TP1_Colin_Coerchon

1er Novembre 2024

1 TP 1 - Exercice 3

Élève : Colin Coërchon

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.metrics import precision_score, recall_score, confusion_matrix
import seaborn as sns
```

1.1 Question 1

J'ai essayé de me baser sur les notations vues dans le cours :

- Hypothèse 0 : H espace de Hilbert et $\Theta_c \subset \Theta$ convexe fermé non vide.
- Hypothèse 1 : J convexe et différentiable sur Θ. $\theta^* \in \arg\min_{\theta \in \Theta_c} J(\theta)$
- Hypothèse 2 : On connait g telle que : $\nabla J(\theta) = \mathbb{E}[g(\theta, W)]$

Algorithme du gradient stochastique :

1. **Input** : $\theta_0 \in \Theta_c$, (ε_k) decroit vers 0, la suite de pas
2. **Repeat** : $W_{k+1} \sim \mathbb{P}_W$ une réalisation suivant $\mathbb{P}_W$:

$$\theta^{k+1} = \text{proj}_{\Theta_c}(\theta^k - \varepsilon^k g(\theta^k, W_{k+1}))$$

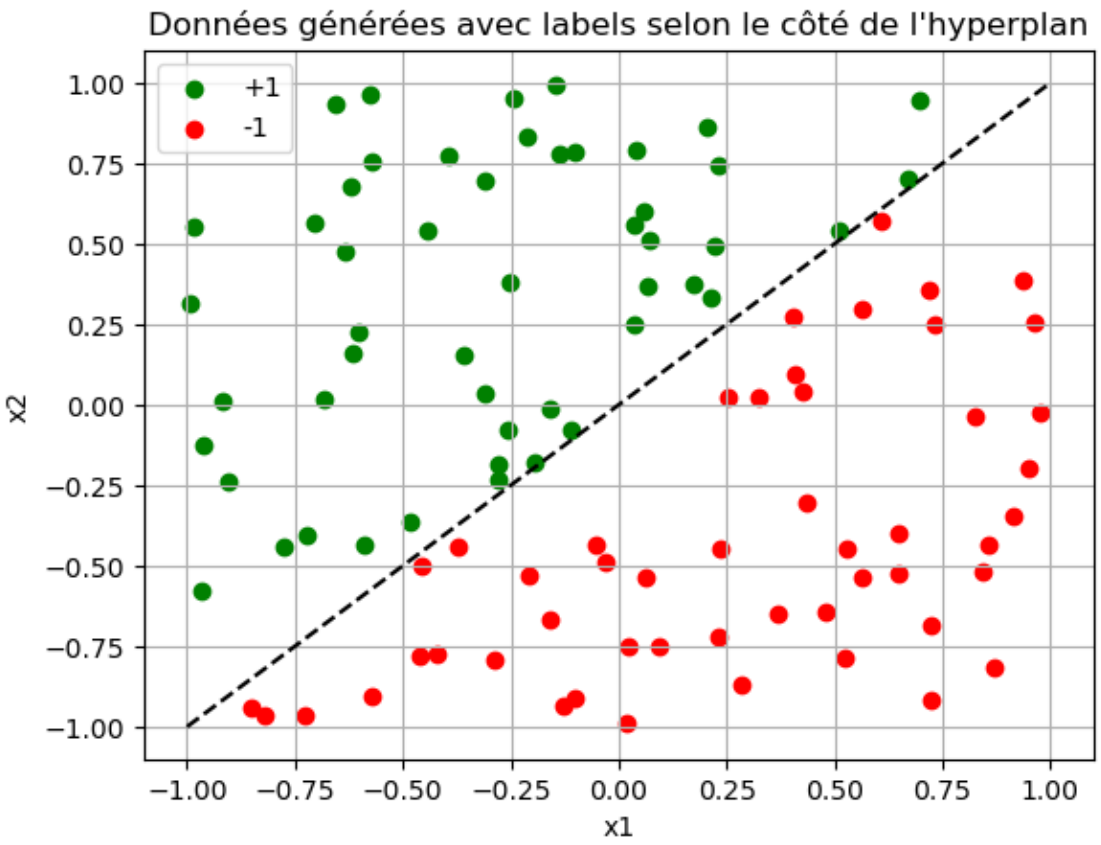
3. **Stop** : critère d'arrêt : $\left|\frac{J(\theta^{k+1}) - J(\theta^k)}{J(\theta^k)}\right| <$ tolérance

[73]:
```
def stochastic_gradient_descent(grad_func, X, y, init_theta,␣
↪learning_rate_schedule, max_iter=1000, tolerance=1e-6):
    n = X.shape[0]
    theta = init_theta
    history = [theta]

    for k in range(max_iter):
        i = np.random.randint(0, n)
        x_i = X[i]
        y_i = y[i]

        grad = grad_func(theta, x_i, y_i)
```

```
        epsilon_k = learning_rate_schedule(k)
        theta_next = theta - epsilon_k * grad

        if np.linalg.norm(theta_next - theta) / (np.linalg.norm(theta) + 1e-8) <␣
↪tolerance:
            break

        theta = theta_next
        history.append(theta)

    return theta, history
```

1.2 Question 2

[74]:
```
def generate_data(n, w_true):
    X = np.random.rand(n, 2) * 2 - 1
    y = np.sign(X @ w_true)
    return X, y

n = 100
w_true = np.array([-1, 1])
X, y = generate_data(n, w_true)


# Visualisation des données avec les deux classes
plt.scatter(X[y == 1][:, 0], X[y == 1][:, 1], color='green', label='+1')
plt.scatter(X[y == -1][:, 0], X[y == -1][:, 1], color='red', label='-1')
x_vals = np.linspace(-1, 1, 100)
y_vals_real = x_vals  # Hyperplan y = x
plt.plot(x_vals, y_vals_real, 'k--')
plt.legend()
plt.xlabel("x1")
plt.ylabel("x2")
plt.title("Données générées avec labels selon le côté de l'hyperplan")
plt.grid()
plt.show()
```

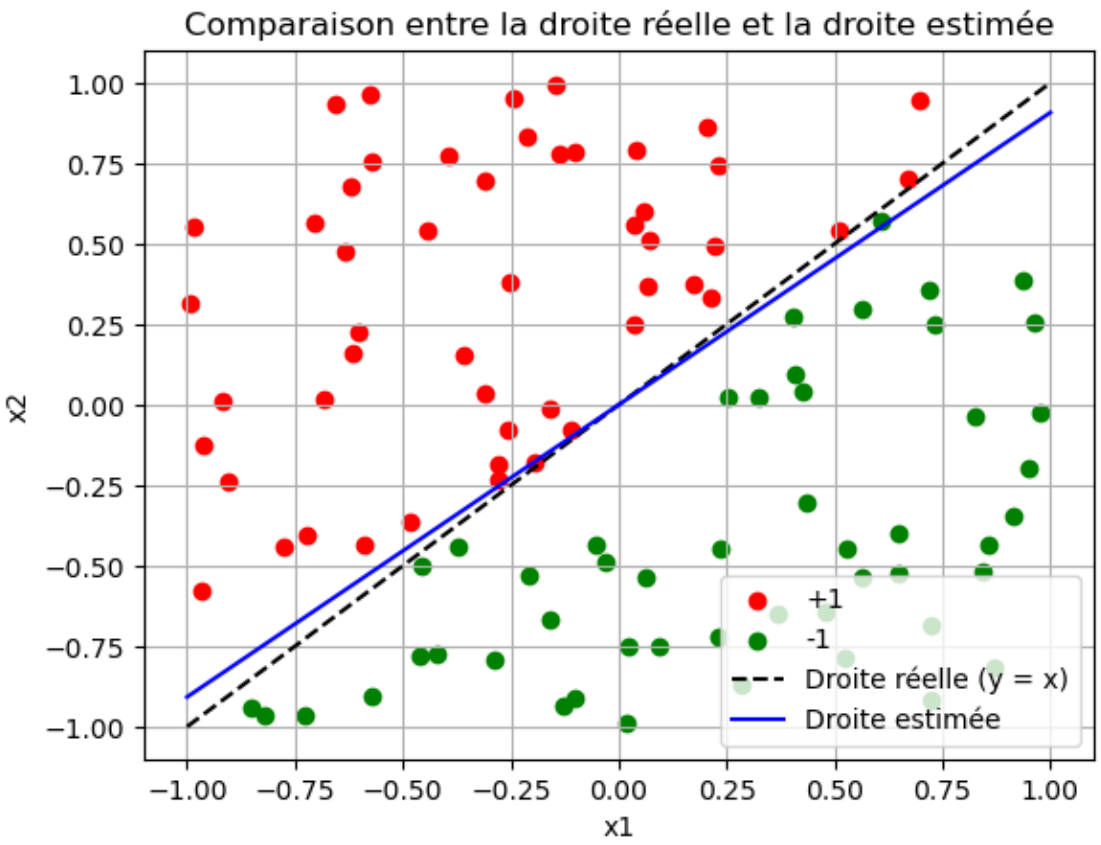


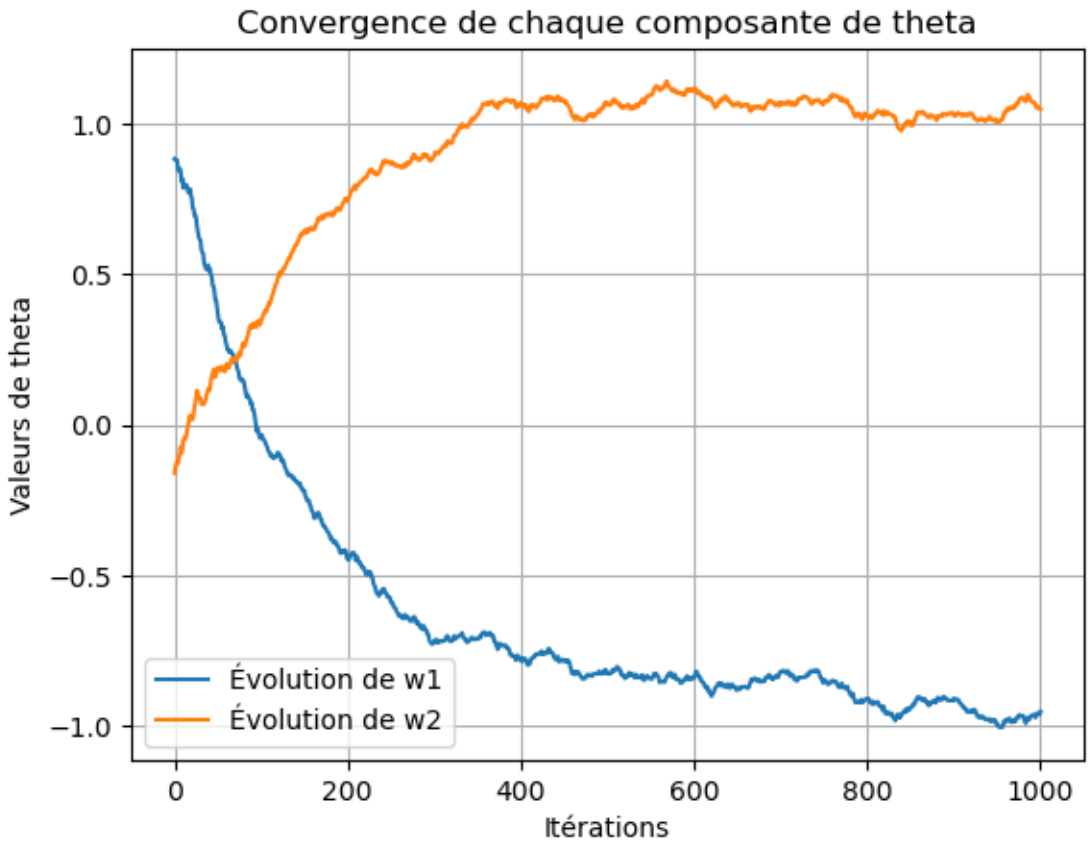
1.3 Question 3

[75]:
```
def risk_gradient(theta, x, y):
    error = y - np.dot(theta, x)
    grad = -2 * error * x
    return grad

def learning_rate_schedule(k):
    return 0.01 / (1 + 0.0001 * k)

init_theta = np.random.uniform(-1, 1, 2)     # Initialisation aléatoire

theta_opt, theta_history = stochastic_gradient_descent(
    grad_func=risk_gradient,
    X=X,
    y=y,
    init_theta=init_theta,
    learning_rate_schedule=learning_rate_schedule
)
```

```
print("Vecteur w* estimé :", theta_opt)
print("Vecteur w̃ réel :", w_true)
print("Distance entre w* et w̃ :", np.linalg.norm(theta_opt - w_true))

plt.scatter(X[y == 1][:, 0], X[y == 1][:, 1], color='red', label='+1')
plt.scatter(X[y == -1][:, 0], X[y == -1][:, 1], color='green', label='-1')

x_vals = np.linspace(-1, 1, 100)
y_vals_real = x_vals
plt.plot(x_vals, y_vals_real, 'k--', label='Droite réelle (y = x)')

if theta_opt[1] != 0:
    slope_estimated = -theta_opt[0] / theta_opt[1]
    y_vals_estimated = slope_estimated * x_vals
    plt.plot(x_vals, y_vals_estimated, 'b-', label='Droite estimée')

plt.xlabel("x1")
plt.ylabel("x2")
plt.legend()
plt.title("Comparaison entre la droite réelle et la droite estimée")
plt.grid()
plt.show()

theta_history = np.array(theta_history)
plt.plot(theta_history[:, 0], label="Évolution de w1")
plt.plot(theta_history[:, 1], label="Évolution de w2")
plt.xlabel("Itérations")
plt.ylabel("Valeurs de theta")
plt.legend()
plt.title("Convergence de chaque composante de theta")
plt.grid()
plt.show()
```

```
Vecteur w* estimé : [-0.95307357  1.04983485]
Vecteur w̃ réel : [-1  1]
Distance entre w* et w̃ : 0.06845145658223643
```


Comparaison entre la droite réelle et la droite estimée
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
x2
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
x1
+1
-1
Droite réelle (y = x)
Droite estimée



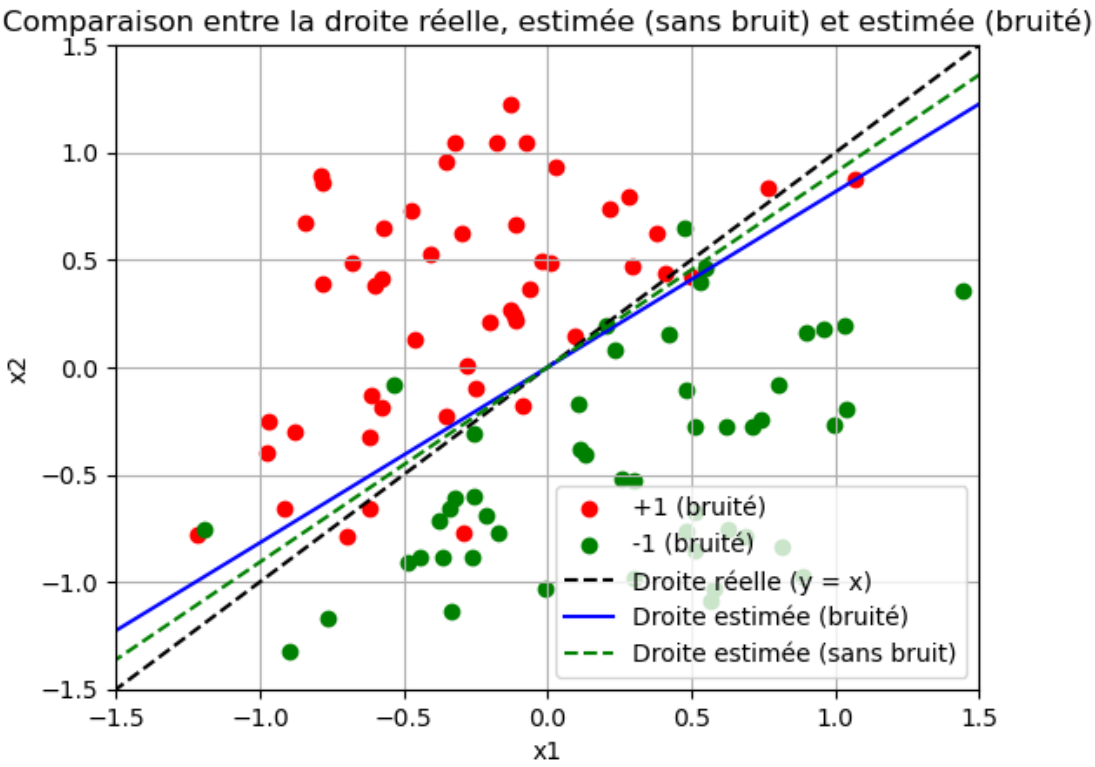
Les résultats sont plutôt bons, même si la convergence met parfois un peu de temps avant de parvenir à des valeurs cohérentes (cela dépend évidemment de l'initialisation de θ_0 qui est aléatoire). On arrive à une valeur de distance entre $w^\star$ et w plutôt bonne.

1.4 Question 4

[76]:
```
# Ajout de bruit gaussien aux données
noise_std = 0.2
X_noisy = X + np.random.normal(0, noise_std, X.shape)

# Réinitialisation de theta pour l'optimisation avec les données bruitées
init_theta_noisy = np.random.uniform(-1, 1, 2)

# Exécution de l'algorithme avec les données bruitées
theta_opt_noisy, theta_history_noisy = stochastic_gradient_descent(
    grad_func=risk_gradient,
    X=X_noisy,
    y=y,
    init_theta=init_theta_noisy,
    learning_rate_schedule=learning_rate_schedule
```

```
)

print("Vecteur w* estimé avec bruit :", theta_opt_noisy)
print("Vecteur w˜ réel :", w_true)
print("Distance entre w* (bruité) et w˜ réel :", np.linalg.norm(theta_opt_noisy␣
 ↪- w_true))

plt.scatter(X_noisy[y == 1][:, 0], X_noisy[y == 1][:, 1], color='red', label='+1␣
 ↪(bruité)')
plt.scatter(X_noisy[y == -1][:, 0], X_noisy[y == -1][:, 1], color='green',␣
 ↪label='-1 (bruité)')

x_vals = np.linspace(-2, 2, 100)
y_vals_real = x_vals
plt.plot(x_vals, y_vals_real, 'k--', label='Droite réelle (y = x)')

if theta_opt_noisy[1] != 0:
    slope_estimated_noisy = -theta_opt_noisy[0] / theta_opt_noisy[1]
    y_vals_estimated_noisy = slope_estimated_noisy * x_vals
    plt.plot(x_vals, y_vals_estimated_noisy, 'b-', label='Droite estimée␣
 ↪(bruité)')

if theta_opt[1] != 0:
    slope_estimated = -theta_opt[0] / theta_opt[1]
    y_vals_estimated = slope_estimated * x_vals
    plt.plot(x_vals, y_vals_estimated, 'g--', label='Droite estimée (sans␣
 ↪bruit)')

plt.xlabel("x1")
plt.ylabel("x2")
plt.xlim(-1.5, 1.5)
plt.ylim(-1.5, 1.5)
plt.legend()
plt.title("Comparaison entre la droite réelle, estimée (sans bruit) et estimée␣
 ↪(bruité)")
plt.grid()
plt.show()
```

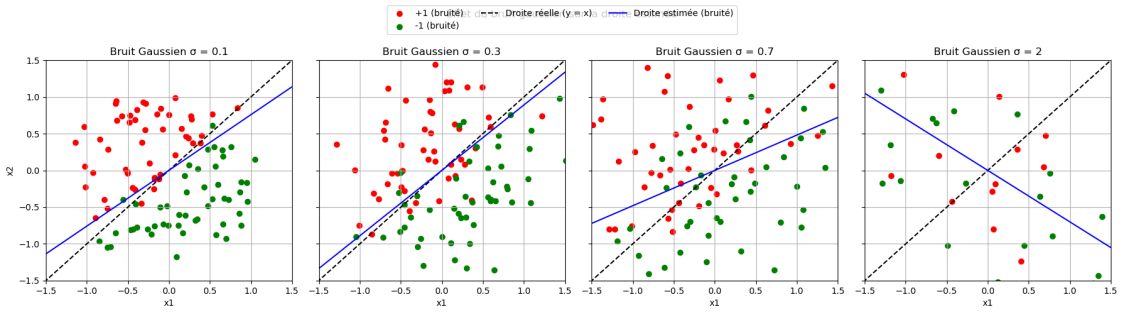
```
Vecteur w* estimé avec bruit : [-0.82063142  1.00382989]
Vecteur w˜ réel : [-1  1]
Distance entre w* (bruité) et w˜ réel : 0.1794094630762116
```



On teste pour différentes valeurs d'écarts type ?

[77]:
```
noise_levels = [0.1, 0.3, 0.7, 2]

fig, axes = plt.subplots(1, len(noise_levels), figsize=(18, 5), sharex=True,␣
↪sharey=True)

for idx, noise_std in enumerate(noise_levels):
    X_noisy = X + np.random.normal(0, noise_std, X.shape)
    init_theta_noisy = np.random.uniform(-1, 1, 2)

    theta_opt_noisy, _ = stochastic_gradient_descent(
        grad_func=risk_gradient,
        X=X_noisy,
        y=y,
        init_theta=init_theta_noisy,
        learning_rate_schedule=learning_rate_schedule
    )

    axes[idx].scatter(X_noisy[y == 1][:, 0], X_noisy[y == 1][:, 1], color='red',␣
↪label='+1 (bruité)')
```

```
    axes[idx].scatter(X_noisy[y == -1][:, 0], X_noisy[y == -1][:, 1],␣
↪color='green', label='-1 (bruité)')

    x_vals = np.linspace(-2, 2, 100)
    y_vals_real = x_vals
    axes[idx].plot(x_vals, y_vals_real, 'k--', label='Droite réelle (y = x)')

    if theta_opt_noisy[1] != 0:
        slope_estimated_noisy = -theta_opt_noisy[0] / theta_opt_noisy[1]
        y_vals_estimated_noisy = slope_estimated_noisy * x_vals
        axes[idx].plot(x_vals, y_vals_estimated_noisy, 'b-', label='Droite␣
↪estimée (bruité)')

    axes[idx].set_title(f"Bruit Gaussien σ = {noise_std}")
    axes[idx].set_xlim(-1.5, 1.5)
    axes[idx].set_ylim(-1.5, 1.5)
    axes[idx].set_xlabel("x1")
    axes[idx].grid()

axes[0].set_ylabel("x2")
handles, labels = axes[0].get_legend_handles_labels()
fig.legend(handles, labels, loc="upper center", ncol=3)
fig.suptitle("Effet du bruit gaussien sur la droite estimée")
plt.tight_layout(rect=[0, 0, 1, 0.95])
plt.show()
```



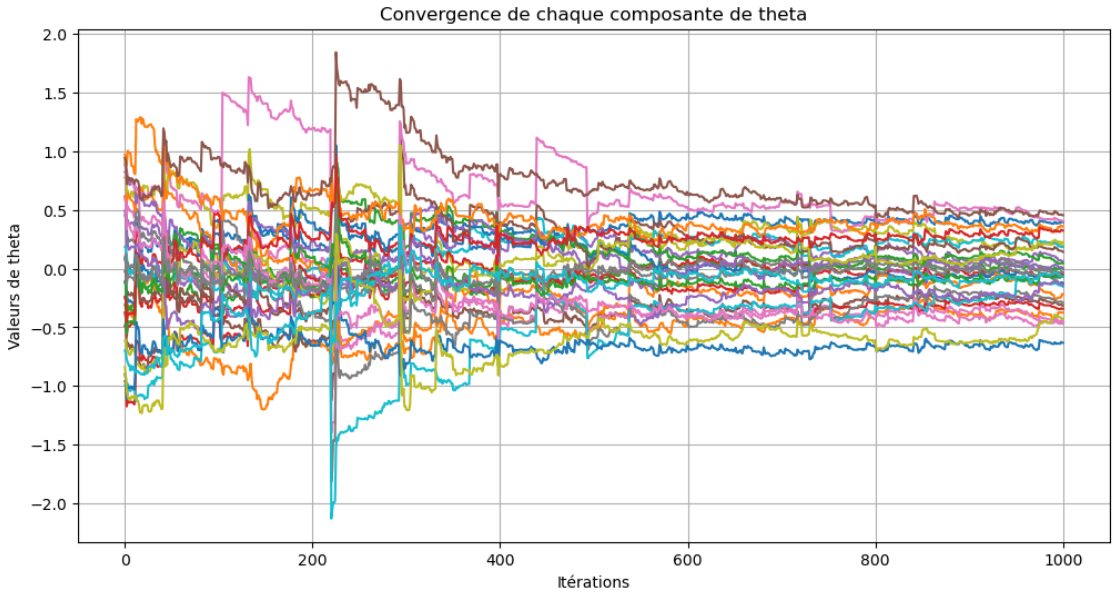
Les résultats sont étonnament plutôt bons, du moment que le bruit n'est pas trop élévé. Jusqu'à $\sigma = 1$, les valeurs prédites $w^\star$ sont tout à fait pertinentes.

1.5 Question 5

[78]:
```
# Chargement des données
data = pd.read_csv(r'wdbc.data', header=None)

# Prétraitement des données
```

```python
data[1] = data[1].map({'M': -1, 'B': 1})  # M -> -1, B -> +1
y = data[1].values  # La colonne de labels
X = data.iloc[:, 2:].values  # Les colonnes de données sans ID ni label

# Normalisation des données
X = (X - np.mean(X, axis=0)) / np.std(X, axis=0)

init_theta = np.random.uniform(-1, 1, X.shape[1])

def risk_gradient(theta, x, y):
    error = y - np.dot(theta, x)
    grad = -2 * error * x
    return grad

def learning_rate_schedule(k):
    return 0.01 / (1 + 0.0001 * k)

theta_opt, theta_history = stochastic_gradient_descent(
    grad_func=risk_gradient,
    X=X,
    y=y,
    init_theta=init_theta,
    learning_rate_schedule=learning_rate_schedule
)

print("Vecteur w* estimé :", theta_opt)
print("Nombre d'itérations :", len(theta_history))
print("Valeur finale du risque empirique :", np.mean([(y[i] - np.dot(theta_opt,␣
 ↪X[i])) ** 2 for i in range(len(y))]))

theta_history = np.array(theta_history)
plt.figure(figsize=(12, 6))
for i in range(X.shape[1]):
    plt.plot(theta_history[:, i], label=f"Évolution de w{i+1}")
plt.xlabel("Itérations")
plt.ylabel("Valeurs de theta")
plt.title("Convergence de chaque composante de theta")
plt.grid()
plt.show()
```

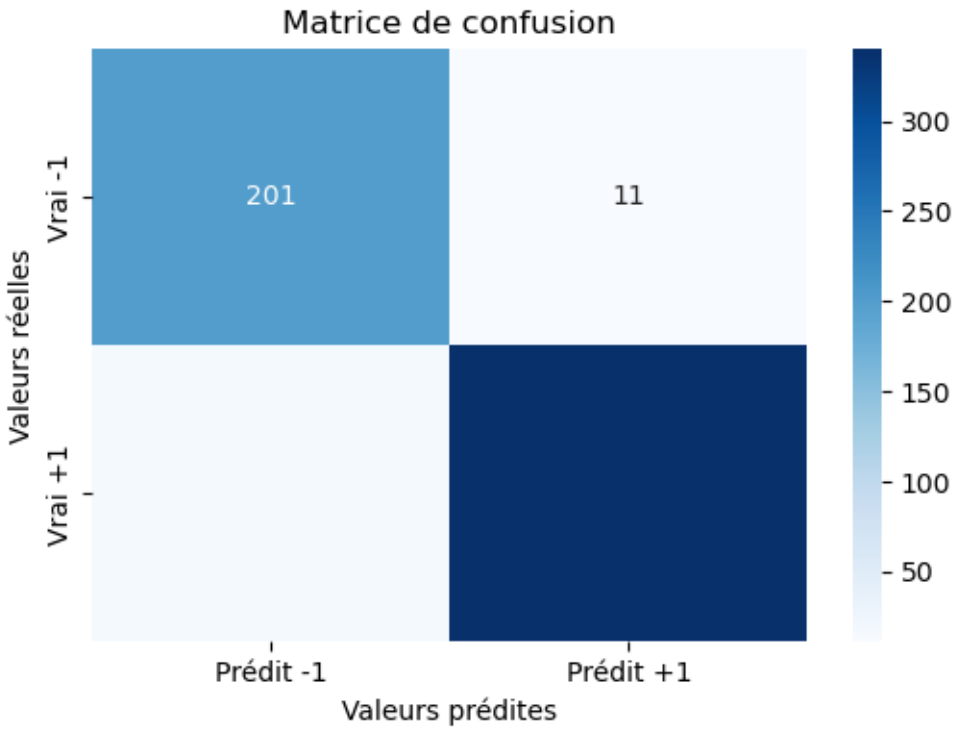
```
Vecteur w* estimé : [ 0.39488361 -0.37576228  0.00503893 -0.05238832  0.05993955
-0.31556966
 -0.44997762 -0.03494324  0.19402095  0.18268566 -0.12311868 -0.21994772
  0.04331271 -0.34347453 -0.00556477  0.14267558  0.40279905 -0.29183357
  0.22295009 -0.15374864 -0.63250594  0.35991443 -0.07134829  0.32294198
 -0.14008277  0.45466412 -0.47001657 -0.03238521 -0.42769137 -0.06908425]
Nombre d'itérations : 1001
Valeur finale du risque empirique : 0.5887622135662421
```



Nous allons maintenant évaluer la pertinence de ces résultats en effectuant, comme cela est présenté dans le petit texte, une estimation de la **précision** et du **rappel** de notre modèle.

[79]:
```
y_pred = np.sign(X @ theta_opt)

# Calcul de la précision et du rappel
precision = precision_score(y, y_pred)
recall = recall_score(y, y_pred)
print("Précision :", precision)
print("Rappel :", recall)

# Calcul et affichage de la matrice de confusion
conf_matrix = confusion_matrix(y, y_pred)
plt.figure(figsize=(6, 4))
sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="Blues", xticklabels=['Prédit␣
↪-1', 'Prédit +1'], yticklabels=['Vrai -1', 'Vrai +1'])
plt.xlabel("Valeurs prédites")
plt.ylabel("Valeurs réelles")
plt.title("Matrice de confusion")
plt.show()
```

```
Précision : 0.9686609686609686
Rappel : 0.9523809523809523
```



Les résultats sont incroyables, alors que l'algorithmes de descente de gradient stochastique n'est pas très complexe. On arrive rapidement à **une précision et un rappel au dessus de 0.9**, ce qui est extrêmement satisfaisant pour un algorithme de classification.