Support Vector Machines nos dados covtype de LIBSVM

Iremos treinar os dados para classificação binária usando os dados `covtype` da biblioteca LIBSVM. Veja também Kaggle-LIBSVM.

Desta vez, iremos usar o algoritmo SVM:

$$\min_{w\in\mathbb{R}^{d\times 1}} f_n(w) = \frac{1}{n}\sum_{i=1}^{n} \max\{0, 1 - y_i(x_i^\top w)\} + \frac{\gamma}{2}\|w\|^2,$$

onde $x_i \in \mathbb{R}^{d\times 1}$, $y_i \in \{-1, 1\}$. Uma diferença em relação à regressão logística é que a função custo não é diferenciável. Um subgradiente num ponto w é dado por

$$g_n(w) = -\frac{1}{n}\sum_{i=1}^{n} 1_{\{y_i(x_i^\top w)<1\}} y_i x_i + \gamma w.$$

Portanto, poderíamos usar o método subgradiente. Neste projeto, entretanto, iremos considerar o caso em que n é muito grande de modo que avaliar $f(w)$ ou $g(w)$ é muito custoso. Note que $f(w)$ é a média empírica de

$$F_i(w) := \max\{0, 1 - y_i(x_i^\top w)\} + \frac{\gamma}{2}\|w\|^2.$$

Alternativamente iremos usar em cada iteração um único ponto da amostra e aplicar o método subgradiente estocástico com o subgradiente:

$$G_i(w) := -1_{\{y_i(x_i^\top w)<1\}} y_i x_i + \gamma w.$$

NOTA: neste projeto os labels devem estar em $\{-1, 1\}$!

In [ ]:
```
# Importação de módulos necesserários:

import matplotlib
import numpy as np
import scipy
import seaborn as sns
import matplotlib.pyplot as plt
import numpy.linalg as la
from sklearn.datasets import load_svmlight_file
from sklearn.utils.extmath import safe_sparse_dot
```

In [ ]:
```
# Lendo os dados do arquivo covtype e guardando na matriz de dados X e vetor de labels y:
```

```
data = load_svmlight_file('./datasets/covtype.bz2')
X, y = data[0].toarray(), data[1]
if (np.unique(y) == [1, 2]).all():
    # Devemos garantir que os labels estão em {-1, 1}
    y[y==1.] = -1
    y[y==2.] = 1

n, d = X.shape # tamanho da amostra, número de features
```

Exercício 1: Funções auxiliares

1. Construa uma função `f(w, l2, m)` que toma o iterado `w` e a penalização γ (= `l2`) e retorna o valor funcional $f_m(w)$ para algum $m \in [n]$.
2. Construa uma função `G(w, i, l2)` toma o iterado `w`, os dados (x_i, y_i) e a penalização γ (= `l2`) e retorna o subgradiente $G_i(w)$.
3. Construa uma função `g(w, l2, m)` que toma o iterado `w` e a penalização γ (= `l2`) e retorna o subgradiente $g_m(w)$ para algum $m \in [n]$ --- isto é, a média de m subgradientes.
4. Construa uma função `gB(w, l2, B)` que toma o iterado `w` e a penalização γ (= `l2`) e retorna o gradiente $g_B(w) = \frac{1}{B}\sum_{i \in I_B} G_i(w)$ para algum $B \in [n]$ onde $I_B \subset [n]$ é escolhido aleatoriamente/uniformente.

In [4]:
```
#Escreva o código aqui
def f(w:np.ndarray, l2:float, m:int) -> float:
    indices = np.random.choice(n, m, replace=False)
    X_m = X[indices]
    y_m = y[indices].reshape(-1, 1)
    hinge_loss = np.maximum(0, 1 - y_m * X_m @ w)
    return np.mean(hinge_loss) + (l2 / 2) * la.norm(w) ** 2

def G(w:np.ndarray, i:int, l2:float) -> np.ndarray:
    x_i = X[i]
    y_i = y[i]
    if y_i * x_i @ w < 1:
        return -y_i * x_i + l2 * w
    else:
        return l2 * w

def g(w:np.ndarray, l2:float, m:int) -> np.ndarray:
    indices = np.arange(m)
    subgradients = np.array([G(w, i, l2) for i in indices])
    return np.mean(subgradients, axis=0)

def gB(w:np.ndarray, l2:float, B:int) -> np.ndarray:
    indices = np.random.choice(n, B, replace=False)
```

```
    subgradients = np.array([G(w, i, l2) for i in indices])
    return np.mean(subgradients, axis=0)
```

Inicialização

Fixaremos:

In [5]:
```
m = 30000
l2 = 1e+2
w0 = np.zeros(d)        # ponto inicial
it_max = 80000          # número de iterações
B = 100
```

Exercício 2: Método subgradiente estocástico 1

Iremos implementar o algoritmo Pegasos

Construa uma função `sgd(f, G, w0, lr, l2, m, it_max)` que toma como entrada as funções `g()` `G()`, o ponto inicial `w0`, a regularização `l2` $(= \gamma)$, passo `lr`, `m` e implementa o método subgradiente estocástico em `it_max` iterações iniciando de `w0`:

$$w_{k+1} = w_k - \alpha_k G_k(w_k),$$

com passo $\alpha_k = \frac{lr}{\gamma k}$. **Na k-ézima iteração, use o ponto amostral (x_k, y_k), na ordem do data set `X,y`**. Esta função deve retornar a sequência

$$k \mapsto f_m(w_k) = \frac{1}{m}\sum_{i=1}^{m} \max\{0, 1 - y_i(x_i^\top w_k)\} + \frac{\gamma}{2}\|w_k\|^2.$$

A função também deve retornar o último iterado.

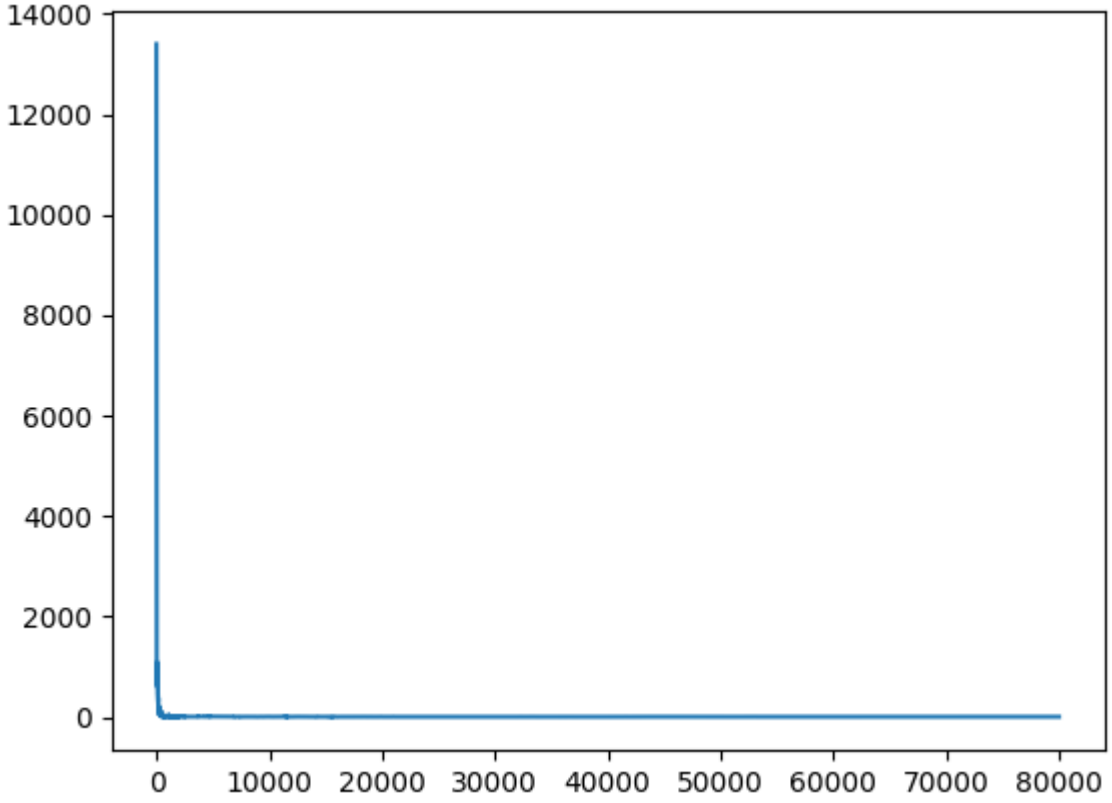
In [6]:
```
#Escreva o código aqui
def sgd(f, G, w0:np.ndarray, lr:float, l2:float, m:int, it_max:int) -> tuple[list[float], np.ndarray]:
    w = w0.copy()
    f_values = list()

    for k in range(it_max):
        alpha_k = lr / (l2 * (k + 1))
        i = k % n

        w -= alpha_k * G(w, i, l2)
```

```
        f_values.append(float(f(w, l2, m)))

    return f_values, w
```

In [7]:
```
# gradient descent
f1 = sgd(f, G, w0, 1e-1, l2, m, it_max)
```

In [8]:
```
plt.plot(f1[0])
```

Out[8]:
```
[<matplotlib.lines.Line2D at 0x7e01ec682090>]
```



Exercício 3: Método subgradiente estocástico 2

Construa uma função `sgd2(f, G, w0, lr, l2, m, it_max)` que toma como entrada as funções `g` `G()`, o ponto inicial `w0`, a regularização `l2` $(= \gamma)$, `m` e implementa o método subgradiente estocástico em `it_max` iterações iniciando de `w0`:

$$w_{k+1} = w_k - \alpha_k G_{i_k}(w_k),$$

com passo $\alpha_k = \frac{lr}{\gamma k}$, onde **na k-ézima iteração, $i_k \in [n]$ é escolhido uniformemente ao acaso, usand o ponto amostral** (x_{i_k}, y_{i_k}). Esta função deve retornar a sequência

$$k \mapsto f_m(w_k) = \frac{1}{m}\sum_{i=1}^{m} \max\{0, 1 - y_i(x_i^\top w_k)\} + \frac{\gamma}{2}\|w_k\|^2.$$

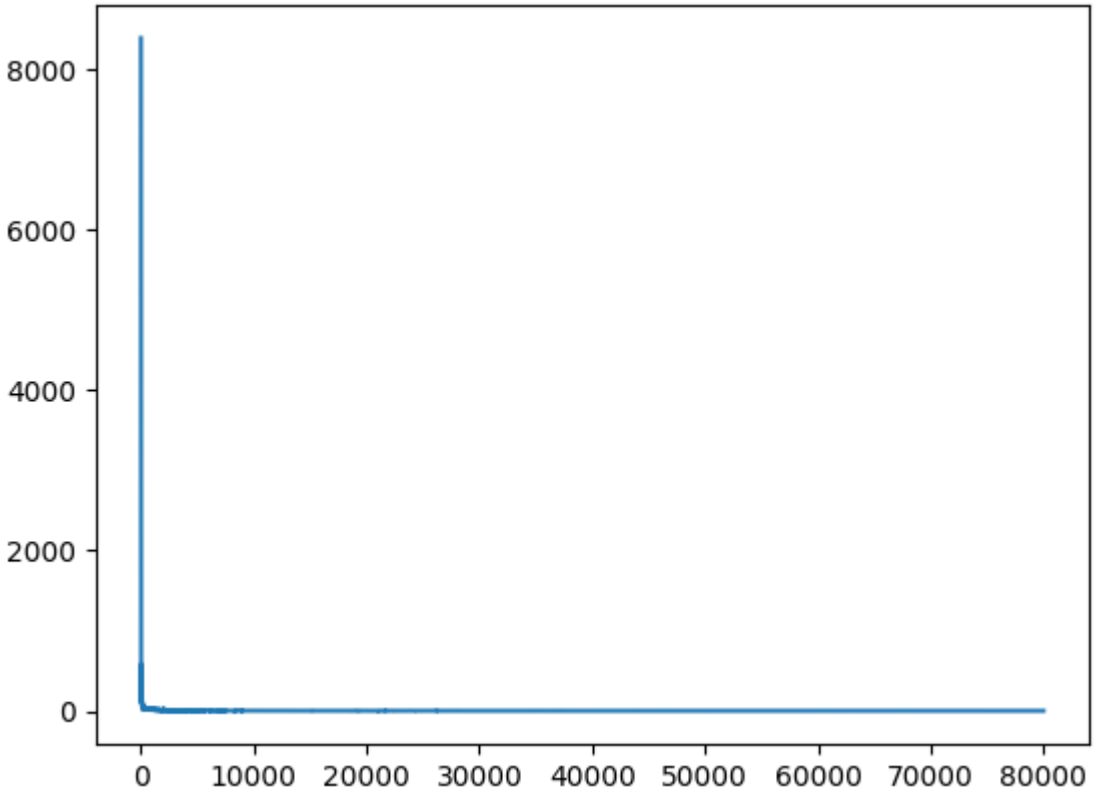
A função também deve retornar o último iterado.

In [9]:
```
#Escreva o código aqui
def sgd2(f, G, w0:np.ndarray, lr:float, l2:float, m:int, it_max:int) -> tuple[list[float], np.ndarray]:
    w = w0.copy()
    f_values = list()

    for k in range(it_max):
        alpha_k = lr / (l2 * (k + 1))
        i_k = np.random.randint(n)

        w -= alpha_k * G(w, i_k, l2)

        f_values.append(float(f(w, l2, m)))

    return f_values, w
```

In [10]:
```
# gradient descent
f2 = sgd2(f, G, w0, 1e-1, l2, m, it_max)
```

In [11]:
```
plt.plot(f2[0])
```

Out[11]:
```
[<matplotlib.lines.Line2D at 0x7e01e97e2650>]
```



Exercício 4: Método subgradiente estocástico 3

Construa agora função `sgd3(f, gB, w0, lr, l2, m, B, it_max)` que toma como entrada as funções `f` `gB()`, o ponto inicial `w0`, a regularização `l2` $(= \gamma)$ e implementa o método subgradiente estocástico com mini-batch size `B` em `it_max` iterações iniciando de `w0`:

$$w_{k+1} = w_k - \alpha_k \cdot \frac{1}{B} \sum_{i \in B_k} G_i(w_k),$$

onde **na k-ézima iteração, $B_k \subset\in [n]$ é escolhido uniformemente ao acaso, usando o mini-batch** $\{(x_i, y_i)\}_{i \in B_k}$. Esta função deve retornar a sequência

$$k \mapsto f_{\text{itmax}}(w_k) = \frac{1}{\text{itmax}} \sum_{i=1}^{\text{itmax}} \max\{0, 1 - y_i(x_i^\top w_k)\} + \frac{\gamma}{2}\|w_k\|^2.$$

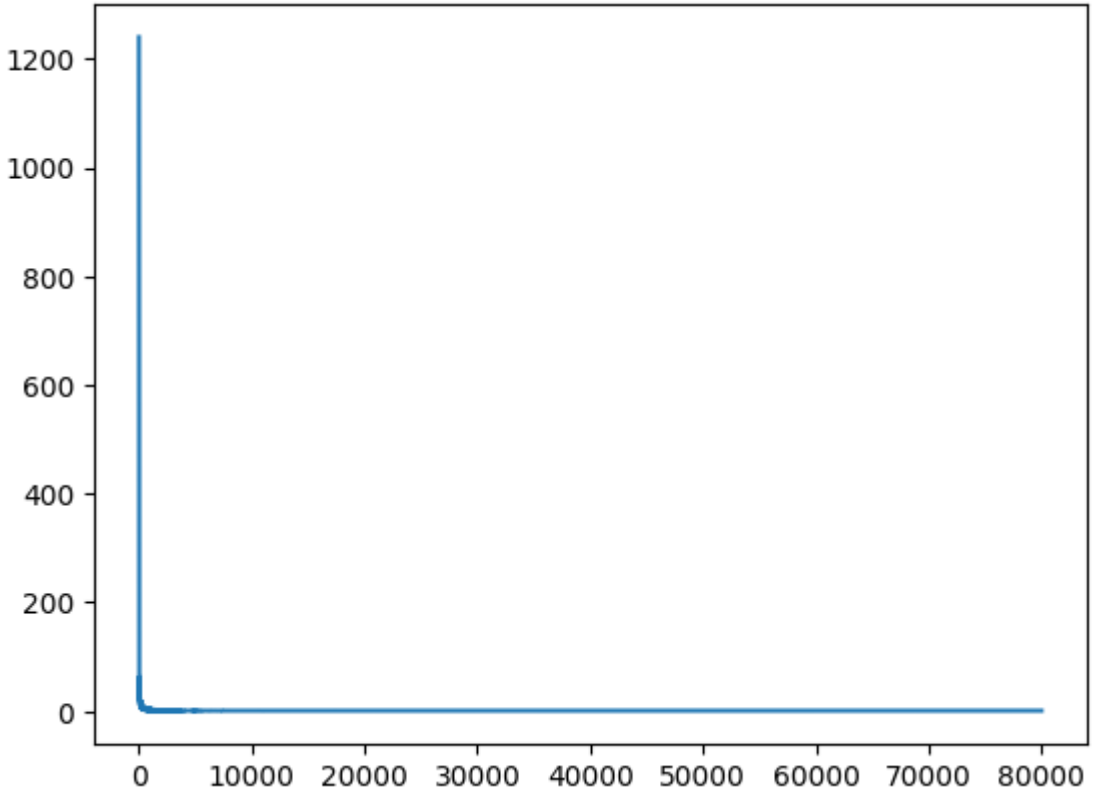
A função também deve retornar o último iterado. Implemente com passo $\alpha_k = \frac{lr}{\gamma k}$.

In [12]:
```python
#Escreva o código aqui
def sgd3(f, gB, w0:np.ndarray, lr:float, l2:float, m:int, B:int, it_max:int) -> tuple[list[float], np.ndarray]:
    w = w0.copy()
    f_values = list()

    for k in range(it_max):
        alpha_k = lr / (l2 * (k + 1))
        w -= alpha_k * gB(w, l2, B)

        f_values.append(float(f(w, l2, m)))

    return f_values, w
```

In [13]:
```python
f3 = sgd3(f, gB, w0, 1e-1, l2, m, B, it_max)
```

In [14]:
```python
plt.plot(f3[0])
```

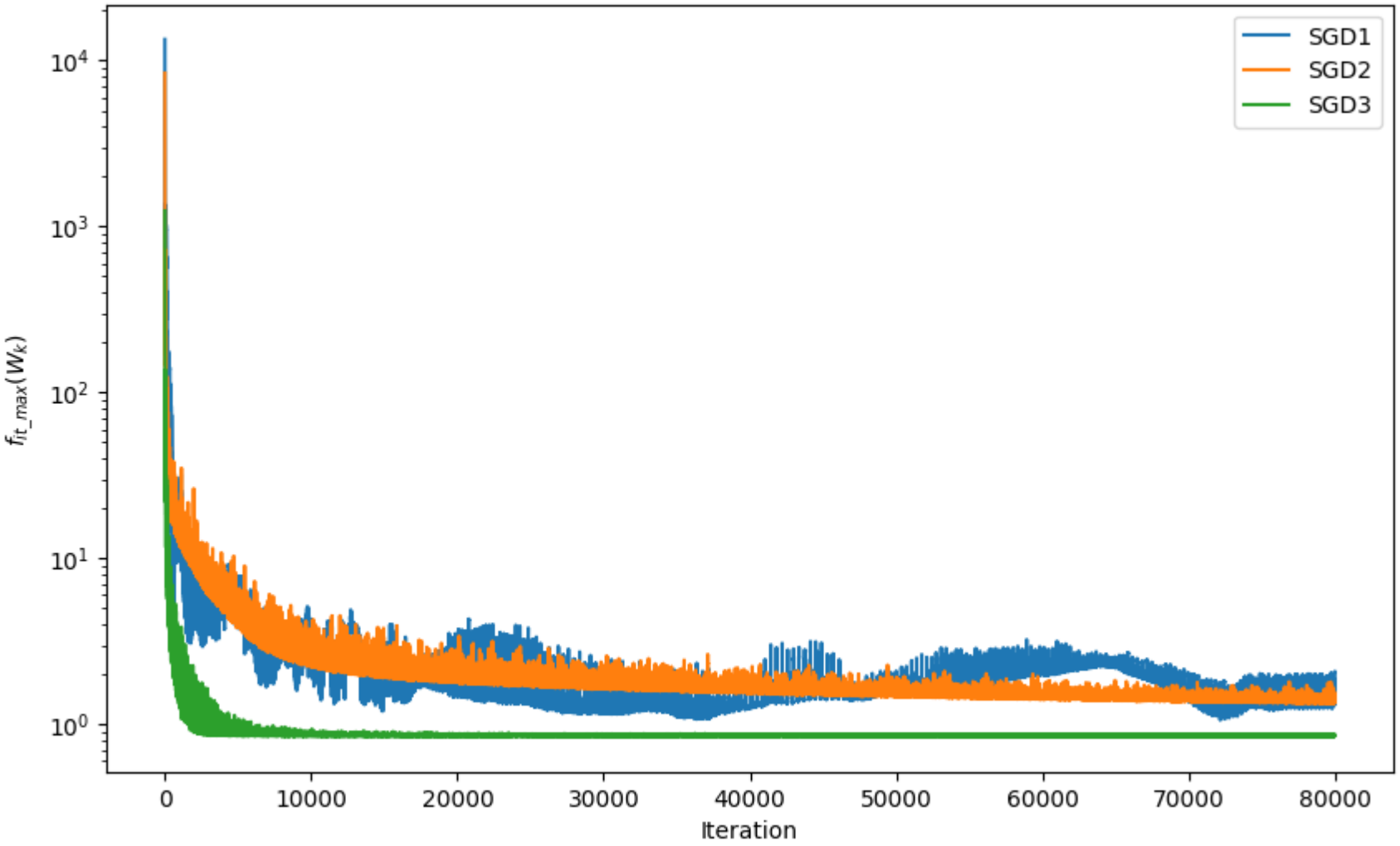
Out[14]:
```
[<matplotlib.lines.Line2D at 0x7e0229f53950>]
```



Exercício 5:

Implemente num mesmo gráfico os erros $f_{\text{itmax}}(w_k)$ de cada método em função no número de iterações.

In [15]:
```
#Escreva o código aqui
plt.figure(figsize=(10, 6))
plt.plot(f1[0], label='SGD1')
plt.plot(f2[0], label='SGD2')
plt.plot(f3[0], label='SGD3')
plt.xlabel('Iteration')
plt.ylabel('$f_{it\_max}(W_k)$')
plt.yscale('log')
plt.legend()
plt.show()
```



Exercício 6:

Nos exercícios anteriores, usamos os m primeiros pontos de dados `y[:m], X[:m]` para plotar a sequência $k \mapsto f_m(w_k)$. Isto não é ideal já que usamos também todo ou parte de `y[:m], X[:m]` para construir a sequência de iterados $k \mapsto w_k$. Refaça os 3 exercícios anteriores mas antes dividindo o data set `y, X` em duas partes `y[n-m:], X[n-m:]` e `y[:n-m], X[:n-m]`. Use o dataset `y[:n-m], X[:n-m]` de tamanho `n-m` para construir a sequência de iterados $k \mapsto w_k$ e `y[n-m:], X[n-m:]` de tamanho `m` para computar $f_m(w)$ para cada iterado w. Plote os 3 gráficos correspondentes e depois o gráfico com os 6 métodos diferentes. Você nota alguma diferença?

In [16]:
```
#Escreva o código aqui
X_train, y_train = X[:n-m], y[:n-m].reshape(-1, 1)
X_val, y_val = X[n-m:], y[n-m:].reshape(-1, 1)
```

```
def fm(w:np.ndarray, l2:float) -> float:
    hinge_loss = np.maximum(0, 1 - y_val * X_val @ w)
    return np.mean(hinge_loss) + (l2 / 2) * la.norm(w) ** 2

def gBm(w:np.ndarray,l2: float, B:int, size:int=n-m) -> np.ndarray:
    indices = np.random.choice(size, B, replace=False)
    subgradients = np.array([G(w, i, l2) for i in indices])
    return np.mean(subgradients, axis=0)
```

In [17]:
```
#Escreva o código aqui
def sgd_m(f, G, w0:np.ndarray, lr:float, l2:float, m:int, it_max:int) -> tuple[list[float], np.ndarray]:
    w = w0.copy()
    f_values = list()

    for k in range(it_max):
        alpha_k = lr / (l2 * (k + 1))
        i = k % (n-m)

        w -= alpha_k * G(w, i, l2)

        f_values.append(float(f(w, l2)))

    return f_values, w

f1m = sgd_m(fm, G, w0, 1e-1, l2, m, it_max)
```

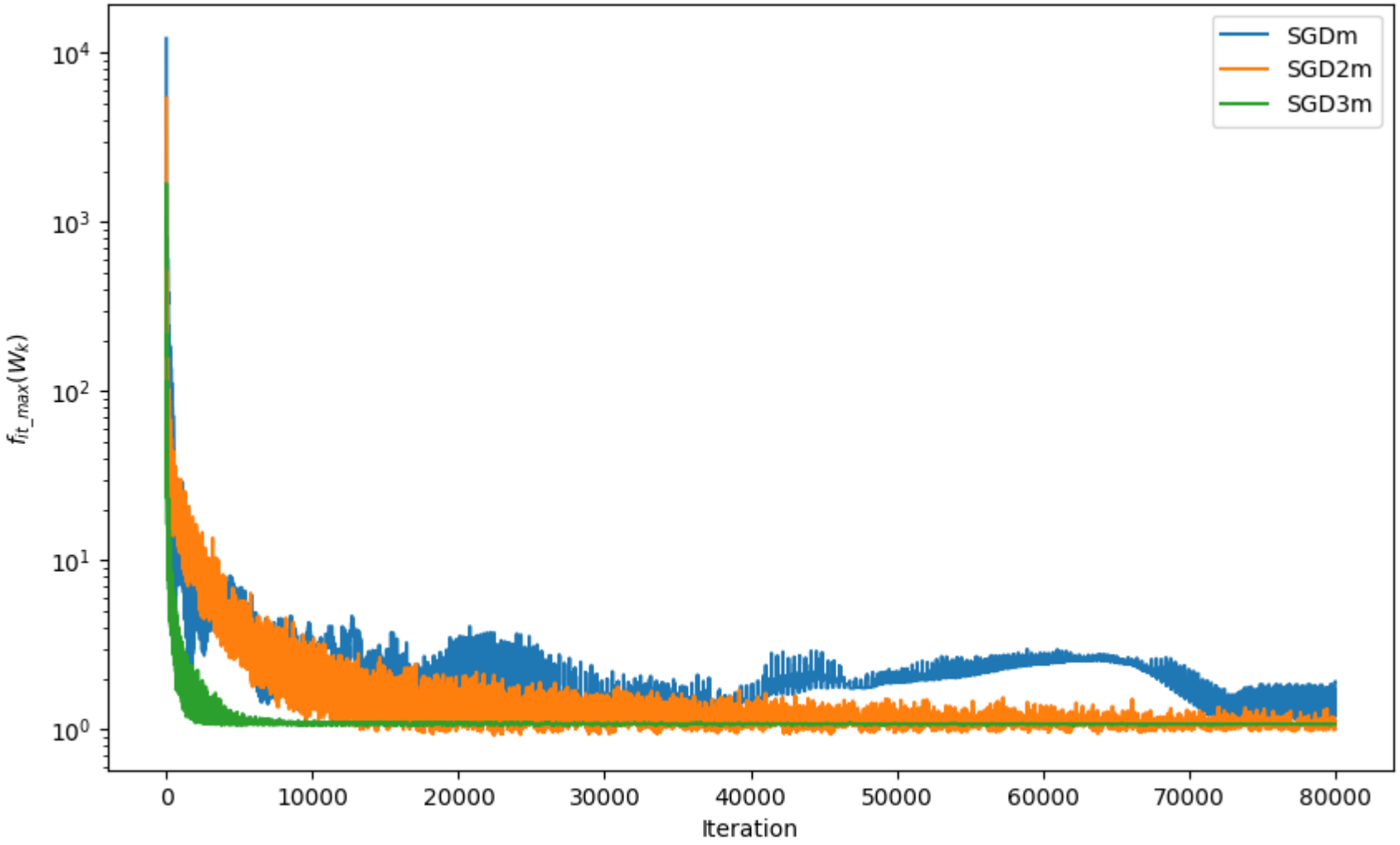
In [18]:
```
#Escreva o código aqui
def sgd2_m(f, G, w0: np.ndarray, lr: float, l2: float, m: int, it_max: int) -> tuple[list[float], np.ndarray]:
    w = w0.copy()
    f_values = list()

    for k in range(it_max):
        alpha_k = lr / (l2 * (k + 1))
        i_k = np.random.randint(n-m)

        w -= alpha_k * G(w, i_k, l2)

        f_values.append(float(f(w, l2)))

    return f_values, w

f2m = sgd2_m(fm, G, w0, 1e-1, l2, m, it_max)
```

In [19]:
```
def sgd3_m(f, gB, w0: np.ndarray, lr: float, l2: float, m: int, B: int, it_max: int) -> tuple[list[float], np.ndarray]:
```

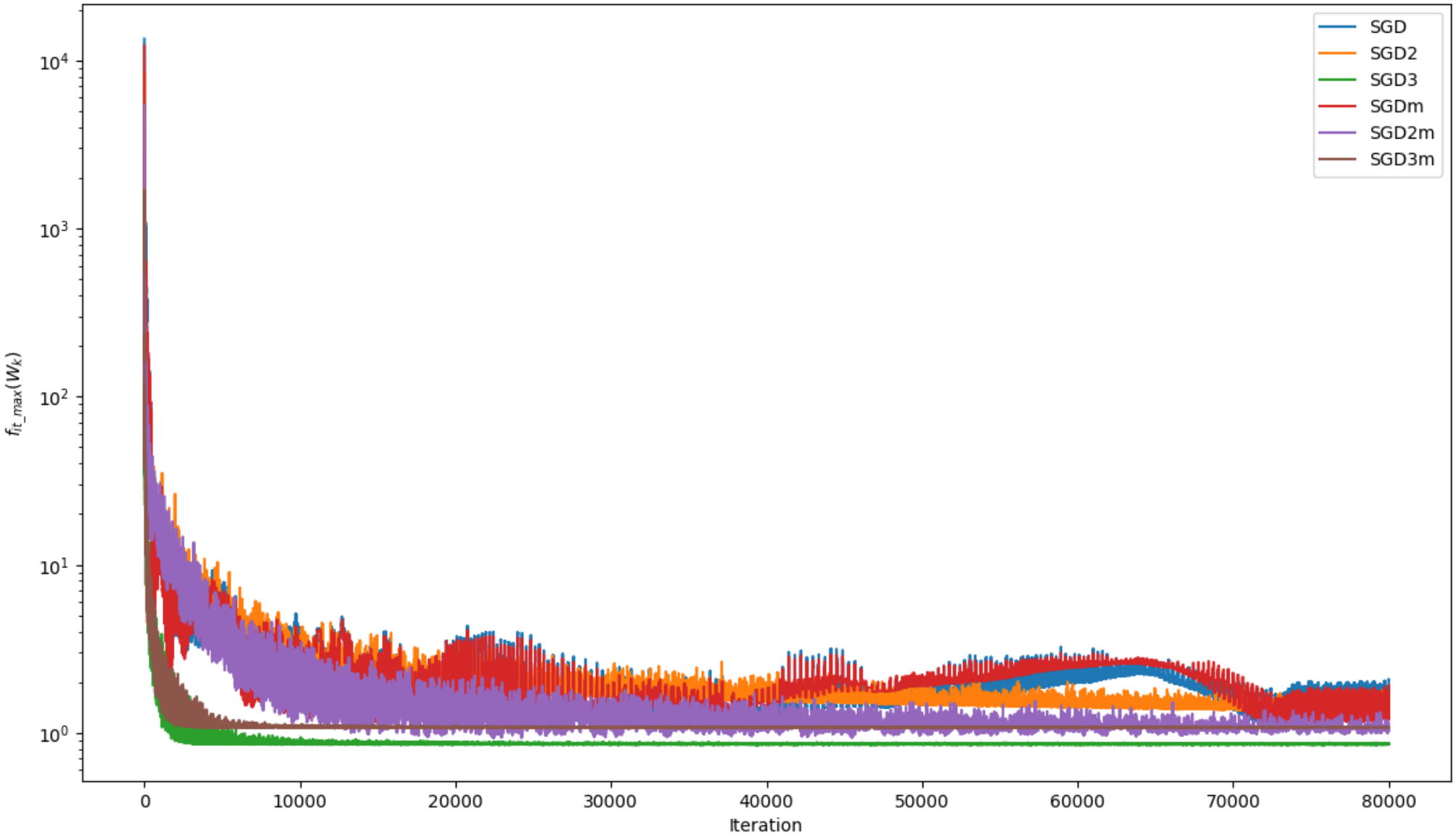
```
    w = w0.copy()
    f_values = list()

    for k in range(it_max):
        alpha_k = lr / (l2 * (k + 1))
        w -= alpha_k * gB(w, l2, B)

        f_values.append(float(f(w, l2)))

    return f_values, w

f3m = sgd3_m(fm, gBm, w0, 1e-1, l2, m, B, it_max)
```

In [20]:

```
#Escreva o código aqui
plt.figure(figsize=(10, 6))
plt.plot(f1m[0], label='SGDm')
plt.plot(f2m[0], label='SGD2m')
plt.plot(f3m[0], label='SGD3m')
plt.xlabel('Iteration')
plt.ylabel('$f_{it\_max}(W_k)$')
plt.yscale('log')
plt.legend()
plt.show()
```



In [21]:
```
#Escreva o código aqui
plt.figure(figsize=(14, 8))
plt.plot(f1[0], label='SGD')
plt.plot(f2[0], label='SGD2')
plt.plot(f3[0], label='SGD3')
plt.plot(f1m[0], label='SGDm')
plt.plot(f2m[0], label='SGD2m')
plt.plot(f3m[0], label='SGD3m')
plt.xlabel('Iteration')
plt.ylabel('$f_{it\_max}(W_k)$')
plt.yscale('log')
plt.legend()
plt.show()
```



Resultados

Os resultados dos métodos foram plotados e comparados em termos do valor funcional $f_{it\_max}(w_k)$ ao longo das iterações. Observamos que as versões modificadas dos métodos (SGDm, SGD2m, SGD3m) apresentam diferenças em relação aos métodos originais, especialmente na convergência e no valor final do funcional. Devido a uma abordagem distinta, usando uma parte dos dados para "treinar" o gradiente, e outra parte não relacionada para "validar" se ele realmente está convergindo, é possível ter maior certeza da generalização do resultado, impedindo que haja um "overfitting", fenômeno conhecido por treinar demais o modelo com dados específicos, levando a uma piora em casos generalizados que não

foram captados no escopo específico, em resumo, separar os dados em duas partes, usando uma para treinar e outra para validar se o treino está funcionando, ajuda a gerar resultados mais seguros que o modelo final possui uma boa generalização.