Faculté des sciences et techniques, Beni Mellal

Machine Learning - TP1

Réalisé par :

- Bel-assal Mohamed

Encadré par :

- Pr. Mourad Nachaoui

Année académique : 2023-2024

Table des matières

1 Introduction

Ce travail pratique de Machine Learning explore les techniques de régression linéaire et polynomiale ainsi que l'utilisation des noyaux en apprentissage automatique. Nous avons généré et analysé un jeu de données synthétiques pour évaluer différentes méthodes de régression et observer l'impact des noyaux sur les prédictions.

2 Preparation des donnees

Tout d'abord, nous générons les données en créant 90 points qui suivent la relation $Y = \cos(2X) + \epsilon$, où $f(X) = \cos(2X)$ représente la fonction sous-jacente que nous voulons étudier, et ϵ est un terme de bruit aléatoire introduit pour simuler la variabilité du monde réel. Les valeurs d'entrée X sont échantillonnées à partir d'une plage ou d'une distribution spécifiée, et le bruit ϵ est généralement tiré d'une distribution normale avec une moyenne nulle et une écart-type choisie pour refléter le niveau désiré de randomité.
Ce processus garantit que le jeu de données généré simule des observations réalistes, où la fonction vraie $f(X) = \cos(2X)$ est obscurcie par le bruit, créant ainsi un cadre pour l'ajustement ou l'analyse du modèle.

```
import numpy as np

np.random.seed(42)
n_points = 90

X = np.random.rand(n_points, 1) * 3
epsilon = np.random.randn(n_points, 1) * 0.1

Y = np.cos(2 * X) + epsilon
```

Note : Les données sont générées en utilisant une fonction cosinus pour représenter la tendance sous-jacente et un bruit aléatoire ajouté pour simuler les variations aléatoires observées dans les données réelles.

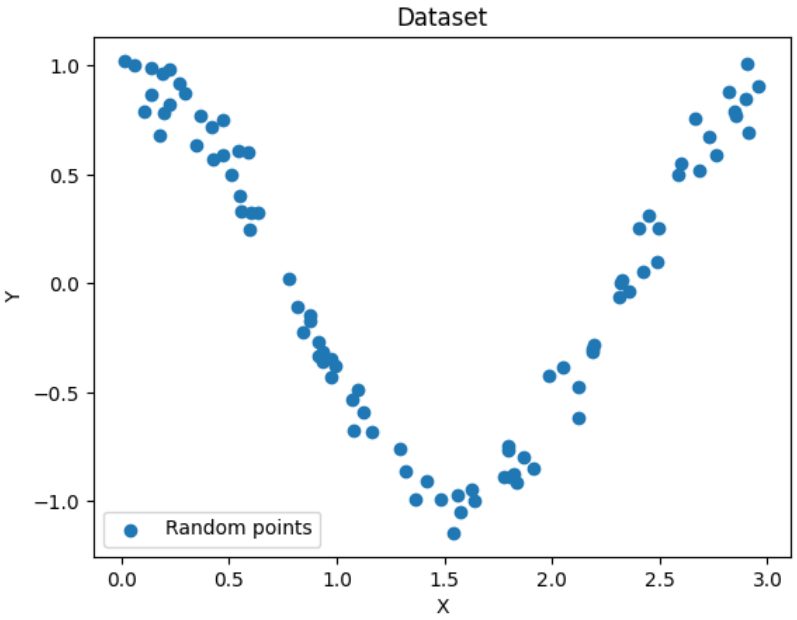

FIGURE 1 – Diagram de dispersion des points de données

Pour la séparation des données, nous allons diviser l'ensemble en ensembles d'entraînement, de test et de validation, chaque ensemble occupant un tiers du total des données.
L'ensemble d'entraînement servira à former le modèle, tandis que l'ensemble de test sera utilisé pour évaluer ses performances. L'ensemble de validation, quant à lui, permettra de peaufiner et ajuster le modèle selon les besoins. Cette approche garantit que notre modèle est bien évalué sur différentes sous-populations des données, offrant une meilleure généralisation et une évaluation plus robuste des performances.

```
indices = np.arange(n_points)
np.random.shuffle(indices)

train_size = n_points // 3
valid_size = n_points // 3

train_indices = indices[:train_size]
valid_indices = indices[train_size:train_size + valid_size]
test_indices = indices[train_size + valid_size:]

X_train, Y_train = X[train_indices], Y[train_indices]
X_valid, Y_valid = X[valid_indices], Y[valid_indices]
X_test, Y_test = X[test_indices], Y[test_indices]
```

Les données sont mélangées pour garantir un découpage aléatoire entre l'entrainement, la validation et le test, en utilisant la fonction `shuffle` de `NumPy` pour mélanger les données avant de les répartir.

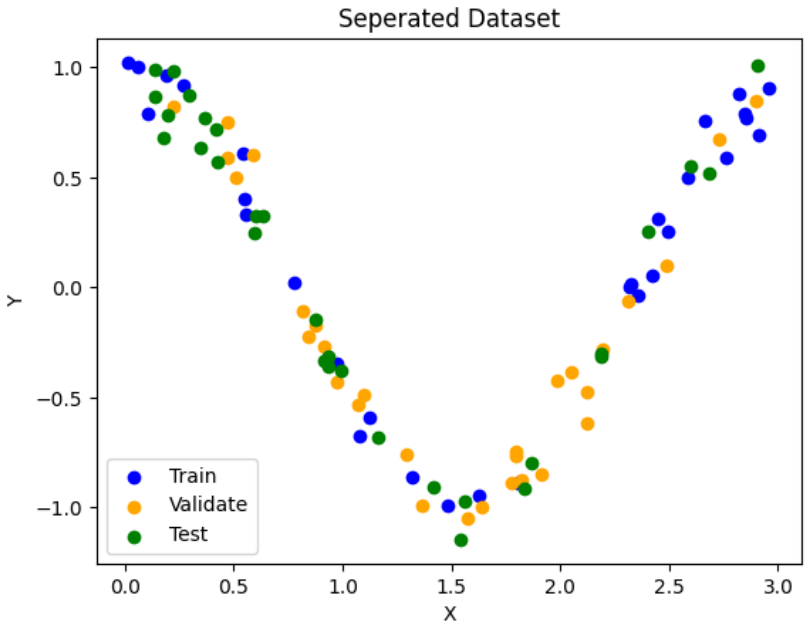

FIGURE 2 – Diagram de dispersion des points de données (Apres la separation)

3 Regression lineaire

La méthode des moindres carrés est utilisée pour estimer les paramètres d'un modèle en minimisant la somme des carrés des écarts entre les valeurs observées et les valeurs prédites.
Formellement, pour un modèle $y = f(x, \theta)$, on cherche à minimiser :

$$S(\theta) = \sum_{i=1}^{n} (y_i - f(x_i, \theta))^2$$

où y_i est la valeur observée, $f(x_i, \theta)$ est la valeur prédite par le modèle, et θ est le vecteur de paramètres.
En régression linéaire, cette méthode permet d'estimer les coefficients de la droite de régression qui minimisent les écarts quadratiques.
Pour estimer les paramètres d'une régression linéaire à l'aide de la méthode des moindres carrés, on cherche les coefficients b (l'interception) et a (le coefficient de la pente) qui minimisent la somme des carrés des écarts entre les valeurs observées et les valeurs estimées.
Les formules pour calculer a et b sont données par :

$$a = \frac{\sum_{i=1}^{n}(X_{\text{train},i} - \bar{X}_{\text{train}}) \cdot (Y_{\text{train},i} - \bar{Y}_{\text{train}})}{\sum_{i=1}^{n}(X_{\text{train},i} - \bar{X}_{\text{train}})^2}$$

$$b = \bar{Y}_{\text{train}} - a \cdot \bar{X}_{\text{train}}$$

où $X_{\text{train},i}$ et $Y_{\text{train},i}$ sont les points individuels des ensembles de données d'entraînement pour X et Y respectivement, $\bar{X}_{\text{train}}$ et $\bar{Y}_{\text{train}}$ sont les moyennes de X et Y dans l'ensemble d'entraînement, et n est le nombre de points de données.
La droite de régression prédite Y_{line} peut être exprimée par :

$$Y_{\text{line}} = b + a \cdot X_{\text{train}}$$

où X_{train} est une valeur donnée pour la variable indépendante, b est l'interception, et a est la pente.

```
mean_train_x = np.mean(X_train)
mean_train_y = np.mean(Y_train)
numerator = np.sum((X_train - mean_train_x) * (Y_train - mean_train_y))
denominator = np.sum((X_train - mean_train_x) ** 2)

a = numerator / denominator
b = mean_train_y - a * mean_train_x

Y_line = b + a * X_train
```

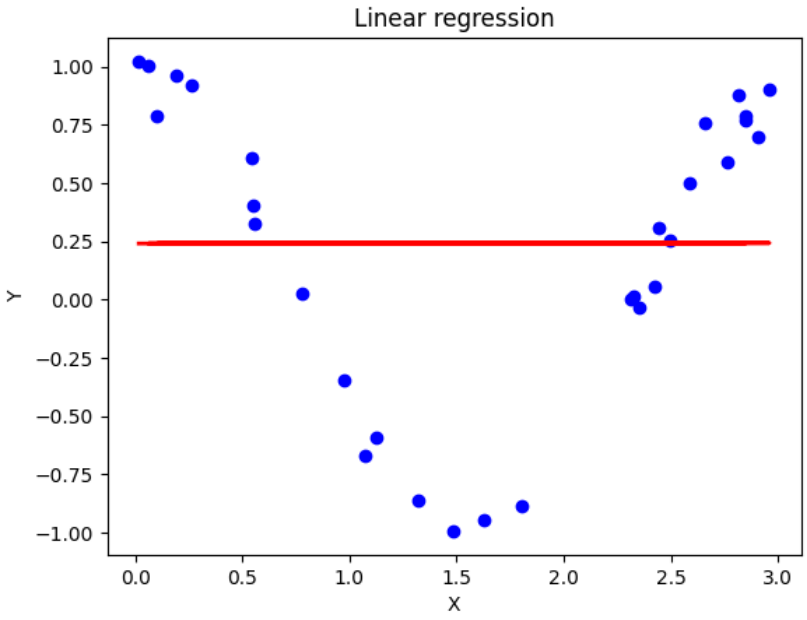

FIGURE 3 – Regression lineaire

4 Equation normale

L'équation normale est un outil efficace pour estimer les coefficients dans un modèle de régression, même lorsque les variables indépendantes sont exprimées sous forme de polynômes de degré p. Voici comment fonctionne le processus :
Les variables indépendantes X sont transformées en leurs puissances jusqu'au degré p. Par exemple, pour un polynôme de degré 2, on forme un vecteur X contenant tous les termes de puissance :

$$X = \begin{pmatrix} 1 & X_1 & X_1^2 \\ 1 & X_2 & X_2^2 \\ \vdots & \vdots & \vdots \\ 1 & X_n & X_n^2 \end{pmatrix}^\top$$

Chaque ligne de X correspond à une observation avec les termes de puissance pour chaque variable indépendante. L'objectif est d'estimer les coefficients A du modèle, qui minimisent la somme des carrés des erreurs entre les valeurs observées de Y et les valeurs prévues par le modèle. Cela se fait par l'estimation de A selon l'équation normale :

$$\beta = (X^\top X)^{-1} X^\top Y$$

Pour prédire Y pour de nouvelles observations, on utilise les coefficients estimés A. Si X_{new} est la matrice de variables indépendantes pour les nouvelles observations, on peut prédire Y_{pred} comme suit :

$$Y_{\text{pred}} = X_{new} \cdot A$$

où X_{new} est une matrice dont chaque ligne correspond à une nouvelle observation avec les termes de puissance appropriés pour chaque variable.

4.1 Estimation des Coefficients (degrés 1 à 15)

Nous mettons en œuvre cette méthode pour estimer les coefficients du polynôme de régression pour des degrés allant de 1 à 15. Cela nous permet de visualiser comment la forme des polynômes influence les prédictions. Les fonctions obtenues pour chaque degré seront représentées sur le même graphe.

Pour chaque degré, la matrice de design X doit être normalisée afin de standardiser les variables indépendantes. Étant donné une matrice de design $X \in \mathbb{R}^{n \times d}$ comportant n données de dimension d, la normalisation vise à transformer les données de manière à les centrer et à les réduire à une plage de valeurs plus petite, améliorant ainsi la performance du modèle.

```
data = {
    'X': X_train.flatten(),
    'Y': Y_train.flatten()
}
df = pd.DataFrame(data)

df_sorted = df.sort_values(by='X', ascending=True)
X_train2 = df[['X']].values
Y_train2 = df['Y'].values
```

Nous avons utilisé un DataFrame et trié les données pour nous assurer qu'elles sont ordonnées, et que chaque valeur 'X' correspond à sa 'Y'.

```
def deg_poly(X, Y, deg):
    X_poly = np.vander(X.flatten(), deg + 1, increasing=True)
    XTX = np.dot(X_poly.T, X_poly)
    XTY = np.dot(X_poly.T, Y)

    A_poly = np.dot(np.linalg.inv(XTX), XTY)
    return A_poly
```

La fonction `deg_poly` calcule les paramètres d'un modèle polynomial en utilisant la matrice de design `X_poly` pour les données `X`. Elle résout l'équation linéaire

$$A_{\text{poly}} = (XTX)^{-1} \cdot XTY$$

pour ajuster un polynôme d'ordre 'deg' aux données d'entrée.

```
colors = plt.cm.viridis(np.linspace(0, 1, 15))

for i, deg in enumerate(range(1, 16)):
    A_poly = deg_poly(X_train2, Y_train2, deg)

    X_grid = np.linspace(min(X_train2), max(X_train2), 100).reshape(-1, 1)
    X_grid_poly = np.vander(X_grid.flatten(), deg + 1, increasing=True)
    Y_grid_pred = np.dot(X_grid_poly, A_poly.flatten())

    plt.plot(X_grid, Y_grid_pred, color=colors[i], label=f"Degree {deg}")
```

On trace des ajustements polynomiaux de degrés allant de 1 à 15. Il utilise une carte de couleurs ('viridis') pour distinguer les différents degrés de polynômes, chaque degré étant associé à une couleur unique. Pour chaque degré de polynôme, les coefficients sont calculés à l'aide de la fonction deg_poly, puis un ensemble de valeurs X est créé pour évaluer l'ajustement polynomique.
Les valeurs prédites Y sont calculées à l'aide de ces coefficients polynomiaux selon la formule :
$Y_{\text{pred}} = X_{\text{poly}} \cdot A_{\text{poly}}$, et les ajustements polynomiaux obtenus sont tracés avec des étiquettes appropriées.

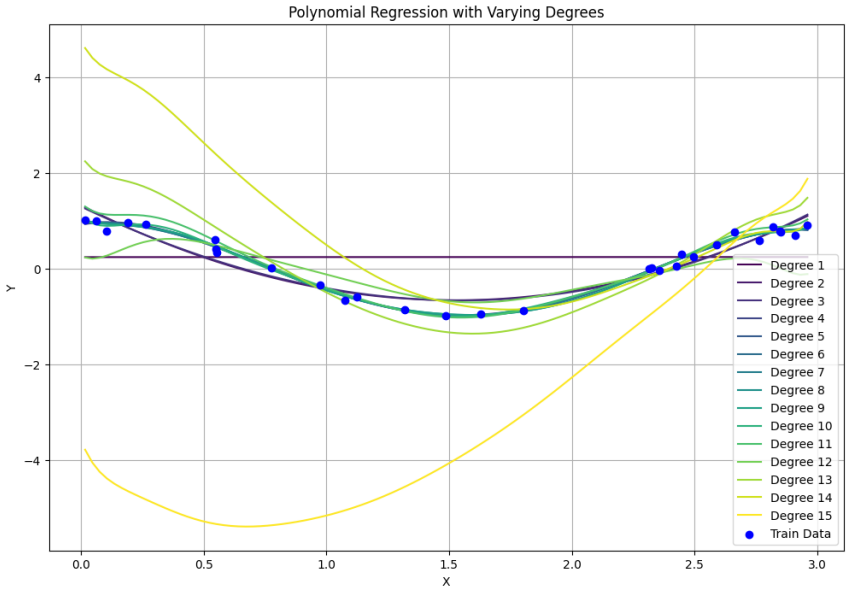

FIGURE 4 – Prediction de Y pour les degrés 1 à 15 (entraînement)

4.2 Erreur de regression - Ensemble d'entraînement

L'erreur quadratique moyenne (MSE) est une mesure couramment utilisée pour évaluer la performance d'un modèle de régression. Elle quantifie la moyenne des carrés des erreurs, c'est-à-dire la différence entre les valeurs prédites par le modèle et les valeurs réelles. Une MSE faible indique un bon ajustement du modèle aux données, tandis qu'une MSE élevée suggère que le modèle ne capture pas bien la tendance des données ou qu'il surajuste les données en capturant le bruit. Pour calculer le MSE pour chaque degré, on utilise la formule suivante pour chaque prédiction :

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

où n est le nombre de points de données de test, y_i est la vraie valeur et $\hat{y}_i$ est la prédiction.

```
mse = np.mean((Y_train.flatten() - Y_train_pred.flatten()) ** 2)
```

On ajoute le calcul du MSE dans la boucle précédente afin de déterminer l'erreur pour chaque degré.

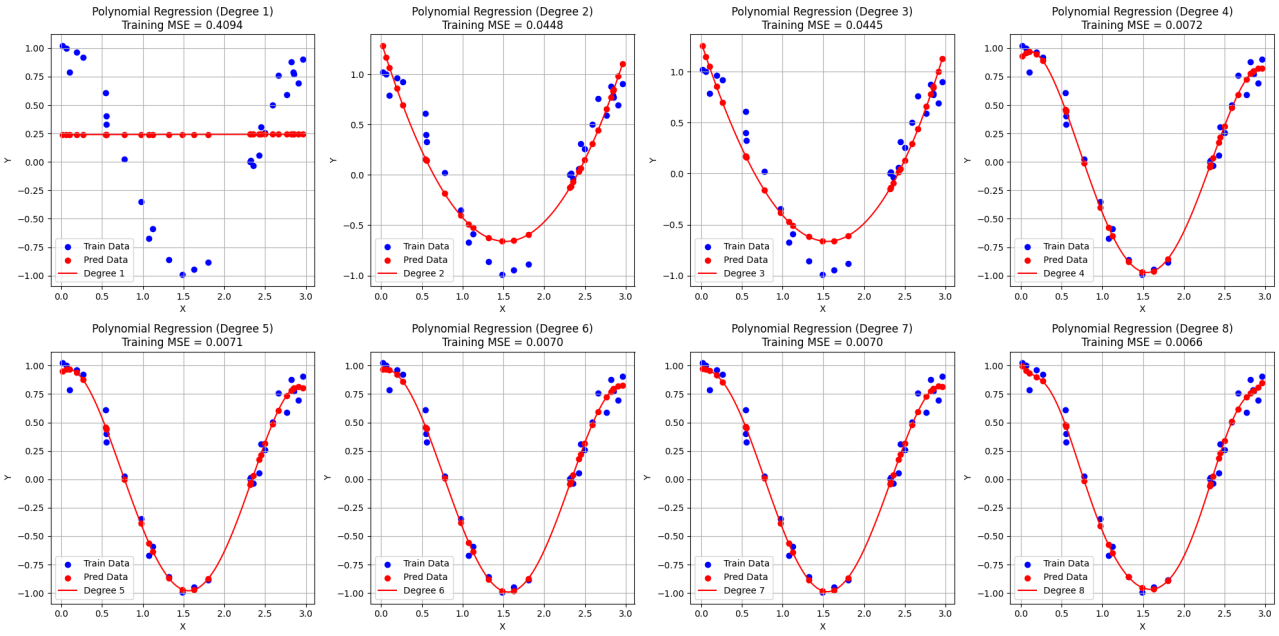

FIGURE 5 – MSE pour les degrés 1 à 8 (entraînement)

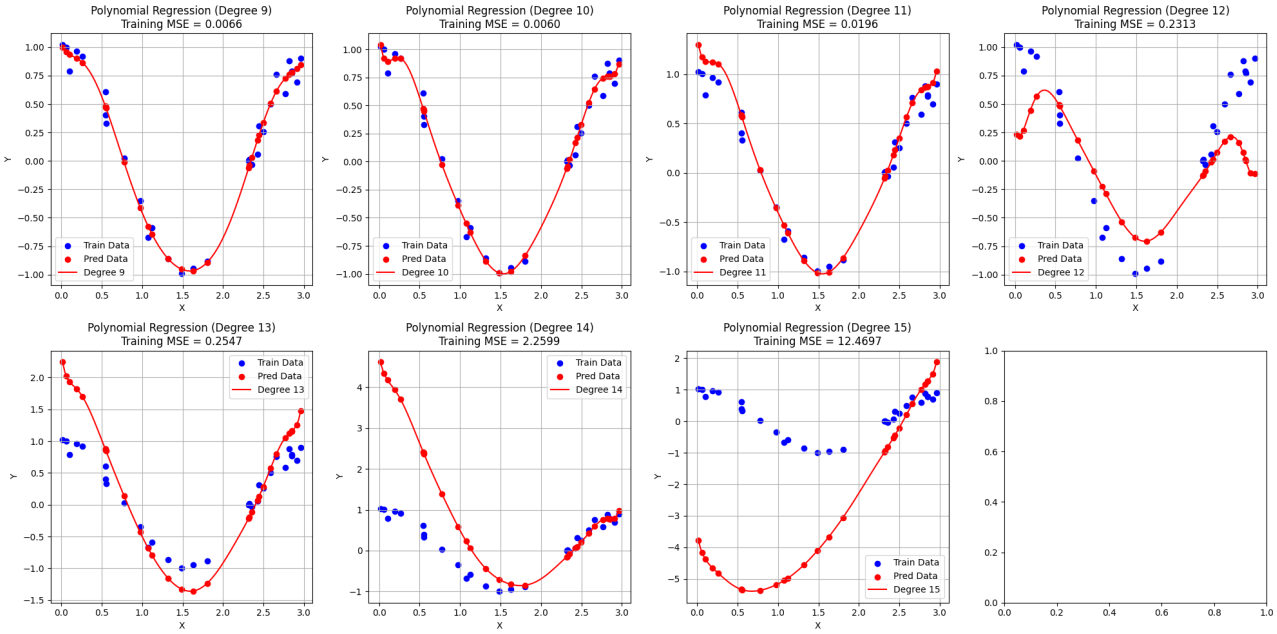

FIGURE 6 – MSE pour les degrés 9 à 15 (entraînement)

Les graphiques présentés montrent les résultats de régressions polynomiales de différents degrés appliquées à l'ensemble de données d'entraînement.

Degré 1 : La régression linéaire ne capture pas bien la forme parabolique des données, ce qui se traduit par une erreur de régression (MSE) élevée de 0.4094.

Degré 2 et 3 : La régression quadratique commence à mieux capturer la forme des données, avec une MSE autour de 0.004.

Degré 4 à 10 : Les régressions polynomiales de degré 3 à 8 montrent une amélioration significative de l'ajustement aux données, avec des MSE très faibles (autour de 0.007 - 0.006). Les courbes rouges suivent de près les points bleus des données d'entraînement.

Degré 11 à 15 : On observe une augmentation significative de la MSE, indiquant un surajustement (overfitting). Les courbes rouges commencent à montrer des oscillations importantes entre les points de données, ce qui est typique du surajustement.

En résumé, il est crucial de choisir un degré de polynomial approprié pour éviter à la fois le sous-ajustement et le surajustement. Dans ce cas, les degrés intermédiaires (environ 3 à 11) semblent offrir le meilleur compromis.

4.3 Erreur de regression - Ensemble de test

Nous répétons le même processus d'entraînement du modèle en gardant le même vecteur de paramètres obtenu précédemment. Cependant, cette fois, au lieu d'utiliser les données d'entraînement, nous appliquons le modèle à des données de test. Le but est de voir comment le modèle se comporte sur de nouveaux ensembles de données qu'il n'a pas vus pendant l'entraînement.

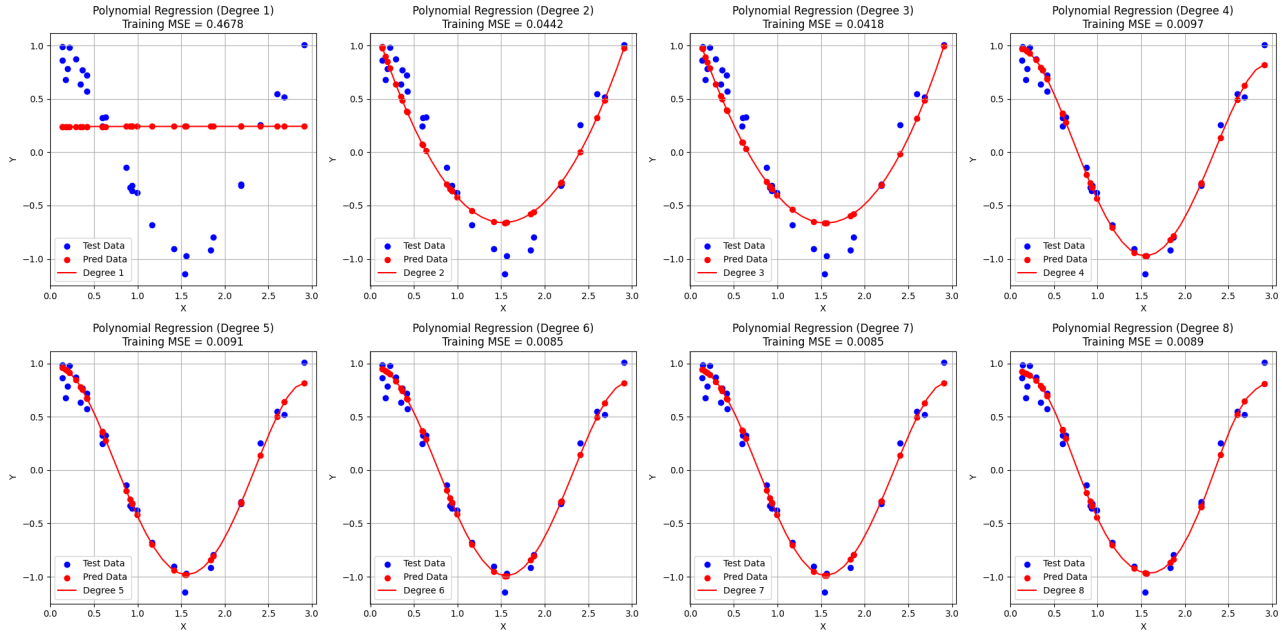

FIGURE 7 – MSE pour les degrés 1 à 8 (test)

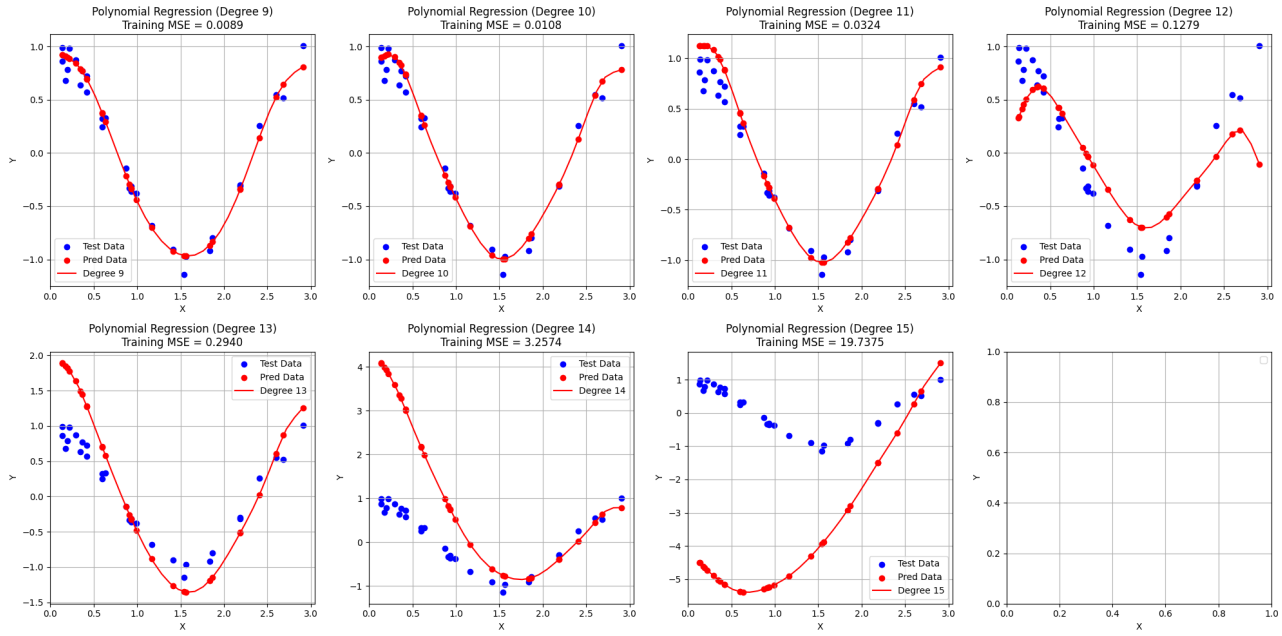

FIGURE 8 – MSE pour les degrés 9 à 15 (test)

Degré 1 : La régression linéaire ne capture pas bien la forme parabolique des données, ce qui se traduit par une erreur de régression (MSE) élevée de 0.4678.

Degré 2 et 3 : La régression quadratique commence à mieux capturer la forme des données, avec une MSE autour de 0.0442.

Degré 4 à 10 : Les régressions polynomiales de degré 4 à 8 montrent une amélioration significative de l'ajustement aux données, avec des MSE très faibles (autour de 0.009). Les courbes rouges suivent de près les points bleus des données d'entraînement.

Degré 11 à 15 : On observe une augmentation significative de la MSE, indiquant un surajustement (overfitting). Les courbes rouges commencent à montrer des oscillations importantes entre les points de données, ce qui est typique du surajustement.

4.4 Comparaison entre entraînement et test

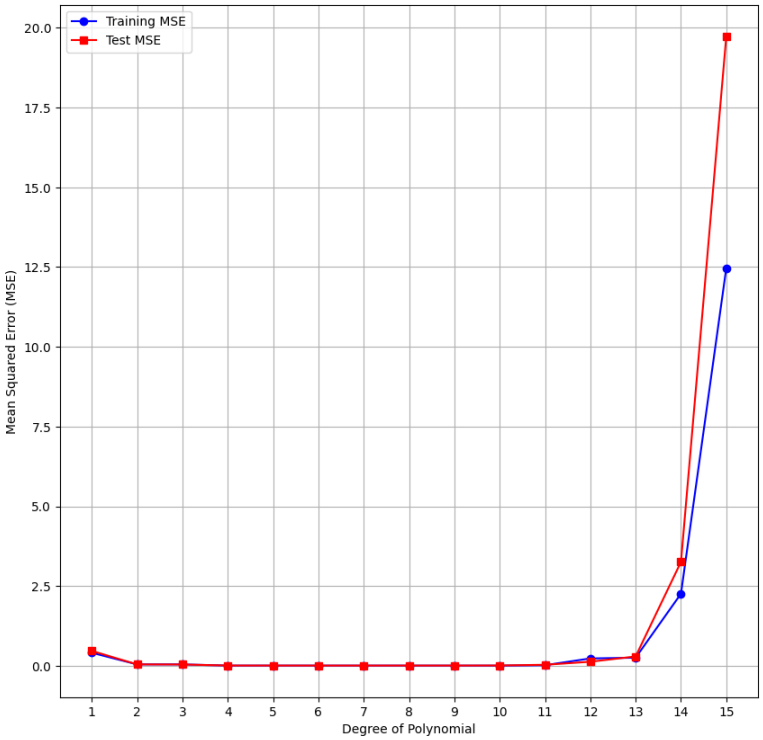

FIGURE 9 – MSE d'entraînement vs MSE de test (1 a 15)

MSE pour l'entraînement : Il diminue avec l'augmentation du degré polynomique jusqu'à un certain point (degré 8), puis commence à augmenter de manière significative. Cela indique que des degrés plus élevés peuvent commencer à surajuster le modèle aux données d'entraînement, capturant le bruit au lieu du patron sous-jacent.

MSE pour les tests : Montre également une tendance où il diminue initialement avec l'augmentation du degré jusqu'au degré 6, puis commence à augmenter significativement à partir du degré 7. Cela suggère que des degrés plus élevés de régression polynomiale sont moins généralisables aux données test non vues, ce qui peut être un signe de surajustement.

La zone idéale pour équilibrer le compromis biais-variance semble être aux degrés 4, 5 ou 6, où les erreurs d'entraînement et de test sont relativement faibles. Au-delà de ces degrés, le modèle devient moins robuste et les performances sur les données de test se détériorent de manière significative.

4.5 Evolution des erreurs en fonction de dataset

Nous augmentons le nombre de points de 90 à 300, et on observe l'évolution des erreurs de test et d'entraînement.

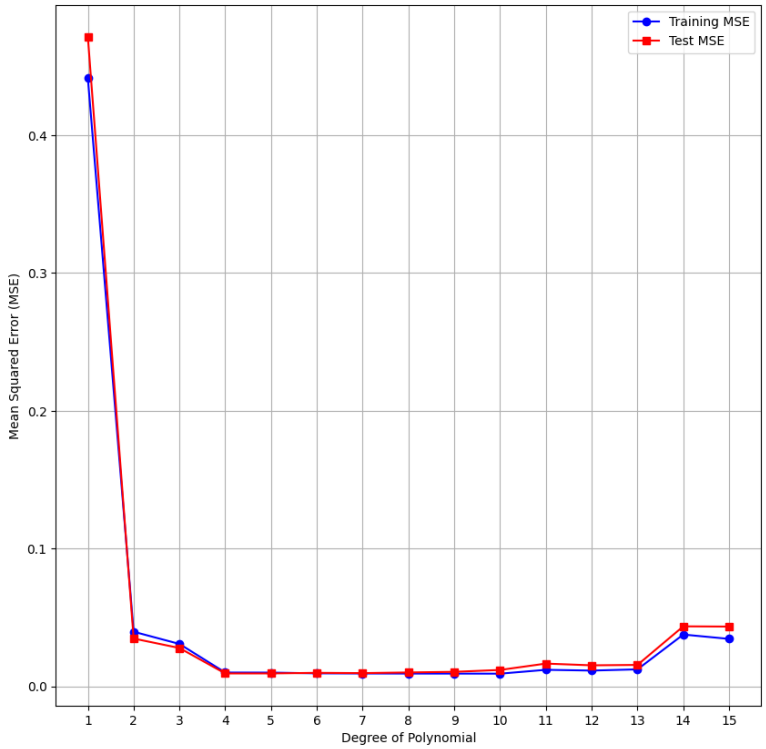

FIGURE 10 – MSE d'entraînement vs MSE de test (1 a 15) - 300 points

Le modèle avec 300 points fonctionne bien sur une large gamme de degrés de polynômes, offrant une performance solide même à des degrés élevés. Cependant, il montre des signes de surapprentissage lorsqu'il est confronté à des polynômes de très haut degré, ce qui peut affecter sa précision générale. D'un autre côté, le modèle avec 90 points fonctionne bien jusqu'à un certain degré de polynôme, mais commence à présenter des signes de surapprentissage sévère lorsqu'il atteint des degrés plus élevés. Cela suggère que, bien que le modèle avec 90 points soit efficace pour des polynômes de faible et moyen degré, il pourrait ne pas être idéal pour des polynômes complexes nécessitant une grande flexibilité.

5 Les noyaux en apprentissage automatique

En apprentissage automatique, un **noyau** (ou *kernel*) est une fonction qui mesure la similarité entre deux données dans un espace. Les noyaux sont souvent utilisés dans des algorithmes comme les Machines à Vecteurs de Support (SVM) pour effectuer des calculs dans des espaces de caractéristiques de dimension élevée sans avoir besoin de transformer explicitement les données dans cet espace. Cela s'appelle le **truc du noyau** (*kernel trick*).

Un noyau $k(x, x')$ satisfait généralement les propriétés suivantes :

1. $k(x, x')$ est symétrique : $k(x, x') = k(x', x)$.
2. La matrice de Gram associée au noyau doit être semi-définie positive.

5.1 Notre noyau spécifique

Nous utiliserons un noyau défini comme suit :

$$k(x, x') = \begin{cases} 1 & \text{si } \|x - x'\| < \lambda, \\ 0 & \text{sinon.} \end{cases}$$

Ce noyau évalue si deux vecteurs x et x' sont proches l'un de l'autre dans un rayon λ. Si la distance euclidienne

entre eux est inférieure à λ, la fonction retourne 1, sinon elle retourne 0. Cela correspond à un noyau très simple basé sur un critère de voisinage. on le programme comme suit :

```
def kernel(x_i, x_j, lambda_):
    return 1 if np.abs(x_i - x_j) < lambda_ else 0
```

Ce type de noyau est utile pour des tâches où les relations locales (proximité) sont plus importantes que des relations globales. Il permet de capturer des structures locales dans les données, ce qui peut améliorer la performance des modèles dans certains contextes spécifiques.

```
y_grid_pred = []
for i in range(len(X_grid)):
    weights = np.array([kernel(X_grid[i], X_train2[j], lambda_) for j in
        range(len(X_train2))])
    numerator = np.sum(weights * Y_train2)
    denominator = np.sum(weights)
    y_grid_pred.append(numerator / denominator if denominator > 0 else 0)

y_grid_pred = np.array(y_grid_pred)
```

5.2 Tests et resultats

Nous allons maintenant tester ce noyau en utilisant différentes valeurs de λ afin d'observer comment ce paramètre influence les prédictions. En effet, λ contrôle le rayon dans lequel deux vecteurs sont considérés comme similaires. Une valeur plus grande de λ rend le noyau moins strict, augmentant ainsi le nombre de paires (x, x') considérées comme proches. À l'inverse, une valeur plus petite de λ restreint davantage les similarités.

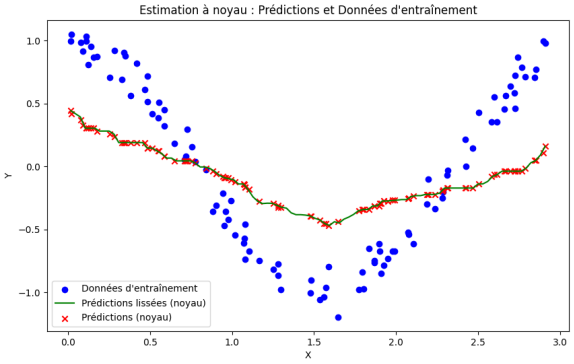

FIGURE 11 – Test du noyau (lamda = 1)

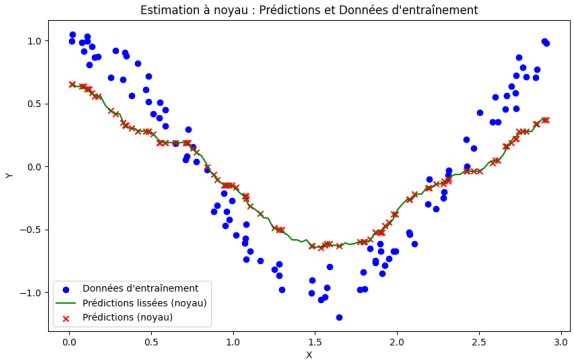

FIGURE 12 – Test du noyau (lamda = 0.75)

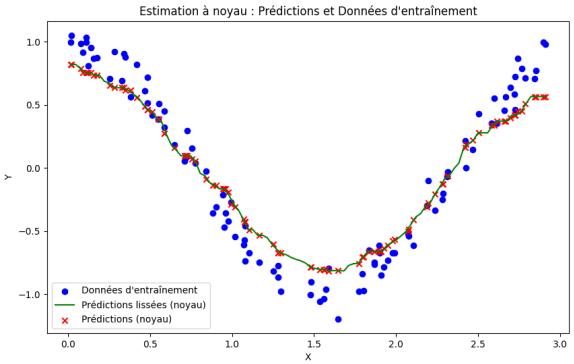

FIGURE 13 – Test du noyau (lamda = 0.5)

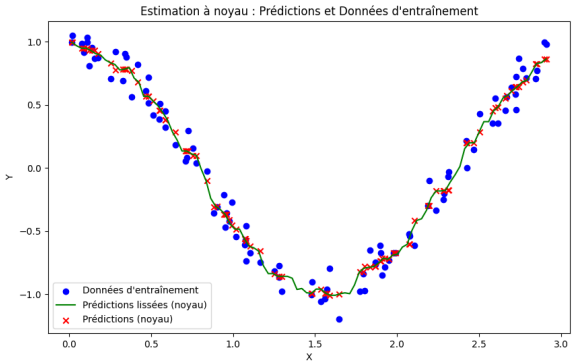

FIGURE 14 – Test du noyau (lamda = 0.1)

Pour évaluer l'impact des différentes valeurs de λ sur les prédictions, nous allons calculer l'**Erreur Relative Moyenne** (MRE). c'est une mesure qui permet d'évaluer la précision des prédictions en comparant les prédictions aux valeurs réelles, normalisée par rapport aux valeurs réelles.

La formule de l'Erreur Relative Moyenne (MRE) est la suivante :

$$\text{MRE} = \frac{1}{n} \sum_{i=1}^{n} \frac{|y_i - \hat{y}_i|}{|y_i|}$$

où :

— y_i est la valeur réelle de l'observation i,

— $\hat{y}_i$ est la prédiction pour l'observation i,

— n est le nombre total d'observations.

Nous allons maintenant calculer la MRE pour chaque valeur de λ.

```
mra = np.mean(np.abs(y_pred_kernel - Y_train2))
print(f"Moyenne des r sidus absolus (noyau) : {mra:.4f}")
```

Voici les résultats pour différentes valeurs de λ :

λ	Moyenne des Résidus Absolus (MRA)
1	0.4435
0.75	0.2978
0.5	0.1528
0.1	0.0853

TABLE 1 – Moyenne des Résidus Absolus (MRA) pour différentes valeurs de λ.

Les résultats montrent que plus la valeur de λ est petite, plus la MRA (erreur) est faible. En effet, lorsque $\lambda = 0.1$, la MRA est la plus basse, indiquant que le modèle fait moins d'erreurs. Cela suggère qu'un λ plus petit, qui se concentre sur des relations plus proches, permet d'obtenir de meilleures prédictions. Toutefois, une valeur trop petite pourrait aussi conduire à un sous-ajustement, donc un équilibre est nécessaire.

6 Conclusion

En conclusion, ce projet a mis en lumière l'importance de choisir le bon degré de polynôme pour éviter le sous-ajustement et le surajustement dans les modèles de régression. L'utilisation de noyaux spécifiques a également montré leur efficacité dans la capture des relations locales dans les données, améliorant ainsi la précision des prédictions.