



**UNIVERSITÉ
DE GENÈVE**

University of Geneva
Department of Science
Bachelor's Degree in Computer Science
Academic Year 2023-2024

Data Mining : Molecular Property Prediction

Supervisors:

Prof. Alexandros Kalousis
Assit. Victor Reyes Martin
Assit. Yoan Boget

Student:

VANSON Nathan
PAPA David

June 23, 2024

Contents

1	Introduction	4
1.1	Contexte et Problématique	4
1.2	Présentation des Données	4
1.3	Objectifs du Projet	5
1.4	Structure du Rapport	5
2	Configuration de l'environnement de travail	7
2.1	Gestion de Version avec GitHub	7
2.2	Gestion des Dépendances avec Poetry	7
2.3	Environnement Python et Bibliothèques	7
2.4	Structuration du Projet	7
3	Prétraitement et Gestion des Données	9
3.1	Chargement des données	9
3.2	Extraction des données	9
3.3	Séparation des données	9
3.4	Corrélation	10
4	Implémentation et Analyse Descriptive des Algorithmes	14
4.1	Baseline K-nearest neighbors (k-NN)	14
4.1.1	Fonctionnement de k-NN	14
4.1.2	Choix du Paramètre k	14
4.1.3	Métriques de Distance dans k-NN	15
4.1.4	Algorithme	15
4.1.5	Avantages et Limitations	16
4.2	Random Forest Regression Models	16
4.2.1	Fonctionnement de Random Forest Regression	17
4.2.2	Structure et Paramètres clés	17
4.2.3	Algorithme	18
4.2.4	Avantages et Limitations	19
4.3	Gradient Boosting Models	19
4.3.1	Fonctionnement du Gradient Boosting	20
4.3.2	Paramètres Clés	20
4.3.3	Algorithme	20
4.3.4	Avantages et Limitations	21
4.4	Neural Networks Models (NN)	21
4.4.1	Fonctionnement du Neuronal Networks	22
4.4.2	Importance des Paramètres	24
4.4.3	Explication du Code	25
4.4.4	Avantages et Limitations	26
5	Évaluation des Modèles	27
5.1	Créations des modèles	27
5.1.1	K-Nearest Neighbors (KNN)	27
5.1.2	Random Forest	27
5.1.3	Gradient Boosting	28
5.1.4	Neural Network	28
5.2	Prédiction	29

5.2.1	Prédictions du pIC50	29
5.2.2	Prédictions du logP	30
5.3	Calcul de l'erreur	31
5.3.1	Métrique d'évaluation	31
5.3.2	Erreur pour le pIC50	32
5.3.3	Erreur pour le logP	33
6	Conclusion	35
7	Contributions	36
8	References	36

List of Figures

1	Molécule Organique	4
2	Exemple de données des molécules	5
3	Organisation et structure du dépôt GitHub du projet	8
4	Matrice de corrélation pour nos données	11
5	Corrélation des paramètres avec le pIC50	12
6	Corrélation des paramètres avec le logP	12
7	Aperçu de l'algorithme KNN	14
8	Effet de K sur les frontières de classes	15
9	Visualisation de l'algorithme Random Forest	17
10	Fonctionnement du gradient boosting	19
11	Exemple d'un réseaux de neuronne	22
12	Exemple de perte d'entraînement et de validation : pIC50 vs logP	24
13	Comparaison des différents modèles pour pIC50	30
14	Comparaison des différents modèles pour logP	31
15	Métriques de regression pour les différents modèles sur pIC50	33
16	Métriques de regression pour les différents modèles sur logP	34

1 Introduction

1.1 Contexte et Problématique

Les molécules jouent un rôle crucial dans notre quotidien, de la formulation des médicaments aux produits de consommation courante comme les cosmétiques et les détergents. Comprendre les propriétés moléculaires est essentiel pour de nombreuses industries, notamment la pharmacologie et la découverte de médicaments, où il est fondamental de prédire des mesures telles que le pIC₅₀, qui évalue la puissance d'un composé à inhiber une cible biologique spécifique, et le logP, qui indique la préférence d'une molécule pour les environnements non polaires (huileux) ou polaires (aqueux). Dans ce projet, nous nous concentrerons sur la prédiction de ces deux propriétés moléculaires essentielles, ce qui peut grandement améliorer la compréhension et le développement de nouveaux composés chimiques.

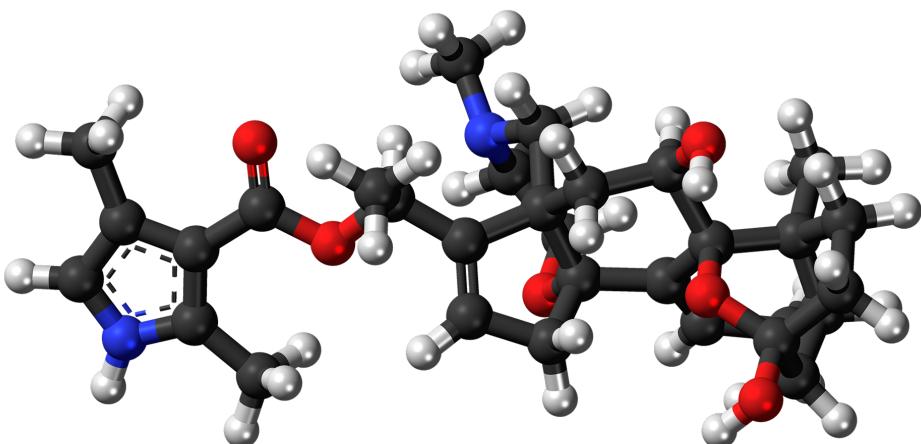


Figure 1: Molécule Organique

1.2 Présentation des Données

Le jeu de données fourni (`data.csv`) contient des valeurs formatées pour 16 087 molécules représentées par des chaînes SMILES (Simplified Molecular Input Line Entry System), accompagnées de leurs valeurs de pIC₅₀ et logP. Chaque molécule est également caractérisée par le nombre d'atomes, information qui peut être déduite des représentations SMILES. Les molécules peuvent être représentées de différentes manières : en séquence (SMILES), en graphe (avec des atomes comme noeuds et des liaisons comme arêtes), ou en structure 3D.

'O=S(=O)(Nc1cccc(-c2cnc3ccccc3n2)c1)c1cccs1'
'O=c1cc(-c2nc(-c3ccc(-c4cn(CCP(=O)(O)O)nn4)cc3[nH]c2-c2ccc(F)cc2)cc[nH]1'
'NC(=O)c1ccc2(c1)nc(C1CCC(O)CC1)n2CCCO'
'NCCCCn1c(C2CCCNCC2)nc2cc(C(N)=O)ccc21'

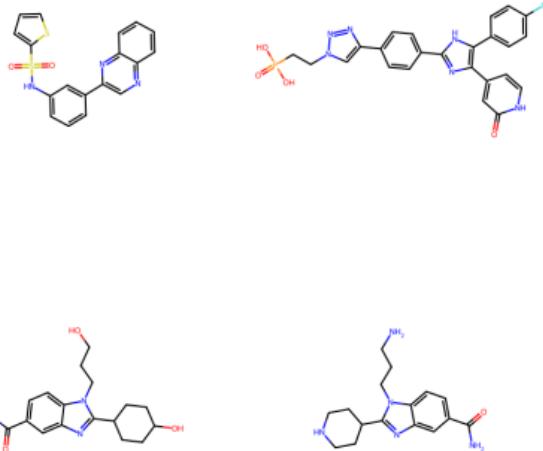


Figure 2: Exemple de données des molécules

1.3 Objectifs du Projet

L'objectif de ce projet est de développer des modèles prédictifs pour estimer le pIC₅₀ et le logP des molécules [1] en utilisant différentes représentations et techniques de modélisation. Les étapes clés incluent la compréhension des données, le choix des représentations moléculaires appropriées, l'extraction des descripteurs pertinents à l'aide d'outils comme RDKit, et l'application de divers algorithmes de machine learning. Nous viserons à optimiser ces modèles par validation croisée et sélectionner les meilleurs modèles en fonction de tests statistiques significatifs. Un accent particulier sera mis sur l'utilisation de trois algorithmes différents en plus d'un modèle de base naïf pour garantir une évaluation complète et robuste.

1.4 Structure du Rapport

La section suivante, Configuration de l'environnement de travail, décrit les outils et technologies mis en place pour réaliser le projet. Elle détaille l'utilisation de GitHub pour le travail commun, Poetry pour la gestion des dépendances, ainsi que l'environnement Python et les bibliothèques utilisées. Cette partie se termine par une explication de la structuration du projet, garantissant une organisation efficace et claire des fichiers et répertoires.

La troisième section, Prétraitement et Gestion des Données, se concentre sur les étapes initiales de manipulation des données. Elle couvre le chargement des données, l'extraction des descripteurs moléculaires, la séparation des données en ensembles d'entraînement et de test. Ces étapes sont cruciales pour préparer les données de manière optimale avant l'application des algorithmes de machine learning.

Dans Implémentation et Analyse Descriptive des Algorithmes, le rapport présente les différents algorithmes de machine learning appliqués, en expliquant leurs spécificités, leur implémentation

et en offrant une analyse de leurs performances.

L'évaluation des modèles est documentée dans la cinquième partie du rapport, où les méthodes d'évaluation sont décrites et les résultats obtenus sont analysés. Cette section permet de comparer les performances des différents modèles et de choisir les plus efficaces.

Enfin, le rapport se termine par une Conclusion qui résume les principales découvertes du projet, évalue la réussite par rapport aux objectifs initiaux et propose des pistes pour des travaux futurs. Cette section permet de mettre en perspective les résultats obtenus et de réfléchir aux améliorations possibles.

2 Configuration de l'environnement de travail

Avant de plonger dans le développement du projet de Data Mining, une configuration appropriée de l'environnement de travail est essentielle pour garantir une collaboration efficace, une gestion de code fluide et une reproductibilité des expériences. Dans cette section, nous détaillerons les éléments clés de notre environnement de travail.

2.1 Gestion de Version avec GitHub

GitHub joue un rôle central dans la gestion de version de notre projet. Nous avons créé un référentiel dédié pour le suivi et le stockage des différentes versions de notre code source. Cela facilite la collaboration entre les membres de l'équipe, permettant des contributions parallèles, la gestion des branches, et la traçabilité des modifications apportées au code.

2.2 Gestion des Dépendances avec Poetry

La gestion des dépendances est crucial pour garantir que notre projet utilise les versions spécifiques des bibliothèques requises. Nous avons choisi Poetry comme gestionnaire de dépendances, offrant une manière simple et cohérente de déclarer, gérer et installer les bibliothèques Python nécessaires. Les détails sur les dépendances sont spécifiés dans le fichier `pyproject.toml`, assurant une reproductibilité entre les environnements.

2.3 Environnement Python et Bibliothèques

Nous travaillons avec Python comme langage principal pour notre projet. L'utilisation d'une multitude de bibliothèques nous servira à effectuer nos différentes opérations lors de ce projet :

- `os` : pour de la manipulation de fichiers.
- `torch` : pour utiliser les GPU pour accélérer les calculs.
- `joblib` : pour gérer efficacement les données sauvegardées.
- `matplotlib` : pour faire des graphiques.
- `pandas` : pour les bases de données.
- `sklearn` : pour ses fonctions d'évaluation de machine learning.
- `numpy` : pour les calculs et la compatibilité avec d'autres librairies.
- `optuna` : pour l'optimisation hyperparamétrique.
- `tqdm` : pour voir la progression lors de l'exécution.
- `rdkit` : pour l'informatique chimique notamment les SMILES [5].
- `seaborn` : pour la carte de chaleur basé sur `matplotlib`.

2.4 Structuration du Projet

Pour maintenir une organisation claire de notre code, nous avons adopté une structure de projet standard. Les différents modules et composants du projet sont regroupés de manière logique dans des répertoires spécifiques :

- `data` : qui contient notre fichier de données.
- `params` : qui contient nos sauvegardes de paramètres.
- `mpp/tuning` : pour nos fonctions sur les hyperparamètres.
- `mpp/models` : qui contient nos implémentations pour les différents models.
- `mpp/utils` : qui contient toutes les fonctions qui peuvent être utiles lors du projet.

Afin d'utiliser tout ce code de manière claire et efficace, nous avons le fichier notebook `results.ipynb` qui utilise et représente tous les résultats souhaités.

📁 data	backup 06/05/24 16:34:39	2 weeks ago
📁 mpp	code fini	13 minutes ago
📁 params	run data	4 days ago
📄 .gitignore	backup 06/05/24 16:34:39	2 weeks ago
📄 README.md	add readme	yesterday
📄 dm_project.pdf	backup 06/05/24 16:34:39	2 weeks ago
📄 poetry.lock	backup 06/05/24 16:34:39	2 weeks ago
📄 pyproject.toml	backup 06/05/24 16:34:39	2 weeks ago
📄 results.ipynb	code fini	13 minutes ago

Figure 3: Organisation et structure du dépôt GitHub du projet

3 Prétraitement et Gestion des Données

3.1 Chargement des données

Pour accéder à nos données, nous utilisons la librairie `pandas` qui possède la fonction `read_csv(...)` qui permet de lire un fichier de données. Nous lui passerons en argument le chemin vers nos données qui se trouvent dans notre dossier `data` avant de supprimer les lignes qui possèdent des données manquantes.

```
self.data = pd.read_csv(self.file_path)
self.data.dropna(inplace=True)
```

3.2 Extraction des données

Maintenant que nous avons accès à notre base de données, nous pouvons constater que celle-ci possède bien quatre colonnes comme décrit dans l'énoncé du problème :

- **SMILES** : Simplified Molecular Input Line Entry System, est une représentation simplifiée d'une molécule.
- **IC50** : une mesure utilisée en pharmacie.
- **num_atoms** : qui indique le nombre d'atomes de la molécule.
- **logP** : une mesure qui indique comment une molécule interagit avec certains solvants.

À l'aide de la librairie `rdkit`, nous pouvons directement analyser une molécule depuis sa représentation SMILES. C'est pourquoi, nous allons pouvoir compter le nombre d'atomes, de type d'atomes, de liaisons et la masse molaire dans chaque molécule afin d'ajouter ces informations dans notre tableau de données.

```
features = []
for smile in self.data['SMILES']:
    mol = Chem.MolFromSmiles(smile)
    features.append({
        'MolWt': Descriptors.MolWt(mol),
        'NumAtoms': Descriptors.HeavyAtomCount(mol),
        'NumBonds': mol.GetNumBonds(),
        'NumC': smile.count('C'),
        'NumO': smile.count('O'),
        'NumN': smile.count('N'),
        'NumCl': smile.count('Cl'),
        'NumF': smile.count('F'),
        'NumBr': smile.count('Br'),
        'NumI': smile.count('I')
    })
features_df = pd.DataFrame(features)
self.data = pd.concat([self.data.reset_index(drop=True),
                     features_df.reset_index(drop=True)], axis=1)
```

Étant donné que nous avons extrait toutes les informations nécessaires, nous allons supprimer la colonne SMILES de notre tableau.

```
self.data.drop(columns=['SMILES'], inplace=True)
```

3.3 Séparation des données

Pour séparer nos données en deux parties, de testes et d'entraînements, nous utiliserons la fonction `train_test_split(...)` de la librairie `sklearn`. Pour cela, nous extrayons d'abord les

colonnes du pic50 et du logP afin de définir :

- X : tableau de nos paramètres
- y_pic50 : la colonnes des pIC50
- y_logP : la colonnes des logP

En plus de cette séparation, il faut définir la quantité de données utilisées pour l'entraînement. Dans notre cas, nous appliquerons un taux de 80% d'entraînement pour 20% de données de test. Afin d'avoir les mêmes données testées pour le pIC50 et le logP on utilise un nombre aléatoire qui définit la manière de sélectionner notre échantillon.

```
X = self.data.drop(['pIC50', 'logP'], axis=1)
y_pic50 = self.data['pIC50']
y_logP = self.data['logP']

X_train, X_test, y_train_pic50, y_test_pic50 = train_test_split(X, y_pic50,
                                                               test_size=test_size, random_state=random_state)

X_train, X_test, y_train_logP, y_test_logP = train_test_split(X, y_logP,
                                                               test_size=test_size, random_state=random_state)
```

3.4 Corrélation

Suite à l'extraction des données [3.2], nous avons effectué une matrice de corrélation afin de voir s'il existait un lien entre certains paramètres de notre tableau :

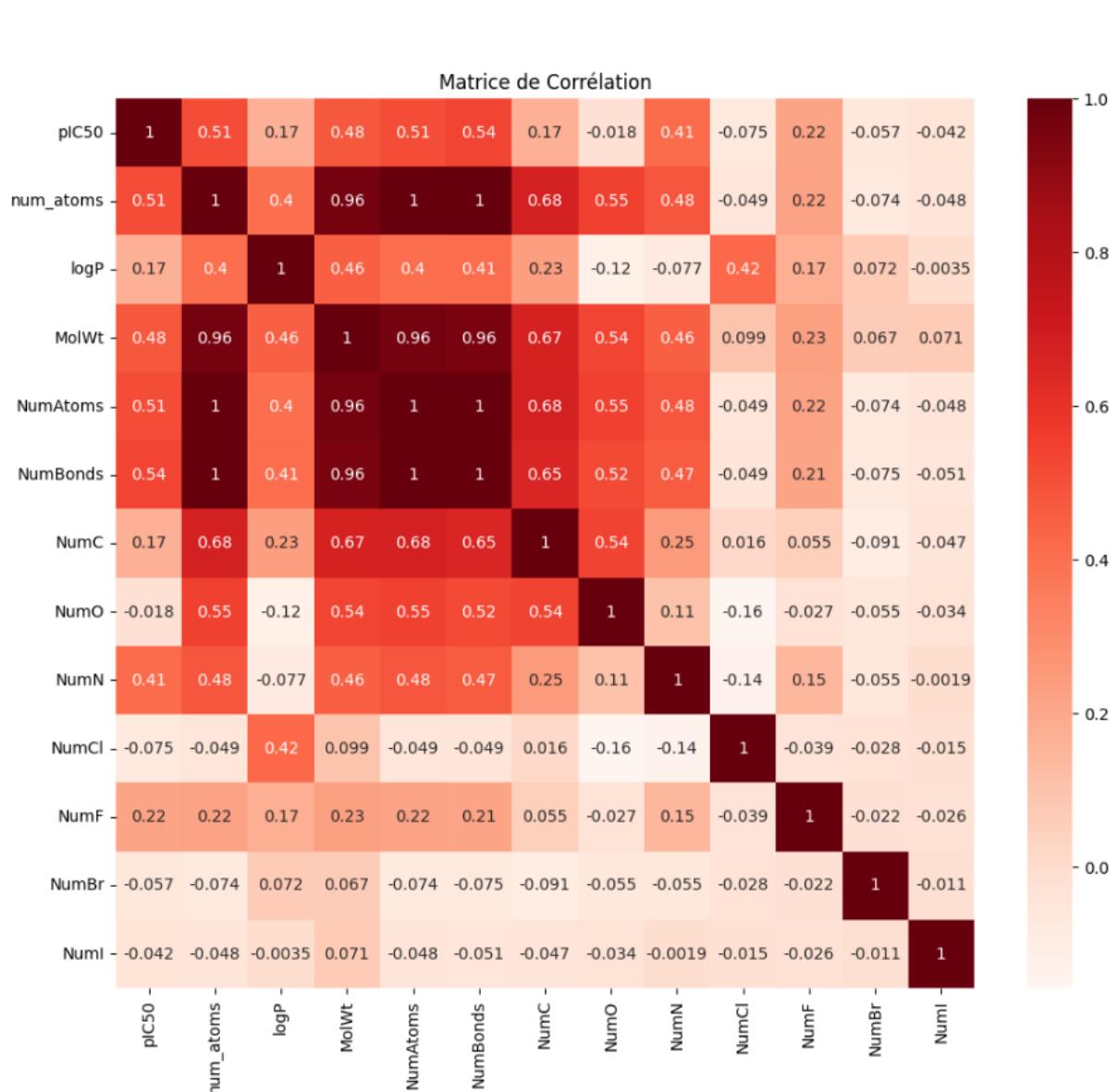


Figure 4: Matrice de corrélation pour nos données

Nous pouvons constater certaines relations fortes notamment entre le nombre d'atomes et de liaisons, cependant, les paramètres qui nous intéressent sont le pIC50 et le logP. C'est pourquoi nous avons opté pour une représentation en colonne. Lorsqu'une corrélation se rapproche de 1, elle indique une correspondance parfaite (si une variable augmente, l'autre augmente proportionnellement), -1 indique une correspondance inversément parfaite (si une variable augmente, l'autre diminue proportionnellement) et enfin une corrélation de 0 indique qu'il n'existe aucun lien entre celles-ci. Ainsi, sur notre représentation, nous avons ajouté un seuil (délimité en rouge) au niveau des valeurs -0.1 et 0.1. Si une colonnes se trouve entre ces seuils, il indique une corrélation si faible qu'elle en devient négligeable.

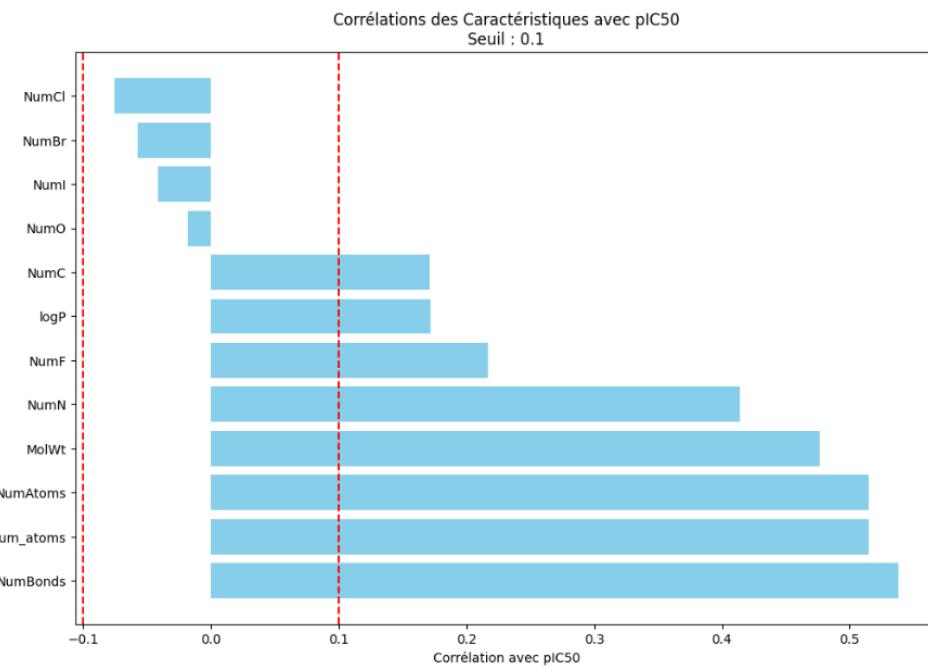


Figure 5: Corrélation des paramètres avec le pIC50

Pour le pIC50, nous constatons donc que sa corrélation la plus prononcée se trouve avec le nombre de liaisons dans la molécule, suivi de près par le nombre d'atomes et la masse molaire. Ces trois éléments ont aussi une corrélation entre elles très élevée ce qui se retrouve dans ce graphique.

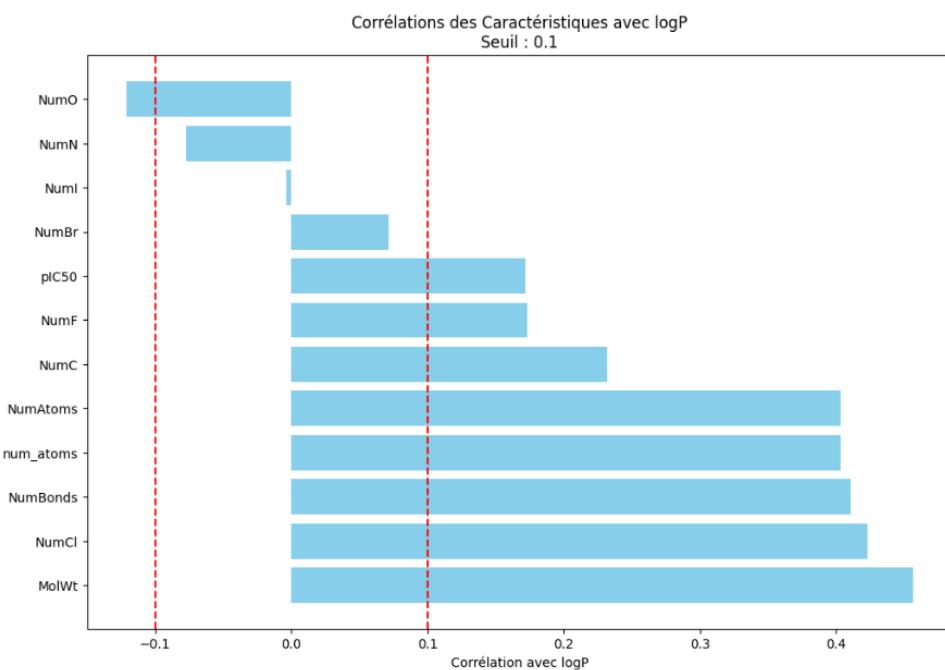


Figure 6: Corrélation des paramètres avec le logP

Pour ce qui est du logP, nous pouvons d'abord remarquer une faible corrélation inverse avec le nombre d'atomes **O**. Cependant, les valeurs pertinentes pour la corrélation avec le logP seront celles de la masse molaire, le nombre d'atomes **Cl** et juste derrière le nombre de liaisons.

On note donc une différence entre nos deux paramètres qui se reflétera aussi dans la suite de notre analyse.

4 Implémentation et Analyse Descriptive des Algorithmes

Le processus de formation et d'évaluation des modèles d'apprentissage automatique pour prédire nos données spécifiques a nécessité un examen et une sélection minutieuse des modèles, suivis de procédures d'évaluation rigoureuses. Ainsi, il sera nécessaire dans un premier temps de présenter tous les modèles utilisés dans le cadre de ce projet, c'est-à-dire notre baseline ainsi que les trois autres modèles.

4.1 Baseline K-nearest neighbors (k-NN)

Dans les problèmes de régression, l'algorithme des K-voisins les plus proches (k-NN) [8] est employé avec une légère modification par rapport à son utilisation dans les tâches de classification. Dans la régression, au lieu de prédire une classe discrète, k-NN prend la moyenne des valeurs cibles des k plus proches voisins pour faire une prédition continue. Cette approche permet au k-NN de traiter des variables cibles continues.

Nous avons choisi le modèle k-NN comme baseline, en partie parce qu'il est reconnu pour sa simplicité et son efficacité en apprentissage supervisé. Il se base sur la proximité entre les données. Afin de déterminer la sortie d'une nouvelle instance, k-NN localise les k points de données les plus proches dans l'ensemble d'apprentissage et effectue en conséquent une prédition en se fondant sur leurs étiquettes.

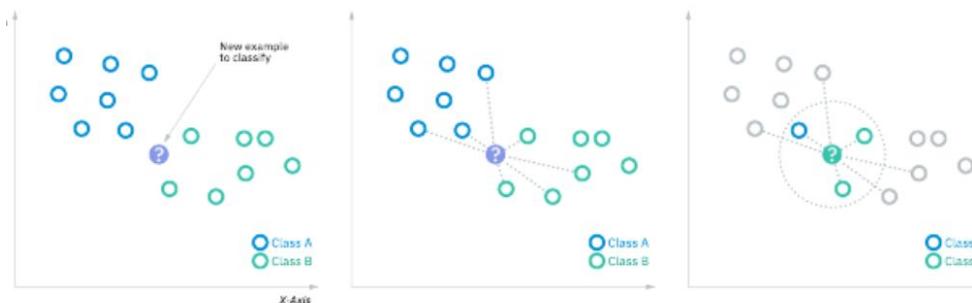


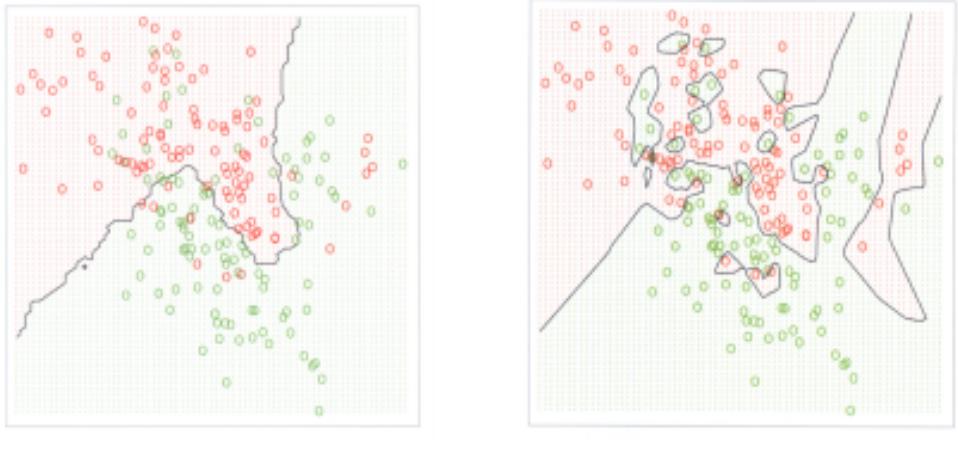
Figure 7: Aperçu de l'algorithme KNN

4.1.1 Fonctionnement de k-NN

Contrairement aux méthodes qui construisent un modèle explicite, k-NN fonctionne comme un algorithme "paresseux", mémorisant directement l'ensemble d'apprentissage. Pour prédire la sortie d'une nouvelle donnée, il identifie les k points les plus similaires dans cet ensemble, en utilisant souvent la distance euclidienne pour mesurer cette similarité [4.1.3].

4.1.2 Choix du Paramètre k

Le paramètre k est déterminant pour l'algorithme [2]. Un petit k peut conduire à un modèle très flexible avec une variance élevée, conduisant à l'augmentation de la sensibilité du bruit dans nos données. À l'inverse un paramètre k trop élevée peut masquer les structures locales des données, ce qui peut affecter grandement la précisions de nos prédictions (étant donné que la variance globale sera plus faible).



$K = 15$

$K = 1$

Figure 8: Effet de K sur les frontières de classes

4.1.3 Métriques de Distance dans k-NN

L'aspect essentiel du k-NN est la détermination de la distance entre les points de données. La mesure de distance la plus couramment utilisée est la distance euclidienne [6] (celle que nous avons utilisée ici), qui mesure la distance en ligne droite entre deux points dans l'espace des caractéristiques. En calculant les distances entre un point de données choisi et ses voisins, le k-NN identifie les k voisins les plus proches sur la base de leurs valeurs de caractéristiques.

$$d(x, y) = \sqrt{\sum_{i=1}^n (x_i - y_i)^2} \quad (1)$$

Il existe également d'autres distances, telles que la distance de Manhattan ou encore la distance de Minkowski qui ont chacune leur avantage et inconvénients. Ainsi, chacune de ces distances peuvent être plus ou moins adaptée en fonction de la nature et de la dimensionnalité des données traitées, ainsi que de la spécificité de la tâche d'apprentissage.

4.1.4 Algorithme

Voici un pseudo-code de l'algorithme k-NN car, dans notre code, nous utilisons la librairie `sklearn` qui possède une méthode pour l'appliquer directement :

Algorithm 1 k-NN Regression Algorithm

```
1: Input: Dataset  $D$ , distance function  $d$ , integer  $K$ , new observation  $X$ 
2: Output: Predicted value  $y_{\text{pred}}$ 
3:
4: function KNN_REGRESSION( $D$ ,  $d$ ,  $K$ ,  $X$ )
5:   // Compute distances from  $X$  to all observations in  $D$ 
6:   distances  $\leftarrow \emptyset$ 
7:   for each observation in  $D$  do
8:     distance  $\leftarrow d(X, \text{observation})$ 
9:     add (observation, distance) to distances
10:  end for
11:
12: // Select the top  $K$  nearest observations
13: sort distances by distance
14:  $K\_nearest \leftarrow$  first  $K$  elements of distances
15:
16: // Perform regression: compute predicted value  $y_{\text{pred}}$ 
17:  $y_{\text{pred}} \leftarrow$  average of  $y$  values in  $K\_nearest$ 
18:
19: return  $y_{\text{pred}}$ 
20: end function
```

4.1.5 Avantages et Limitations

- **Avantages :**

- Facilité de compréhension et d'implémentation.
- Performant pour des ensembles de données de taille modeste.
- Peut être utilisé pour des problèmes de classification et de régression, ce qui lui garantit une bonne flexibilité.

- **Limitations :**

- Les performances se dégradent avec l'accroissement de la dimensionnalité.
- Sensibilité aux caractéristiques superflues et à l'échelle des données.
- Coût élevé en temps de calcul et en mémoire pour les grands ensembles de données.

4.2 Random Forest Regression Models

Random Forest [4] est un puissant algorithme d'apprentissage d'ensemble qui combine plusieurs arbres de décision pour faire des prédictions. Il est largement utilisé pour les tâches de classification et de régression en raison de sa robustesse et de sa capacité à traiter des ensembles de données complexes.

Cet algorithme consiste en une collection d'arbres de décision, où chaque arbre est formé sur un sous-ensemble différent de données. L'idée de base de Random Forest est d'introduire le hasard de deux manières principales : l'échantillonnage aléatoire des données de formation et la sélection aléatoire des caractéristiques. Ces deux sources d'aléa contribuent à la diversité des arbres de l'ensemble et permettent de réduire l'adaptation excessive.

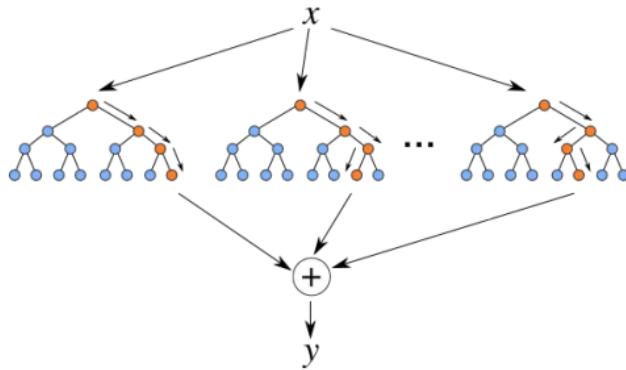


Figure 9: Visualisation de l'algorithme Random Forest

4.2.1 Fonctionnement de Random Forest Regression

Contrairement aux méthodes qui construisent un modèle unique, le modèle de régression Random Forest fonctionne comme un ensemble d'arbres de décision. En effet, chaque arbre est entraîné sur un sous-ensemble aléatoire de l'ensemble d'apprentissage et utilise une sélection aléatoire de variables pour diviser les données à chaque noeud.

Afin de prédire la sortie d'une nouvelle donnée en régression, chaque arbre dans la forêt prédit une valeur. Ensuite, l'ensemble des prédictions de tous les arbres est généralement moyenné pour obtenir la prédiction finale. Ainsi :

Pour chaque arbre de décision, nommé $h_i(x)$, il y'a une prédiction initiale d'une valeur \hat{y}_i pour une nouvelle instance d'entrée x . On répète l'étape n fois jusqu'à avoir parcouru l'ensemble des arbres de décision, avec n représentant le nombre d'arbre de décision. De ce fait, la prédiction finale du modèle Random Forest pour x est obtenue en moyennant les prédictions de l'ensemble des arbres de décision :

$$F(x) = \frac{1}{N} \sum_{i=1}^N h_i(x)$$

Chaque arbre de décision est alors construit indépendamment en utilisant diverses méthodes tels que l'indice de Gini (classification) ou encore des critères de réduction de la variance (régression). Cette approche permet à ce modèle de bien généraliser tout en évitant le surapprentissage, grâce à la diversité introduite par les sous-ensembles aléatoires d'apprentissage et de variabiles.

4.2.2 Structure et Paramètres clés

Voici les paramètres permettant de constituer et de configurer un modèle de Random Forest, sur lesquels nous nous sommes appuyer :

- **n_estimators** : Ce paramètre représente le nombre d'arbres dans la forêt. Un grand nombre d'arbres peut améliorer la performance prédictive et peut également augmenter le temps de calcul ainsi que l'utilisation de la mémoire.

- **max_depth** : Il s'agit de la profondeur maximale des arbres. Plus la profondeur est grande,

plus les différents arbres du modèles auront la capacité de modéliser des relations plus complexes, mais au prix d'un potentiel surapprentissage si ces derniers deviennent trop spécifiques aux données d'entraînement.

Il y'a également d'autres paramètres, comme `min_samples_split` ou encore `max_features` qui sont responsables respectivement du nombre d'échantillons requis pour diviser un nœud, et du nombre de caractéristiques à considérer lors de la recherche de la meilleure division.

4.2.3 Algorithme

Voici un pseudo-code de l'algorithme pour le Random Forest car, dans notre code, nous utilisons la librairie `sklearn` qui possède une méthode pour l'appliquer directement :

Algorithm 2 Random Forest Regression Algorithm

```
1: Input: Dataset  $D$ , number of trees  $n\_estimators$ , maximum depth  $max\_depth$ , minimum samples to split  $min\_samples\_split$ , minimum samples per leaf  $min\_samples\_leaf$ , maximum features  $max\_features$ , bootstrap sampling  $bootstrap$ , new observation  $X\_new$ 
2: Output: Predicted value  $y_{pred}$ 
3:
4: function RANDOMFORESTREGRESSION( $D, n\_estimators, max\_depth, min\_samples\_split,$ 
 $min\_samples\_leaf, max\_features, bootstrap, X\_new$ )
5:   Initialize an empty list  $forest$ 
6:   for  $i = 1$  to  $n\_estimators$  do
7:     if  $bootstrap$  is True then
8:       Create bootstrap sample  $(X\_sample, y\_sample)$  from  $D$ 
9:     else
10:       $X\_sample, y\_sample \leftarrow D$ 
11:    end if
12:    Initialize a DecisionTreeRegressor with  $max\_depth$ ,  $min\_samples\_split$ ,
 $min\_samples\_leaf$ ,  $max\_features$ 
13:    Fit the tree on  $(X\_sample, y\_sample)$ 
14:    Append the tree to  $forest$ 
15:  end for
16:
17:
18: function PREDICT( $X\_new$ )
19:   Initialize an empty list  $predictions$ 
20:   for each tree in  $forest$  do
21:     Predict the value for  $X\_new$  using the tree
22:     Append the predicted value to  $predictions$ 
23:   end for
24:    $y_{pred} \leftarrow$  average of  $predictions$ 
25:   return  $y_{pred}$ 
26: end function
27: end function
```

4.2.4 Avantages et Limitations

- **Avantages :**

- Capable de traiter des ensembles de données à haute dimension comportant un grand nombre de caractéristiques.
- Résistances non négligeable aux valeurs aberrantes et au bruit dans les données.
- Fourni des prédictions précises en raison de sa capacité à combiner plusieurs arbres de décision et à réduire le surajustement.

- **Limitations :**

- Ne peuvent donner de bons résultats sur des ensembles de données déséquilibrés si la classe minoritaire est sous-représentée.
- Plus difficile à interpréter que des modèles plus simples tel que la régression logistique.
- Coût élevé en temps de calcul quand elle dispose d'un grand nombre d'arbres et de caractéristiques.

4.3 Gradient Boosting Models

Les modèles de Gradient Boosting (GBM) [7] sont des techniques d'ensemble qui combinent plusieurs arbres de décision pour former un modèle puissant et robuste. En régression, GBM construit des arbres de décision successifs, chaque nouvel arbre corrigeant les erreurs des arbres précédents, ce qui permet d'améliorer progressivement les prédictions.

Nous avons choisi le Gradient Boosting comme modèle avancé, car il est reconnu pour sa capacité à capturer des relations complexes dans les données et à fournir des performances élevées sur une variété de tâches de régression. Le GBM itère sur les arbres pour minimiser les résidus, ce qui en fait une méthode efficace pour modéliser les relations non linéaires.

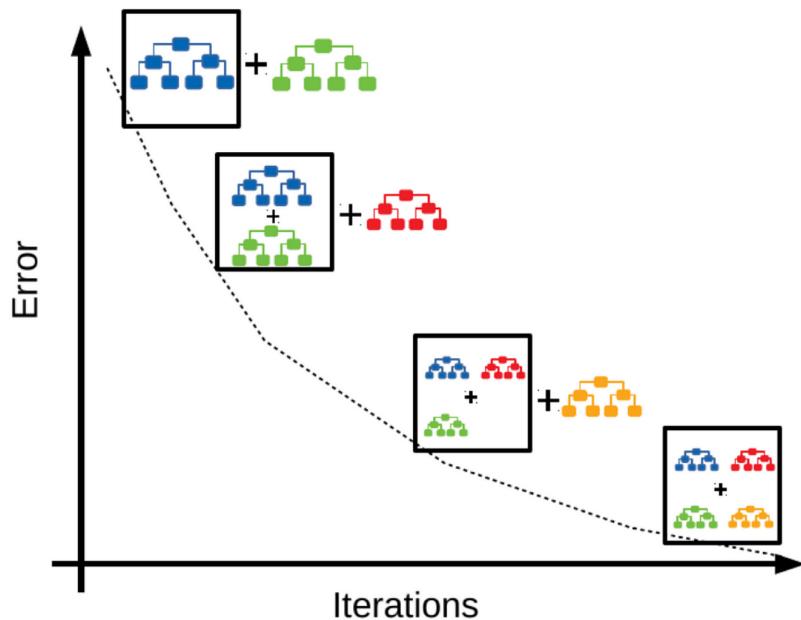


Figure 10: Fonctionnement du gradient boosting

4.3.1 Fonctionnement du Gradient Boosting

Le Gradient Boosting est une méthode séquentielle qui construit des modèles faibles (souvent des arbres de décision) de manière additive. Chaque nouvel arbre est ajusté sur les erreurs des arbres précédents de sorte à minimiser la fonction de perte $L(y, F(x))$, qui mesure l'écart entre les valeurs cibles y et les prédictions du modèle $F(x)$. Plutôt que de minimiser la perte en ajustant tous les paramètres du modèle en une fois, le Gradient Boosting procède de manière additive. À chaque étape m , un nouvel arbre est construit pour réduire les résidus.

On commence par un modèle initial $F_0(x)$, souvent une moyenne des valeurs cibles y pour de la régression : $F_0(x) = \text{moyenne}(y)$. Pour chaque itération m , les résidus sont calculés en prenant le gradient de la fonction de perte par rapport aux prédictions actuelles $F_{m-1}(x)$. Il s'agit d'un gradient représentant la direction d'ajustement pour améliorer le modèle. Pour la régression, les résidus sont simplement les erreurs entre les valeurs observées et les prédictions :

$$r_i^{(m)} = - \left[\frac{\partial L(y_i, F_{m-1}(x_i))}{\partial F_{m-1}(x_i)} \right]$$

Un nouvel arbre de régression $h_m(x)$ est ajusté selon $(x_i, r_i^{(m)})$, où x_i est l'ensemble des caractéristiques et $r_i^{(m)}$ les résidus calculés, pour essayer de mieux prédire ces erreurs. Le modèle est mis à jour en ajoutant une fraction des prédictions de l'arbre nouvellement construit, modulée par un taux d'apprentissage η .

$$F_m(x) = F_{m-1}(x) + \eta h_m(x)$$

On répète ces étapes pour un nombre fixé d'itérations ou jusqu'à ce que les résidus soient minimisés.

4.3.2 Paramètres Clés

Le Gradient Boosting est contrôlé par une multitude d'hyperparamètres. De notre côté, nous avons opté sur la variation de trois d'entre eux :

- `n_estimators` : qui représente le nombre d'étapes de boosting à effectuer. L'augmentation du nombre d'itérations est assez résistante au surajustement, donc un grand nombre entraîne généralement de meilleures performances.
- `learning_rate` : Le taux d'apprentissage réduit la contribution de chaque arbre, permettant une correction plus graduelle et souvent plus stable des erreurs. Il existe un compromis entre `learning_rate` et `n_estimators`.
- `max_depth` : La profondeur maximale limite le nombre de noeuds dans nos arbres. Ajuster ce paramètre permet d'obtenir de meilleures performances. En limitant le nombre de noeuds, nous le rendons moins susceptible de capter les fluctuations spécifiques et donc obtenons une meilleure généralisation.

4.3.3 Algorithme

Voici un pseudo-code de l'algorithme du Gradient Boosting car, dans notre code, nous utilisons la librairie `sklearn` qui possède une méthode pour l'appliquer directement :

Algorithm 3 Gradient Boosting Regression Algorithm

```
1: Input: Dataset  $D$ , learning rate  $\eta$ , nbr iterations  $N$ , loss function  $L$ , valeurs cibles  $y$ , profondeur max  $d$ 
2: Output: Predicted model  $F(x)$ 
3:
4: function GRADIENT BOOSTING REGRESSION( $D, \eta, N, L, y, d$ )
5:   Initialize model  $F_0(x) \leftarrow moy(y)$ 
6:   for each iteration  $m = 1$  to  $N$  do
7:     // Compute the residuals
8:      $r_i \leftarrow -\frac{\partial L(y_i, F_{m-1}(x_i))}{\partial F_{m-1}(x_i)}$  for all  $i$ 
9:
10:    // Fit a new tree  $h_m$  to the residuals
11:    Fit  $h_m$  to  $(x_i, r_i)$  with max_depth= $d$ 
12:
13:    // Update the model
14:     $F_m(x) \leftarrow F_{m-1}(x) + \eta h_m(x)$ 
15:  end for
16:
17:  return  $F_N(x)$ 
18: end function
```

4.3.4 Avantages et Limitations

- **Avantages :**

- Réduit efficacement les erreurs de prédiction.
- Flexibilité pour divers types de données et tâches, notamment la régression, la classification binaire, et la classification multi-classes.
- Robuste face aux jeux de données déséquilibrés grâce à l'ajustement progressif.
- Capture les interactions complèxes entre les caractéristiques de manière additive.
(résidus des arbres précédents)
- Moins influencé par les valeurs aberrantes

- **Limitations :**

- Complexité computationnelle : coûteux en termes de calcul et de mémoire.
- La performance optimale nécessite un réglage fin de nombreux hyperparamètres.
- Le modèle peut être sensible au bruit, surtout si la profondeur des arbres n'est pas limitée.
- Plus difficiles à interpréter que les modèles simples.
- Instabilité : les performances peuvent varier considérablement en fonction des choix des paramètres.

4.4 Neural Networks Models (NN)

Les réseaux de neurones sont des modèles d'apprentissage profond qui imitent le fonctionnement des neurones dans le cerveau humain. Ils sont composés de plusieurs couches de neurones (unités de traitement) qui transforment les entrées en sorties via des opérations mathématiques.

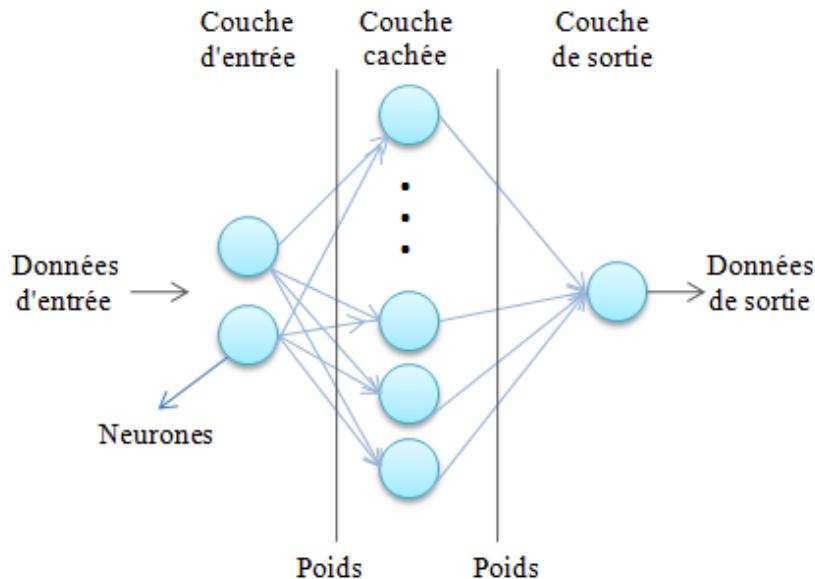


Figure 11: Exemple d'un réseaux de neuronne

Nous avons choisi le réseaux de neurones comme troisième et dernier modèle en raison de sa capacité impressionnante à modéliser des relations complexes et non linéaires entre les variables. Grâce à leur architecture flexible et adaptable, ils peuvent automatiquement apprendre et extraire des caractéristiques pertinentes à partir des données brutes, offrant ainsi des prédictions plus précises que les modèles traditionnels. De plus, les réseaux de neurones bénéficient d'outils et de bibliothèques avancés, facilitant leur développement, optimisation et déploiement en production, ce qui les rend idéaux pour des applications nécessitant une grande précision et robustesse. Dans le cadre de ce projet, nous avons choisi de l'implémenter manuellement afin de s'assurer un maximum de son efficacité et en prenant en compte tous les paramètres nécessaire dont il dispose.

4.4.1 Fonctionnement du Neuronal Networks

Architecture

Comme dit plus haut, le réseaux de neurones permet d'imiter le fonctionnement des neurones dans le cerveau humain. En particulier, l'architecture d'un tel modèle se décrit comme ceci :

- **Couches d'Entrée :** Il s'agit de la couche d'input qui va recevoir nos données/caractéristiques X pour l'observation.
- **Couches Cachées :** Une ou plusieurs couches cachées sont composées de neurones qui appliquent des transformations non linéaires aux entrées. Chaque neurone dans une couche cachée calcule une somme pondérée de ses entrées, applique une fonction d'activation et transmet le résultat à la couche suivante.
- **Couche de Sortie :** C'est la dernière couche du réseaux de neurones, où nous recevons les prédictions \hat{y} du réseau.

Propagation

Pour l'implémentation de notre modèle, nous avons opté pour une Propagation Avant (Forward Propagation). En effet, les entrées passent à travers le réseaux couche par couche, et donc mathématiquement, pour chaque neuronne j dans une couche l , la sortie $a_j^{(l)}$ est calculé comme suit :

$$a_j^{(l)} = g^{(l)} \left(\sum_i w_{ij}^{(l)} a_i^{(l-1)} + b_j^{(l)} \right)$$

- $w_{ij}^{(l)}$ sont les poids entre les neurones de la couche $l - 1$ et la couche l
- $b_j^{(l)}$ est le biais du neurone j dans la couche l .
- $g^{(l)}$ est la fonction d'activation de la couche l .

Fonction d'activation

La fonction d'activation est un élément essentiel pour garantir le bon fonctionnement d'un réseau de neurones, car elle est nécessaire pour introduire de la non-linéarité dans le modèle. Cette non-linéarité est cruciale car elle permet d'ajouter de la complexité aux relations que le réseau peut apprendre. L'utilisation de fonctions d'activation non linéaires est préférée par rapport aux fonctions linéaires, car ces dernières se limiteraient à des combinaisons linéaires des entrées sans être capables de saisir des nuances plus subtiles dans les données.

Dans notre réseaux, nous avons fait le choix d'utiliser la fonction ReLU [12] pour les couches intermédiaires de notre réseaux car c'est une pratique courante pour ces neurones, elle a l'avantage d'être efficace computationnellement pour les valeurs négatif et pour le calcul du gradiant lors de la rétropropagation

Early Stopping

Le Early Stopping est une technique de régularisation utilisée lors de l'entraînement de modèles d'apprentissage automatique, en particulier dans les réseaux de neurones profonds. L'objectif est d'arrêter l'entraînement avant que le modèle ne commence à surapprendre, c'est-à-dire à apprendre le bruit spécifique au jeu de données d'entraînement au détriment de sa capacité à généraliser de nouvelles données.

En effet, au fur et à mesure que l'entraînement progresse, la performance sur les données d'entraînement continue généralement de s'améliorer. Cependant, si le modèle commence à surapprendre, sa performance sur le jeu de données de validation commencera à se détériorer.

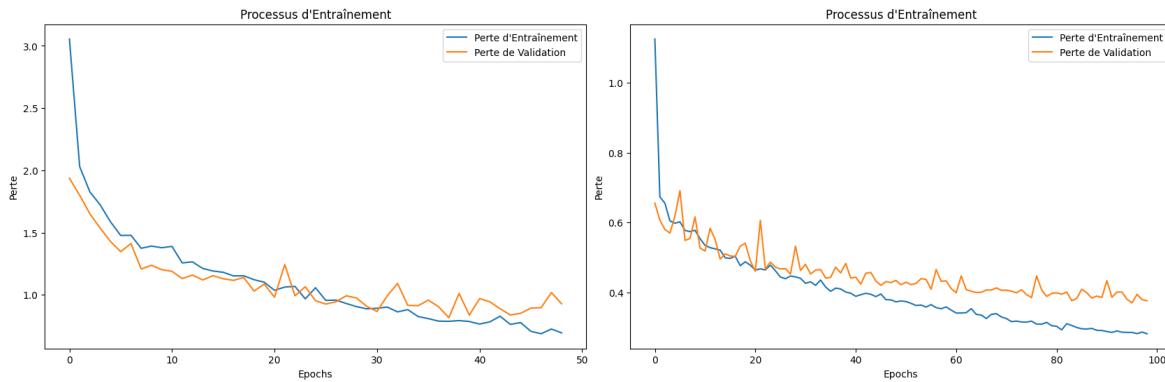


Figure 12: Exemple de perte d’entraînement et de validation : pIC50 vs logP

L’avantage majeur du Early Stopping est sa simplicité, son efficacité, et ne nécessitant aucun ajustement supplémentaire des hyperparamètres du modèle, contrairement à d’autres techniques de régularisation comme la régularisation L1 ou L2 (bien que non utilisé au sein de ce projet). Cependant, le choix du moment optimal pour arrêter l’entraînement peut dépendre de nombreux facteurs, y compris la variabilité des performances sur le jeu de données de validation, ce qui peut nécessiter une attention particulière et une expérience plus affiner.

4.4.2 Importance des Paramètres

L’importance de la configuration des paramètres dans un réseau de neurones est cruciale, car elle détermine non seulement la vitesse et l’efficacité de l’apprentissage, mais aussi la capacité du modèle à généraliser correctement de nouvelles données et à éviter les problèmes de sous-ajustement et de surajustement. Voici une description des paramètres clés et leur impact sur le modèle :

- **learning_rate** : C’est le taux d’apprentissage, qui va déterminer la taille des mises à jours des poids du modèle à chaque itération de l’entraînement. Un taux d’apprentissage élevé peut empêcher la convergence du modèle et entraîne des oscillations autour du minimum local, alors que un faible taux d’apprentissage peut ralentir l’entraînement, du fait qu’il a besoin de plus d’époques pour atteindre la convergence.
- **epochs** : Les époques désignent le nombre d’itérations à travers l’ensemble complet des données d’entraînement (pendant une époque, le modèle parcourt chaque point de données exactement une fois, en utilisant des batches pour mettre à jours les poids à chaque itérations). Peu d’époques conduisent à un underfitting (sous-ajustement), tandis que trop d’époques peut à l’inverse entraîner un overfitting (surajustement).
- **batch_size** : Déterminent le nombre d’échantillons de données traités simultanément par le modèle à chaque itération d’une époque. Un nombre trop petit de batches implique plus de mises à jours des poids (convergence plus stable et plus lente), tandis que c’est l’inverse quand ce dernier est trop élevé.
- **hidden_layers** : Constitue les couches cachées pour capturer les relations complexes et caractéristiques non linéaires dans les données. Peu de couche implique la limitation du modèle dans les captures de ces relations, tandis que trop de couche peut augmenter le risque d’overfitting et la difficulté à entraîner.
- **hidden_sizes** : La taille des couches. Une taille trop petite peut réduire la capacité d’apprentissage, tandis qu’une taille trop élevée augmente la complexité computationnelle ainsi que le risque de surapprentissage.

- **patience** : C'est le nombre d'époques pendant lesquelles l'algorithme continue de s'entraîner sans amélioration de la performance de validation avant de déclencher un arrêt anticipé. Ces avantages et inconvénients découlent des autres paramètres.

4.4.3 Explication du Code

Premièrement, on commence par normalisés nos données d'entraînement et de validation en utilisant la méthode `StandardScaler` de la librairie `sklearn` afin que les caractéristiques aient une moyenne nulle et un écart type unitaire :

```
X_train = self.scaler.fit_transform(X_train)
X_val = self.scaler.transform(X_val)
```

Ensuite, nous transformons l'ensemble de ces données en objet `TensorDataset` de la librairie `torch` qui seront ensuite utilisés afin de créer des chargeur de données (`DataLoader`). Cela va nous permettre de gérer les batches durant l'entraînement.

```
train_dataset = TensorDataset(torch.tensor(X_train, dtype=torch.float32),
                             torch.tensor(y_train.values, dtype=torch.float32))
val_dataset = TensorDataset(torch.tensor(X_val, dtype=torch.float32),
                           torch.tensor(y_val.values, dtype=torch.float32))

train_loader = DataLoader(train_dataset, batch_size=self.batch_size, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=self.batch_size, shuffle=False)
```

Après cela, pour chaque époque, nous effectuons l'ensemble des batches de données d'entraînement en initialisant d'abord les gradients à zéro, puis en effectuant les prédictions ainsi que les opérations nécessaires avec le modèle, en passant par la rétropropagation pour calculer les gradients.

```
for epoch in tqdm(range(self.epochs), desc=f"Entraînement des Époques ({self.epochs})"):
    epoch_train_loss = 0
    self.model.train()
    for X_batch, y_batch in train_loader:
        self.optimizer.zero_grad()
        outputs = self.model(X_batch)
        loss = self.criterion(outputs.squeeze(), y_batch)
        loss.backward()
        self.optimizer.step()
        epoch_train_loss += loss.item()
    epoch_train_loss /= len(train_loader)
    training_loss.append(epoch_train_loss)
```

Enfin, le modèle passe en mode évaluation et boucle sur les batches de données de validation afin de prédire les sorties et le calcul de la perte en conséquent. La perte de validation moyenne par époque est ensuite calculée et ajoutée à une liste `validation-loss` pour le suivi :

```
self.model.eval()
with torch.no_grad():
    for X_batch, y_batch in val_loader:
        outputs = self.model(X_batch)
        loss = self.criterion(outputs.squeeze(), y_batch)
        epoch_val_loss += loss.item()
    epoch_val_loss /= len(val_loader)
    validation_loss.append(epoch_val_loss)
```

Pour plus de détails sur le code, voir l'archive zip.

4.4.4 Avantages et Limitations

- **Avantages :**

- Les réseaux de neurones peuvent capturer des relations non linéaires complexes entre les variables, cela permet de les rendre adaptables pour une grande variété de problèmes.
- Ils peuvent être utilisés efficacement avec différents types de données et peuvent apprendre des caractéristiques pertinentes directement à partir des données brutes.
- Un bon réglage des hyperparamètres et disposer d'un nombre conséquent de données permet à ce modèle de fournir des performances de prédictions supérieures à la plupart des autres modèles classiques.

- **Limitations :**

- Un tel modèle est forcément coûteux en termes de ressources de calculs informatiques, nécessite souvent l'utilisation du GPU, et peut être chronophage
- Les décisions prises par les réseaux de neurones peuvent être difficiles à interpréter et à expliquer.
- Malgré tout, les hyperparamètres des réseaux de neurones doivent être réglés de la meilleure façon possible afin d'éviter l'entraînement de performances médiocres.

5 Évaluation des Modèles

Afin de pouvoir juger l'efficacité de nos différents modèles [4.], il faut pouvoir analyser les résultats fournis par ceux-ci. L'objectif principal est de minimiser les erreurs entre les prédictions du modèle et les valeurs réelles, ce qui permet de déterminer la qualité du modèle.

5.1 Créations des modèles

Comme nous souhaitons traiter quatre modèles différents, on va d'abord chercher à avoir le meilleur modèle de chaque. Pour trouver les paramètres optimaux de chacun de nos algorithmes, nous utilisons la librairie `optuna` qui facilite l'exploration efficace de l'espace des hyperparamètres. Dans notre fichier `mpp/tuning/tune.py`, nous effectuons ces opérations d'optimisations.

```
study = optuna.create_study(direction="minimize", pruner=optuna.pruners.MedianPruner())
```

Nous créons un objet `optuna` où l'objectif est de trouver l'ensemble d'hyperparamètres qui minimisent une métrique de coût [5.3.1] et où un essai en cours est arrêté dès qu'il n'est pas assez prometteur.

Ensuite, nous bouclons un nombre de fois défini (dans notre cas 50 fois) afin d'avoir une base assez large d'essais. On affiche aussi une barre de progression avec la librairie `tqdm`.

```
for _ in tqdm(range(n_trials), desc=f"Hyperparameter Tuning ({self.model_name})"):
    study.optimize(self.objective, n_trials=1)
```

Enfin, nous allons optimiser seulement les paramètres désirés avec une fouchette des valeurs à explorer.

5.1.1 K-Nearest Neighbors (KNN)

Comme ce modèle est assez simple, le seul paramètre que nous cherchons à optimiser est le nombre de voisins pris en compte.

```
if self.model_name == "KNN":
    n_neighbors = trial.suggest_int("n_neighbors", 1, 20)
    model = BaselineKNN(n_neighbors=n_neighbors)
```

On définit un intervalle entre un et vingt car, au-delà, nous lisserions trop nos résultats.

Après execution du programme pour l'estimation de notre paramètre du pIC50, on obtiendra une valeur de 3 pour l'optimiser alors que pour le logP la valeur optimale est de 4.

5.1.2 Random Forest

Pour la définition de notre random forest, nous varions deux hyperparamètres : le nombre d'arbre (`n_estimators`) et leur profondeur (`max_depth`).

```
elif self.model_name == "Random Forest":
    n_estimators = trial.suggest_int("n_estimators", 50, 200)
    max_depth = trial.suggest_int("max_depth", 10, 40)
    model = RandomForestModel(n_estimators=n_estimators, max_depth=max_depth)
```

Le nombre d'arbres par forêt est défini dans un intervalle entre 50 et 200 alors que la profondeur de nos arbres variera entre 10 et 40. La limitation de ces valeurs est assez importante d'un point de vu du temps de calculs.

Pour notre modèle évaluant le pIC50, les meilleurs paramètres retenus sont : 192 arbres dans notre forêt avec une profondeur de 27. Pour ce qui est du logP, les valeurs optimales valent : 127 arbres d'une profondeur de 24.

5.1.3 Gradient Boosting

De manière très similaire au Random Forest [5.1.2], pour le gradient boosting, nous modifions les paramètres pour le nombre d'arbre (`n_estimators`), leur profondeur (`max_depth`) et le taux d'apprentissage (`learning_rate`).

```
elif self.model_name == "Gradient Boosting":  
    n_estimators = trial.suggest_int("n_estimators", 50, 200)  
    learning_rate = trial.suggest_loguniform("learning_rate", 1e-5, 1e-1)  
    max_depth = trial.suggest_int("max_depth", 3, 20)  
    model = GradientBoostingModel(n_estimators=n_estimators,  
                                   learning_rate=learning_rate, max_depth=max_depth)
```

L'intervalle pour le nombre d'arbres est semblable à celui du Random Forest, cependant, la profondeur des arbres a été réduite entre 3 et 20 afin d'avoir de meilleurs arbres de généralisation. Quant au taux d'apprentissage, on le définit entre $10^{-5} = 0.00001$ et $10^{-1} = 0.1$ qui sont toutes les deux inférieures à 1 qui représenterait un taux d'apprentissage de 100%.

Les valeurs retenues pour le modèle de gradient boosting pour le pIC50 est de 196 arbres de profondeur 10 avec un taux d'apprentissage de 0.08798833734776297. Alors que pour le logP, nous avons un choix de 185 arbres de profondeur 9 avec un taux d'apprentissage de 0.0958597129072779

5.1.4 Neural Network

Le réseau de neurone étant très complexe et fait manuellement, nous pouvons varier un grand nombre de ses hyperparamètres afin de l'optimiser correctement :

- le taux d'apprentissage (`lr`)
- les époques (`epochs`)
- la taille des échantillons traités (`batch_size`)
- le nombre de couches cachées (`hidden_layer`)
- la taille des couches (`hidden_sizes`)
- la patience (`patience`)

```
elif self.model_name == "Neural Network":  
    lr = trial.suggest_loguniform("lr", 1e-5, 1e-1)  
    epochs = trial.suggest_int("epochs", 10, 100)  
    batch_size = trial.suggest_int("batch_size", 30, 150)  
    hidden_layers = trial.suggest_int("hidden_layers", 1, 3)  
    hidden_sizes = [trial.suggest_int(f"hidden_size_{i}", 50, 200) for i in  
                   range(hidden_layers)]  
    patience = trial.suggest_int("patience", 5, 20)  
    model = NeuralNetworkModel(input_size=self.X.shape[1], hidden_sizes=hidden_sizes,  
                               lr=lr,  
                               epochs=epochs, batch_size=batch_size, patience=patience)
```

Chacun de ces hyperparamètres à son intervalle et nous donnera des résultats pour le pIC50 et le logP :

- le taux d'apprentissage (`lr`) : entre $10^{-5} = 0.00001$ et $10^{-1} = 0.1$ comme pour le Gradient Boosting [5.1.3]. On a 0.0021666814143117574 pour le pIC50 et 0.0056863426884850195 pour le logP.
- les époques (`epochs`) : entre 10 et 100 pour itérer un nombre de fois assez significatif. Avec 87 époques pour le pIC50 et 99 pour le logP.
- la taille des échantillons traités (`batch_size`) : entre 30 et 150 et qui nous donne comme résultats, 121 pour le pIC50 et 96 pour le logP.
- le nombre de couches cachées (`hidden_layer`) : limité entre 1 et 3 afin de ne pas demander trop de temps de calculs et donnant logiquement comme valeur, un 3 pour le pIC50 et le logP.
- la taille des couches (`hidden_sizes`) : qui est entre 50 et 200 et qui peut avoir une taille différente à chaque couche qui sera de [54, 73, 115] pour le pIC50 et [135, 183, 164] pour le logP.
- la patience (`patience`) : qui va de 5 à 20 et qui est de 11 pour le modèle du pIC50 et 16 pour celui du logP.

5.2 Prédictions

Maintenant que nos modèles sont formés, il est possible de les essayer sur nos données de tests. La librairie `sklearn` offre la fonction `predict(...)` qui prend en argument les données d'entrées et nous donne leurs estimations. Pour le réseau de neurones que nous avons implémenté manuellement, nous évaluons nos tests directement depuis le modèles avec :

```
predictions = self.model(torch.tensor(X_test_scaled, dtype=torch.float32)).squeeze()  
predictions = predictions.numpy()
```

Nous avons, dès à présent, les valeurs estimées par nos modèles ainsi que les véritables valeurs depuis notre base de données.

5.2.1 Prédictions du pIC50

Pour nos estimations du pIC50, nous pouvons noter que sa valeur ne peut pas être négative. C'est pourquoi nous avons ajusté les valeurs prédites inférieures à 0 en leur donnant une valeur nulle. Ainsi, nos prédictions faisaient plus de sens et pouvaient directement être comparées aux résultats attendus. Pour pouvoir mieux visualiser ces différences, nous nous sommes penchés sur les 300 premiers échantillons de nos tests pour en faire des graphiques.

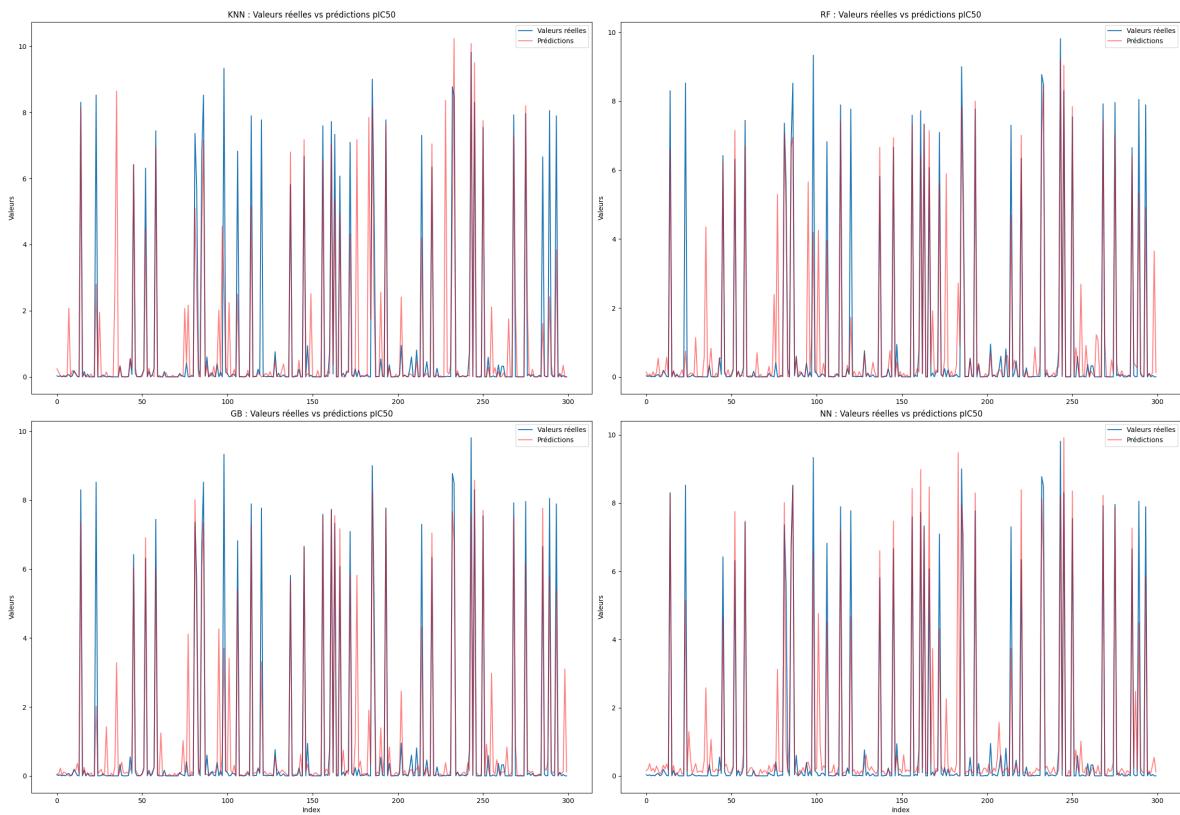


Figure 13: Comparaison des différents modèles pour pIC50

5.2.2 Prédictions du logP

Les valeurs du logP n'étant pas bornées, aucun ajustement n'était nécessaire et avons donc pu procéder directement à l'affichage de nos courbes sur des graphiques pour voir les différences avec nos données réelles.

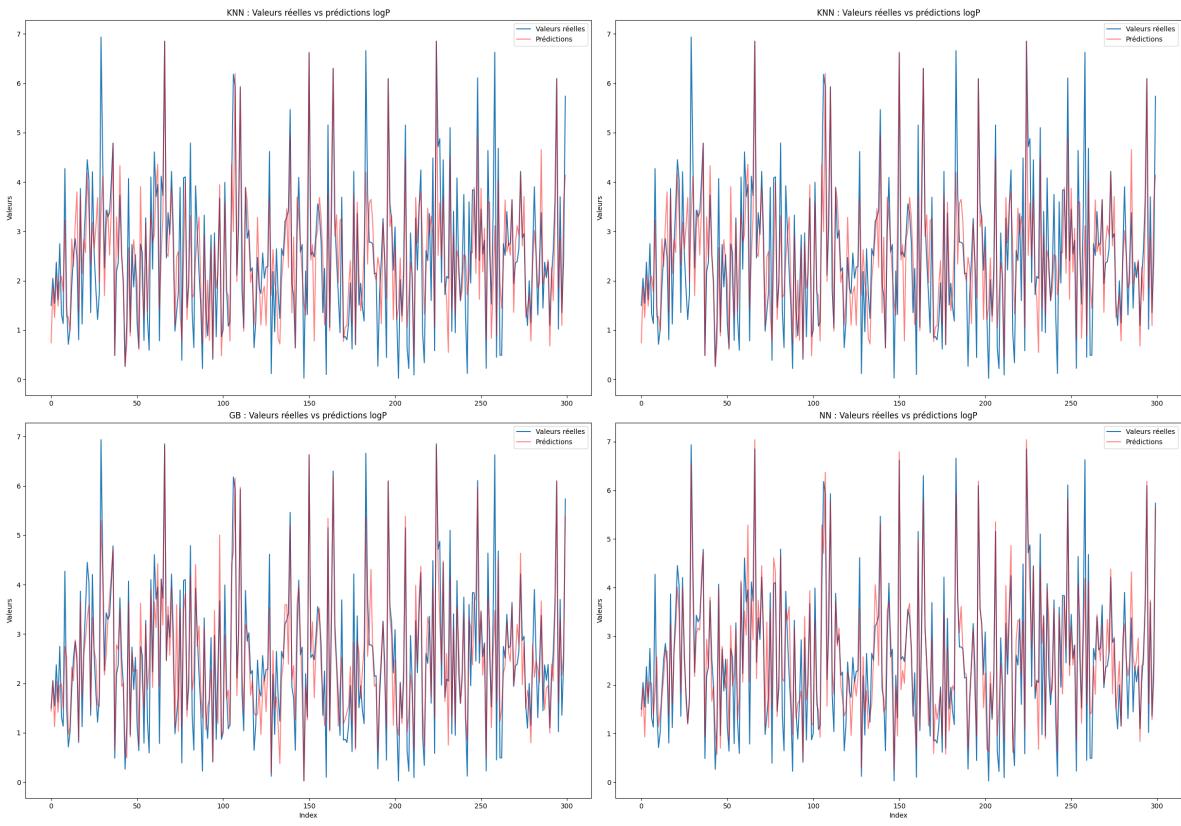


Figure 14: Comparaison des différents modèles pour logP

5.3 Calcul de l'erreur

Pour évaluer l'efficacité de nos modèles, nous prenons en compte les résultats obtenus dans nos prédictions [5.2] pour les comparer avec nos valeurs réelles.

5.3.1 Métrique d'évaluation

Il existe plusieurs manières d'évaluer des modèles mais toutes comparent les valeurs réelles y et les valeurs prédites \hat{y} de taille n . Dans le cas de la régression, nous avons opté pour différentes métriques [3] qui déterminent différents types d'erreurs. Afin d'avoir un avis assez critique sur les nôtres, nous avons opté pour quatre méthodes différentes où plus le résultat est proche de 0, mieux c'est :

Mean Squared Error (MSE) - Erreur Quadratique Moyenne

Elle mesure la moyenne des carrés des erreurs entre les valeurs prédites et les valeurs réelles.[11]

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

Root Mean Squared Error (RMSE) - Racine Carrée de l'Erreur Quadratique Moyenne

Elle se base sur le MSE mais est plus intuitive avec des valeurs comparables à nos valeurs de sorties.[13]

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

Mean Absolute Error (MAE) - Erreur Absolue Moyenne

La MAE [9] donne une idée de l'erreur absolue moyenne, ne pénalisant pas autant les grandes erreurs comme le fait la MSE.

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

Mean Absolute Percentage Error (MAPE) - Erreur Moyenne Absolue en Pourcentage

La MAPE [10] mesure la moyenne des valeurs absolues des erreurs en pourcentage par rapport aux valeurs réelles.

$$MAPE = \frac{1}{n} \sum_{i=1}^n \left| \frac{y_i - \hat{y}_i}{y_i} \right| * 100\%$$

À noter que comme on divise par y_i , si cette valeur est 0 ou très proche nous obtenons des erreurs de calculs.

5.3.2 Erreur pour le pIC50

Pour l'évaluation de nos modèles pour le pIC50, Il est important de noter que son évaluation avec la MAPE n'est pas pertinante car, comme mentionné précédemment, nous pouvons avoir de valeurs de pIC50 qui valent 0 et donc erronneront son calcul. Nous nous concentrerons donc sur les trois autres graphes :

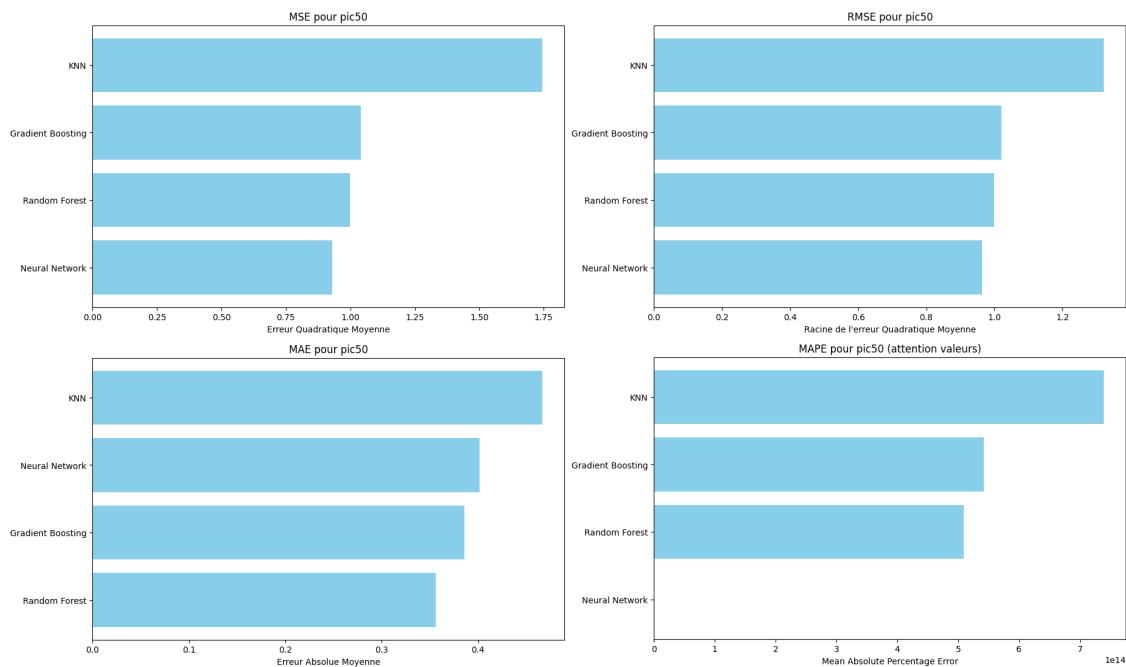


Figure 15: Métriques de regression pour les différents modèles sur pIC50

Nous constatons rapidement que le KNN est notre modèle le moins efficace, résultat attendu de par sa simplicité. Logiquement le MSE et le RMSE nous donnent un résultat similaire avec la meilleure évaluation pour le réseau de neurones qui n'est pas la même conclusion qu'avec le MAE qui a pour meilleur modèle le Random Forest. Cette différence d'évaluation peut s'expliquer un modèle qui ferait moins d'erreur mais lorsqu'il en fait, ces erreurs sont assez grossières. On voit que les valeurs pour nos trois modèles autre que le KNN donnent des résultats assez proche et sont donc plutôt performant dans l'ensemble.

5.3.3 Erreur pour le logP

Pour nos modèles sur le logP les résultats sont beaucoup plus homogènes :

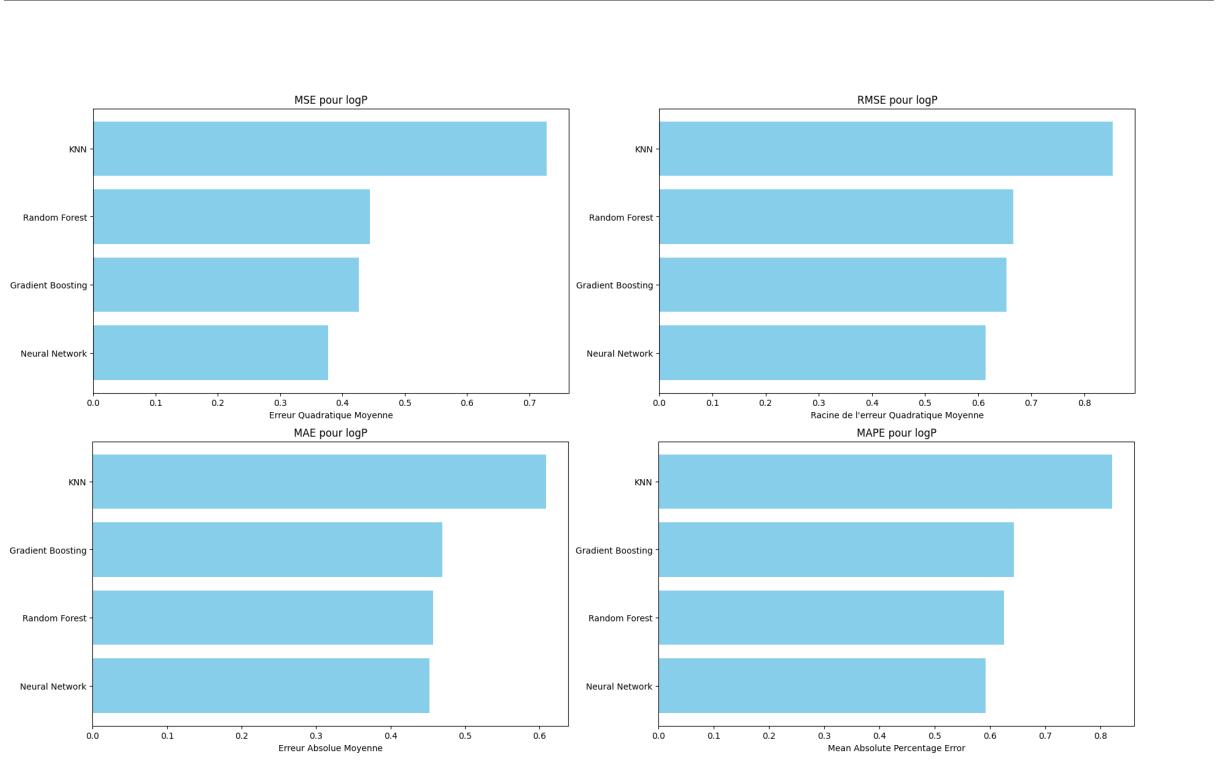


Figure 16: Métriques de regression pour les différents modèles sur logP

Comme pour le pIC50 [5.3.2], le modèle le moins efficace est celui du KNN qui est largement en dessous des trois autres. Cependant, ici toutes nos mesures s'accorde sur le fait que notre réseau de neurones produit les meilleurs résultats. On observe juste une inversion entre le Random Forest et le Gradient Boosting selon si l'erreur calculée est quadratique ou absolue.

6 Conclusion

Au terme de cette étude approfondie sur les prédictions des propriétés de molécules, nous avons pu comprendre la structure et le fonctionnement de différents algorithmes de machine learning et leur implémentation plus ou moins complèxes avec l'aide des librairies correspondantes. Le but initial étant d'évaluer le meilleur modèle pour prédire des valeurs, nous avons pu constater l'efficacité des modèles de machine learning, avec les réseaux de neurones se démarquant comme la meilleure approche et le K-Nearest Neighbors comme méthode trop basique.

Etant donné que le réseaux de neurones s'est avérée supérieure dans l'analyse de nos résultats et dans leurs comparaison, il aurait été tout de même intéressant de pousser ces expériences en testant par exemple différentes architectures de réseaux de neurones, comme les réseaux convolutifs (CNN) ou encore les réseaux récurrents (RNN), qui aurait pu offrir potentiellement des gains de performances supplémentaires. D'autres approches auraient encore pu être effectuées comme une approche de régression logistique ou bien une approche de Long Short-Term Memory (LSTM).

Ces efforts d'optimisation et d'expansion de notre cadre expérimental sont susceptibles de révéler des insights précieux sur la généralisabilité et l'efficacité opérationnelle de nos approches, contribuant ainsi à l'avancement de la recherche et à l'application pratique des propriété des molécules.

7 Contributions

- Connaissance théoriques, informatiques et mathématiques : KALOUSIS Alexandros, MARTIN Reyes Victor, BOGET Yoan.
- Implémentation du projet et rédaction du rapport : VANSOON Nathan, PAPA David.

8 References

- [1] *A new approach for simultaneous calculation of pIC50 and logP through QSAR/QSPR modeling on anthracycline derivatives: a comparable study.* Fereydoun Sadeghi. Abbas Afkhami- Tayyebeh Madrakian, Raouf Ghavami. 2021.
- [2] Alexandros Kalousis. *Data Mining.* Faculty of Sciences, University of Geneva. 2024.
- [3] Kobia. *Analysis of Regression Metrics.* URL: <https://kobia.fr/regression-metrics-quelle-metrique-choisir/>.
- [4] Medium. *Random Forest Regression.* URL: <https://levelup.gitconnected.com/random-forest-regression-209c0f354c84>.
- [5] RDKit Documentation. URL: <https://www.rdkit.org/docs/index.html>.
- [6] Wikipédia. *Euclidean Distance.* URL: https://en.wikipedia.org/w/index.php?title=Euclidean_distance&oldid=1196460530.
- [7] Wikipédia. *Gradient Boosting.* URL: https://en.wikipedia.org/wiki/Gradient_boosting.
- [8] Wikipédia. *K-Nearest Neighbors (KNN).* URL: https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm.
- [9] Wikipédia. *MAE (Mean Absolute Error).* URL: https://en.wikipedia.org/wiki/Mean_absolute_error.
- [10] Wikipédia. *MAPE (Mean Absolute Percentage Error).* URL: https://en.wikipedia.org/wiki/Mean_absolute_percentage_error.
- [11] Wikipédia. *MSE (Mean Squared Error).* URL: https://en.wikipedia.org/wiki/Mean_squared_error.
- [12] Wikipédia. *Rectifier (Neural Networks).* URL: [https://en.wikipedia.org/w/index.php?title=Rectifier_\(neural_networks\)&oldid=1199800058](https://en.wikipedia.org/w/index.php?title=Rectifier_(neural_networks)&oldid=1199800058).
- [13] Wikipédia. *RMSE (Root Mean Squared Error).* URL: https://fr.wikipedia.org/wiki/Racine_de_l%27erreur_quadratique_moyenne.