

Machine Learning TP4: Régression linéaire régularisé

Melvin DUBEE - Tanguy ROUDAUT

FIPASE 24

28 octobre 2023

1 Régression linéaire régularisée

Nous avons déjà appliqué la régression linéaire dans le *TP1*, le principe sera le même, mais cette fois-ci nous allons appliquer le principe de régularisation rencontré dans le *TP2* pour la régression logistique. L'objectif est de prédire la quantité d'eau qui s'écoule d'un barrage à l'aide de la variation du niveau d'eau dans un réservoir.

Pour rappel, l'hypothèse dans le cas d'une régression linéaire est la suivante :

$$h_{\theta}(x) = x^T \theta = \theta_0 + \theta_1 x_1 \quad (1)$$

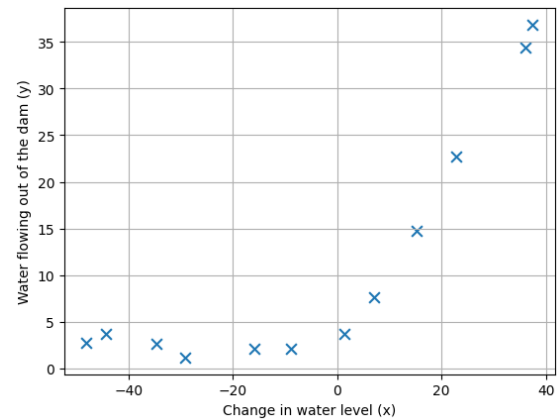


FIGURE 1 – Visualisation du jeu de données

1.1 Fonction de coût $J(\theta)$ régularisée

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})^2 + \underbrace{\frac{\lambda}{2m} \left[\sum_{j=1}^n \theta_j^2 \right]}_{(a)} \quad (2)$$

- (a) La régularisation nous permet d'atténuer tous les coefficients θ en fonction du paramètre λ . Plus celui-ci est petit, plus θ sera atténué.

```
1 def linearRegCostFunction(X, y, theta, Lambda):
2     m,n = X.shape
3     theta = theta.reshape((n,1))
4     h = X @ theta
5     J = (1/(2*m)) * np.sum((h - y) ** 2) + (Lambda/(2*m)) * np.sum(theta[1:] ** 2)
6
7     return J.flatten()
8
9 """ output
10 Cost at theta = [1 1]: 303.993192
11 (this value should be about 303.993192)
12 """
```

Listing 1 – Fonction linearRegCostFunction

1.2 Descente de gradient régularisée

$$\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{m} \sum_{i=0}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x_j^{(i)} \quad \text{pour } j = 0 \quad (3)$$

$$\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{m} \left[\sum_{i=0}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x_j^{(i)} + \underbrace{\lambda \theta_j}_{(a)} \right] \quad \text{pour } j \geq 1$$

(a) Ici, on adapte également la régularisation sur la descente de gradient comme pour le coût $J(\theta)$. Conventionnellement on ne régularise pas θ_0 puisqu'il s'agit du biais.

Le calcul du coût $J(\theta)$ permet de mesurer la qualité de la prédiction, si le coût est faible alors notre prédiction est proche des valeurs réelles et inversement si le coût est important.

On remarque ici avec l'équation 2 qui utilise l'équation 1, que les seules valeurs qui puissent influencer notre coût est θ et λ . Pour réduire notre coût, nous devons donc minimiser θ , c'est l'objectif de la descente de gradient, minimiser $J(\theta)$.

```

1 def linearRegCostFunction(X, y, theta, Lambda):
2     m,n = X.shape # number of training examples
3     theta = theta.reshape((n,1)) # in case where theta is a vector (n,)
4     h = X @ theta
5     J = (1/(2*m)) * np.sum((h - y) ** 2) + (Lambda/(2*m)) * np.sum(theta[1:] ** 2)
6     grad = (1/m) * (X.T @ (h - y))
7     grad[1:] += (Lambda/m * theta[1:])
8
9     return J.flatten(), grad.flatten()
10 """ output
11 Gradient at theta = [1 1]: [-15.303016 598.250744]
12 (this value should be about [-15.303016 598.250744])
13 """

```

Listing 2 – Fonction linearRegCostFunction

1.3 Visualisation de la régression linéaire régularisée

À l'aide de la fonction `trainLinearReg` qui utilise elle-même notre fonction 2 nous pouvons calculer les valeurs optimales de θ .

On constate que notre régression n'est pas optimale, on peut même dire que nous sommes en sous-apprentissage.

Le problème ne vient pas de λ étant donné qu'il vaut 0, la régularisation n'a donc aucune utilité. Si nous obtenons un résultat similaire c'est dû aux nombres insuffisants de caractéristiques, notre θ est seulement de dimension 2. Pour résoudre ce problème, nous pouvons modifier nos caractéristiques sous forme polynomiale, cette méthode sera appliquée plus tard.

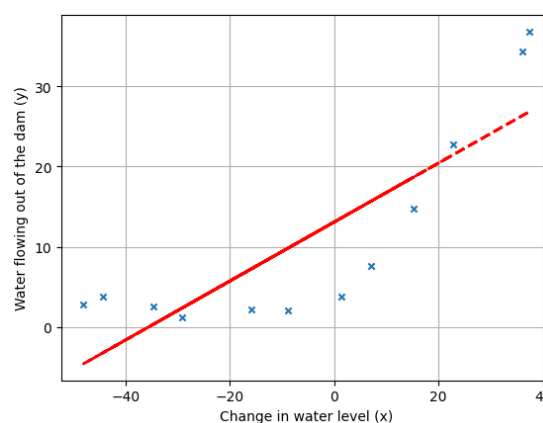


FIGURE 2 – Régression linéaire régularisée

2 Bias et Variance

La notion entre bias et variance est importante, puisqu'il nous permet de prendre connaissance de la performance de notre système. Si on est en sous-apprentissage alors le bias est élevé, dans le cas d'un surapprentissage c'est la variance qui est élevée. De plus, à l'aide de ces notions il est possible de déterminer la performance de notre système et de déduire le paramètre de régularisation optimal λ cf 3.2.

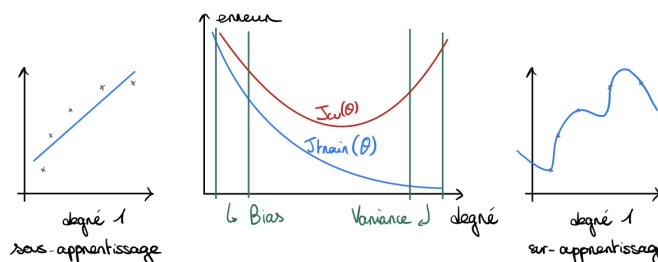


FIGURE 3 – Bias-Variance impacte

2.1 Courbe d'apprentissage

La courbe d'apprentissage consiste à tracer l'erreur d'apprentissage et de validation en fonction de l'ensemble de données.

Pour ce faire, nous avons besoin de calculer ces erreurs à l'aide du jeu de données et du jeu de validation. La fonction de coût nous permet d'obtenir l'erreur sur la prédiction, nous pouvons réutiliser cette fonction pour chaque nouvel échantillon.

```

1 def learningCurve(X, y, Xval, yval, Lambda):
2     m, _ = X.shape
3     error_train = np.zeros((m, 1))
4     error_val = np.zeros((m, 1))
5
6     for i in range(m):
7         theta = trainLinearReg(X[:i+1,:], y[:i+1], Lambda)
8         error_train[i, _] = linearRegCostFunction(X[:i+1,:], y[:i+1], theta, 0)
9         error_val[i, _] = linearRegCostFunction(Xval, yval, theta, 0)
10
11     return error_train, error_val
12
13 """ output
14 Training Examples  Train Error validation Error
15 0      0.000000    205.121096
16 1      0.000000    110.300407
17 2      3.286595    45.010229
18 3      2.842678    48.368911
19 4      13.154049   35.865164
20 5      19.443963   33.829961
21 6      20.098522   31.970985
22 7      18.172859   30.862446
23 8      22.609405   31.135996
24 9      23.261462   28.936206
25 10     24.317250   29.551431
26 11     22.373906   29.433816
27 """

```

Listing 3 – Fonction learningCurve

Les erreurs obtenues nous permettent de tracer la courbe d'apprentissage.

Nous pouvons observer que le biais est élevé, la courbe d'entraînement et de validation converge vers une erreur d'environ 25. Ce qui confirme notre hypothèse émise à la section 1.3, notre modèle est en sous-apprentissage dû aux faibles nombres de caractéristiques.

Cette application nous permet également de comprendre l'importance de l'erreur de validation. Avec l'ensemble du jeu de données de validation, nous pouvons affiner nos paramètres pour optimiser nos prédictions.

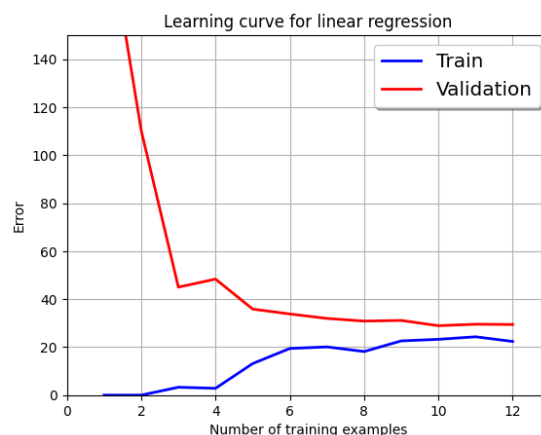


FIGURE 4 – Courbe d'apprentissage

3 Régression polynomiale

Nous avons précédemment démontré à plusieurs reprises que notre système comporte un nombre insuffisant de caractéristiques, notre biais est élevé (*sous-apprentissage*).

À l'aide de la fonction 4 nous pouvons augmenter le nombre d'entités de X , c'est-à-dire les données du niveau de l'eau. L'objectif est de réaliser un nouveau vecteur contenant X allant de la puissance 1 à p , chaque colonne représente une puissance différente.

```
1 def polyFeatures(X, p):
2     X_poly = np.zeros((X.shape[0], p))
3     for i in range(1, p+1):
4         X_poly[:, i-1] = X[:, 0]**i
5     return X_poly
```

Listing 4 – Fonction polyFeatures

3.1 Apprentissage de la régression polynomiale

Finalement, nos données deviennent des termes polynomiaux. Dans ce cas, le principe de régression est le même, nous avons simplement augmenté le nombre de caractéristiques pour éviter le sous-apprentissage. Cependant, élever à la puissance nos termes peut donner des valeurs différentes et non régulières, nous allons donc normaliser nos données avant de réaliser la régression linéaire et afficher nos courbes d'apprentissage.

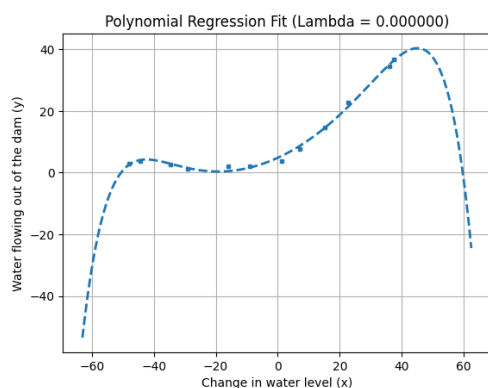


FIGURE 5 – Régression polynomiale $\lambda = 0$

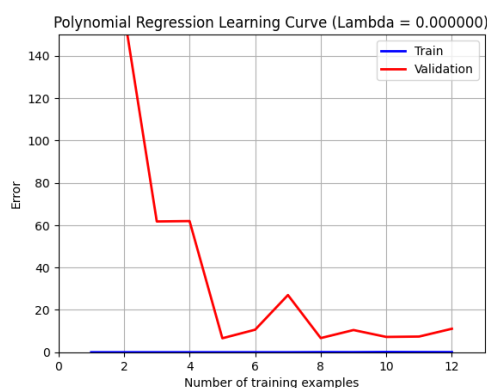


FIGURE 6 – Courbe d'apprentissage $\lambda = 0$

Dans ce premier cas, nous n'avons pas appliqué de régulation, comme pour la partie 1, $\lambda = 0$. Avec un θ de dimension 2, nous avons constaté un sous-apprentissage, ici c'est l'inverse. On le constate dans un premier temps avec la figure 5, où notre régression linéaire est particulièrement ajustée à notre ensemble de données. Par la suite la figure 6, nous montre que notre erreur d'entraînement est très faible. La présence d'une variance élevée entre l'erreur d'entraînement et de validation nous suggère que nous sommes en surapprentissage.

3.2 Réglage du paramètre de régularisation λ

Maintenant que nous sommes en surapprentissage avec $\lambda = 0$, nous allons utiliser notre régulation mise en œuvre dans la partie 1 pour différentes valeurs grâce au code suivant.

```
1 for Lambda in [0, 1, 100]:
2     theta = trainLinearReg(X_poly, y, Lambda, maxiter=10)
3     error_train, error_val = learningCurve(X_poly, y, X_poly_val, yval, Lambda)
4     # Plot training data and fit & learning curves (Error vs Number of training examples)
5     # code donné...
```

Listing 5 – Régularisation pour différent λ

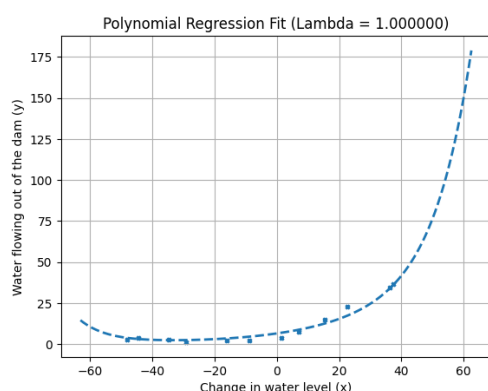


FIGURE 7 – Régression polynomiale $\lambda = 1$

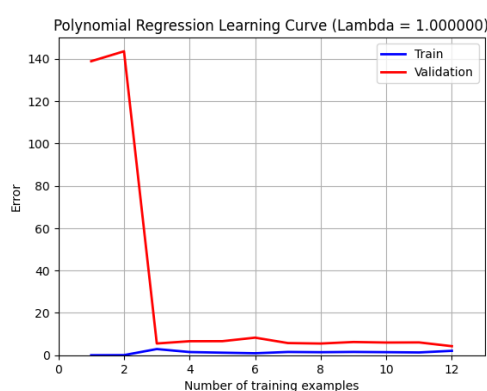


FIGURE 8 – Courbe d'apprentissage $\lambda = 1$

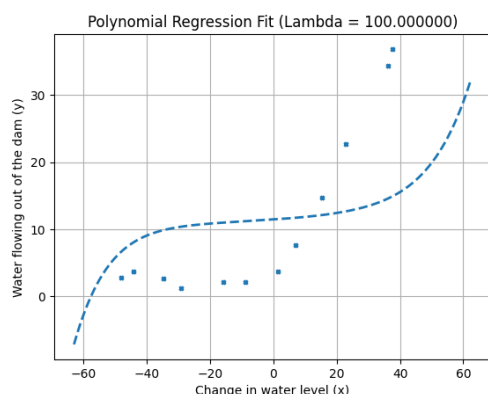


FIGURE 9 – Régression polynomiale $\lambda = 100$

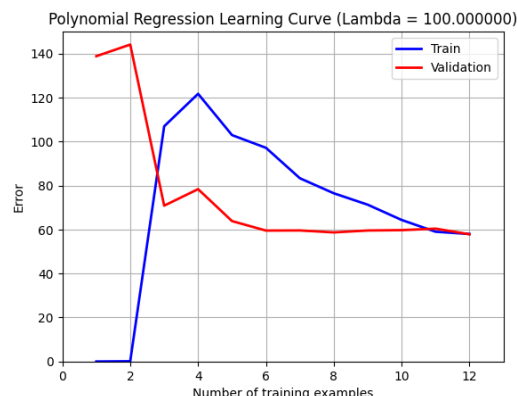


FIGURE 10 – Courbe d'apprentissage $\lambda = 100$

Pour $\lambda = 1$, nos résultats sont satisfaisants. L'erreur d'apprentissage est faible ainsi que l'erreur de validation. Alors que pour $\lambda = 100$, nous pouvons observer une erreur importante et donc un biais élevé (*sous-apprentissage*). Nous pouvons considérer que la valeur de λ la plus appropriée est proche de 1.

3.3 Sélection de λ avec un jeu de validation

Nous avons pu constater qu'une valeur appropriée pour λ se rapproche de 1, mais il ne serait pas judicieux d'affiner notre paramètre manuellement. Pour ce faire nous allons entrainer notre modèle pour une différente valeur de λ puis calculer nos erreurs. Il est important à noter que pour calculer nos erreurs nous pouvons utiliser notre fonction *linearRegCostFunction* avec une valeur de $\lambda = 0$.

```

1 def validationCurve(X, y, Xval, yval):
2     lambda_vec = np.array([0, 0.001, 0.003, 0.01, 0.03, 0.1, 0.3, 1, 3, 10])
3     error_train = np.zeros(lambda_vec.size)
4     error_val = np.zeros(lambda_vec.size)
5
6     for index, Lambda in enumerate(lambda_vec):
7         theta = trainLinearReg(X, y, Lambda)
8         error_train[index], _ = linearRegCostFunction(X, y, theta, 0)
9         error_val[index], _ = linearRegCostFunction(Xval, yval, theta, 0)
10
11     return lambda_vec, error_train, error_val
12
13 """ output
14 Lambda    Train Error  Validation Error
15 0.000000   0.052931   11.079058
16 0.001000   0.137758   12.755159
17 0.003000   0.172035   16.593054
18 0.010000   0.221503   16.946651
19 0.030000   0.281857   12.831260
20 0.100000   0.459305   7.587272
21 0.300000   0.921748   4.636853
22 1.000000   2.076202   4.260622
23 3.000000   4.901350   3.822897
24 10.000000  16.092208   9.945501
25 """

```

Listing 6 – Fonction validationCurve

Après avoir tracé nos erreurs en fonction de λ , nous pouvons sélectionner une valeur λ pour laquelle notre erreur de validation est minimale.

La courbe de validation nous indique que l'erreur est minimale pour une valeur de $\lambda \approx 3$. Si l'on analyse plus en détail la sortie de notre fonction validationCurve, on constate que pour $\lambda = 3$ notre erreur de validation est de 3.822897.

Pour la suite de ce TP nous choisirons donc $\lambda = 3$ comme paramètre de régularisation.

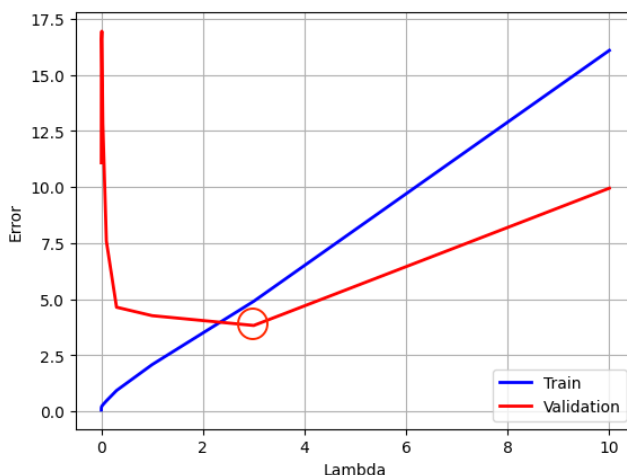


FIGURE 11 – Courbe de validation en fonction de λ

3.4 Calcul de $J_{test}(\theta)$

Nous savons que notre modèle est efficace pour des données qu'il connaît, c'est-à-dire celles que nous avons utilisées pour entraîner notre modèle. Quand est-il pour un ensemble de données inconnus, X_{test} ou X_{poly_test} sous forme polynomiale ? Pour entraîner notre modèle, nous avons sélectionné les paramètres précédents, considérés comme optimaux : X_{poly} , y , $\Lambda=3$, $maxiter=100$.

```

1 Lambda = 3
2 theta = trainLinearReg(X_poly, y, Lambda, maxiter=100)
3
4 Jtest, _ = linearRegCostFunction(X_poly_test, ytest, theta, 0) # Lambda=0, pas de régularisation pour
   ↪ déterminer l'erreur
5 print("The test set error is:", Jtest)
6
7 """output
8 The test set error is: [3.85989019]
9 """

```

Listing 7 – Fonction validationCurve

L'erreur obtenue pour notre ensemble de données inconnues par le modèle est de 3.85989019, ce qui est plus que satisfaisant, notre résultat est proche de l'erreur obtenue pour un jeu de donnée connue. Nous pouvons considérer que notre modèle est correctement paramétré.

3.5 Exerice facultatif

Dans le cas de petit ensemble d'apprentissages comme le nôtre, il est parfois plus intéressant de tracer nos courbes sur la moyenne des erreurs de plusieurs exemples choisis au hasard.

Dans notre cas nous avons un ensemble de longueurs 12, l'idée serait donc de choisir au hasard un certain nombre d'échantillons de cet ensemble, d'obtenir les θ minimaux et de calculer les erreurs d'entraînement et de validation.

Il faut réaliser ces étapes un certain nombre de fois, puis faire la moyenne de nos erreurs. L'algorithme est assez simple (*cf bloc de code 8*).

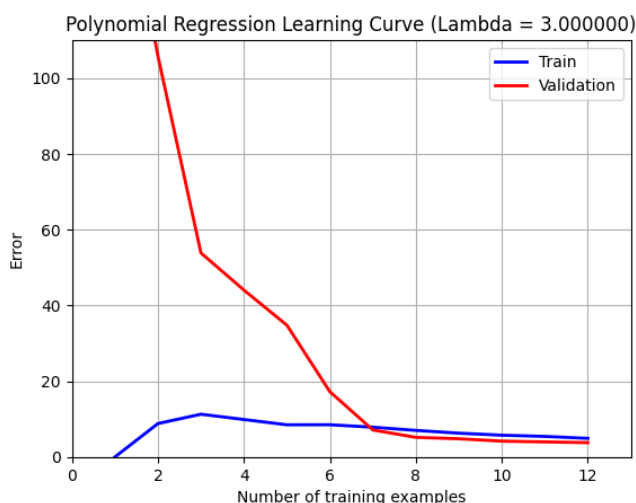


FIGURE 12 – Courbe de validation en fonction de λ

Après avoir obtenu la moyenne de nos erreurs, nous pouvons tracer l'erreur d'entraînement et de validation en fonction du nombre d'échantillons. Dans notre cas, nous avons sélectionné une valeur de $\lambda = 3$, pour comparer les résultats avec ceux précédemment obtenus.

Nous pouvons considérer à l'aide de la figure 12 que notre modèle est bien entraîné, nous n'observons pas de variance ou de biais élevé. L'erreur obtenue est proche des résultats précédents.

```

1  num_iterations = 50
2  Lambda = 0.01
3  errors_train = []
4  errors_val = []
5
6  i = np.random.randint(5, len(X) + 1) # sélection de i hasardeuse
7
8  for _ in range(num_iterations):
9      i_train = np.random.choice(len(X), i, replace=False) # sélection des échantillons dans X
10     i_val = np.random.choice(len(Xval), i, replace=False) # sélection des échantillons dans Xval
11
12     X_random = X_poly[i_train]
13     y_random = y[i_train]
14     X_val_random = X_poly_val[i_val]
15     y_val_random = yval[i_val]
16
17     X_random = np.column_stack((np.ones(X_random.shape[0]), X_random)) # ajout du terme bias
18     X_val_random = np.column_stack((np.ones(X_val_random.shape[0]), X_val_random))
19
20     theta = trainLinearReg(X_random, y_random, Lambda, maxiter=100) # entraînement du modèle
21     error_train, error_val = learningCurve(X_random, y_random, X_val_random, y_val_random, Lambda) #
    ↪ calcul des erreurs
22
23     errors_train.append(error_train) # sauvegarde des erreurs
24     errors_val.append(error_val)
25
26 error_train = np.mean(np.array(errors_train), axis=0) # moyenne des erreurs
27 error_val = np.mean(np.array(errors_val), axis=0)
28
29 plt.figure() # plot des erreurs en fonction du nombre d'échantillons
30 plt.plot(range(1,i+1), error_train, color='b', lw=2, label='Train')
31 plt.plot(range(1,i+1), error_val, color='r', lw=2, label='Validation')
32 plt.title('Polynomial Regression Learning Curve (Lambda = %f)' % Lambda)
33 plt.xlabel('Number of training examples')
34 plt.ylabel('Error')
35 plt.xlim(0, i+1)
36 plt.ylim(0, 110)
37 plt.legend()
38 plt.grid()

```

Listing 8 – Calcul des erreurs avec plusieurs échantillons choisis au hasard

4 Questions

1. Comment découpe-t-on un jeu de données pour entraîner et évaluer un modèle de prédiction ?

Pour entraîner et évaluer un modèle de prédiction, notre jeu de données est découpé en trois groupes :

- **Training set** : Données utilisées pour entraîner notre modèle.
- **Validation set** : Données utilisées pour ajuster les paramètres de notre modèle.
- **Test set** : Données inconnues par le modèle et utilisées l'évaluer.

2. **Définissez ce qu'est la capacité d'un modèle de prédiction (c'est-à-dire sa complexité). Sur quel ensemble l'évalue-t-on ?**

La capacité d'un modèle de prédiction est sa capacité à modéliser des systèmes complexes, ce qui n'est pas le cas d'une régression linéaire simple. La capacité d'un modèle est évaluée avec l'ensemble de validation, celui-ci permet d'ajuster les paramètres de notre modèle et le rendre plus complexe. Si celui-ci est trop complexe alors notre système peut être en surapprentissage, dans le cas contraire en sous-apprentissage.

3. **À quoi sert l'ensemble de validation ?**

L'ensemble de validation est utilisé pour ajuster et améliorer la performance des paramètres de notre modèle. Par exemple dans ce TP nous avons pu sélectionner une valeur de λ optimal.

4. **À quoi sert la "courbe d'apprentissage" ?**

La courbe d'apprentissage permet d'observer l'évolution des erreurs d'entraînement et de validation en fonction du nombre d'échantillons. Elle nous permet de nous assurer de la performance du modèle et la présence de bias élevé (*sous-apprentissage*) ou de variance élevée (*surapprentissage*).