C2 OPTIMIZACIÓN DE UN CLASIFICADOR DE SPAM MEDIANTE REGRESIÓN LOGÍSTICA

November 26, 2025

1 Contexto

En el ámbito de la Ingeniería en Computación y la Ciberseguridad, la detección de correos no deseados (spam) es un problema clásico de **clasificación binaria**.

Imaginemos un servidor de correos que procesa millones de mensajes diarios. No podemos escribir reglas manuales para todo, ya que los spammers evolucionan. Necesitamos un modelo estadístico que aprenda la probilidad de que un correo sea spam basándose en sus características.

El problema de optimización: El modelo recibe un vector de características x (por ejemplo, frecuencia de la palabra "gratis", longitud del asunto, presencia de enlaces, etc.) y debe calcular una probabilidad. Nuestro objetivo es encontrar el vector de pesos (coeficientes) β óptimo que minimice el error entre la probabilidad predicha y la etiqueta real del correo (spam = 1, no-spam = 0).

Desde la estadística, esto equivale a maximizar la verosimilitud de los datos observados dado el modelo. Computacionalmente, convertimos esto en un problema de **minimización de la log-verosimilitud negativa**.

2 Fundamento matemático

En un problema de clasificación binaria (spam vs. no-spam), la variable objetivo Y solo puede tomar dos valores: $\{0, 1\}$. Definimos nuestra hipótesis $h_\beta(x)$ como la probabilidad estimada de que $Y = 1$ dado un vector de características x:

$$h_\beta(x) = \mathsf{P}(Y = 1|x; \beta)$$

Por consiguiente, la probabilidad de que sea la clase 0 es:

$$\mathsf{P}(Y = 0|x; \beta) = 1 - h_\beta(x)$$

Podemos combinar estas dos ecuaciones en una sola expresión matemática compacta que funciones tanto si $Y = 1$ como si $Y = 0$. Esta es la función de masa de probabilidad de una **distribución Bernuilli**:

$$\mathsf{P}(Y = y|x; \beta) = (h_\beta(x))^y(1 - h_\beta(x))^{1-y}$$

Ahora, supongamos que tenemos m muestras de entrenamiento independientes entre sí. La probabilidad de observar **todo** el conjunto de datos (etiquetas y) simultáneamente es el producto de las probabilidades individuales. A esto lo llamamos **verosimilitud** (L):

$$L(\beta) = \prod_{i=1}^{m} \mathsf{P}(Y = y^{(i)} | x^{(i)}; \beta)$$

Sustituyendo la ecuación de Bernoulli:

$$L(\beta) = \prod_{i=1}^{m} \left(h_\beta(x^{(i)})\right)^{y^{(i)}} \left(1 - h_\beta(x^{(i)})\right)^{1-y^{(i)}}$$

El objetivo de la estadística inferencial es encontrar los parámetros β que **maximicen** esta probabilidad $L(\beta)$. Es decir, queremos los parámetros que hagan más probable haber observado los datos que realmente tenemos.

Trabajar con productos de probabilidades (números entre 0 y 1) es computacionalmente peligroso en ingeniería:

1. **Desbordamiento numérico (underflow):** Multiplicar muchos números pequeños tiende rápidamente a cero, perdiendo precisión.
2. **Derivadas complejas:** Derivar un producto gigante es muy difícil.

Para solucionar esto, aplicamos el **logaritmo natural** (log). Como el logaritmo es una función monótona creciente, el β que maximiza L también maximiza $\log(L)$.

$$\ell(\beta) = \log(L(\beta))$$

Aplicando propiedades de logaritmos:

$$\ell(\beta) = \sum_{i=1}^{m} \left[y^{(i)} \log\left(h_\beta(x^{(i)})\right) + (1 - y^{(i)}) \log\left(1 - h_\beta(x^{(i)})\right)\right]$$

En optimización numérica, los algorimtos como BFGS están diseñados para buscar **mínimos**, no máximos.

Por lo tanto:

1. Multiplicamos por -1 para invertir la función.
2. Dividimos por m (el número de datos) para obtener el costo promedio. Esto es útil para que la magnitud del costo no dependa del tamaño del conjunto de datos.

Esto nos da finalmente la función de costo **log-verosimilitud negativa** (tabién se le suele llamar, entropía cruzada binaria):

$$J(\beta) = -\frac{1}{m}\ell(\beta)$$

$$J(\beta) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log\left(h_\beta(x^{(i)})\right) + (1-y^{(i)})\log\left(1-h_\beta(x^{(i)})\right)\right]$$

Esta función es **convexa**, lo que es una propiedad crucial, ya que, garantiza que cualquier mínimo local encontrado por el algoritmo es el mínimo global.

Utilizamos la función sigmoide para mapear cualquier valor real a un intervalo $(0, 1)$, interpretandolo como probabilidad.

$$h_\beta(x) = \frac{1}{1+e^{-\beta^\top x}}$$

3 Algoritmo seleccionado: Quasi-Newton (BFGS)

Para este problema, seleccionamos el algoritmo **BFGS (Broyden-Fletcher-Goldfarb-Shanno)**:

- **¿Por qué?** En problemas de ingenería real con muchas características (miles de palabras en un diccionario spam), calcular la segunda derivada (matriz Hessiana) es computacionalmente costoso ($O(n^2)$ en espacio y $O(n^3)$ para invertirla).
- **Mecanismo:** BFGS utiliza la información de los gradientes pasados para construir una **aproximación** de la matriz Hessiana inversa iteración tras iteración.
- **Ventaja:** Converge mucho más rápido que el Descenso de Gradiente y es más eficiente que el método de Newton puro.

4 Implementación

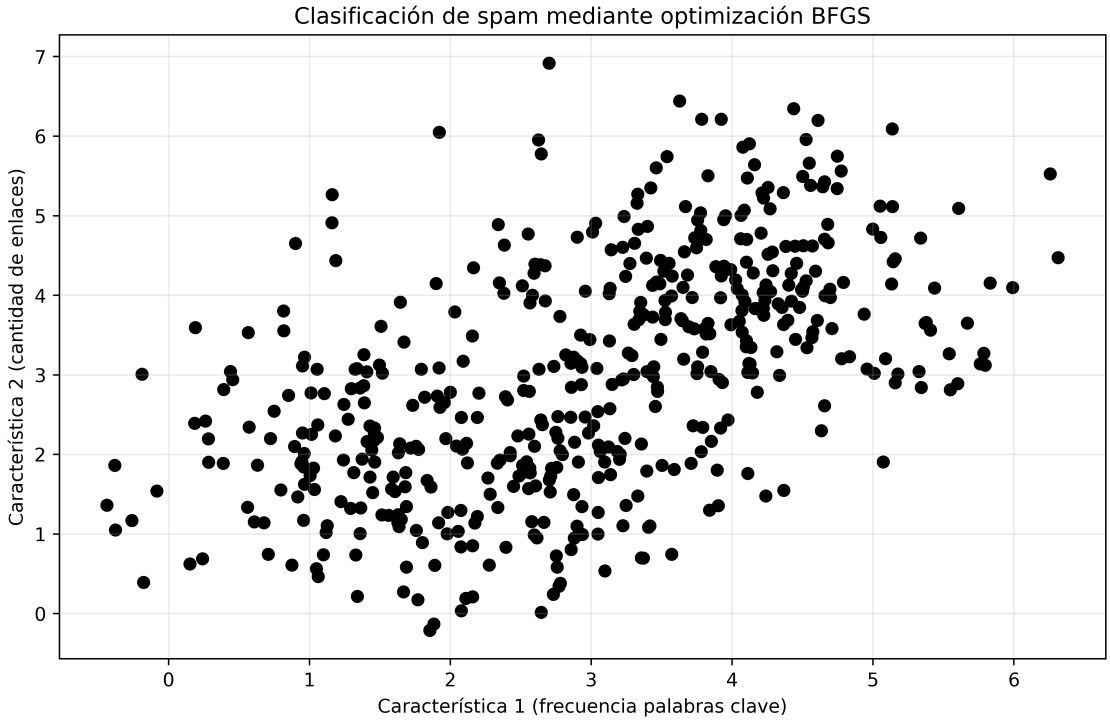
[1]:
```
# Cargar librerías necesarias
# ---------------------------
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import minimize

# Configuraciones iniciales
# -------------------------
plt.rcParams["figure.dpi"] = 600
```

[2]:
```
# Generación de datos simulados
# -----------------------------
np.random.seed(2025)

# Simulamos 2 características:
# x1: Frecuencia de palabra clave (por ejemplo, "oferta", "gratis")
# x2: Cantidad de enlaces en el correo
# m: número de correos
m = 500
```

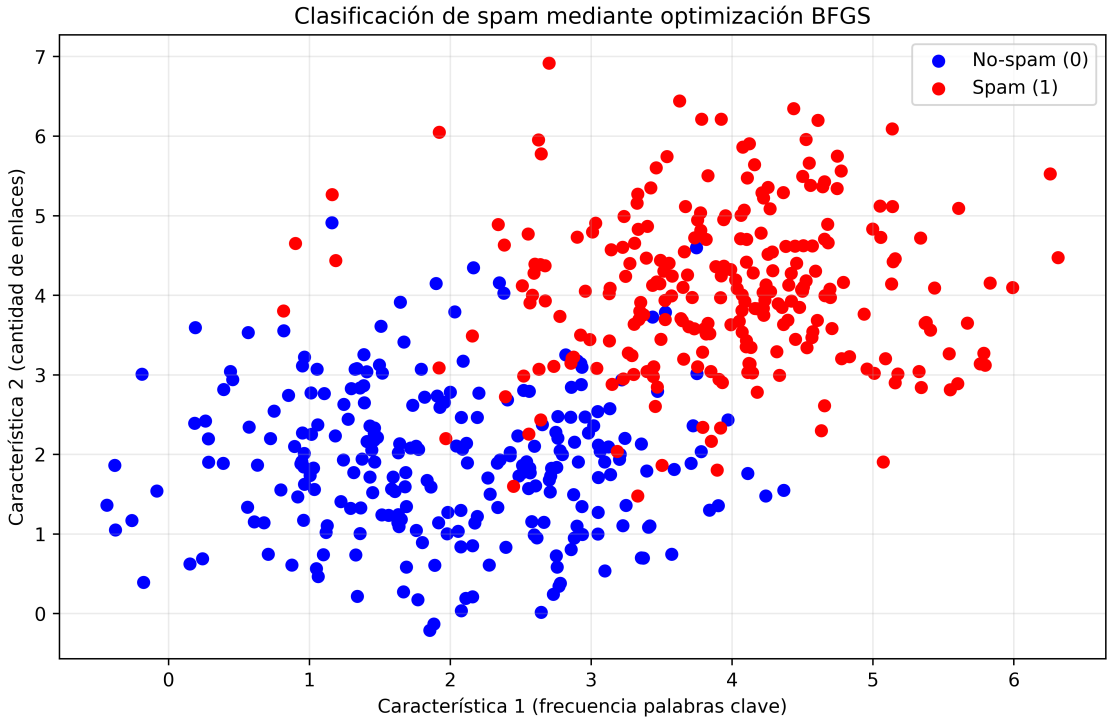
```
# Generamos correos No-spam (clase 0)
# Centrados en (2, 2) con dispersión
X_no_spam = np.random.randn(250, 2) + [2, 2]
y_no_spam = np.zeros(250)

# Generamos correos Spam (clase 1)
# Centrados en (4, 4) con dispersión
X_spam = np.random.randn(250, 2) + [4, 4]
y_spam = np.ones(250)

# Concatemanos los datos
X_brutos = np.vstack((X_no_spam, X_spam))
y = np.hstack((y_no_spam, y_spam))

# Añadimos el sesgo (intercepto): Columna de 1's al inicio
# Esto permite que la recta de separación no tenga que
# pasar por el origen (0, 0)
X = np.column_stack((np.ones(m), X_brutos))

print(f"Dimensiones de X: {X.shape}")
```

```
Dimensiones de X: (500, 3)
```

[3]:
```
# Visualización de los datos
# --------------------------
fig, ax = plt.subplots(figsize = (10, 6))

ax.scatter(X_brutos[:, 0], X_brutos[:, 1], color = "black")
ax.set_xlabel("Característica 1 (frecuencia palabras clave)")
ax.set_ylabel("Característica 2 (cantidad de enlaces)")
ax.set_title("Clasificación de spam mediante optimización BFGS")
plt.grid(True, alpha = 0.25)
plt.show()
```



[4]:
```
# Visualización de los datos
# --------------------------
fig, ax = plt.subplots(figsize = (10, 6))

ax.scatter(X_brutos[y == 0][:, 0], X_brutos[y == 0][:, 1], color = "blue",␣
 ↪label = "No-spam (0)")
ax.scatter(X_brutos[y == 1][:, 0], X_brutos[y == 1][:, 1], color = "red", label␣
 ↪= "Spam (1)")
ax.set_xlabel("Característica 1 (frecuencia palabras clave)")
ax.set_ylabel("Característica 2 (cantidad de enlaces)")
ax.set_title("Clasificación de spam mediante optimización BFGS")
ax.legend()
plt.grid(True, alpha = 0.25)
plt.show()
```



[5]:
```
# Definición de funciones numéricas
# ---------------------------------
def sigmoide(z):
    # Clip para evitar desbordamiento numérico (overflow) en exp()
    z = np.clip(z, -500, 500)
    return 1.0 / (1.0 + np.exp(-z))

def funcion_objetivo(beta, X, y):
    m = len(y)
    h = sigmoide(np.dot(X, beta))
    epsilon = 1e-5
    log_ver_neg = -(1 / m) * np.sum(y * np.log(h + epsilon) + (1 - y) * np.
 ↪log(1 - h + epsilon))
    return log_ver_neg
```

[6]:
```
# Ejecución de la optimización
# ----------------------------

# Inicialización de los coeficienes (betas)
beta_inicial = np.zeros(X.shape[1])

# Ejecución
resultado = minimize(fun = funcion_objetivo,
```

```
                    x0 = beta_inicial,
                    args = (X, y),
                    method = "BFGS")

beta_optimo = resultado.x
print(resultado)
```

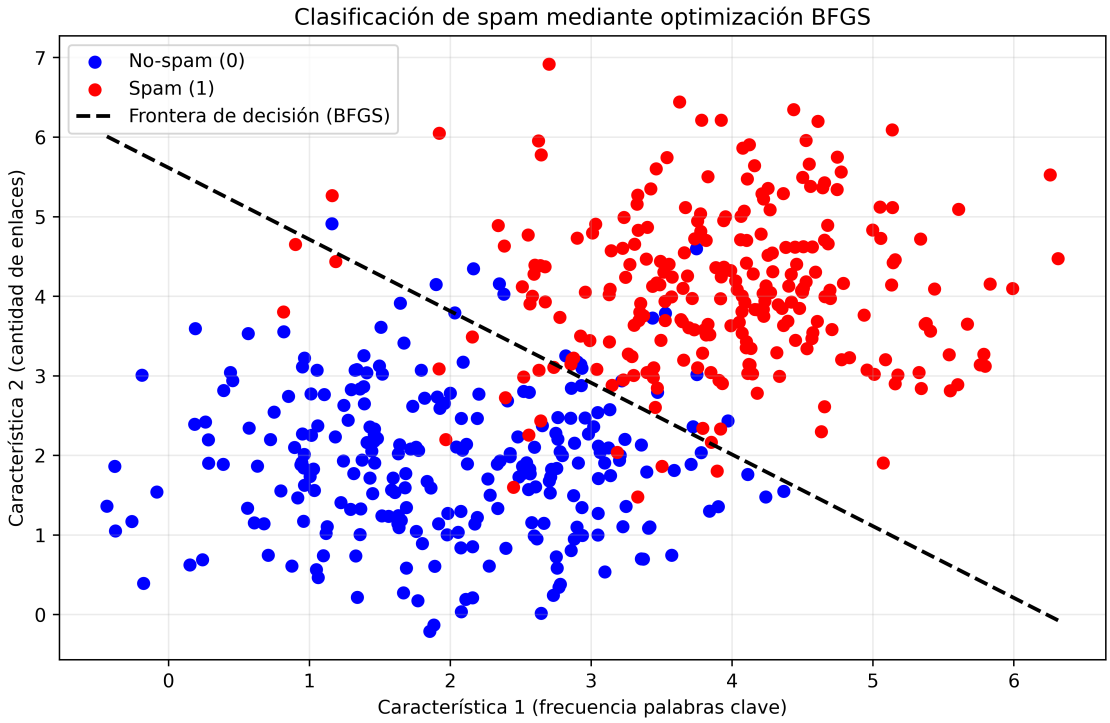
```
  message: Optimization terminated successfully.
  success: True
   status: 0
      fun: 0.17307559612890794
        x: [-1.208e+01  1.938e+00  2.151e+00]
      nit: 20
      jac: [-3.180e-06 -9.762e-06 -7.916e-06]
 hess_inv: [[ 7.352e+02 -1.261e+02 -1.197e+02]
            [-1.261e+02  3.153e+01  1.130e+01]
            [-1.197e+02  1.130e+01  3.003e+01]]
     nfev: 92
     njev: 23
```

[7]:
```
# Resultados
print("=== RESULTADOS ===")
print(f"Costo final: {resultado.fun:.4f}")
print(f"Coeficientes óptimos (beta): {beta_optimo}")
```

```
=== RESULTADOS ===
Costo final: 0.1731
Coeficientes óptimos (beta): [-12.07880185   1.93788125   2.15137755]
```

[8]:
```
# Visualización de la frontera de decisión
fig, ax = plt.subplots(figsize = (10, 6))

ax.scatter(X_brutos[y == 0][:, 0], X_brutos[y == 0][:, 1], color = "blue",␣
 ↪label = "No-spam (0)")
ax.scatter(X_brutos[y == 1][:, 0], X_brutos[y == 1][:, 1], color = "red", label␣
 ↪= "Spam (1)")

# Calcular la recta de decisión
# La frontera es donde h(x) = 0.5 => beta0 + beta1 * x1 + beta2 * x2 = 0
# Despejamos x2: x2 = -(beta0 + beta1 * x1) / beta2
x1_valores = np.array([np.min(X_brutos[:, 0]), np.max(X_brutos[:, 0])])
x2_valores = -(beta_optimo[0] + beta_optimo[1] * x1_valores) / beta_optimo[2]
ax.plot(x1_valores, x2_valores, "k--", linewidth=2, label = "Frontera de␣
 ↪decisión (BFGS)")

ax.set_xlabel("Característica 1 (frecuencia palabras clave)")
ax.set_ylabel("Característica 2 (cantidad de enlaces)")
ax.set_title("Clasificación de spam mediante optimización BFGS")
```

```
ax.legend()
ax.grid(True, alpha = 0.25)
plt.show()
```



Juan F. Olivares Pacheco (jfolivar@uda.cl)

Universidad de Atacama, Facultad de Ingeniería, Departamento de Matemática