Exploración y preprocesado de datos

01 - Tipos de datos

- Usamos datos tabulados
  - N ~ filas ~ ejemplos
  - D ~ columnas ~ dimensiones

Discretos [1,2,3]

Continuos [45.6, 78.6]

Categóricos
- Pueden tener o no relación de orden.
- Ej: ['rojo', 'azul'] vs ['Nunes','Martes']
- Requieren codificación (relación biyectiva)

Codificación en pandas

Codificación con enteros

Util cuando las categorias tienen un orden intrínseco

Reduce la dimensionalidad de los datos

Nda eficiente

Puede introducir un orden artificial

Manualmente XDDDD

- zip(X, Y) = toma dos o más secuencias iterables (como listas, tuplas, etc.) y las combina en una sola secuencia de tuplas.
- dict --> crea un diccionario con dos argumentos con claves o lista de tuplas
- unique() df_categorico[Atributo 1].unique() # pilla valores unicos de una columna
- map() --> para aplicar una funcion a cada elemento del iterable y devolver los resultados como un nuevo iterable [columna]

1 sola categoria

Python ofign el cod:

```
codes1 = df_categorico[Atributo 1].cat.codes
categ1 = df_categorico[Atributo 1].cat.categories
code_to_categ1 = dict(zip(codes1, df_categorico[Atributo 1]))
```

Usamos `'cat.codes' y 'cat.categories' y creamos un diccionario.

Ejemplos resumen:

```
categorias1 = df_categorico[Atributo 1].unique()
categ_to_code1= {string: i for i, string in enumerate(categorias1)}
codes1 = df_categorico[Atributo 1].map(categ_to_code1)
```

Creamos diccionario y usamos .map() para aplicar

Todas a la vez

```
# 1) Identificamos cols categ/num
    cat_cols = df_categorico.select_dtypes(include='category').columns.tolist()

# 2) Creamos una df con las codificadas
    df_calculo = pd.DataFrame(columns=cat_cols)

# 3) Bucle para recorrer y codificar. mientras hacemos un diccionario de diccionarios para descodificar despues

    for col in cat_cols:
        codes = df_categorico[col].cat.codes
        categ = df_categorico[col].cat.categories
        df_calculo[col] = codes
        dict_decod[col] = code_to_categ
```

{'Alfa': 0, 'Beta': 1, 'Gamma': 2} {0: 'Alfa', 1: 'Beta', 2: 'Gamma'}

LabelEncoding - con sklearn

Label Encoder

```
from sklearn.preprocessing import LabelEncoder
import pandas as pd

# df = pd.DataFrame({'Atributo Categórico': ['Clase', 'Perro', 'Playa']})

# Label Encoder es una clase, tenemos que instanciarla como objeto:
    le = LabelEncoder()
    df['Atributo Categórico'] = le.fit_transform(df['Atributo Categórico'])

# Para ver el diccionario de codificación (nos vamos a los atributos del objeto):
    print(le.__dict__)
```

Codificación One-hot

Conserva info de todas las categorias

Adecuado para algoritmos que no manejan variables ordinales

Aumenta la dimensionalidad de los datos: menos eficiente

Una columna por cada categoría distinta

```
# Decodificamos el X1:
df_one_hot = pd.get_dummies(df_categorico['Atributo 1'])

# Juntamos DFs:
df2= pd.concat([df_categorico,df_one_hot], axis=1)
```

Obtenemos tantas columnas dummies como categorias haya en X1

02 - Exploración, preprocesado y separación

Exploración de los datos

Descripción básica estadística

N.D = df.shape → Nos da tamaño: Ejemplos y Dimensión

trainSet.dtypes / trainSet.info() → Tipos de datos; .info() da también cuantos tipos diferentes hay

trainSet.describe() → Descripción estadísticos -moda

trainSet.mode(axis=0, dropna=False)

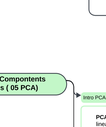
#dropna=False es para ignorar valores nulos

Visualización de la distribución de los datos

Para obtener info **cualitativa** y hacernos una idea de las distribuciones :

Usamos método de pandas: .plotting.scatter_matrix

```
pd.plotting.scatter_matrix(df, alpha=0.1)
```



- El gráfico (i,j) es el transpuesto de (j,i).
- En la diagonal se muestra un histograma de los valores que tuma cada atributo.
- Dado que no hay distribuciones normales, los valores típicos de media y desviación típica no son estimadores de la distribución.
- Si los histogramas tienen dos máximos, son **distribuciones multimodales**

Poca práctica si hay muchas cols

03 - Ingeniería de características

Preprocesado de los datos

Tratamiento de valores perdidos

Localización

Buscar

`replace("?",pd.NA)` En algunos df, utilizan "?" hay que reemplazarlo por el tipo de datos NA de pandas

`isnull() o isna()`

```
df = df.replace("?", pd.NA)
missing_data = df.isna()
```

df.isnull().sum(axis=1) #columnas

#filas (entiendo axis=0)

Eliminación

Eliminar valores nulos

`df.dropna()`

```
df_sin_nulos_filas = df.dropna()
df_sin_nulos_columnas = df.dropna(axis=1)
```

Eliminamos todos los que tengan algún nulo

```
# Eliminamos si hay muchos atributos perdidos

umbral_filas = 0.2  # Por ejemplo, definimos un umbral del 20%
filas_a_eliminar = df[df.isnull().mean(axis=1) > umbral_filas].index
df_sin_filas = df.drop(filas_a_eliminar)

umbral_columnas = 0.2  # Por ejemplo, definimos un umbral del 20%
columnas_a_eliminar = df.columns[df.isnull().mean() > umbral_columnas]
df_sin_columnas = df.drop(columnas_a_eliminar, axis=1)
```

Imputación

Imputación Univariada → Asignar un estadístico a todos los NaN

`df.fillna(df.median)` # Lo cambiamos por algun estadístico: median, mean, mode

```
df_imputado= df.fillna(df.median(axis=0))
```

- Opcionalmente se puede añadir una columna a la derecha indicando que se han imputado valores (0/1).

Problemas:
- No hacer cuando son marcas de tiempo → usariamos interpolate()
- Ojo si caso, por ejemplo, si falta el sexo, podemos asignar femenino a un cancer de próstata o similares sin sentido.

Imputación Multivariada → Calculamos el NaN con regresión del resto de atributos

Estimamos el valor faltante con el resto de columnas con regresión:

$$f(x_1, x_2, x_4) = \hat{x}_3$$

Para varios valores faltantes el proceso se repite de forma iterativa, calculando el valor correspondiente. Lo cual es goofy. Usaremos IterativeImputer de sklearn, que veremos más adelante.

sklearn.impute.IterativeImputer

Aumento de la dimensionalidad

Características generadas por nosotros

Si tenemos conocimiento del campo del problema, ej: densidad de población:

```
trainDF['population_density'] = trainDF['population'] / trainDF['households']
```

Características polinómicas

```
from sklearn.preprocessing import PolynomialFeatures
    degree = 2
    interaction_only = False  #Hace todas las combinaciones, si lo ponemos en True hace solo las nuevas

    polyf = PolynomialFeatures(degree=2,interaction_only=False)

    polyf.set_output(transform="pandas")

    polyf.fit(df)
    df_poly = polyf.transform(df)
```

No hacer fit con test, solo la primera vez(train) para evitar sobre ajuste.

- **¿Por qué aparece una columna de unos a la izquierda del dataframe transformado?**

  Se puede eliminar con la opción `include_bias=False` , aunque esta columna se usa para calcular el término independiente de un modelo.

Escalados (no alteran la dimensión)

Escalado al intervalo unidad → MinMaxScaler

Hace que todas las características tengan la misma escala

$$z_{esc} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

```
from sklearn.preprocessing import MinMaxScaler

# Escalado al intervalo unidad
scalerUnit = MinMaxScaler().set_output(transform="pandas")
scalerUnit.fit(df)
#El objeto almacena max + min de cada col; podemos verlo con scalerUnit.__dict__
scaleUnit_df = scalerUnit.transform(df)
```

Escalado al máximo en valor absoluto → MaxAbsScaler

Escala cada columna de forma que: { 1. Si todos son positivos, [0,1]; 2. Si todos negativos, [-1,0]; 3. Hay ambos, [-1,1] }

$$x_{esc} = \frac{x}{\max(|x|)}$$

```
from sklearn.preprocessing import MaxAbsScaler

# Escalado al maximo de los valores absolutos
scalerMaxAbs= MaxAbsScaler().set_output(transform="pandas")
scalerMaxAbs.fit(df) #Almacena max de cada col
scaleMaxAbs_df= scalerMaxAbs.transform(df)
```

Estandarización → StandardScaler

Para cada columna hace: media=0 y desviación típica=1

Esto no las convierte en distribución normal.

$$z_{std} = \frac{x - \mu}{\sigma}$$

```
from sklearn.preprocessing import StandardScaler

#  Estandarización
scalerStd = StandardScaler().set_output(transform="pandas")
scalerStd.fit(df) #Almacena media y desviación
scaleStd_df = scalerStd.transform(df)
```

Reducción de la dimensionalidad

Filtrados univariados (04 Reducción y filtrado)

Filtrado por varianza

Recordamos que la varianza es el promedio de las diferencias con la media elevadas al cuadrado. Cuando la varianza se acerca a 0, **hay poca variabilidad en los datos**, por lo que aportaran poca información a nuestro modelo.

$$Var(x) = \frac{1}{N}\sum_{i=1}^{N}(\mu - x_i)^2 \qquad \sigma(x) = \sqrt{Var(x)},$$

pandas: df.var(ddof=0) #degrees of freedom

```
from sklearn.feature_selection import VarianceThreshold
threshold=0.1

selector = VarianceThreshold(threshold=threshold)
selector.set_output(transform="pandas")
selector.fit(dfboo)
df_filtered = selector.transform(dfboo)
```

¡¿CUIDADO?!
La varianza es sensible al orden de magnitud de los datos:
Var(c · X) = c^2 · Var(X)
Es decir, **tanto si hacemos Escalado al intervalo unidad o Estandarizar y queremos filtrar por varianza, debemos hacerlo antes.**

Filtrado por correlación

La correlación entre dos variables mide la relación lineal entre dos variables, por lo que con tener en cuenta una para nuestro modelo será suficiente.

Por eso filtraremos las que tengan correlación **próxima a +1 y -1** en la matriz de correlación.

```
import matplotlib.pyplot as plt

corr_mat = df.corr()
plt.matshow(corr_mat, cmap='Blues')
plt.colorbar()
plt.show()
```

- No se ve afectado por la escala.
- Más eficiente en problemas de regresion.

Filtrado por información mutua

La información mutua o ganancia de información mide la reducción de la incertidumbre (entropía) de una variable aleatoria X cuando se conoce el valor de otra Y.

Información mutua y Entropía son conceptos que pertenecen a la Teoría de la Información, y son la base teórica de los árboles de decisión. Por tanto dejamos esta técnica para más adelante, cuando expliquemos ese tema.

Principal Components Analysis (05 PCA)

Intro PCA + sklearn

PCA es una técnica para **sustituir** con nuevas características mediante combinaciones lineales de las que tenemos inicialmente. (Mientras la pérdida de info con X componentes o Y%)

- d<<D (**vista y nueva dimensionalidad**)
- **Para "eliminar un vector" proyectamos en otro, para perder la mínima info posible.**
- **El resultado de Z será una combinación lineal de las características de X.**

1. Estandarizar

Para algoritmos basados en distancias

2. PCA

```
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
# Estandarizar
scalerStd = StandardScaler().set_output(transform="pandas")
scalerStd.fit(df) #Almacena media y desviación
scaledStd_df = scalerStd.transform(df)

# PCA
n_components = 2  # Si ponos 0.9 pilla descomerme el ratio de varianza explicada total
pca = PCA(n_components = n_components).set_output(transform="pandas")
pca.fit(scaledStd_df)
df_pca = pca.transform(scaledStd_df)

# Ver varianza explicada
variance_ratio = pca.explained_variance_ratio_
```

Autovectores y autovalores:

$Au = \lambda u$

Propiedades:
- Una matriz A non tiene n autovalores si |A| = 0
- Matriz cuadrada A ∈ ℝn×n tiene n autovalores si |A|≠0
- Sea λ ∈ ℝ+nm (por tanto NO cuadrada), si λ es autovalor de XᵀY X también lo es de X·Xᵀ·Y

Matriz de covarianza Σ

$$\Sigma_{i,j} = \frac{1}{N}\sum_{k=1}^{N}\left((x_i^{(k)} - \mu_i)(x_j^{(k)} - \mu_j)\right)$$

Como estandarizamos, μ=0; nos queda simplemente:

$$\Sigma_{i,j} = \frac{1}{N}\sum_{k=1}^{N}\left(x_i^{(k)} \cdot x_j^{(k)}\right) = x_i^{\top} x_j$$

Y aplicamos para calcular eficientemente: $\Sigma = \mathbf{X}^\top \mathbf{X}$

Explicación teórica chill

La matriz de covarianza nxn es simétrica, por lo tanto al aplicar una transformación lineal, encontramos **n vectores perpendiculares**, tantos como el tamaño de la matriz. --> De ahí sacamos base ortonormal (vectores unitarios).

$$\Sigma = \begin{bmatrix} Var(X_1) & Cov(X_1,X_2) & \cdots & Cov(X_1,X_n) \\ Cov(X_n,X_1) & Var(X_2) & \cdots & Cov(X_2,X_n) \\ \vdots & \vdots & \ddots & \vdots \\ Cov(X_n,X_1) & Cov(X_n,X_2) & \cdots & Var(X_n) \end{bmatrix}$$

autovalor / autovector: $\Sigma \cdot v = \lambda \cdot v$

Sacar autovalor: polinomio característico

$|\lambda \cdot \mathbb{I} - \Sigma| = 0$ → las raíces de la ecuación lis autovalores

Sacar autovectores:

Despejamos v en cada ecuación: $\Sigma \cdot v + a1 \cdot v \rightarrow$ v1, ..., vn

$\Sigma \cdot v + a2 \cdot v$

Proceso PCA:
1. Matriz covarianza - sacamos autovalores y autovectores.
2. Se ordenan los autovectores de mayor a menor (segun su autovalor)
3. Normalizamos autovectores seleccionados (base ortonormal)
4. Multiplicamos matriz original (X) por la base, de forma que se reduce su dimensión con autovectores principales.

PCA:

Una vez tenemos v1,..., vn, ordenamos de mayor a menor (según autovalores, ya - los autovalores representan varianza recibe los autovectores recibe un [1,0]), proyectaremos en el hiperplano definido por los autovectores. Recordemos que los vectores serán perpendiculares entre sí.

minimizar residuo: $\arg\min_{\mathbf{u}} \left\{ \sum_{i=1}^{m} \mathbf{u}_1^\top \mathbf{x}_i \right\}$ = $\arg\max_{\mathbf{u}} \left\{ \sum_{i=1}^{m} \mathbf{u}_1^\top \mathbf{x}_i \right\}$ → maximizar proyección

Problema de optimización cuadrática con restricciones: método multiplicadores de Lagrange. (Completo en colab).

Luego, multiplicamos la matriz de datos originales (X) por estas 4 columnas de la matriz de autovectores para proyectar los datos en el nuevo espacio de menor dimensión definido por los autovectores principales, obteniendo Z.

Cada ejemplo de Z es una combinación lineal de las características de X.

Entrenamiento

Supervised learning

Cada ejemplo está emparejado a un valor objetivo o a una etiqueta.

- *X (datos)*
- *Y – ground truth*

07- Modelos lineales

$$f(\mathbf{x}; \mathbf{w}) = w_0 + w_1 x_1 + \cdots + w_D x_D.$$

$\mathbf{w} = (w_1, \ldots, w_D)^\top$ → Vector característico del hiperplano del modelo

$w_0 + \mathbf{w}^\top \mathbf{x} = 0.$ → Término independiente

Linear Regresion

Regresión Lineal

La regresión consiste en aprender a emparejas los ejemplos dados con un valor objetivo o target que toma valores en un intervalo continuo.

El modelo de regresión lineal es lo que obtenemos **al ajustar por mínimos cuadrados un conjunto de pares de puntos**, pero generalizado a **D≥2 dimensiones**.

```
from sklearn.import linear_model

reg = linear_model.LinearRegression() #Objeto
reg.fit(X_train,Y_train)
y_hat1 = reg.predict(X_test)

#Para ver los parámetros aprendidos, vemos atributos del objeto:
w= reg.coef_  #pendiente
w0= reg.indercept_  #indep. term.
```

Cálculo matricial "manual" (costoso computacionalmente):

$$\mathbf{Y} = \mathbf{X}\mathbf{w}\,,$$
$$\mathbf{X}^\top\mathbf{Y} = \mathbf{X}^\top\mathbf{X}\mathbf{w}\,,$$
$$(\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{Y} = (\mathbf{X}^\top\mathbf{X})^{-1}(\mathbf{X}^\top\mathbf{X})\mathbf{w}\,,$$
$$(\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{Y} = \mathbf{I}\mathbf{w}\,,$$
$$\mathbf{w}^* = (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{Y}$$

```
from sklearn.preprocessing import add_dummy_feature
import numpy as np
def matrix_solution(X, Y):
    X = add_dummy_feature(X)
    XtX = np.matmul(X.T, X)
    invXtX = np.linalg.inv(XtX)
    w_star = np.matmul(invXtX, X.T)
    w_star = np.matmul(w_star, Y)
    return w_star
w_star = matrix_solution(X,Y)
y_hat2 = w_star[0] + np.matmul(X,w_star[1:])
```

Para hacerlo manualmente, hay que añadir variable dummy

Clasificación con regresión lineal

Sigue siendo una regresión, pero con una **función discriminante**: **Normalmente la función Signo.**

Función Discriminante → **Ecuación explícita** → **Plano de decisión**

$$\hat{y} = \begin{cases} 0 & \text{si} \quad f(\mathbf{x};\mathbf{w}) > \theta \\ 1 & \text{si} \quad f(\mathbf{x};\mathbf{w}) \leq \theta \end{cases} \qquad w_0 + w_1 x_1 + w_2 x_2 = 0 \qquad x_2 = \left(-\frac{w_1}{w_2}\right) + \left(-\frac{w_1}{w_2}\right) x_1$$

```
X = df[['x1','x2']]
Y = df['y']
reg.fit(X,Y)

def decision_plane(reg, x1):
    w0 = reg.intercept_
    w = reg.coef_
    x2 = (-w0/w[1]) + (-w[0]/w[1])*x1
    return x2

x1 = np.linspace(-4,4,10)
x2 = decision_plane(reg, x1)

colors=df['y'].copy()  # Colorines
colors[colors==-1]='r'
colors[colors==1]='b'
df.plot.scatter(x='x1',y='x2',c=colors)
plt.plot(x1,x2,'k')
plt.show()
```

El objeto reg solo hace falta instanciarlo una vez, se puede volver a aprender con otros datos. (Por eso no se ha añadido)

Aplicar formula x2: (fijarse que reg._intercept_ es w0 y las w asociadas a cada 'x' las tenemos en el vector w ... empezamos con w[0] (sería w1), w[1] (sería w2)

Logisic Regresion

Explicacion

Necesitamos una función "sigmoide", con forma de S, que cumple las condiciones para ser una **distribución de probabilidad**.

Nombre	Expresión
Logística	$S(x) = \frac{1}{1+e^{-x}}$
Tangente hiperbólica	$S(x) = \tanh(x)$
Arcotangente	$S(x) = \arctan(x)$
Función error	$S(x) = \frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2} dt$
y también funciones algebraicas como	$S(x) = \frac{x}{\sqrt{1+x^2}}$

En Linear Regression, la decisión de asignar una etiqueta se hacía según quede arriba o abajo de la regresión, pero podía estar muy cerca o muy lejos de la línea. Eso nos da igual.

Con logistic regression **saturamos** el valor de la función f(x,w) para q la probabilidad de pertenecer a una etiqueta sea 1 o 0.

#Logit es la función lineal

$$\Pr(y = +1 | \mathbf{x}; \mathbf{w}) = \frac{1}{1 + e^{-\mathbf{w}^\top \mathbf{x}}}$$

Linear/Regression + Función Logistica

```
#-- Obtenemos los logits
logit = reg.predict(X)

#-- Convertimos los logits en probabilidad de pertenecer a la clase "1"
def logistic_fun(x):
    return 1 / (1 + np.exp(-x))

df["logit"]=logit
df["p(y=1)"]=logistic_fun(logit)
df.head()
```

Logistic Regression

```
from sklearn import linear_model
logreg = linear_model.LogisticRegression()

logreg.fit(X,Y)
y_hat = logreg.predict(X)  # Prediccion de clase
prob = logreg.predict_proba(X)  # clase + prob

# el método predit_proba() devuelve 2 columnas. una para cada clase.
```

LogisticRegression en SKL tiene un término de regularización L2 por defecto (penalty="l2")

- L1(Regularización de Lasso) penaliza la suma de los valores absolutos de los coeficientes.
- L2 (Regularización de Ridge) penaliza la suma de los cuadrados de los coeficientes

LR: función discriminante

La regresión logística **también** tiene una función discriminante:

$$\hat{y} = \begin{cases} +1 & \text{si} \quad p \geq \theta \\ -1 & \text{si} \quad p < \theta \end{cases} \quad \text{donde} \quad p = \frac{1}{1 + e^{-(\mathbf{w}^\top \mathbf{x} + w_0)}}$$

$\cos\theta = 0.5$

Según el dominio de la aplicación, puede interesar ser más o menos laxos con el umbral.

10 - Minimizar función de pérdida

Algoritmos de optimización

Descenso del gradiente

Explicación

El gradiente es un vector que marca la dirección de máxima pendiente de una función en un punto dado.

Sea $\mathcal{L}(\mathbf{w}) : \mathbb{R}^D \rightarrow \mathbb{R}$, entonces el gradiente de $\mathcal{L}$ respecto de $\mathbf{w}$ es el vector:

$$\nabla_{\mathbf{w}}\mathcal{L} = \left[\frac{\partial \mathcal{L}}{\partial w_1}, \frac{\partial \mathcal{L}}{\partial w_2}, \ldots, \frac{\partial \mathcal{L}}{\partial w_D}\right]$$

- Lo/s ej una superficie en un espacio D+1 dimensional. (Con 2 valores, genera parábola 3D). **El (x,y) forma un espacio 2D, y la z la dimensión del error**
- Al "viajar" por R^D obtenemos valores de L(w)
- El gradiente en un punto concreto w, nos indica la dirección en la que hay que moverse para **SUBIR** legar más rápido que en ninguna dirección.

Una vez tenemos el espacio R (ejemplo parábora 3D), elegimos un punto w y nos vamos moviendo en la dirección marcada por el gradiente en ese punto, y así de forma sucesiva tal que:

$$\mathbf{w}^{(nuevo)} = \mathbf{w} + \nabla_{\mathbf{w}}\mathcal{L}(\mathbf{w})$$

Para encontrar el mínimo, siguiendo el gradiente hay que dar la vuelta a la función:

$$\arg\min_{\mathbf{w}} \mathcal{L}(\mathbf{w}) = \arg\max_{\mathbf{w}} \left(-\mathcal{L}(\mathbf{w})\right)$$

La función L debe ser **derivable** en todas las direcciones.

Algoritmo

El *Descenso del gradiente* es un algoritmo para localizar el mínimo de una función de manera iterativa.

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} \mathcal{L}(\mathbf{Y}, f(\mathbf{X}; \mathbf{w}))$$

```
[1]  w_now <- w_inicial
[2]  lr <- learning_rate
[3]  flag_stop <- False
[4]  iter <- 0
[5]  while not flag_stop:
[6]      w_new <- update_rule(w_now, lr, gradient_L)
[7]      w_now <- w_new
[8]      flag_stop <- eval_stop(iter, w_now, fun_L)
```

Regla de actualización:

$$\mathbf{w}^{(nuevo)} = \mathbf{w}^{(actual)} - \eta \cdot \nabla_{\mathbf{w}}\mathcal{L}(\mathbf{w}^{(actual)})$$

learning rate: No cuanto sube el gradiente, sino un %. Entonces nos va a ir a 1%

Condición de parada:
- El número iteraciones alcanza un tope
- El valor de L ya no decrece más.

Funciones de pérdida

Funciones de pérdida para regresión lineal

En estos casos $f(\mathbf{x};\mathbf{w}) : (\mathbb{R}^D; \mathbb{R}^D) \rightarrow \mathbb{R}$ y el valor objetivo, *y*, también pertenece a R.

Características:
- Calcular $\hat{\delta}_{(i)} = (y^{(i)} - f(\mathbf{x}^{(i)}, \mathbf{w}))$ para cada $i = 1, 2, \ldots, N$.
- Convertir $\hat{\delta}_{(i)}$ en un valor positivo para todo *i*.
- Hacer el promedio para todos los ejemplos.

MSE | Mean Squared Error

Creamos la clase MSE:

- ***Mean Squared Error*** $\mathcal{L}(\mathbf{Y}, f(\mathbf{X};\mathbf{w})) = \frac{1}{N}\sum_{i=1}^{N}\left(y^{(i)} - f(\mathbf{x}^{(i)};\mathbf{w})\right)^2$

```
class MSE:
    def __init__(self, F):
        self.F = F

    def val(self, x, yhat):
        return np.mean((yhat - self.F.val(x))**2)

    def grad_w(self, x):
        yhat= self.F.val(x)
        return  2*(yhat-yhat)*self.F.grad_w(x)
```

¿Añadimos np .para que acepte vectores y matrices

MAE | Mean Absolut Error

- ***Mean Absolute Error*** $\mathcal{L}(\mathbf{Y}, f(\mathbf{X};\mathbf{w})) = \frac{1}{N}\sum_{i=1}^{N}\left|y^{(i)} - f(\mathbf{x}^{(i)};\mathbf{w})\right|$

```
class MAE:
    def __init__(self, F):
        self.F = F

    def val(self, x, yhat):
        yhat= self.F.val(x)
        return np.mean(np.abs(yhat - yhat))

    def grad_w(self, x, yhat):
        yhat= self.F.val(x)
        return  np.sign(yhat-yhat)*self.F.grad_w(x)
```

Log-cosh

- ***Log-cosh*** $\mathcal{L}(\mathbf{Y}, f(\mathbf{X};\mathbf{w})) = \frac{1}{N}\sum_{i=1}^{N}\log\left(\cosh\left(y^{(i)} - f(\mathbf{x}^{(i)})\right)\right)$

```
class LogCosh:
    def __init__(self, F):
        self.F = F

    def val(self, x, yhat):
        yhat= self.F.val(x)
        return  np.log(np.cosh(yhat-yhat))

    def grad_w(self, x, yhat):
        yhat= self.F.val(x)
        return  np.tanh(yhat-yhat) * self.F.grad_w(x)
```

Con regularización (11)

Hasta aquí , solo hemos obtenido w* mediante descenso de gradiente respecto de pérdida, es decir, la única regla era minimizar:

$$\mathcal{L}(\mathbf{Y}, f(\mathbf{X};\mathbf{w}))$$

La regularización consiste en añadir más reglas, t.q. la función objetivo del problema de optimización se convierte en:

$$\mathcal{L}(\mathbf{Y}, f(\mathbf{X};\mathbf{w})) + R(\mathbf{w})$$

Hiperparámetros

- *Los parámetros son aquellos elementos que podemos utilizar para ajustar un modelo a nuestro conjunto de datos específico.*
- *Los hiperparámetros son aquellos elementos que se han añadido al modelo o al algoritmo para controlar la ejecución.*
- **0.20**
- *Meaning rate (gradient descent)* ---controla velocidad a la que viajamos por el espacio
- o para dara dar peso al regularizador
- / (elastic net) para dar más peso a Lasso.

Métodos de regularización en Regresión Lineal:

Regularización Lasso (L1)

Añadimos a la función de pérdida:

$$\mathcal{R}_E = \alpha \sum_{i=1}^{D} |w_i|$$

- α: 0: Mismo MP que en R.L.
- α > 0: Algunos parametros se anulan.
- ∞ : Todos los parámetros se anulan.

Norma-1 de w:

$$\|\mathbf{w}\|_1 = \sum_{i=1}^{D} |w_i|$$

- Finalidad: Tener menos parámetros.
- Viene de *Least Absolute Shrinkage and Selection Operator*, por la propiedad de seleccionar y reducir el número de parámetros.

- **SKL**: sklearn.linear_model.Lasso
- En otros modelos, "penalty='l1'"

Regularización Ridge (L2)

Añadimos a la función de pérdida:

$$\mathcal{R}_E = \frac{\alpha}{2} \sum_{i=1}^{D} (w_i)^2$$

- α: 0: Mismo MP que en R.L.
- α > 0: Impide que los parámetros se separen mucho o crezcan demasiado.
- ∞ : Todos los parámetros se anulan.

Norma-2 de w:

$$\|\mathbf{w}\|_2 = \frac{1}{2} \sum_{i=1}^{D} (w_i)^2$$

- El factor 1/2 se añade para que al derivar el término el exponente se cancele con él.
- **SKL**: sklearn.linear_model.Ridge
-

Regularización Elastic Net

Añadimos a la función de pérdida una combinación lineal de ambos:

$$\mathcal{R}_E = \alpha_1 \mathcal{R}_L + \alpha_2 \mathcal{R}_R = \alpha_1 \|\mathbf{w}\|_1 + \frac{\alpha_2}{2} \|\mathbf{w}\|_2^2$$

Para más control e interpretación de α1 y α2, se utiliza combinación lineal convexa:

$$\mathcal{R}_E = \alpha \left(r \cdot \mathcal{R}_L + (1 - r) \cdot \mathcal{R}_R\right)$$

r ∈ [0,1] (0 = Ridge, 1 = Lasso)

- **SKL**: sklearn.linear_model.ElasticNet
- En otros modelos, "penalty=elasticNet"

Gradiente respecto de pérdidas de regresión

Implementación

Creamos la clase F que contendrá el modelo: $f(\mathbf{x}, \mathbf{w}) = \mathbf{w}^\top \mathbf{x}$.

La clase tiene 2 métodos:
- val: _ devuelve función para un vector de parámetros y un ej
- grad_w: _ devuelve derivada del f respecto de w

```
class F:
    def __init__(self, w):
        self.w = w

    def val(self, x):
        return np.matmul(self.w.T, x).reshape(-1)  #nivel pa crear array

    def grad_w(self, x):
        return x
        # se hace la derivada respecto w_0 de (w*0.T *x = w;
```

Creamos la clase de la pérdida que usemos:
- **MAE**
- **MSE**
- **Log-cosh**

Creamos la clase Descenso Gradiente:

```
class GradientDescent:
    def __init__(self, loss, lr = 0.1, Niter = 10):
        self.loss = loss
        self.lr = lr
        self.Niter = Niter

    def update(self,x,ytrue):
        self.loss.F.w = self.loss.F.w - ( self.lr *
self.loss.grad_w(x,ytrue) )

    def fit(self,x,ytrue):
        steps = [self.loss.F.w]
        for k in range(self.Niter):
            self.update(x,ytrue)
            steps.append(self.loss.F.w)
        return steps
```

Ejecutamos:

```
# Ejecución del descenso del gradiente
fx = f(x_init)
Loss = MSE(fx)  #Modificamos pérdida
GD = GradientDescent(Loss, lr=learning_rate, Niter=num_iter)
steps = GD.fit(x,ytrue)
```

Evaluación

12- Evaluación de modelos

Calidad modelos regresión

La métrica típica en regresión es el valor:

$$R^2(y, \hat{y}) = 1 - \frac{\sum_{i=0}^{N}(y^{(i)} - \hat{y}^{(i)})^2}{\sum_{i=0}^{N}(y^{(i)} - \bar{y})^2}$$

media / estimación

Con esa división estamos comparando el cuadrado de los residuos (los errores que cometemos) con la varianza (la suma de todos ejemplos) con la dispersión de los targets respecto de su media.

$R^2 \leq 1$ sólo si todos los residuos son 0. Por tanto, **cuanto más cercano a 1, mejor es nuestro modelo.**

sklearn.metrics.r2_score

NOTA: También se puede utilizar la propia función de pérdida, el "problema" es que NO devuelve un valor acotado, dependerá de los datos. R^2 siempre tiene valor máximo 1.

Calidad modelos de clasificación binaria:

Matriz de confusión

Total population: n = P + N		Predicted condition	
		Positive (PP)	**Negative (PN)**
Actual condition	**Positive (P)**	**True positive (TP)**	**False negative (FN)**
	Negative (N)	**False positive (FP)**	**True negative (TN)**

Medidas derivadas:

- ***Precision***: *Precision=1 → todos los ejemplos estimados positivos lo eran.*
- ***Recall***: *Recall=1 →todos los ejemplos positivos han sido estimados correctamente.*
- ***F1-score***: *Media entre precision y recall*
- ***Accuracy***: *porcentaje aciertos*

Total population: n = P + N		Predicted condition		Informedness, bookmaker informedness (BM) = TPR + TNR − 1	Prevalence threshold (PT) = √(TPR × FPR) − FPR / TPR − FPR
		Predicted Positive (PP)	**Predicted Negative (PN)**		
Actual condition	**Positive (P)**	**True positive (TP), hit**	**False negative (FN), type II error, miss, underestimation**	**True positive rate (TPR), recall, sensitivity (SEN), probability of detection, hit rate** = TP / P = 1 − FNR	**False negative rate (FNR), miss rate** = FN / P = 1 − TPR
	Negative (N)	**False positive (FP), type I error, false alarm, overestimation**	**True negative (TN), correct rejection**	**False positive rate (FPR), probability of false alarm, fall-out** = FP / N = 1 − TNR	**True negative rate (TNR), specificity (SPC), selectivity** = TN / N = 1 − FPR
Prevalence = P / n	**Positive predictive value (PPV), precision** = TP / PP = 1 − FDR	**False omission rate (FOR)** = FN / PN = 1 − NPV	**Positive likelihood ratio (LR+)** = TPR / FPR	**Negative likelihood ratio (LR−)** = FNR / TNR	
Accuracy (ACC) = (TP + TN) / n	**False discovery rate (FDR)** = FP / PP = 1 − PPV	**Negative predictive value (NPV)** = TN / PN = 1 − FOR	**Markedness (MK), deltaP (Δp)** = PPV + NPV − 1	**Diagnostic odds ratio (DOR)** = LR+ / LR−	
Balanced accuracy (BA) = (TPR + TNR) / 2	**F1 score** = 2 PPV × TPR / PPV + TPR = 2 TP / 2 TP + FP + FN	**Fowlkes–Mallows index (FM)** = √PPV × TPR	**Matthews correlation coefficient (MCC)** = √TPR × TNR × FPR × FNR − √FNR × FPR × FOR × FNR / √TPR × TNR + FPR × FNR + TPR × FOR + FOR × FNR	**Threat score (TS), critical success index (CSI), Jaccard index** = TP / TP + FN + FP	

Curva ROC y AUROC

Dado un modelo que estima la probabilidad de $p(y=1|x)$, tal que $yhat=1$ si $p(y=1|x) > U$, la curva ROC es el trazado de pares (FPR Y TPR) cuando hacemos variar el umbral *U*.

La curva ROC es diferente a la métrz de confusión porque se calcula para todo el rango de posibles umbrales U, mientras que la matriz de confusión ya asume un U fijado.

Midiendo el área bajo la curva ROC (AUROC) podemos comparar modelos: cuanto mayor es AUROC, mejor es el modelo.

```
# Se hace entrenamiento regresion logistica:

from sklearn.metrics import roc_curve, auc, confusion_matrix

# Entrenamiento del modelo de regresión logística
model = LogisticRegression()
model.fit(trainX, trainY)

# Cálculo de las probabilidades
y_score = model.predict_proba(testX)[:, 1]

# Cálculo de la curva ROC y AUC
fpr, tpr, thresholds = roc_curve(testY, y_score)
roc_auc = auc(fpr, tpr)

# Visualización de la curva ROC
plt.plot(fpr, tpr)
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
```

01-fichero-entrenamiento

May 15, 2024

1 Aprendizaje Automático: Práctica sobre Diagnóstico de Enfermedades

1.1 Exploración y preprocesado de datos:

1.1.1 Exploración de datos

1. Escribir un informe con una descripción de cada atributo de la columna que sea informativa y útil.

[ ]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import pickle

from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import PolynomialFeatures, StandardScaler
from sklearn.decomposition import PCA
from sklearn.feature_selection import VarianceThreshold
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import confusion_matrix
from sklearn.metrics import roc_curve, auc
```

[ ]:
```
# Suponiendo que estás cargando desde un archivo CSV
X_train = pd.read_csv('/content/drive/MyDrive/TrainML/retoML1_X_train.csv',␣
 ↪delimiter=';')
Y_train=pd.read_csv('/content/drive/MyDrive/TrainML/retoML1_Y_train.csv',␣
 ↪delimiter=';')
X_test = pd.read_csv('/content/drive/MyDrive/TrainML/retoML1_X_test.csv',␣
 ↪delimiter=';')

# Ahora X_train debería tener múltiples columnas, separadas correctamente
print(X_train.columns)  # Verifica las columnas
```

```
Index(['Unnamed: 0', 'age', 'bp', 'sg', 'al', 'su', 'rbc', 'pc', 'pcc', 'ba',
'bgr', 'bu', 'sc', 'sod', 'pot', 'hemo', 'pcv', 'wbcc', 'rbcc', 'htn', 'dm',
```

```
'cad', 'appet', 'pe', 'ane'], dtype='object')
```

[ ]:
```
# Comprobar el tipo de datos de los Data Frames y "qué pinta tienen"
print("\n Muestra del X_train:\n", X_train.head(), "\n Muestra del Y_train:\n",
  ↪Y_train.head() )
print("\nInfo X_train: \n")
X_train.info()

print("\nInfo Y_train:\n")
Y_train.info()
```

```
 Muestra del Y_train:
    Unnamed: 0   age    bp     sg   al  …   dm  cad appet  pe ane
0          195  70.0  90.0   1.02  2.0  …  yes  yes  poor  no  no
1          358  47.0  60.0   1.02  0.0  …   no   no  good  no  no
2          173  17.0  70.0  1.015  1.0  …   no   no  good  no  no
3          306  52.0  80.0   1.02  0.0  …   no   no  good  no  no
4          377  64.0  70.0   1.02  0.0  …   no   no  good  no  no

[5 rows x 25 columns]
 Muestra del Y_train:
    Unnamed: 0   class
0          195     ckd
1          358  notckd
2          173     ckd
3          306  notckd
4          377  notckd

Info X_train:

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 320 entries, 0 to 319
Data columns (total 25 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   Unnamed: 0  320 non-null    int64
 1   age         320 non-null    object
 2   bp          320 non-null    object
 3   sg          320 non-null    object
 4   al          320 non-null    object
 5   su          320 non-null    object
 6   rbc         320 non-null    object
 7   pc          320 non-null    object
 8   pcc         320 non-null    object
 9   ba          320 non-null    object
 10  bgr         320 non-null    object
 11  bu          320 non-null    object
```

```
 12  sc           320 non-null    object
 13  sod          320 non-null    object
 14  pot          320 non-null    object
 15  hemo         320 non-null    object
 16  pcv          320 non-null    object
 17  wbcc         320 non-null    object
 18  rbcc         320 non-null    object
 19  htn          320 non-null    object
 20  dm           320 non-null    object
 21  cad          320 non-null    object
 22  appet        320 non-null    object
 23  pe           320 non-null    object
 24  ane          320 non-null    object
dtypes: int64(1), object(24)
memory usage: 62.6+ KB

Info Y_train:

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 320 entries, 0 to 319
Data columns (total 2 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   Unnamed: 0  320 non-null    int64
 1   class       320 non-null    object
dtypes: int64(1), object(1)
memory usage: 5.1+ KB
```

[ ]:
```
# Reemplazar "?" con NaN en todas las columnas
X_train.replace('?', np.nan, inplace=True)
Y_train.replace('?', np.nan, inplace=True)

X_test.replace('?', np.nan, inplace=True)
```

[ ]:
```
# Columnas categóricas
columnas_categoricasX = ['rbc', 'pc', 'pcc', 'ba', 'htn', 'dm', 'cad', 'appet',␣
 ↪'pe', 'ane']

# Convertir columnas categóricas a tipo category en X_train y Y_train
X_train[columnas_categoricasX] = X_train[columnas_categoricasX].
 ↪astype('category')
X_test[columnas_categoricasX] = X_test[columnas_categoricasX].astype('category')

Y_train['class'] = Y_train['class'].astype('category')

# Columnas numéricas
```

```
columnas_numericas = ['age', 'bp', 'sg', 'al', 'su', 'bgr', 'bu', 'sc', 'sod',␣
 ↪'pot', 'hemo', 'pcv', 'wbcc', 'rbcc']

# Convertir columnas numéricas a tipo float en X_train
X_train[columnas_numericas] = X_train[columnas_numericas].astype(float)
X_test[columnas_numericas] = X_test[columnas_numericas].astype(float)
```

[ ]:
```
# Comprobar el tipo de datos de los data frames
X_train.info()
Y_train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 320 entries, 0 to 319
Data columns (total 25 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   Unnamed: 0  320 non-null    int64
 1   age         313 non-null    float64
 2   bp          311 non-null    float64
 3   sg          289 non-null    float64
 4   al          289 non-null    float64
 5   su          287 non-null    float64
 6   rbc         200 non-null    category
 7   pc          274 non-null    category
 8   pcc         317 non-null    category
 9   ba          317 non-null    category
 10  bgr         284 non-null    float64
 11  bu          304 non-null    float64
 12  sc          306 non-null    float64
 13  sod         250 non-null    float64
 14  pot         249 non-null    float64
 15  hemo        281 non-null    float64
 16  pcv         263 non-null    float64
 17  wbcc        237 non-null    float64
 18  rbcc        214 non-null    float64
 19  htn         318 non-null    category
 20  dm          318 non-null    category
 21  cad         318 non-null    category
 22  appet       319 non-null    category
 23  pe          319 non-null    category
 24  ane         319 non-null    category
dtypes: category(10), float64(14), int64(1)
memory usage: 42.0 KB
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 320 entries, 0 to 319
Data columns (total 2 columns):
 #   Column      Non-Null Count  Dtype
```

```
---  ------      --------------  -----
 0   Unnamed: 0  320 non-null    int64
 1   class       320 non-null    category
dtypes: category(1), int64(1)
memory usage: 3.1 KB
```

[ ]:
```
print(X_train)
```

```
     Unnamed: 0   age     bp   sg   al  …   dm  cad appet  pe  ane
0           195 70.00  90.00 1.02 2.00  …  yes  yes  poor  no   no
1           358 47.00  60.00 1.02 0.00  …   no   no  good  no   no
2           173 17.00  70.00 1.01 1.00  …   no   no  good  no   no
3           306 52.00  80.00 1.02 0.00  …   no   no  good  no   no
4           377 64.00  70.00 1.02 0.00  …   no   no  good  no   no
..          …   …    …  …  …  …  …  …   …  ..  …
315         158 71.00  60.00 1.02 3.00  …  yes  yes  good  no   no
316           8 52.00 100.00 1.01 3.00  …  yes   no  good  no  yes
317         392 57.00  80.00 1.02 0.00  …   no   no  good  no   no
318         317 58.00  70.00 1.02 0.00  …   no   no  good  no   no
319         300 45.00  60.00 1.02 0.00  …   no   no  good  no   no

[320 rows x 25 columns]
```

#####**Variables de X tipo float64:**

1. **age**: Representa la edad de los pacientes.
2. **bp**: Indica la presión arterial de los pacientes.
3. **sg**: Indica la densidad específica de la orina.
4. **al**: Representa el nivel de albuminuria en la orina.
5. **su**: Significa la cantidad de glucosa en suero.
6. **bgr**: Representa el nivel de glucosa en sangre en ayunas.
7. **bu**: Indica la concentración de urea en sangre.
8. **sc**: Representa la concentración de creatinina sérica en sangre.
9. **sod**: Indica la concentración de sodio en sangre.
10. **pot**: Representa la concentración de potasio en sangre.
11. **hemo**: Indica el nivel de hemoglobina en la sangre.
12. **pcv**: Representa el volumen de células empaquetadas en la sangre.
13. **wbcc**: Indica el recuento de glóbulos blancos en la sangre.
14. **rbcc**: Representa el recuento de glóbulos rojos en la sangre.

Descartamos hacer una representación gráfica de las variables para obtener información cualitativa y hacernos una idea de las distribuciones, ya que hay bastantes columnas. En su lugar mostraremos una tabla con estadísticos básicos para las variables numéricas:

[ ]:
```
desc = X_train[columnas_numericas].describe()
mode = X_train[columnas_numericas].mode().iloc[0]
mode_df = mode.to_frame().T.rename(index={0: 'mode'})
descripcion = pd.concat([desc, mode_df])
print(descripcion)
```

```
          age      bp      sg      al      su  …     pot    hemo     pcv      wbcc
rbcc
count  313.00  311.00  289.00  289.00  287.00  …  249.00  281.00  263.00    237.00
214.00
mean    51.45   76.59    1.02    1.04    0.47  …    4.75   12.54   38.84   8424.47
4.72
std     17.51   13.27    0.01    1.35    1.11  …    3.55    2.93    8.90   2956.03
1.03
min      2.00   50.00    1.00    0.00    0.00  …    2.50    3.10    9.00   2200.00
2.10
25%     41.00   70.00    1.01    0.00    0.00  …    3.90   10.40   33.00   6500.00
4.00
50%     55.00   80.00    1.02    0.00    0.00  …    4.50   12.60   40.00   8100.00
4.75
75%     65.00   80.00    1.02    2.00    0.00  …    4.90   15.00   45.00   9800.00
5.50
max     90.00  180.00    1.02    5.00    5.00  …   47.00   17.80   54.00  26400.00
8.00
mode    65.00   80.00    1.02    0.00    0.00  …    5.00   15.00   41.00   9800.00
4.50

[9 rows x 14 columns]
```

[ ]:
```python
# Establecer opciones de visualización
pd.set_option('display.float_format', lambda x: '%.2f' % x)  # Formato de los␣
 ↪números
pd.set_option('display.width', 1000)  # Ancho de visualización
pd.set_option('display.max_columns', 10)  # Máximo de columnas mostradas

num_columnas = descripcion.shape[1]
num_por_grupo = 5

for i in range(0, num_columnas, num_por_grupo):
    print(descripcion.iloc[:, i:i+num_por_grupo])
    print("\n")
```

```
          age      bp      sg      al      su
count  313.00  311.00  289.00  289.00  287.00
mean    51.45   76.59    1.02    1.04    0.47
std     17.51   13.27    0.01    1.35    1.11
min      2.00   50.00    1.00    0.00    0.00
25%     41.00   70.00    1.01    0.00    0.00
50%     55.00   80.00    1.02    0.00    0.00
75%     65.00   80.00    1.02    2.00    0.00
max     90.00  180.00    1.02    5.00    5.00
mode    65.00   80.00    1.02    0.00    0.00
```

	bgr	bu	sc	sod	pot
count	284.00	304.00	306.00	250.00	249.00
mean	146.32	57.77	3.20	137.78	4.75
std	76.91	51.12	6.24	11.17	3.55
min	22.00	1.50	0.40	4.50	2.50
25%	99.00	27.00	0.90	135.00	3.90
50%	122.00	44.00	1.30	139.00	4.50
75%	159.25	66.00	2.88	142.00	4.90
max	490.00	391.00	76.00	163.00	47.00
mode	99.00	25.00	1.20	135.00	5.00

	hemo	pcv	wbcc	rbcc
count	281.00	263.00	237.00	214.00
mean	12.54	38.84	8424.47	4.72
std	2.93	8.90	2956.03	1.03
min	3.10	9.00	2200.00	2.10
25%	10.40	33.00	6500.00	4.00
50%	12.60	40.00	8100.00	4.75
75%	15.00	45.00	9800.00	5.50
max	17.80	54.00	26400.00	8.00
mode	15.00	41.00	9800.00	4.50

#####**Variables de X tipo category:**

[ ]:
```
print("\nDominio de las categorías en X_train:")
for columna in columnas_categoricasX:
    print(f"{columna}: {X_train[columna].cat.categories.tolist()}")
```

```
Dominio de las categorías en X_train:
rbc: ['abnormal', 'normal']
pc: ['abnormal', 'normal']
pcc: ['notpresent', 'present']
ba: ['notpresent', 'present']
htn: ['no', 'yes']
dm: ['no', 'yes']
cad: ['no', 'yes']
appet: ['good', 'poor']
pe: ['no', 'yes']
ane: ['no', 'yes']
```

1. **rbc**: Este atributo indica el recuento de glóbulos rojos en la sangre. Cuenta con 2 categorías: `abnormal` y `normal`.
2. **pc**: Indica la apariencia de las células presentes en la orina. Cuenta con 2 categorías: `abnormal` y `normal`.
3. **pcc**: Representa la presencia de cilindros patológicos en la orina. Cuenta con 2 categorías:

`present` y `notpresent`.

4. **ba**: Indica la presencia de gránulos hialinos en la orina. Cuenta con 2 categorías: `present` y `notpresent`.
5. **htn**: Este atributo indica la presencia de hipertensión. Cuenta con 2 categorías: `yes` y `no`.
6. **dm**: Representa la presencia de diabetes mellitus. Cuenta con 2 categorías: `yes` y `no`.
7. **cad**: Indica la presencia de enfermedad arterial coronaria. Cuenta con 2 categorías: `yes` y `no`.
8. **appet**: Representa el apetito del paciente. Cuenta con 2 categorías: `good` y `poor`.
9. **pe**: Indica la presencia de edema periférico. Cuenta con 2 categorías: `yes` y `no`.
10. **ane**: Representa la presencia de anemia. Cuenta con 2 categorías: `yes` y `no`.

Una vez hayamos hecho el tratamiento de valores nulos, codificaremos las variables de tipo categórico con Label Encoding, ya que puede ser conflictivo si lo hacemos antes.

#####**Variables de Y tipo category:**

[ ]:
```
print("\nDominio de las categorías en Y_train:")
print(Y_train['class'].cat.categories.tolist())
```

```
Dominio de las categorías en Y_train:
['ckd', 'notckd']
```

1. **class**. Representa si el paciente tiene o no la afección. Hay dos categorías: `nockd` paciente sin afección y `ckd`: paciente con afección.

[ ]:
```
# Con minúscula tras separación para diferenciar
x_train, x_val, y_train, y_val = train_test_split(X_train, Y_train, test_size=0.
↪20, random_state=12)
print(y_val)
```

```
     Unnamed: 0   class
191         126     ckd
81          296  notckd
217         333  notckd
174          42     ckd
309         237     ckd
..          …     …
201         239     ckd
51           86     ckd
16          319  notckd
268         172     ckd
188         139     ckd

[64 rows x 2 columns]
```

[ ]:
```
x_train.shape
x_val.shape
```

[ ]: (64, 25)

1.1.2 Tratamiento de valores faltantes

2. Procesar los datos para tratar los valores perdidos. Para ello se deben considerar las siguientes cuestiones:

- ¿Eliminar ese ejemplo porque tiene valores perdidos?
- ¿Eliminar ese atributo **porque** tiene valores perdidos?
- ¿Imputar los valores perdidos?

A la salida de este paso consideramos que los atributos ya son características

Eliminar valores faltantes Para eliminar columnas y filas con valores perdidos, hemos establecido un umbral del 20% para eliminarlas:

Problema: Simplemente con este criterio nos quedaba train con 23 y test con 24 columnas... Si eliminabamos la intersección, no eliminabamos ninguna por lo que si después imputabamos los datos con los métodos conocidos hasta ahora, podríamos alterar la distribución de los datos. Por ello, hemos optado por eliminar tanto las columnas q tienen 20% de valores faltantes en train y en test, quedándonos 23 cols.

[ ]:
```
# Paso 1: Preprocesamiento en x_train

# Eliminar filas con valores faltantes en x_train
umbral_filas = 0.2  # Umbral del 20% para eliminar filas
x_train.dropna(axis=0, thresh=int(x_train.shape[1] * (1 - umbral_filas)),␣
 ↪inplace=True)

# Eliminar columnas con valores faltantes en x_train
umbral_columnas = 0.2  # Umbral del 20% para eliminar columnas
columnas_a_eliminar_train = x_train.columns[x_train.isnull().mean() >␣
 ↪umbral_columnas]
x_train.drop(columnas_a_eliminar_train, axis=1, inplace=True)


# Eliminar filas en y_train que no tienen correspondencia en x_train
y_train = y_train.loc[x_train.index]

# Eliminar las mismas columnas en x_val
x_val.drop(columnas_a_eliminar_train, axis=1, inplace=True)
```

[ ]:
```
print(x_train.shape, y_train.shape)
print(x_val.shape)
```

```
(219, 23) (219, 2)
(64, 23)
```

[ ]:
```
columnas_con_nulos = x_train.columns[x_train.isnull().any()].tolist()

valores_nulos_por_columna = x_train.isnull().sum()

print("Columnas con valores nulos:")
print(columnas_con_nulos)
print(valores_nulos_por_columna)
```

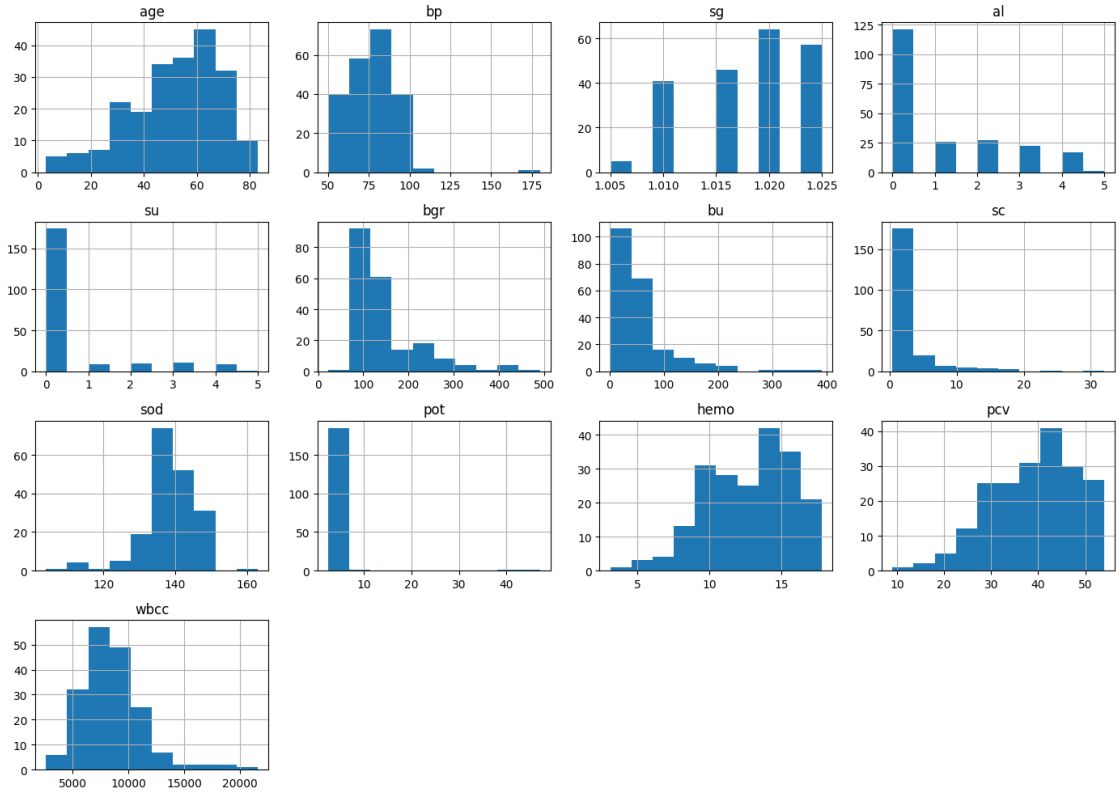
```
Columnas con valores nulos:
['age', 'bp', 'sg', 'al', 'su', 'pc', 'pcc', 'ba', 'bgr', 'bu', 'sc', 'sod',
'pot', 'hemo', 'pcv', 'wbcc', 'htn', 'dm', 'cad', 'appet', 'pe', 'ane']
Unnamed: 0     0
age            3
bp             5
sg             6
al             5
su             5
pc            14
pcc            3
ba             3
bgr           15
bu             5
sc             3
sod           31
pot           31
hemo          16
pcv           21
wbcc          36
htn            2
dm             2
cad            2
appet          1
pe             1
ane            1
dtype: int64
```

Imputar valores faltantes

[ ]:
```
# Visualizar histogramas de columnas numéricas
print("Representación columnas tipo float:")

tipos_de_datos = x_train.dtypes

columnas_numericas = tipos_de_datos[tipos_de_datos != 'category'].index.tolist()
columnas_numericas = columnas_numericas[1:] #Quitamos el índice

x_train[columnas_numericas].hist(figsize=(14, 10))
plt.tight_layout()
```

```
plt.show()
```

Representación columnas tipo float:



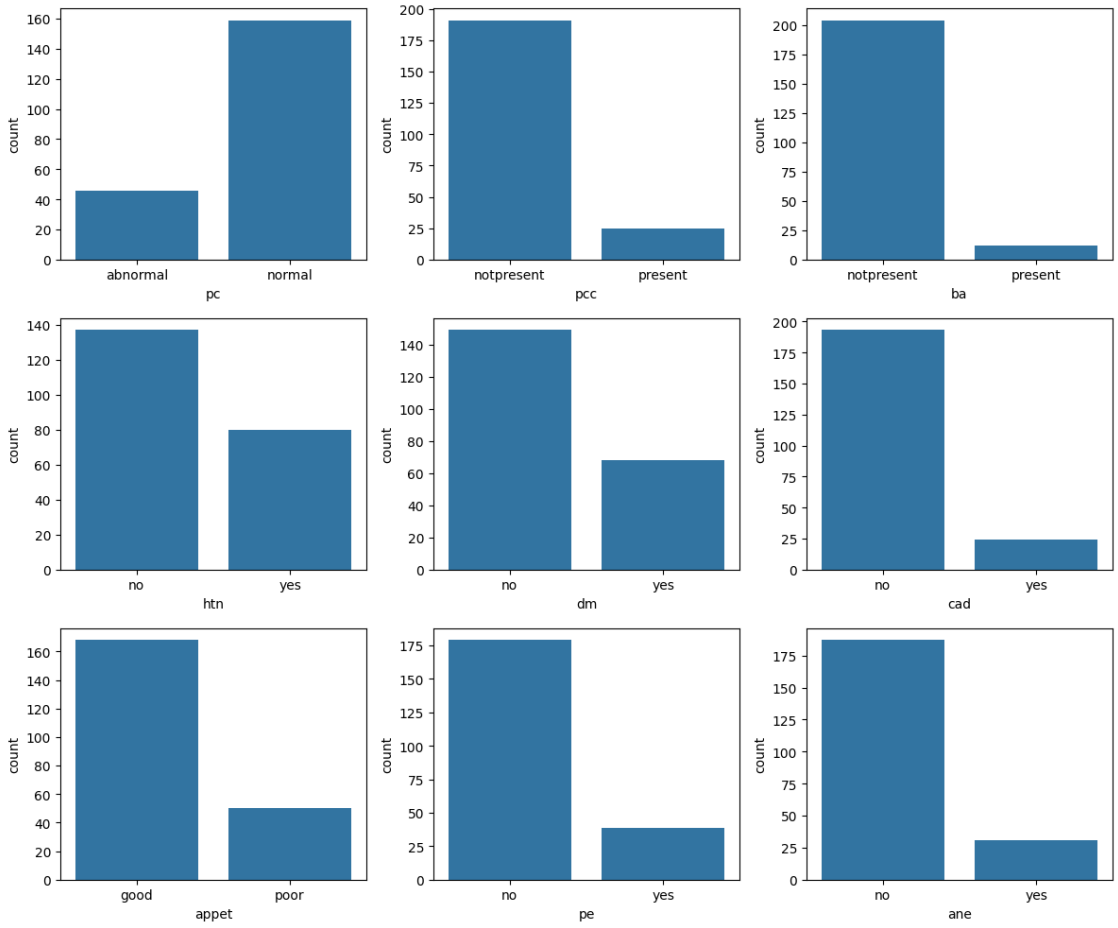
#####**Imputación univariada variables numéricas** Observamos que no son distribuciones normales, por lo que escogemos utilizar la mediana para hacer imputación univariada.

[ ]:
```
columnas_numericas = x_train.select_dtypes(include=['float64']).columns.tolist()
diccionario_medianas = {}

for column in columnas_numericas:
    median_train = x_train[column].median()
    x_train[column].fillna(median_train, inplace=True)
    x_val[column].fillna(median_train, inplace=True)
    diccionario_medianas[column] = median_train

print("Diccionario de medianas en X_train:", diccionario_medianas)
```

```
Diccionario de medianas en X_train: {'age': 55.0, 'bp': 80.0, 'sg': 1.02, 'al':
0.0, 'su': 0.0, 'bgr': 119.5, 'bu': 41.5, 'sc': 1.2, 'sod': 139.0, 'pot': 4.5,
'hemo': 13.2, 'pcv': 40.0, 'wbcc': 8100.0}
```

#####**Imputación univariada variables categóricas**

[ ]:
```
# Visualizar histogramas de columnas categóricas
columnas_categoricas = x_train.select_dtypes(include=['category']).columns.
  ↪tolist()

fig, axes = plt.subplots(3, 3, figsize=(12, 10))
for i, columna in enumerate(columnas_categoricas):
    sns.countplot(x=columna, data=x_train, ax=axes[i//3, i%3])
plt.tight_layout()
plt.show()
```

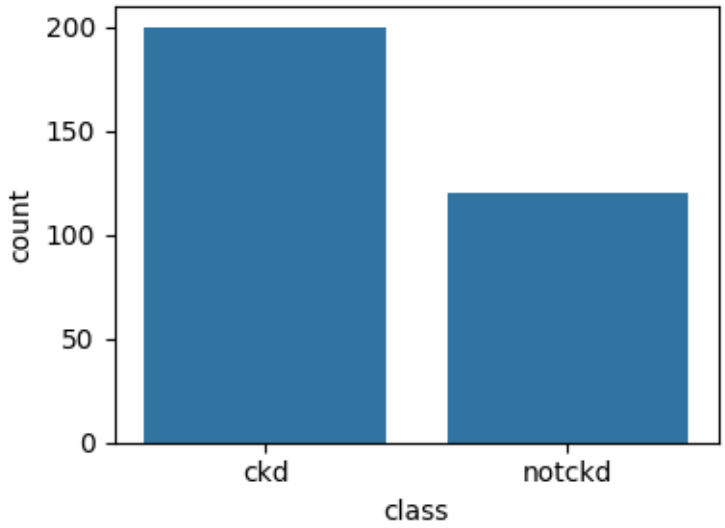


[ ]:
```
# Diccionario de modas de las variables categóricas para imputar
columnas_categoricas = x_train.select_dtypes(include=['category']).columns.
  ↪tolist()
diccionario_modas = {}

for column in columnas_categoricas:
    moda_train = x_train[column].mode()[0]
    x_train[column].fillna(moda_train, inplace=True)
```

```
    x_val[column].fillna(moda_train, inplace=True)

    diccionario_modas[column] = moda_train

print("Diccionario de modas en X_train:", diccionario_modas)
```

```
Diccionario de modas en X_train: {'pc': 'normal', 'pcc': 'notpresent', 'ba':
'notpresent', 'htn': 'no', 'dm': 'no', 'cad': 'no', 'appet': 'good', 'pe': 'no',
'ane': 'no'}
```

[ ]:
```
# Representar distribución variable objetivo
plt.figure(figsize=(4, 3))
sns.countplot(x='class', data=Y_train)
plt.tight_layout()
plt.show()
```



1.1.3 Codificación categorica

[ ]:
```
# Código para codificarlas (asumiendo que ya se han gestionado los valores␣
↪faltantes)

le = LabelEncoder()

# Variables categóricas de X
for columna in columnas_categoricas:
    x_train[columna] = le.fit_transform(x_train[columna])
    x_val[columna] = le.fit_transform(x_val[columna])
```

```
# Variable objetivo en Y
y_train['class'] = le.fit_transform(y_train['class'])
y_val['class'] = le.fit_transform(y_val['class'])
```

1.2 Ingeniería de características

3. Procesar las características para decidir si se elimina alguna de ellas.

Funciones modulares para ingeniería de características

[ ]:
```
def polyf_numerico (df_train, degree, interaction_only):
    # Crear características polinómicas de las numéricas

    cols_numericas = df_train.select_dtypes(include=float).columns.tolist()
    cols_categoricas = df_train.select_dtypes(include='category').columns.
 ↪tolist()

    polyf= PolynomialFeatures(degree=degree, interaction_only=interaction_only)
    polyf.set_output(transform="pandas")
    polyf.fit(df_train[cols_numericas])
    df_polyf_num = polyf.transform(df_train[cols_numericas])
    df_polyf_num = pd.concat([df_polyf_num, df_train[cols_categoricas]], axis=1)

    return df_polyf_num, polyf


def polyf (df_train, degree, interaction_only):
    # Crear características polinómicas de todos

    polyf= PolynomialFeatures(degree=degree, interaction_only=interaction_only)
    polyf.set_output(transform="pandas")
    polyf.fit(df_train)
    df_polyf= polyf.transform(df_train)

    return df_polyf, polyf


def filtrar_var_train(df_train, threshold):

    selector = VarianceThreshold(threshold=threshold)
    selector.set_output(transform="pandas")
    selector.fit(df_train)
    X_train_fil_var = selector.transform(df_train)

    return X_train_fil_var, selector
```

```
def filtrar_corr_train (df_train, umbral_correlacion):
    correlation_matrix = df_train.corr().abs()

    # Identificar las características altamente correlacionadas
    upper_tri = correlation_matrix.where(np.triu(np.ones(correlation_matrix.
 ↪shape), k=1).astype(bool))
    high_correlation_features = [column for column in upper_tri.columns if␣
 ↪any(upper_tri[column] > umbral_correlacion)]

    # Eliminar características altamente correlacionadas
    df_train = df_train.drop(columns=high_correlation_features)

    return df_train, high_correlation_features


def std_pca_train (df_train, n_components):

    # Estandarizar
    scalerStd = StandardScaler().set_output(transform="pandas")
    scalerStd.fit(df_train)
    df_train_scl = scalerStd.transform(df_train)

    # Aplicar PCA
    pca = PCA(n_components=n_components).set_output(transform="pandas")
    pca.fit(df_train_scl)
    df_train_pca = pca.transform(df_train_scl)

    return df_train_pca, scalerStd, pca

# Procesado 1: filtrado por varianza, estandarizacion y PCA (los datos se␣
 ↪modifican al invocar)
def procesado_train_1 (df_train, threshold, n_components):
    df_train, selector= filtrar_var_train(df_train, threshold)
    df_train, scaler, pca =std_pca_train(df_train, n_components)
    return df_train, selector, scaler, pca

def procesado_test_1(df_test, selector, scaler, pca):
    df_test_fil_var =selector.transform(df_test)
    df_test_scl= scaler.transform(df_test_fil_var)
    df_test_pca= pca.transform(df_test_scl)

    return df_test_pca


# Procesado 2: polynomialfeatures, estandarizacion, PCA
```

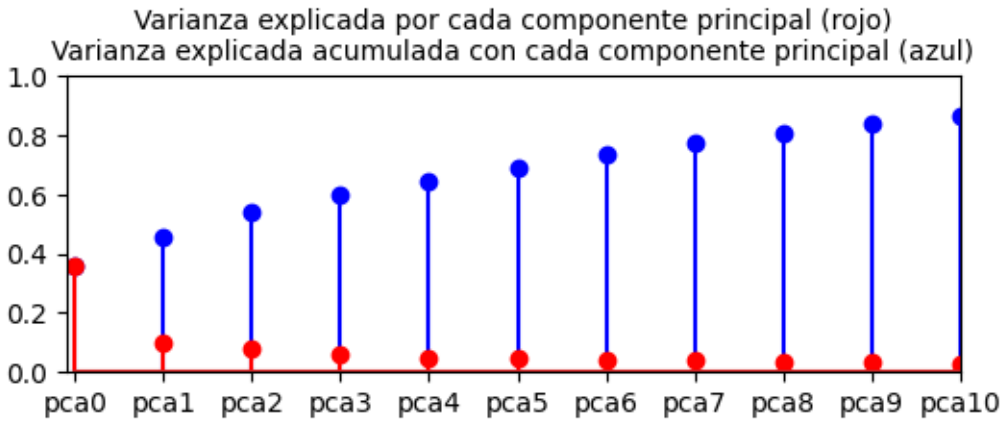
```
def procesado_train_2 (df_train, pf_degree, pf_interaction_only,␣
 ↪pf_numeric_only, pca_n_components):
    if pf_numeric_only:
      df_train, polyf = polyf_numerico(df_train, pf_degree, pf_interaction_only)
    else:
      df_train, polyf = polyf(df_train, pf_degree, pf_interaction_only)

    df_train, scaler, pca = std_pca_train(df_train, pca_n_components)

    return df_train, polyf, scaler, pca

def procesado_test_2(df_test, pf_degree, pf_interaction_only, pf_numeric_only,␣
 ↪polyf, scaler, pca):
    if pf_numeric_only:
        df_test, polyf = polyf_numerico(df_test, pf_degree, pf_interaction_only)
    else:
        df_test, polyf = polyf(df_test, pf_degree, pf_interaction_only)

    df_test = polyf.transform(df_test)
    df_test = scaler.transform(df_test)
    df_test = pca.transform(df_test)
    return df_test
```

[ ]:
```
# #  Configuraciones distintas para procesado 1 (filtrado por varianza,␣
 ↪estandarización, PCA):

# Configuración modelo 1_1
x_train_1_1, selector_1_1, scaler_1_1, pca1_1= procesado_train_1(x_train,␣
 ↪threshold=0.1, n_components=0.85)
x_val_1_1= procesado_test_1(x_val, selector_1_1, scaler_1_1, pca1_1)

# Configuración modelo 1_2
x_train_1_2, selector_1_2, scaler_1_2, pca1_2= procesado_train_1(x_train,␣
 ↪threshold=0.25, n_components=5)
x_val_1_2= procesado_test_1(x_val, selector_1_2, scaler_1_2, pca1_2)

# Configuración modelo 1_3
x_train_1_3, selector_1_3, scaler_1_3, pca1_3= procesado_train_1(x_train,␣
 ↪threshold=0.1, n_components=2)
x_val_1_3= procesado_test_1(x_val, selector_1_3, scaler_1_3, pca1_3)

# # Configuraciones distintas para procesado 2 (polynomialfeatures,␣
 ↪estandarización, PCA):
x_train_2_1, polyf_2_1, scaler_2_1, pca2_1 = procesado_train_2(x_train,␣
 ↪pf_degree=2,
```

```
 ↪pf_interaction_only=True,
 ↪pf_numeric_only=True,
 ↪pca_n_components=0.85)
```

[ ]:
```
# Función para generar el gráfico de explicación de varianza por número de componentes

def plot_PCA(pca):
  plt.stem(pca.explained_variance_ratio_.cumsum(),'b')
  plt.stem(pca.explained_variance_ratio_,'r')

  titleStr = 'Varianza explicada por cada componente principal (rojo)'
  titleStr = titleStr+'\n'
  titleStr = titleStr+'Varianza explicada acumulada con cada componente principal (azul)'
  plt.title(titleStr, fontsize=10)
  ax = plt.gca()
  ax.axis([-0.1,1.1,0,1])
  ax.set_xticks([i for i in range(pca.n_components_)])
  ax.set_xticklabels(["pca"+str(i) for i in range(pca.n_components_)])

  fig = plt.gcf()
  fig.set_size_inches(6,2)

  plt.show()
```

[ ]:
```
plot_PCA(pca1_1)
pca1_1.n_components_
sum(pca1_2.explained_variance_ratio_)
sum(pca1_3.explained_variance_ratio_)
```



[ ]: 0.4576948478897684

##Modelo lineal : Regresión Logística

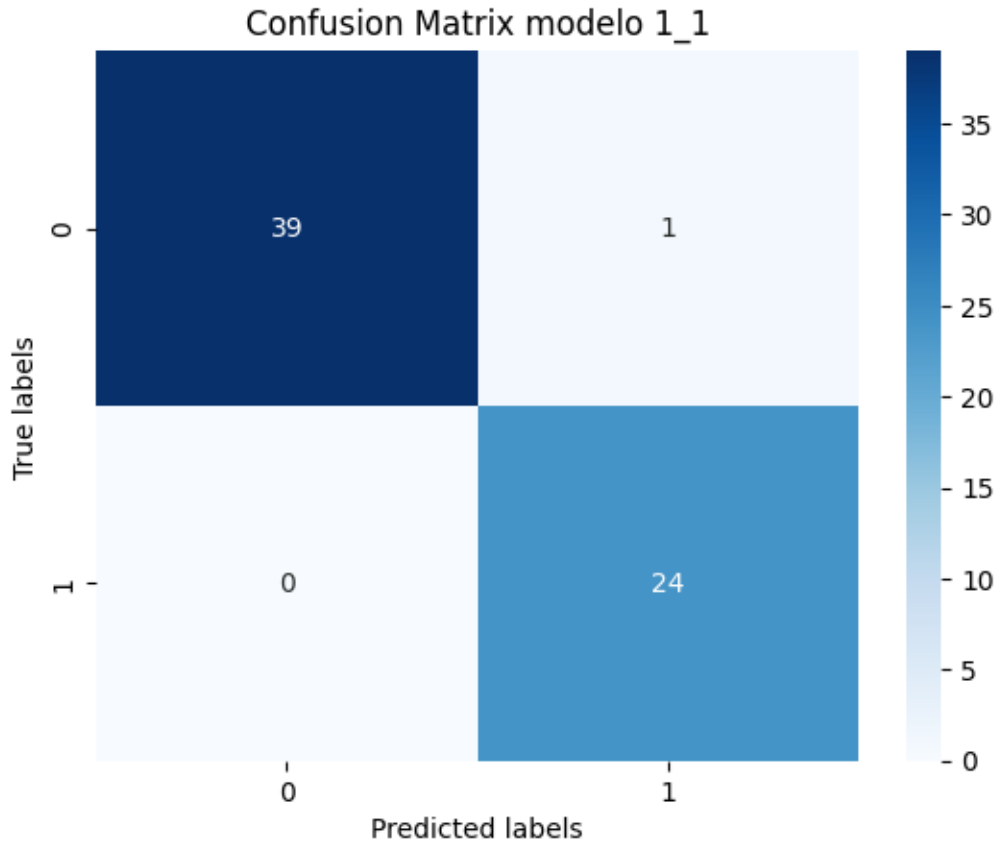
[ ]:
```
# # Modelo 1:

logreg = LogisticRegression(penalty="l1", solver='liblinear') # No se aprecian␣
↪diferencias con l2
target = y_train['class']

# Ajustar el modelo con los datos de entrenamiento
logreg.fit(x_train_1_1, target)

# Realizar predicciones en el conjunto de validación
y_hat = logreg.predict(x_val_1_1)

# Calcular las probabilidades de las predicciones en el conjunto x_train_1_1
prob = logreg.predict_proba(x_train_1_1)

# print(y_hat, prob)
```

[ ]:
```
# # Modelo 1_2:

logreg2 = LogisticRegression(penalty="l1", solver='liblinear')
target = y_train['class']

# Ajustar el modelo con los datos de entrenamiento
logreg2.fit(x_train_1_2, target)

# Realizar predicciones en el conjunto de validación
y_hat2 = logreg2.predict(x_val_1_2)

# Calcular las probabilidades de las predicciones en el conjunto x_train_1_2
prob2 = logreg2.predict_proba(x_train_1_2)

# print(y_hat, prob)
```

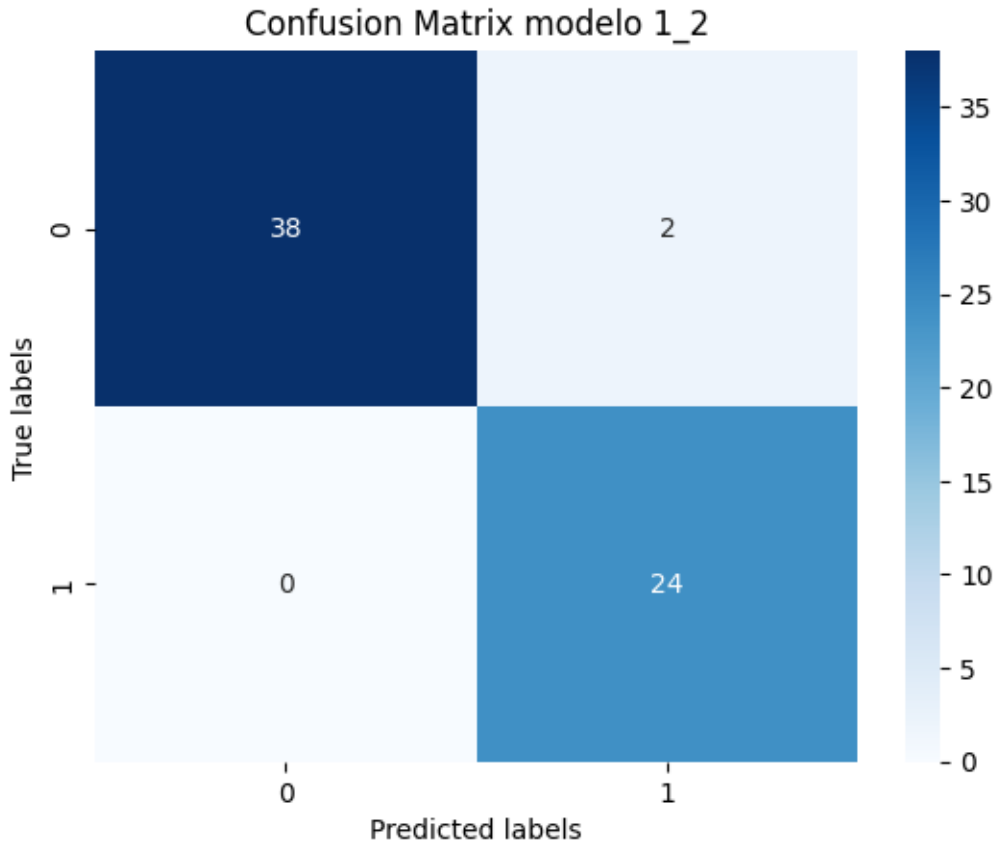
[ ]:
```
# # Modelo 1_3:

target = y_train['class']

logreg3 = LogisticRegression(penalty="l1", solver='liblinear')

# Ajustar el modelo con los datos de entrenamiento
logreg3.fit(x_train_1_3, target)
```

```
# Realizar predicciones en el conjunto de validación
y_hat3 = logreg3.predict(x_val_1_3)

# Calcular las probabilidades de las predicciones en el conjunto x_train_1_3
prob3 = logreg3.predict_proba(x_train_1_3)

# print(y_hat, prob)
```

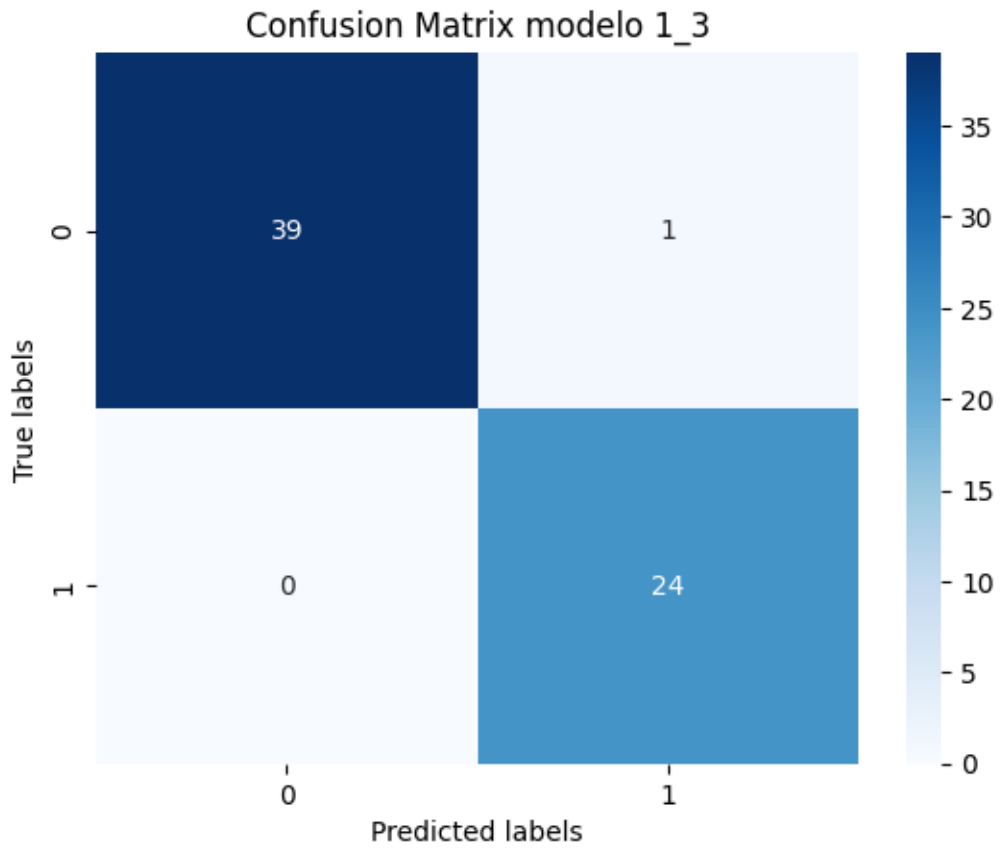
[ ]:
```
# Calcular la matriz de confusión
y_val_array = y_val['class'].values
cm1_1 = confusion_matrix(y_val_array, y_hat)

# Visualizar la matriz de confusión
sns.heatmap(cm1_1, annot=True, cmap='Blues', fmt='g')
plt.xlabel('Predicted labels')
plt.ylabel('True labels')
plt.title('Confusion Matrix modelo 1_1')
plt.show()
```



[ ]:
```
y_val_array = y_val['class'].values
cm1_2 = confusion_matrix(y_val_array, y_hat2)

sns.heatmap(cm1_2, annot=True, cmap='Blues', fmt='g')
plt.xlabel('Predicted labels')
plt.ylabel('True labels')
plt.title('Confusion Matrix modelo 1_2')
plt.show()
```

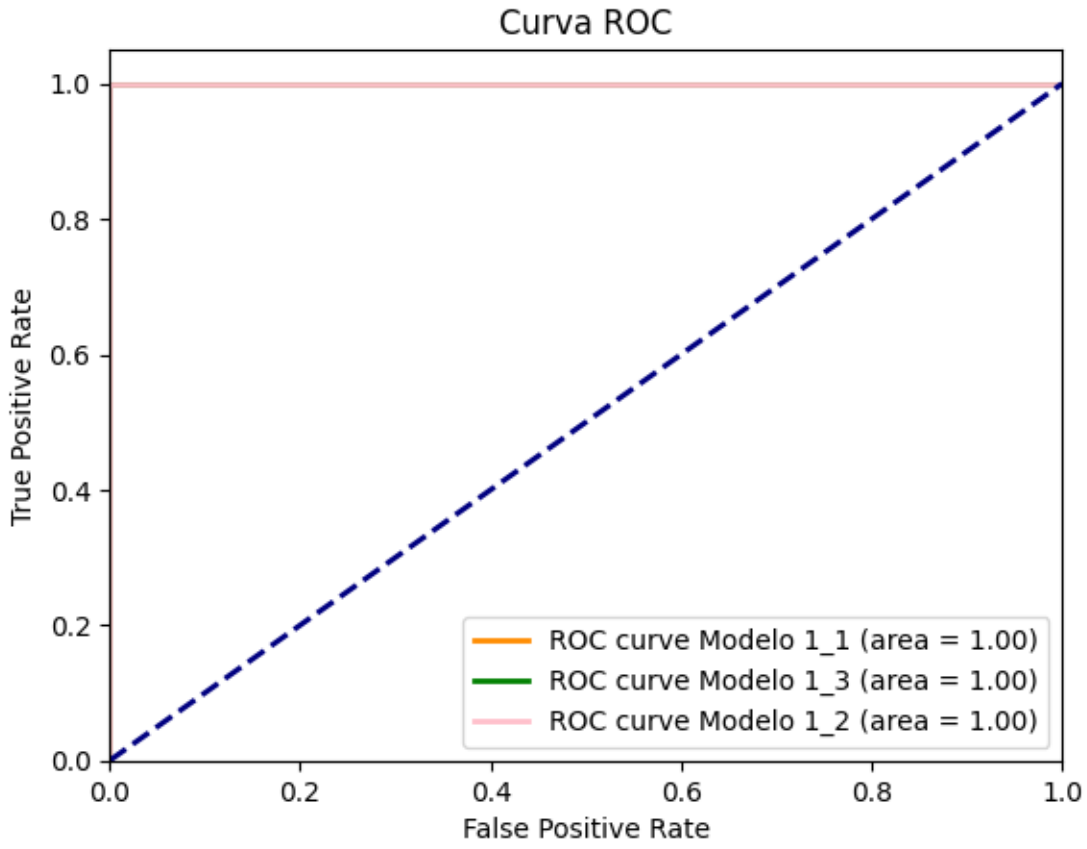


[ ]:
```
cm1_3 = confusion_matrix(y_val_array, y_hat3)

sns.heatmap(cm1_3, annot=True, cmap='Blues', fmt='g')
plt.xlabel('Predicted labels')
plt.ylabel('True labels')
plt.title('Confusion Matrix modelo 1_3')
plt.show()
```



[ ]:
```
# Calcular las probabilidades de las predicciones en el conjunto de validación
prob_val1_1 = logreg.predict_proba(x_val_1_1)
prob_val1_2 = logreg2.predict_proba(x_val_1_2)
prob_val1_3 = logreg3.predict_proba(x_val_1_3)


# Calcular la curva ROC y el área bajo la curva (AUC)
fpr1_1, tpr1_1, thresholds1_1 = roc_curve(y_val_array, prob_val1_1[:, 1])
roc_auc1_1 = auc(fpr1_1, tpr1_1)

fpr1_2, tpr1_2, thresholds1_2 = roc_curve(y_val_array, prob_val1_2[:, 1])
roc_auc1_2= auc(fpr1_2, tpr1_2)

fpr1_3, tpr1_3, thresholds1_3= roc_curve(y_val_array, prob_val1_3[:, 1])
roc_auc1_3 = auc(fpr1_3, tpr1_3)


# Visualizar la curva ROC para ambos modelos en la misma gráfica
plt.figure()
```

```
plt.plot(fpr1_1, tpr1_1, color='darkorange', lw=2, label='ROC curve Modelo 1_1␣
 ↪(area = %0.2f)' % roc_auc1_1)
plt.plot(fpr1_3, tpr1_3, color='green', lw=2, label='ROC curve Modelo 1_3 (area␣
 ↪= %0.2f)' % roc_auc1_3)
plt.plot(fpr1_2, tpr1_2, color='pink', lw=2, label='ROC curve Modelo 1_2 (area␣
 ↪= %0.2f)' % roc_auc1_2)

plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Curva ROC')
plt.legend(loc="lower right")
plt.show()
```



[ ]:
```
# Calcular los valores necesarios para el cálculo manual del F1-Score
listay_hat=[y_hat, y_hat3]
```

```
def f1_score (y_hat, y_val_array):
    TP = np.sum((y_hat == 1) & (y_val_array == 1))
    FP = np.sum((y_hat == 1) & (y_val_array == 0))
    FN = np.sum((y_hat == 0) & (y_val_array == 1))

    # Calcular la precisión y el recall
    precision = TP / (TP + FP)
    recall = TP / (TP + FN)

    # Calcular el F1-Score
    f1_score = 2 * (precision * recall) / (precision + recall)

    return f1_score


f1_score1= f1_score(y_hat,y_val_array)
f1_score3= f1_score(y_hat3,y_val_array)
f1_score2=f1_score(y_hat2,y_val_array)
```

[ ]:
```
print(f1_score1, f1_score2, f1_score3)
```

```
0.9795918367346939 0.9600000000000001 0.9795918367346939
```

[ ]:
```
# Guardar los objetos en un archivo usando pickle
with open('modelo_1_1_objetos.pkl', 'wb') as f:
    pickle.dump((selector_1_1, scaler_1_1, pca1_1,logreg), f)
```

02-fichero-test

May 15, 2024

[ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
```

[ ]:
```
import pandas as pd
import numpy as np
import pickle

from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import PolynomialFeatures, StandardScaler
from sklearn.decomposition import PCA
from sklearn.feature_selection import VarianceThreshold
from sklearn.feature_selection import VarianceThreshold
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression
```

[ ]:
```
# Modificar ruta según ubicación:
X_test = pd.read_csv('/content/retoML1_X_test.csv', delimiter=';')

# Ruta al archivo en tu Google Drive
ruta_al_archivo = '/content/modelo_1_1_objetos.pkl'

# Abrir el archivo y cargar los objetos
with open(ruta_al_archivo, 'rb') as archivo:
    objetos_cargados = pickle.load(archivo)
```

##Funciones preprocesado y procesado

[ ]:
```
def preprocesado_atributos (df, columnas_categoricas, columnas_numericas):
  df.replace('?', np.nan, inplace=True)
  # Convertir tipos
  df[columnas_categoricas]=df[columnas_categoricas].astype('category')
  df[columnas_numericas]=df[columnas_numericas].astype(float)
  return df
```

```
def tratar_valores_faltantes(df, columnas_eliminadas_train,␣
 ↪dict_median_numeric_vars, dict_modes_category_vars):
  # Eliminar columnas que hemos eliminado en test
  df.drop(columnas_eliminadas_train, axis=1, inplace=True)

  # Imputar valores faltantes
  # Para variables numéricas
  for var, median_value in dict_median_numeric_vars.items():
      df[var].fillna(median_value, inplace=True)
      df[var] = df[var].astype(float)

  # Para variables categóricas
  for var, mode_value in dict_modes_category_vars.items():
      df[var].fillna(mode_value, inplace=True)
      df[var] = df[var].astype('category')

  return df


def encoding(df):
  columnas_categoricas = df.select_dtypes(include=['category']).columns.tolist()

  le = LabelEncoder()
  for columna in columnas_categoricas:
      df[columna] = le.fit_transform(df[columna])

  return df
```

[ ]:
```
def filtrar_var_train(df_train, threshold):

    selector = VarianceThreshold(threshold=threshold)
    selector.set_output(transform="pandas")
    selector.fit(df_train)
    X_train_fil_var = selector.transform(df_train)

    return X_train_fil_var, selector


def filtrar_corr_train (df_train, umbral_correlacion):
    correlation_matrix = df_train.corr().abs()

    # Identificar las características altamente correlacionadas
    upper_tri = correlation_matrix.where(np.triu(np.ones(correlation_matrix.
 ↪shape), k=1).astype(bool))
    high_correlation_features = [column for column in upper_tri.columns if␣
 ↪any(upper_tri[column] > umbral_correlacion)]
```

```
    # Eliminar características altamente correlacionadas
    df_train = df_train.drop(columns=high_correlation_features)

    return df_train, high_correlation_features


def std_pca_train (df_train, n_components):
    # Estandarizar
    scalerStd = StandardScaler().set_output(transform="pandas")
    scalerStd.fit(df_train)
    df_train_scl = scalerStd.transform(df_train)

    # Aplicar PCA
    pca = PCA(n_components=n_components).set_output(transform="pandas")
    pca.fit(df_train_scl)
    df_train_pca = pca.transform(df_train_scl)

    return df_train_pca, scalerStd, pca

# Procesado 1: filtrado por varianza, estandarización y PCA (los datos se␣
 ↪modifican al invocar)
def procesado_train_1 (df_train, threshold, n_components):
    df_train, selector= filtrar_var_train(df_train, threshold)
    df_train, scaler, pca =std_pca_train(df_train, n_components)
    return df_train, selector, scaler, pca

def procesado_test_1(df_test, selector, scaler, pca):
    df_test_fil_var =selector.transform(df_test)
    df_test_scl= scaler.transform(df_test_fil_var)
    df_test_pca= pca.transform(df_test_scl)

    return df_test_pca
```

0.1 Datos extraídos

[ ]:
```
# Datos necesarios para procesar
columnas_categoricas = ['rbc', 'pc', 'pcc', 'ba', 'htn', 'dm', 'cad', 'appet',␣
 ↪'pe', 'ane']
columnas_numericas = ['age', 'bp', 'sg', 'al', 'su', 'bgr', 'bu', 'sc', 'sod',␣
 ↪'pot', 'hemo', 'pcv', 'wbcc', 'rbcc']

columnas_eliminadas_train = pd.Index(['rbc', 'rbcc'])
# Con todos los datos filtramos rbcc columnas_eliminadas_train = pd.
 ↪Index(['rbc', 'rbcc'])
# Con la igualación de clases no se eliminó :
```

```
columnas_eliminadas_train = pd.Index(['rbc', 'rbcc'])
dict_median_numeric_vars = {'age': 55.0, 'bp': 80.0, 'sg': 1.02, 'al': 0.0,␣
 ↪'su': 0.0, 'bgr': 119.5, 'bu': 41.5, 'sc': 1.2, 'sod': 139.0, 'pot': 4.5,␣
 ↪'hemo': 13.2, 'pcv': 40.0, 'wbcc': 8100.0}

dict_modes_category_vars = {'pc': 'normal', 'pcc': 'notpresent', 'ba':␣
 ↪'notpresent', 'htn': 'no', 'dm': 'no', 'cad': 'no', 'appet': 'good', 'pe':␣
 ↪'no', 'ane': 'no'}
```

##Preprocesado

[ ]:
```
X_test= preprocesado_atributos(X_test, columnas_categoricas, columnas_numericas)
X_test=␣
 ↪tratar_valores_faltantes(X_test,columnas_eliminadas_train,dict_median_numeric_vars=dict_med
 ↪dict_modes_category_vars=dict_modes_category_vars)
X_test= encoding(X_test)
```

0.2 Ingenería características

[ ]:
```
# Crear y asignar nombres a cada objeto
selector = objetos_cargados[0]
scaler = objetos_cargados[1]
pca = objetos_cargados[2]
log_reg = objetos_cargados[3]

# Usar los objetos como sea necesario
```

[ ]:
```
# Procesar
X_test_procesado= procesado_test_1(X_test, selector, scaler, pca)
```

##Predicción

[ ]:
```
# Predecir
predicciones = log_reg.predict(X_test_procesado)
print(predicciones)
```

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

#Postprocesado y exportación

[ ]:
```
mapeo = {0: 'ckd', 1: 'notckd'}
predicciones= np.array([mapeo[pred] for pred in predicciones])
```

[ ]:
```
# Crear un data frame con la lista de predicciones
df = pd.DataFrame({'class': predicciones})
```

```
# Nombre del archivo CSV de salida
nombre_archivo = 'retoML1_Y_test.csv'

# Exportar el data fFrame a un archivo CSV
df.to_csv(nombre_archivo, index=False)
```