Machine Learning TP4: Régression linéaire régularisé

Melvin DUBEE - Tanguy ROUDAUT FIPASE 24

27 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 appliqué 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 résrvoir

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

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

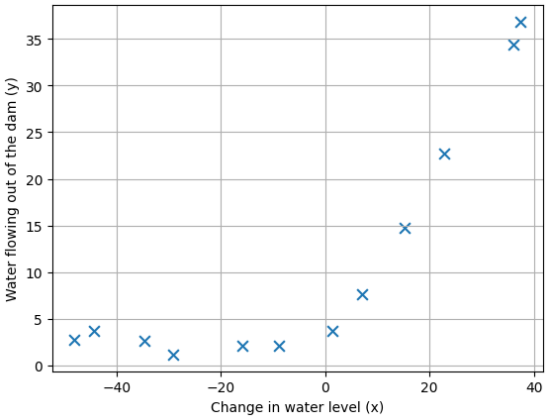

FIGURE 1 – Visualisation du jeu de données

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

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

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

```python
def linearRegCostFunction(X, y, theta, Lambda):
    m,n = X.shape
    theta = theta.reshape((n,1))
    h = X @ theta
    J = (1/(2*m)) * np.sum((h - y) ** 2) + (Lambda/(2*m)) * np.sum(theta[1:] ** 2)

    return J.flatten()

""" output
Cost at theta = [1  1]: 303.993192
(this value should be about 303.993192)
"""
```

Listing 1 – Fonction linearRegCostFunction

1.2 Descente de gradient régularisée

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

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

(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 puisse 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 $J(\theta)$.

```python
def linearRegCostFunction(X, y, theta, Lambda):
    m,n = X.shape # number of training examples
    theta = theta.reshape((n,1)) # in case where theta is a vector (n,)
    h = X @ theta
    J = (1/(2*m)) * np.sum((h - y) ** 2) + (Lambda/(2*m)) * np.sum(theta[1:] ** 2)
    grad = (1/m) * (X.T @ (h - y))
    grad[1:] += (Lambda/m * theta[1:])

    return J.flatten(), grad.flatten()
""" output
Gradient at theta = [1  1]:  [-15.303016 598.250744]
(this value should be about [-15.303016 598.250744])
"""
```

Listing 2 – Fonction linearRegCostFunction

1.3 Visualisation de la régression linéaire régularisée

A l'aide de la fonction *trainLinearReg* qui utilise elle même notre fonction 2 nous pouvons calculer les valeurs optimal de θ.
On constatent que notre régression n'est pas optimal, on peut même dire que nous somme 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 est dû aux nombres insuffisant de caractéristiques, notre θ est seulement de dimension 2. Pour résoudres se problème et augmenter la dimension de θ nous pouvons modifié nos caractéristiques sous forme polynomial.

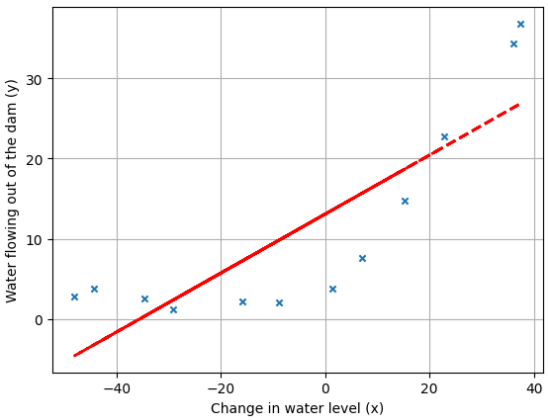

FIGURE 2 – Régression linéaire régularisée

2 Bias et Variance

La notion entre bias et variance est important, 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 sur-apprentissage c'est la variance qui est élevé. De plus, à l'aide de c'est notions il est possible de débuger notre algorithme et de déterminer le paramètre de régularisation optimal λ *cf 3.2.*

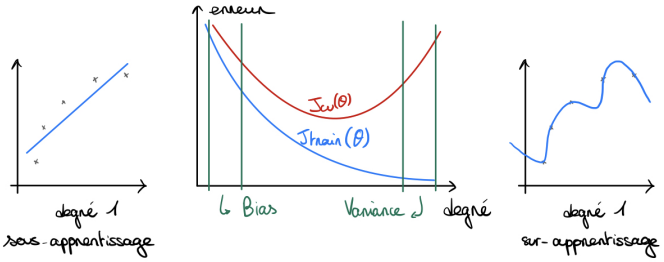

FIGURE 3 – Bias-Variance impacte

2.1 Courbe d'apprentissage

La courbe d'apprentissage consiste à tracé l'erreur d'apprentissage et de validation en fonction de l'ensemble de données.
Pour ce faire nous avons besoin de calculer c'est 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 nouveau échantillon.

```python
def learningCurve(X, y, Xval, yval, Lambda):
    m, _ = X.shape
    error_train = np.zeros((m, 1))
    error_val = np.zeros((m, 1))

    for i in range(m):
        theta = trainLinearReg(X[:i+1,:], y[:i+1], Lambda)
        error_train[i], _ = linearRegCostFunction(X[:i+1,:], y[:i+1], theta, 0)
        error_val[i], _ = linearRegCostFunction(Xval, yval, theta, 0)

    return error_train, error_val

""" output
Training Examples        Train Error        validation Error
          0                0.000000         205.121096
          1                0.000000         110.300407
          2                3.286595         45.010229
          3                2.842678         48.368911
          4                13.154049         35.865164
          5                19.443963         33.829961
          6                20.098522         31.970985
          7                18.172859         30.862446
          8                22.609405         31.135996
          9                23.261462         28.936206
          10                24.317250         29.551431
          11                22.373906         29.433816
"""
```

Listing 3 – Fonction learningCurve

Les erreurs obtenus nous permette de tracer la courbe d'apprentissage.
Nous pouvons observer que le bias est élevé, la courbe d'entrainement et de validation converge vers une erreur d'environ 25. Ce qui confirme notre hypothèse émis à la section 1.3, notre modèle est en sous apprentissage dû aux faible nombre de caractéristiques.

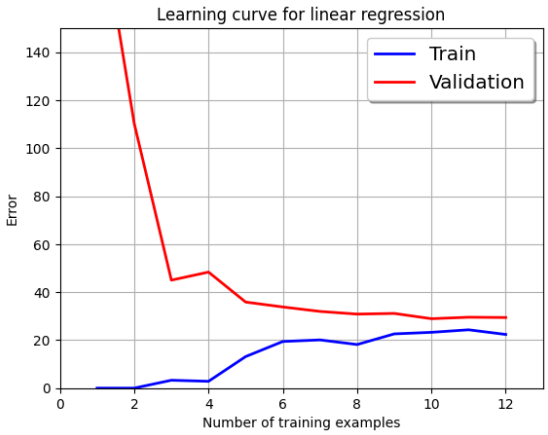

FIGURE 4 – Courbe d'apprentissage

3 Régression polynomiale

Nous avons précédements démontré à plusieurs reprises que notre système comporte un nombre insufisant de caractéristiques, notre bias est élevé *(sous-apprentissage)*.
A l'aide de la fonction 4 nous pouvons augmenter le nombre d'entité 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 colone 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 augmenter le nombre de caractéristiques pour éviter le sous-apprentissage. Cependant élevé 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.

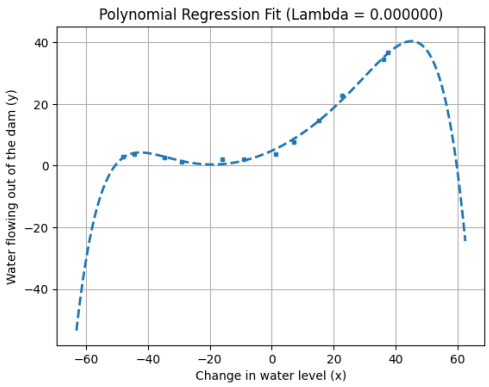

FIGURE 5 – Régression polynomiale $\lambda = 0$

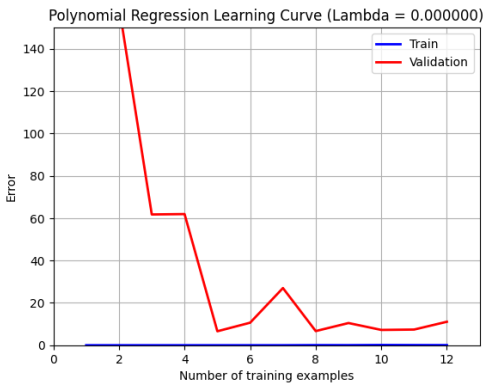

FIGURE 6 – Courbe d'apprentissage $\lambda = 0$

Dans ce premier cas nous n'avons pas appliqué de régulation, comme pour la partie 1, $\lambda = 0$. Avec un theta de dimension 2, nous avions 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 ajusté a notre ensemble de données. Par la suite la figure 6, nous montre que notre erreur d'entrainement est très faible. La présence d'une variance élevé entre l'erreur d'entrainement et de validation nous suggère que nous sommme en sur-apprentissage.

3.2 Réglage du paramètre de régularisation λ

Maintenant que nous somme en sur-apprentissage avec $\lambda = 0$, nous allons utilisé notre régulation mis en œuvre dans la partie 1 pour différente valeurs grâce au code suivant.

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

Listing 5 – Régularisation pour différent λ

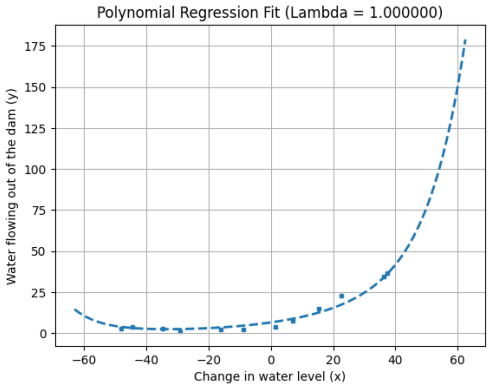

FIGURE 7 – Régression polynomiale $\lambda = 1$

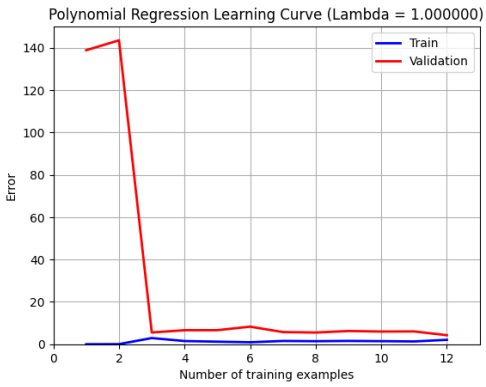

FIGURE 8 – Courbe d'apprentissage $\lambda = 1$

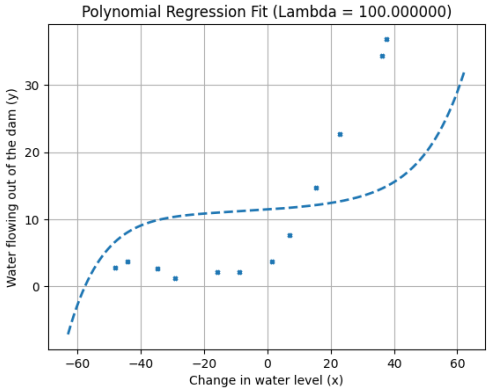

FIGURE 9 – Régression polynomiale $\lambda = 100$

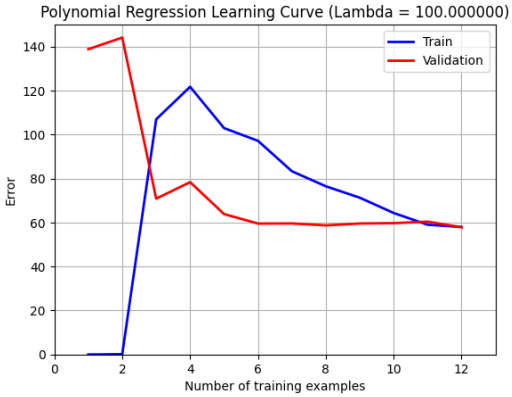

FIGURE 10 – Courbe d'apprentissage $\lambda = 100$

Pour $\lambda = 1$, nos résultats sont satisfaisant. 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 bias élevé *(sous-apprentissage)*. Nous pouvons considérer que la valeur de λ la plus approprié est proche de 1.

3.3 Sélection de λ avec un jeu de validation

Nous avons pu constater qu'une valeur approprié 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 différente valeur de λ puis calculer nos erreur. Il est important a noté que pour calculer nos erreurs nous pouvons utiliser notre fonction *linearRegCostFunction* avec une valeur de $\lambda = 0$.

```python
def validationCurve(X, y, Xval, yval):
    lambda_vec = np.array([0, 0.001, 0.003, 0.01, 0.03, 0.1, 0.3, 1, 3, 10])
    error_train = np.zeros(lambda_vec.size)
    error_val = np.zeros(lambda_vec.size)

    for index, Lambda in enumerate(lambda_vec):
        theta = trainLinearReg(X, y, Lambda)
        error_train[index], _ = linearRegCostFunction(X, y, theta, 0)
        error_val[index], _ = linearRegCostFunction(Xval, yval, theta, 0)

    return lambda_vec, error_train, error_val

""" output
 Lambda    Train Error        Validation Error
 0.000000        0.052931        11.079058
 0.001000        0.137758        12.755159
 0.003000        0.172035        16.593054
 0.010000        0.221503        16.946651
 0.030000        0.281857        12.831260
 0.100000        0.459305        7.587272
 0.300000        0.921748        4.636853
 1.000000        2.076202        4.260622
 3.000000        4.901350        3.822897
 10.000000        16.092208        9.945501
"""
```

Listing 6 – Fonction validationCurve

Après avoir tracer nos erreurs en fonction de λ, nous pouvons sélectionner une valeur λ pour laquelle notre erreur de validation est minimal.
La courbe de validation nous indique que l'erreur est minimal pour une valeur de $\lambda \approx 3$. Si 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.

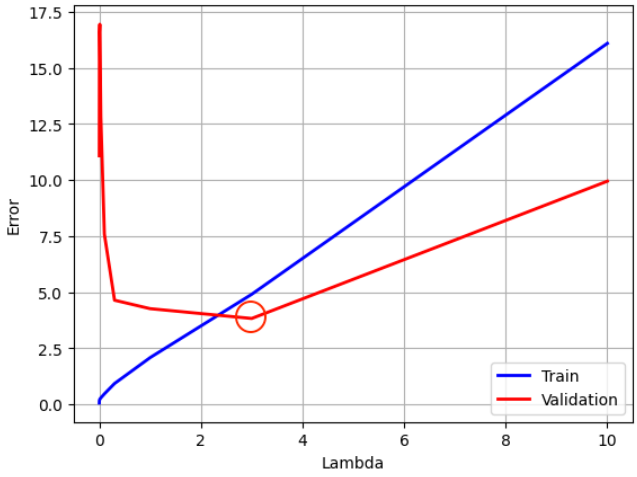

FIGURE 11 – Courbe de validation en fonction de λ

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

Nous savons que notre modèle est efficace pour des données qu'il connait, c'est à dire celles que nous avons utilisés pour entrainer notre modèle. Quand est-il pour un ensemble de données inconnus *Xtest* ou *X_poly_test* sous forme polynomiale. Pour entrainer notre modèle nous avons sélectionner les paramètres précédents considérer comme optimaux : *X_poly, y, Lambda, maxiter=100.*

```
Lambda = 3
theta = trainLinearReg(X_poly, y, Lambda, maxiter=100)

Jtest, _ = linearRegCostFunction(X_poly_test, ytest, theta, 0) # Lambda=0, pas de régularisation pour déterminer l'erreur
print("The test set error is:", Jtest)

"""output
The test set error is: [3.85989019]
"""
```

Listing 7 – Fonction validationCurve

L'erreur obtenus pour notre ensemble de données inconnus par le modèle est de 3.85989019, ce qui est plus que satisfaisant, notre résultat est proche de l'erreur obtenu pour un jeu de donnée connus. Nous pouvons considérer que notre modèle est correctement parametré.

3.5 Exerice facultatif

Dans le cas de petit ensemble d'apprentissage comme le notre il est parfois plus intéressent de tracer nos courbes sur la moyenne des erreurs de plusieurs exemples choisis au hasard.

Dans notre cas nous avons un ensemble de longueur 12, l'idée serait donc de choisir au hasard un certain nombre d'échantillon de cette ensemble, d'obtenir les theta minimaux et de calculer les erreurs d'entrainement et de validation.
Il faut réaliser c'est étapes un certain nombre de fois, puis faire la moyenne de nos erreurs. L'algorithme est assez simple *(cf bloc de code 8)*.

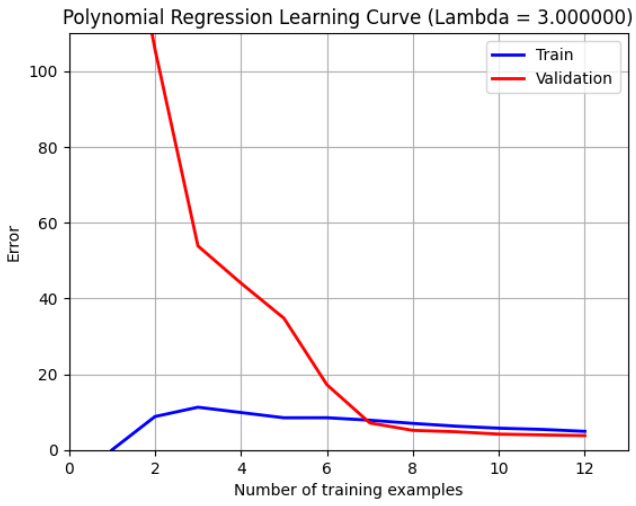

FIGURE 12 – Courbe de validation en fonction de λ

Après avoir obtenus la moyenne de nos erreurs, nous pouvons tracer l'erreur d'entrainement et de validation en fonction du nombre d'échantillon. Dans notre cas nous avons sélectionner une valeur de $\lambda = 3$, pour comparer les résultats avec ceux précédements obtenus.
Nous pouvons considérer à l'aide de la figure 12 que notre modèle est bien entrainé, nous n'observons pas de variance ou de bias élevé. L'erreur obtenus est proche des résultats précédents.

```python
num_iterations = 50
Lambda = 0.01
errors_train = []
errors_val = []

i = np.random.randint(5, len(X) + 1) # selection de i hasardeuse

for _ in range(num_iterations):
    i_train = np.random.choice(len(X), i, replace=False) # selection des echantillons dans X
    i_val = np.random.choice(len(Xval), i, replace=False) # selection des echantillons dans Xval

    X_random = X_poly[i_train]
    y_random = y[i_train]
    X_val_random = X_poly_val[i_val]
    y_val_random = yval[i_val]

    X_random = np.column_stack((np.ones(X_random.shape[0]), X_random)) # ajout du terme bias
    X_val_random = np.column_stack((np.ones(X_val_random.shape[0]), X_val_random))

    theta = trainLinearReg(X_random, y_random, Lambda, maxiter=100) # entrainement du model
    error_train, error_val = learningCurve(X_random, y_random, X_val_random, y_val_random, Lambda) #
    ↪  calcul des erreurs

    errors_train.append(error_train) # sauvegarde des erreurs
    errors_val.append(error_val)

error_train = np.mean(np.array(errors_train), axis=0) # moyenne des erreurs
error_val = np.mean(np.array(errors_val), axis=0)

plt.figure()  # plot des erreurs en fonction du nombre d'échantillon
plt.plot(range(1,i+1), error_train, color='b', lw=2, label='Train')
plt.plot(range(1,i+1), error_val, color='r', lw=2, label='Validation')
plt.title('Polynomial Regression Learning Curve (Lambda = %f)' % Lambda)
plt.xlabel('Number of training examples')
plt.ylabel('Error')
plt.xlim(0, i+1)
plt.ylim(0, 110)
plt.legend()
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 entrainer et évaluer un modèle de prédiction notre jeu de données est découpé en trois groupes :
   — **Training set :** Données utilisées pour entrainer notre modèle.
   — **Validation set :** Données utilisées pour ajuster les paramètres de notre modèle.
   — **Test set :** Données inconus 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 sytèmes complexes, ce qui n'est pas le cas d'une régression linéaire simple. La capacité d'un modèle est évaluer 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 sur-apprentissage, dans le cas contraire en sous-apprentissage.

3. **A 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. **A quoi sert la "courbe d'apprentissage" ?**
   La courbe d'apprentissage permet d'observer l'évolution des erreurs d'entrainement et de validation en fonction du nombre d'échantillon. Elle nous permet de s'assurer de la performance du modèle et la présence de bias élevé *(sous-apprentissage)* ou de variance élevé *(sur-apprentissage)*.