

Regresion Logistica en Python

En este notebook aprenderas la codificacion basica para **regresion logistica** en python.

Primero, instalaremos la libreria celluloid en este notebook. Recuerda que en Google Colab ya esta instalado la mayoria de librerias que usaremos. Para hacer esto, ejecutamos la siguiente celda.

Ejecute esta celda solo una vez. Verá una larga lista de instrucciones que aparecen en la pantalla. Deje que se ejecute hasta que se instale todos los requisitos.

```
In [ ]: !pip install celluloid #- SOLO PARA LOS QUE TRABAJAN EN COLAB !!!
```

1. Adquisicion de datos

Cargando el conjunto de datos de bank.xlsx

```
In [ ]: # Conexion a Google Colaborative - SOLO PARA LOS QUE TRABAJAN EN COLAB !!!  
from google.colab import drive  
drive.mount('/content/drive')
```

```
In [ ]: import numpy as np  
import pandas as pd  
pd.options.mode.chained_assignment = None  
df = pd.read_excel('bank.xlsx')
```

```
In [ ]:
```

Inspeccionemos los datos usando head

```
In [ ]: df.head()
```

Este conjunto de datos contiene informacion acerca de los cursos de excel. Este conjunto de datos incluye "Cantidad", "Volumen_Soles", "LoginsMes" y "Segmento" para un total de 5591 registros.

Utilice el método `value_counts` para evaluar la distribución de los diferentes valores para la columna `Segmento`

```
In [ ]: df['Segmento'].value_counts()
```

Hay diferentes tipos de segmentos: *Estandar*, *Preferente*, *Primum* y *Otro*

Distribucion de los datos

Estudiemos cómo se distribuyen estas categorías trazando los datos.

```
In [ ]: %matplotlib inline  
import seaborn as sns
```

```
sns.set()
sns.pairplot(df, hue='Segmento', height=2)
```

Los gráficos de arriba muestran cómo los atributos numéricos `Cantidad`, `Volumen_Soles` y `LoginMes` se relacionan entre sí para las diferentes categorías de `Segmento`. Cada gráfico muestra una variable diferente en el eje vertical y horizontal. Los puntos de datos se colorean de acuerdo con su valor en la columna `Segmento`. Las gráficas de la diagonal principal corresponden a la distribución de cada variable. Observe estos gráficos para intentar extraer información

En este notebook usaremos los datos anteriores para **entrenar un modelo para predecir el tipo de segmento**. Por lo tanto, usaremos `Segmento` como variable dependiente y las columnas restantes como variables independientes.

2. Preparacion de datos

Comencemos generando la matriz de características y la matriz del target. Para hacer esto, cree dos nuevas variables, X e y. El primero debe contener todos los datos de las variables independientes. El segundo solo debe almacenar los datos correspondientes a la variable dependiente.

```
In [ ]: x = df.drop(columns=['Segmento'])
y = df[['Segmento']]
```

Si ambas variables están definidas correctamente, debería mostrar la salida `(5591, 3)` `(5591,)` cuando ejecute la celda de abajo. Si no vuelve a la celda anterior y modifica su contenido.

```
In [ ]: print(x.shape, y.shape)
```

Valores faltantes

El conjunto de datos puede contener faltante. Compruebe si faltan valores en "X". Hay diferentes formas de hacerlo. Intente usar las funciones `isnull` y `notnull`, o incluso, `count` para obtener el número de valores faltantes en cada columna.

```
In [ ]: x.isnull().sum()
```

Como se puede observar, las variables no tienen valores faltantes.

Variables categoricas

El conjunto de datos \$y\$ tiene una variable categórica: `Segmento`.

Para poder entrenar nuestro modelo, recuerde que **necesitamos transformar todos los atributos en numéricos**. Usamos un metodo para esta variable:

- `Segmento`. Este atributo tiene cuatro valores diferentes: `Estandar`, `Preferente`, `Primum`, y `Otros`. Entonces para este atributo, cambiaremos las

categorias `Estandar`, `Preferente`, y `Otros` en un valor numérico `0`, y para la categoría `Primum` en un valor numérico `1`.

```
In [ ]: y[ 'Segmento' ] = y[ 'Segmento' ].replace(['Otro','Estandar','Preferente'],0)
y[ 'Segmento' ] = y[ 'Segmento' ].replace(['Primum'],1)
y.head()
```

Normalización de los datos

Una vez que todos los datos estén en forma numérica, tendremos que normalizar para asegurarnos de que todos los atributos tengan el mismo rango.

```
In [ ]: from sklearn.preprocessing import StandardScaler
sc_X = StandardScaler()
X_normalized = sc_X.fit_transform(X)
```

3. Creación inicial del modelo

Regresión logística

El modelo que usaremos es una **regresión logística**. Entrene este modelo utilizando los datos normalizados y sus correspondientes entradas de target.

Importe el modelo `LogisticRegression` de `sklearn.linear_model`, cree una instancia y entrénelo con los datos.

```
In [ ]: from sklearn.linear_model import LogisticRegression
lr = LogisticRegression(solver="lbfgs", random_state=0, C = 1e20)
lr.fit(X_normalized, y.values.ravel())
```

Accuracy

Evaluamos el rendimiento (el ajuste) del modelo utilizando el accuracy. Esta métrica calcula el porcentaje de entradas predichas correctamente

$$\text{Accuracy}(y, \hat{y}) = \frac{1}{n_{\text{samples}}} \sum^{n_{\text{samples}}-1}_{i=0} \mathbb{1}(\hat{y}_i = y_i)$$

Utilice la función `precision_score` de `sklearn` para calcular la accuracy del modelo

```
In [ ]: from sklearn.metrics import accuracy_score

y_predict = lr.predict(X_normalized)
accuracy_score(y, y_predict)
```

4. División de entrenamiento/prueba

El primer paso es dividir el conjunto de datos en conjuntos de entrenamiento y de prueba. El primero se utilizará para entrenar el modelo y el segundo para la evaluación.

```
In [ ]: from sklearn.model_selection import train_test_split
```

```
x_train, x_test, y_train, y_test = train_test_split(X_normalized, y.values.r
```

Fit/Predict

Utilice la función fit para entrenar al modelo

```
In [ ]: lr = LogisticRegression(solver="lbfgs", random_state=0, C = 1e20)
lr.fit(x_train, y_train)
```

Utilice la función predict para obtener predicciones para los datos de prueba

```
In [ ]: y_predict = lr.predict(x_test)
```

Accuracy

Calcule el accuracy sobre los datos de prueba

```
In [ ]: accuracy_score(y_test, y_predict)
```

Matriz de confusión

```
In [ ]: from sklearn.metrics import plot_confusion_matrix
import matplotlib.pyplot as plt
plot_confusion_matrix(lr,
                      x_test,
                      y_test,
                      display_labels=['Primum', 'No primum'],
                      cmap=plt.cm.Blues)
plt.grid(False)
```

Métricas de desempeño asociadas a la Matriz de confusión

Además de accuracy, también hay otras métricas que podemos utilizar para evaluar nuestro modelo.

Recall

```
In [ ]: from sklearn.metrics import recall_score
recall_score(y_test, y_predict, average='weighted')
```

Precision

```
In [ ]: from sklearn.metrics import precision_score
precision_score(y_test, y_predict, average='weighted')
```

F1-measure

```
In [ ]: from sklearn.metrics import f1_score
f1_score(y_test, y_predict, average='weighted')
```

5. Validacion cruzada

Finalmente, recuerde que una parte crucial del aprendizaje automático es tener una **estimación confiable del rendimiento del modelo**. El uso de una división de entrenamiento/prueba es un primer paso para obtener una estimación no sesgada del rendimiento. Sin embargo, las resultados dependen en gran medida de la división específica, es decir, diferentes divisiones pueden dar resultados diferentes.

Para ver esto, ejecute la celda a continuación. Aquí, entrenamos 10 modelos diferentes, utilizando 10 divisiones diferentes. El parámetro `random_state` nos permite modificar los puntos que se asignan a los conjuntos de entrenamiento y prueba. Tenga en cuenta que muchos resultados son diferentes.

```
In [ ]: for random_state in range(1,100,10):
    X_train, X_test, y_train, y_test = train_test_split(X_normalized, y.values)
    #X_train_normalized = min_max_scaler.fit_transform(X_train)
    #X_test_normalized = min_max_scaler.transform(X_test)
    lr = LogisticRegression(solver="lbfgs", random_state=0, C = 1e20)
    #dt = DecisionTreeClassifier()
    lr.fit(X_train, y_train)
    print('accuracy_score: {}'.format(accuracy_score(y_test, lr.predict(X_te
```

Para superar este problema, utilizamos la validación cruzada.

Utilice el `cross_val_score` de sklearn para obtener una estimación del rendimiento del modelo utilizando **10 folds**.

```
In [ ]: from sklearn.model_selection import cross_val_score

scores = cross_val_score(lr, X_normalized, y.values.ravel(), cv=10)
scores.mean()
```

Ahora repita lo mismo que el anterior, pero esta vez calcule las puntuaciones de validación cruzada para un número de folds que van de 2 a 50. Esto significa que debe calcular un `cross_val_score` para cada valor de fold de 2 a 50. Intente idear una manera de hacerlo. Recuerde que para poder utilizar esta información, debe almacenar el resultado de cada iteración. Use una lista llamada `mean_scores` para este propósito.

```
In [ ]: from sklearn.model_selection import cross_val_score

min_folds = 2
max_folds = 50
mean_scores = []
for n in range(min_folds, max_folds):
    scores = cross_val_score(lr, X_normalized, y.values.ravel(), cv=n)
    mean_scores.append(scores.mean())
```

Evolución del accuracy

Veamos cómo cambia el accuracy estimada con el número de fold.

```
In [ ]: import matplotlib.pyplot as plt
```

```
plt.plot(range(min_folds, max_folds), mean_scores)
plt.ylim(0.55, 0.95)
plt.xlabel('folds')
plt.ylabel('Mean accuracy')
plt.show()
```

6. Graficos en 3 dimensiones

Para realizar los graficos, se trabajaran con dos variables dependientes: "Cantidad" y "Volumen_Soles", y con los primeros 50 registros del conjunto de datos.

```
In [ ]: #X_normalized.shape
df2 = df[0:50]
df2['Segmento'] = df2['Segmento'].replace(['Otro', 'Estandar', 'Preferente'], 0)
df2['Segmento'] = df2['Segmento'].replace(['Primum'], 1)
X_train2 = np.array(df2[['Cantidad', "Volumen_Soles"]])
y_train2 = np.array(df2[['Segmento']])
sc_X = StandardScaler()
x_train2 = sc_X.fit_transform(X_train2)
```

```
In [ ]: len(y_train2)
```

Regression logistica multiple

```
# Import libraries:
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm
from mpl_toolkits.mplot3d import Axes3D
import celluloid
from celluloid import Camera
from scipy.special import expit
from matplotlib import gridspec

# Define logistic regression model:
class LogisticRegression(object):
    def __init__(self, x, y, lr=0.1):
        self.lr=lr
        n=x.shape[1] # determine the number of independent variables
        self.w=np.ones((1,n))*(0) # initialize weight matrix and set weights
        self.b=0.5 # set starting value for b to 0.5

    def predict(self, x): # returns prediction
        z=x@self.w.T + self.b # @: matrix multiplication
        p=expit(z) # logistic sigmoid function
        return p

    def cost(self, x, y): #cost function
        z=x@self.w.T + self.b
        p=expit(z)
        return - np.mean(y*np.log(p) + (1-y)*np.log(1-p)) #Cross-entropy cost

    def step(self, x, y):
        z=x@self.w.T + self.b
        p=expit(z)

    # Partial derivatives:
    dw= np.mean((p - y)*x, axis=0)      # dJ/dw
```

```

        db = np.mean(p-y)                      # dJ/db
        self.w = self.w - dw*self.lr # update w
        self.b= self.b- db*self.lr # update b

    def fit(self,x,y,numberOfEpochs=100000):
        # Create arrays to store weights, biases, costs, predicted y-values
        #... and predicted y-values for connection lines in:
        self.AllWeights=np.zeros((numberOfEpochs, x.shape[1]))
        self.AllBiases=np.zeros((numberOfEpochs, x.shape[1]))
        self.AllCosts=np.zeros((numberOfEpochs, x.shape[1]))
        self.All_cl= np.zeros((numberOfEpochs,len(x))) # cl: # predicted y-v

        for step in range(numberOfEpochs):
            # for each step of gradient descent assign new parameter value t
            self.AllWeights[step]=self.w
            self.AllBiases[step]=self.b
            self.AllCosts[step]=self.cost(x,y)
            self.All_cl[step]=(self.predict(x)).T.flatten()
            self.step(x,y) # update parameter values

```

```

In [ ]: # Train multiple logistic regression model:
epochs_=100000 # number of epochs for training
model=LogisticRegression(x_train2,y_train2, lr=0.1)
model.fit(x_train2,y_train2, numberOfEpochs=epochs_)

# Return final model parameters and costs:
print("----- Multiple logistic regression model:")
print("Final weights: "+ str(model.w))
print("Final bias: "+ str(model.b))
print("Final costs: " + str(model.cost(x_train2,y_train2)))

# Define new logistic reg. model with fixed-intercept:
class LogisticRegression_fixed_b(object):
    def __init__(self,x,y,b,lr=0.01):
        self.lr=lr
        n=x.shape[1]
        self.w=np.array([[[-0.1,-0.1]]]) # set starting values for weights to
        self.b=np.array([[b]]) # fixed value for b

    def predict(self,x):
        p=expit( x @self.w.T + self.b)
        return p

    def cost(self,x,y):
        p=expit(x @ self.w.T + self.b)
        return - np.mean(y*np.log(p) + (1-y)*np.log(1-p))

    def step(self, x,y):
        p=expit(x @ self.w.T + self.b)
        e = p - y
        dw= np.mean(e*x, axis=0)
        self.w = self.w - dw*self.lr

    def fit(self, x,y, numberOfEpochs=1000000):
        self.AllWeights= np.zeros((numberOfEpochs, x.shape[1]))
        self.AllBiases= np.zeros(numberOfEpochs)
        self.AllCosts= np.zeros(numberOfEpochs)
        self.All_cl= np.zeros((numberOfEpochs,len(x)))
        for step in range(numberOfEpochs):
            self.AllWeights[step]=self.w
            self.AllCosts[step]=self.cost(x,y)
            self.All_cl[step]=(self.predict(x)).T.flatten() # predicted y-v
            self.step(x,y)

```

```
# Set y-intercept value and train fixed-intercept logistic regression model:
b_fixed= float(model.b) # set b to the y-intercept value the previous model
model=LogisticRegression_fixed_b(x_train2,y_train2,b_fixed, lr=0.001)
model.fit(x_train2,y_train2, numberOfEpochs=epochs_)

# Stored parameter values of fixed-intercept logistic regression model:
w0=model.AllWeights.T[0]
w1=model.AllWeights.T[1]
c=model.AllCosts
cl=model.All_cl

# Return final parameters and costs of fixed-intercept model:
print("----- Multiple logistic regression model (with fixed y-intercept):")
print("Final weights: " + str(model.w))
print("Final bias: " + str(model.b))
print("Final costs: " + str(model.cost(x_train2,y_train2)))
```

En la regresión logística múltiple, pretendemos ajustar una curva 3D a nuestros datos de entrenamiento. Por esta razón, necesitamos calcular los valores de y a partir de valores de los atributos x_0 y x_1 . Por último, podemos dar a conocer los valores de los parámetros finales y los costos representados en las animaciones para asegurarnos de que visualizamos aproximadamente la convergencia del modelo a pesar de restringir sustancialmente el número de épocas utilizadas.

```
In [ ]: def pred_3d_curve(X,w,b):    # predicts y-values for regression curve spanned
          p=expit(X @ w.T + b)
          return p

n0s = np.linspace(-13, 13, 35)
n1s = np.linspace(-13, 13, 35)
N1, N2 = np.meshgrid(n0s, n1s) # create meshgrid for regression curve

# Meshgrid for plots of costs:
def CrossEntropy_cost(x,y,w,b):  # (same as for simple logistic regression!)
    p=expit(x @ w.T + b)
    return - np.mean(y*np.log(p) + (1-y)*np.log(1-p))

m0s = np.linspace(-0.12, 0.5, 35)
m1s = np.linspace(-0.135, 0.6, 35)
M1, M2 = np.meshgrid(m0s, m1s) # create meshgrid for surface plot/contour plot
zs_1 = np.array([CrossEntropy_cost(x_train2,y_train2,
                                    np.array([[wp0,wp1]]), np.array([[b_fixed]])))
                for wp0, wp1 in zip(np.ravel(M1), np.ravel(M2))])
Z_1 = zs_1.reshape(M1.shape) # z-values of surface plot/contour plot

# Create plot:
fig = plt.figure(figsize=(8,10)) # create figure
gs = gridspec.GridSpec(2, 1, height_ratios=[1.3, 1]) # set height ratio of subplots

# Customize subplots:
label_font_size = 25 # size of label fonts
tick_label_size= 17 # size of tick labels
ax0=fig.add_subplot(gs[0], projection="3d")
ax0.set_title("Logistic regression curve (3D)", fontsize=20) # set title
ax0.view_init(elev=38., azim=-25)
ax0.set_xlabel(r'$x_0$', fontsize=label_font_size, labelpad=8)
ax0.set_ylabel(r'$x_1$', fontsize=label_font_size, labelpad=7)
ax0.set_zlabel("y", fontsize=label_font_size, labelpad=6)
ax0.tick_params(axis='both', which='major', labelsize=tick_label_size)
ax0.tick_params(axis='x', pad=3, which='major', labelsize=tick_label_size)
```

```

ax0.tick_params(axis='y', pad=-2, which='major', labelsize=tick_label_size)
ax1=fig.add_subplot(gs[1], projection="3d")
ax1.view_init(elev=38., azim=-25)
ax1.view_init(elev=38., azim=140)
ax1.tick_params(axis='both', which='major', labelsize=tick_label_size)
ax1.tick_params(axis='x', pad=3, which='major', labelsize=tick_label_size)
ax1.tick_params(axis='y', pad=-2, which='major', labelsize=tick_label_size)
ax1.set_xlabel(r'$w_0$', fontsize=label_font_size, labelpad=14)
ax1.set_ylabel(r'$w_1$', fontsize=label_font_size, labelpad=14)
ax1.set_zlabel("costs", fontsize=label_font_size, labelpad=3)
ax1.set_xticks([0.5, 0.3, 0.1, -0.1])
ax1.set_xticklabels(["0.5", "0.3", "0.1", "-0.1"], fontsize=tick_label_size)
ax1.set_yticks([0.6, 0.4, 0.2, 0])
ax1.set_yticklabels(["0.6", "0.4", "0.2", "0"], fontsize=tick_label_size)
ax1.set_zticks([0.6, 0.7, 0.8, 0.9, 1.0])
ax1.set_zticklabels(["0.6", "0.7", "0.8", "0.9", "1.0"], fontsize=tick_label_size)

# Define which epochs to plot:
a1=np.arange(0,20,2).tolist()
a2=np.arange(20,100,10).tolist()
a3=np.arange(100,200,20).tolist()
a4=np.arange(200,1700,50).tolist()
points_=a1+a2+a3+a4

camera = Camera(fig) # create camera
for i in points_:

    # Plot regression curve:
    w=np.array([[w0[i],w1[i]]])
    zs_0 = np.array([pred_3d_curve(np.array([[wp0,wp1]]),w, np.array([[b_fix
                    for wp0, wp1 in zip(np.ravel(N1), np.ravel(N2))]]))
    z_0 = zs_0.reshape(N1.shape) # z-values of regression curve
    ax0.plot_surface(N1, N2, z_0, rstride=1, cstride=1,
                     alpha=0.4,cmap=cm.coolwarm,
                     antialiased=False)

    # Scatter plot of training data:
    ax0.scatter(x_train2.T[0],x_train2.T[1], y_train2.flatten(),
                marker="x",s=28*2,color="black")

    # Plot dashed connection lines:
    cl_=cl[i]
    for j in range(len(x_train2)):
        x,y,z = [x_train2[j][0],x_train2[j][0],[x_train2[j][1],x_train2[j][1]
        ax0.scatter(x,y,z, color='black',s=0.5)
        ax0.plot(x,y,z, color='black', linewidth=1.5,linestyle='dashed',
                 alpha=0.4)

    # Surface plot of costs:
    ax1.plot_surface(M1, M2, z_1, rstride=1, cstride=1,
                     alpha=0.73,cmap=cm.coolwarm)

    # Plot trajectory of gradient descent:
    ax1.plot(w0.flatten()[0:i],w1.flatten()[0:i],c.flatten()[0:i],
             linestyle="dashed", linewidth=2, color="black")
    ax1.scatter(w0.flatten()[i],w1.flatten()[i],c.flatten()[i],
                marker="o",s=80*2, color="black")

    # Customize legends:
    ax0.legend([f'costs: {np.round(c[i],3)}'], loc=(0, 0.8),
              fontsize=17)
    ax1.legend([f'epochs: {i}'], loc=(0, 0.8),
              fontsize=17)

```

```

    plt.tight_layout()
    camera.snap() # take snapshot after each iteration

animation = camera.animate(interval = 130,
                           repeat = False, repeat_delay = 0) # create animation
from IPython.display import HTML
HTML(animation.to_html5_video())

```

Además, también podemos representar la trayectoria de descenso de gradiente a través de una gráfica de contorno.

```

In [ ]: fig = plt.figure(figsize=(10,10)) # create figure
gs = gridspec.GridSpec(2, 1, height_ratios=[2,1.2]) # set width ratio of subplots

levels = [0.6,0.7,0.8] # set levels for contour lines

# Customize subplots:
label_font_size = 25 # size of label fonts
tick_label_size= 17 # size of tick labels
fig.suptitle('Logistic regression curve (3D)', fontsize=20, y=0.95)
ax0=fig.add_subplot(gs[0], projection="3d") # configure plots as described above
ax0.view_init(elev=28., azim=-30)
ax0.tick_params(axis='both', which='major', labelsize=tick_label_size)
ax0.set_xlabel(r'$x_0$', fontsize=label_font_size, labelpad=9)
ax0.set_xticks([-10,-5.0,0,5.0,10])
ax0.set_xticklabels(["-10","-5.0","0","5.0",10], fontsize=tick_label_size)
ax0.set_yticks([-10,-5.0,0,5.0,10])
ax0.set_yticklabels(["-10","-5.0","0","5.0",10], fontsize=tick_label_size)
ax0.set_ylabel(r'$x_1$', fontsize=label_font_size, labelpad=9)
ax0.set_zlabel(r'$y$', fontsize=label_font_size, labelpad=5)
ax0.tick_params(axis='z', pad=5, which='major', labelsize=tick_label_size)
ax0.tick_params(axis='both', which='major', labelsize=tick_label_size)
ax0.set_zticks([0,0.2,0.4,0.6,0.8,1])
ax0.set_zticklabels(["0","0.2","0.4","0.6","0.8", "1.0"], fontsize=tick_label_size)
ax0.tick_params(axis='x', pad=1, which='major', labelsize=tick_label_size)
ax0.tick_params(axis='y', pad=-2, which='major', labelsize=tick_label_size)
ax0.tick_params(axis='z', pad=2, which='major', labelsize=tick_label_size)
ax1=fig.add_subplot(gs[1])
ax1.tick_params(axis='both', which='major', labelsize=tick_label_size)
ax1.set_xlabel(r'$w_0$', fontdict=None, labelpad=2, fontsize=label_font_size)
ax1.set_ylabel(r'$w_1$', fontdict=None, labelpad=-2, fontsize=label_font_size)

camera = Camera(fig)
for i in points_:

    # Plot regression curve:
    w=np.array([[w0[i],w1[i]]])
    b_ = b_fixed
    zs_0 = np.array([pred_3d_curve(np.array([[wp0,wp1]]),w, np.array([[b_fix]]),
                                    for wp0, wp1 in zip(np.ravel(N1), np.ravel(N2)))])
    Z_0 = zs_0.reshape(N1.shape) # z-values of regression curve
    ax0.plot_surface(N1, N2, Z_0, rstride=1, cstride=1,
                     alpha=0.4,cmap=cm.coolwarm,
                     antialiased=False)

    # Scatter plot of training data:
    ax0.scatter(x_train2.T[0],x_train2.T[1] , y_train2, marker="x", s=11**2)

    # Plot dashed connection lines:
    cl_=cl[i]
    for j in range(len(x_train2)):
        x,y,z = [x_train2[j][0],x_train2[j][1],[x_train2[j][1],x_train2[j][0]]]
        ax0.scatter(x,y,z, color='black',s=0.5)

```

```

        ax0.plot(x,y,z, color='black', linewidth=1.5,linestyle='dashed',alpha=0.85)

    # Contour plot of costs:
    cp = ax1.contour(M1, M2, z_1,levels, colors='black', # contour plot
                      linestyles='dashed', linewidths=1)
    plt.clabel(cp, inline=1,fmt='%1.1f', fontsize=15 ) # add labels to contours
    cp = plt.contourf(M1, M2, z_1, alpha=0.85,cmap=cm.coolwarm) # filled contours

    # Plot trajectory of gradient descent:
    plt.scatter(w0[i],w1[i],marker='o', s=13**2, color='black' )
    plt.plot(w0[0:i],w1[0:i], linestyle="dashed", color='black' )

    # Customize legends:
    ax0.legend([f'costs: {np.round(c[i],3)}'], loc=(0.72, 0.80), fontsize=17)
    ax1.legend([f'epochs: {i}'], loc=(0.72, 0.85), fontsize=17)
    plt.tight_layout()
    camera.snap()

animation = camera.animate(interval = 100, # "interval": delay between frames
                           repeat = False, repeat_delay = 0)
HTML(animation.to_html5_video())
#animation.save('LogReg_3.gif', writer = 'imagemagick')

```

Función de costo de entropía cruzada (CE) vs Error cuadrático medio (MSE)

La función de pérdida de entropía cruzada (CE) en la regresión logística permite demostrar matemáticamente que la función de coste CE es convexa con exactamente un mínimo, que es el mínimo global. Por el contrario, la aplicación de la función de costo de MSE en la regresión logística da como resultado una función de costo no convexa. Se muestra un enfoque gráfico para comparar ambos métodos con los datos de entrenamiento.

```

In [ ]:
w0s_ce = np.linspace(-0.12, 0.5, 35)
w1s_ce = np.linspace(-0.12, 0.6, 35)
M1_ce, M2_ce = np.meshgrid(w0s_ce, w1s_ce) # create meshgrid
zs_ce = np.array([CrossEntropy_cost(x_train2,y_train2, # apply CE-cost function
                                    np.array([[wp0,wp1]]), np.array([[b_fixed]]))
                  for wp0, wp1 in zip(np.ravel(M1_ce), np.ravel(M2_ce))])
z_ce = zs_ce.reshape(M1_ce.shape) # costs calculated with CE

# MSE:
def MSE_cost(x,y, w,b): # define MSE-cost function
    pred=expit(x @ w.T + b)
    # Calculate mean squared error between predicted and actual y-values:
    mse=np.mean((y-pred)**2, axis=0)
    return mse

m0s_mse = np.linspace(-5, 5, 100)
m1s_mse = np.linspace(-5, 5, 100)
M1_mse, M2_mse = np.meshgrid(m0s_mse, m1s_mse) # create meshgrid
zs_mse = np.array([MSE_cost(x_train2,y_train2, # apply MSE-cost function
                            np.array([[wp0,wp1]]), np.array([[b_fixed]]))
                   for wp0, wp1 in zip(np.ravel(M1_mse), np.ravel(M2_mse))])
z_mse = zs_mse.reshape(M1_mse.shape) # costs calculated with MSE

fig = plt.figure(figsize=(10,8))
gs = gridspec.GridSpec(1, 2, width_ratios=[1, 1.2]) # set width ratio
ax0=fig.add_subplot(gs[0],projection='3d')
ax1=fig.add_subplot(gs[1],projection='3d')

# Customize subplots:

```

```

tick_label_size=16
label_font_size=23
ax0.view_init(elev=50., azim=-30)
ax0.tick_params(axis='both', which='major', labelsize=tick_label_size)
ax0.set_xlabel(r'$w_0$', fontsize=label_font_size, labelpad=12)
ax0.set_ylabel(r'$w_1$', fontsize=label_font_size, labelpad=5)
ax0.set_zlabel("costs", fontsize=label_font_size, labelpad=-33)
ax0.tick_params(axis='both', which='major',
                 labelsize=tick_label_size)
ax0.tick_params(axis='x', pad=7, which='major', labelsize=tick_label_size)
ax0.tick_params(axis='y', pad=0, which='major', labelsize=tick_label_size)
ax0.tick_params(axis='z', pad=7, which='major', labelsize=tick_label_size)
ax0.set_xticks([0.5, 0.3, 0.1, -0.1])
ax0.set_xticklabels(["0.5", "0.3", "0.1", "-0.1"], fontsize=tick_label_size)
ax0.set_yticks([0.6, 0.4, 0.2, 0])
ax0.set_yticklabels(["0.6", "0.4", "0.2", "0"], fontsize=tick_label_size)
ax0.set_zticks([0.6, 0.7, 0.8, 0.9, 1.0])
ax0.set_zticklabels(["0.6", "0.7", "0.8", "0.9", "1.0"], fontsize=tick_label_size)
ax1.view_init(azim=120, elev=50)
ax1.tick_params(axis='both', which='major', labelsize=tick_label_size)
ax1.set_xlabel(r'$w_0$', fontsize=label_font_size, labelpad=3)
ax1.set_ylabel(r'$w_1$', fontsize=label_font_size, labelpad=6)
ax1.set_zlabel("costs", fontsize=label_font_size, labelpad=-31)
ax1.tick_params(axis='both', which='major',
                 labelsize=tick_label_size)
ax1.tick_params(axis='x', pad=0, which='major', labelsize=tick_label_size)
ax1.tick_params(axis='y', pad=0, which='major', labelsize=tick_label_size)
ax1.tick_params(axis='z', pad=7, which='major', labelsize=tick_label_size)

# Surface plots of costs calculated with CE:
ax0.plot_surface(M1_ce, M2_ce, Z_ce, rstride=1, cstride=1, alpha=1, cmap=cm.coolwarm)

# Surface plots of costs calculated with MSE:
ax1.plot_surface(M1_mse, M2_mse, Z_mse, rstride=1, cstride=1, alpha=1.0,
                 antialiased=False, cmap=cm.coolwarm) #cm.terrain

plt.tight_layout()
plt.show()

```

In []:

```

In [ ]: fig = plt.figure(figsize=(10,10))
ax1 = fig.add_subplot(111)

levels=15 # set number of levels for contour plot
label_font_size = 25 # size of label fonts
tick_label_size= 17 # size of tick labels

ax1.tick_params(axis='both', which='major', labelsize=tick_label_size)
ax1.set_xlabel(r'$w_0$', fontsize=label_font_size, labelpad=0)
ax1.set_ylabel(r'$w_1$', fontsize=label_font_size, labelpad=-8)

# Mark 'global' minimum ("x"):
ax1.plot([0.08953984], [0.19334633], marker="X", color="black", markersize=10)

# Mark local minimum (asterisk):
ax1.plot([-0.97], [4.2], marker="*", color="black", markersize=12)

# Contour plot & contourf plot:
cp = ax1.contour(M1_mse, M2_mse, Z_mse, levels, colors='black',
                  linestyles='dashed', linewidths=1)
cp = plt.contourf(M1_mse, M2_mse, Z_mse, alpha=0.85, cmap=cm.coolwarm)

```

```
cbar=plt.colorbar() # plot colorbar  
cbar.ax.tick_params(labelsize=14) # set font size of colorbar
```

In []: