

Regresión Logística

IMPORTANTE: En este ejercicio solamente se podrá importar `numpy`, `matplotlib.pyplot` y la base de datos a utilizar (el resto deberá ser implementación propia).

La base de datos MNIST posee imágenes de los dígitos manuscritos (del 0 al 9). Se desea entrenar un clasificador que, a partir de una imagen, prediga si el dígito que aparece en ella es *par* o *impar*.

(a) Exploración de Datos:

1. Cargar la base de datos utilizando `tensorflow.keras.datasets.mnist.load_data`.
2. Utilizando `imshow` (matplotlib) represente 10 muestras del conjunto de testeo elegidas al azar.
3. Tanto para el conjunto de entrenamiento como el de testeo, generar las etiquetas necesarias para esta tarea.

(b) Preprocesamiento:

1. Implementar la normalización (estandarización). La misma debe descartar los *features* con varianza nula, centrar y escalar (volviendo la media nula y la varianza unitaria). El código debe estar estructurado de la siguiente manera:

```
class Normalizar:
    # Opcional, para inicializar atributos o declarar hiperparámetros
    def __init__(self,...



    # Etapa de entrenamiento
    def fit(self,X):

    # Aplicar la transformación
    def transform(self,X):

    # Entrenar y aplicar la transformación
    def fit_transform(self,X):
```

2. Aplicar la normalización a los conjuntos de datos.

(c) Análisis Teórico:

1. Calcular la función inversa $\sigma^{-1}(p)$ para $p \in (0, 1)$, donde $\sigma(z)$ es la función sigmoide.
2. Sea $p = \sigma(z)$, calcular la derivada $\sigma'(z)$. : Si el resultado se expresa en función de p , se simplifica bastante.
3. Hallar una expresión analítica para la función costo y su gradiente. : Tenga en cuenta las expresiones asociadas a una regresión logística de dos clases.

(d) Regresión Logística

1. Implementar una regresión logística binaria utilizando gradiente descendente. El código debe estar estructurado de la siguiente manera:

```
class RegresionLogistica:
    # Opcional, para inicializar atributos o declarar hiperparámetros
    def __init__(self,...

    # Etapa de entrenamiento
    def fit(self,X):

    # Etapa de testeo soft
    def predict_proba(self,X):
```

```
# Etapa de testeo hard
def predict(self,X):

# Computar el Accuracy
def accuracy(self,X,y):

# Computar la Cross-Entropy
def cross_entropy(self,X,y):
```

2. Entrenar el algoritmo eligiendo un *learning rate* y una cantidad de iteraciones tal que el riesgo empírico de entrenamiento parezca converger. Graficar dicho riesgo en función del número de iteraciones.
3. Reportar el *accuracy* y la *cross entropy* de entrenamiento y testeo.

(e) *Curva ROC*: A partir de la salida de `predict_proba` del conjunto de testeo, implementar la curva ROC.

(f) **OPCIONAL**: Implementar su propio **Pipeline** de manera que le permita encapsular la normalización con el clasificador. Efectuarlo de la forma más general posible.