Rapport Projet Machine Learning

Altruy Alan
Matricule : 200161
alan.altruy@student.umons.ac.be

Delabie Xavier
Matricule : 201120
xavier.delabie@student.umons.ac.be

Vanduynslager Estebane
Matricule : 220191
estebane.vanduynslager@student.umons.ac.be

FS
UMons
Année Académique 2023-2024

Sections

1 Section 1 : Exploration et analyse de données

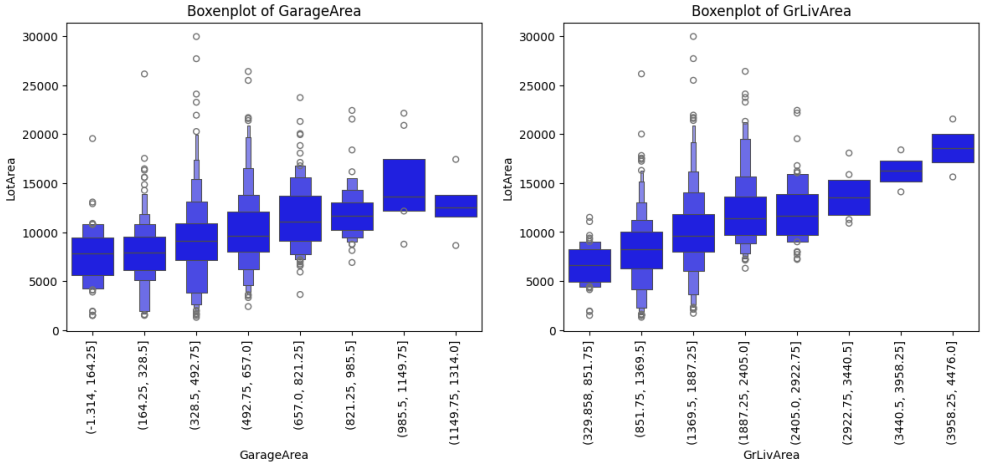
Dans le cadre de l'analyse de données, nous avons décidé de créer un boxenplot pour chacune des caractéristiques présentes dans le jeu de données. Ces données ont ensuite été examinées afin d'évaluer leur potentiel intérêt pour une sélection ultérieure.

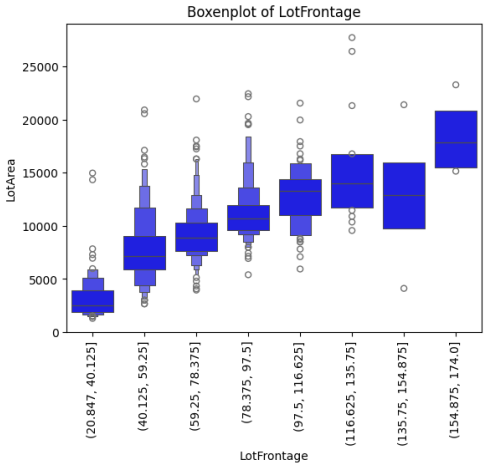
Lors de l'analyse des boxenplots, nous avons également constaté plusieurs valeurs aberrantes (par exemple, des valeurs de LotArea supérieures à 200 000). Ces erreurs sont probablement dues à des fautes de frappe. Nous nous en occuperons plus tard lors du nettoyage des données.

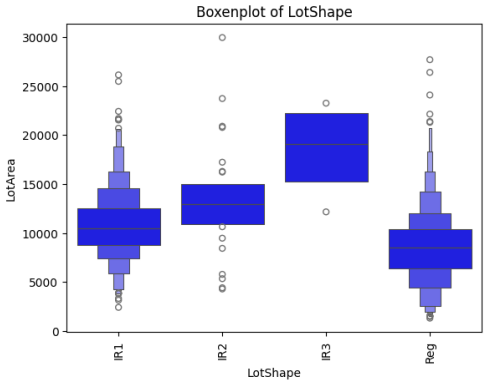
Le choix des caractéristiques s'est donc principalement basé sur l'allure des boxenplots. Si ces derniers présentaient des différences notables dans leur taux de défaut marginal, alors la caractéristique correspondante a été sélectionnée pour être utilisée dans les futurs algorithmes. Les autres caractéristiques, quant à elles, ne seront pas utilisées lors des phases suivantes de la recherche. Les caractéristiques intéressantes résultantes de ce processus sont les suivantes :

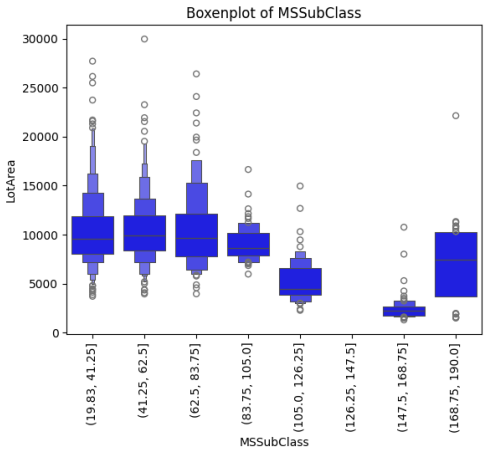
GarageArea, GrLivArea, LotFrontage, LotShape, MSSubClass, TotalBsmtSF, TotRmsAbvGrd.

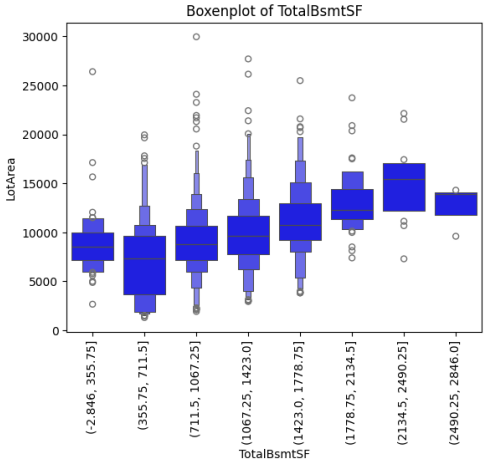
Accompagnées de leur boxenplot respectifs :

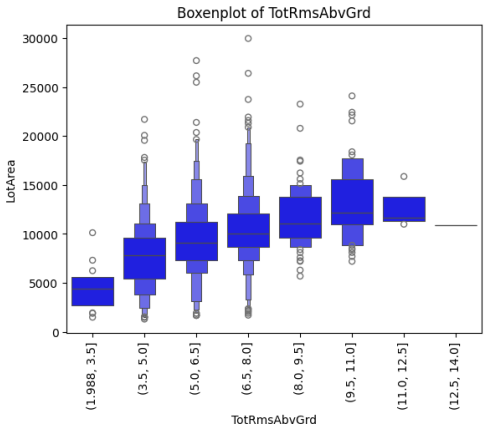



Boxenplot of LotFrontage
LotArea
LotFrontage
(20.847, 40.125]
(40.125, 59.25]
(59.25, 78.375]
(78.375, 97.5]
(97.5, 116.625]
(116.625, 135.75]
(135.75, 154.875]
(154.875, 174.0]


Boxenplot of LotShape
LotArea
LotShape
IR1
IR2
IR3
Reg


Boxenplot of MSSubClass
LotArea
MSSubClass
(19.83, 41.25]
(41.25, 62.5]
(62.5, 83.75]
(83.75, 105.0]
(105.0, 126.25]
(126.25, 147.5]
(147.5, 168.75]
(168.75, 190.0]


Boxenplot of TotalBsmtSF
LotArea
TotalBsmtSF
(-2.846, 355.75]
(355.75, 711.5]
(711.5, 1067.25]
(1067.25, 1423.0]
(1423.0, 1778.75]
(1778.75, 2134.5]
(2134.5, 2490.25]
(2490.25, 2846.0]


Boxenplot of TotRmsAbvGrd
LotArea
TotRmsAbvGrd
(1.988, 3.5]
(3.5, 5.0]
(5.0, 6.5]
(6.5, 8.0]
(8.0, 9.5]
(9.5, 11.0]
(11.0, 12.5]
(12.5, 14.0]

De plus, nous avons également testé nos résultats avec quelques données qui ne sont pas présentes dans cette liste. Les features qui ont été testées couplées aux features séléctionnées avec les boxenplots et qui ont renvoyé un bon résultat sont :

Alley, Heating, MSZoning, Neighborhood, Street, YearBuilt.

Il n'est pas surprenant que ces données impactent la caractéristique "LotArea" ciblée. Cependant, elles n'ont pas été vraiment choisies grâce à leur boxenplot, car ceux-ci ne renvoyaient que peu d'informations sur l'impact par rapport à la feature ciblée.

Finalement, nous avons implémenté une fonction qui se charge de sélectionner les features importantes de manière dynamique grâce à un modèle de ***Random Forest*** qui nous renseigne sur l'importance des features grâce à sa méthode ***feature_importances_***. Cette fonction permet donc de trouver d'autres caractéristiques dans le cas où nos données d'entraînement étaient trop biaisées et non représentatives de la réalité.

2 Section 2: Méthodologie

2.1 Section 2.1 : Idée générale

La méthode vers laquelle nous nous sommes orientés sont les arbres de décisions et plus précisément la méthode ***Random Forest***. Ceux-ci, à l'aide de quelques implémentations et des features intéressantes trouvées dans la section 1, se sont montrés très efficaces pour classifier efficacement les caractéristiques dans le but de dégager un bon résultat pour "LotArea".

Les hyperparamètres que nous avons déterminer pour notre modèle sont les suivants :

- criterion = poisson
- bootstrap = True
- n_estimators = 500
- min_samples_split = 5
- random_state = 4
- n_jobs = 'nombre de threads du processeur'

Ils ont été adoptés suite à de nombreux tests, car ces paramètres renvoyaient les meilleurs résultats.

Ces hyperparamètres sont facilement descriptibles. Nous spécifions que notre forêt comportera 500 arbres. De plus, il est nécessaire d'avoir au minimum 5 données dans une feuille pour qu'un split de celle-ci ait lieu, et que le critère d'évaluation de chaque split de l'arbre suit une loi de Poisson.

Le paramètre ***bootstrap*** permet, comme son nom l'indique, de diviser notre jeu de données et de le mélanger pour chaque arbre de la forêt, permettant ainsi d'avoir une variation dans les données de train afin de ne pas sur-ajuster.

Le paramètre ***random_state = 4*** bloque l'algorithme de RandomForest afin de garder ses valeurs de bootstrap et de ne pas être trop aléatoire sur ses prédictions, garantissant ainsi la reproductibilité des résultats. La valeur 4 a été choisie car elle correspondait le mieux aux données après de nombreux tests.

Le paramètre ***n_jobs*** permet quant à lui de paralléliser l'algorithme de RandomForest afin d'optimiser le temps d'exécution. Par défaut, ce paramètre équivaut au nombre de cœurs du processeur sur lequel l'algorithme tourne.

Nous avons choisi de ne pas définir de valeur maximale pour la **profondeur de l'arbre** afin de permettre au modèle de capturer au mieux la complexité des relations présentes dans les données. En fixant une limite à la profondeur de l'arbre, nous risquions de compromettre les performances de prédiction. En laissant l'algorithme décider de la profondeur optimale de l'arbre, nous permettons au modèle de s'adapter de manière flexible aux données, ce qui peut conduire à de meilleures performances prédictives. Cependant, cette approche nécessite une surveillance attentive pour éviter le sur-ajustement, c'est pourquoi nous avons déterminé des réglages pour chacun des autres hyperparamètres.

Finalement, nous avons implémenté une fonction ***get_params*** qui permet de retourner les hyperparamètres optimaux parmi une grille d'hyperparamètres à essayer, donnée en paramètre.

Pour se faire, nous utilisons un modèle ***RandomSearchCV*** avec une cross-validation de 5 et 100 itérations pour déterminer les meilleurs hyperparamètres dans un délai raisonnable.

2.2 Section 2.2 : Récupération de données

Les données récupérées du fichier "train.csv" sont conservées dans le dataframe ***df***. Le format de ses données non-numériques est changé pour ***'category'***. Les données ***X_train***, ***y_train***, ***X_test*** et ***y_test*** sont calculées en fonction du mode dans lequel se trouve l'algorithme (phase de prédiction de données ou phase d'analyse).

Lors de la phase d'analyse, les données de ***df*** sont scindées (80% pour le train et 20% pour le test). La variable ***'LotArea'*** est stockée dans les dataframes ***y_train*** et ***y_test*** respectivement.

Lors de la phase de prédiction (soumission), les données de "test.csv" sont sauvegardées dans les dataframes ***X_test*** et ***y_test***. Dans ce cas, les données stockés dans ***df*** seront stockés dans les dataframes ***X_train*** et ***y_train*** vu qu'il n'y a pas lieu de générer un dataframe de test à partir de ***df***.

2.3 Section 2.3 : Exploitation de données

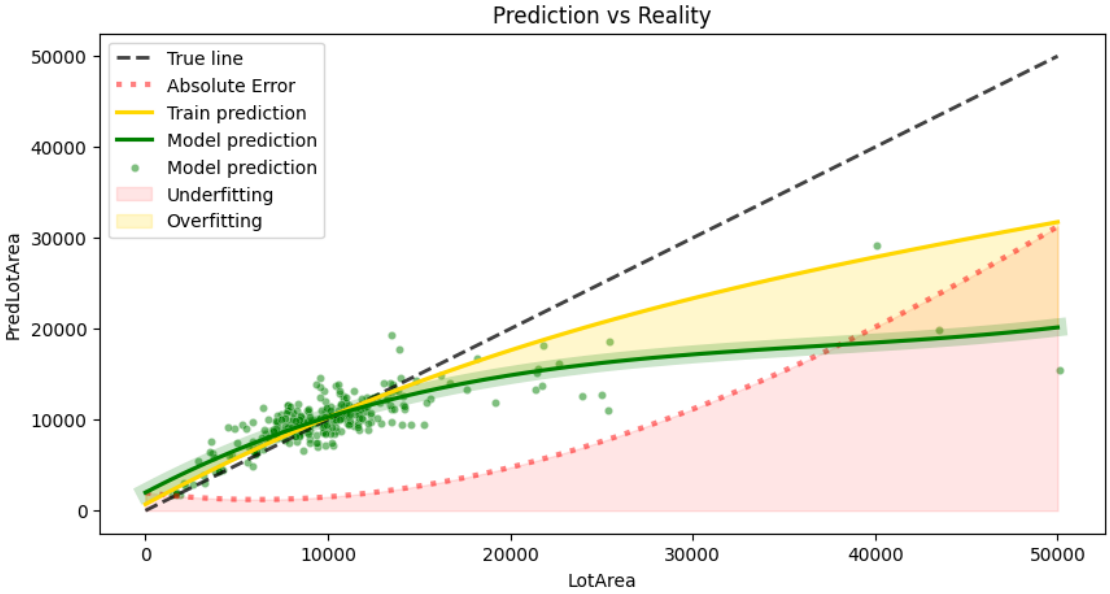
Nous avons décidé d'ignorer les valeurs de ***'LotArea'*** dépassant 80 000, car elles entraînaient l'algorithme sur des résultats extrêmes et peu représentatifs de la situation générale et augmentaient donc trop le biais.

2.4 Section 2.4 : Fitting de notre modèle

Nous avons réalisé de nombreux tests sur différents paramètres afin de prévenir le sur-ajustement de notre modèle. Pour ce faire, nous avons principalement comparé l'erreur quadratique moyenne (MSE) de notre modèle sur l'ensemble d'entraînement à celle sur l'ensemble de test.

Notre objectif était de réduire au maximum cette différence. Cette approche nous permet d'évaluer dans quelle mesure notre modèle sur-ajuste les données.

Cependant, il est également crucial que cette différence ne soit pas trop faible, car cela indiquerait un sous-ajustement, c'est-à-dire que notre MSE serait également trop faible. Nous cherchons ainsi à trouver un compromis entre le sur-ajustement et le sous-ajustement afin de minimiser l'erreur globale de notre modèle.



On peut voir sur le graphe que l'erreur MSE du test est proche du train tant que ***'LotArea'*** ne dépasse pas 15000/20000. Cela se justifie par le fait que plus de 95 % des données du train ont une valeur de ***'LotArea'*** inférieure à 20 000. Notre modèle a donc plus de mal à prédire ces valeurs plus extrêmes, cela se traduit également par une augmentation de l'erreur absolue une fois ***'LotArea'*** passé au-dessus de 20 000.

2.5 Section 2.5 : Seconde approche: Empilement de modèles

Cette sous section présente le second algorithme mis en oeuvre dans le cadre de la compétition Kaggle. Cet algorithme n'a pas énormément été investi, nous nous sommes principalement focalisé sur notre modèle imposé.

Le modèle choisi est un ***StackingRegressor***, un modèle d'empilement de plusieurs modèles. Nous utilisons deux estimateurs principaux, le ***GradientBoosting*** et le ***LassoCV***. Le modèle ***GradientBoosting*** permet de bien fitter les données mais lors de nos tests, nous avons remarqué qu'il avait tendance à beaucoup trop sur-ajuster, c'est pourquoi nous l'avons couplé avec le ***LassoCV*** qui permet, grâce à sa Cross-Validation de minimiser ce sur-ajustement et de plus s'approcher de la réalité. Pour terminer, l'estimateur final est un ***RidgeCV*** qui permettra une fois de plus d'accentuer la minimisation du sur-ajustement tout en conservant des prédictions très correctes grâce au ***GradientBoosting***.

Nous avons tout de même obtenu de bons résultats avec cet algorithme malgré le manque d'implication. Nous avons essentiellement procédé de la même manière que pour le premier modèle. La méthode de séléction des features est identique et notre manière de choisir les hyperparamètres des modèles est semblable à l'exception que nous ayons adapté la grille de paramètres

Voici les hyperparamètres sélectionnés pour le ***LassoCV***:

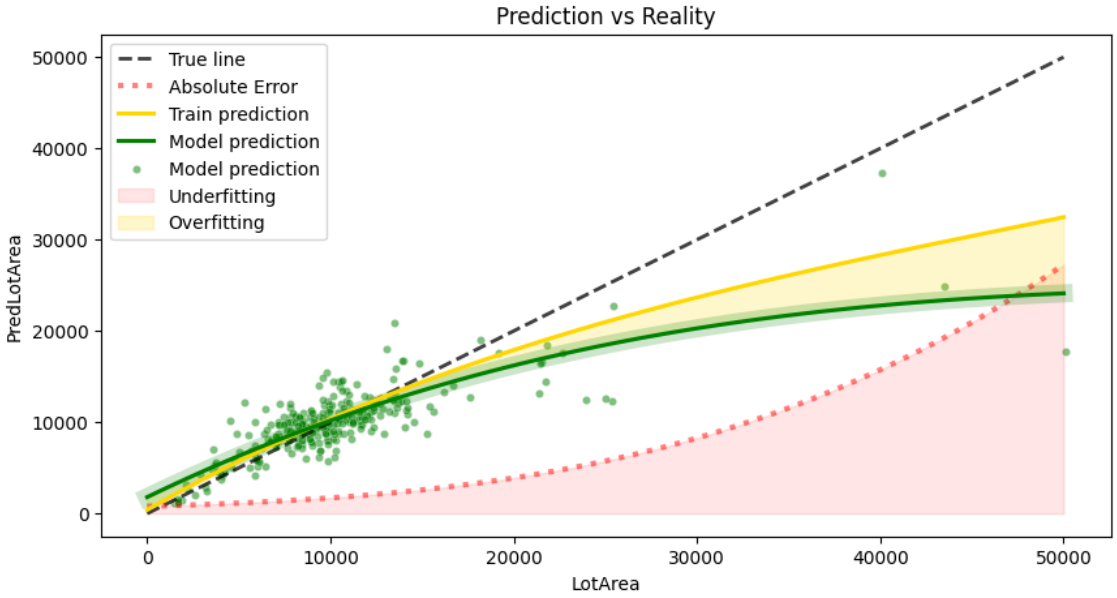
- n_alphas = 300
- eps = 0.0001
- cv = 10

Et voici les hyperparamètres sélectionnés pour le ***GradientBoosting***:

- subsample = 0.95
- random_state = 3
- n_estimators = 150
- min_samples_split = 9
- min_samples_leaf = 8
- max_features = sqrt
- max_depth = 11
- learning_rate = 0.05

2.6 Section 2.6 : Seconde approche: Fitting + Comparaison avec le premier modèle

La méthode pour prévenir le sur-ajustement est la même que pour le ***Random-Forest***. Voici le graphe du fitting:



On remarque que, comme pour le premier modèle, notre modèle commence à sous-ajuster lorsque ***'LotArea'*** est supérieur à 20 000 pour les mêmes raisons. On constate également que l'écart entre le train et le test est plus minime ce qui se traduit par des MSE pour le train et le test plus proche l'un de l'autre. Ce modèle a donc tendance à moins sur-ajuster les données mais on remarque que l'erreur absolue moyenne des prédictions, lorsque ***'LotArea'*** est sous 20 000, est un peu plus élevée que le premier modèle, ce qui se traduit par un sur-ajustement plus faible.

3 Section 3: Résultats et Discussion

3.1 Section 3.1 : Récupération de résultats

Pour réagir aux différents résultats obtenus, nous avons implémenté la formule de l'erreur racine carrée entre la valeur train et la valeur test pour des features déterminées. Cette formule est rappelée ci-dessous :

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

Avec :

1. n, le nombre de points du set de données
2. y_i, la vraie valeur de la variable "LotArea"
3. $\hat{y}$_i la valeur prédite pour "LotArea"

Cette méthode nous permet d'obtenir une approximation du score que nos implémentations vont générer. C'est grâce à cette méthode que nous avons pu réaliser des tests avec des sets de paramètres et de features pour déterminer lesquels renvoyaient les meilleurs résultats.
Comme cité précédemment, le score n'est pas la seule valeur implémentée pour analyser nos résultats. L'erreur quadratique moyenne, l'erreur absolue moyenne, maximale et minimale ainsi que la somme des carrés des résidus. L'objectif est de déterminer quelles combinaisons de paramètres et de features donnent une erreur minimale le plus souvent possible.

Cette analyse des résultats se trouve dans la section ***Experiment*** du code et comprend, en plus des analyses décrites ci-dessus, un affichage graphique de ces différentes informations.

3.2 Section 3.2 : Analyse des résultats

Les résultats sous forme graphique obtenus pour les deux modèles se trouvent dans les sections respectivement ***2.4*** et ***2.6*** du document. A ces graphes, nous pouvons ajouter quelques informations importantes concernant les résultats:

Time	2.00 s
Kaggle Score	0.24691
MSE Test	128.6303
MSE train	54.6578
To be near to 1	0.4249
Erreur absolue moyenne	1987.93
Erreur absolue maximale	34625.95
Erreur absolue minimale	11.56

Time	8.88 s
Kaggle Score	0.24677
MSE Test	112.9118
MSE train	66.9116
To be near to 1	0.5926
Erreur absolue moyenne	1975.71
Erreur absolue maximale	32407.32
Erreur absolue minimale	0.26

On y trouve, à gauche le score du modèle ***Random Forest*** et à droite celui du ***StackingRegressor***.

On peut y décerner que le score est très acceptable pour les deux modèles. En plus de cela, l'erreur moyenne est très faible. On constate cépendant que le temps d'execution de l'algorithme de ***RandomForest*** est drastiquement plus faible.

En ce qui concerne les erreurs absolues, la valeur maximale reflète un résultat pour une valeur de ***'LotArea'*** très élevée. Ces valeurs ont tendance à présenter des caractéristiques anormales, ce qui justifie une grande imprécision pour les approcher. Cependant, ces cas sont isolés et présents en faible quantité, ce qui ne pose pas vraiment de problème. Les graphiques mentionnées précédemment expliquent la tendance du modèle de prédiction par rapport aux valeurs de test et de train. Voici un exemple explicatif de ces graphes:

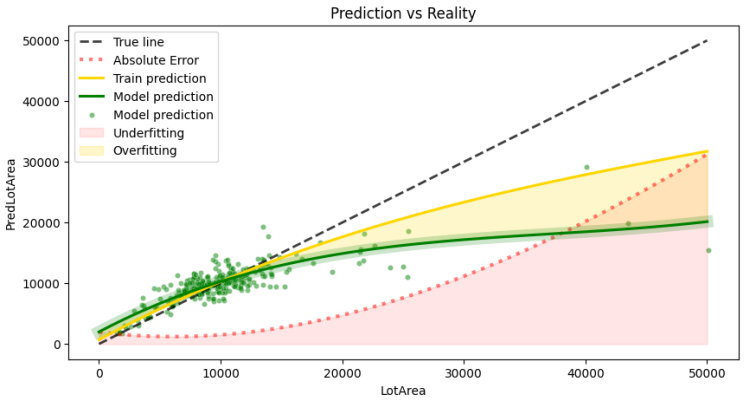


Figure représentant le RandomForest

Ce graphique nous montre bien que la courbe du modèle de prédiction (en vert) cherche à épouser au maximum les valeurs réelles représentées par la ligne noire en pointillés. La zone rouge représente le sous-ajustement alors que la zone jaune représente le sur-ajustement. C'est visuellement plus facile à contrôler, l'idéal serait que ces zones n'existent pas, ce qui signifierait qu'il n'y ait ni sous-ajustement ni sur-ajustement.

3.3 Section 3.3 : Discussion

Ces résultats nous ont permis d'approcher la feature ***'LotArea'*** le plus possible grâce aux caractéristiques sélectionnées dans la section 1, à la sélection de paramètres trouvés en analysant les résultats obtenus et surtout au modèle ***RandomForest*** choisi qui aura fait la différence lors du score final.

La deuxième place que nous avons obtenu dans la compétition Kaggle est d'ailleurs le fruit des prédictions de notre modèle ***RandomForest***, ce qui nous pousse à dire qu'il est un bon modèle pour ce jeu de données contenant pas mal de données aberrantes.