Modelos supervisados © EDICIONES ROBLE, S.L.

Indice

Modelos supervisados ... **3**

I. Introducción ... 3

II. Objetivos ... 3

III. Predicción de valores continuos con regresión lineal ... 3

3.1. Estimación del modelo mediante mínimos cuadrados ... 5

3.2. Evaluación de la regresión ... 8

3.3. Regresión lineal con scikit-learn ... 9

3.4. Regresión lineal múltiple ... 11

3.5. Validación fuera de muestra ... 12

3.6. Variables categóricas ... 13

IV. Clasificación mediante regresión logística ... 16

4.1. Evaluación de los resultados en problemas de clasificación ... 16

4.2. Regresión logística ... 18

4.3. Regresión logística con scikit-learn ... 19

V. Árboles de decisión ... 23

5.1. Implementación de árboles en scikit-learn ... 26

5.2. Random forests: combinación de árboles de decisión ... 31

VI. Otros modelos supervisados ... 32

6.1. Máquinas vectores de soporte ... 32

6.2. Naïve Bayes ... 34

VII. Resumen ... 35

Ejercicios ... **36**

Caso práctico ... 36

Solución ... 36

Recursos ... **38**

Enlaces de Interés ... 38

Bibliografía ... 38

Glosario. ... 38

Modelos supervisados

I. Introducción

En esta unidad se presentarán algunas de las principales técnicas utilizadas para la creación de modelos de aprendizaje supervisado. Estos modelos se caracterizan por ser entrenados mediante conjuntos de datos en los que la solución al problema planteado es conocida, es decir, el conjunto de entrenamiento se divide en una o varias variables independientes y una dependiente (la solución del problema). Estos modelos permiten aprender patrones basándose en la experiencia.

La unidad se iniciará con los modelos de regresión lineal. Estos modelos permiten predecir valores numéricos como, por ejemplo, una puntuación en un ejercicio, la temperatura o los litros de lluvia recogidos en un día. Además, se repasarán algunas de las métricas existentes para evaluar la calidad de una regresión y los procesos para identificar cuándo un modelo se encuentra sobreajustado.

Posteriormente se presentarán varios modelos de clasificación que permiten asignar etiquetas a registros en conjuntos de datos: la regresión logística, los árboles de decisión, random forest, las máquinas vector soporte y los clasificadores bayesianos ingenuos. En esta parte también se verán algunas de las métricas existentes más utilizadas para evaluar la calidad de este tipo de modelos.

Esta unidad se complementa con un notebook Python en el que se incluye todo el código utilizado y algunos ejemplos adicionales a los que se hace referencia en el texto. Es aconsejable seguir el texto con este notebook U3_Codigos

II. Objetivos

Los objetivos que los alumnos alcanzarán tras el estudio de esta unidad son:

- Comprender el proceso del gradiente descendente.
- Construir modelos utilizando la regresión lineal.
- Conocer las métricas para evaluar la calidad de una regresión.
- Construir modelos utilizando la regresión logística.
- Conocer las métricas para evaluar la calidad de un problema de clasificación.
- Construir modelos utilizando árboles de decisión.
- Construir modelos utilizando máquinas de vector soporte.
- Construir modelos utilizando un clasificador bayesiano ingenuo.

III. Predicción de valores continuos con regresión lineal

La primera técnica de aprendizaje supervisado que se va a estudiar en esta unidad es la regresión lineal. Los modelos de regresión lineal son unos de los más utilizados debido a que muestran una gran estabilidad frente a pequeños cambios en los datos de entrenamiento y los resultados son fácilmente interpretables. Además, su estudio es de gran interés, ya que son la base de muchas otras técnicas no lineales más avanzadas.

En los modelos de regresión se busca relacionar el valor que toma una variable (a la que se denomina variable dependiente) en función del valor de una o varias variables cuyos valores son conocidos (denominadas variables independientes). La relación entre la variable dependiente y la independiente es determinada por los parámetros del modelo. Generalmente, los valores que toma la variable dependiente solamente se conocen en un conjunto de datos de entrenamiento que es utilizado para estimar los parámetros.

Matemáticamente un modelo de regresión lineal se puede describir mediante la siguiente ecuación:

$$y(x) = \omega_0 + \omega_1 x$$

En esta fórmula, y es la variable dependiente, ω_0 es el parámetro que determina el valor en el origen, ω_1 es el parámetro que determina la influencia de la variable independiente sobre la dependiente y x es la variable independiente.

En general, los modelos de regresión se pueden interpretar fácilmente utilizando una perspectiva geométrica. Para los casos en los que solamente existe una variable explicativa, se puede utilizar un sistema de coordenadas cartesiano en el que se sitúa la variable independiente en el eje de abscisas (o eje x) y la variable dependiente en el eje de ordenadas (o eje y). En esta gráfica el modelo que relaciona las características se puede representar mediante una recta.

Por ejemplo, en la figura 2.1. se muestra un modelo de regresión lineal que relaciona las unidades producidas en una factoría en función de las horas trabajadas. En esta figura, los puntos rojos representan un conjunto de datos históricos y la línea azul el modelo.

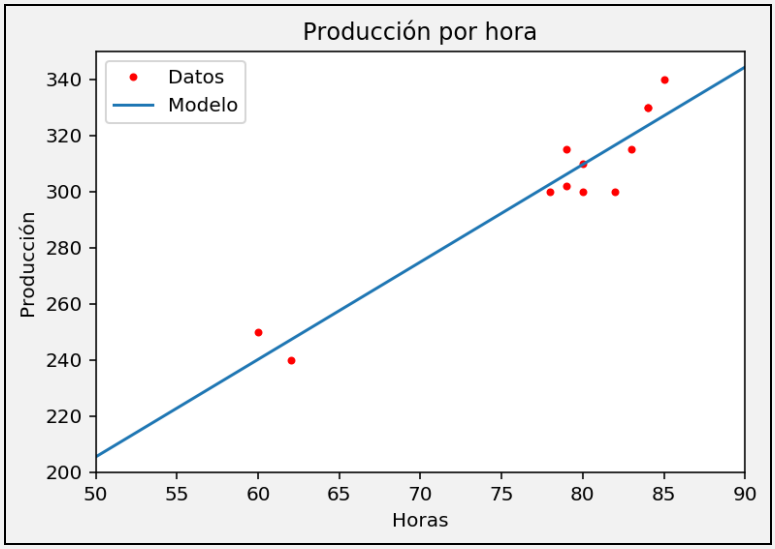


Figura 2.1. Ejemplo de un modelo de regresión lineal. *Fuente:* elaboración propia.

En la figura 2.1. se puede observar que el modelo no reproduce perfectamente los valores de todos los datos utilizados para el entrenamiento, pero ofrece una buena aproximación de las observaciones. Esto es así porque, como se ha comentado en la unidad anterior, los modelos son una simplificación de la realidad y, generalmente, no disponen de toda la información necesaria para realizar la predicción.

3.1. Estimación del modelo mediante mínimos cuadrados

El proceso de entrenamiento consiste en buscar los parámetros que reducen el error total cometido por el modelo en el conjunto de datos de entrenamiento. Para esto es necesario definir una función que mida el error, la cual suele ser llamada función de esfuerzo. En el caso de las regresiones lineales, una de las funciones de esfuerzo más utilizada es el error cuadrático:

$$J(\omega_0, \omega_1) = \frac{1}{2m} \sum_{i=1}^{m} (y(x_i; \omega_0, \omega_1) - y_i)^2$$

En ella J representa la función de esfuerzo, m es el número de datos en el conjunto de entrenamiento, x_i es la variable independiente para el registro i e y_i es la variable dependiente para el registro i.

Los parámetros que minimizan la función de esfuerzo se suelen obtener utilizando el método del gradiente descendente. Este método utiliza el gradiente de la función de esfuerzo en el espacio de parámetros para modificar los valores de un conjunto de parámetros iniciales, los cuales pueden ser aleatorios o haber sido obtenidos mediante algún procedimiento, de forma iterativa hasta que se alcanzan unos valores que minimizan la función de esfuerzo. La expresión que se utiliza en cada una de las iteraciones para obtener los nuevos parámetros es:

$$w_i := w_i - \alpha \frac{\partial}{\partial\, w_i} J(w_0, w_1)$$

donde α es el coeficiente de aprendizaje y $\frac{\partial}{\partial\, w_i} J(w_0, w_1)$ indica la derivada parcial de la función de esfuerzo respecto al parámetro w_i. El coeficiente de aprendizaje indica la velocidad con la que el procedimiento aprende, un valor bajo de este coeficiente puede hacer que el algoritmo no converja o tarde demasiado, mientras que un valor demasiado grande puede hacer que no se encuentren los mínimos globales.

Representación gráfica del algoritmo del gradiente descendente

La figura 2.2. es una explicación gráfica del algoritmo del gradiente descendente, donde se muestra la función de esfuerzo frente al valor del parámetro. En esta figura la función de esfuerzo se presenta mediante una línea de puntos. El algoritmo comienza con el valor del parámetro marcado por el número 1. En este punto se obtiene el gradiente de la función de esfuerzo para substraérselo al parámetro, lo que lleva al punto 2. En este caso se repite el procedimiento para llegar al punto 3 y 4. En el punto 4 la variación es tan pequeña que se puede considerar que se ha llegado al punto de convergencia. Aunque no sea necesariamente el valor mínimo de la función, sino uno próximo.

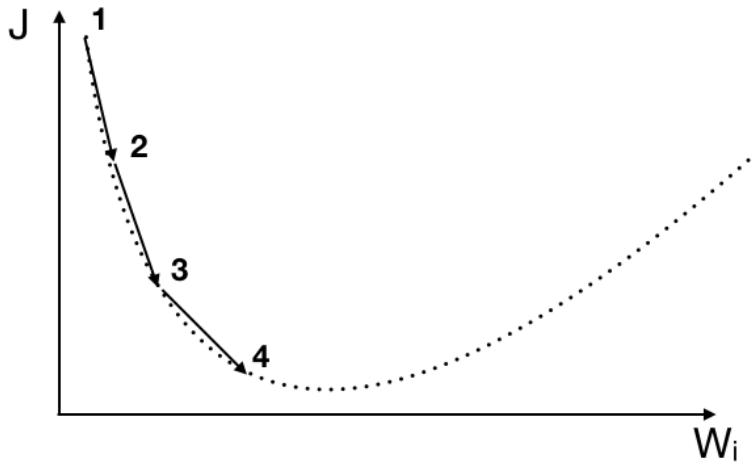


Figura 2.2. Representación gráfica de algoritmo del gradiente descendente.

Este algoritmo se puede implementar en Phyton; para ello, primero se han de escribir las ecuaciones necesarias para actualizar los parámetros:

$$w_o := w_o + \alpha \frac{1}{n} \sum_{i=1}^{n} \left(y_i + (w_o + w_1 x_i)\right)$$

y

$$w_1 := w_1 + \alpha \frac{1}{n} \sum_{i=1}^{n} \left(y_i + (w_o + w_1 x_i)\right) x_i$$

Implementación en Python del método del gradiente descendente

A partir de lo cual se puede hacer una implementación del método como la que se muestra en la figura 2.3. En esta implementación se define la función stepGradient para actualizar los valores de los parámetros mediante el gradiente y el coeficiente de aprendizaje. Es importante destacar que el gradiente debe calcularse para cada uno de los parámetros de forma independiente y posteriormente se actualizan todos los valores; realizar primero el cálculo para uno de los parámetros y posteriormente para el otro es un error muy habitual que lleva a resultados erróneos. La segunda función (fitGradient) realiza el ajuste de los datos iterando hasta que la diferencia entre los dos conjuntos de parámetros es inferior a un límite dado, en el caso del ejemplo 10^{-6}, o se ha alcanzado una cantidad máxima de iteraciones, en este caso 30. Limitar el número de iteraciones es importante para evitar que, en caso de que no se alcance el criterio de convergencia, el algoritmo no termine su ejecución nunca. Finalmente, en el código se crea un conjunto de datos de entrenamiento, en el que son conocidos los parámetros, al que se añade un ruido blanco, tras ello se evalúan los resultados.

```
def stepGradient(par, x, y, learningRate):
    b_0_gradient = 0
    b_1_gradient = 0
    N = float(len(x))

    for i in range(0, len(x)):
        b_0_gradient += (2/N) *         (y[i] - (par[0] + par[1] * x[i]))
        b_1_gradient += (2/N) * x[i] * (y[i] - (par[0] + par[1] * x[i]))

    new_b_0 = par[0] + (learningRate * b_0_gradient)
    new_b_1 = par[1] + (learningRate * b_1_gradient)

    return [new_b_0, new_b_1]

def fitGradient(par, x, y, learningRate, maxDifference = 1e-6, maxIter = 30):
    prev_step = par[:]
    num_iter = 0;

    num_iter += 1
    results = stepGradient(prev_step, trX, trY, learningRate)
    difference = abs(prev_step[0] - results[0]) + abs(prev_step[1] - results[1])

    while ((difference > maxDifference) & (num_iter < maxIter)):
        num_iter += 1
        prev_step = results
        results = stepGradient(prev_step, trX, trY, learningRate)
        difference = abs(prev_step[0] - results[0]) + abs(prev_step[1] - results[1])

    return results

trX = np.linspace(-2, 2, 101)
trY = 3 + 2 * trX + np.random.randn(*trX.shape) * 0.33

print fitGradient([1,1], trX, trY, 0.05)
```

Figura 2.3. Implementación en Python del método del gradiente descendente.

Al ejecutar el código de la figura 2.3. se observará que se obtienen valores próximos a los esperados, lo que indica que la implementación del gradiente descendente ha sido realizada correctamente.

Representación gráfica de la función de esfuerzo

La función de esfuerzo se puede representar gráficamente para ver la forma que toma. Para esto solamente se ha de crear una malla de puntos y calcular el valor de la función de esfuerzo en todos los puntos, el resultado de muestra en la figura 2.4.

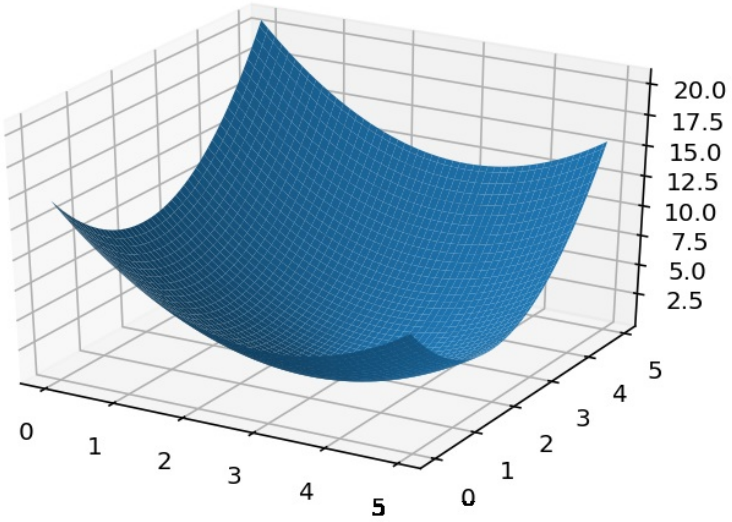


Figura 2.4. Representación gráfica de la función de esfuerzo. *Fuente:* elaboración propia.

3.2. Evaluación de la regresión

Una vez obtenido un modelo es necesario conocer la calidad del mismo a la hora de predecir resultados, para lo cual existen diferentes estimadores. Uno, que se deriva de la función de esfuerzo utilizada en la unidad anterior, es el error cuadrático medio (MSE, *Mean Squared Error*) y se define como el valor medio de la diferencia entre la variable independiente y la predicción al cuadrado, es decir:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - y(x_i))^2$$

Este estimador mide los errores al cuadrado, penalizando los puntos en los que la discrepancia entre la variable independiente y la predicción es mayor. En caso de que no se desee tener en cuenta esta mayor penalización de los puntos con mayor discrepancia se puede utilizar el error absoluto medio (MAE, *Mean Absolute Error*) que se define como el promedio de la diferencia entre la variable dependiente y la predicción del modelo:

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|(y_i - y(x_i)|$$

En los dos estimadores que se han definido, la existencia de valores atípicos, en los cuales el valor estimado por el modelo muestra una gran discrepancia con los originales, puede distorsionar los resultados. Un estimador que evita estos problemas, ya que es robusto frente a los valores atípicos, es la mediana del error absoluto (MedAE, *Median Absolute Error*), que se define como:

$$MedAE = median(|y_1 - y(x_1)|, |y_2 - y(x_2)|, ..., |y_n - y(x_n)|)$$

Al utilizar la mediana, se evita que la existencia de un valor atípico afecte al resultado.

Anotación: coeficiente de determinación

Uno de los estimadores más utilizados es el coeficiente de determinación o R^2. Este coeficiente es un estadístico que se define como 1 menos el cociente entre la varianza de los residuos, partido por la varianza de la variable dependiente, es decir:

$$R^2 = 1 - \frac{\sigma_r^2}{\sigma^2}$$

con

$$\sigma_r^2 = \sum_{i=1}^{n}(y_i - y(x_i))^2$$

y

$$\sigma^2 = \sum_{i=1}^{n}(y_i - \bar{y})^2$$

Aquí $\bar{y}$ representa la media de los valores de la variable dependiente.

El R^2 se puede interpretar como el porcentaje de la variabilidad total de la variable dependiente respecto a la media que se puede explicar con el modelo. Idealmente su valor debería ser 1, lo que indicaría que el modelo reproduce perfectamente los datos del conjunto de entrenamiento. En el caso de un modelo constante, que reproduce la media de la variable dependiente, el valor de R^2 será 0.

En scikit-learn los estimadores se pueden encontrar en sklearn.metrics y cada uno de los definidos previamente se encuentra implementado en las funciones:

- error cuadrático medio: mean_squared_error
- error absoluto medio: mean_absolute_error
- mediana del error absoluto: median_absolute_error
- R^2: r2_score

3.3. Regresión lineal con scikit-learn

En scikit-learn los modelos de regresión lineal se implementan con la clase LinearRegression, que se encuentra en lineal_model. Por ejemplo, en la figura 2.5. se muestra el código que permite obtener el modelo de la figura 2.1.

```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import median_absolute_error

# Conjunto de datos
x = [[80], [79], [83], [84], [78], [60], [82], [85], [79], [84], [80], [62]]
y = [[300], [302], [315], [330], [300], [250], [300], [340], [315], [330], [310], [240]]

# Creación del modelo
model = LinearRegression()
model.fit(x, y)

# Obtención de estimaciones
print 'Con 70 horas la producción sería:', model.predict([[70]])
print

# Predicción del modelo
y_pred = model.predict(x);

# Obtención de los parametros de ajuste
print 'w_0', model.intercept_[0]
print 'w_1', model.coef_[0][0]

print 'R^2', model.score(x, y)
print 'Error cuadrático medio', mean_squared_error(y_pred, y)
print 'Error absoluto medio', mean_absolute_error(y_pred, y)
print 'Mediana del error absoluto', median_absolute_error(y_pred, y)
```

Figura 2.5. Implementación en Python de un modelo de regresión básico.

El código de la figura 2.5. importa la clase LinearRegression en la primera línea. Posteriormente define un conjunto de datos de entrenamiento en los que x son las horas trabajadas e y las unidades producidas. A continuación, se crea el modelo llamando al constructor y se ajusta utilizando los datos de entrenamiento con el método fit. Una vez que se ha ajustado el modelo, se puede predecir un valor para cualquier dato utilizando el método predict, el código se ha evaluado en 70, obteniéndose un valor de 274,87. A continuación se calculan las estimaciones para los puntos utilizados durante el proceso de entrenamiento. Finalmente se muestran los valores para los parámetros w_0 (31,74) y w_1 (3,47) y los estimadores R^2 (0,91), MSE (74,68), MAE (7,52) y MedAE (6,79). El valor de R^2 se estima en el modelo, por lo que no ha sido necesario importar el método r2_score.

Como ya se ha comentado, una de las ventajas de los modelos lineales es que son fácilmente interpretables. El valor de w_1 es el aumento de las unidades que se observaría en la variable dependiente (unidades producidas), cada vez que se aumenta una unidad la variable independiente (horas trabajadas). Por otro lado, w_0 es el valor de la variable dependiente cuando la variable independiente es 0. ¿El modelo está diciendo que cuando no se trabaja se producen 31 unidades? Sí, el modelo así construido indica que existirá una producción fija sin trabajar. Para solucionar este problema se ha de crear un modelo en el que el término independiente sea 0. Eso se puede hacer añadiendo la opción fit_intercept = False cuando se construye el objeto, como en el código de la figura 2.6.

```
# Modelo sin término de independiente
model_ni = LinearRegression(fit_intercept = False)
model_ni.fit(x, y)

# Obtención de estimaciones
print 'Con 70 horas la producción sería:', model_ni.predict([[70]])[0]
print

# Obtención de los parametros de ajuste
print 'w_0', model_ni.intercept_
print 'w_1', model_ni.coef_[0][0]
print 'R^2', model_ni.score(x, y)
```

Figura 2.6. Implementación de un modelo de regresión sin término de intercepción.

Los resultados que se obtienen al ejecutar la figura 2.6. son parecidos a los anteriores, ahora w_0 es 0, w_1 es 3,87 y R^2 es 0,89. Estos resultados son más coherentes con la intuición, lo que indica que este modelo reproduce mejor la realidad que el anterior, a pesar de que el valor de R^2 sea menor.

El número de parámetros son grados de libertad en el modelo, si estos se aumentan generalmente se obtiene una regresión que reproduce mejor los datos utilizados durante el entrenamiento, aunque no siempre se traduce en la consecución de resultados más fiables, ya que se puede haber producido lo que se llama sobreajuste. El modelo ha memorizado los datos de entrada en lugar de obtener un patrón de los mismos. En la siguiente sección se estudiará la regresión lineal múltiple, en la que se aumenta el número de parámetros del modelo, y posteriormente se analizarán formas de detectar el sobreajuste.

3.4. Regresión lineal múltiple

Los modelos lineales se pueden extender para utilizar más de una variable independiente, de forma que con los modelos así construidos se pueda tener en cuenta que el valor de la variable dependiente está condicionado por más de una variable independiente. Matemáticamente, un modelo de regresión múltiple se puede escribir de la siguiente forma:

$$y(x_1, x_2, \cdots, x_n) = w_0 + \sum_{i=1}^{n} w_i x_i$$

Un caso particular de regresión lineal múltiple es la regresión polinómica. En este caso se utiliza una única variable independiente pero elevada a un grado diferente. Estos modelos se pueden escribir de la forma:

$$y(x) = \sum_{i=0}^{n} w_i x^i$$

Nótese que en esta ocasión el término w_0 se ha incluido dentro del sumatorio al comenzar la cuenta en 0 y sin olvidar que cualquier valor elevado a 0 es la unidad.

Continuando con el ejemplo de las horas trabajadas, se pueden probar modelos polinómicos en el número de horas. Para esto es necesario importar el PolynomialFeatures, que se encuentra dentro de preprocessing. Utilizando un objeto PolynomialFeatures se puede crear un conjunto de datos en el que la primera columna son los datos elevados a 0, la segunda son los datos elevados a 1 y así hasta el grado que se indique durante la construcción del objeto. Un parámetro opcional de PolynomialFeatures es include_bias, con el que se puede omitir la columna elevada a grado cero, por defecto, su valor será cierto. El código para crear un modelo de grado 2 se puede ver en la figura 2.7.

```
from sklearn.preprocessing import PolynomialFeatures

poly_2  = PolynomialFeatures(degree = 2, include_bias = False)
x_2     = poly_2.fit_transform(x)

model_2 = LinearRegression(fit_intercept = False)
model_2.fit(x_2, y)

# Obtención de los parametros de ajuste
print 'w_1', model_2.coef_[0][0]
print 'w_2', model_2.coef_[0][1]
print 'R^2', model_2.score(x_2, y)
```

Figura 2.7. Implementación de un modelo de grado 2.

En el código de la figura 2.7. se ha indicado al constructor del objeto que no incluya la columna con el grado cero para evitar que el modelo tenga término independiente. Como se ha visto anteriormente, la inclusión del término independiente produce modelos no realistas en este problema. Al ejecutar el código se observa que el valor de R^2 es mayor que el obtenido con el modelo lineal, lo que indica que reproduce mejor los datos de entrenamiento. Este comportamiento se observa a medida que se aumenta el grado del polinomio utilizado, ya que se aumentan los grados de libertad del modelo.

Al igual que se ha visto antes, un valor de R^2 mayor no significa que el modelo sea más realista. Para comprobar esto se puede representar un modelo de grado 1 y grado 5, como el que se muestra en la figura 2.8. En esta figura se puede ver claramente que el modelo de grado 5 reproduce mejor los datos en el conjunto de entrenamiento y que también va a cero cuando el número de horas trabajas es 0 (nótese la línea pegada al eje de ordenadas). Sin embargo, los valores fuera del conjunto de entrenamiento aparentan no ser correctos. Esto es lo que se conoce como sobreajuste u overfitting: el modelo tiene demasiados grados de libertad y "memoriza" los datos en lugar de obtener la tendencia que existe en los mismos. En próximas secciones se analizarán algunas técnicas para identificar si un modelo ha sido sobreajustado o no.

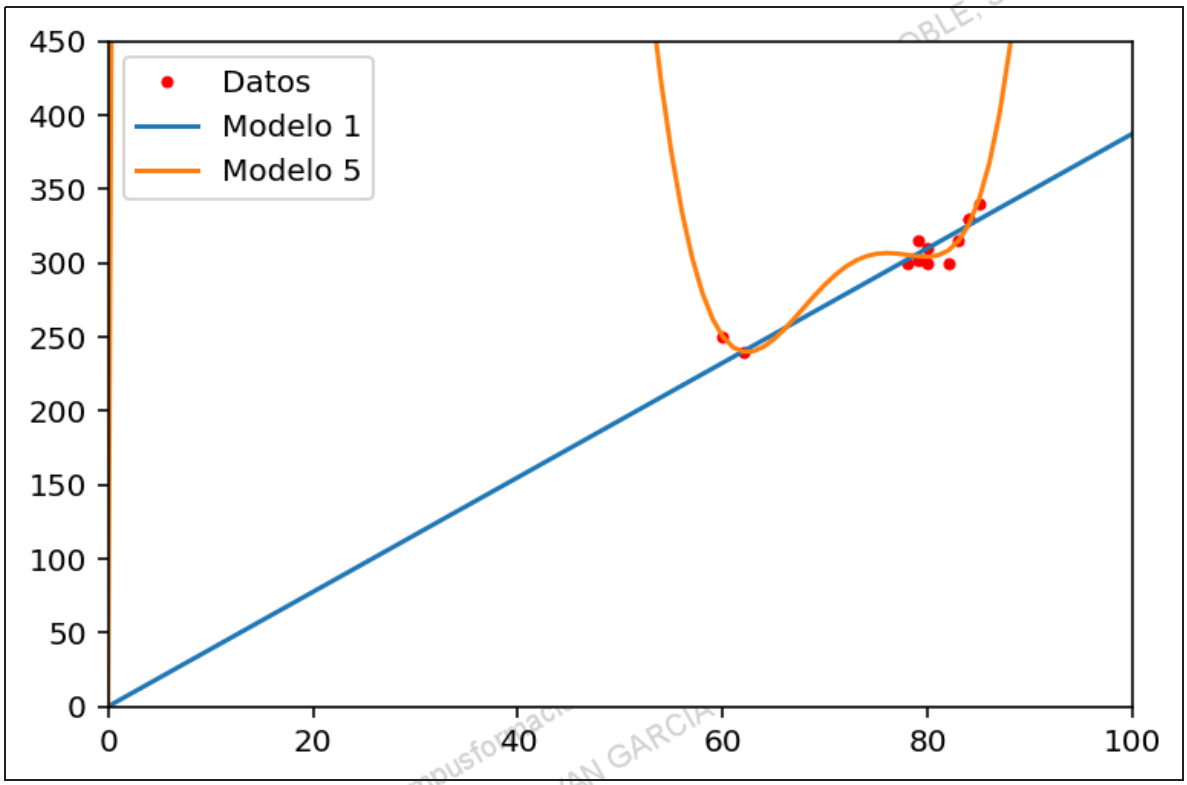


Figura 2.8. Modelos de regresión polinómica de grado 1 y 5. *Fuente:* elaboración propia.

3.5. Validación fuera de muestra

El sobreajuste en un modelo de regresión lineal se puede detectar fácilmente representado las predicciones realizadas por modelo frente a los datos. Este método presenta un problema cuando existen más de dos variables independientes, ya que no es posible hacer una representación gráfica en estos casos, por lo que es necesario utilizar otra aproximación para identificar si un modelo muestra o no sobreajuste. Una aproximación es realizar el entrenamiento con un subconjunto de los datos (denominado grupo de entrenamiento) y, al mismo tiempo, guardar otro diferente para la validación (denominado grupo de test), de este modo, si el modelo "memoriza" los resultados del conjunto de entrenamiento, las predicciones en el conjunto de test no serán correctas. Esta técnica se conoce como validación fuera de muestra.

1

Para poner en práctica la validación fuera de muestra es necesario disponer de un conjunto de datos con suficientes muestras como para poder dividirlo en dos. Generalmente, la mayor cantidad de datos se utilizan en el subgrupo de entrenamiento (70% - 80%) y el resto en el de test. Con esta finalidad, se utilizará el conjunto de datos incluidos en scikit-learn con el precio de vivienda en Boston en función de 13 variables independientes. La implementación de un modelo lineal se puede consultar en la figura 2.9. Al ejecutar este código se observará que el R^2 del modelo es 0,74, aunque no se puede saber si el resultado es fruto del sobreajuste, ya que el número de variables independientes es trece.

```
from sklearn.datasets import load_boston

# Importación de los datos de vienda de Boston
boston = load_boston()

# Creación de un modelo
model_boston = LinearRegression()
model_boston.fit(boston.data, boston.target)

print "R^2:", model_boston.score(boston.data, boston.target)
```

Figura 2.9. Implementación de un modelo de regresión lineal para predecir el precio de la vivienda en Boston.

Para saber si existe sobreajuste en el modelo de la figura 2.9. se puede utilizar el código de la figura 2.10. En este se ha importado la función train_test_split, que permite dividir un conjunto de datos en dos: uno de entrenamiento y otro de test. Una vez creado el modelo con el conjunto de entrenamiento se puede comprobar que el valor de R^2 es similar en ambos conjuntos de datos, por lo que en esta ocasión el modelo no aparenta estar sobreajustado.

2

```
from sklearn.model_selection import train_test_split

x_train, x_test, y_train, y_test = train_test_split(boston.data, boston.target)

# Creación de un modelo
model = LinearRegression()
model.fit(x_train, y_train)

predit_train = model.predict(x_train)
predit_test = model.predict(x_test)

# Evaluación de R2
print 'R2 en entrenamiento es: ', model.score(x_train, y_train)
print 'R2 en validación es: ', model.score(x_test, y_test)
```

Figura 2.10. Validación fuera de muestra para el modelo de precios de la vivienda en Boston.

La función train_test_split utiliza por defecto un 75% de los datos para la muestra de entrenamiento y un 25% para el test. Estos porcentajes se pueden modificar utilizando las opciones train_size para el conjunto de entrenamiento o test_size para indicar el porcentaje del conjunto de prueba. Además, también se puede fijar la semilla utilizada para separar el conjunto de datos con la finalidad de garantizar la reproducibilidad de los resultados empleando la opción random_state.

3.6. Variables categóricas

Todas las variables que se han utilizado hasta ahora en los ejemplos tenían valores numéricos, por lo que se han podido utilizar directamente para la creación de los modelos. Es habitual que en los conjuntos de datos existan variables que son categóricas, como puede ser un nombre de ciudad, que no pueden ser utilizadas directamente en los modelos, aunque pueden contener información útil.

Para incluir la información de una variable categórica en un modelo se ha de crear primero una variable dummy para cada uno de los niveles de la original. Estas se crean con el valor uno, cuando el nivel de la variable original coincida con el de la dummy y cero en cualquier otro caso. Por ejemplo, los datos utilizados para estimar la producción en función de las horas trabajadas provenían de tres factorías diferentes, estos datos se muestran en la figura 2.11. Esta información puede ser útil para conseguir un modelo que permita reproducir mejor los resultados.

Figura 2.11. Conjunto de datos original de las horas y las factorías

	Horas	Factoria
0	80	Factoria 1
1	79	Factoria 2
2	83	Factoria 3
3	84	Factoria 1
4	78	Factoria 2
5	60	Factoria 3
6	82	Factoria 1
7	85	Factoria 2
8	79	Factoria 3
9	84	Factoria 1
10	80	Factoria 2
11	62	Factoria 3

En la figura 2.12. se puede observar cómo quedaría el conjunto de datos de la figura 2.11. cuando son creadas las variables dummies y eliminada la variable Factoria. Se puede ver que se han creado tres variables nuevas: la Factoria 1 tiene un valor de uno en las filas 0, 3, 6 y 9, que se corresponde con aquellas en las que originalmente se encontraba la etiqueta asociada a la nueva variable. Se puede apreciar lo mismo para el resto de variables.

Figura 2.12. Conjunto de datos con las variables dummies creadas.

	Horas	Factoria 1	Factoria 2	Factoria 3
0	80	1	0	0
1	79	0	1	0
2	83	0	0	1
3	84	1	0	0
4	78	0	1	0
5	60	0	0	1
6	82	1	0	0
7	85	0	1	0
8	79	0	0	1
9	84	1	0	0
10	80	0	1	0
11	62	0	0	1

Figura 2.13. Implementación de un modelo con variables categóricas.

```
import pandas as pd

# Conjunto de datos con las factorias
x = [[80, 'Factoria 1'], [79, 'Factoria 2'], [83, 'Factoria 3'],
     [84, 'Factoria 1'], [78, 'Factoria 2'], [60, 'Factoria 3'],
     [82, 'Factoria 1'], [85, 'Factoria 2'], [79, 'Factoria 3'],
     [84, 'Factoria 1'], [80, 'Factoria 2'], [62, 'Factoria 3']]
y = [[300], [302], [315], [330], [300], [250], [300], [340], [315], [330], [310], [240]]

# Conversion de las datos a DataFrame
x_0 = pd.DataFrame(x, columns = ['Horas', 'Factoria'])
y = pd.DataFrame(y)

# Creación de variables dummies
x = pd.concat([x_0['Horas'], pd.get_dummies(x_0['Factoria'])], axis = 1)

# Polinomio de grado 5
model_dummies = LinearRegression(fit_intercept = False)
model_dummies.fit(x, y)

print "Modelo dummies - R^2:", model_dummies.score(x, y)
```

En la figura 2.13. se muestra un ejemplo de código en el que se implementa una regresión lineal con variables dummies. Para llevar a cabo esta tarea es necesario importar pandas, ya que en este paquete se encuentra la función que permite crear estas variables: get_dummies. En este ejemplo se ha añadido el nuevo conjunto de datos a partir del cual se han creado las variables. Finalmente, se ha llevado a cabo el ajuste obtenido, un valor del R^2 de 0,92, que es superior a los valores obtenidos previamente.

Es importante resaltar que una variable categórica no es aquella en la que los campos son cadenas de texto, sino aquella en la que cada uno de los valores se corresponde con una categoría, aunque venga indicada con un valor numérico. Por ejemplo, en el caso anterior, la factoría podría estar identificada con el id de la misma (1, 2 o 3) y no debería ser utilizada directamente, ya que así se le daría a la Factoría 2 el doble de peso que a la Factoría 1 y a la Factoría 3 el triple que a la primera. En la unidad 4 se ampliará la discusión sobre los diferentes tipos de características que se pueden encontrar en los conjuntos de datos.

El conjunto de variables dummies que se obtiene a partir de una variable siempre muestra multicolinealidad. Esto quiere indicar que las variables muestran una fuerte correlación entre ellas, pudiéndose obtener los valores de una en función del resto. Esto es obvio, ya que, debido a su construcción, cualquiera de las variables se puede construir a partir del resto. Los conjuntos de datos con multicolinealidad suelen dar problemas a la hora de entrenar los modelos. En la unidad de selección de variables se estudiarán técnicas para su eliminación.

IV. Clasificación mediante regresión logística

En la sección anterior se ha visto una técnica para relacionar el valor continuo que toma una variable dependiente en función de una o varias variables independientes.

En esta sección se estudiará una técnica para los conjuntos de datos en los que la variable dependiente solamente puede tomar un par de valores, lo que generalmente se etiqueta como positivos (P) y negativos (N).

Este tipo de problemas son conocidos como clasificadores de clase binarios.

4.1. Evaluación de los resultados en problemas de clasificación

Una predicción en un problema de clasificación de clase binaria únicamente puede dar lugar a cuatro posibles resultados:

- Verdaderos Positivos (TP).
- Verdaderos Negativos (TN).
- Falsos Positivos (FP) o Error tipo I.
- Falsos Negativos (FN) o Error de tipo II.

Una forma de mostrar estos resultados es mediante la matriz de confusión o tabla de contingencia, que se puede construir de la forma:

$$\begin{vmatrix} TP & FP \\ FN & TN \end{vmatrix}$$

En el caso de obtener una predicción perfecta, la matriz que se obtiene es diagonal, ya que no existen errores de tipo I (FP) ni de tipo II (FN).

A partir de estas definiciones, se puede construir un conjunto de métricas para evaluar la calidad de una predicción:

Precisión (en inglés, Accuracy)

El porcentaje de aciertos totales.

$$A = \frac{TP + TN}{TP + FP + TN + FN}$$

Exactitud (en inglés, Precision)

El porcentaje de predicciones positivas que son acertadas.

$$P = \frac{TP}{TP + FP}$$

Exhaustividad (en inglés, Recall)

El porcentaje de valores positivos que son identificados en la predicción.

$$R = \frac{TP}{TP + FN}$$

La existencia de varias métricas en los problemas de clasificación es necesaria, ya que, en la mayoría de las ocasiones, el coste de un error de tipo I es diferente al de tipo II. Por ejemplo, en el caso de identificar contrataciones fraudulentas, un falso positivo se traducirá en la pérdida de un cliente, mientras que los falsos negativos serán el valor del fraude. Generalmente, los costes en cada escenario suelen ser diferentes dependiendo del negocio, por eso en unos casos se deberá prestar más atención a la exactitud y en otros a la exhaustividad.

Para evaluar la calidad de la predicción se puede utilizar la curva ROC, en la que se representa en el eje x la ratio de falsos positivos y en el eje y la ratio de verdaderos positivos. Se puede ver un esquema de esta gráfica en la figura 2.14. El porcentaje del área que cubre la curva se conoce como Área Bajo la Curva (*AUC*, *Area Under the Curve*) y, cuanto más cercano sea a 1, mejor será la predicción.

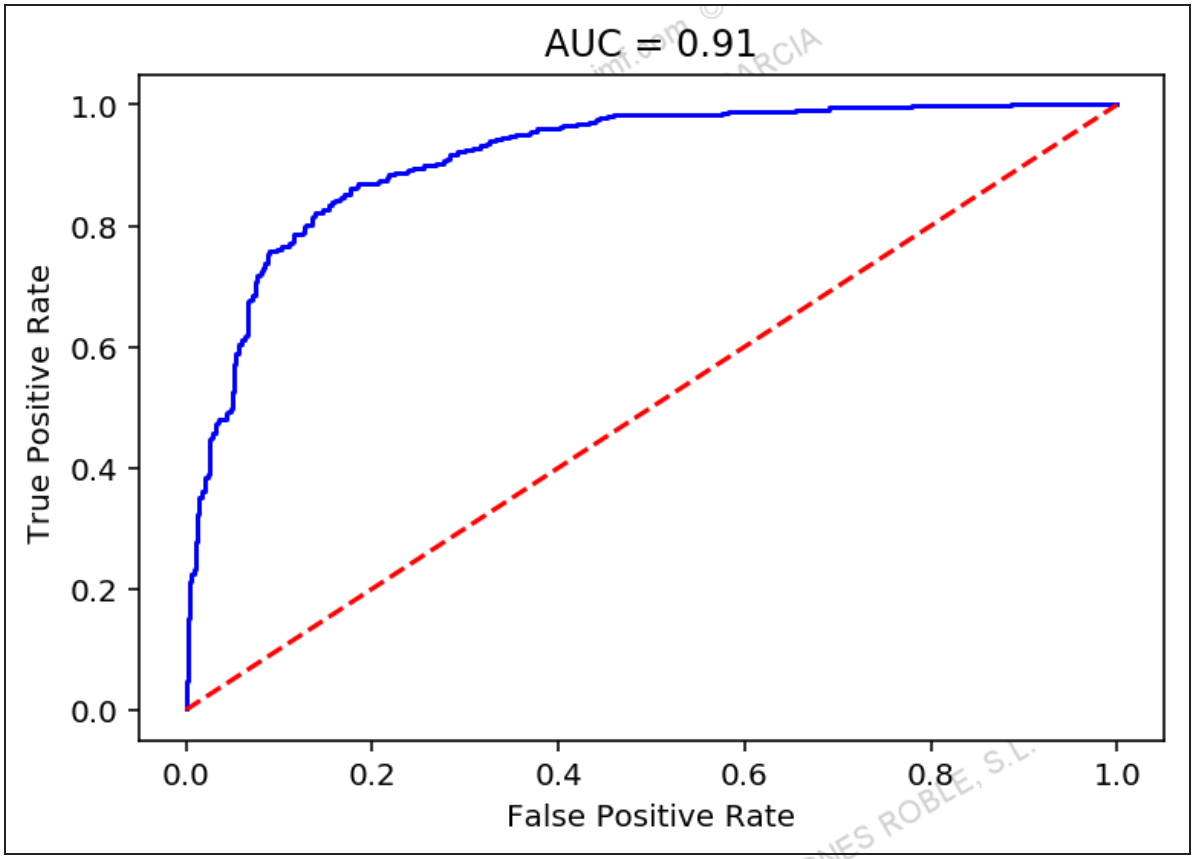


Figura 2.14. Curva ROC y definición del área bajo la curva.

4.2. Regresión logística

Los problemas de clasificación binaria se pueden abarcar empleando las técnicas de regresión vistas en la sección anterior. Para ello, se puede utilizar una curva logística que responde a la siguiente forma:

$$y(x_1, x_2, \cdots, x_n) = \frac{1}{1 + e^{\sum_i w_i x_i}}$$

Se observa que esta curva solamente puede tomar valores entre 0, cuando el sumatorio de la exponencial se hace infinito, y 1, cuando el sumatoria de la exponencial se hace menos infinito. Su valor se puede interpretar como la probabilidad de que el registro sea positivo y para obtener una predicción simplemente se ha de definir un valor umbral a partir del cual se asume una predicción positiva y por debajo de él, negativa. Por defecto, se suele usar el valor 0,5 de umbral, pero este se puede modificar en función de si se desea dar prioridad a reducir el error de tipo I o de tipo II.

La forma de esta curva se puede ver en la figura 2.15. para un caso unidimensional. La pendiente de la curva y la posición del punto de inflexión vienen determinados por los valores de los parámetros w_i, que han de ser estimados en el proceso de regresión. Para esto se puede utilizar el gradiente descendente, que se ha estudiado anteriormente, utilizando la función de esfuerzo adecuada para este problema.

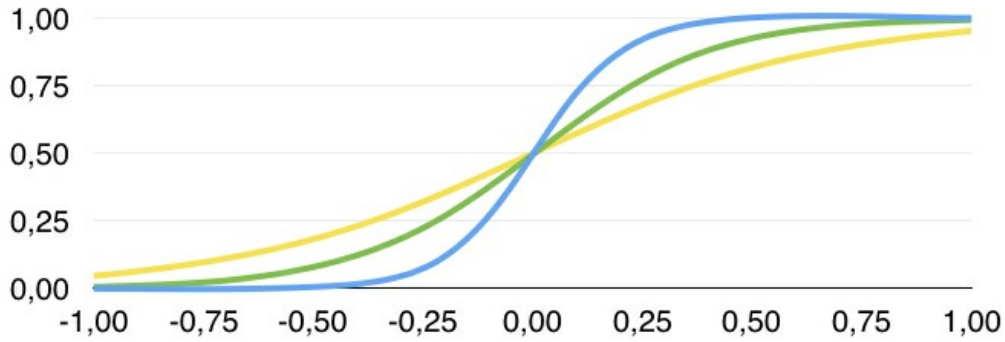


Figura 2.15. Regresión logística. *Fuente:* elaboración propia.

4.3. Regresión logística con scikit-learn

En Python, la regresión logística, así como las métricas de rendimiento para los modelos de clasificación, se encuentran disponibles en scikit-learn. Además, también se pueden encontrar conjuntos de datos precargados o funciones para la creación aleatoria de los mismos.

1

En la figura 2.16. se muestra un código de ejemplo en el que se implementa una regresión logística. En este código, lo primero que se hace es importar el método make_classification para crear un conjunto de datos aleatorios, concretamente se ha creado un conjunto de 2500 registros, con tres variables independientes, sin redundancia y fijando la semilla del generador de números aleatorios en 1. El hecho de fijar la semilla permite que los resultados obtenidos no cambien cada vez que se ejecute el código. Una vez creado el conjunto de datos, se divide en dos conjuntos utilizando el método visto anteriormente, en la sección de validación fuera de muestra, y se fija, igualmente, el valor de la semilla.

```
from sklearn.datasets import make_classification
from sklearn.linear_model.logistic import LogisticRegression
from sklearn.metrics import confusion_matrix

# Creción de un conjunto de entrenamiento
X, y = make_classification(n_samples    = 2500,
                           n_features   = 3,
                           n_redundant  = 0,
                           random_state = 1)

# Creación de un conjunto de enrenamietno y test
x_train, x_test, y_train, y_test = train_test_split(X, y, random_state = 1)

# Ajuste del modelo logístico
classifier   = LogisticRegression().fit(x_train, y_train)
y_train_pred = classifier.predict(x_train)
y_test_pred  = classifier.predict(x_test)

# Obtención de matriz de confusión
confusion_matrix_train = confusion_matrix(y_train, y_train_pred)
confusion_matrix_test = confusion_matrix(y_test, y_test_pred)

print 'La matriz de confusión para entrenamiento es'
print confusion_matrix_train
print 'La matriz de confusión para test es'
print confusion_matrix_test
```

Figura 2.16. Implementación de una regresión logística.

2

Una vez dividido el conjunto de datos en dos, se procede al entrenamiento de un modelo utilizando el método fit de un objeto LogisticRegression. Con este objeto se pueden obtener las predicciones utilizando el método predict. A partir de las predicciones se utilizará la función confusion_matrix para obtener la matriz de confusión. Concretamente, para el conjunto de entrenamiento se tiene:

$$\begin{vmatrix} 862 & 90 \\ 97 & 826 \end{vmatrix}$$

Mientras que para el de test, el resultado es:

$$\begin{vmatrix} 267 & 35 \\ 28 & 294 \end{vmatrix}$$

Las matrices de confusión muestran que la predicción es buena, ya que hay pocos registros fuera de la diagonal principal, donde se sitúan los falsos positivos y los falsos negativos. Para comparar los resultados y verificar si existe o no sobreajuste, se pueden normalizar las matrices dividiéndolas por el número total de registros de cada una, obteniendo así los porcentajes de cada tipo de acierto y fallo. El código para normalizar las matrices se muestra en la figura 2.17.

```
print 'La matriz de confusión para entrenamiento normalizada es'
print confusion_matrix_train / double(sum(confusion_matrix_train))
print 'La matriz de confusión para test normalizada es'
print confusion_matrix_test / double(sum(confusion_matrix_test))
```

Figura 2.17. Normalización de las matrices de confusión.

Al ejecutar el código de la figura 2.17., la matriz de confusión de entrenamiento normalizada que se obtiene es:

$$\begin{vmatrix} 0{,}46 & 0{,}05 \\ 0{,}05 & 0{,}44 \end{vmatrix}$$

Mientras que en el caso de la de test, se obtiene:

$$\begin{vmatrix} 0{,}43 & 0{,}05 \\ 0{,}04 & 0{,}47 \end{vmatrix}$$

Las matrices son similares, lo que indica que posiblemente no exista sobreajuste.

3

La existencia o no de sobreajuste se puede comprobar también utilizando las métricas que se han visto anteriormente, para lo cual se han de importar las funciones accuracy_score, precision_score y recall_score para obtener la precisión, exactitud y exhaustividad, respectivamente. Esto se puede hacer utilizando el código de la figura 2.18.

```
from sklearn.metrics import accuracy_score, precision_score, recall_score

print 'Resultados en el conjunto de entrenamiento'
print ' Precisión:', accuracy_score(y_train, y_train_pred)
print ' Exactitud:', precision_score(y_train, y_train_pred)
print ' Exhaustividad:', recall_score(y_train, y_train_pred)
print ''
print ' Resultados en el conjunto de test'
print ' Precisión:', accuracy_score(y_test, y_test_pred)
print ' Exactitud:', precision_score(y_test, y_test_pred)
print ' Exhaustividad:', recall_score(y_test, y_test_pred)
```

Figura 2.18. Obtención de las métricas de rendimiento del modelo de clasificación logístico.

Así, se obtienen los siguientes resultados para el conjunto de entrenamiento:

- Precisión: 0,90.
- Exactitud: 0,90.
- Exhaustividad: 0,89.

Y para el conjunto de test:

- Precisión: 0,90.

- Exactitud: 0,89.
- Exhaustividad: 0,91.

Estas conclusiones permiten confirmar que, posiblemente, no exista sobreajuste en este ejemplo. Este resultado es el esperado, ya que los conjuntos de datos para el entrenamiento creados mediante las funciones de scikit-learn son adecuados para la creación de modelos.

4

Finalmente, también hay disponibles funciones para representar la curva ROC y obtener el AUC del modelo. Para esto se han de importar la función roc_curve, para obtener las ratios de falsos positivos y verdaderos negativos, y auc, para poder calcular el valor del AUC. En la figura 2.19. se muestra un ejemplo para obtener la curva ROC y el AUC en el conjunto de datos de test.

```
from sklearn.metrics import roc_curve, auc

false_positive_rate, recall, thresholds = roc_curve(y_test, y_test_pred)
roc_auc = auc(false_positive_rate, recall)

print 'AUC:', auc(false_positive_rate, recall)

plot(false_positive_rate, recall, 'b')
plot([0, 1], [0, 1], 'r--')
title('AUC = %0.2f' % roc_auc)
```

Figura 2.19. Código para la representación de la curva ROC y el Área Bajo la Curva (AUC).

El valor del ROC que se obtiene con el código de la figura 2.19. es 0,90 y la gráfica resultante se muestra en la figura 2.20.

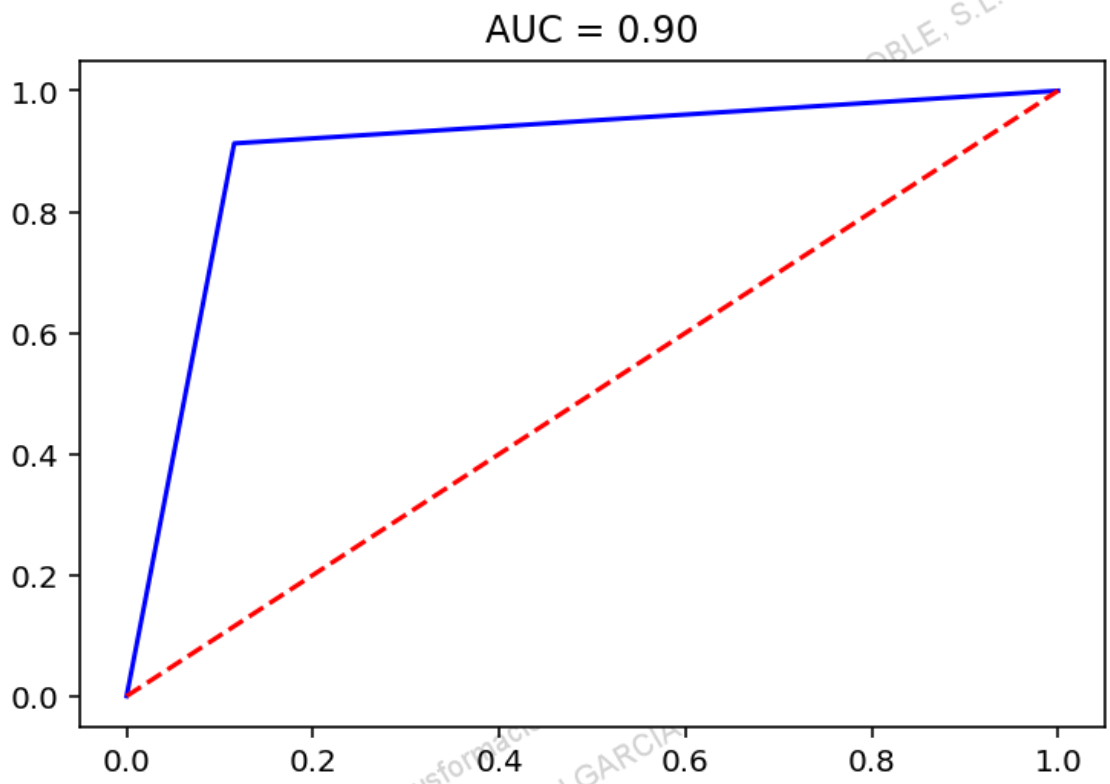


Figura 2.20. Curva ROC obtenida en el ejemplo de la figura 19.

5

Como se ha comentado anteriormente, los resultados de la regresión logística se pueden interpretar como la probabilidad de que cada uno de los registros pertenezca a una de las dos clases y se puede utilizar un umbral diferente en función de si se desea favorecer un tipo de error u otro. Actualmente, esto no se puede realizar con el método predict de la clase LogisticRegression, de modo que es necesario obtener las probabilidades con el método predict_proba y realizar las predicciones manualmente. En la figura 2.21 se muestra un código de ejemplo para realizar esta operación. predict_proba retorna una matriz con dos columnas y tantas filas como registros existan en los datos, en la que la primera columna tiene la probabilidad de que el registro sea falso y la segunda de que sea cierto.

```
prob = classifier.predict_proba(x_test)

y_th = np.ones(len(y_test), dtype=bool)

for th in (0.7, 0.3):
    for i in range(len(y_test)):
        y_th[i] = prob[i][1] > th

    print 'Precisión ', th, ':', accuracy_score(y_test, y_th)
    print 'Exactitud ', th, ':', precision_score(y_test, y_th)
    print 'Exhaustividad ', th, ':', recall_score(y_test, y_th)
    print
```

Figura 2.21. Predicciones con diferentes umbrales de selección.

Al ejecutar el código de la figura 2.21. se obtienen los siguientes resultados, cuando el umbral de decisión se ha fijado en 0,7:

- Precisión: 0.91.
- Exactitud: 0.95.
- Exhaustividad: 0.87.

Si el umbral se fija en 0,3, los valores son:

- Precisión: 0.88.
- Exactitud: 0.83.
- Exhaustividad: 0.96.

En ambos casos la precisión del modelo se ve reducida: la exactitud aumenta en el primer caso (umbral 0,7), mientras que se reduce en el segundo (umbral 0,3) y con la exhaustividad ocurre lo contrario. Esto es así porque en el primer caso se han reducido los falsos positivos a costa de aumentar los falsos negativos, mientras que en el segundo ha ocurrido justamente lo contario.

V. Árboles de decisión

Los árboles de decisión son una familia de modelos muy popular en aprendizaje automático para resolver problemas de clasificación. Una de sus principales ventajas es la facilidad con la que se pueden visualizar e interpretar sus resultados basándose en reglas, lo que permite no solo obtener un resultado, sino inspeccionar los motivos por los que se llega a una predicción. Por ejemplo, en un modelo de predicción en el que se busque identificar operaciones fraudulentas, se puede explorar el proceso lógico que utiliza el algoritmo y conocer las variables que llevan a una conclusión dada. En cuanto a sus desventajas, los árboles de decisión pueden crear modelos complejos que no generalicen bien los resultados, es decir, se llega fácilmente a modelos que sobreajustan el conjunto de entrenamiento. Esto es debido a que, si el entrenamiento no se realiza cuidadosamente, los modelos obtenidos pueden llegar a generar una regla específica para cada uno de los casos en el conjunto del entrenamiento. Este problema se puede mitigar fijando la profundidad a la que se puede llegar.

En la figura 2.22. se puede ver un ejemplo de la forma en la que funcionan los árboles de decisión. En esta figura se muestra un modelo que asigna la probabilidad a un cliente en función de tres variables: la edad, los ingresos y el estado civil. En este caso, comenzado por la edad, se puede comprobar y asignar la probabilidad utilizando una serie de reglas.

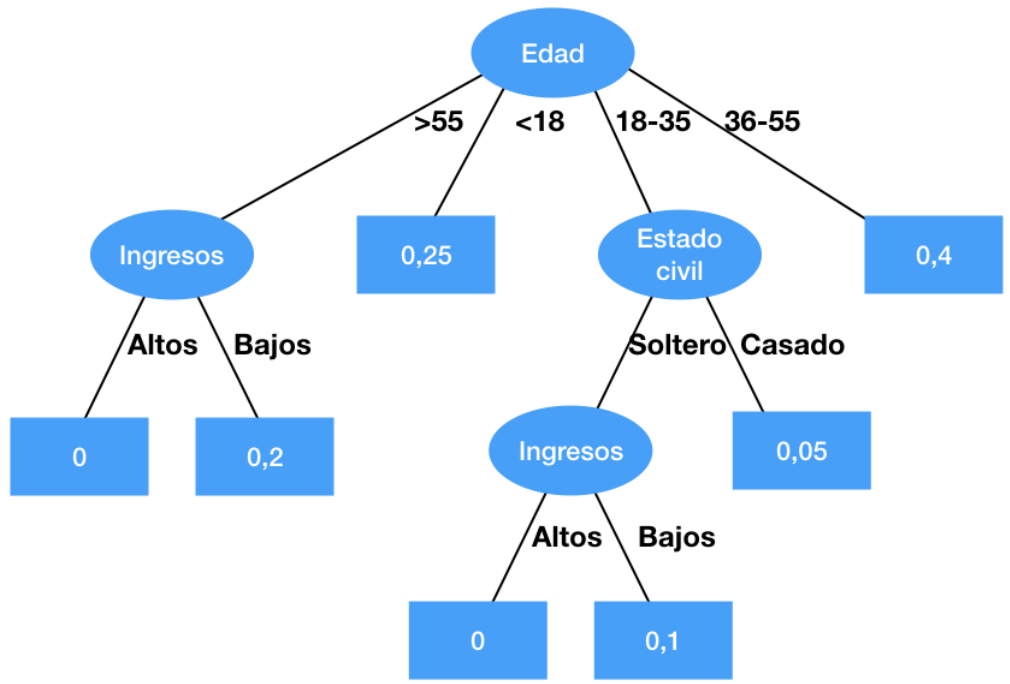


Figura 2.22. Ejemplo de un árbol de decisión. *Fuente:* elaboración propia.

Al igual que en los modelos de regresión, el entrenamiento de un árbol de decisión se basa en la búsqueda del valor extremo de una función. En este caso, la función objetivo es maximizar el valor de la ganancia de la información (IG, *Information Gain*) en cada una de las divisiones o nodos del árbol. Esta función se define como:

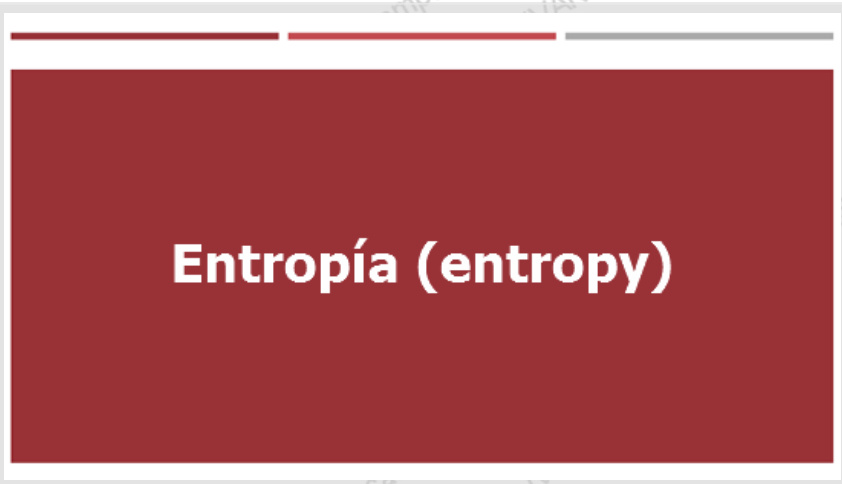
$$IG(x_p, y) = I(x_p) - \sum_{i=1}^{n} \frac{N_i}{N_p} I(x_i)$$

x_p es el conjunto del nodo padre, y es la variable dependiente, I es la medida de la impureza, i representa a los subnodos del nodo padre, N_p es el número total de registros en el nodo padre, N_i es el número total de registro en cada uno de ellos y x_i es el conjunto de datos en cada uno de los subnodos. Como se puede apreciar, el IG es sencillamente la diferencia de la impureza entre el nodo padre y los subnodos. En los árboles de decisión binarios, como los que se encuentran implementados en scikit-learn, se puede reducir el espacio de búsqueda, ya que solamente existen dos subnodos. Esto significa que la IG se puede escribir de la siguiente forma:

$$IG(x_p, y) = I(x_p) - \frac{N_T}{N_p}1(x_T) - \frac{N_F}{N_p}I(x_F)$$

En ella, se ha definido x_T como el subconjunto de casos positivos y x_F como el de casos negativos.

En los árboles de decisión binarios se suelen utilizar como funciones de impurezas uno de estos tres tipos:



La entropía se define como:

$$I_E(x) = -\sum_{i=1}^{c} p(i \mid x) \log_2(p(i \mid x))$$

$p(i \mid x)$ es el porcentaje de muestras que pertenecen a la clase i del nodo x y c representa el número de clases en los que se divide. Es importante destacar que esta definición solamente es válida cuando no existen clases vacías, es decir, cuando $p(i \mid x) \neq 0$. En los modelos de clasificación binarios, cuando se da esto, el valor de la entropía es 0 por definición.

El índice Gini se puede interpretar como un criterio para minimizar la probabilidad de error en la clasificación. Este se puede definir como:

$$I_G(x) = \sum_{i=1}^{c} p(i \mid x)(-p(i \mid x)) = 1 - \sum_{i=1}^{c} p(i \mid x)^2$$

Finalmente, el error de clasificación es uno menos el porcentaje máximo de muestras que pertenecen a la clase, es decir:

$$I_{CE}(x) = 1 - max\{p(i \mid x)\}$$

5.1. Implementación de árboles en scikit-learn

En scikit-learn, los árboles de decisión se implementan utilizando la clase DecisionTreeClassifier. En el momento de la creación del objeto es importante tener en cuenta un par de parámetros:

Es la máxima profundidad del árbol, en caso de que no se indique, el algoritmo no terminará de dividir hasta separar completamente las clases. Generalmente, si no se indica el valor, los modelos resultantes suelen mostrar sobreajuste.

Es el criterio utilizado para dividir los grupos.

También se puede fijar la semilla utilizando la opción random_state para garantizar que los resultados sean repetibles.

1

En la figura 2.23. se muestra un ejemplo de árbol para el conjunto de datos que se ha utilizado hasta ahora.

```
from sklearn.tree import DecisionTreeClassifier

dt_classifier = DecisionTreeClassifier(criterion = 'entropy',
                                       random_state = 1).fit(x_train, y_train)
y_pred        = dt_classifier.predict(x_train)

print 'Precisión:', accuracy_score(y_train, y_pred)
print 'Exactitud:', precision_score(y_train, y_pred)
print 'Exhaustividad:', recall_score(y_train, y_pred)
```

Figura 2.23. Implementación de un árbol de decisión.

Al ejecutar el código de la figura 2.23. se observa que el modelo es perfecto, del cual se obtienen los siguientes resultados:

- Precisión: 1.
- Exactitud: 1.
- Exhaustividad: 1.

Si se obtienen unos resultados tan buenos es fácil sospechar que el modelo se encuentra sobreajustado, lo que se puede verificar obteniendo las mismas métricas para el conjunto de entrenamiento:

- Precisión: 0.90.
- Exactitud: 0.92.
- Exhaustividad: 0.80.

2

Esto ha resultado así porque no se ha especificado un valor para el parámetro max_depth. Si se fija este en 4, como en el código de la figura 2.24. se obtienen otros resultados.

```
dt_classifier = DecisionTreeClassifier(max_depth = 4,
                                       criterion = 'entropy',
                                       random_state = 1)
dt_classifier.fit(x_train, y_train)
y_pred = dt_classifier.predict(x_train)

print 'Precisión:', accuracy_score(y_train, y_pred)
print 'Exactitud:', precision_score(y_train, y_pred)
print 'Exhaustividad:', recall_score(y_train, y_pred)
```

Figura 2.24. Implementación de un árbol de decisión fijando la profundidad en 4.

Concretamente, los resultados del código de la figura 2.24. son:

- Precisión: 0,92.
- Exactitud: 0,93.
- Exhaustividad: 0,91.

Y al validar este modelo en el conjunto de test se obtiene valores similares:

- Precisión: 0,91.
- Exactitud: 0,9.
- Exhaustividad: 0,92.

3

A partir esto se puede deducir que el posible sobreajuste ahora es menor que en el caso anterior. Finalmente, en la figura 2.25. se muestra la curva ROC de este último modelo para el cual el valor del área bajo la curva es 0,93.

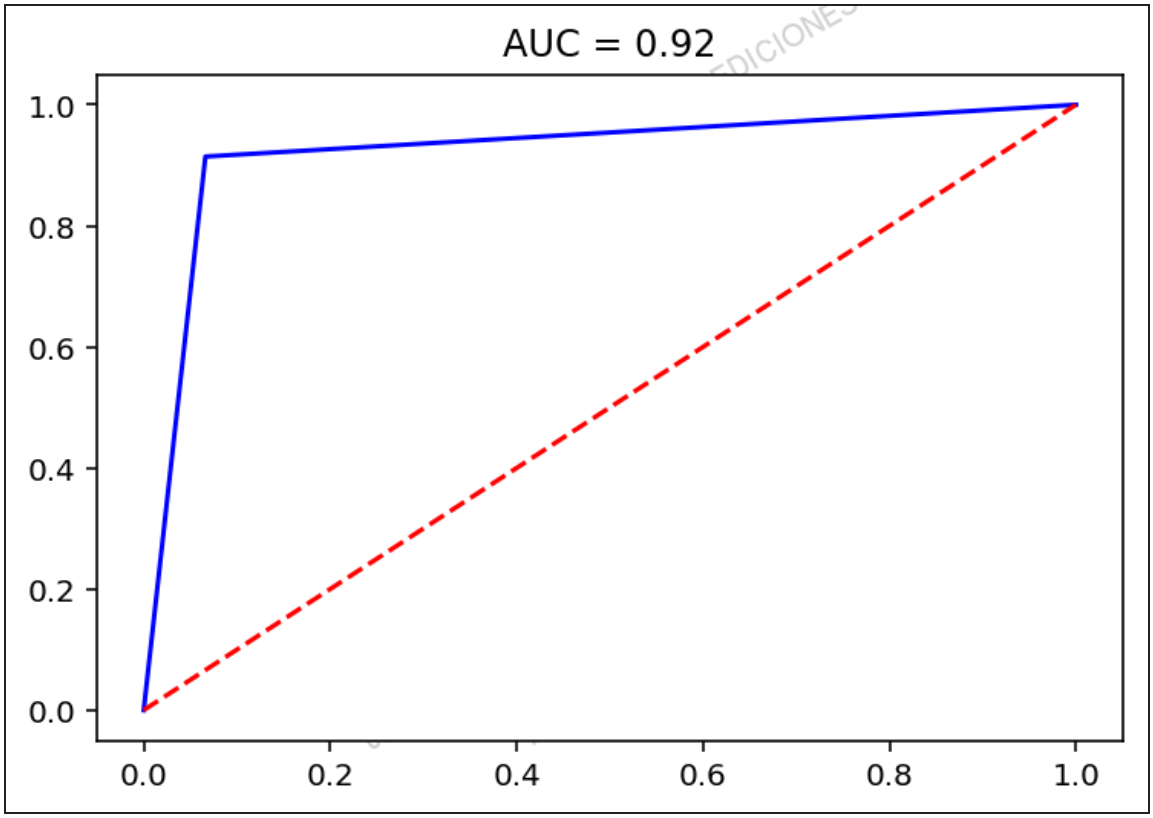


Figura 2.25. Curva ROC de un modelo basado en árboles de decisión. *Fuente:* elaboración propia

Una de las características de los árboles de decisión es que se puede obtener la importancia de cada una de las variables. Este valor se consigue mediante la propiedad feature_importances_ del clasificador. Este valor puede ser utilizado para seleccionar las características que se usan en el modelo. Por ejemplo, se puede eliminar la característica menos importante utilizando el código de la figura 2.26.

Las técnicas para la selección de las características en los modelos se estudiarán en profundidad en su propia unidad.

```
best_features = range(len(dt_classifier.feature_importances_))
best_features.pop(np.argmin(dt_classifier.feature_importances_))

dt_classifier = DecisionTreeClassifier(max_depth = 4,
                                       criterion = 'entropy',
                                       random_state = 1)
dt_classifier.fit(x_train[:, best_features], y_train)
y_pred        = dt_classifier.predict(x_train[:,best_features])

print 'Precisión:', accuracy_score(y_train, y_pred)
print 'Exactitud:', precision_score(y_train, y_pred)
print 'Exhaustividad:', recall_score(y_train, y_pred)
```

Figura 2.26. Eliminación de la característica menos importante de un árbol de decisión.

Los resultados del código de la figura 2.26. son similares a los obtenidos previamente, a pesar de utilizar una variable menos. Son los siguientes:

- Precisión: 0,92.
- Exactitud: 0,93.
- Exhaustividad: 0,91.

5.2. Random forests: combinación de árboles de decisión

Random forests es una técnica con la que se pretende unir varios árboles de decisión para crear un clasificador más preciso.

Un modelo random forest se estima utilizando el siguiente algoritmo:

1. Creación de un conjunto de muestras de los datos de entrenamiento. Esta selección se realiza aleatoriamente con reemplazo de los datos, es decir, los registros seleccionados pueden volver a serlo.
2. Creación de un modelo de árbol de decisión para cada uno de los conjuntos de entrenamiento seleccionados previamente.
3. Se pueden repetir los procesos 1 y 2 varias veces.
4. Agregación de los resultados mediante votación.

Los modelos de random forests son más estables que los árboles de decisión, aunque no son tan fáciles de interpretar. Esta mayor estabilidad se observa al comprobar que el sobreajuste de los modelos sin podar no son tan claros. A la hora de crear un modelo random forest es importante indicar el número de estimadores que se van a utilizar mediante el comando n_estimators.

1

Por ejemplo, en la figura 2.27. se muestra el código para implementar un modelo con random forests.

```
from sklearn.ensemble import RandomForestClassifier

rf_classifier = RandomForestClassifier(criterion = 'entropy',
                                       n_estimators = 10,
                                       random_state = 1).fit(x_train, y_train)
y_pred        = rf_classifier.predict(x_train)

print 'Precisión:', accuracy_score(y_train, y_pred)
print 'Exactitud:', precision_score(y_train, y_pred)
print 'Exhaustividad:', recall_score(y_train, y_pred)
```

Figura 2.27. Implementación de un clasificador random forest.

Los resultados que se obtienen con el código de la figura 2.27. son:

- Precisión: 0,99.
- Exactitud: 0,99.
- Exhaustividad: 0,98.

Al validar los resultados se observa que, en esta ocasión, también hay sobreajuste:

- Precisión: 0,90.
- Exactitud: 0,92.
- Exhaustividad: 0,90.

2

El sobreajuste, al igual que en el caso de los árboles de decisión, se puede evitar limitando la profundidad del árbol. Para esto se puede utilizar el código de la figura 2.28., en el que se ha añadido la opción max_depth = 4 en el constructor del clasificador.

```
rf_classifier = RandomForestClassifier(criterion = 'entropy',
                                       n_estimators = 10,
                                       max_depth = 4,
                                       random_state = 1)
rf_classifier.fit(x_train, y_train)
y_pred = rf_classifier.predict(x_train)

print 'Precisión:', accuracy_score(y_train, y_pred)
print 'Exactitud:', precision_score(y_train, y_pred)
print 'Exhaustividad:', recall_score(y_train, y_pred)
```

Figura 2.28. Implementación de un clasificador random forest limitando la profundidad de los árboles.

Los resultados que se obtienen son:

- Precisión: 0,89
- Exactitud: 0,92
- Exhaustividad: 0,85

Que son similares a los que se obtienen en el conjunto de test:

- Precisión: 0.88.
- Exactitud: 0.90.
- Exhaustividad: 0,87.

VI. Otros modelos supervisados

Una vez vistas en profundidad tres de las principales técnicas de aprendizaje supervisado disponibles en scikit-learn, se van a presentar por encima otras dos: las máquinas vector soporte y los clasificadores bayesianos ingenuos.

Las redes neuronales son otra técnica de interés, que se estudiarán más adelante en su propia unidad.

6.1. Máquinas vectores de soporte

Las Máquinas de Vector de Soporte (SVM, *Support Vector Machines*) son un conjunto de algoritmos en los que se utiliza un hiperplano para separar los puntos etiquetados con diferentes categorías, dejando los puntos de una categoría a un lado del plano y los de otra al otro. Un hiperplano es la generalización del concepto de plano que divide el espacio tridimensional en dos con cualquier número de dimensiones. Por ejemplo, en un espacio unidimensional (una recta) el hiperplano es un punto, mientras que en un espacio bidimensional (un plano) el hiperplano es una línea.

Lo que caracteriza a los SVM es la forma en la que obtienen el hiperplano óptimo que permita separar el espacio en dos. Se realiza buscando aquel que tenga la máxima distancia con respecto a los puntos que estén más cerca del mismo. Esto hace que a las SVM también se les pueda conocer como clasificadores de margen máximo.

En scikit-learn, el objeto que permite crear este tipo de modelos se llama SVM y su uso es similar al de los modelos que se han visto anteriormente. En la figura 2.29 se muestra un ejemplo de código necesario para su implementación.

```
from sklearn.svm import SVC

svm_classifier = SVC().fit(x_train, y_train)
y_pred         = svm_classifier.predict(x_train)

print 'Precisión:', accuracy_score(y_train, y_pred)
print 'Exactitud:', precision_score(y_train, y_pred)
print 'Exhaustividad:', recall_score(y_train, y_pred)
```

Figura 2.29. Implementación de un modelo de una máquina de vector de soporte.

El resultado del modelo basado en SVM es:

- Precisión: 0,92.
- Exactitud: 0,91.
- Exhaustividad: 0,92.

Y en el conjunto de test se obtienen unos resultados similares:

- Precisión: 0,92.
- Exactitud: 0,91.
- Exhaustividad: 0,95.

Lo que indica que no existe sobreajuste. Por otro lado, el área bajo la curva del modelo es de 0,92. La gráfica de la curva ROC para el conjunto de pruebas se muestra en la figura 2.30. En esta figura se puede ver que el modelo ajusta bastante bien, como era de esperar para este conjunto de datos.

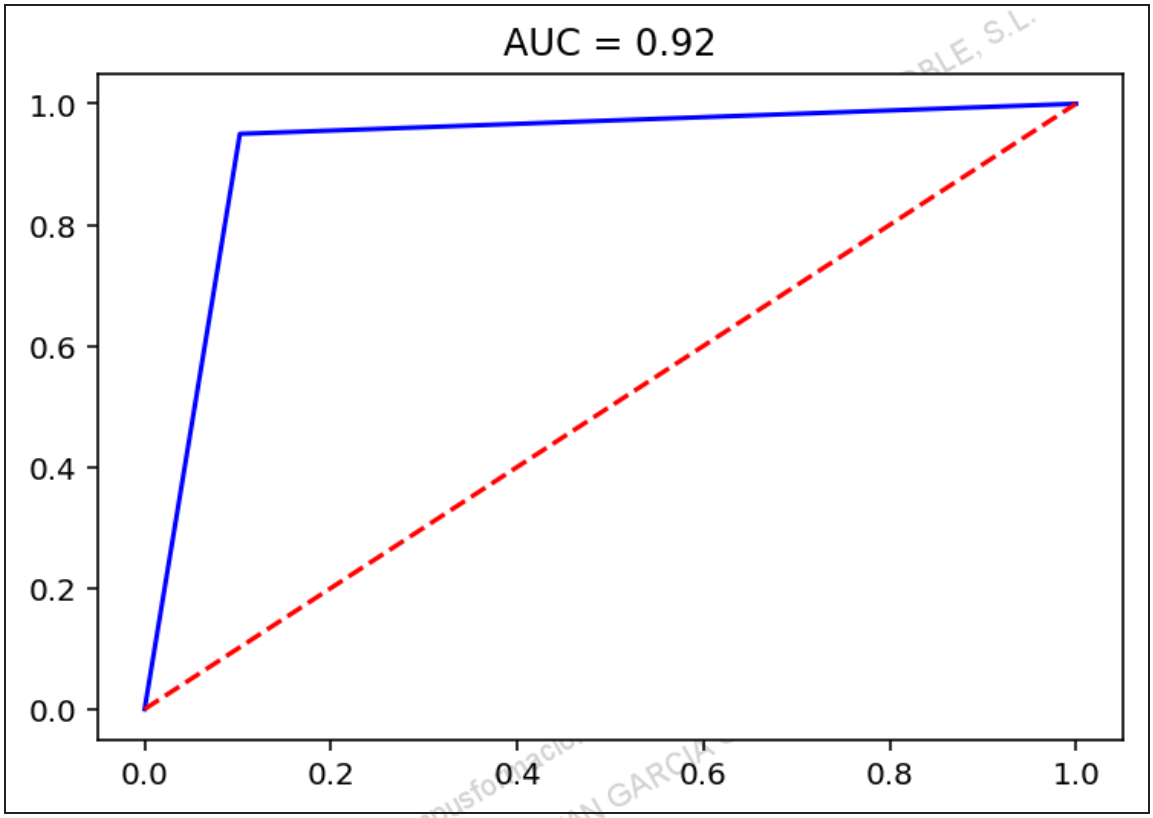


Figura 2.30. Curva ROC del modelo basado en SVM. *Fuente:* elaboración propia.

6.2. Naïve Bayes

Un clasificador Bayesiano Ingenuo (Naïve Bayes) es un clasificador probabilístico que se basa en el teorema de Bayes y en algunas hipótesis simplificadoras adicionales. El teorema de Bayes se define de la siguiente forma:

$$P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B)}$$

Y un clasificador bayesiano ingenuo se puede escribir como:

$$P(A \mid x_1, x_2, \cdots, x_n) = \frac{1}{P(\bar{x})} P(A) \prod_{1=1}^{n} P(x_i \mid A)$$

$P(\bar{x})$ es un factor de escala que depende de los valores de $x_1, x_2, \cdots, x_n$, es decir, una constante si son conocidos los valores de x_i.

La creación de un modelo bayesiano ingenuo requiere importar el objeto GaussianNB en Python. Su utilización es similar a la de los modelos estudiados en las secciones anteriores. En la figura 2.31. se muestra un ejemplo del código necesario para la creación de un modelo.

```
from sklearn import naive_bayes

nb_classifier = naive_bayes.GaussianNB().fit(x_train, y_train)
y_pred        = nb_classifier.predict(x_train)

print 'Precisión:', accuracy_score(y_train, y_pred)
print 'Exactitud:', precision_score(y_train, y_pred)
print 'Exhaustividad:', recall_score(y_train, y_pred)
```

Figura 2.31. Implementación de un modelo bayesiano ingenuo.

El resultado del modelo basado en un clasificador bayesiano ingenuo es:

- Precisión: 0,88.
- Exactitud: 0,87.
- Exhaustividad: 0,89.

Y en el conjunto de test se obtienen unos resultados similares:

- Precisión: 0,88.
- Exactitud: 0,87.
- Exhaustividad: 0,92.

En esta ocasión tampoco se observa sobreajuste en los resultados. La curva ROC para el conjunto de test se muestra en la figura 2.32., donde se puede apreciar que el área bajo la curva es de 0,88.

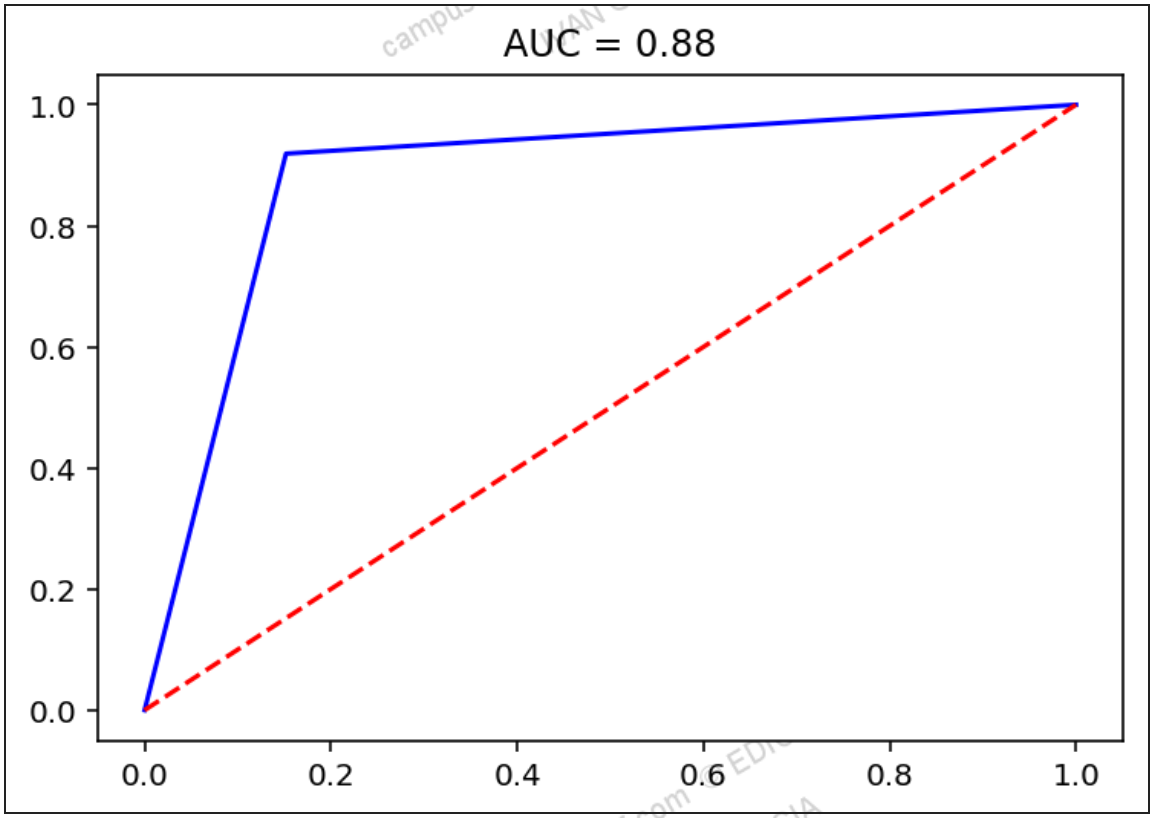


Figura 2.32. Curva ROC para el modelo bayesiano ingenuo.

VII. Resumen

En esta unidad se han visto las principales técnicas supervisadas para la creación de modelos disponibles en scikit-learn y se enumeran en la siguiente lista:

- Regresión lineal.
- Regresión logística.
- Árboles de decisión.
- Random forest.
- Máquinas de vector de soporte (SVM).
- Clasificadores bayesianos ingenuos (Naïve Bayes).

Mediante la regresión lineal se pueden crear modelos que capaces de reproducir valores continuos. El resto de modelos se utilizan para clasificar categorías.

Ejercicios

Caso práctico

Construir un modelo que estime la calidad de los vinos en función de propiedades físico-químicas.

Para esto se han de utilizar los datos contenidos en el archivowinequality-white.csv, donde se puede consultar la evaluación hecha por expertos en vino blanco y un conjunto de sus características como la acidez y el pH.

Se puede encontrar una descripción detallada de los campos de este archivo en la página web de The UCI Machine Learning Repository.

Solución

En el caso práctico se va a intentar encontrar un modelo que estime la calidad del vino blanco. Para esto es necesario importar un conjunto de datos que contiene distintas características físico-químicas de varios vinos y la calificación de expertos.

Para llevar a cabo el análisis, lo primero que hay que hacer es cargar los datos del archivo CSV con pandas. En el conjunto de datos importado se encuentran juntas la variable dependiente y las independientes, por lo que es necesario separarlas en dos conjuntos: el primeo x con las variables independientes y el segundo y con la variable dependiente.

Una vez separadas las variables independientes y la dependiente, se ha de crear un conjunto de entrenamiento y test utilizando la función train_test_split, la cual es necesaria para identificar la existencia del posible sobreajuste en el modelo. Una vez que se ha hecho esto, se puede crear el modelo y comprobar los resultados obtenidos en ambos conjuntos de datos.

El código necesario para realizar este proceso se muestra en la figura 2.33.

```
import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split

wine = pd.read_csv('winequality-white.csv', sep = ';')

# Separación de la variable objetivo y las explicativas
target = 'quality'
features = list(wine.columns)
features.remove('quality')

x = wine[features]
y = wine[target]

x_train, x_test, y_train, y_test = train_test_split(x, y)

# Creación de un modelo
model = LinearRegression()
model.fit(x_train, y_train)

predit_train = model.predict(x_train)
predit_test = model.predict(x_test)

# Evaluación de R2
print 'R2 en entrenamiento es: ', model.score(x_train, y_train)
print 'R2 en validación es: ', model.score(x_test, y_test)
```

Figura 2.33. Modelo de regresión de la calidad del vino.

Los resultados obtenidos son un R^2 de 0,289 en entrenamiento y de 0,253 en validación. Esto indica que posiblemente exista sobreajuste en el conjunto, por lo que es necesario revisar las características utilizadas. Los procedimientos para la selección de variables se verán en la unidad 4.

El código completo del caso se puede encontrar en elnotebook U3_caso.

Recursos

Enlaces de Interés

https://archive.ics.uci.edu/ml/datasets/Wine+Quality
https://archive.ics.uci.edu/ml/datasets/Wine%20Quality
The UCI Machine Learning Repository

Bibliografía

- ***Applied Logistic Regression.*** : Hosmer, D. W., Lemeshow, S. y Sturdivant, R. X. Wiley; 2103. Third Edition.
- ***Bootstrapping Machine Learning.*** : Dorard, L. [En línea] URL disponible en http://www.louisdorard.com/machine-learning-book/
- ***Designing Machine Learning Systems with Python.*** : Julian, D. Birmingham: Packt Publishing; 2016.
- ***Mastering Machine Learning With scikit-learn*** : Hackeling, G. Birmingham: Packt Publishing; 2014.
- ***Scikit-learn Cookbook.*** : Hauck, T. Birmingham: Packt Publishing; 2014.

Glosario.

- **Coeficiente de aprendizaje**: El valor que indica la velocidad a la que aprende el algoritmo de gradiente descendente.
- **Error de tipo I**: Se da cuando, en un modelo de clasificación, un caso negativo se identifica como positivo, también se conoce como Falso positivo.
- **Error de tipo II**: Se da cuando, en un modelo de clasificación, un caso positivo se identifica como negativo, también se conoce como Falso negativo.
- **Falsos negativos (FN)**: En un modelo de clasificación, son los eventos que, siendo positivos, son identificados como negativos por el modelo.
- **Falsos positivos (FP)**: En un modelo de clasificación, son los eventos que, siendo negativos, son identificados como positivos por el modelo.
- **Función de esfuerzo**: Es la función que define el error que comete un modelo supervisado sobre un conjunto de datos.
- **Grupo de entrenamiento**: En validación fuera de muestra, se llama de este modo al subgrupo de datos utilizados durante el proceso de entrenamiento.
- **Grupo de test**: En validación fuera de muestra, se llama de este modo al subgrupo de datos utilizados para validar el modelo.
- **Multicolinealidad**: Fuerte correlación entre las variables independientes de un modelo.

- **SVM (Support Vector Machines)**: Máquinas de vector de soporte.
- **Validación fuera de muestra**: Método consistente en dividir el conjunto de datos en dos para entrenar un modelo con datos diferentes a los utilizados para validar.
- **Variable dependiente**: Es la variable cuyo valor se pretende explicar en los modelos supervisados.
- **Variables independientes**: Son el conjunto de variables que se utilizan en los modelos supervisados para su construcción
- **Verdaderos negativos (TN)**: En un modelo de clasificación, son los eventos que son negativos y son identificados correctamente por el modelo.
- **Verdaderos positivos (TP)**: En un modelo de clasificación, son los eventos que son positivos y son identificados correctamente por el modelo.