Machine Learning: Regresión logística multiclase y aplicación para la clasificación de terrenos

David Tizne Ondiviela

Trabajo de fin de grado de Matemáticas
Universidad de Zaragoza

Director del trabajo: Ricardo López Ruiz
10 de julio de 2024

Resumen

El *Machine Learning* está formado por **un conjunto de técnicas matemáticas e informáticas** cuya finalidad es extraer información a partir de un conjunto de datos. Sin embargo, supone un **cambio de paradigma**, puesto que no extrae información con la finalidad de realizar estadísticas, sino **con el propósito de realizar predicciones sobre nuevos datos**.

El **objetivo** de este trabajo será **determinar qué tipo de planta es la más abundante en un conjunto de muestras** recogidas en diferentes terrenos de los Estados Unidos, con la **finalidad de comprobar que todo lo definido a lo largo de este trabajo es suficiente para realizar un algoritmo capaz de aprender de los datos**. Para ello, **estimaremos la probabilidad** de que cada tipo de planta sea la predominante basándonos en varios datos del conjunto, como pueden ser la elevación, la inclinación, la presencia o ausencia de determinadas sustancias, las distancias a fuentes de agua...

Para lograrlo, aplicaremos uno de los **métodos de clasificación** más conocidos dentro del conjunto de técnicas del *Machine Learning*: la **regresión logística**. Este método se basa en definir dos funciones: una primera función cuyo resultado es el vector de probabilidades estimadas y una segunda función (que depende de la primera) que mide el tamaño de los errores cometidos en la estimación de las probabilidades.

La finalidad de la segunda función mencionada, comúnmente conocida como **función de error** o **función de coste**, será la de tener una función sobre la cuál se tratará de hallar la combinación de parámetros que minimizan su valor. De esta forma, a menor error cometido en la estimación de probabilidades, mayor proporción de aciertos (en general).

Sin embargo, **hallar el mínimo de esta función no será fácil, puesto que suele depender de muchos parámetros**. Por tanto, se suelen utilizar otras técnicas que permiten hallar mínimos de forma computacionalmente aceptable. En nuestro caso, utilizaremos **descenso de gradiente**, siendo esta una técnica que permite **estimar estos parámetros mediante una actualización iterativa** de los mismos en la dirección del vector gradiente y el sentido contrario al mismo. Como veremos, el inconveniente de este método será que **no garantiza llegar a un mínimo global**, sino que puede caer en mínimos locales. No obstante, aceptaremos esta situación, puesto que el cálculo analítico del mínimo global no es aceptable en tiempo computacional.

En el Capítulo 2 explicaremos la definición de las funciones antes mencionadas y la técnica de descenso de gradiente, teniendo así el algoritmo de regresión logística en su versión más sencilla: aquella en la que cada muestra puede ser clasificada en una de las dos clases posibles. Posteriormente, en el Capitulo 3 **extenderemos este modelo** para generalizarlo al caso en el que cada muestra puede ser clasificada en una de K posibles clases.

Una vez tengamos explicada toda la teoría del modelo en su versión generalizada, procederemos a **calcular de manera explícita el vector gradiente del modelo en cada punto**. Este será utilizado posteriormente en el Capítulo 5 para implementar el modelo.

Con toda la teoría definida y todos los cálculos necesarios realizados, pasaremos entonces a la **implementación en *Python* todas las funciones que hayamos definido** durante los capítulos anteriores. Una vez implementadas, podremos **realizar distintos modelos** para probar qué parámetros influyen más y cuáles menos, así como estimar los valores de estos con los que conseguimos los mejores resultados. Nuestro **objetivo final** será conseguir un modelo que clasifique el mayor número posible de muestras de manera correcta.

Una vez hayamos probado a realizar varios ajustes con diferentes parámetros, procederemos a **eva-**

luar los modelos obtenidos con la finalidad de determinar cuál de todas las pruebas realizadas es el modelo que maximiza los aciertos conseguidos. Seleccionado este modelo, realizaremos sobre él más **métricas de rendimiento**, que habrán sido definidas previamente en el Capítulo 4 y nos permitirán tener información más concreta sobre qué clases son las que se aciertan más y cuales menos, con la finalidad de poder plantear mejoras en el modelo.

Finalmente, tras haber analizado nuestro mejor modelo con estas métricas, **plantearemos algunas posibles mejoras**. Entonces, ajustaremos unos últimos modelos con estas ideas y analizaremos los resultados obtenidos para comprobar si hemos mejorado el modelo o no.

Todos los procedimientos previos necesarios en las variables para poder ser utilizadas en el modelo, así como la implementación de las funciones, el análisis de los modelos y las posibles mejoras estarán realizados en el Capítulo 5.

Además, en los anexos estarán recogidos todos los códigos realizados para la implementación de la regresión logística multiclase.

Abstract

The *Machine Learning* is **a set of mathematical and computational techniques** whose purpose is to extract information from a dataset. However, it suppose a **paradigm change**, because it does not extract information with the purpose of doing statistics, but **with the objective of making predictions about new data**.

The **main objective** of this work is **determine which type of plant is the most abundant in a set of samples** taken in differents terrains in the United States, **with the final purpose of checking that everything that we have defined along this work is enough for doing an algorithm capable of learning from our data**. For doing that, **we will estimate the probability** for every type of plant to be the predominant based on some characteristics from the dataset, such as the elevation, the slope, the presence or absence of some substances, the distance to springs of water...

To achive our goal, we will use one of the most known **classification methods** within the set of *Machine Learning* techniques: the **logistic regression**. This method is based on the definition of two functions: a first function that give us a vector with the estimated probabilities and a second function (which depends on the first one) that measures the size of the errors that are made when we estimate these probabilities.

The purpose of the second function, commonly known as **error function** or **cost function**, will be having a function in which we will try to find the combination of parameters that minimize its value. In this way, the minimum error we make in the estimation of the probabilities, the biggest proportion of correct answers we will have (in general terms).

However, **finding the minimal of this function will not be easy, because it usually depends on lots of parameters**. For this reason, other techniques which allow us to find minimals in a computational acceptable times are commonly used. In our case, we will use **gradient descent**, which is a technique that allow us to **estimate the parameters with an iterative update** of themselves in the direction of the gradient and with the opposite way. As we will see later, the disadvantage of this method is that **it does not guarantee to reach the global minimum**, as it can end up in a local one. Despite this fact, we will accept this situation as the analitic calculation of the minimum is not acceptable in computational time.

In the Chapter 2 we will explain the definition of these functions that we mentioned above, as well as the technique of gradient descent, having in this way the most simple version of the logistic regression algorithm: the one in which every sample can be classified in one of two possible classes. Later, in the Chapter 3 **we will extend this model** making a generalization in which every sample can be classified in one of K possible classes.

Once we will have explained all the theory of the model in its generalizated version, **we will calculate in an explicit way the gradient vector of the model in every single point**. This vector will be used later in the Chapter 5 for implementing the model.

With all the theory defined and all the necessary calculations made, we will start the ***Python* implementation of all the functions that we have defined** in the previous chapters. Once we will have finished the implementation, we will be able **to make different models** in order to check which parameters are more influential, as well as to get the values that allow us to get better results. Our **final goal** will be to get a model that can classify the biggest number of samples in an appropiate way.

Once we will have try to do some fits with different parameters, we will start to **evaluate** the obtained models in order to determine which of them is the one that maximize the number of correct answers. With this model selected, we will calculate more **efficiency metrics** of it, that we will have defined previously

in the Chapter 4 and they will allow us to have more specific information about which classes are the ones that we fit better and which we fit worse, all of that with the purpose of considering improvements for our model.

Finally, after having analized our best model with these metrics, **we will consider some possible improvements**. Then, we will fit some models with these ideas and we will analize the obtained results to check if we have an improvement or not.

All the previous procedures which are neccesary in our variables to prepare them to be use in our models, as well as the implementation of the functions, the models analysis and the possible improvements are in the Chapter 5.

Therefore, in the appendixes we can find the collection of all the codes made for the implementation of the multiclass logistic regression.

Índice general

Capítulo 1

Introducción al Machine Learning

El *Machine Learning* comprende una gran variedad de técnicas matemáticas e informáticas, donde todas ellas tratan de **extraer información de un conjunto de datos** con la finalidad de **predecir un comportamiento** sobre nuevos datos a partir de la información disponible. Por tanto, supone un **cambio de paradigma**: los algoritmos tradicionales tomaban un conjunto de datos y, mediante una serie de reglas, determinaban la salida correspondiente. En cambio, los algoritmos de *Machine Learning* reciben como entrada los datos y las salidas y calculan a partir de estos la **regla** que se les aplica.

Según cómo se entrene a estos algoritmos, se clasifican principalmente en dos grupos:

- **Supervisados:** Aquellos algoritmos donde los datos de entrada tienen asociada la salida correspondiente. Comprende los algoritmos de regresión y de clasificación entre otros.

- **No supervisados:** Algoritmos cuyos datos de entrada no tienen salida asociada. Este conjunto está formado por los algoritmos que buscan patrones, estructuras, tendencias... como pueden ser los algoritmos de *clustering*.

En este trabajo nos centraremos en el algoritmo supervisado de clasificación llamado **regresión logística**, que explicaremos en detalle en el Capítulo 2. Sin embargo, antes de comenzar con este método realizamos una introducción a los algoritmos supervisados, tanto de clasificación como de regresión.

1.1. Algoritmos supervisados

Los algoritmos supervisados comprenden todas aquellas técnicas y modelos en los que el conjunto de datos utilizado para entrenar al algoritmo lleva asociado, para cada muestra, **la salida que se espera que el modelo dé como resultado**. Es el caso de los algoritmos de clasificación y de regresión.

1.1.1. Algoritmos de clasificación

Supongamos que tenemos K grupos distintos. Queremos entonces determinar, para cada muestra, **en cuál o cuáles de esos grupos encajaría mejor** según los datos de los que disponemos. Los algoritmos de clasificación son precisamente aquellos que tratan de predecir este tipo de cuestiones.

Un **ejemplo clásico** de estos problemas es el siguiente: Dada una opinión sobre una película y métricas obtenidas a partir de esta (como pueden ser el número de palabras, el número de signos de interrogación, la existencia o no de signos de exclamación, etc.) clasificar la opinión como desfavorable, neutra o favorable hacia la película.

Los **algoritmos de clasificación más conocidos** son los árboles de decisión, la máquina de soporte vectorial y la regresión logística.

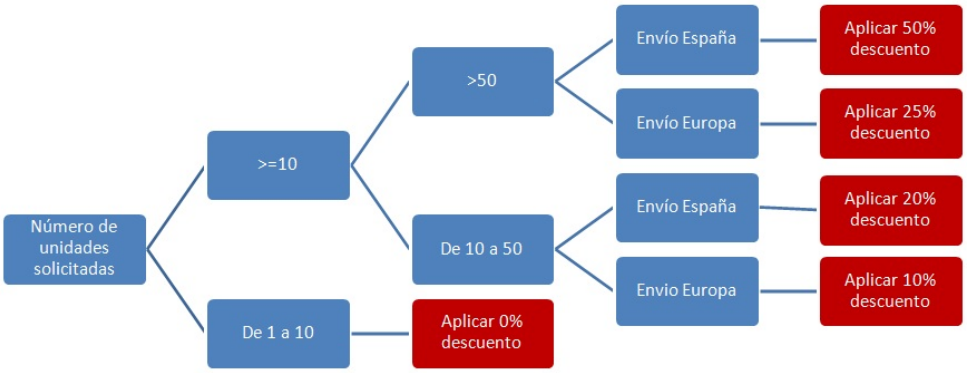

Figura 1.1: Ejemplo de árbol de decisión [2].

Árboles de decisión

Los árboles de decisión [1] son una técnica cuya idea base consiste en ir realizando **divisiones del espacio** en función de algunas características hasta ser capaces de clasificar las entradas según la región del espacio a la que pertenezcan. Al final del proceso tendremos un árbol cuyos nodos internos serán nodos de decisión y cuyas hojas serán las soluciones predichas. Un ejemplo de estos árboles puede verse en la Figura 1.1.

Hay diversas técnicas para decidir qué división hacer en cada iteración. La más común es mediante la **entropía** y la **ganancia de información**. Denotamos por c el número de clases en un conjunto cualquiera S y llamamos p_i a la proporción de muestras en la clase i del conjunto S. Así, tenemos que la fórmula de la entropía es la mostrada en la ecuación (1.1). Esta indica cómo de mezclados se encuentran los datos en el conjunto S, siendo 0 máximo orden y 1 máximo desorden.

$$H(S) = -\sum_{i=1}^{c} p_i log_c(p_i) \in [0,1] \tag{1.1}$$

Así, la **ganancia de información** se define como la diferencia entre la entropía antes de realizar la división y el promedio ponderado de las entropías después de realizar la división según el atributo A.

$$Ganancia(A,S) = H(S) - \sum_{j=1}^{d} \frac{|S_j|}{|S|} H(S_j) \tag{1.2}$$

Donde d es el número de subconjuntos de S totales después de realizar la nueva división, S_j cada uno de los subconjuntos, $|S_j|$ el número de individuos en el conjunto S_j, $|S|$ el número de individuos totales y $H(S_j)$ la entropía del subconjunto S_j. Así, la partición a elegir en cada paso será aquella que maximice la ganancia de información.

Máquina de soporte vectorial

Estos modelos tratan de **encontrar el hiperplano que mejor separa la muestra**, de tal forma que las entradas sean clasificadas en función de cuál sea la región del espacio al que pertenecen. Aunque son similares a los árboles de decisión, este método consiste en una **generalización** de los mismos. Esto se debe a que las divisiones realizadas en los primeros son siempre paralelas a los ejes, mientras que en las máquinas de soporte vectorial no es necesario.

El principal objetivo en este método es **encontrar la dirección de este hiperplano**. Una vez hallada, tratan de maximizar la distancia existente entre este y los puntos más cercanos de cada subespacio.

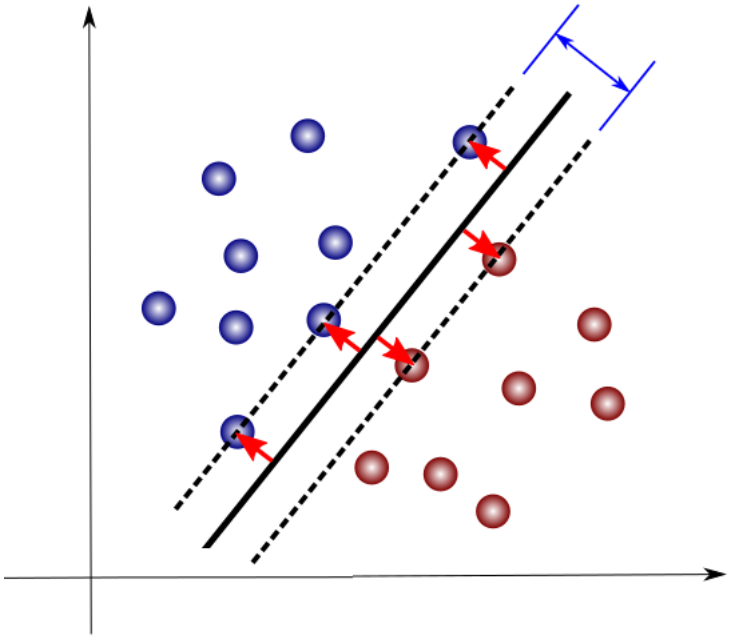

Figura 1.2: Ejemplo del método de máquina de soporte vectorial [3].

Puede verse una imagen ilustrativa de esta técnica en la Figura 1.2, donde los colores azul y rojo indican las dos clases y la línea negra es el hiperplano que mejor las separa.

1.1.2. Algoritmos de regresión

Los algoritmos de regresión son aquellos que tratan de predecir el valor de una variable **continua** en función del valor de una o más variables explicativas. Un **ejemplo clásico** de estos métodos es predecir el peso de una persona en función de su altura.

Se pueden realizar regresiones de varios tipos dependiendo de con qué funciones busquemos explicar los datos. Sin embargo, el método más extendido es el de **regresión lineal**.

Regresión lineal

Partimos de un conjunto de datos

$$\left\{(x_i, y_i) \in \mathbb{R}^{(\rho+1)} \times \mathbb{R} \mid i \in \{1, \cdots, N\}\right\}$$

Donde N es el número de observaciones, $x_i = (1, x_{i1}, \cdots, x_{i\rho})$ es un vector de dimensión $(\rho + 1)$ formado por un 1 del término independiente seguido de ρ variables explicativas, e y_i es la respuesta asociada a esas variables. Entonces, buscamos explicar la variable y_i de forma lineal en las x_i. Es decir, buscamos unos coeficientes $\beta_0, \cdots, \beta_\rho$ de tal forma que, si denotamos

$$Y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix} \qquad X = \begin{bmatrix} 1 & x_{11} & \cdots & x_{1\rho} \\ 1 & x_{21} & \cdots & x_{2\rho} \\ \vdots & & & \\ 1 & x_{N1} & \cdots & x_{N\rho} \end{bmatrix} \qquad \beta = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_\rho \end{bmatrix} \qquad \varepsilon = N_N(0, \sigma^2 I_N) = \begin{bmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \vdots \\ \varepsilon_N \end{bmatrix}$$

Tengamos:

$$Y = X\beta + \varepsilon \tag{1.3}$$

Donde ε representa el error en la aproximación y se asume una distribución normal de media cero y desviación típica σ. Por tanto, si lo miramos componente a componente, tenemos que la variable explicada es lineal en cuanto a las variables explicativas:

$$y_i = \beta_0 + \beta_1 x_{i1} + \cdots + \beta_\rho x_{i\rho} + \varepsilon_i$$

Sin embargo, cabe destacar que **estas variables pueden haber sido transformadas** con respecto a las variables originales. Un ejemplo común de ello es tomar, por ejemplo, el logaritmo de los datos de entrada.

Ahora, el objetivo será hallar una estimación de los coeficientes que **minimice el error cometido** en la aproximación. Si denotamos por $\hat{\beta}_i$ a nuestra estimación del parámetro β_i, tenemos que la estimación del valor de y_i será:

$$\hat{y}_i = \hat{\beta}_0 + \hat{\beta}_1 x_{i1} + \cdots + \hat{\beta}_\rho x_{i\rho}$$

Entonces, para estimar los parámetros, utilizaremos el **método de mínimos cuadrados**. Por tanto, definimos la función $\phi : \mathbb{R}^{p+1} \to \mathbb{R}$ de la siguiente manera:

$$\phi(\beta) = (Y - X\beta)^T (Y - X\beta)$$

Es decir, por (1.3) la función $\phi(\beta)$ es la suma de errores al cuadrado en caso de que β fuese el vector de coeficientes del modelo. Derivando con respecto a β e igualando a cero para hallar el mínimo valor de la función, se tiene que la mejor aproximación de los parámetros es:

$$\hat{\beta} = (X^T X)^{-1} X^T Y$$

Como hemos comentado anteriormente, este tipo de aproximaciones pueden funcionar bien cuando la variable de respuesta es **continua**. Sin embargo, cuando queremos aproximar una salida **discreta**, los métodos de regresión presentarán grandes errores (en general) debido a que el modelo dará como resultado una función cuya imagen estará en $\mathbb{R}$ y no en nuestro conjunto discreto de posibles respuestas. Por ello, en estos casos se utiliza el **Modelo de Regresión Logística**, que procedemos a explicar en el siguiente capítulo.

Capítulo 2

Regresión Logística

Como hemos comentado en el capítulo anterior, los modelos de regresión no son adecuados para resolver problemas de clasificación, puesto que el conjunto de datos de salida (que en este caso serán las posibles clases) es un conjunto de datos discreto. Sin embargo, **estos modelos pueden adaptarse a las características particulares de estos problemas**.

Comenzamos indicando las asunciones que se realizan en estos modelos. **En la regresión logística se asume** [4]:

- Variable de respuesta binaria.
- Observaciones independientes.
- Ausencia de valores atípicos extremos.
- Ausencia de multicolinealidad.

Procedemos entonces a realizar una formulación matemática del modelo.

2.1. Formulación del modelo

Supongamos que tenemos un conjunto de datos que queremos clasificar en dos clases, denotadas por los dígitos 0 y 1. Es decir, ahora $y_i \in \{0,1\} \forall i \in \{1,\cdots,N\}$. Entonces, buscamos estimar la probabilidad de pertenecer a una de las dos clases. En particular, la de pertenecer a la clase denotada por 1. Así, sea:

$$p(x) = P(y = 1 \mid x)$$

La función más habitual para realizar esta aproximación es la **función logística** (también conocida como **función sigmoidea**) denotada por $\sigma : \mathbb{R} \to (0,1)$ y que se define de la siguiente forma:

$$\sigma(t) = \frac{1}{1+e^{-t}}$$

Así, consideramos t como una función de los datos de entrada del problema. En particular, supondremos t lineal. Siguiendo la notación del apartado 1.1.2, asumiremos:

$$t(x_i) = x_i\beta = \sum_{j\in\{0,\cdots,\rho\}} \beta_j x_{ij}$$

Entonces, denotando por S el conjunto de observaciones de la muestra, podemos definir la función $\hat{p} : S \to (0,1)$ de la siguiente manera: Dado $x_i \in S$:

$$\hat{p}(x_i) = \sigma(t(x_i)) = \frac{1}{1+exp(-t(x_i))} = \frac{1}{1+exp(-\sum_{j\in\{0,\cdots,\rho\}} \beta_j x_{ij})}$$

A partir de esta expresión, tenemos:

$$\hat{p}(x_i) = \frac{1}{1+exp(-t(x_i))} \longleftrightarrow 1+exp(-t(x_i)) = \frac{1}{\hat{p}(x_i)} \longleftrightarrow exp(-t(x_i)) = \frac{1-\hat{p}(x_i)}{\hat{p}(x_i)} \tag{2.1}$$

Finalmente, tomando logaritmos:

$$\log\left(\frac{\hat{p}(x_i)}{1-\hat{p}(x_i)}\right) = t(x_i) = \sum_{j\in\{0,\cdots,\rho\}} \beta_j x_{ij} \tag{2.2}$$

En la siguiente sección daremos unas definiciones con el objetivo de interpretar la igualdad que acabamos de obtener.

2.2. Interpretación de parámetros

Comenzamos la sección dando unas definiciones sencillas.

Definición 1. Llamamos **odd** [5] de un suceso al cociente entre la probabilidad de que ocurra el suceso y la probabilidad de su complementario. Es decir, si denotamos por p la probabilidad de un suceso, $p \in (0,1)$, su odd es:

$$O = \frac{p}{1-p}$$

Vemos que el odd pertenece al intervalo $(0,+\infty)$. La interpretación de este cociente es **cuántas veces es más probable** que ocurra un suceso a que ocurra su complementario.

Definición 2. Supongamos que fragmentamos el espacio muestral en dos grupos en función de alguna variable del estudio. Llamamos **odd ratio** de un suceso al cociente entre el odd del suceso en un grupo y el odd del suceso en el otro grupo.

Es decir, si denotamos G_1, G_2 los grupos y p_{G_1}, p_{G_2} la probabilidad del suceso en cada grupo, $p_{G_1}, p_{G_2} \in (0,1)$, entonces el odd ratio es:

$$OR = \frac{\frac{p_{G_1}}{1-p_{G_1}}}{\frac{p_{G_2}}{1-p_{G_2}}}$$

El odd ratio, al igual que el odd, está en el intervalo $(0,+\infty)$. Su valor nos da una **medida del grado de asociación** del suceso con respecto a la variable utilizada para crear los dos grupos:

- Si $OR \in (0,1)$ la probabilidad de que ocurra el suceso es mayor en el grupo G_2.
- Si $OR = 1$ el suceso no está relacionado con la variable.
- Si $OR \in (1,+\infty)$ la probabilidad de que ocurra el suceso es mayor en el grupo G_1.

Una vez vistas estas definiciones, vemos en la expresión (2.2) que **los parámetros de los modelos de regresión logística son los coeficientes del hiperplano que contiene el logaritmo de los odds**. Ahora, por la expresión (2.1) vemos que, si denotamos por $O(x_i)$ el odd de la probabilidad asociada a las variables x_i, tenemos:

$$O(x_i) = \frac{p(x_i)}{1-p(x_i)} = exp\left(t(x_i)\right) = exp\left(\sum_{j\in\{0,\cdots,\rho\}} \beta_j x_{ij}\right) = \prod_{j=0}^{\rho} exp(\beta_j x_{ij})$$

Así, supongamos que tenemos dos vectores de variables. Denotamos a estos $x_1 = (x_{11}, x_{12}, \cdots, x_{1\rho})$ y $x_2 = (x_{21}, x_{22}, \cdots, x_{2\rho})$. Supongamos que todas componentes son iguales salvo la componente h, en la que $x_{2h} = x_{1h} + 1$. Entonces:

$$OR = \frac{O(x_2)}{O(x_1)} = \frac{\prod_{j=0}^{p} exp(\beta_j x_{2j})}{\prod_{j=0}^{p} exp(\beta_j x_{1j})} = exp(\beta_h)$$

Es decir, $exp(\beta_h)$ es el odds ratio de la variable x_h y es una medida de cómo crece o decrece la probabilidad cuando esta variable crece en una unidad.

2.3. Ajuste del modelo

Al igual que en la sección 1.1.2 se ha ajustado el modelo minimizando una función que habíamos definido, el ajuste de los parámetros en regresión logística se realiza de manera similar.

Sin embargo, **el uso de la función de error de mínimos cuadrados ahora no es adecuada** debido a que nuestra salida se encuentra en el intervalo (0,1), de manera que, al elevar las diferencias al cuadrado, los valores disminuyen en vez de aumentar [6].

Por esta razón, en el caso de regresión logística, se parte de la ecuación de verosimilitud [7]. Como la variable de respuesta es dicotómica y asumimos que son independientes, la distribución que estas siguen es una binomial. Así, la ecuación de verosimilitud es:

$$\prod_{i=1}^{N} \hat{p}(x_i)^{y_i}(1-\hat{p}(x_i))^{1-y_i} \in [0,1]$$

Por tanto, el objetivo sería **maximizar esta función**. Sin embargo, maximizar esta función es equivalente a maximizar el logaritmo de la misma, puesto que el logaritmo es una función monótona estrictamente creciente. Así:

$$\log\left(\prod_{i=1}^{N} \hat{p}(x_i)^{y_i}(1-\hat{p}(x_i))^{1-y_i}\right) = \sum_{i=1}^{N} \log\left(\hat{p}(x_i)^{y_i}(1-\hat{p}(x_i))^{1-y_i}\right) =$$

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

Finalmente, por razones históricas, en los modelos de *Machine Learning* siempre se trata de minimizar la función que nos permite ajustar el modelo. Por esta razón, en lugar de maximizar la expresión anterior **se tratará de minimizar la siguiente función**:

$$-\frac{1}{N}\sum_{i=1}^{N} (y_i \log(\hat{p}(x_i)) + (1-y_i)\log(1-\hat{p}(x_i))) \tag{2.3}$$

Donde se ha introducido la constante $\frac{1}{N}$ con el fin de minimizar el promedio del logaritmo de los errores. La razón de la inclusión de este término es **quitar dependencia con respecto al número de muestras utilizadas** puesto que, en caso contrario, se tendría que cuantas menos muestras se utilizaran, menos términos se sumarían y el valor de la función sería menor (en general). Además, notemos que **el producto por una constante mayor que cero no afecta a los puntos donde se alcanzan los mínimos**.

De esta forma, disponemos de una función sobre la cuál se puede aplicar el método de **descenso de gradiente** que veremos a continuación para encontrar su mínimo.

2.4. Descenso de gradiente

Comenzamos definiendo el gradiente de una función.

Definición 3. Dada una función $f(x_1, x_2, \cdots, x_n)$, llamamos **gradiente** de la función al vector

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \cdots, \frac{\partial f}{\partial x_n}\right)$$

Ahora, vemos la propiedad del gradiente que más nos interesa para optimizar nuestros modelos [8]:

Proposición 2.1. *La dirección de máximo crecimiento del valor de una función viene dada por la dirección y el sentido de su gradiente.*

Demostración. *Sea $f : U \to \mathbb{R}$ una función de clase $C^1(U)$ con $U \subset \mathbb{R}^n$. Sea u un vector unitario y sea $\boldsymbol{x} \in U$. Denotamos por $D_u f(\boldsymbol{x})$ la derivada direccional de f en la dirección de u. Tenemos:*

$$D_u f(\boldsymbol{x}) = < \nabla f, u >$$

Por las propiedades del producto escalar, denotando $\alpha_{u,v}$ el ángulo entre u y v, tenemos:

$$cos(\alpha_{u,v}) = \frac{< u, v >}{||u||||v||}$$

Por tanto, despejando el producto escalar y sustituyendo en la primera expresión, como el vector u es unitario, obtenemos:

$$D_u f(\boldsymbol{x}) = ||u||||\nabla f||cos(\alpha_{u,\nabla f}) = ||\nabla f||cos(\alpha_{u,\nabla f})$$

Así, si fijamos el punto $\boldsymbol{x} \in U$, tenemos que el máximo valor de la derivada direccional se obtiene cuando $cos(\alpha_{u,\nabla f}) = 1$. Es decir, cuando $\alpha_{u,\nabla f} = 0$ y, por tanto, cuando el vector direccional u tiene la dirección y el sentido del gradiente. □

Corolario 2.2. *La dirección de máximo decrecimiento del valor de una función viene dada por la dirección del vector unitario $\frac{\nabla f}{||\nabla f||}$ en el sentido opuesto al mismo.*

Demostración. *Análogamente a la demostración anterior. La dirección de mínimo valor de la derivada direccional se da cuando $\alpha_{u,\nabla f} = \pi$. Esto se da cuando el vector u tiene la dirección y el sentido de $-\nabla f$. Así, normalizando tenemos $u = -\frac{\nabla f}{||\nabla f||}$* □

Por tanto, si queremos minimizar la función deberemos ir desplazando los parámetros en la dirección y sentido dados por $-\frac{\nabla f}{||\nabla f||}$.

Sin embargo, cabe destacar que, aunque el algoritmo encuentra un mínimo de la función, este **puede no ser el mínimo global**, como es el ejemplo de la Figura 2.1: Si empezamos en el punto A el algoritmo acabará en el punto C (**mínimo local**), mientras que si comenzamos en el punto B acabaremos en el D (**mínimo global**).

La presencia de estas situaciones puede provocar una oscilación entre varios mínimos, dando como resultado la no convergencia del algoritmo. Es por ello que se introduce la **tasa de aprendizaje**.

2.4.1. Tasa de aprendizaje

Definición 4. Llamamos **tasa de aprendizaje** [9] al factor que determina cómo de grandes son los pasos en cada iteración del algoritmo. Lo denotaremos con α.

Así, si denotamos por F a la función de coste del modelo y β^i a los parámetros en la iteración i-ésima, tenemos que la **ecuación iterativa** que va determinando la evolución de los parámetros es:

$$\beta^{i+1} = \beta^i - \alpha \frac{\nabla F(\beta^i)}{||\nabla F(\beta^i)||} \tag{2.4}$$

Además, la elección del parámetro α es importante, puesto que tomarlo demasiado grande puede causar la no convergencia del algoritmo, mientras que tomarlo demasiado pequeño aumenta en gran medida el tiempo de convergencia. La representación de estas situaciones puede verse en la Figura 2.2.

Pese a la importancia de la elección de la tasa de aprendizaje, **no hay reglas que determinen qué valores concretos debe tomar este parámetro**. En general, el parámetro se halla en el intervalo $(0, 1)$ y se encuentra más próximo al 0 que al 1. Además, hay varios planteamientos a la hora de elegirlo, como

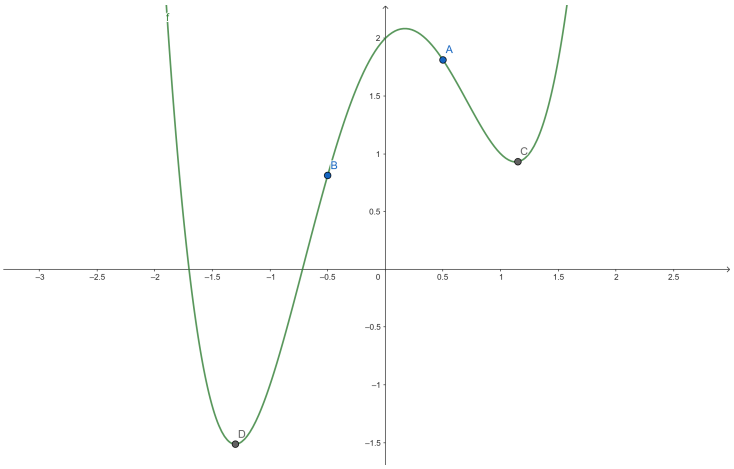

Figura 2.1: Ejemplo de varios mínimos con la función $x^4 - 3x^2 + x + 2$

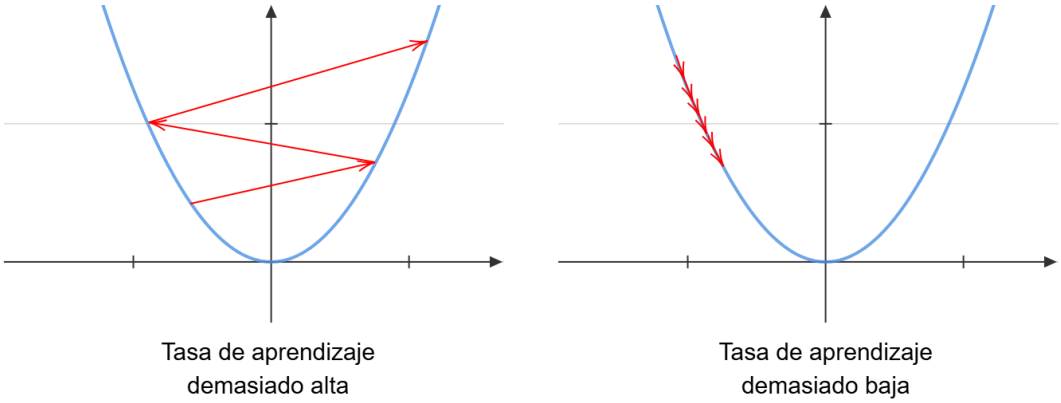

Figura 2.2: A la izquierda un ejemplo de tasa de aprendizaje demasiado alta. A la derecha, un ejemplo con tasa demasiado baja.

puede ser hacer varias pruebas con valores constantes y ver cuál funciona mejor o tomarlo al principio más grande e ir reduciéndolo conforme avanzan las iteraciones.

El algoritmo de descenso de gradiente termina cuando se cumple el **criterio de parada**, como pueden ser cotas en el número de iteraciones o en el aprendizaje realizado:

$$||\beta^{i+1} - \beta^i||_\infty < K$$

Con $K \in \mathbb{R}$ una constante.

2.4.2. Métodos

Cuando se trata de minimizar una función mediante descenso de gradiente, el proceso se puede hacer de tres formas distintas:

- **Descenso estocástico:** Actualizar el valor de los parámetros cada vez que se evalúa el gradiente en un punto concreto de la muestra.
- **Descenso por lotes:** Calcular el gradiente para todos los puntos del conjunto de entrenamiento y hallar el promedio de todos estos. Una vez hecho esto, actualizar los parámetros según la dirección del vector resultado.
- **Descenso por mini-lotes:** Similar al anterior, tomando subconjuntos de un tamaño determinado del conjunto de entrenamiento.

La elección de qué técnica utilizar dependerá de si buscamos **velocidad de convergencia** (caso de descenso de gradiente estocástico), **eficiencia computacional** (caso de descenso de gradiente por lotes) o un **equilibrio entre ambos**.

Con todo lo definido en este capítulos estaríamos en disposición de realizar un modelo de regresión logística sencillo. En el siguiente capítulo extenderemos el modelo que acabamos de definir con la intención de generalizarlo al caso en que el conjunto de posibles clases tiene más de dos elementos.

Capítulo 3

Regresión Logística Multiclase

Consideramos ahora el problema en el que **el conjunto de posibles clases no tiene dos elementos, sino K**. Es decir, ahora $y_i \in \{1,2,\cdots,K\}$. Un ejemplo de estos problemas sería clasificar una opinión como buena, neutral o mala en función de su contenido.

3.1. Formulación del modelo

Buscamos entonces construir una función [10] $h : S \to [0,1]^K$ con S el conjunto de entradas posibles de tal forma que, dado $x \in S$:

$$h(x) = \begin{bmatrix} P(y=1|x) \\ \vdots \\ P(y=K|x) \end{bmatrix} \tag{3.1}$$

Para ello, se asume que las K clases no disponen de un orden.

Así, análogamente al uso de la función sigmoidea en el capítulo anterior, ahora definiremos la función softmax con un propósito similar.

Definición 5. Sea $v = (v_1, v_2, \cdots, v_n) \in \mathbb{R}^n$. Llamamos función **softmax** a la función de $\mathbb{R}^n \to \mathbb{R}^n$ definida como:

$$softmax(v) = \frac{1}{\sum_{k=1}^{n} e^{v_k}} \begin{bmatrix} e^{v_1} \\ \vdots \\ e^{v_n} \end{bmatrix}$$

Vemos que $softmax(v) \in (0,1)^n \ \forall v \in \mathbb{R}^n$ y se verifica trivialmente que la suma de las componentes es 1. Así, sea $w \in \mathbb{R}^{K\times(\rho+1)}$ una matriz de pesos y sea w_i la i-ésima fila de esta matriz. Asumiremos entonces que la función h definida en la ecuación (3.1) es de la forma:

$$h_w(x) = softmax([w_1 x, w_2 x, \cdots, w_K x]) = \frac{1}{\sum_{i=1}^{K} e^{w_i x}} \begin{bmatrix} e^{w_1 x} \\ \vdots \\ e^{w_K x} \end{bmatrix} = \begin{bmatrix} \hat{p}_1(x;w) \\ \vdots \\ \hat{p}_K(x;w) \end{bmatrix} \tag{3.2}$$

Donde $\hat{p}_k(x;w)$ es la probabilidad estimada de pertenencia a la clase k con la entrada x dados los pesos w. En particular:

$$\hat{p}_k(x;w) = \frac{e^{w_k x}}{\sum_{i=1}^{K} e^{w_i x}}$$

Vemos ahora una proposición que nos será útil para demostrar que en el caso $K = 2$ el modelo propuesto ahora coincide con el propuesto en el capítulo anterior.

Lema 3.1. *La función h_w es invariante ante traslaciones constantes de todas las filas de pesos.*

Demostración. *Sea $u \in \mathbb{R}^{(\rho+1)}$ fijo y supongamos $w'_k = w_k + u \; \forall k \in \{1, \cdots, K\}$. Tenemos:*

$$\hat{p}_k(x; w') = \frac{e^{w'_k x}}{\sum_{i=1}^{K} e^{w'_i x}} = \frac{e^{(w_k+u)x}}{\sum_{i=1}^{K} e^{(w_i+u)x}} = \frac{e^{w_k x} e^{ux}}{\sum_{i=1}^{K} e^{w_i x} e^{ux}} = \frac{e^{w_k x}}{\sum_{i=1}^{K} e^{w_i x}} = \hat{p}_k(x; w) \; \square$$

Así, a partir de este resultado, podemos demostrar la siguiente proposición:

Proposición 3.2. *En el caso de K=2, el modelo de regresión logística con función sigmoidea y con función softamx coinciden.*

Demostración. *Suponemos $w = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}$ la matriz de pesos. Entonces:*

$$h_w(x) = \frac{1}{e^{w_1 x} + e^{w_2 x}} \begin{bmatrix} e^{w_1 x} \\ e^{w_2 x} \end{bmatrix}$$

Realizando un desplazamiento de valor w_1:

$$h_w(x) = \frac{1}{e^{(w_1-w_1)x} + e^{(w_2-w_1)x}} \begin{bmatrix} e^{(w_1-w_1)x} \\ e^{(w_2-w_1)x} \end{bmatrix} = \frac{1}{1 + e^{(w_2-w_1)x}} \begin{bmatrix} 1 \\ e^{(w_2-w_1)x} \end{bmatrix}$$

Llamando $\hat{w} = -(w_2 - w_1)$:

$$h_w(x) = \frac{1}{1 + e^{-\hat{w}x}} \begin{bmatrix} 1 \\ e^{-\hat{w}x} \end{bmatrix}$$

Donde vemos que la primera componente de la función $h_w(x)$ es la función sigmoidea y la segunda es la probabilidad de su complementario. $\square$

Finalmente, el error cometido en un punto x_i donde la salida asociada a dicho punto es y_i será:

$$E(x_i, y_i; w) = -log(\hat{p}_{y_i}(x_i; w)) \tag{3.3}$$

Así, extendemos la función de coste de la regresión logística definida en (2.3) a la función:

$$L(x, y; w) = \frac{1}{N} \sum_{i=1}^{N} E(x_i, y_i; w) = -\frac{1}{N} \sum_{i=1}^{N} \sum_{k=1}^{K} \mathbf{1}_{\{y_i=k\}} log(\hat{p}_k(x_i; w)) \tag{3.4}$$

Donde $\mathbf{1}_{\{y_i=k\}}$ es la función indicador de la clase k. Notemos que la igualdad se da porque cada x_i está asociada a una única clase y_i.

3.2. Cálculo del vector gradiente

Igual que en el capítulo anterior, la función de coste se minimiza mediante descenso de gradiente. Hallamos entonces el gradiente. Para ello, comenzamos calculando la siguiente derivada:

$$\frac{\partial}{\partial w_{lj}}(w_k x_i) = \frac{\partial}{\partial w_{lj}}(w_{k0} + w_{k1} x_{i1} + \cdots + w_{k\rho} x_{i\rho}) = \mathbf{1}_{\{l=k\}} x_{ij} \tag{3.5}$$

Así, podemos calcular ahora la siguiente derivada:

$$\frac{\partial E}{\partial w_{lj}}(x_i, y_i; w) = \frac{\partial}{\partial w_{lj}} \left(-log \left(\frac{e^{w_{y_i} x_i}}{\sum_{h=1}^{K} e^{w_h x_i}} \right) \right) = \frac{\partial}{\partial w_{lj}} \left(-log\left(e^{w_{y_i} x_i}\right) + log \left(\sum_{h=1}^{K} e^{w_h x_i} \right) \right) =$$

$$= \frac{\partial}{\partial w_{lj}} \left(-w_{y_i} x_i + log \left(\sum_{h=1}^{K} e^{w_h x_i} \right) \right) = \frac{\partial}{\partial w_{lj}}(-w_{y_i} x_i) + \frac{\partial}{\partial w_{lj}} \left(log \left(\sum_{h=1}^{K} e^{w_h x_i} \right) \right) =^{(3.5)}$$

$$= -\mathbf{1}_{\{l=y_i\}} x_{ij} + \frac{\frac{\partial}{\partial w_{lj}} \left(\sum_{h=1}^{K} e^{w_h x_i} \right)}{\sum_{h=1}^{K} e^{w_h x_i}} = -\mathbf{1}_{\{l=y_i\}} x_{ij} + \frac{\sum_{h=1}^{K} \frac{\partial}{\partial w_{lj}} \left(e^{w_h x_i} \right)}{\sum_{h=1}^{K} e^{w_h x_i}} =$$

$$= -\mathbf{1}_{\{l=y_i\}} x_{ij} + \frac{\sum_{h=1}^{K} e^{w_h x_i} \frac{\partial}{\partial w_{lj}} \left(w_h x_i \right)}{\sum_{h=1}^{K} e^{w_h x_i}} = -\mathbf{1}_{\{l=y_i\}} x_{ij} + \frac{\sum_{h=1}^{K} e^{w_h x_i} \mathbf{1}_{\{l=h\}} x_{ij}}{\sum_{h=1}^{K} e^{w_h x_i}} =$$

$$= -\mathbf{1}_{\{l=y_i\}} x_{ij} + \frac{e^{w_l x_i} x_{ij}}{\sum_{h=1}^{K} e^{w_h x_i}} = x_{ij} \left(\hat{p}_l(x_i; w) - \mathbf{1}_{\{l=y_i\}} \right) \tag{3.6}$$

Así, mediante esta expresión, podemos hallar la siguiente matriz de derivadas:

$$\begin{bmatrix} \frac{\partial}{\partial w_{1,0}} E(x_i, y_i; w) & \frac{\partial}{\partial w_{1,1}} E(x_i, y_i; w) & \cdots & \frac{\partial}{\partial w_{1,\rho}} E(x_i, y_i; w) \\ \frac{\partial}{\partial w_{2,0}} E(x_i, y_i; w) & \frac{\partial}{\partial w_{2,1}} E(x_i, y_i; w) & \cdots & \frac{\partial}{\partial w_{2,\rho}} E(x_i, y_i; w) \\ \vdots & \vdots & & \vdots \\ \frac{\partial}{\partial w_{K,0}} E(x_i, y_i; w) & \frac{\partial}{\partial w_{K,1}} E(x_i, y_i; w) & \cdots & \frac{\partial}{\partial w_{K,\rho}} E(x_i, y_i; w) \end{bmatrix} =^{(3.6)}$$

$$\begin{bmatrix} \hat{p}_1(x_i; w) - \mathbf{1}_{\{y_i=1\}} & x_{i,1} \left(\hat{p}_1(x_i; w) - \mathbf{1}_{\{y_i=1\}} \right) & \cdots & x_{i,\rho} \left(\hat{p}_1(x_i; w) - \mathbf{1}_{\{y_i=1\}} \right) \\ \hat{p}_2(x_i; w) - \mathbf{1}_{\{y_i=2\}} & x_{i,1} \left(\hat{p}_2(x_i; w) - \mathbf{1}_{\{y_i=2\}} \right) & \cdots & x_{i,\rho} \left(\hat{p}_2(x_i; w) - \mathbf{1}_{\{y_i=2\}} \right) \\ \vdots & \vdots & & \vdots \\ \hat{p}_K(x_i; w) - \mathbf{1}_{\{y_i=K\}} & x_{i,1} \left(\hat{p}_K(x_i; w) - \mathbf{1}_{\{y_i=K\}} \right) & \cdots & x_{i,\rho} \left(\hat{p}_K(x_i; w) - \mathbf{1}_{\{y_i=K\}} \right) \end{bmatrix} = \tag{3.7}$$

$$= \begin{bmatrix} \hat{p}_1(x_i; w) - \mathbf{1}_{\{y_i=1\}} \\ \hat{p}_2(x_i; w) - \mathbf{1}_{\{y_i=2\}} \\ \vdots \\ \hat{p}_K(x_i; w) - \mathbf{1}_{\{y_i=K\}} \end{bmatrix} x_i \tag{3.8}$$

Finalmente, considerando el **isomorfismo** natural entre $\mathbb{R}^{K \times (\rho+1)} \sim \mathbb{R}^{K(\rho+1)}$ ya podemos calcular el **gradiente** de todos nuestros parámetros mediante la matriz anterior. Es decir:

$$\nabla E(x_i, y_i; w) =^{(3.7)} \left(\hat{p}_1(x_i; w) - \mathbf{1}_{\{y_i=1\}} \quad x_{i,1} \left(\hat{p}_1(x_i; w) - \mathbf{1}_{\{y_i=1\}} \right) \quad \cdots \quad x_{i,\rho} \left(\hat{p}_K(x_i; w) - \mathbf{1}_{\{y_i=K\}} \right) \right) \tag{3.9}$$

Además, vemos que:

$$\frac{\partial L}{\partial w_{lj}}(x_i, y_i; w) = \frac{\partial}{\partial w_{lj}} \left(\frac{1}{N} \sum_{i=1}^{N} E \right) (x_i, y_i; w) = \frac{1}{N} \sum_{i=1}^{N} \left(\frac{\partial E}{\partial w_{lj}}(x_i, y_i; w) \right)$$

Y así, tenemos:

$$\nabla L(x_i, y_i; w) = \frac{1}{N} \sum_{i=1}^{N} \nabla E(x_i, y_i; w) \tag{3.10}$$

Así, utilizaremos la ecuación (3.9) para el **descenso de gradiente estocástico** y la ecuación (3.10) para el **descenso de gradiente por lotes o mini-lotes**. Sin embargo, según la ecuación dada en (2.4) en la que se indica cómo se actualizan los pesos con el gradiente, tenemos que este se utiliza **normalizado**. Por tanto, en el Capítulo 5 lo podremos calcular sin necesidad de incluir el término $\frac{1}{N}$, puesto que este es constante una vez decidido el conjunto de datos.

Con lo definido en este capítulo ya podríamos comenzar con el caso práctico. Sin embargo, antes de iniciarlo vamos a definir en el siguiente capítulo un conjunto de medidas que nos permitirán evaluar el rendimiento de nuestros modelos.

Capítulo 4

Evaluación de modelos

Una vez realizado el modelo, debemos comprobar cómo de bien ajusta los datos. Es importante realizar la **validación del modelo** sobre una parte del conjunto de datos **que no haya sido utilizada para entrenarlo**. Esto se debe a que queremos comprobar cómo de bien funciona el modelo para predecir nuevos datos, pero no los datos cuya respuesta ya conocemos.

En este capítulo daremos entonces varias definiciones que nos permitirán medir el rendimiento de nuestros modelos.

4.1. Matriz de confusión

Comenzamos dando la definición que más nos va a ayudar a conseguir este propósito.

Definición 6. Dado un modelo de clasificación con K posibles clases, llamamos **matriz de confusión** a una matriz $\mathbb{N}^{K\times K}$ construida de la siguiente forma: La entrada (i, j) de la matriz es el número de elementos de la clase i que han sido clasificados como elementos de la clase j por el modelo.

Puede verse un ejemplo de una matriz de confusión en la Figura 4.1. En ella, los valores diagonales serían los **clasificados correctamente**, mientras que los extradiagonales serían los **clasificados incorrectamente**.

Además, no sólo nos aporta información sobre cuantos elementos han sido mal clasificados, sino que también **nos permite saber en qué clases se están clasificando** estos elementos.

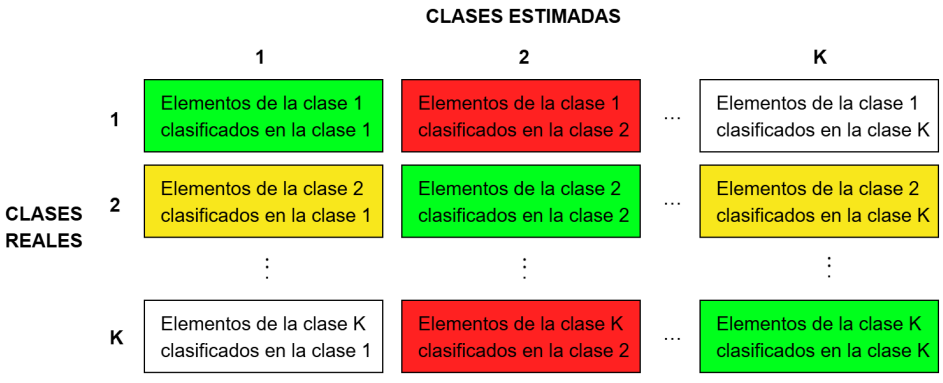

Figura 4.1: Diagrama de la matriz de confusión.

4.2. Medidas de rendimiento

La matriz de confusión, que ha sido definida en la sección anterior, también permite **definir otras medidas de rendimiento** a partir de ella [11], que son las que daremos a continuación. En todas ellas denotaremos C a la matriz de confusión y C_{ij} la entrada de la fila i y columna j.

Definición 7. Llamamos **accuracy** al número de entradas clasificadas correctamente sobre el total de muestras.

$$Accuracy = \frac{\sum_{i=1}^{K} C_{ii}}{\sum_{i=1}^{K}\sum_{j=1}^{K} C_{ij}} \in [0,1] \tag{4.1}$$

Es decir, la accuracy indica la proporción de muestras clasificadas correctamente por el modelo. En la Figura 4.1 sería el resultado de dividir la suma de todas las casillas verdes entre la suma de todas las casillas de la matriz.

Definición 8. Fijamos una clase k. Llamamos **precisión** al cociente de elementos clasificados correctamente en esta clase sobre el total de elementos clasificados como pertenecientes a esta clase.

$$Precision_k = \frac{C_{kk}}{\sum_{i=1}^{K} C_{ik}} \in [0,1] \tag{4.2}$$

Así, la precisión es la probabilidad de que un elemento clasificado en la clase k haya sido clasificado correctamente. Por ejemplo, tomando k=2 en la Figura 4.1, tendríamos que la precisión sería el resultado de dividir la casilla verde de la segunda columna entre su valor más la suma de todas las casillas rojas.

Definición 9. Fijamos una clase k. Llamamos **recall** al cociente entre las muestras clasificadas correctamente como pertenecientes a la clase k y el número total de muestras pertenecientes a la clase k.

$$Recall_k = \frac{C_{kk}}{\sum_{i=1}^{K} C_k i} \in [0,1] \tag{4.3}$$

Por tanto, el recall es una medida que indica la proporción de muestras de la clase k que el modelo clasifica correctamente. Por ejemplo, tomando k=2 en la Figura 4.1, tendríamos que el recall sería el resultado de dividir la casilla verde de la segunda fila entre ella misma más la suma de todas las casillas de color amarillo.

En general, **es muy difícil obtener valores muy altos en todas las categorías simultáneamente**. Por tanto, dependiendo de nuestro problema deberemos fijarnos más en unas u otras medidas. Sin embargo, hay problemas en los que tanto la precisión como el recall son importantes. En estos casos se define la siguiente medida.

Definición 10. Fijamos una clase k. Llamamos **F1-score** [12] a la media armónica entre la precisión y el recall.

$$F1\_score_k = 2 * \frac{Precision_k * Recall_k}{Precision_k + Recall_k} \in [0,1]$$

Así, esta medida se maximiza cuando tanto la precisión como el recall alcanzan valores altos, penalizando las altas diferencias entre ambos.

Con todas las definiciones dadas en este capítulo se pueden valorar los modelos de clasificación para sacar conclusiones sobre sus resultados. Así, estamos en condiciones de comenzar con el siguiente capítulo, donde realizaremos una aplicación práctica de un modelo de clasificación.

Capítulo 5

Aplicación para la clasificación de terrenos

Para la aplicación se utilizarán los datos de la referencia [13]. Este conjunto de datos está formado por diferentes medidas sobre terrenos de los Estados Unidos, como pueden ser la elevación, inclinación del terreno, presencia o ausencia de determinadas sustancias, etc. El **objetivo** será entonces clasificar los terrenos según el tipo de árbol que sea más abundante en estos, habiendo **7 posibles opciones**: Abeto, pino contorto, pino ponderoso, álamo-sauce, álamo temblón, abeto de Douglas y Krummholz. La **finalidad** de esta clasificación será la de comprobar que somos capaces de desarrollar un algoritmo que aprenda de los datos únicamente con lo definido en los capítulos previos.

La explicación más en detalle de todas las variables está recogida en el Anexo A.

5.1. Preparación de los datos

Lo primero que debemos hacer es **tener en cuenta las hipótesis que asumen los modelos de regresión logística**, que han sido comentadas en el Capítulo 2.

Comenzamos entonces **evitando la multicolinealidad entre variables**. Para ello, calculamos la matriz de correlaciones mediante el código mostrado en la Figura B.1, obteniendo así la imagen mostrada en la Figura 5.1.

Vemos que no podemos distinguir a simple vista qué variables son las correladas. Por tanto, mediante el código de la Figura C.1 podremos **eliminar las variables cuya correlación sea superior a un determinado valor**. En cada prueba realizada posteriormente se indicará que valor ha sido utilizado.

Una vez eliminadas estas variables, vemos que todas las restantes presentan un tipo numérico. Sin embargo, sabemos que la mayoría son variables **binarias**. Mediante el código de la función mostrada en la Figura C.2 cambiamos el tipo de estas variables a '*Categorical*'. Esto nos permitirá identificar sobre qué variables realizar las siguientes operaciones.

Así, una vez corregida la multicolinealidad, el siguiente paso es **evitar los datos atípicos extremos**. Para ello, contemplamos **dos posibles estrategias**: eliminar los datos atípicos o sustituirlos por los cuantiles 0.25 y 0.75 en función de si son datos atípicos demasiado pequeños o demasiado grandes, respectivamente.

Antes de trabajar con este tipo de datos, **realizamos un gráfico de tipo boxplot** para cada una de las variables numéricas que hay presentes en el conjunto de datos. Estas variables son: *Elevation*, *Horizontal_Distance_To_Roadways* (HDTR) y *Horizontal_Distance_To_Fire_Points* (HDTFP). Este gráfico lo realizamos mediante la función B.2 y puede verse en la Figura 5.2, donde destaca la **gran presencia de valores atípicos**, así como la **falta de simetría** en las dos últimas variables.

Completamos con un **gráfico de las distancias de Cook**, en el que podemos ver cuánto se van los datos influyentes con respecto al umbral. Este es realizado mediante la función de la Figura B.3 y puede verse en la Figura 5.3. Cabe destacar que, aunque parezca que la gran mayoría de datos son atípicos, esto se debe a la gran cantidad de datos en el eje X, siendo muy superior a la resolución de la gráfica.

Una vez estudiados estos gráficos, utilizamos las funciones de la Figura C.3 y la Figura C.4 para **detectar los datos atípicos e influyenes** respectivamente. Una vez identificados, podemos aplicar las

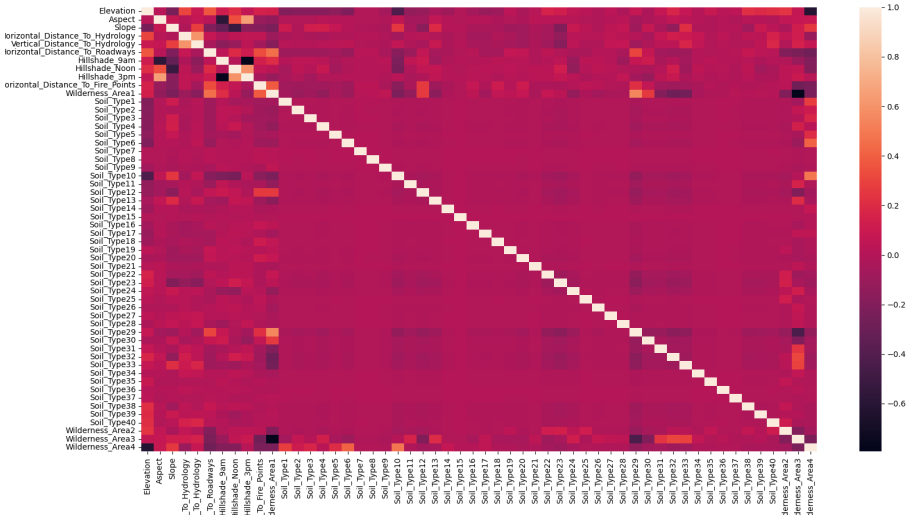

Figura 5.1: Matriz de correlaciones entre las variables

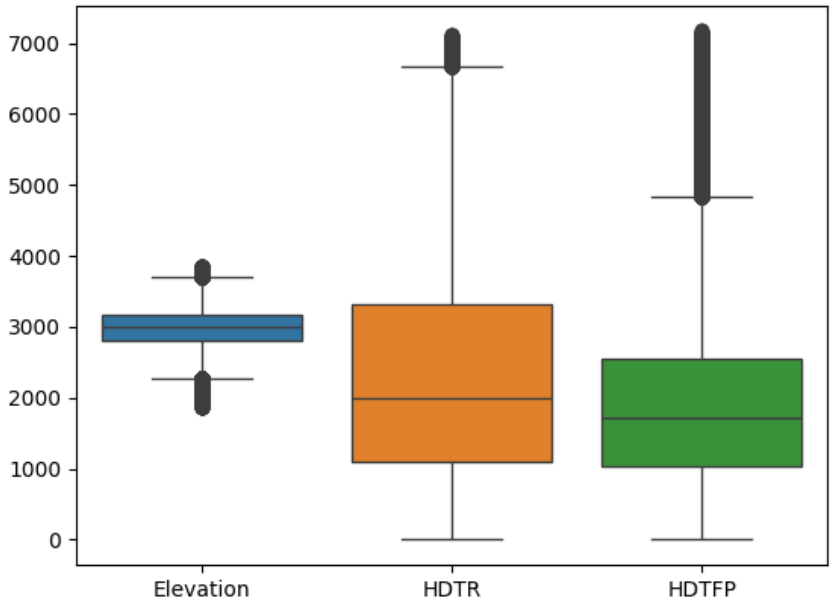

Figura 5.2: Boxplot de los datos de tipo numérico.

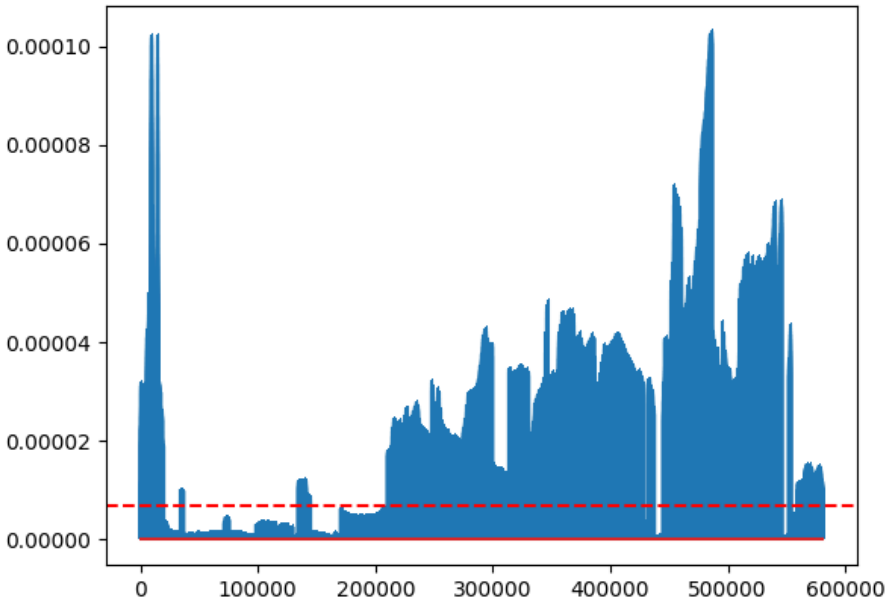

Figura 5.3: Distancias de Cook del modelo.

dos estrategias anteriores mediante la función de la Figura C.5 para eliminarlos o la de la Figura C.6 para llevarlos a los cuantiles. Más adelante, en la información asociada a cada prueba realizada, se especificará qué estrategia se ha aplicado.

Una vez realizado este análisis procedemos a la **estandarización de los datos** mediante la función mostrada en la Figura C.7, haciendo que todas las variables numéricas tengan media cero y desviación típica uno.

Ahora, tras realizar estos pasos, estaríamos en disposición de aplicar el algoritmo explicado en el Capítulo 3. Sin embargo, si mostramos la **cantidad de muestras en cada clase** obtenemos el gráfico mostrado en la Figura 5.4. Vemos que **la distribución de las clases es muy desigual**, lo que genera problemas de ajuste en el modelo para las clases minoritarias. Así, aplicando una estrategia de **sobremuestreo de las clases minoritarias** mediante la función de la Figura C.8 obtenemos como resultado una distribución idéntica de todas las clases.

Finalmente, tenemos los datos preparados para aplicarles el algoritmo descrito en el Capítulo 3.

5.2. Implementación del modelo

Para poder aplicar el algoritmo a los datos necesitamos previamente **implementar varias funciones recogidas en este trabajo**.

5.2.1. Programación del algoritmo

Comenzamos programando la **función softmax** definida en la ecuación (3.2) mediante la función de la Figura D.1, que nos permitirá calcular, para cada punto, las probabilidades estimadas de pertenecer a cada tipo de terreno.

Posteriormente, programamos la **funciones de error** y la **función de coste** definidas por las ecuaciones (3.3) y (3.4) respectivamente. Estas están recogidas en la Figura D.2.

Ahora, para tratar de encontrar el mínimo de estas funciones y lograr así el mejor ajuste programamos el **descenso de gradiente**. Para ello:

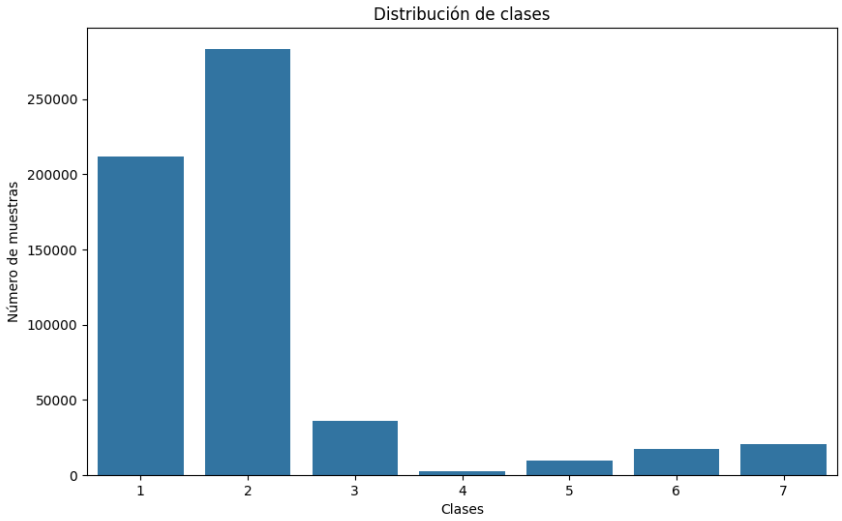

Figura 5.4: Distribución de clases original.

- Hallamos el **gradiente** realizando las operaciones descritas en la ecuación (3.8) mediante la función de la Figura D.3.
- Extendemos el cálculo del gradiente a un **lote completo**. Para ello, realizamos las operaciones descritas en la ecuación (3.10) mediante la función recogida en la Figura D.4.
- **Actualizamos los pesos** en cada iteración según la ecuación (2.4) mediante la función de la Figura D.5. Aquí tendremos en cuenta que, si la norma del gradiente es cero, ya hemos hallado un mínimo.

Así, una vez ajustado el modelo, podemos **obtener la clase con mayor probabilidad estimada** mediante la función de la Figura D.6.

Finalmente, la función que nos permite realizar todo el **proceso de ajuste** mediante las funciones definidas hasta el momento es la recogida en la Figura D.7, mientras que la función que nos permite inicializar todos los parámetros necesarios está en la Figura D.8.

5.2.2. Programación de las métricas de rendimiento

En esta sección realizamos la **implementación de todas las funciones que nos permiten evaluar nuestro modelo**, y que han sido definidas previamente en el Capítulo 4.

Comenzamos con la implementación de la función que nos permite calcular la **matriz de confusión** del modelo, que puede verse en la Figura E.1.

Posteriormente, programamos la función que nos permite hallar **el accuracy, la precisión y el recall**. Todas estas funciones pueden verse en la Figura E.2, la Figura E.3 y la Figura E.4 respectivamente.

Finalmente, como el cálculo del accuracy y de la función de coste requieren de cálculos similares, podemos realizar una **optimización** mediante la fusión de las funciones que calculan ambas para **reducir el tiempo que tardarán todas las iteraciones** de ajuste del modelo. La función conjunta que optimiza el tiempo de cálculo es la mostrada en la Figura E.5.

5.3. Ajuste de varios modelos

Con todas las funciones implementadas en la sección anterior, podemos comenzar a ajustar modelos. Sin embargo, **todavía faltan elementos por determinar**, como son desde decidir qué tasa de aprendizaje

N	Procesado de las variables				Parámetros del modelo					Resultados		
	$Corr_1$	$Corr_2$	Out	N_{var}	Tam	T_{apr}	A_{apr}	W_0	W_1	Coste	Acc	Tiempo
1	0,5	0,075	J	19	50.000	0,1	0	Z	0	8,46	0,612	14
2	0,5	0,075	J	19	50.000	0,01	0	Z	0	4,41	0,609	12
3	0,5	0,075	J	19	50.000	0,001	0	Z	0	1,17	0,609	12
4	0,5	0,075	J	19	50.000	0,0001	0	Z	0	1.63	0,611	12
5	0,5	0,075	J	19	50.000	0,0005	0	Z	0	1,22	0,606	13
6	0,5	0,025	J	30	50.000	0,001	0	Z	0	1,17	0,615	16
7	0,5	0	J	44	50.000	0,001	0	Z	0	1,13	0,614	20
8	0,5	0,01	J	34	50.000	0,001	0	Z	0	1,14	0,614	17
9	0,4	0	S	40	50.000	0,001	0	Z	0	1,19	0,510	19
10	0,6	0	S	47	50.000	0,001	0	Z	0	1,18	0,538	26
11	0,5	0,025	J	30	50.000	0,1	0	Z	1000	1,03	0,687	16
12	0,5	0,025	C	30	50.000	0,1	0	Z	1000	1,07	0,651	16
13	0,5	0	J	44	50.000	0,1	0	Z	1000	1,00	0,703	20
14	0,5	0	J	44	50.000	0,4	2:1	Z	1000	0,80	0,717	20
15	0,5	0	J	44	50.000	0,8	2:1	Z	1000	0,79	0,717	20
16	0,5	0	J	44	150.000	0,8	2:1	Z	3000	0,77	0,716	79
17	0,5	0	J	44	50.000	0,8	2:1	Z	50000	0,78	0,715	39
18	0,5	0	J	44	150.000	-	2:1	16	3000	0,71	0,731	60
19	0,5	0	J	44	150.000	-	2:1	18	3000	0,69	0,734	77

Cuadro 5.1: Información de las pruebas realizadas y los resultados obtenidos.

tomar y si actualizarla o no hasta decidir qué estrategia tomar con el tratamiento de outliers y datos influyentes, como se comento en la sección 5.1.

Para todas estas decisiones **no existen métodos analíticos** cuyo resultado sean los valores exactos que tomar para conseguir el mejor rendimiento. Sin embargo, existen diferentes librerías que permiten aproximar estos parámetros. A pesar de ello, **optaremos por realizar diferentes pruebas con distintos valores para ver cómo estos pueden ir influyendo en los resultados finales**.

Así pues, realizamos todas las pruebas que están recogidas en el Cuadro 5.1, donde **N** indica el número de prueba.

En el **procesado de variables**, la columna $\mathbf{Corr_1}$ indica el valor de correlación a partir del cual eliminamos las variables correladas con el fin de evitar la colinealidad, $\mathbf{Corr_2}$ el valor de correlación mínimo que deben tener las variables con la respuesta para no ser eliminadas (en caso de que reduzcamos el número de variables), **Out** indica la estrategia a seguir con los datos atípicos e influyentes (J si los buscamos en el conjunto completo y los eliminamos, S si los buscamos para cada clase por separado y los eliminamos, C si los llevamos a los cuantiles 0.25 y 0.75), $\mathbf{N_{var}}$ el número de variables que quedan tras las eliminaciones.

Con respecto a los **parámetros del modelo**, la columna **Tam** indica el tamaño del conjunto de entrenamiento, $\mathbf{T_{apr}}$ indica la tasa de aprendizaje inicial, $\mathbf{A_{apr}}$ indica cómo se actualiza la tasa de aprendizaje (0 indica que no se actualiza, i:j indica que se divide la tasa por i cada j veces que el coste deja de decrecer), $\mathbf{W_0}$ indica el valor inicial de los pesos (Z si inicializados a cero, un número n si comenzamos desde los pesos resultado de la prueba n), $\mathbf{W_1}$ indica cómo se actualizan los pesos (0 para descenso de gradiente estocástico, otro número para descenso por lotes de ese tamaño).

Finalmente, en el apartado de **resultados** tenemos que la columna **Coste** indica el coste final del modelo, **Accuracy** el accuracy final obtenido y **Tiempo** el tiempo que ha tardado en realizarse el ajuste medido en minutos.

Así pues, en las **pruebas 1 a 5** mantenemos fijos todos los datos salvo la **tasa de aprendizaje**, para poder comprobar cuál es la escala más adecuada. Vemos que esta debe estar por debajo de 0,001 para reducir el coste, siendo precisamente este el mejor valor de los probados.

Clase	1	2	3	4	5	6	7
1	42616	8982	100	0	2338	833	63
2	14113	24740	2069	54	7968	5186	662
3	0	278	38723	11018	1949	1586	1427
4	0	0	6471	48504	0	0	0
5	0	9312	2173	0	32574	7770	3362
6	0	3008	2106	0	5455	44422	647
7	0	0	0	0	0	3963	50821

Cuadro 5.2: Matriz de confusión del modelo con mejor accuracy.

Clase	Precisión
1	0.775112
2	0.53411
3	0.74984
4	0.81415
5	0.64780
6	0.69700
7	0.89188

Clase	Recall
1	0.77580
2	0.45153
3	0.70430
4	0.88229
5	0.59020
6	0.79841
7	0.92812

Cuadro 5.3: A la izquiera precisión del modelo. A la derecha, recall.

Una vez probado esto, en las **pruebas 6 a 8** buscamos el mejor dato de **correlación para eliminar las variables menos importantes**, comprobando que lo mejor es no eliminar ninguna.

Posteriormente, en las **pruebas 9 y 10** comprobamos otros posibles valores para **eliminar las variables correladas entre sí**, comprobando que lo mejor es eliminar las correlaciones superiores a 0,5.

En las **pruebas 11 a 13** probamos distintas **estrategias de tratamiento de outliers** en un **descenso de gradiente por mini-lotes**, obteniendo mejores resultados con la eliminación de los datos atípicos.

Una vez probado esto, en las pruebas **14 a 17** probamos a ir **actualizando progresivamente la tasa de aprendizaje**, dividiendo esta a la mitad cada vez que el coste crece en una iteración. Además, lo probamos para **distintos tamaños de lotes y de muestras** obteniendo el mejor coste en el conjunto con 150.000 datos y con iteraciones de 3.000 elementos.

En las **pruebas 18 y 19** probamos a **seguir ajustando los modelos obtenidos en las pruebas 16 y 18 respectivamente**, comprobando que el modelo sigue mejorando progresivamente.

5.4. Análisis del mejor modelo

Aunque los resultados finales de los modelos están recogidos en el Cuadro 5.1, tenemos que **a veces la mejor accuracy y el menor coste no se han dado en iteraciones finales, sino en intermedias**. Concretamente, la **mayor accuracy** ha sido obtenida en una iteración intermedia del modelo 19 con un valor de **0.7347**, mientras que el **menor coste** se ha dado en otra iteración intermedia del modelo 19 con valor de **0.6939**. Destacar que **la evaluación del accuracy del modelo debe realizarse sobre el conjunto de test** y no en el de aprendizaje, donde el modelo consigue **0.7330** de accuracy.

Por tanto, **estudiamos el rendimiento del modelo con mejor accuracy más en detalle**, mediante las métricas explicadas en el Capítulo 4. Comenzamos viendo su **matriz de confusión**, que puede verse en el Cuadro 5.2.

A partir de esta matriz, **calculamos el resto de métricas**. Los datos de **precisión** los calculamos mediante la expresión (4.2) y los de **recall** mediante la expresión (4.3). Ambos son mostrados en el Cuadro 5.3.

Gracias a estos datos, vemos que **nuestro principal problema de clasificación está con los elementos de la clase 2**, puesto que tienen tasas de precisión y recall muy inferiores a los demás elementos.

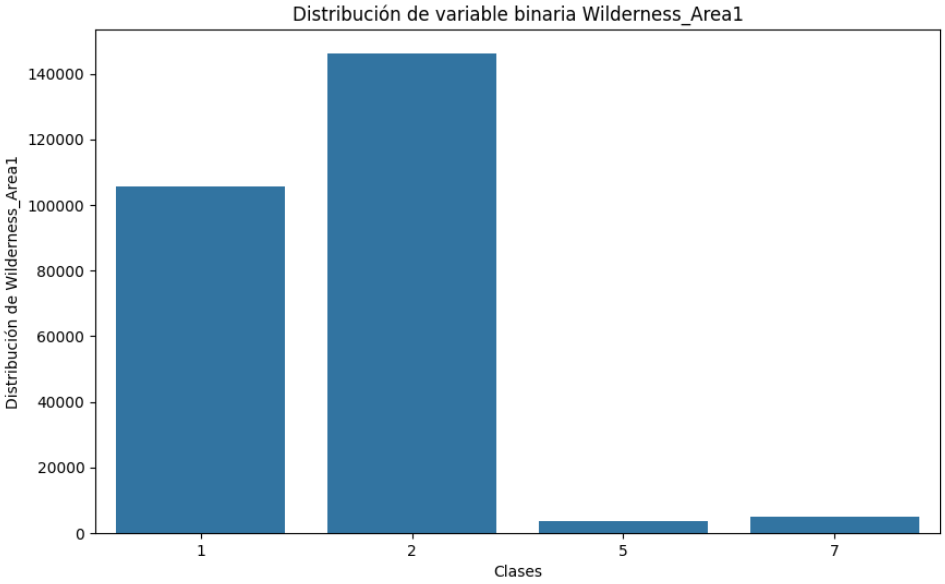

Figura 5.5: Distribución de la variable *Wilderness_Area1* por clases.

Además, **las otras dos clases que mayores problemas presentan son la 5 y la 6**. Sin embargo, si miramos detalladamente los datos, vemos que **parte de este problema es debido a confusiones con la clase 2**. En base a esta información, vamos a plantear alguna posible mejora del modelo.

5.5. Mejora del modelo

Con la finalidad de intentar mejorar el modelo, vamos a centrarnos en **observar más detenidamente las variables que han salido del conjunto de datos** como resultado del procesamiento de datos para ver si alguna de ellas diferencia a las clases 2, 5 y 6.

Tras una observación detenida, vemos que **la variable candidata a reincorporarse al conjunto de datos sería la variable *Wilderness_Area1***, pues su distribución por clases puede verse en la Figura 5.5, donde vemos que es muy abundante en los terrenos de tipo 2, mientras que en los terrenos 5 y 6 es muy infrecuente o inexistente.

Así pues, realizamos **dos pruebas de ajuste del modelo incluyendo esta variable**, que pueden verse en el Cuadro 5.4 con los **índices 20 y 21**. Vemos que obtenemos resultados muy similares al modelo analizado en la sección anterior. Sin embargo, **estos resultados has sido obtenidos únicamente con 50 iteraciones de entrenamiento**, mientras que el anterior ha sido obtenido con 150, luego podemos considerarlo una leve mejora.

Por otro lado, la mejor **no es tan significativa como cabría esperar** viendo la Figura 5.5. Entonces, viendo la ecuación (3.9), vemos que si una característica toma el valor 0 entonces el gradiente de la componente asociada es 0 y el parámetro no se actualiza. Por tanto, **al codificar las variables binarias como 0 o 1 podemos obtener información de la presencia de una sustancia, pero no de su ausencia**.

Por este motivo, **recodificamos todas las variables binarias** como 1 si la sustancia está presente y -1 en caso contrario y realizamos las **pruebas 22 a 24** obteniendo resultados un poco peores. Sin embargo, vemos que **los tiempos de ajuste del modelo se reducen significativamente**.

Procedemos entonces a realizar las mismas pruebas pero con **conjuntos de datos de mayor tamaño**. Los resultados pueden verse en las **pruebas 25 a 27** obteniendo resultados un poco mejores pero que han requerido más tiempo para ser ajustados.

Una vez realizadas estas pruebas, **probamos a coger el mejor modelo obtenido y seguir ajustándolo**. Estas son las **pruebas 28 a 32**, donde tenemos que el modelo va mejorando hasta llegar a una

N	Procesado de las variables				Parámetros del modelo					Resultados		
	$Corr_1$	$Corr_2$	Out	N_{var}	Tam	T_{apr}	A_{apr}	W_0	W_1	Coste	Acc	Tiempo
20	0,5	WA1	J	45	50.000	0,8	2:2	Z	0	0,73	0,730	20
21	0,5	WA1	J	45	50.000	0,8	2:1	Z	0	0,75	0,730	20
22	0,5	WA1	J	45	50.000	0,8	2:1	Z	0	0,85	0,708	6
23	0,5	WA1	J	45	50.000	0,8	2:1	Z	0	0,77	0,714	7
24	0,5	0	J	44	50.000	0,8	2:1	Z	0	0,78	0,712	6
25	0,5	WA1	J	45	150.000	0,8	2:1	Z	3000	0,76	0,718	19
26	0,5	WA1	J	45	150.000	0,8	2:2	Z	3000	0,75	0,719	19
27	0,5	WA1	J	45	1.000.000	0,8	2:2	Z	100.000	0,74	0,720	122
28	0,5	WA1	J	45	150.000	0,8	2:2	25	3000	0,71	0,730	19
29	0,5	WA1	J	45	150.000	-	2:2	28	3000	0,70	0,734	21
30	0,5	WA1	J	45	150.000	0,1	2:2	29	3000	0,68	0,738	24
31	0,5	WA1	J	45	150.000	0,1	2:2	30	3000	0,67	0,739	25
32	0,5	WA1	J	45	150.000	0,1	2:2	31	3000	0,67	0,739	25
33	0,5	WA1	J	45	150.000	*	2:2	Z	3000	3.19	0,739	25
34	0,5	WA1	J	45	150.000	-	2:2	33	3000	1,23	0,762	19
35	0,5	WA1	J	45	150.000	-	2:2	34	3000	1,04	0,770	18

Cuadro 5.4: Información de pruebas realizadas y resultados.

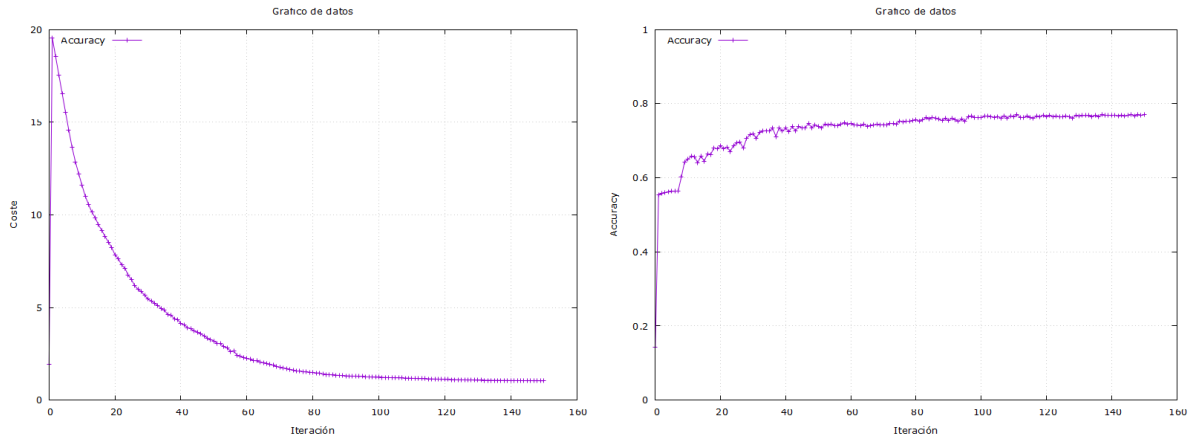

Figura 5.6: A la izquierda gráfica de coste. A la derecha gráfica de accuracy.

accuracy de 0,739 con un coste de 0,67. Vemos que estos vuelven a ser resultados muy similares a los obtenidos anteriormente. Esto nos lleva a pensar que **puede darse el caso de que exista un mínimo local en torno al cero cuyo valor sea el que estamos obteniendo**. Para comprobarlo, realizamos la **prueba 33**, en la que tomamos como tasa de aprendizaje inicial el valor 50. De esta forma, **tras calcular el gradiente la primera vez, desplazamos los parámetros de manera significativa con el fin de evitar el supuesto mínimo local**. Una vez actualizados la primera vez, ponemos una tasa de aprendizaje de 0,8 y procedemos con la forma habitual. Denotamos esta característica como * en la columna denotada T_{apr} en el Cuadro 5.4.

El resultado de esta última prueba es una accuracy idéntica al modelo 32 pero con un coste muy superior. Aunque en principio parecería un mal dato, tenemos que **hemos obtenido la misma accuracy con sólo un ajuste** (cuando el modelo anterior a requerido 5 reajustes). Además, al tener un coste tan alto, todavía tiene margen de mejorar más. Por tanto, en las **pruebas 34 y 35** seguiremos ajustando el mismo modelo, llegando a un **accuracy de 0,770** en el conjunto de entrenamiento y **0,771** en el conjunto de test, **siendo este el mayor que hemos conseguido hasta el momento**. En la Figura 5.6 puede verse cómo han evolucionado el coste y el accuracy con el paso de las iteraciones.

Clase	1	2	3	4	5	6	7
1	41154	10903	67	0	1791	858	390
2	12765	27209	2545	44	7165	3786	1354
3	0	351	45426	5273	1705	1794	488
4	0	0	10778	44487	0	0	0
5	36	7409	2943	0	36109	7093	1468
6	0	51	1730	0	5382	48113	0
7	0	0	0	0	0	0	54608

Cuadro 5.5: Matriz de confusión del modelo mejorado.

Clase	Precisión
1	0.76275
2	0.59249
3	0.71549
4	0.89324
5	0.69238
6	0.78050
7	0.93654

Clase	Recall
1	0.74604
2	0.49590
3	0.82537
4	0.80498
5	0.655834
6	0.87041
7	1.0

Cuadro 5.6: A la izquiera precisión del modelo. A la derecha, recall.

Calculamos entonces de nuevo la matriz de confusión y las medidas de rendimiento, obteniendo los datos mostrados en el Cuadro 5.5 y el Cuadro 5.6 respectivamente. En estas vemos una mejora leve general del modelo, a pesar de que la clase 2 sigue siendo la peor clasificada.

Finalmente, destacar que, para realizar **predicciones sobre nuevos datos**, estos deberían ser normalizados igual que los datos que hemos utilizado.

5.6. Conclusiones

Durante el proceso de ajuste de los modelos, **hemos probado varios valores para todos los parámetros** que definen su comportamiento, como pueden ser el valor de la tasa de aprendizaje, la actualización de la misma, el tamaño del conjunto de datos, el tratamiento de los datos atípicos, etc.

Recordemos que **no existen métodos analíticos** que nos den los mejores valores para estos parámetros, o al menos no en un **tiempo computacionalmente aceptable**. Bien es cierto que hay librerías que consiguen aproximar estos valores, pero el **propósito de este trabajo** no era obtener los mejores resultados posibles, sino ver que la teoría que hemos definido en este documento es suficiente para desarrollar un algoritmos que aprenda de los datos.

Así, con respecto al accuracy de nuestro modelo, vemos que:

- Una **clasificación al azar** en una de las siete clases hubiera supuesto un accuracy de $\frac{1}{7} \approx 14,2857\,\%$.
- Teniendo en cuenta que no todas las clases están igual de distribuidas en nuestros datos, la **clasificación en la clase mayoritaria** hubiera supuesto un accuracy de $\frac{283301}{581012} \approx 48,7599\,\%$.
- **Nuestro último modelo** ha conseguido un accuracy de $77,1153\,\%$.

Por tanto, podemos concluir que nuestro modelo efectivamente ha aprendido de los datos disponibles, habiéndose cumplido así el objetivo del presente trabajo.

Apéndice A

Descripción de las variables

En este anexo realizaremos una descripción de los datos del Data Frame.

- **Elevation:** Mide la elevación sobre el nivel del mar en metros.
- **Aspect:** Orientación medida en azimuts (grados con respecto al polo norte dado un plano de referencia).
- **Slope:** Inclinación del terreno en grados.
- **Horizontal_Distance_To_Hydrology**: Distancia a la fuente de agua más cercana en la superficie terrestre en metros.
- **Vertical_Distance_To_Hydrology**: Distancia a la fuente de agua subterránea más cercana en metros.
- **Horizontal_Distance_To_Roadways:** Distancia a la camino más cercano en metros.
- **Hillshade_9am:** Reflectancia de luz medida a las 9:00 en el solsticio de verano. Índice de 0 a 255.
- **Hillshade_noon:** Reflectancia de luz medida a las 12:00 en el solsticio de verano. Índice de 0 a 255.
- **Hillshade_3pm:** Reflectancia de luz medida a las 15:00 en el solsticio de verano. Índice de 0 a 255.
- **Horizontal_Distance_To_Fire_Points:** Distancia en metros al foco de incencio forestal más cercano.
- **Wilderness_Area(1-4):** Cuatro variables binarias indicando la presencia (1) o ausencia (0) de un determinado tipo de naturaleza.
- **Soil_Type(1-40):** Cuarenta variables binarias indicando la presencia (1) o ausencia (0) de determinados tipos de sustancias en los terrenos.

Finalmente, tenemos la variable respuesta:

- **Cover_Type:** Variable discreta con valores 1-7 indicando las siguientes especies de árboles:
    - **1:** Abeto.
    - **2:** Pino contorto.
    - **3:** Pino ponderoso.
    - **4:** Álamo-Sauce.
    - **5:** Álamo temblón.
    - **6:** Abeto de Douglas.
    - **7:** Krummholz.

Apéndice B

Funciones para las representaciones gráficas

En este anexo se recoge el **código utilizado para realizar representaciones gráficas de los datos**. Las funciones aquí recogidas son:

- **Figura B.1**: Realizar el gráfico de la **matriz de correlaciones** en escala de colores.
- **Figura B.2**: Función que permite realizar el gráfico de tipo **boxplot** de los datos.
- **Figura B.3**: Realiza el gráfico de las **distancias de Cook** de las variables.
- **Figura B.4**: Función que permite realizar el **histograma** de los datos numéricos.

Nótese que, en los códigos recogidos en las figuras, LaTeXno permite escribir palabras acentuadas ni letras 'ñ', motivo por el cuál estas has sido sustituidas por letras sin acentuar o letras 'n' respectivamente.

```
1         import matplotlib.pyplot as plt
2         import seaborn as sns
3
4         # Imprime una matriz a colores de las correlaciones entre
5         # las variables del dataFrame <data>
6         def imprimirMatrizCorrelaciones(data):
7             correlaciones = data.corr()
8             plt.figure(figsize=(16,16))
9             sns.heatmap(correlaciones, annot = False)
10            plt.show()
```

Figura B.1: Función que representa la matriz de correlaciones en escala de colores.

```
import matplotlib.pyplot as plt
import seaborn as sns

# Realiza el grafico tipo boxplot de <data>
def boxplotDatos(data):
    sns.boxplot(data=data)
    plt.show()
    return
```

Figura B.2: Función para realizar el gráfico boxplot de los datos.

```
import matplotlib.pyplot as plt
import numpy as np
import statsmodels.api as sm

# Realiza el grafico de las distancias de Cook en <data>
# con respecto a las columnas indicadas en <columnas>
def graficoDistanciasCook(data, respuesta, columnas):
    X = data[columnas]
    X = sm.add_constant(X)
    modelo = sm.OLS(respuesta, X).fit()

    # Obtener influencias
    influencias = modelo.get_influence()

    # Obtener distancias de Cook
    distanciasCook = influencias.cooks_distance[0]

    # Identificar puntos con alta distancia de Cook
    umbral = 4 / len(X)

    # Grafico de distancia de Cook
    plt.stem(np.arange(len(distanciasCook)), distanciasCook,
        markerfmt=",")
    plt.axhline(y=umbral, color='r', linestyle='--')
    plt.show()

    return
```

Figura B.3: Función que realiza el gráfico de las distancias de Cook.

```
import matplotlib.pyplot as plt

# Realiza el histograma de <data>
def histogramaDatos(data):
    data.hist(bins=30, figsize=(10, 8))
    plt.show()
    return
```

Figura B.4: Función para realizar el histograma de los datos.

Apéndice C

Funciones para el procesado de las variables

En este anexo se recoge el **código de las funciones que nos permiten dejar las variables preparadas para ser procesadas por el algoritmo**. Las funciones aquí recogidas son las siguientes:

- **Figura C.1**: Para cada par de variables correladas entre sí en nuestros datos, eliminamos una de ellas. De esta forma, **obtenemos un conjunto de variables poco correladas**.
- **Figura C.2**: Para aquellas variables que sean binarias, **cambiamos el tipo de dato de numérico a 'Categorical'**.
- **Figura C.3**: Función que **muestra que datos son outliers**.
- **Figura C.4**: Función que **muestra que datos son influyentes**.
- **Figura C.5**: Función que permite **eliminar datos atípicos e influyentes**.
- **Figura C.6**: Función que permite **llevar los datos atípicos a los cuantiles 0.25 o 0.75** según sean datos demasiado pequeños o demasiado grandes respectivamente.
- **Figura C.7**: **Estandariza las variables** de tipo numérico, haciendo que tengan media 0 y desviación típica 1.
- **Figura C.8**: Función para **equilibrar el número de elementos de cada clase**, de forma que todas clases sean equitativas.
- **Figura C.9**: Eliminamos las variables cuya correlación con la variable respuesta sea inferior a un determinado valor. Así, **reducimos el número de variables** manteniendo las que en principio aportan más información.

Nótese que, en los códigos recogidos en las figuras, LaTeXno permite escribir palabras acentuadas ni letras 'ñ', motivo por el cuál estas has sido sustituidas por letras sin acentuar o letras 'n' respectivamente.

```
# Elimina las variables correladas del dataFrame <data>
# que tengan correlacion superior a <limite>, dejando unicamente
# una de estas
def eliminarElementosCorrelados(data, limite):
    print(f"Hallando la matriz de correlaciones...")
    matriz = data.corr().abs()
    dim, _ = matriz.shape

    for j in range(dim-1,0,-1):
        for i in range(dim-1,0,-1):
            if i != j and matriz.iloc[i,j] > limite:
                print(f"\tEliminado: {data.columns[j]}")
                data = data.drop(data.columns[j], axis=1)
                break

    return data
```

Figura C.1: Función que nos deja el dataFrame sin variables correladas entre sí.

```
import pandas as pd

# Cambia el tipo de las variables binarias de numerico
# a 'Categorical'
def categorizar(data):
    numeroValoresDistintos = data.nunique()
    for columna in data.columns:
        if numeroValoresDistintos[columna] == 2:
            data[columna] = pd.Categorical(data[columna])
    return data
```

Figura C.2: Función que cambia el tipo de las variables de binario a 'Categorical'

```
import numpy as np
from scipy import stats

# Calculamos los valores atipicos
def calculoDeOutliers(data):
    z_scores = np.abs(stats.zscore(data._get_numeric_data()))
    outliers = np.where(z_scores > 3)
    print(data.iloc[outliers[0]])
    return
```

Figura C.3: Función que permite detectar qué datos son outliers.

```
import numpy as np
import statsmodels.api as sm

# Calcula la lista de los datos influyentes en <data>
# con respecto a las columnas indicadas en <columnas>
def calculoDeDatosInfluyentes(data, respuesta, columnas):
    X = data[columnas]
    X = sm.add_constant(X)
    modelo = sm.OLS(respuesta, X).fit()

    # Obtener influencias
    influencias = modelo.get_influence()

    # Obtener distancias de Cook
    distanciasCook = influencias.cooks_distance[0]

    # Identificar puntos con alta distancia de Cook
    umbral = 4 / len(X)
    puntos_influyentes = np.where(distanciasCook > umbral)

    return puntos_influyentes
```

Figura C.4: Función que permite detectar qué datos son influyentes.

```
import heapq

# Elimina los datos atipicos e influyentes del conjunto
# <data> con la respuesta <respuesta> asociada.
def eliminarAtipicosInfluyentes(data, respuesta):
    # Calculamos outliers e influyentes
    outliers = calculoDeOutliers(data)
    influyentes = calculoDeDatosInfluyentes(data,respuesta,
        data.select_dtypes(include=['int64',
            'float64']).columns)

    return
        data.drop(
            list(heapq.merge(outliers[0],influyentes[0]))),
        respuesta.drop(
            list(heapq.merge(outliers[0],influyentes[0])))
```

Figura C.5: Función que permite eliminar datos atípicos e influyentes.

```
import heapq

# Calcula los datos atipicos e influyentes de cada columna
# y los lleva al cuantil 0.25 o 0.75 en funcion de si son
# datos demasiado pequenos o grandes respectivamente
def llevarOutliersAlCuantil(data, respuesta):
    for columna in data.select_dtypes(include=['int64',
        'float64']).columns:
        # Calculamos outliers e influyentes
        outliers = calculoDeOutliers(data)
        influyentes =
            calculoDeDatosInfluyentes(data,respuesta,columna)

        # Juntamos las listas
        lista_indices =
            list(heapq.merge(outliers[0],influyentes[0]))

        # Calculamos los cuantiles
        cuantil75 = data[columna].quantile(0.75)
        cuantil25 = data[columna].quantile(0.25)

        # Transformamos esos datos
        for indice in lista_indices:
            if data.at[indice,columna] >= cuantil75:
                data.at[indice,columna] = cuantil75
            else:
                data.at[indice,columna] = cuantil25

    return data, respuesta
```

Figura C.6: Función que permite llevar a los cuantiles 0.25 y 0.75 los datos atípicos e influyentes.

```
from sklearn.preprocessing import StandardScaler

# Estandariza las variables numericas, haciendo que tengan
# media 0 y desviacion tipica 1
def estandarizarVariablesNumericas(data):
    scaler = StandardScaler()
    columnas_numericas = data.select_dtypes(include=['int64',
        'float64']).columns
    data[columnas_numericas] =
        scaler.fit_transform(data[columnas_numericas])

    return data
```

Figura C.7: Función de estandarización de variables.

```
from imblearn.over_sampling import SMOTE

# Equilibramos el numero de muestras de cada clase
def remuestreo(data, respuestas):
    smote = SMOTE(random_state=42)
    return smote.fit_resample(data, respuestas)
```

Figura C.8: Función de remuestreo

```
        import pandas as pd
        from queue import PriorityQueue

        # Elminina las variables de <data> que tengan una correlacion
        # con <respuesta> inferior a <limite>
        def eliminarVariablesMenosImportantes(data, respuesta, limite):
            # Hallar correlaciones con la respuesta
            dataFrame = pd.concat([data,Y], axis=1)
            correlaciones = dataFrame.corr().\
            abs()[respuesta.columns[0]].drop('Cover_Type')

            # Introducir en una lista con prioridades
            lista = PriorityQueue()
            for i in range(len(correlaciones)):
                lista.put((correlaciones.iloc[i],
                           correlaciones.index[i]))

            # Eliminar las variables que tengan correlacion
            # inferior a <limite>
            while not lista.empty():
                correlacion, elemento = lista.get()
                if correlacion < limite:
                    data = data.drop(elemento, axis=1)
                else:
                    break

            return data
```

Figura C.9: Función que disminuye el número de variables según la correlación con la respuesta

Apéndice D

Funciones para realizar la regresión logística

En este anexo se recogen todas las funciones y clases que permiten la implementación del modelo de regresión logística descrito en este trabajo.

- **Figura D.1**: Función que permite **calcular la probabilidad** que tiene una muestra de pertenecer a cada uno de los terrenos.
- **Figura D.2**: Funciones que permiten describir la **función de error** en un punto y la **función del coste** del modelo.
- **Figura D.3**: Función que calcula el **gradiente** en cada punto.
- **Figura D.4**: Función que calcula el **gradiente por lotes**.
- **Figura D.5**: Función que **actualiza los pesos** en la dirección del gradiente y con el sentido opuesto al mismo.
- **Figura D.6**: Función que devuelve la **clase con la máxima probabilidad estimada**.
- **Figura D.7**: Función que **ajusta los parámetros del modelo**.
- **Figura D.8**: Función que **inicializa todos los parámetros y variables** del modelo.

Nótese que, en los códigos recogidos en las figuras, LaTeXno permite escribir palabras acentuadas ni letras 'ñ', motivo por el cuál estas has sido sustituidas por letras sin acentuar o letras 'n' respectivamente.

```
        import numpy as np

        # Devuelve la probabilidad estimada de la caracteristica <x>
        # de pertenecer a cada una de las posibles clases
        def probabilidades(self,x):
            productoEscalar = [ (np.dot(self.w[i,:],x)) for i in
                range(self.nClases) ]
            exponenciales = np.exp(productoEscalar)
            sumatorio = np.sum(exponenciales)

            # Comprobamos que no haya errores
            if sumatorio <= 0 or np.isnan(sumatorio):
                print("\033[31mERROR:")
                print(f"Exponenciales: \n\t{exponenciales}")
                print(f"Suma de probabilidades: \n\t{sumatorio}")
                print(f"Pesos del modelo: \n{self.W}")
                print(f"Muestra: \n{x}")
                print(f"Exponentes: {productoEscalar}\033[0m")
                exit(0)

            return exponenciales / sumatorio

        # Devuleve la probabilidad estimada dadas las
        # caracteristicas <x> de pertenecer a la clase <y>
        def probabilidadEstimada(self, x, y):
            return self.probabilidades(x)[y-1]
```

Figura D.1: Arriba: Función para calcular la probabilidad de pertenencia a cada clase. Abajo: Función que permite calcular la probabilidad de pertenecer a una clase determinada.

```
        import numpy as np

        # Devuelve el error cometido para el punto <x> si se deberia
        # clasificar como perteneciente a la clase <y>
        def funcionDeCosteEnUnPunto(self, x, y):
            return -np.log(self.probabilidadEstimada(x,y))


        # Devuelve el valor de la funcion de coste evaluada en todos
        # los puntos
        def funcionDeCoste(self):
            costes = [
                (self.funcionDeCosteEnUnPunto(self.X_train.iloc[i],
                self.Y_train.iloc[i])) for i in range
                (self.X_train.shape[0]) ]

            return np.sum(costes)
```

Figura D.2: Funciones de error y de coste.

```
        import numpy as np

        # Calcula el gradiente para el punto <x> con clase <y>
        def gradiente(self, x, y):
            # Calculamos la probabilidad de pertenecer a cada clase
            probabilidades = np.reshape(self.probabilidades(x), (1,
                self.nClases))

            # Restamos la funcion indicador
            probabilidades[0,y-1] = probabilidades[0,y-1] - 1

            # Realizamos el producto matricial
            return np.dot(probabilidades.T, np.matrix(x))
```

Figura D.3: Función de cálculo del gradiente.

```
        import numpy as np

        # Calcula el gradiente de un lote comenzando en la muestra
        # situada en la posicion <inicio>
        def gradienteIteracion(self, inicio):
            gradiente = np.zeros(self.w.shape)

            for i in range(self.tam_iteraciones):
                x = self.X_train.iloc[inicio + i]
                y = self.Y_train.iloc[inicio + i]
                gradiente = gradiente + self.gradiente(x,y)

            return gradiente
```

Figura D.4: Función de cálculo del gradiente por lotes.

```
        import numpy as np

        # Actualiza los pesos en la direccion de <gradiente> con el
        # sentido opuesto. Devuelve True si hay actualizacion, False
        # en caso contrario
        def actualizarPesos(self, gradiente):
            norma = np.linalg.norm(gradiente)
            if norma != 0:
                self.w = self.w - self.tasa_aprendizaje * gradiente /
                    norma
                return True
            else:
                return False
```

Figura D.5: Función de actualización de pesos.

```
        # Devulve la clase con mayor probabilidad para <x>
        def claseEstimada(self, x):
            probabilidades = self.probabilidades(x)
            return list(probabilidades).index(max(probabilidades)) + 1,
                probabilidades
```

Figura D.6: Función que calcula la clase estimada.

```
        import time

        # Ajustamos el modelo con los datos disponibles para
        # el entrenamiento
        def fit(self):
            inicio_global = time.time()
            # Abrimos los ficheros de escritura de resultados
            toca_actualizar_tasa_aprendizaje = False
            with open(f"fichero_accuracy_{self.indice}.txt", "w") as
                f_accuracy:
                with open(f"fichero_coste_{self.indice}.txt", "w") as
                    f_coste:
                    # Calculamos la accuracy y el coste iniciales
                    inicio = time.time()
                    anterior_accuracy, anterior_coste =
                        self.accuracyCostes()
                    self.guardarYMostrarDatos(0, anterior_coste,
                        anterior_accuracy, f_coste, f_accuracy, inicio)

                    # Bucle de aprendizaje
                    for it in range(self.num_iteraciones):
                        inicio = time.time()

                        # Calculamos el gradiente del lote
                        gradiente = self.gradienteIteracion(it*
                            self.tam_iteraciones)

                        # Actualizamos los pesos con lo obtenido
                        if not self.actualizarPesos(gradiente):
                            print(f"Los pesos ya no se actualizan mas")
                            return

                        # Calculamos accuracy y coste actuales
                        accuracy, coste = self.accuracyCostes()
                        self.guardarYMostrarDatos(it+1, coste,
                            accuracy, f_coste, f_accuracy, inicio)

                        # Actualizamos la tasa de aprendizaje
                        if coste > anterior_coste:
                            if toca_actualizar_tasa_aprendizaje:
                                self.tasa_aprendizaje =
                                    self.tasa_aprendizaje / 2
                                toca_actualizar_tasa_aprendizaje = False
                            else:
                                toca_actualizar_tasa_aprendizaje = True

                        anterior_accuracy = accuracy
                        anterior_coste = coste

                    f_coste.close()
                f_accuracy.close()
            final_global = time.time()
            print(f"\n\n\nTiempo total empleado:
                {final_global - inicio_global}\n\n")
            return
```

Figura D.7: Función que ajusta el modelo.

```
        import numpy as np
        import pandas as pd

        # Inicializa los parametros del modelo
        def __init__(self, X, Y, tasa_aprendizaje, indice):
            # Anadimos la columna de unos
            X['Ones'] = np.ones(len(X))

            # Dividimos conjunto de aprendizaje y test
            self.X_train, self.X_test, self.Y_train, self.Y_test =
                train_test_split(X, Y, test_size=0.2, random_state=1)

            # Calculamos el numero de clases distintas
            self.nClases = Y.nunique().iloc[0]

            # Definimos la matriz de pesos
            self.w = np.zeros((self.nClases,X.shape[1]))

            # Almacenamos la tasa de aprendizaje
            self.tasa_aprendizaje = tasa_aprendizaje
            self.indice = indice

            # Numero de iteraciones de entrenamiento y su tamano
            self.num_iteraciones = 50
            self.tam_iteraciones = int(self.X_train.shape[0] /
                self.num_iteraciones)

            # Matriz de confusion
            self.confusion =
                np.zeros((self.nClases,self.nClases),dtype=int)

            # Imprimir informacion
            print(f"Numero de clases: {self.nClases}")
            print(f"Tasa de aprendizaje: {self.tasa_aprendizaje}")
            print(f"Numero de iteraciones: {self.num_iteraciones}")
            print(f"Tamano de las iteraciones: {self.tam_iteraciones}")
            print(f"Tamano del conjunto de entremaiento:
                {self.X_train.shape[0]}")

            return
```

Figura D.8: Función que inicializa los parámetros del modelo.

Apéndice E

Funciones de métricas de rendimiento

Este anexo recoge la implementación de las **funciones que permiten evaluar el rendimiento** que presenta nuestro modelo.

- **Figura E.1**: Función que calcula la **matriz de confusión** del modelo.
- **Figura E.2**: Funciones que calculan la métrica de **accuracy** del modelo.
- **Figura E.3**: Calcula la **precisión** de cada clase.
- **Figura E.4**: Calcula el **recall** de cada clase.
- **Figura E.5**: Función que calcula el **accuracy** y el valor de la **función de coste** de forma simultánea, siendo así más **eficiente**.

Nótese que, en los códigos recogidos en las figuras, LaTeXno permite escribir palabras acentuadas ni letras 'ñ', motivo por el cuál estas has sido sustituidas por letras sin acentuar o letras 'n' respectivamente.

```
1         import numpy as np
2
3         # Rellena la matriz de confusion de la muestra a partir
4         # de los datos de <X> y sus respuestas asociadas en <Y>
5         def matrizConfusion(self, X, Y):
6             self.confusion =
                  np.zeros((self.nClases,self.nClases),dtype=int)
7             for i in range(X.shape[0]):
8                 clase_estimada, _ = self.claseEstimada(X.iloc[i])
9                 self.confusion[Y.iloc[i]-1,clase_estimada-1] += 1
10            return
```

Figura E.1: Función que calcula la matriz de confusión.

```
        # Devuelve el accuracy del modelo
        def accuracyTrain(self):
            self.matrizConfusion(self.X_train, self.Y_train)
            return sum((self.confusion[i,i]) for i in
                range(self.nClases)) / self.X_train.shape[0]

        # Devuelve el accuracy del modelo con el conjunto de test
        def accuracyTest(self):
            self.matrizConfusion(self.X_test, self.Y_test)
            return sum((self.confusion[i,i]) for i in
                range(self.nClases)) / self.X_test.shape[0]
```

Figura E.2: Arriba función que calcula el accuracy en el conjunto de entrenamiento. Abajo la función que calcula el accuracy en el conjunto de test.

```
        # Devuelve la precision de cada clase. Requiere haber
        # calculado previamente la matriz de confusion.
        def precisionPorClases(self):
            print(f"\033[32m\nPrecision por clases:\033[0m")
            for i in range(self.nClases):
                print(f"\tClase {i+1}: \t{self.confusion[i,i] /
                    sum(self.confusion[:,i])}")
```

Figura E.3: Función que calcula la precisión.

```
        # Devuelve el recall de cada clase. Requiere haber
        # calculado previamente la matriz de confusion.
        def recallPorClases(self):
            print(f"\033[32m\nRecall por clases:\033[0m")
            for i in range(self.nClases):
                print(f"\tClase {i+1}: \t{self.confusion[i,i] /
                    sum(self.confusion[i,:])}")

            return
```

Figura E.4: Función que calcula el recall.

```
        import numpy as np

        # Calculo del accuracy y del valor de la funcion de coste
        # de forma mas eficiente. Devuelve <accuracy>,<funcion de coste>
        def accuracyCostes(self):
            aciertos = 0
            costes = 0

            # Bucle sobre todos los datos
            for i in range(self.X_train.shape[0]):
                clase_estimada, probabilidades =
                    self.claseEstimada(self.X_train.iloc[i])

                costes = costes -
                    np.log(probabilidades[self.Y_train.iloc[i]-1])
                if clase_estimada == self.Y_train.iloc[i]:
                    aciertos = aciertos + 1

            return aciertos / self.X_train.shape[0], np.double(costes)
```

Figura E.5: Función que calcula el accuracy y el valor de la función de coste.

Bibliografía

[1] IBM. ¿qué es un árbol de decisión? URL: https://www.ibm.com/es-es/topics/decision-trees. Consultado en marzo de 2024.

[2] Wikipedia. Arbol de decisión. URL: https://es.wikipedia.org/wiki/%C3%81rbol_de_decisi%C3%B3n#/media/Archivo:Arbol_decision.jpg. Consultado en marzo de 2024.

[3] Nikhil Agnihotri. Classification of machine learning algorithms. URL: https://www.engineersgarage.com/machine-learning-algorithms-classification/. Consultado en junio de 2024.

[4] StatLogos Web. Introducción a la regresión logística en 2024. URL: https://statologos.com/regresion-logistica/#:~:text=Aqu%C3%AD%20hay%20algunos%20ejemplos%20de%20cu%C3%A1ndo%20podr%C3%ADamos%20usar,respuesta%20%3D%20%C2%ABreclutado%C2%BB%20o%20%C2%ABno%20reclutado%C2%BB%29%20M%C3%A1s%20elementos. Consultado en marzo de 2024.

[5] Julian Cardenas. Odd ratio: Qué es y cómo se interpreta. URL: https://networkianos.com/odd-ratio-que-es-como-se-interpreta/. Consultado en marzo de 2024.

[6] Geeks for Geeks Web. Cost function in logistic regression. URL: https://www.geeksforgeeks.org/ml-cost-function-in-logistic-regression/. Consultado en marzo de 2024.

[7] Remy Lau. Cross-entropy, negative log-likelihood, and all that jazz. URL: https://towardsdatascience.com/cross-entropy-negative-log-likelihood-and-all-that-jazz-47a95bd2e81. Consultado en junio de 2024.

[8] Libre Texts Web. Las derivadas direccionales y las propiedades del gradiente. URL: https://espanol.libretexts.org/Matematicas/Calculo/Calculo_en_Varias_Variables_(ETS_Ingenieria_de_la_Universidad_de_Sevilla)/1._DERIVADAS_PARCIALES/1.6._Las_derivadas_direccionales_y_las_propiedades_del_gradiente. Consultado en marzo de 2024.

[9] Salim Oyinlola. Gradient descent - machine learning algorithm example. URL: https://www.freecodecamp.org/news/gradient-descent-machine-learning-algorithm-example/. Consultado en marzo de 2024.

[10] Shuiwang Ji y Yaochen Xie. Logistic regression: From binary to multi-class. URL: https://people.tamu.edu/~sji/classes/LR.pdf. Consultado en abril de 2024.

[11] Admond Lee. Comprensión de la matriz de confusión y cómo implementarla en python. URL: https://www.datasource.ai/es/data-science-articles/comprension-de-la-matriz-de-confusion-y-como-implementarla-en-python. Consultado en junio de 2024.

[12] Natasha Sharma. Understanding and applying f1 score: Ai evaluation essentials with hands-on coding example. URL: https://arize.com/blog-course/f1-score/#:~:text=F1%20score%20computes%20the%20average,of%20the%20predictions%20were%20correct. Consultado en marzo de 2024.

[13] Jock Blackard. Covertype. UCI Machine Learning Repository, 1998. DOI: https://doi.org/10.24432/C50K5N.