Antonio Gallego Peñalver

Documentación Tarea Adaline

adalyne.py

En la línea 1 se añade la clase AdalineGD(), que contiene Adaline con descenso de gradiente. Se compone de los parámetros eta (ratio de aprendizaje), n_iter (numero de iteraciones, épocas) y random_state (semilla para generar valores aleatorios). Compuesto de los atributos w_ (pesos del modelo) y cost_ (función de coste)

Constructor

En la línea 22 hay un constructor que guarda los parámetros de entrada (los mencionados en el párrafo anterior) en variables.

```
def __init__(self, eta=0.01, n_iter=50, random_state=1):
    self.eta = eta  # Guarda el ratio de aprendizaje
    self.n_iter = n_iter  # Guarda el número de iteraciones
    self.random_state = random_state  # Guarda la semilla aleatoria
```

- eta es el ratio de aprendizaje, que vale 0,01
- n_iter es el número de épocas, vale 50
- random_state fija una semilla aleatoria, para tener los mismos números aleatorios

Cada uno de los parámetros se guardan como atributos del objeto

Función fit()

En la línea 27 hay una función llamada fit(), que es para entrenar el modelo.

```
def fit(self, X, y):
    rgen = np.random.RandomState(self.random_state)
    self.w_ = rgen.normal(loc=0.0, scale=0.01, size=1 + X.shape[1])
    self.cost_ = []
```

- X es la matriz de datos de entrada
- Y el vector con la etiquetas asignadas
- rgen genera números aleatorios usando la semilla generada en el constructor, para usar los pesos aleatoriamente
- self.w_ son los pesos del modelo. Se crean con valores cercanos a 0
- rgen.normal() genera números aleatorios de una distribución normal
- self.cost_ guarda el coste de cada época. Con esto se sabe si el modelo está aprendiendo

En la misma función se puede observar un bucle, para entrenar el modelo. El bucle itera el número de epocas que le hemos asignado con n_iter

```
for i in range(self.n_iter):
            net_input = self.net_input(X)
            output = self.activation(net_input)
            errors = (y - output)
            matrizvector=X.T.dot(errors)
            self.w_[1:] += self.eta *
matrizvectorself.w_[0] += self.eta * errors.sum()
            cost = (errors**2).sum() / 2.0
            self.cost_.append(cost)
        return self
```

- net_input = self.net_input(X) calcula la entrada neta, que son la combinacion lineal de entradas y pesos
- output pasa la entrada neta a la funcion de activacion
- errors = y – output calcula los errores. Lo que predijo el modelo menos el resultado esperado
- self.w_[1:] += self.eta * X.T.dot(errors) actualiza los pesos menos el bias.
- cost Calcula el error cuadrático medio
- self.cost_.append(cost) guarda el coste en la lista

La función devuelve el modelo entrenado

Función net_input()

En la línea 51 hay una función que calcula la entrada neta.

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

- Calcula el resultado de multiplicar los pesos por las entradas
- Np.dot(X, self.w_[1:]) multiplica las entradas por sus pesos
- + self.w_[0]: suma el bias.

Función activation()

En la línea 54 hay una función que hace de activación lineal

```
def activation(self, X):
        """Compute linear activation"""
        return X
```

Función predict()

En la línea 58 está la función predict(), para hacer la predicción final

```
def predict(self, X):
        """Retorna la etiqueta de clase después de un paso"""
        return np.where(self.activation(self.net_input(X)) >= 0.0, 1, -1)
```

- net_input(x) calcula la entrada neta
- El valor anterior se pasa a la función de activacion (activation())
- np.where(>= 0.0, 1, -1) Indica que si la salida es mayor o igual que 0, la prediccion vale 1. Si es menor que 0, valdrá -1
- Al final devuele un array con las etiquetas predichas.