optim_tarea09

May 15, 2024

1 Curso de Optimización I

1.1 Tarea 9

1.2 Jairo Saul Diaz Soto

Descripción:	Fechas
Fecha de publicación del documento:	**Mayo 5, 2024**
Fecha límite de entrega de la tarea:	**Mayo 12, 2024**

1.2.1 Indicaciones

- Envie el notebook con los códigos y las pruebas realizadas de cada ejercicio.
- Si se requiren algunos scripts adicionales para poder reproducir las pruebas, agreguelos en un ZIP junto con el notebook.
- Genere un PDF del notebook y envielo por separado.

1.3 Ejercicio 1 (5 puntos)

Construir un clasificador binario basado en el método de regresión logística. Puede revisar las notas de las ayudantías 10 y 11. En particular, podemos tomar de referencia el artículo (minka-logreg.pdf) que aparece en la Ayudantía 10:

> "A comparison of numerical optimizers for logistic regression". Thomas P. Minka

Para usar la notación de este artículo, tenemos un conjunto de datos y cada dato puede pertener a una de dos clases. Las clases se identifican con las etiquetas "-1" y "1". Para hacer la clasificación se necesita determinar un vector $\mathbf{w}$ que se usa para calcular la probabilidad de que un dato $\mathbf{x}_i \in \mathbb{R}^n$ pertenezca a la clase $y_i \in \{-1, 1\}$ mediante la evaluación de la función sigmoide:

$$\sigma(\mathbf{x}_i, y_i, \mathbf{w}) = \frac{1}{1 + \exp(-y_i \mathbf{w}^\top \mathbf{x}_i)}.$$

Cada vector $\mathbf{x}_i$ está formado por el valor de ciertas características asociadas el individuo i-ésimo.

Dada una colección de datos etiquetados $(\mathbf{x}_1, y_1), ..., (\mathbf{x}_m, y_m)$, se mide el error de clasificación mediante

$$L(\mathbf{w}) = \sum_{i=1}^{m} \log(1 + \exp(-y_i \mathbf{w}^\top \mathbf{x}_i)) + \frac{\lambda}{2} \mathbf{w}^\top \mathbf{w}.$$

El segundo término de la expresión anterior penaliza la magnitud de la solución $\mathbf{w}$ dependiendo del valor de λ.

En general, los datos se almacenan en una matriz de modo de cada vector $\mathbf{x}_i$ es una fila de la matriz $\mathbf{X}$ y las etiquetas y_i son las componentes de un vector $\mathbf{y}$:

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}_1^\top \\ \mathbf{x}_2^\top \\ \vdots \\ \mathbf{x}_m^\top \end{bmatrix}, \qquad \mathbf{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix}.$$

1. Muestre que el gradiente de $L(\mathbf{w})$ está dado por

$$\nabla_w L(\mathbf{w}) = -\sum_{i=1}^{m} (1 - \sigma(\mathbf{x}_i, y_i, \mathbf{w})) y_i \mathbf{x}_i + \lambda \mathbf{w}.$$

2. Programar las funciones

$$\sigma(\mathbf{X}, \mathbf{y}, \mathbf{w}), \quad L(\mathbf{w}) \quad \text{y} \quad \nabla_w L(\mathbf{w}).$$

- Conviene programar la función sigmoide para que pueda recibir la matriz $\mathbf{X}$ y el vector $\mathbf{y}$, en lugar de dar un vector $\mathbf{x}_i$ y su etiqueta y_i, para que evalue todos los datos y devuelva un vector con probabilidades

$$\begin{pmatrix} \sigma(\mathbf{x}_1, y_1, \mathbf{w}) \\ \sigma(\mathbf{x}_2, y_2, \mathbf{w}) \\ \vdots \\ \sigma(\mathbf{x}_m, y_m, \mathbf{w}) \end{pmatrix}.$$

- Una vez que se tiene ese vector de probabilidades, se puede calcular el gradiente de $ L()$.

3. Aplique el método de descenso máximo para minimizar la función $L(\mathbf{w})$. Use backtracking para calcular el tamaño de paso α_k, de modo $\mathbf{w}_{k+1} = \mathbf{w}_k + \alpha_k \mathbf{p}_k$, donde

$$\mathbf{p}_k = -\mathbf{g}_k = -\nabla_w L(\mathbf{w}_k)$$

Una vez que se ha calculado el minimizador $\mathbf{w}_*$ de $L(\mathbf{w})$ puede usar la función $predict(\mathbf{X}, \mathbf{w}_*)$, codificada en la siguiente celda, para predecir las etiquetas de los datos que están en la matriz $\mathbf{X}$. Así, esta función devuelve un vector que tiene las etiquetas -1 o 1 que se asigna cada dato (fila) en la matriz $\mathbf{X}$ de acuerdo a la probabilidad que tiene ese dato de pertenecer a una de las clases.

Nota: Hay que implementar la función `sigmoid()` como se indica en el Punto 2 para poder ejecutar la función `predict()`.

La función `predict()` es la respuesta del clasificador en cada dato de la matriz $\mathbf{X}$.

[2]:
```
def sigmoid(X, y, w):
    sol = np.zeros(len(X))
    for i, x in enumerate(X):
        sol[i] = sigmoide(x, y[i], w)
    return sol


# Función para predecir la clase de cada dato (fila) en la matriz X
# Devuelve un arreglo del tamaño de la cantidad de filas de X que tiene
# las etiquetas -1 o 1  que se predicen para cada dato.
# Para calcular las etiquetas, se calcula el vector que tiene las probabilidades
# de que los datos pertenezcan a la clase 1. Si la probabilidad es mayor que 0.
 ↪5,
# se asigna la clase 1. En caso contrario se asigna la clase -1.
#
def predict(X, w):
    # Vector de prediciones. Se inicializa como si todas las etiquetas fueran 1
    y_pred = np.ones(X.shape[0])
    # Vector de probalidades de que los datos pertenezcan a la clase 1
    vprob  = sigmoid(X, np.ones(X.shape[0]), w)
    # Se obtienen los índices de los datos que tienen una probabilidad menor a␣
 ↪0.5
    ii     = np.where(vprob<=0.5)[0]
    # Se cambia la etiqueta por -1 para todos los datos con probabilidad menor␣
 ↪a 0.5
    y_pred[ii] = -1
    return y_pred
```

En general, dado un conjunto de datos, se toma una parte de ellos para construir el clasificador. Ese subconjunto se llama el **conjunto de entrenamiento**. El resto de los datos se usan para evaluar el desempeño del clasificador y se llama el **conjunto de prueba**.

Para evaluar el desempeño del clasificador hay varias métricas. El código de la siguiente celda muestra: - Cómo leer los datos de un archivo, - separarlos en el conjunto de entrenamiento y validación, - estandarizar los datos de cada conjunto, - agregar una columna formada por 1's a los datos. Si no se hace esto, en lugar de usar el producto $\mathbf{w}^\top \mathbf{x}_i$, se tendría que usar $b + \mathbf{w}^\top \mathbf{x}_i$ y calcular el bias b por separado. Al agregar esta columna de 1's a los datos, es como equivalente a que el bias b forme parte del vector $\mathbf{w}$. - Se calcula la matriz de confusión que en su diagonal muestra la cantidad de datos en los que la predicción de la clase que hace el clasificador es correcta, mientras que los elementos fuera de la diagonal son la cantidad de datos mal clasificados. - Se evalúa la exactitud (accuracy) del clasificador. Entre más cerca esté este valor a 1, es mejor el desempeño del clasificador.

El conjunto de datos corresponde a un estudio en el que se miden 13 características a una muestra de 303 individuos, descritas en

Heart disease

Cada registro tiene una etiqueta que indica la presencia (etiqueta 1) de una enfermedad del corazón,

o que no la tiene (etiqueta 0). Esta última etiqueta la cambiamos por "-1" para que coincida con la notación del artículo.

El objetivo es tomar una parte de los datos para crear el clasificador y medir el desempeño del clasificador con el resto los datos, haciendo que el clasificador prediga a que clase pertenece cada dato del conjunto de prueba y comparando las predicciones con la verdadera etiqueta.

[3]:
```
import pandas as pd
import numpy as np

# Lectura de los datos
data = pd.read_csv('heart.csv')
print('Dimensiones de la tabla:', data.shape)
data.head()
```

```
Dimensiones de la tabla: (303, 14)
```

[3]:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	target
0	63	1	3	145	233	1	0	150	0	2.3	0	0	1	1
1	37	1	2	130	250	0	1	187	0	3.5	0	0	2	1
2	41	0	1	130	204	0	0	172	0	1.4	2	0	2	1
3	56	1	1	120	236	0	1	178	0	0.8	2	0	2	1
4	57	0	0	120	354	0	1	163	1	0.6	2	0	2	1

[4]:
```
# Esto muestra cuántos datos se tienen en la clase '0' y en la clase '1'
data.groupby(['target']).size()
```

[4]:
```
target
0    138
1    165
dtype: int64
```

[5]:
```
from sklearn.preprocessing import StandardScaler
# data splitting
from sklearn.model_selection import train_test_split
# data modeling
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay,␣
 ↪accuracy_score
from sklearn.linear_model import LogisticRegression

# Cambiamos la etiqueta 0 por -1
data.loc[data['target']==0, 'target'] = -1
```

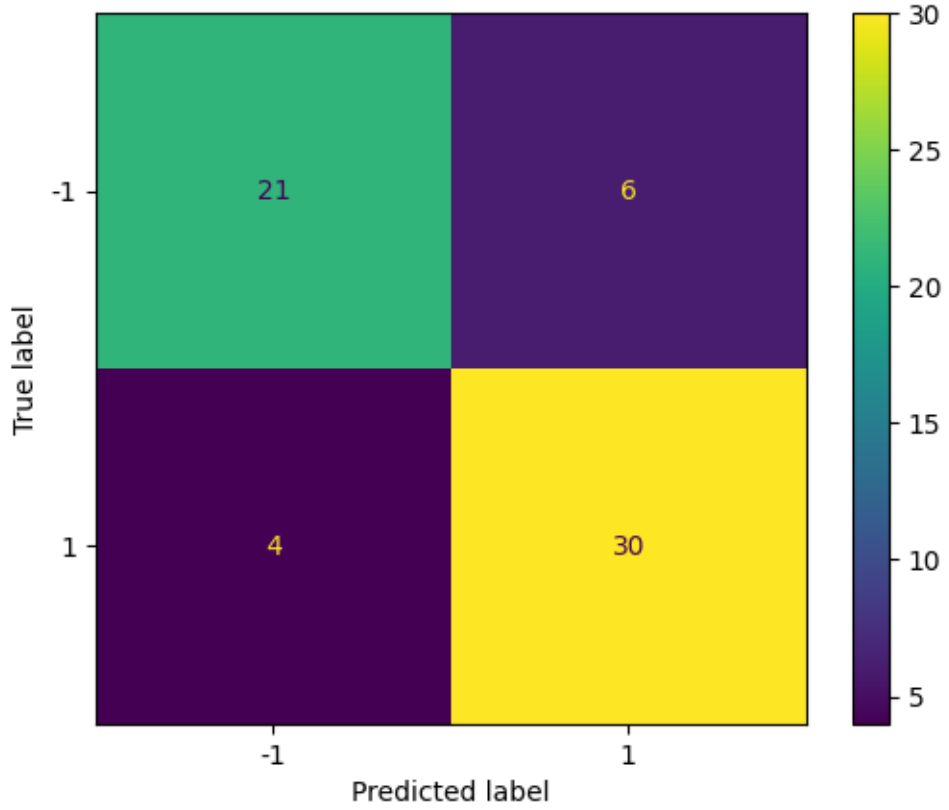
```
# Vector de etiquetas
y  = data["target"]

# Matriz de datos
X  = data.drop('target',axis=1)

# Se usa el 20% de los datos para crear el conjunto de prueba
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20,␣
 ↪random_state = 0)

# Se estandariza cada columna de la matriz de datos para evitar que por tener␣
 ↪diferentes
# rangos de valores cada columna (variable), afecte al algoritmo de optimización
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Número de muestras del conjunto de entrenamiento
ntrain  = X_train.shape[0]
# Se agrega una columna de 1's para que el bias b forme pare del vector w
X_train = np.hstack((np.ones((ntrain,1)), X_train))

# Número de muestras del conjunto de prueba
ntest   = X_test.shape[0]
# Se agrega una columna de 1's para que el bias b forme pare del vector w
X_test  = np.hstack((np.ones((ntest,1)), X_test))

# Se convierte los dataframes a una matriz de numpy
y_train = y_train.to_numpy()
y_test = y_test.to_numpy()

# Se entrena el clasificador de regresión logística
lr    = LogisticRegression(fit_intercept=False)
model = lr.fit(X_train, y_train)

# Imprimimos las componentes de w
w = np.squeeze(model.coef_)
print('w = ')
print(w)

# Se calcula las predicciones para el conjunto de prueba
y_predict = model.predict(X_test)
```

```
w =
[ 0.11473422 -0.07505859 -0.8633645   0.79654126 -0.19299413 -0.24740498
 -0.13380743  0.09214989  0.50584806 -0.47867262 -0.64584814  0.13438099
 -0.88457233 -0.45989107]
```

[6]:
```
# Se mide el desempeño calculando la matriz de confusión y la exactitud
conf_matrix = confusion_matrix(y_test, y_predict)
acc_score = accuracy_score(y_test, y_predict)
print("\nAccuracy:", acc_score,'\n')

disp = ConfusionMatrixDisplay(confusion_matrix=conf_matrix,
                              display_labels=model.classes_)
disp.plot()
```

```
Accuracy: 0.8360655737704918
```

[6]:
```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x224f474f3d0>
```



4. Pruebe el algoritmo de optimización usando $\mathbf{w}_0 = (1, 1, ..., 1)$, el número de iteraciones máximas $N = 500$, la tolerancia para terminar el algoritmo $\tau = \sqrt{ntrain}\epsilon_m^{1/3}$ y para el algoritmo de backtracking $\rho = 0.5, c_1 = 0.001$ $N_b = 500$.

   Cree un clasificador usando $\lambda = 0.001$ y otro clasificador usando $\lambda = 1.0$.

   En cada caso use la función $predict(X_{test}, w_*)$ para obtener el vector de predicciones de la clase para el conjunto de prueba y use el código de la celda anterior para obtener la matriz

de confusión y la exactitud del clasificador, para ver cual de los dos tiene mejor desempeño.

1.3.1 Solución:

Sea:

$$L(\mathbf{w}) = \sum_{i=1}^{m} \log(1 + \exp(-y_i \mathbf{w}^\top \mathbf{x}_i)) + \frac{\lambda}{2}\mathbf{w}^\top \mathbf{w}$$

Entonces al aplicar el gradiente se obtiene que:

$$\nabla L(\mathbf{w}) = \nabla \sum_{i=1}^{m} \log(1 + \exp(-y_i \mathbf{w}^\top \mathbf{x}_i)) + \nabla \frac{\lambda}{2}\mathbf{w}^\top \mathbf{w}$$

$$\nabla L(\mathbf{w}) = \sum_{i=1}^{m} \nabla \log(1 + \exp(-y_i \mathbf{w}^\top \mathbf{x}_i)) + \lambda \mathbf{w}$$

$$\nabla L(\mathbf{w}) = \sum_{i=1}^{m} \frac{1}{1 + \exp(-y_i \mathbf{w}^\top \mathbf{x}_i)} \exp(-y_i \mathbf{w}^\top \mathbf{x}_i)\,(-y_i \mathbf{x}_i) + \lambda \mathbf{w}$$

Ahora bien, siendo que la funcion sigmoide se define como acontinuacion

$$\sigma(\mathbf{x}_i, y_i, \mathbf{w}) = \frac{1}{1 + \exp(-y_i \mathbf{w}^\top \mathbf{x}_i)}$$

Entonces

$$1 - \sigma(\mathbf{x}_i, y_i, \mathbf{w}) = 1 - \frac{1}{1 + \exp(-y_i \mathbf{w}^\top \mathbf{x}_i)}$$

$$1 - \sigma(\mathbf{x}_i, y_i, \mathbf{w}) = \frac{\exp(-y_i \mathbf{w}^\top \mathbf{x}_i)}{1 + \exp(-y_i \mathbf{w}^\top \mathbf{x}_i)}$$

Por lo tanto, sustituyendo lo anterior, es posible reescribir el gradiente de la funcion de costo tal que:

$$\nabla L(\mathbf{w}) = -\sum_{i=1}^{m} (1 - \sigma(\mathbf{x}_i, y_i, \mathbf{w}))\,(y_i \mathbf{x}_i) + \lambda \mathbf{w}$$

[7]:
```
## Funcion sigmoide
def sigmoide(x, y, w):
    arg = -y*(x@w)
    return 1.0 / (1+np.exp(arg))

## Funcion de costo
def loss_func(w,X, y, l):
    cons = (l/2.0)*(w@w)
    sum = 0.0
    for i, x in enumerate(X):
        arg = -y[i]* (w@x)
        sum += np.log(1+np.exp(arg))
```

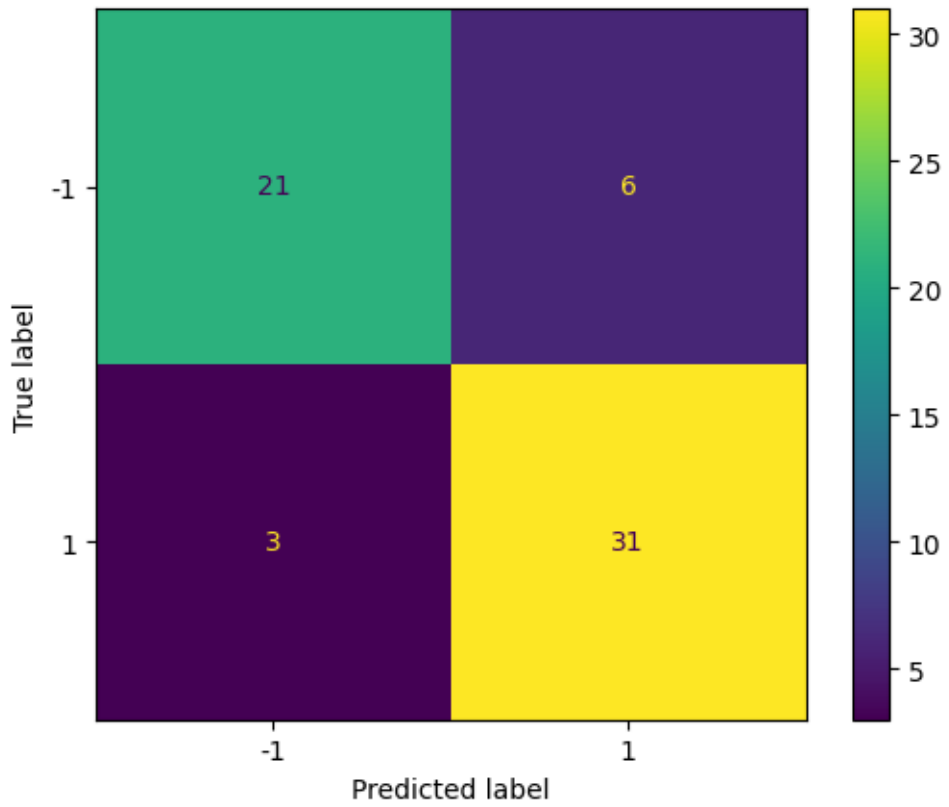
```
    return (cons + sum)

## Gradiente de la funcion de costo
def loss_grad(w, X, y, l):
    sol = np.zeros_like(w)
    cons = l*w
    sum = 0.0
    for i, x in enumerate(X):
        sum += (1 - sigmoide(x, y[i], w)) * (y[i]*x)
    return (cons - sum)

## Backtracking cond. de armijo
def bt_armijo(a0, rho, c0, x0, f_func, grad_fun, p0, NMax):
    a = a0
    f = f_func(x0)
    g = grad_fun(x0)
    for k in range(NMax):
        if f_func(x0 + (a*p0)) <= f + (c0*a*(g@p0)):
            return a
        a *= rho
    return a

## Metodo de descenso maximo
def descmax_bt(f_fun, grad_fun, x0, tau, NMax, a0, rho, c0, KMax):
    record = list()
    for k in range(NMax):
        g = grad_fun(x0)
        p = -g
        a = bt_armijo(a0, rho, c0, x0, f_fun, grad_fun, p, KMax)
        if len(x0) == 2:
            record.append([x0[0], x0[1], a, i])
        if a * np.linalg.norm(p) < tau:
            return x0, k, True, np.array(record)
        x0 += a*p
    return x0, k, False, np.array(record)
```

[8]:
```
## Probando el algoritmo
w0 = np.ones(X_train.shape[1])
NMax = 500
tau = np.sqrt(ntrain) * pow(np.finfo(float).eps, 1.0/3.0)
rho = 0.5
c1 = 0.001
KMax = 500
a0 = 1.0

lamb = 0.001
```

```
def loss_func_(w):
    return loss_func(w, X_train, y_train, lamb)

def loss_grad_(w):
    return loss_grad(w, X_train, y_train, lamb)

wl1__, ki, bl, arr = descmax_bt(loss_func_, loss_grad_, w0, tau, NMax, a0, rho,␣
 ↪c1, KMax)

y_predl1 = predict(X_test, wl1__)

# Se mide el desempeño calculando la matriz de confusión y la exactitud
conf_matrix = confusion_matrix(y_test, y_predl1)
acc_score = accuracy_score(y_test, y_predl1)
print("\nAccuracy:", acc_score,'\n')

disp = ConfusionMatrixDisplay(confusion_matrix=conf_matrix,
                              display_labels=model.classes_)
disp.plot()
```

```
Accuracy: 0.8524590163934426
```

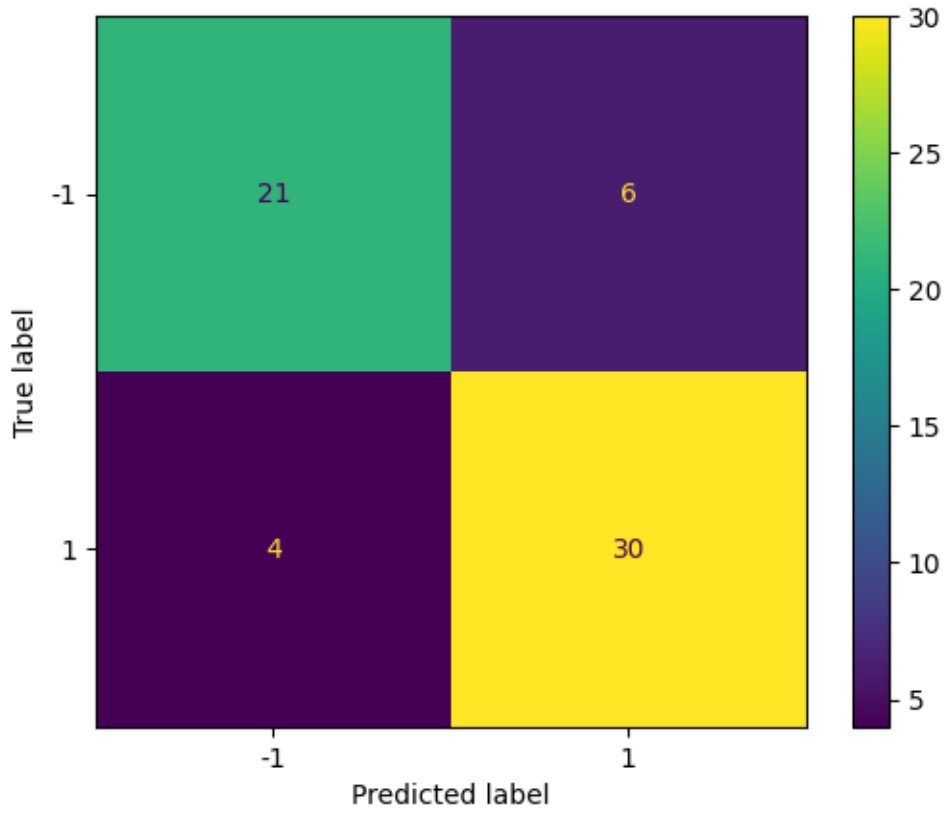
[8]: <sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x224f896c850>



[9]:
```
lamb = 1.0

def loss_func_(w):
    return loss_func(w, X_train, y_train, lamb)

def loss_grad_(w):
    return loss_grad(w, X_train, y_train, lamb)

wl2__, ki, bl, arr = descmax_bt(loss_func_, loss_grad_, w0, tau, NMax, a0, rho,␣
 ↪c1, KMax)

y_predl2 = predict(X_test, wl2__)

# Se mide el desempeño calculando la matriz de confusión y la exactitud
conf_matrix = confusion_matrix(y_test, y_predl2)
acc_score = accuracy_score(y_test, y_predl2)
print("\nAccuracy:", acc_score,'\n')

disp = ConfusionMatrixDisplay(confusion_matrix=conf_matrix,
                              display_labels=model.classes_)
disp.plot()
```

```
Accuracy: 0.8360655737704918
```

[9]: `<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x224fb007cd0>`



1.4 Ejercicio 2 (5 puntos)

Usando el método de Gauss-Newton (Algoritmo 1 de la Clase 26) ajustar el modelo

$$h(t; N_{max}, r, t_0) = \frac{N_{max}}{1 + \exp(-r(t - t_0))}.$$

La variable t representa el tiempo. Los parámetros del modelo son N_{max}, r, t_0.

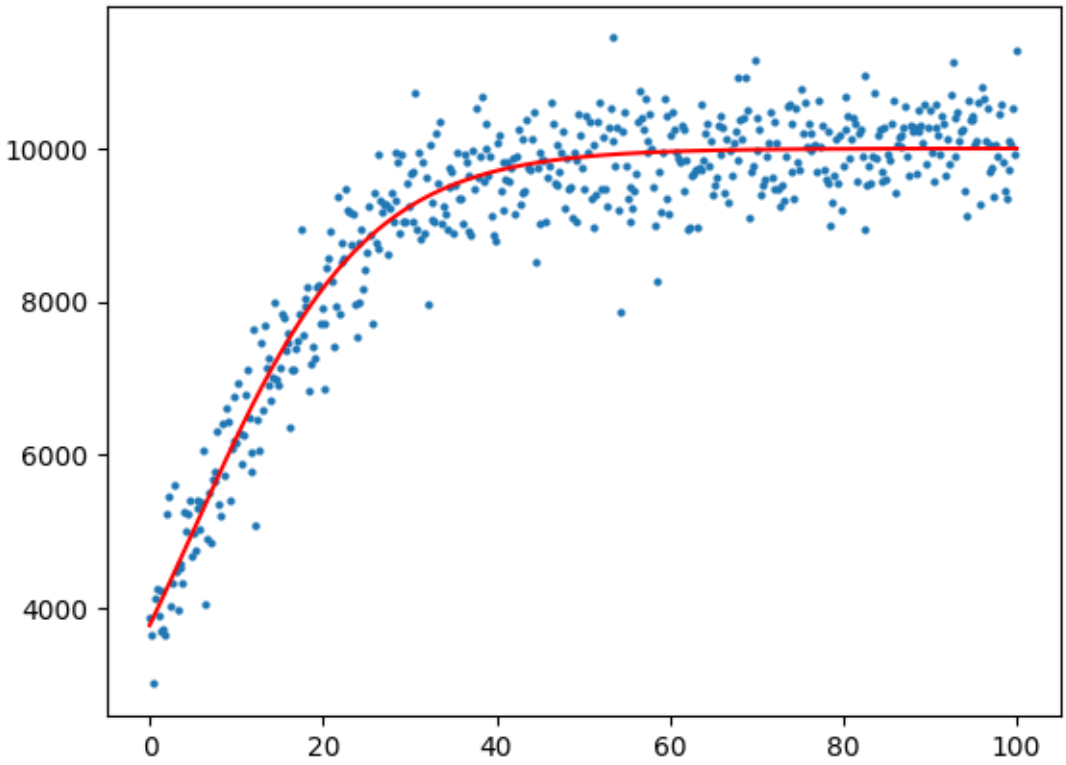
Considere el conjunto de datos $\{(t_1, y_1), (t_2, y_2), ..., (t_m, y_m)\}$ que generaron en la Ayudantía 12.

Los datos están almacenados los vectores $\mathbf{T}$ y $\mathbf{Y}$:

$$\mathbf{T} = \begin{pmatrix} t_1 \\ t_2 \\ \vdots \\ t_m \end{pmatrix}, \qquad \mathbf{Y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix}.$$

[10]:
```
import numpy as np
import matplotlib.pyplot as plt

def fnc_h(t, N_max, r, t0):
    return N_max / (1 + np.exp(-r * (t - t0)))

m = 500
rnd_scale = 5e2
params_hat = (N_max_hat, r_hat, t0_hat) = (1e4, 0.1, 5)
T = np.linspace(0, 100, m)
Y = fnc_h(T, *params_hat) + rnd_scale * np.random.randn(m)

plt.plot(T, Y, 'o', markersize=2)
plt.plot(T, fnc_h(T, *params_hat), 'r')
```

[10]:
```
[<matplotlib.lines.Line2D at 0x224fb00e510>]
```



Para resolver el problema de mínimos cuadrados no lineales hay que definir los residuales como la diferencia entre los que predice el modelo $ h(t_i; N_{max}, r, t_0)$ y el valor observado y_i:

$$r_i(N_{max}, r, t_0) = h(t_i; N_{max}, r, t_0) - y_i, \quad i = 1, 2, ..., m.$$

Si definimos $\mathbf{z} = (N_{max}, r, t_0)$, la función de residuales está dada por

$$\mathbf{R}(\mathbf{z}) = \begin{pmatrix} r_1(\mathbf{z}) \\ r_2(\mathbf{z}) \\ \vdots \\ r_m(\mathbf{z}) \end{pmatrix}.$$

Hay que calcular los parámetros $\mathbf{z} = (N_{max}, r, t_0)$ resolviendo el problema de mínimos cuadrados no lineales.

$$\min_z \ f(\mathbf{z}) = \frac{1}{2} \sum_{i=1}^{m} r_i^2(\mathbf{z}) = \frac{1}{2} \left[\mathbf{R}(\mathbf{z})\right]^\top \mathbf{R}(\mathbf{z}).$$

1. Programe el método de Gauss-Newton de acuerdo con Algoritmo 1 de la Clase 26. Haga que la función devuelva el último punto $\mathbf{z}_k$, el vector $\mathbf{p}_k$ y el número de iteraciones k realizadas.
2. Programe las funciones $\mathbf{R}(\mathbf{z})$, $f(\mathbf{z})$ y la función que calcula matriz Jacobiana $\mathbf{J}(\mathbf{z})$ de $\mathbf{R}(\mathbf{z})$ para el modelo $h(t_i; N_{max}, r, t_0)$.
3. Aplique el método de Gauss-Newton partiendo del punto inicial $\mathbf{z}_0 = (1000, 0.2, 0)$, una tolerancia $\tau = \epsilon_m^{1/3}$

   Imprima el punto $\mathbf{z}_k$ que devuelve el algoritmo, el valor $f(\mathbf{z}_k)$, el número de iteraciones k realizadas y la norma de $\mathbf{p}_k$.
4. Grafique los datos y la curva del modelo usando los valores del punto inicial $\mathbf{z}_0$ y del punto $\mathbf{z}_k$ que devuelve el algoritmo, como lo hicieron en la ayudantía.

1.4.1 Solución:

[62]:
```
def bt_armijo(a0, rho, c0, x0, f_func, grad, p0, NMax):
    a = a0
    f = f_func(x0)
    for k in range(NMax):
        if f_func(x0 + (a * p0)) <= f + (c0 * a * (grad @ p0)):
            return a
        a *= rho
    return 0.001

def gauss_newton(ffun, resfun, jacfun, x0, Y, T, NMax, tau, a0, rho, c1, NBmax):
    xk = x0
    for i in range(NMax):
        rk = resfun(T, Y, *xk)
```

```
        jk = jacfun(xk, T, Y)
        gk = jk.T @ rk  # Gradiente de la función de costo
        Lk = np.linalg.cholesky(jk.T @ jk)
        pk = np.linalg.solve(Lk.T, np.linalg.solve(Lk, -gk))
        if np.linalg.norm(pk) < tau:
            return xk, pk, i, True
        ak = bt_armijo(a0, rho, c1, xk, lambda z: ffun(z, T, Y), gk, pk, NBmax)
        xk += ak * pk
    return xk, pk, NMax, False

def fnc_h(t, N_max, r, t0):
    return N_max / (1 + np.exp(-r * (t - t0)))

def res_h(T, Y, N_max, r, t0):
    return fnc_h(T, N_max, r, t0) - Y

def jac_h(z, T, Y):
    N_max, r, t0 = z
    jac = np.zeros((len(T), len(z)))
    exp_term = np.exp(-r * (T - t0))
    denom = 1 + exp_term
    jac[:, 0] = 1 / denom
    jac[:, 1] = N_max * (T - t0) * exp_term / (denom**2)
    jac[:, 2] = -N_max * r * exp_term / (denom**2)
    return jac

def f_fun(z, T, Y):
    return 0.5 * np.linalg.norm(res_h(T, Y, *z))**2
```

[69]:
```
z0 = np.array([1e3, 0.2, 0])
tol = pow(np.finfo(float).eps, 1.0/3.0)

m = 500
rnd_scale = 5e2
params_hat = (N_max_hat, r_hat, t0_hat) = (1e4, 0.1, 5)
T = np.linspace(0, 100, m)
Y = fnc_h(T, *params_hat) + rnd_scale * np.random.randn(m)

zk, pk, k, bl = gauss_newton(f_fun, res_h, jac_h, z0, Y, T, 5000, tol, 1.0, 0.
↪5, 0.001, 500)
```

[70]:
```
print('El valor obtenido de los parámetros es:', zk)
print('El número de iteraciones fue:', k)
print('Se cumplió con el criterio de paro:', bl)
print('La norma del gradiente fue:', np.linalg.norm(pk))
```

```
El valor obtenido de los parámetros es: [1.00195951e+04 1.00444871e-01
5.12449694e+00]
```

```
El número de iteraciones fue: 11
Se cumplió con el criterio de paro: True
La norma del gradiente fue: 5.568781485219931e-06
```

[72]:
```
z0 = np.array([1e3, 0.2, 0])
plt.plot(T, Y, 'o', markersize=2)
plt.plot(T, fnc_h(T, *params_hat), 'r', label='Original')
plt.plot(T, fnc_h(T, *z0), 'g', label='Punto inicial')
plt.plot(T, fnc_h(T, *zk), 'b', label='Solución final')
plt.legend()
plt.show()
```