC_s,
   especificidad_VC_MC_s, sensibilidad_VC_MC_s,
   FP_VC_MC_s, FN_VC_MC_s, precision_VC_MC_s,
   VPN_VC_MC_s, medida_F_VC_MC_s, auc_VC_MC_s)

# Calculamos las medidas de efectividad, que serán la
   media de las correspondientes de cada iteración.
acierto_global_VC_MC_d <- mean(acierto_global_d)
error_global_VC_MC_d <- mean(error_global_d)
Kappa_VC_MC_d <- mean(Kappa_d)
especificidad_VC_MC_d <- mean(especificidad_d)
sensibilidad_VC_MC_d <- mean(sensibilidad_d)
FP_VC_MC_d <- mean(FP_d)
FN_VC_MC_d <- mean(FN_d)
precision_VC_MC_d <- mean(precision_d)
VPN_VC_MC_d <- mean(VPN_d)
medida_F_VC_MC_d <- mean(medida_F_d)

# Guardamos todas las medidas de validación.
datos_VC_MC_d <- c(acierto_global_VC_MC_d,
   error_global_VC_MC_d, Kappa_VC_MC_d,
   especificidad_VC_MC_d, sensibilidad_VC_MC_d,
   FP_VC_MC_d, FN_VC_MC_d, precision_VC_MC_d,
```

```
   VPN_VC_MC_d, medida_F_VC_MC_d)


### Bootstrap.

# Número de iteraciones.
B <- 200

# Definimos un vector por cada medida de validación de
   longitud 200.
acierto_global_s <- numeric(B)
error_global_s <- numeric(B)
Kappa_s <- numeric(B)
especificidad_s <- numeric(B)
sensibilidad_s <- numeric(B)
FP_s <- numeric(B)
FN_s <- numeric(B)
precision_s <- numeric(B)
VPN_s <- numeric(B)
medida_F_s <- numeric(B)
auc_s <- numeric(B)
acierto_global_d <- numeric(B)
error_global_d <- numeric(B)
Kappa_d <- numeric(B)
especificidad_d <- numeric(B)
sensibilidad_d <- numeric(B)
FP_d <- numeric(B)
FN_d <- numeric(B)
precision_d <- numeric(B)
VPN_d <- numeric(B)
medida_F_d <- numeric(B)

# Creamos un bucle de 200 iteraciones.
for (i in 1:B){

  # Selección aleatoria del conjunto de instancias
     etiquetadas.
  muestra <- sample(unir, rep = T)

  # Diferencias del conjunto de instancias etiquetadas
```

```
   y el vector anterior.
distinto <- setdiff(unir, muestra)

# Selección del conjunto de entrenamiento y test en
   cada iteración. El conjunto de entrenamiento (al
   que le aplicamos el algoritmo), serán las
   instancias seleccionadas de forma aleatoria con
   repetición y las instancias no etiquetadas, y el
   conjunto test serán las instancias etiquetadas
   restantes.
xtrain <- x[c(muestra, nas), ]
ytrain <- y[c(muestra, nas)]
xtest <- x[distinto, ]
ytest <- y[distinto]
m <- length(ytest)

# Creamos el modelo Self-training, y calculamos las
   predicciones y las medidas de efectividad.
modelo_s <- selfTraining(x = xtrain, y = ytrain,
   learner = learner_s, learner.pars =
   learner_pars_s, pred = predict_s, pred.pars =
   predict_pars_s)
pred_s <- predict(modelo_s, xtest)
tabla_s <- table(pred_s, ytest)
acierto_global_s[i] <- (tabla_s[1, 1] + tabla_s[2,
   2]) / m
error_global_s[i] <- (tabla_s[1, 2] + tabla_s[2, 1])
   / m
Kappa_s[i] <- (tabla_s[1, 1] + tabla_s[2, 2] - m / c)
   / (m - m / c)
especificidad_s[i] <- tabla_s[1, 1] / (tabla_s[1, 1]
   + tabla_s[1, 2])
sensibilidad_s[i] <- tabla_s[2, 2] / (tabla_s[2, 1] +
   tabla_s[2, 2])
FP_s[i] <- tabla_s[1, 2] / (tabla_s[1, 1] +
   tabla_s[1, 2])
FN_s[i] <- tabla_s[2, 1] / (tabla_s[2, 1] +
   tabla_s[2, 2])
precision_s[i] <- tabla_s[2, 2] / (tabla_s[2, 2] +
   tabla_s[1, 2])
```

```
VPN_s[i] <- tabla_s[1, 1] / (tabla_s[1, 1] +
   tabla_s[2, 1])
medida_F_s[i] <- 2 * tabla_s[1, 1] / (2 * tabla_s[1,
   1] + tabla_s[1, 2] + tabla_s[2, 1])

# Para calcular el AUC por separado.
fx <- modelo_s$model$data$x
fy <- modelo_s$model$data$y
modelo_roc <- naive_bayes(fx, fy)
pred_s <- predict(modelo_roc, xtest, type = "prob")[,
   2]
roc_s <- prediction(pred_s, ytest)
auc_s[i] <- as.numeric(performance(roc_s,
   "auc")@y.values)

# Creamos el modelo Democratic Co-learning, y
   calculamos las predicciones y las medidas de
   efectividad.
modelo_d <- democratic(x = xtrain, y = ytrain,
   learners = learner_d, learners.pars =
   learner_pars_d, preds = predict_d, preds.pars =
   predict_pars_d)
pred_d <- predict(modelo_d, xtest)
tabla_d <- table(pred_d, ytest)
acierto_global_d[i] <- (tabla_d[1, 1] + tabla_d[2,
   2]) / m
error_global_d[i] <- (tabla_d[1, 2] + tabla_d[2, 1])
   / m
Kappa_d[i] <- (tabla_d[1, 1] + tabla_d[2, 2] - m / c)
   / (m - m / c)
especificidad_d[i] <- tabla_d[1, 1] / (tabla_d[1, 1]
   + tabla_d[1, 2])
sensibilidad_d[i] <- tabla_d[2, 2] / (tabla_d[2, 1] +
   tabla_d[2, 2])
FP_d[i] <- tabla_d[1, 2] / (tabla_d[1, 1] +
   tabla_d[1, 2])
FN_d[i] <- tabla_d[2, 1] / (tabla_d[2, 1] +
   tabla_d[2, 2])
precision_d[i] <- tabla_d[2, 2] / (tabla_d[2, 2] +
   tabla_d[1, 2])
```

```
  VPN_d[i] <- tabla_d[1, 1] / (tabla_d[1, 1] +
     tabla_d[2, 1])
  medida_F_d[i] <- 2 * tabla_d[1, 1] / (2 * tabla_d[1,
     1] + tabla_d[1, 2] + tabla_d[2, 1])

}

# Mostramos la curva ROC relativa a la última iteración
   calculada.
plot(performance(roc_s, "tpr", "fpr"), main = "Curva
   ROC. Bootstrap, Self-Training, última iteración",
   xlab = "1 - Especificidad", ylab = "Sensibilidad")
abline(a = 0, b = 1, col = "blue", lty = 2)
legend("bottomright", legend = paste("AUC =",
   round(auc_s[i], 4)))
```




```
# Calculamos las medidas de efectividad, que serán la
   media de las correspondientes de cada iteración.
acierto_global_B_s <- mean(acierto_global_s)
error_global_B_s <- mean(error_global_s)
Kappa_B_s <- mean(Kappa_s)
especificidad_B_s <- mean(especificidad_s)
sensibilidad_B_s <- mean(sensibilidad_s)
FP_B_s <- mean(FP_s)
```

```
FN_B_s <- mean(FN_s)
precision_B_s <- mean(precision_s)
VPN_B_s <- mean(VPN_s)
medida_F_B_s <- mean(medida_F_s)
auc_B_s <- mean(auc_s)

# Guardamos todas las medidas de validación.
datos_B_s <- c(acierto_global_B_s, error_global_B_s,
   Kappa_B_s, especificidad_B_s, sensibilidad_B_s,
   FP_B_s, FN_B_s, precision_B_s, VPN_B_s,
   medida_F_B_s, auc_B_s)

# Calculamos las medidas de efectividad, que serán la
   media de las correspondientes de cada iteración.
acierto_global_B_d <- mean(acierto_global_d)
error_global_B_d <- mean(error_global_d)
Kappa_B_d <- mean(Kappa_d)
especificidad_B_d <- mean(especificidad_d)
sensibilidad_B_d <- mean(sensibilidad_d)
FP_B_d <- mean(FP_d)
FN_B_d <- mean(FN_d)
precision_B_d <- mean(precision_d)
VPN_B_d <- mean(VPN_d)
medida_F_B_d <- mean(medida_F_d)

# Guardamos todas las medidas de validación.
datos_B_d <- c(acierto_global_B_d, error_global_B_d,
   Kappa_B_d, especificidad_B_d, sensibilidad_B_d,
   FP_B_d, FN_B_d, precision_B_d, VPN_B_d, medida_F_B_d)


### Tabla de resultados.

# Unimos todos los resultados de Self-training, y
   renombramos filas y columnas.
tabla_s <- round(cbind(datos_VS_s, datos_VC_K_s,
   datos_VC_MC_s, datos_B_s), 4)
rownames(tabla_s) <- c("Acierto global", "Error
   global", "Kappa", "Especificidad", "Sensibilidad",
   "Falsos positivos", "Falsos negativos", "Precisión",
```

```
   "Valor predictivo negativo", "Medida-F", "AUC")
colnames(tabla_s) <- c("Validación simple", "Validación
   cruzada k-pliegues", "Validación cruzada Monte
   Carlo", "Bootstrap")
tabla_s
```

Tabla de resultados de *Self-training*

	Validación simple	Validación cruzada k-pliegues	Validación cruzada Monte Carlo	Bootstrap
Acierto global	0.7254	0.7172	0.7164	0.7182
Error global	0.2746	0.2828	0.2836	0.2818
Kappa	0.4508	0.4344	0.4327	0.4364
Especificidad	0.7318	0.7242	0.7239	0.7252
Sensibilidad	0.6998	0.6885	0.6851	0.6890
Falsos positivos	0.2682	0.2758	0.2761	0.2748
Falsos negativos	0.3002	0.3115	0.3149	0.3110
Precisión	0.3953	0.3709	0.3713	0.3729
Valor predictivo negativo	0.9068	0.9075	0.9060	0.9075
Medida-F	0.8099	0.8055	0.8048	0.8061
AUC	0.7551	0.7454	0.7432	0.7463

```
# Unimos todos los resultados de Democratic
   Co-learning, y renombramos filas y columnas.
tabla_d <- round(cbind(datos_VS_d, datos_VC_K_d,
   datos_VC_MC_d, datos_B_d), 4)
rownames(tabla_d) <- c("Acierto global", "Error
   global", "Kappa", "Especificidad", "Sensibilidad",
   "Falsos positivos", "Falsos negativos", "Precisión",
   "Valor predictivo negativo", "Medida-F")
colnames(tabla_d) <- c("Validación simple", "Validación
   cruzada k-pliegues", "Validación cruzada Monte
   Carlo", "Bootstrap")
tabla_d
```

Tabla de resultados de *Democratic Co-learning*

	Validación simple	Validación cruzada k-pliegues	Validación cruzada Monte Carlo	Bootstrap
Acierto global	0.8278	0.8243	0.8177	0.7946
Error global	0.1722	0.1757	0.1823	0.2054
Kappa	0.6555	0.6486	0.6355	0.5893
Especificidad	0.8257	0.8243	0.8194	0.8071
Sensibilidad	0.8333	0.8253	0.8137	0.7633
Falsos positivos	0.1743	0.1757	0.1806	0.1929
Falsos negativos	0.1667	0.1747	0.1863	0.2367
Precisión	0.6430	0.6410	0.6307	0.6093
Valor predictivo negativo	0.9293	0.9250	0.9206	0.8963
Medida-F	0.8744	0.8717	0.8670	0.8493

Como vemos, los resultados de ambos modelos son bastante buenos, siendo mejor el *Democratic Co-learning*.

Bibliografía

[1] Friedman, J., Hastie, T., & Tibshirani, R. (2001). *The elements of statistical learning*. Springer

[2] Chapelle, O., Schölkopf, B. & Zien, A. (2006). *Semi-Supervised Learning*.

[3] Albalate, A., & Minker, W. (2013). *Semi-supervised and unsupervised machine learning: novel strategies*. John Wiley & Sons.

[4] Zhu, X. (2008). *Semi-Supervised Learning Literature Survey*. University of Wisconsin.

[5] Gammerman, A., Vovk, V., & Vapnik, V. (2013). Learning by transduction. *arXiv preprint arXiv:1301.7375*.

[6] Xu, Z., King, I., & Lyu, M. R. (2010). *More Than Semi-supervised Learning: A Unified View on Learning with Labeled and Unlabeled Data*. LAP Lambert Academic Publishing.

[7] Jiao, Y., & Du, P. (2016). Performance measures in evaluating machine learning based bioinformatics predictors for classifications. *Quantitative Biology, 4*(4), 320-330.

[8] Fushiki, T. (2011). Estimation of prediction error by using K-fold cross-validation. *Statistics and Computing, 21*(2), 137-146.

[9] Xu, Q. S., & Liang, Y. Z. (2001). Monte Carlo cross validation. *Chemometrics and Intelligent Laboratory Systems, 56*(1), 1-11.

[10] Davison, A. C., & Hinkley, D. V. (1997). *Bootstrap methods and their application*. Cambridge University Press.

[11] Tanha, J., van Someren, M., & Afsarmanesh, H. (2017). Semi-supervised self-training for decision tree classifiers. *International Journal of Machine Learning and Cybernetics, 8*(1), 355-370.

[12] Chawla, N. V., & Karakoulas, G. (2005). Learning from labeled and unlabeled data: An empirical study across techniques and domains. *Journal of Artificial Intelligence Research, 23*, 331-366.

[13] Li, H., & Yu, B. (2014). Error rate bounds and iterative weighted majority voting for crowdsourcing. *arXiv preprint arXiv:1411.4086*.

[14] Zhu, X., & Goldberg, A. B. (2009). Introduction to semi-supervised learning. *Synthesis lectures on artificial intelligence and machine learning*, 3(1), 1-130.

[15] González, M., Rosado-Falcón, O., & Rodríguez, J. D. *ssc: Semi-Supervised Classification Methods*. 2019. R package version 2.1-0. https://CRAN.R-project.org/package=ssc.

[16] R Core Team, *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2020. https://www.R-project.org/.