U02 PRÁCTICA 2

ADALINE Y FUNCIÓN LOSS: DESCENSO POR GRADIENTE

SISTEMAS DE APRENDIZAJE AUTOMÁTICO

IES SERRA PERENXISA
TORRENT (VALENCIA)

APRENDIZAJE EN ADALINE

ACTIVIDAD 1: IMPLEMENTAR ADALINE CON GDB EN PYTHON

PASO 1: IMPLEMENTAR ADALINE.

Como Adaline es parecido al perceptrón, cogeremos el código de este último y lo copiamos al fichero ***Adaline.py*** y modificaremos su método ***fit()*** que es donde están las diferencias en como actualiza los pesos para minimizar la función de coste usando el algoritmo de descenso por gradiente.

```python
# coding: utf-8
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap

class AdalineGDB(object):
    """Clasificador ADAptive LInear NEuron de <alumno>.
    Parámetros:
    -----------
    eta: float Learning rate (entre 0.0 y 1.0)
    n_iter: int max. repasos que da (iteraciones) al dataset train
    semilla: int; semilla para el GNA
    Atributos:
    ----------
    w_: 1d-array pesos que aprende después de entrenar.
    coste_: lista valores de la función de coste SSE en cada época (repaso)
    """
    def __init__(self, eta=0.01, n_iter=50, semilla=1):
        self.eta = eta
        self.n_iter = n_iter
        self.semilla = semilla
```

El método **fit()** actualiza los pesos después de evaluar a todos los ejemplos de entrenamiento.

```python
    def fit(self, X, y):
        """ Entrena (Aprende)
        Parámetros:
        -----------
        X: {array}, estructura = [n_ejemplos, n_características]
        y: array, estructura = [n_ejemplos] valores target/label
        Devuelve:
        ---------
        self: object
        """
        rgen = np.random.RandomState(self.semilla)
        self.w_ = rgen.normal(loc=0.0, scale=0.01, size= 1 + X.shape[1]) # pesos
        self.coste_ = []
        for i in range(self.n_iter):
            entradas = self.entradas(X)
            salidas = self.activacion(entradas)
            errores = (y - salidas)
            self.w_[1:] += self.eta * X.T.dot(errores)
            self.w_[0] += self.eta * errores.sum()
            coste = (errores**2).sum() / 2.0
            self.coste_.append(coste)
        return self

    def entradas(self, X):
        return np.dot(X, self.w_[1:]) + self.w_[0]

    def predict(self, X):
        return np.where(self.entradas(X) >= 0.0, 1, -1)

    def activacion(self, X):
        """Calcula la activación lineal"""
        return X
```

PREGUNTA 1: La fórmula de la línea 43 **`self.eta * errores.sum()`** es la actualización de:

O El bias, el parámetro `w0` de la neurona.
O El resto de parámetros `w1 ... wn` de la neurona.

PREGUNTA 2: Si eliminamos o comentamos la línea 40 del código

O No aplicamos la función de activación y la neurona no hará bien su trabajo.
O No pasa nada porque al ser la función identidad en realidad no hace falta aplicarla, no hace cambios, se usa para que se vea que la tiene.

PASO 2: CARGAR DATASET DE PRUEBA.

Preguntamos por el fichero a leer (ruta para llegar al fichero ***`iris.data`***) y vamos a entrenar un clasificador basado en ***`AdalineGDB`*** para que aprenda a clasificar lirios en dos clases (***`iris-setosa`*** e ***`iris-versicolor`***) usando como características la longitud de los sépalos y la de los pétalos.

```
# ## PASO 2: PREPARAR EL DATASET Y ENTRENAR ADALINE
lugar = input('Teclee fichero con dataset Iris: ')
df = pd.read_csv(lugar, header=None, encoding='utf-8')
# Preparar los datos de entrenamiento de Iris
y = df.iloc[0:100, 4].values                # seleccionar ejemplos de setosa y versicolor
y = np.where(y == 'Iris-setosa', -1, 1)  # Codifica -1 (Iris-setosa) +1 (Iris-versicolor)
X = df.iloc[0:100, [0, 2]].values        # extraer longitud de sépalos y pétalos
```

PASO 3: ENTRENAR MODELO Y VER GRÁFICA DE FUNCIÓN COSTE.

Vamos a comenzar con un ***`learning rate`*** de 0.5 y dibujar como se comporta la función de coste:

```
1 learning_rate= 0.5
2 ada1 = AdalineGDB(n_iter=20, eta=learning_rate).fit(X, y)
3 fig, ax = plt.subplots()
4 ax.plot(range(1, len(ada1.coste_)+1), np.log10(ada1.coste_), marker='o')
5 ax.set_yticks(np.log10(ada1.coste_))
6 plt.xlabel('Epocas')
7 plt.ylabel('Coste SSE')
8 plt.title(f'Learning rate {learning_rate}')
9 print("Último coste:", ada1.coste_[-1])
```

El gráfico no nos dice mucho porque el último valor es 8.895 x 10^{125}, es decir es (8895000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000). Cuando tenemos este escenario (datos muy grandes en comparación a otros) necesitamos un gráfico de una altura de kilómetros para visualizar los datos pequeños y que los grandes no anulen su dibujo. Si no tenemos pantallas tan grandes, tenemos otra posibilidad para graficar datos de escalas tan diferentes que es pasarlos a escala logarítmica. Copia el código anterior y en la línea 4 y 5 cambia ***`ada1.coste_`*** por su logaritmo en base 10: ***`np.log10(ada1.coste_)`***.

En cada paso, el algoritmo de aprendizaje de esta neurona procesa todos los datos de entrenamiento "***`modo batch`***": para cada instancia o ejemplo de datos predice el resultado y mide el error que comete, cuando ha estudiado todas las instancias (una época) calcula la función de coste y cambia el valor de sus parámetros para bajar el error. Esto debe hacer que cuanto más estudie más aprenda y el error que cometa vaya bajando.

PREGUNTA 3: Entrega el nuevo gráfico y responde:

- ¿Cómo se ve ahora el gráfico?
- ¿Qué función de coste está usando? (marca una) (_)SSE (_)MSE (_)RMSE (_)GD
- A la vista del gráfico ¿Cuanto más estudia más aprende?
- ¿Qué valor de error comete cuando ha estudiado las 20 veces? ¿Y en escala logarítmica?

PASO 4: ENCONTRAR UN VALOR PARA EL LEARNING RATE.

El error debe ir bajando a medida que estudie, si no lo hace hay algún problema. Vamos a comenzar ajustando el ***learning rate.*** Copia el código anterior en una nueva celda de un *notebook* y vuelve a usar escala lineal (quita los logaritmos de los datos que usamos). Prueba con 0.1 (dividiendo por 5 el anterior) y luego sigue dividiendo por 10 el valor del learning_rate y probando hasta que consigas el mínimo error.

PREGUNTA 4: Indica los valores que has encontrado:

- Valor del ***learning rate*** que hace mínimo el error usando 20 épocas:
- ¿Qué valor de error consigues?

PASO 5: PREPROCESAR LOS DATOS ANTES DE ENTRENAR.

Si escalamos los valores de las características los algoritmos de tipo descenso por gradiente funcionarán mejor. Vamos a escalar los datos a mano antes de iniciar el proceso de aprendizaje. Crea una nueva celda donde primero normalizamos las características. A continuación vuelve a copiar el código anterior y ajusta de nuevo el ***learning_rate*** comenzando por 0.5.

```
media_X = np.mean(X, axis=0)        # media de las columnas
desviacion_X = np.std(X, axis=0)
print("medias:", media_X)
print("desviaciones:", desviacion_X)
X_normal = X - media_X / desviacion_X
#--- Volver a entrenar AdalineGDB(), ajustar learning rate y graficar aprendizaje
```

PREGUNTA 5: Entrega el resultado de la ejecución y responde:

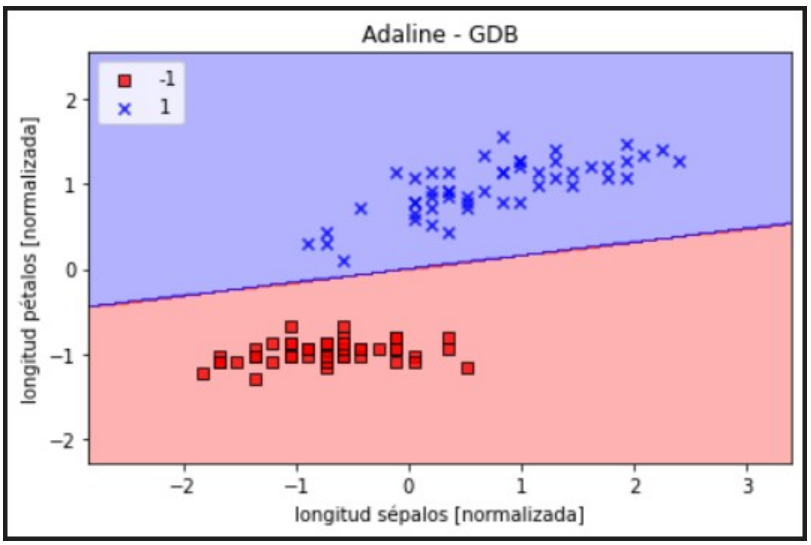
- Valor del ***learning rate*** que hace mínimo el error usando 20 épocas:
- ¿Qué valor de error consigues?
- ¿Mejor o peor si escalas las características?
- Marca las opciones que sean ciertas:
  - [_]Con características escaladas el Descenso por Gradiente aprende más rápido.
  - [_]Para la misma cantidad de entrenamiento (estudio) el Descenso por Gradiente aprende más con características escaladas.
  - [_]El learning rate puede ser mayor con características escaladas.

PASO 5: MOSTRAR FRONTERA DE DECISIÓN.

Crea esta función en una nueva celda a la que pasamos datos de 2 características (2D), la clase de cada instancia (clase binaria: dos clases) y un clasificador y nos dibuje la frontera de decisión que calcula el clasificador.

```
# ## PASO 5: función para dibujar regiones de decisión
def plot_regiones(X, y, clasificador, resolucion=0.02):
    marcadores = ('s', 'x', 'o', '^', 'v')
    colores = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colores[:len(np.unique(y))])
    # Dibujar la superficie de decision
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolucion),
                           np.arange(x2_min, x2_max, resolucion))
    Z = clasificador.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    Z = Z.reshape(xx1.shape)
    plt.contourf(xx1, xx2, Z, alpha=0.3, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())
    # Dibujar la clase de los ejemplos
    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0], y=X[y == cl, 1], alpha=0.8,
                    c=colores[idx], marker=marcadores[idx],
                    label=cl, edgecolor='black')
```

Llama a la función pasando los datos de entrenamiento normalizados, la columna del target y el objeto ***adalineGDB()*** que acabamos de entrenar. Luego pon títulos y etiquetas a los ejes. Debe quedarte un gráfico similar a este:



ACTIVIDAD 2. MEJORAS PARA MUCHOS DATOS: DGS.

Imagina que en vez de 100 ejemplos tuviésemos un dataset con millones de ellos. En ese caso, realizar cada iteración (época) sería muy costoso porque hay que trabajar con todos. Así que entrenar puede llegar a ser una tarea lenta. Esta modalidad del descenso por gradiente que estamos usando se denomina Batch. Una alternativa sería la modalidad del descenso por gradiente estocástico[1] (DGS) también conocido como online o iterativo. Además suele utilizarse un learning rate cambiante:

$$\frac{c_1}{[number\ of\ iterations] + c_2}$$

Donde c1 y c2 son constantes. Esto hace que el tamaño de los pasos vaya decreciendo a medida que el algoritmo va realizando más aprendizaje, porque no entrará en el mínimo global de manera tan directa como el DGB, sino que irá dando más saltos erráticos en todas direcciones. Esto hace que le cueste más dirigirse hacia la zona adecuada donde está el mínimo.

PREGUNTA 6: Copia el código de AdalineDGB.py en AdalineDGS.py (o a otra celda del notebook).

Modificamos el constructor donde añadimos dos variables de instancia:

```
19      def __init__(self, eta=0.01, n_iter=50, semilla=1, desordenado=True):
20          self.eta = eta
21          self.n_iter = n_iter
22          self.semilla = semilla
23          self.desordenado = desordenado
24          self.w_inicializado = False
```

El método para entrenarlo:

1 Estocástico es sinónimo de aleatorio.

```
26      def fit(self, X, y):
27          """ Entrena (Aprende)
28          Parámetros:
29          -----------
30          X: {array}, estructura = [n_ejemplos, n_características]
31          y: array, estructura = [n_ejemplos] valores target/label
32          Devuelve:
33          ---------
34          self: object
35          """
36          self._inicializa_pesos(X.shape[1])
37          self.coste_ = []
38          for i in range(self.n_iter):
39              if self.desordena:
40                  X,y = self._desordena(X,y)
41              coste= []
42              for xi, target in zip(X,y):
43                  coste.append(self._actualiza_pesos(xi,target))
44              costeMedio = sum(coste) / len(y)
45              self.coste_.append(costeMedio)
46          return self
```

Además hacemos un método para entrenar online:

```
48      def parcial_fit(self, X, y):  # Entrena sin reordenar
49          if not self.w_inicializado:
50              self._inicializa_pesos(X.shape[1])
51          if y.ravel().shape[0] > 1:
52              for xi, target in zip(X,y):
53                  self._actualiza_pesos(xi, target)
54          else:
55              self._actualiza_pesos(X, y)
56          return self
```

Y añadimos métodos auxiliares para desordenar, inicializar pesos y actualizarlos en el entrenamiento.

```
58      def _desordena(self, X, y):
59          r = self.rgen.permutation(len(y))
60          return X[r], y[r]
61
62      def _inicializa_pesos(self, m):
63          self.rgen = np.random.RandomState(self.semilla)
64          self.w_ = self.rgen.normal(loc=0.0, scale=0.01, size= 1 + m)
65          self.w_inicializado = True
66
67      def _actualiza_pesos(self, xi, target):
68          salida = self.activacion(self.entradas(xi))
69          error = target - salida
70          self.w_[1:] += self.eta * xi.dot(error)
71          self.w_[0] +=  self.eta * error
72          coste = 0.5 * error**2
73          return coste
```

PREGUNTA 7: Actualiza el código y lo adaptas para crear y entrenar un objeto **`AdalineDGS`** que tenga 15 iteraciones y un learning rate de 0.01, repite lo mismo que hicimos con **`AdalineDGB`** y me pasas los gráficos de las regiones y el entrenamiento.

ACTIVIDAD 3. USAR MODELOS YA PROGRAMADOS.

Normalmente no tendremos que programar a bajo nivel los algoritmos, de hecho, casi siempre los usaremos los que ya tienen prefabricados las librerías. Por ejemplo, el ***Perceptron*** y ***Adaline*** usados como clasificadores son muy básicos, pero los tenemos implementados en las librería ***scikit-learn***. Librerías como esta no solo ofrecen una gran cantidad de algoritmos implementados sino también muchas utilidades para realizar de manera rápida la validación, la medición del desempeño, etc. Si no la tienes instalada aún en tu entorno puedes hacerlo mediante el comando (o alguno similar):

```
pip install scikit-learn
```

Vamos a utilizar el dataset iris que ya viene también preinstalado en esta librería y entrenaremos un perceptrón para que clasifique usando la longitud y la anchura de los pétalos de una orquídea en una de tres categorías (no dos como programamos nosotros anteriormente) sino tres clases. En primer lugar cargamos el dataset y lo dividimos en una parte para entrenamiento y otra para validar como funciona con el método ***train_test_split()*** indicando que el 30% de los datos serán para validar (45 ejemplos) y 70% para entrenar (105 ejemplos).

```
1   # -*- coding: utf-8 -*-
2   from sklearn import datasets
3   import numpy as np
4
5   iris = datasets.load_iris()
6   X = iris.data[:, [2, 3]]
7   y = iris.target
8   print('Diferentes clases:', np.unique(y))
9
10  from sklearn.model_selection import train_test_split
11  X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
12                                                      random_state=1, stratify=y)
13  print('Contar Labels en y:', np.bincount(y))
14  print('Contar Labels en y_train:', np.bincount(y_train))
15  print('Contar Labels en y_test:', np.bincount(y_test))
```

El propio método, antes de realizar la partición desordena los datos por si acaso estaban todos los de una misma clase agrupados. Las clases son 3 (Iris-setosa, Iris-versicolor e Iris-virginica) pero ya están codificadas con los enteros 0, 1, 2.

PREGUNTA 8: Marca las respuestas sobre estos parámetros de la función **`train_test_split()`:**

El parámetro **`ramdon_state=valor`** nos va a permitir:

- O Que si ejecutamos dos veces el código, dará los mismos resultados porque inicializa todo lo que ocurra de manera aleatoria al mismo valor.
- O Que todo ocurra de manera impredecible y difícil de repetir.

El parámetro **`stratify = y`** nos va a permitir:

- O Que los datos queden balanceados, que tengan cantidades similares de cada clase.
- O Que los ejemplos se cojan en grupos como si formasen estratos.

El parámetro **`size = 0.3`** nos va a permitir:

- O Que el 30% de los datos se usan para entrenar.
- O Que el 30% de los datos se usan para testar el modelo.
- O Que no se cogen el 30% de los datos para mejorar la aleatoriedad de los datos.

Escalamos las características normalizándolas o estandarizándolas para mejorar el funcionamiento del algoritmo de gradiente por descenso. Se aplica la transformación por separado al train y al test pero se ha configurado de la misma manera usando los datos del train. Es decir, siempre se usa la misma media y desviación para normalizar ambos conjuntos de datos.

```
17  from sklearn.preprocessing import StandardScaler
18  sc = StandardScaler()
19  sc.fit(X_train)
20  X_train_normalizado = sc.transform(X_train)
21  X_test_normalizado = sc.transform(X_test)
```

Modificamos el método **`plot_regiones()`** para que sea capaz de mostrar varias regiones que crean las fronteras de decisión que define un modelo que es el parámetro **`clasificador`**.

```
33  def plot_regiones(X, y, clasificador, test_idx=None, resolucion=0.02):
34      markers = ('s', 'x', 'o', '^', 'v')
35      colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
36      cmap = ListedColormap(colors[:len(np.unique(y))])
37      # plot the decision surface
38      x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
39      x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
40      xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolucion),
41                             np.arange(x2_min, x2_max, resolucion))
42      Z = clasificador.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
43      Z = Z.reshape(xx1.shape)
44      plt.contourf(xx1, xx2, Z, alpha=0.3, cmap=cmap)
45      plt.xlim(xx1.min(), xx1.max())
46      plt.ylim(xx2.min(), xx2.max())
47      for idx, cl in enumerate(np.unique(y)):
48          plt.scatter(x=X[y == cl, 0], y=X[y == cl, 1], alpha=0.8, c=colors[idx],
49                      marker=markers[idx], label=cl, edgecolor='black')
50      # highlight test samples
51      if test_idx:
52          X_test, y_test = X[test_idx, :], y[test_idx]
53          plt.scatter(X_test[:, 0], X_test[:, 1], c='y', edgecolor='black',
54                      alpha=1.0, linewidth=1, marker='o', s=100, label='test')
```

Usando este método, crea el gráfico:

```
56  X_combined_std = np.vstack((X_train_normalizado, X_test_normalizado))
57  y_combined = np.hstack((y_train, y_test))
58  plot_regiones(X=X_combined_std, y=y_combined, clasificador=ppn, test_idx=range(105, 150))
59  plt.xlabel('longitud pétalos [estandarizado]')
60  plt.ylabel('ancho de pétalos [estandarizado]')
61  plt.legend(loc='upper left')
62  plt.show()
```

PREGUNTA 9: Escribe el código y lo ejecutas. Cuando generes el gráfico, añade tu nombre al título y añade la sentencia que lo guarde en el fichero `U02_P02_09.png`.