UNIVERSITE SULTAN MOULAY SLIMANE
ECOLE NATIONALE DES SCIENCES APPLIQUEES
KHOURIBGA

Master Big Data et Aide à la Décision

Realisé par

MARZAQ KHALID

- Recherche Opérationnelle 2 -
TP 1 : Régression non-linéaire

Année Universitaire : 2023-2024

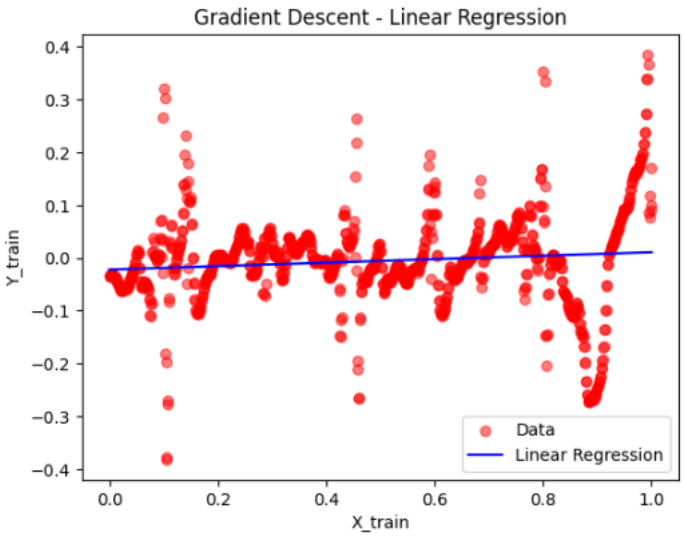
Partie 1

Régression linéaire

L'entraînement de notre modèle a été réalisé avec la méthode de descente de gradient, impliquant 1000 itérations et un taux d'apprentissage de 0,01. Cette stratégie a été déterminante pour optimiser progressivement les paramètres, aboutissant à un modèle affiné et performant.

Code

```
[122] import numpy as np
      import matplotlib.pyplot as plt

      # Ajouter une colonne de biais à X_train
      X_train_b = np.c_[np.ones((1000, 1)), X_train]

      # Définir les hyperparamètres
      learning_rate = 0.01
      n_iterations = 1000
      theta = np.random.randn(2, 1)

      # Fonction de coût
      def compute_cost(X, Y, theta):
          m = len(Y)
          predictions = X.dot(theta)
          cost = (1/(2*m)) * np.sum(np.square(predictions - Y))
          return cost

      # Descente de gradient
      for iteration in range(n_iterations):
          gradients = (1/len(Y_train)) * X_train_b.T.dot(X_train_b.dot(theta) - Y_train)
          theta = theta - learning_rate * gradients
          cost = compute_cost(X_train_b, Y_train, theta)

      # Visualisation des résultats
      plt.scatter(X_train, Y_train, alpha=0.5, color='red',label='Data')
      plt.plot(X_train, X_train_b.dot(theta), color='blue', label='Linear Regression')
      plt.title('Gradient Descent - Linear Regression')
      plt.xlabel('X_train')
      plt.ylabel('Y_train')
      plt.legend()
      plt.show()
```


Gradient Descent - Linear Regression
Y_train
X_train
Data
Linear Regression
0.4
0.3
0.2
0.1
0.0
−0.1
−0.2
−0.3
−0.4
0.0
0.2
0.4
0.6
0.8
1.0

Partie 2

Régression non-linéaire

2.1 Préparation des Données

Nous avons téléchargé l'ensemble de données **EMG** et l'avons divisé en 1000 échantillons pour l'entraînement et 100 échantillons pour les tests.

Code

```
[1]  import pandas as pd
     import numpy as np
     import math

     dt = pd.read_csv("EMG.csv")
```

```
[4]  X_train = dt["X"].head(1000)
     Y_train = np.array(dt["Y"].head(1000)).reshape(1000,1)

     X_test = dt["X"].tail(100)
     Y_test = np.array(dt["Y"].tail(100)).reshape(100, 1)
```

	X	Y
0	0.00025	-0.0333
1	0.00050	-0.0350
2	0.00075	-0.0350
3	0.00100	-0.0300
4	0.00125	-0.0300
...	...	...
50855	12.71400	0.0050
50856	12.71420	0.0067
50857	12.71450	0.0050
50858	12.71480	0.0050
50859	12.71500	0.0083

50860 rows × 2 columns

Nous avons normalisé les ensembles de données d'entraînement et de test en utilisant la méthode de normalisation min-max. Cette approche consiste à mettre à l'échelle les valeurs de chaque caractéristique de manière à ce qu'elles se situent dans la plage [0, 1], en utilisant la formule $X_{\text{norm}} = \frac{X - X_{\min}}{X_{\max} - X_{\min}}$. Cette normalisation uniformise les échelles des caractéristiques, favorisant ainsi une convergence plus rapide et une meilleure performance du modèle lors de l'entraînement.

Code

```
[7]  def normaliz_data(X):
       min_x = np.min(X)
       max_x = np.max(X)
       return (X-min_x)/(max_x-min_x)
```

```
[8]  X_train = normaliz_data(X_train)
     X_test = normaliz_data(X_test)
```

- Préparation de la matrice K_σ

Nous avons préparé les matrices K_σ nécessaires à l'algorithme en utilisant les données d'entraînement. Ces matrices représentent les noyaux (*kernels*) calculés à l'aide d'une fonction de

similarité, telle que la fonction exponentielle du noyau gaussien. Chaque élément $K_\sigma(x_i, x_j)$ de la matrice K_σ correspond à la similarité entre les exemples d'entraînement x_i et x_j après application de la fonction de noyau.

Code

```
[9]  def k(x, y, sigma):
         return np.exp(-np.square(x - y) / (2 * sigma**2))
```

```
[10] def matrice_k_alpha(x, sigma):
         x = np.array(x)
         length = len(x)
         matrice = np.zeros((length, length))

         for i in range(length):
             matrice[i, :] = k(x[i], x, sigma)

         return matrice
```

```
[11] grand_k = matrice_k_alpha(X_train,0.001)
```

2.2 Choix des Paramètres

Les paramètres optimaux de notre modèle ont été choisis via une validation croisée, évaluant ainsi différentes combinaisons pour minimiser l'erreur de prédiction.

Code

```
[60] import numpy as np

     def calcule_des_f(alpha, X):
         myList = []
         for i in X.values:
             myList.append(f(alpha, X.values, i))
         return myList

     def f(alpha, X, pred):
         s = 0
         for i in range(len(X)):
             s += alpha[i] * k(pred, X[i], 0.001)
         return s
```

```
[61] import numpy as np
     from sklearn.model_selection import KFold
     from sklearn.metrics import mean_squared_error
     from sklearn.model_selection import GridSearchCV
     from itertools import product

     liste_lam = [0.1, 0.5]
     liste_lr = [0.001, 0.01]

     param_grid = {'lam': liste_lam, 'lr': liste_lr}

     kf = KFold(n_splits=5, shuffle=True, random_state=42)

     performances_par_parametres = []

     for train_index, val_index in kf.split(X_train):
         X_train_fold, X_val_fold = X_train[train_index], X_train[val_index]
         Y_train_fold, Y_val_fold = Y_train[train_index], Y_train[val_index]

         alpha = np.zeros((len(X_train_fold), 1))

         combinaisons_parametres = product(liste_lam, liste_lr)
         for lam, lr in combinaisons_parametres:
             print(f"lam : {lam}, lr:{lr}")
             grand_k = matrice_k_alpha(X_train_fold, 0.001)

             alpha = np.zeros((len(X_train_fold), 1))

             i = 0
             while True:
                 alpha = alpha - lr * gradJ(grand_k, alpha, Y_train_fold, lam)
                 i += 1
                 if i >= 2000:
                     break

             mse = mean_squared_error(Y_val_fold, calcule_des_f(alpha,X_val_fold))
             performances_par_parametres.append({'mse': mse, 'lam': lam, 'lr': lr})
```

```
[62] performances_par_parametres
```

```
[{'mse': 0.010171763733598042, 'lam': 0.1, 'lr': 0.001},
 {'mse': 0.013585669804857918, 'lam': 0.1, 'lr': 0.01},
 {'mse': 0.009267617099194212, 'lam': 0.5, 'lr': 0.001},
 {'mse': 0.01166099489595817, 'lam': 0.5, 'lr': 0.01},
 {'mse': 0.00978797219222877, 'lam': 0.1, 'lr': 0.001},
 {'mse': 0.013323753712641497, 'lam': 0.1, 'lr': 0.01},
 {'mse': 0.008903841231945302, 'lam': 0.5, 'lr': 0.001},
 {'mse': 0.011320882155551935, 'lam': 0.5, 'lr': 0.01},
 {'mse': 0.008199291958195099, 'lam': 0.1, 'lr': 0.001},
 {'mse': 0.013475227090536808, 'lam': 0.1, 'lr': 0.01},
 {'mse': 0.007286337942646148, 'lam': 0.5, 'lr': 0.001},
 {'mse': 0.010796559148567555, 'lam': 0.5, 'lr': 0.01},
 {'mse': 0.010346288477443202, 'lam': 0.1, 'lr': 0.001},
 {'mse': 0.014574593256406199, 'lam': 0.1, 'lr': 0.01},
 {'mse': 0.00933579291003929, 'lam': 0.5, 'lr': 0.001},
 {'mse': 0.011919754908237571, 'lam': 0.5, 'lr': 0.01},
 {'mse': 0.011123317452206937, 'lam': 0.1, 'lr': 0.001},
 {'mse': 0.011633761378777618, 'lam': 0.1, 'lr': 0.01},
 {'mse': 0.010366231046669748, 'lam': 0.5, 'lr': 0.001},
 {'mse': 0.01072234465492337, 'lam': 0.5, 'lr': 0.01}]
```

Suite à l'analyse des résultats, les paramètres optimaux sélectionnés pour notre modèle sont un taux d'apprentissage (lr) de 0.001 et un paramètre de régularisation (lambda) de 0.5. Ces choix sont le fruit de notre approche de validation croisée.

2.3 Application de l'Algorithme Descente de gradient

Algorithm 1: Algorithme de descente de gradient [3]

- $\alpha_0, \lambda, \text{tol}, y, \{x_i \text{ pour } i = 1, \ldots, n\}, \eta_0$ et σ;
  - Calculer K_σ;
  - Itérer $\alpha_{t+1} = \alpha_t - \eta_t \nabla_\alpha J(\alpha_t)$, jusqu'à ce que $J(\alpha_{t+1}) \leq \text{tol}$ (avec η_t calculé par la méthode d'Armijo);
- La solution restaurée α.

Nous avons entraîné notre modèle sur 2000 itérations en utilisant les paramètres optimaux, soit un taux d'apprentissage (lr) de 0.001 et un paramètre de régularisation ($lambda$) de 0.5. Cette procédure d'entraînement a été mise en œuvre afin d'optimiser les performances du modèle et d'ajuster ses poids pour une meilleure adéquation aux données d'entraînement.

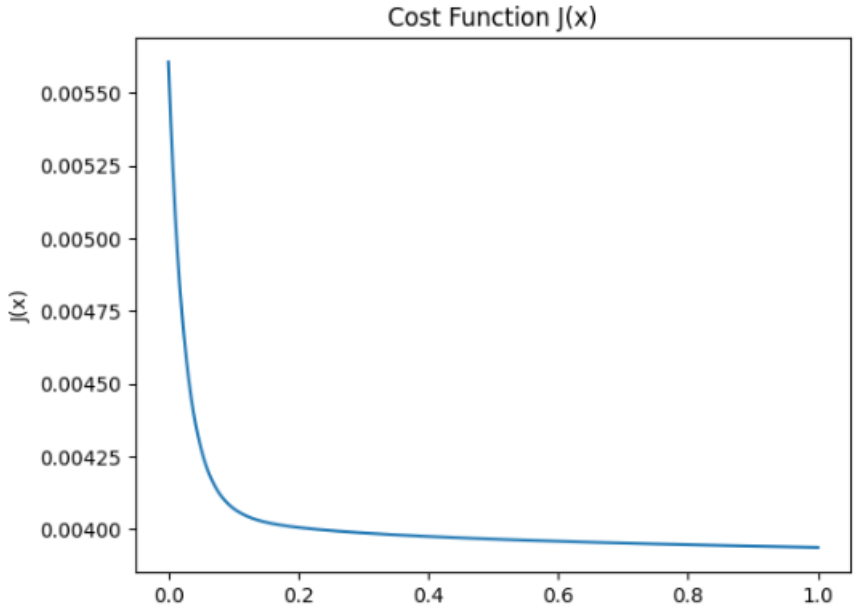
Code

```
[63] def gradJ(K, alpha, y, lmbd):
         # Calculate the gradient of J
         term1 = 2 * np.dot(K.T, np.dot(K, alpha) - y)
         term2 = 2 * lmbd * np.dot(K.T, np.dot(K, alpha))
         gradient = term1 + term2

         return gradient
```

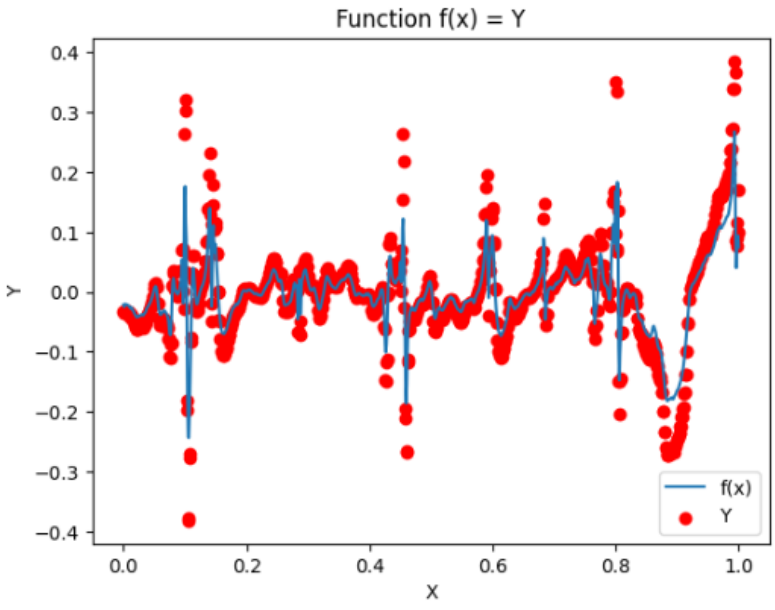
```
[64] def costFunction(alphai,k,y,l):
       yhat=np.dot(k,alphai)
       J=np.linalg.norm(yhat-y,ord=2)*2+l*np.linalg.norm(yhat,ord=2)*2
       return J/len(y)
```

```
[71] i=0
     alpha = np.zeros((1000,1))
     lam = 0.5
     lr = 0.001
     cost_list = []
     grand_k = matrice_k_alpha(X_train, 0.001)
     while True:
       alpha = alpha - lr*gradJ(grand_k,alpha,Y_train,lam)
       loss = costFunction(alpha,grand_k,Y_train,lam)
       cost_list.append(loss)
       i+=1
       if i>=2000 or loss < 0.0001:
         break
```

```python
import numpy as np
import matplotlib.pyplot as plt


# Generate a range of x values
x_values = np.linspace(0, 1, 2000)


# Plot the cost function
plt.plot(x_values, cost_list)
plt.title('Cost Function J(x)')
plt.xlabel('Parameter Vector x')
plt.ylabel('J(x)')
plt.show()
```




```python
[81] plt.plot(X_train, calcule_des_f(alpha,X_train), label='f(x)')
     plt.scatter(X_train, Y_train, color='red', label='Y')
     plt.title('Function f(x) = Y')
     plt.xlabel('X')
     plt.ylabel('Y')
     plt.legend()
     plt.show()
```



2.4 Comparaison de Gradient descent avec Adam et SGD

2.4.1 Adaptive Moment Estimation

Optimiseur Adam (Adaptive Moment Estimation) [2] est un algorithme d'optimisation très populaire utilisé dans l'entraînement de réseaux neuronaux, en particulier dans le domaine de l'apprentissage profond. Il combine des techniques d'optimisation stochastique avec des mises à jour adaptatives des taux d'apprentissage.

Algorithm 2: Algorithme Adam

Data: Paramètres initiaux : α (taux d'apprentissage), β_1, β_2 (coefficients d'ajustement exponentiel), ϵ (stabilité numérique)

Data: Paramètres à optimiser : w (poids du modèle)

Data: Données : mini-lot de données, itérations t

Initialisation : $m \leftarrow 0$, $v \leftarrow 0$, $t \leftarrow 0$;

while *critère d'arrêt non atteint* **do**

- $t \leftarrow t + 1$;
- Calculer le gradient g_t sur un mini-lot de données;
- **Moments du gradient** : $m_t \leftarrow \beta_1 \cdot m + (1 - \beta_1) \cdot g_t$;
- **Moments du carré du gradient** : $v_t \leftarrow \beta_2 \cdot v + (1 - \beta_2) \cdot (g_t)^2$;
- **Correction de biais** : $\hat{m}_t \leftarrow \frac{m_t}{1-\beta_1^t}$;
- $\hat{v}_t \leftarrow \frac{v_t}{1-\beta_2^t}$;
- **Mise à jour des poids** : $w \leftarrow w - \alpha \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t}+\epsilon}$;

Sortie : Séquence de mises à jour des poids w

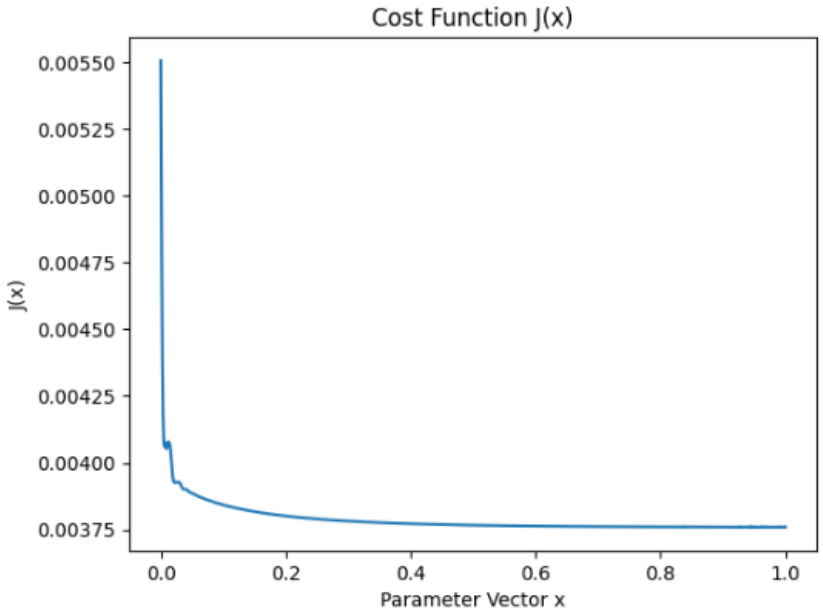
2.4.1.1 Entrainement du modéle

Nous avons entraîné notre modèle sur un ensemble de données comprenant 1000 échantillons pendant 2000 itérations. Lors de cet entraînement, nous avons choisi un taux d'apprentissage α de 0.001 et un paramètre de régularisation λ de 0.5.

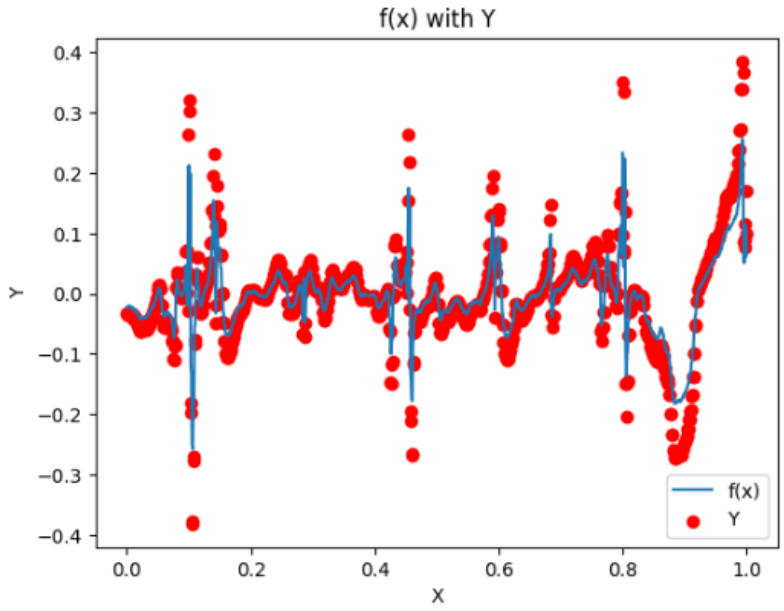
Code

```
[21] import numpy as np

     def adam(x, y, alphai, Ksigma, l, eps, nu, beta1=0.9, beta2=0.999):
         J = []
         vdw = np.zeros(alphai.shape)
         sdw = np.zeros(alphai.shape)
         gJ = gradJ(Ksigma, alphai, y, l)

         for i in range(2000):
             vdw = beta1 * vdw + (1 - beta1) * gJ
             sdw = beta2 * sdw + (1 - beta2) * gJ**2
             vdwCorr = vdw / (1 - beta1**(i+1))
             sdwCorr = sdw / (1 - beta2**(i+1))
             alphai = alphai - nu * (vdwCorr / (np.sqrt(sdwCorr) + eps))

             gJ = gradJ(Ksigma, alphai, y, l)

             J.append(costFunction(alphai,Ksigma,y,l))

         return alphai, J
```

```
import numpy as np
import matplotlib.pyplot as plt

alpha = np.zeros((1000,1))
# Generate a range of x values
x_values = np.linspace(0, 1, 2000)

alpha,j = adam(x=X_train,y=Y_train,alphai=alpha,Ksigma=grand_k,l=lam,eps=0.001,nu=0.01)
# Plot the cost function
plt.plot(x_values, j)
plt.title('Cost Function J(x)')
plt.xlabel('Parameter Vector x')
plt.ylabel('J(x)')
plt.show()
```



```
[26] plt.plot(X_train, calcule_des_f(alpha,X_train), label='f(x)')
     plt.scatter(X_train, Y_train, color='red', label='Y')
     plt.title('f(x) with Y')
     plt.xlabel('X')
     plt.ylabel('Y')
     plt.legend()
     plt.show()
```



2.4.2 Descente de Gradient Stochastique (SGD)

Descente de Gradient Stochastique (SGD) [1] est un algorithme d'optimisation largement utilisé dans le domaine de l'apprentissage automatique et de l'apprentissage profond. Contrairement à la descente de gradient classique, qui met à jour les poids du modèle en fonction de la moyenne des gradients calculés sur l'ensemble de l'ensemble de données, la SGD effectue des mises à jour de poids après chaque échantillon (ou mini-lot) individuel.

Algorithm 3: Descente de Gradient Stochastique (SGD)

Data: Paramètres initiaux : α (taux d'apprentissage)
Data: Paramètres à optimiser : w (poids du modèle)
Data: Données : ensemble d'entraînement, nombre d'itérations T

Initialisation : $w \leftarrow$ valeurs initiales;
for $t \leftarrow 1$ **to** T **do**
    Choisir un échantillon (ou mini-lot) aléatoire du jeu d'entraînement;
    Calculer le gradient g_t de la fonction de perte par rapport à w pour cet échantillon;
    Mise à jour des poids : $w \leftarrow w - \alpha \cdot g_t$;
Sortie : Poids du modèle mis à jour w

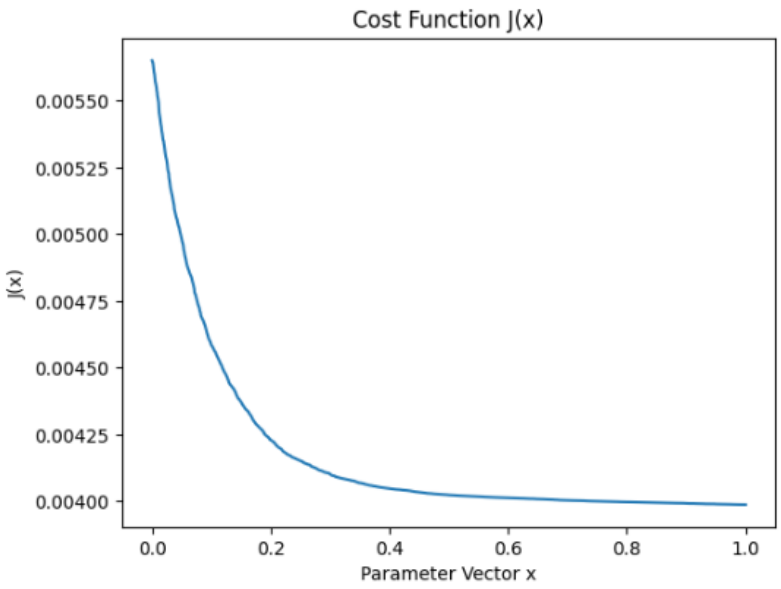
2.4.2.1 Entrainement du modéle

Nous avons entraîné notre modèle sur un ensemble de données comprenant 1000 échantillons pendant 2000 itérations. Lors de cet entraînement, nous avons choisi un taux d'apprentissage α de 0.001 et un paramètre de régularisation λ de 0.5.

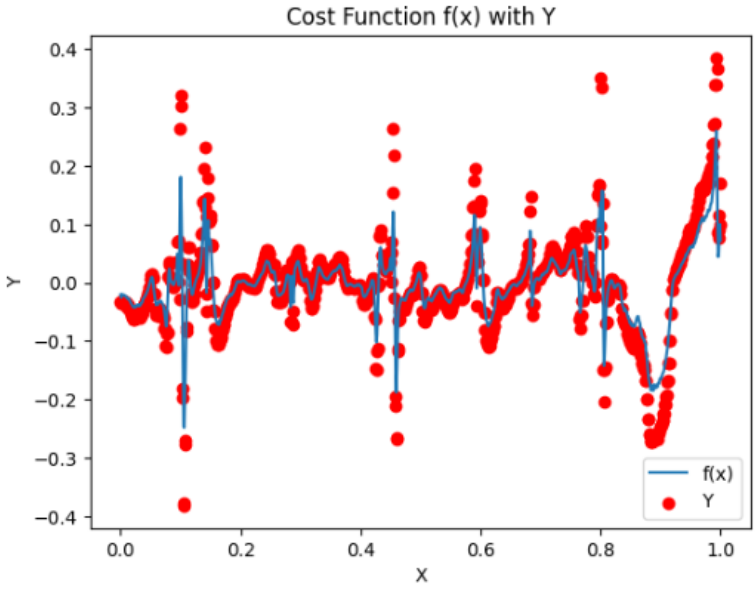
Code

```
[31] def sgd(x,y,alphai,Ksigma,l,eps,nu):
         J=[]
         gJ = gradJ(Ksigma, alphai, y, l)
         for i in range(2000):
             a=[np.random.randint(0,x.shape[0]) for _ in range(32)]
             alphai[a]-=nu*gJ[a]
             gJ=gJ = gradJ(Ksigma, alphai, y, l)
             J.append(costFunction(alphai,Ksigma,y,l))
         return alphai,J
```

```
[33] import numpy as np
     import matplotlib.pyplot as plt

     alpha = np.zeros((1000,1))

     # Generate a range of x values (assuming x is a 1D vector)
     x_values = np.linspace(0, 1, 2000)
     alpha,j_sgd = sgd(x=X_train,y=Y_train,alphai=alpha,Ksigma=grand_k,l=lam,eps=0.001,nu=0.01)

     # Plot the cost function
     plt.plot(x_values, j_sgd)
     plt.title('Cost Function J(x)')
     plt.xlabel('Parameter Vector x')
     plt.ylabel('J(x)')
     plt.show()
```



```
plt.plot(X_train, calcule_des_f(alpha,X_train), label='f(x)')
plt.scatter(X_train, Y_train, color='red', label='Y')
plt.title('f(x) with Y')
plt.xlabel('X')
plt.ylabel('Y')
plt.legend()
plt.show()
```



2.4.3 Évaluation des performances

Nous avons réalisé des évaluations de performance en utilisant différentes méthodes d'optimisation sur un ensemble de test composé de 100 échantillons. Les résultats obtenus, exprimés en termes de Root Mean Squared Error (RMSE) et Mean Squared Error (MSE), sont les suivants :

```
[59] import numpy as np

     def rmse(y_real, y_pred):
         return np.sqrt(np.mean((y_real - y_pred)**2))
```

```
[60] def mse(y_real, y_pred):
         return np.mean((y_real - y_pred)**2)
```

```
[61] print(f"RMSE pour Gradient descent : {rmse(calcule_des_f(alpha,X_test),Y_test)}")
     print(f"RMSE pour Adam : {rmse(calcule_des_f(alphadam,X_test),Y_test)}")
     print(f"RMSE pour Stochastique Gradient descent : {rmse(calcule_des_f(alphasgd,X_test),Y_test)}")

     RMSE pour Gradient descent : 0.020296079820172053
     RMSE pour Adam : 0.050432251324413285
     RMSE pour Stochastique Gradient descent : 0.012964982580067933
```

```
[62] print(f"MSE pour Gradient descent : {mse(calcule_des_f(alpha,X_test),Y_test)}")
     print(f"MSE pour Adam : {mse(calcule_des_f(alphadam,X_test),Y_test)}")
     print(f"MSE pour Stochastique Gradient descent : {mse(calcule_des_f(alphasgd,X_test),Y_test)}")

     MSE pour Gradient descent : 0.0004119308560667952
     MSE pour Adam : 0.0025434119736487855
     MSE pour Stochastique Gradient descent : 0.00016809077330146496
```

Ces résultats témoignent des performances différentes des algorithmes d'optimisation. La Descente de Gradient Stochastique (SGD) semble avoir la meilleure performance en termes de RMSE et MSE, suivie par la Descente de Gradient classique, et Adam ayant une performance légèrement inférieure.

En analysant ces résultats, nous pouvons conclure que la SGD a démontré une efficacité particulière sur cet ensemble de test spécifique. Toutefois, il est essentiel de noter que la performance peut varier en fonction des caractéristiques spécifiques des données et des paramètres d'entraînement. Cette analyse permet d'orienter le choix de la méthode d'optimisation en fonction des besoins spécifiques du problème.

Bibliographie

[1] L. Bottou. Stochastic gradient descent tricks. In *Neural Networks : Tricks of the Trade : Second Edition*, pages 421–436. Springer, 2012.

[2] D. P. Kingma and J. Ba. Adam : A method for stochastic optimization. *arXiv preprint arXiv :1412.6980*, 2014.

[3] S. Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv :1609.04747*, 2016.