Random Forest

Le random Forest est un ensemble d'arbres de décision, chacun un peu différent, et on moyenne leurs prédictions. La différence principale vient du fait que chaque arbre est entraîné sur un échantillon bootstrap (tiré avec remise) du jeu d'entraînement, et à chaque nœud on n'autorise qu'un sous-ensemble aléatoire de variables pour choisir la coupe.

À l'entraînement, on choisit le nombre d'arbres à construire via l**'hyperparamètre n_estimator**, puis on répète la même recette : on crée un bootstrap, on fait pousser un arbre en se limitant à un nombre aléatoire de variables disponibles à chaque split, et on s'arrête selon des règles de complexité (profondeur maximale (max_depth), nombre minimal d'exemples par feuille, gain minimal). On conserve tous les arbres. En régression, on prend la moyenne des prédictions de tous les arbres ; en classification, on fait voter et on peut interpréter la probabilité d'une classe comme la proportion d'arbres qui la choisissent.

Ce qui fait la qualité d'une forêt, c**'est l'équilibre entre la "force" moyenne des arbres et leur corrélation**. Plus on autorise d'attributs à chaque split, plus un arbre a de chance de tomber sur la variable "parfaite" et donc d'être fort, mais plus les arbres se ressemblent, donc plus ils se corrèlent. À l'inverse, restreindre le nombre de variables testées par split rend les arbres plus divers, ce qui diminue la corrélation au prix d'arbres individuels un peu moins bons. La performance du modèle agrégé naît de ce compromis. **Augmenter le nombre d'arbres réduit presque toujours la variance et stabilise le modèle ; au delà de quelques centaines, le gain se tasse, mais il n'y a pas de sur-apprentissage dû au nombre d'arbres lui-même, seulement un coût de calcul.**

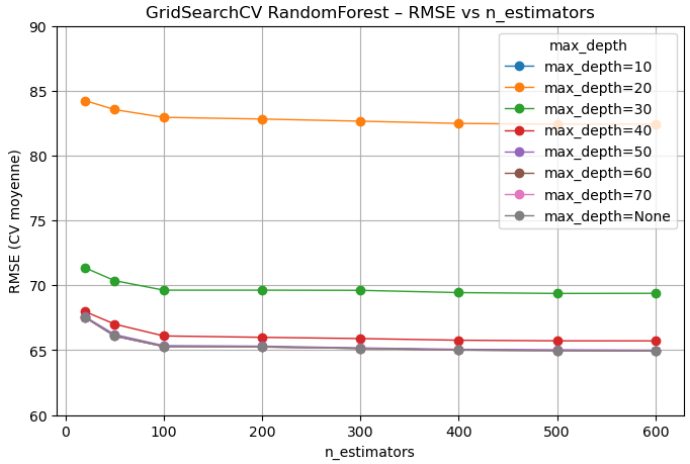
NB : quand on moyenne, le bruit individuel s'annule et la variance du modèle chute. C'est souvent un excellent point de départ parce que ça marche bien sans trop de réglages.

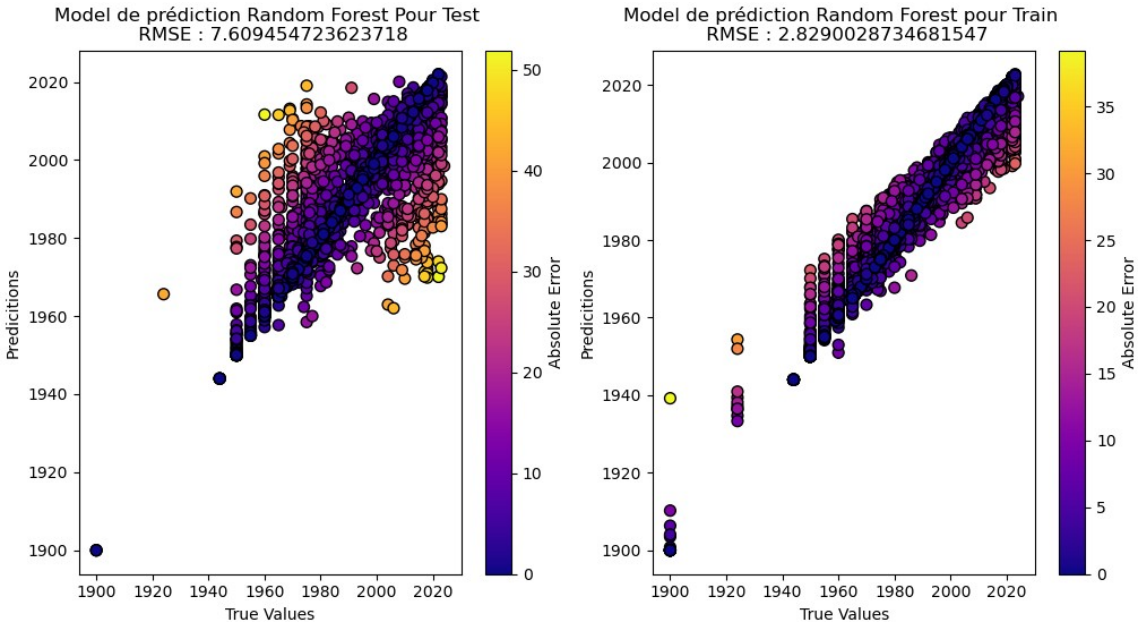
Les hyperparamètres qui pilotent vraiment le modèle sont :
- **Le nombre d'arbres** (n_estimators) q**ui fixe la stabilité statistique** ; on monte jusqu'à ce que la courbe de validation se stabilise.
- **La profondeur maximale (max_depth)** et la **taille minimale des feuilles (min_samples_leaf)** qui **contrôlent la complexité des arbres** ; des arbres très profonds collent au bruit, des arbres plus courts généralisent mieux.
- Le **nombre de variables candidates par split (max_features)** "log2" ou un entier en régression) façonne le compromis force/diversité ; en classification on prend souvent la racine du nombre de colonnes ("sqrt"), en régression on peut garder une fraction plus grande et éventuellement descendre si les arbres se ressemblent trop (log2" ou un entier en régression).
- On peut également activer l'estimation "out-of-bag" : chaque arbre n'a pas vu environ un tiers des lignes, on peut donc mesurer une performance de validation gratuite en les utilisant comme pseudo-test pendant l'entraînement. C'est très pratique pour choisir une zone raisonnable d'hyperparamètres avant une vraie validation croisée.

Sur la préparation des données, **scikit-learn n'accepte pas les valeurs manquantes et les valeurs non numériques** : il faut donc les imputer. Pour les variables catégorielles, un one-hot simple fonctionne bien ; un encodage ordinal sans ordre réel peut induire des splits arbitraires.

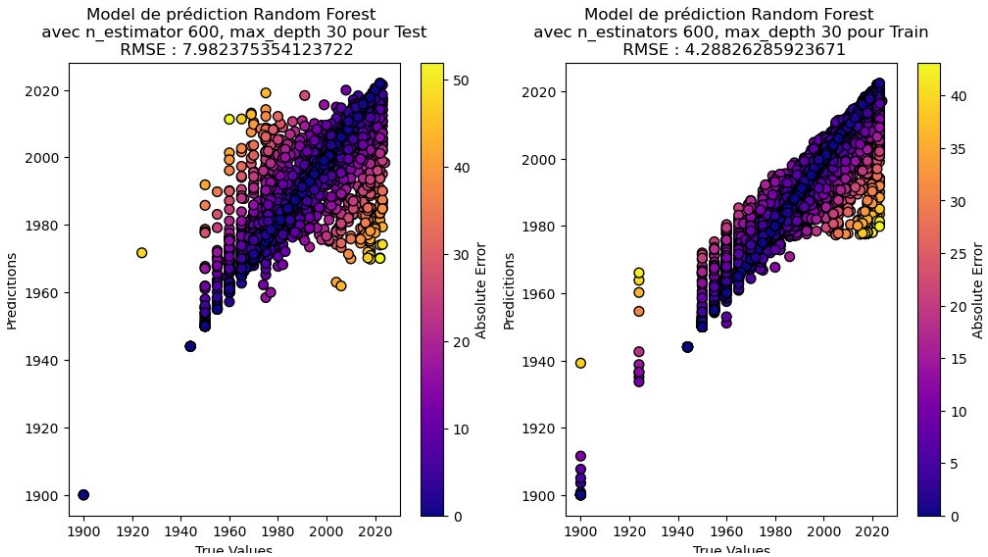
Dans le jeu de donnée, j'ai joué sur n_estimators et max_depth pour estimer le meilleure modèle.

Meilleurs paramètres Random Forest : {'Regressor__max_depth': None, 'Regressor__n_estimators': 600}

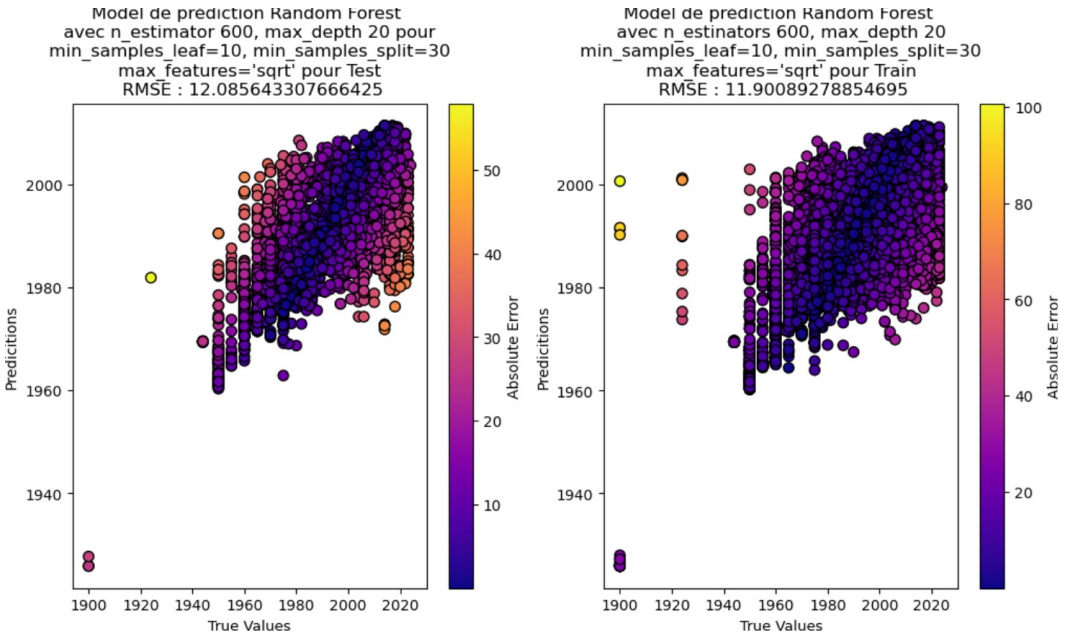




Si on laisse le meilleur modèle choisit avec un max_depth = None, on voit très nettement que mon modèle ici over-fit.
Généralement un arbre sans limite n'est presque jamais un bon choix par défaut. Un arbre sans limite finit vite par **mémoriser le bruit** : il crée des feuilles minuscules qui collent aux particularités de l'échantillon d'entraînement. Ça donne un modèle **à forte variance** : il performe très bien sur le train mais **généralise mal**. C'est aussi plus **lent** à entraîner et à prédire (beaucoup de nœuds), plus **gourmand en mémoire**, et nettement **moins stable** : une petite variation des données peut changer fortement la structure de l'arbre.



Si on change max_depth par 30, l'é**cart train/test net overfit toujours mais est moindre**
RMSE train ≈ **4.29** vs RMSE test ≈ **7.98**. Un peu d'écart est normal en forêt, mais là il est large : les arbres trop profonds mémorisent des micro-motifs du train qui ne généralisent pas.
On peut voir également un **fort biais.** Sur le test, les points hauts (vraies valeurs récentes) sont plutôt **sous-prédits**, et les valeurs basses **sur-prédits**. Classique avec des arbres profonds qui moyennent : on observe une pente < 1 autour de la diagonale, et des erreurs plus fortes aux extrêmes.
En jouant sur les autres paramètres, j'obtiens moins d'overfiting mais un plus mauvais modèle de prédiction :

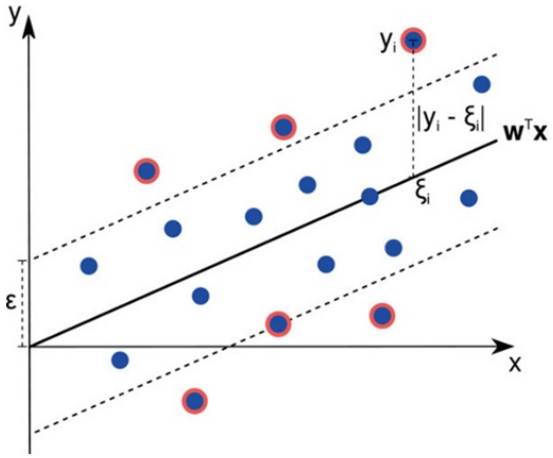


Avec plus de temps, j'aurais continuer à faire un grid_search sur les autres paramètres.

- **Limiter la complexité des arbres** : **baisser** `max_depth` (p.ex. 10–20) **et/ou** `min_samples_leaf` (5–20) et `min_samples_split` (20–50). Ces trois leviers réduisent les feuilles minuscules qui overfit.
- **Tester** `max_features` plus restrictif (`"sqrt"`, `0.3–0.7`) pour décorréler les arbres.
- **Garder 300–600 arbres** : au-delà, la variance baisse un peu mais n'efface pas le surfit causé par la profondeur.

Machine à vecteurs de support (SVM / SVR)

Le Support Vector Regression (SVR) est un algorithme de régression qui peut être appliqué aux régressions linéaires ainsi qu'aux régressions non-linéaires. Elle s'appuie sur le principe de machine à vecteur de support et est utilisée pour prévoir des variables continues ordonnées. Contrairement à la régression linéaire simple où le but est de minimiser le taux d'erreur, dans la régression à vecteur de support, l'idée est de faire rentrer l'erreur dans une marge prédéterminée et de maximiser l'écart entre les classes. En d'autres mots, cette forme de régression tente d'estimer la meilleure valeur au sein d'une marge, connue sous le nom de ε-tube.



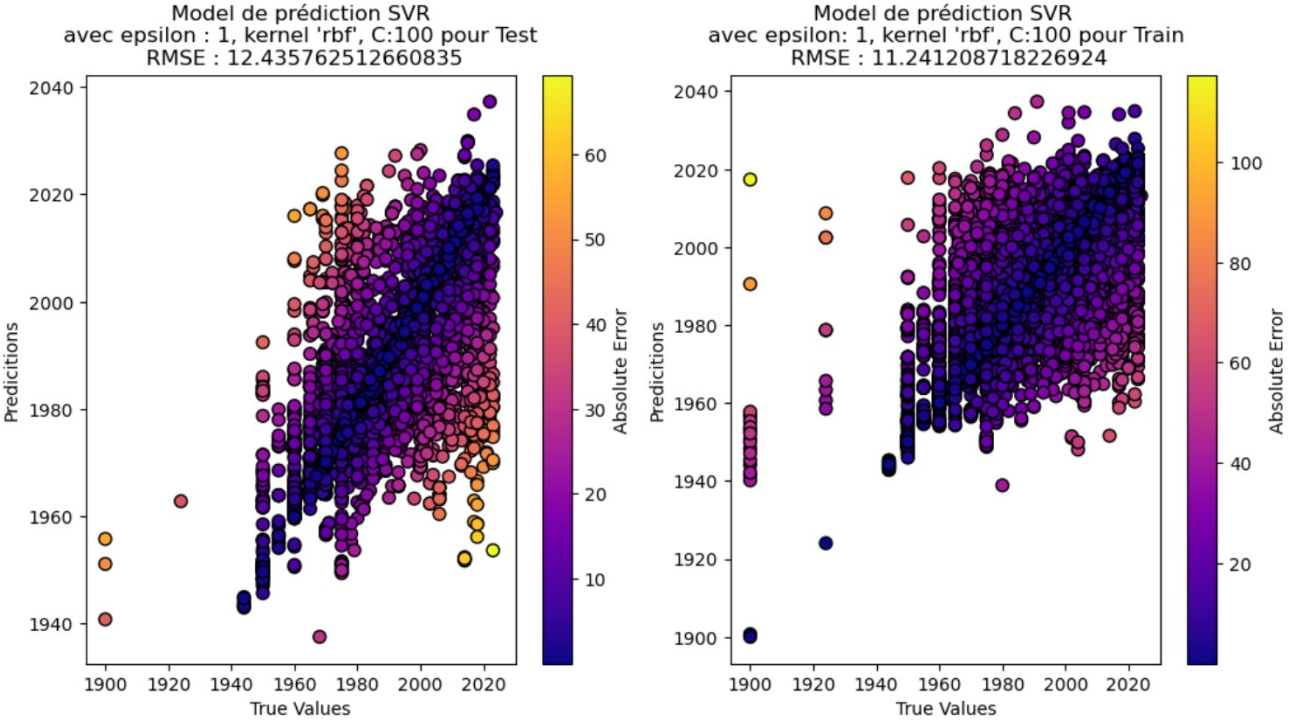
Sur ce graphe, la ligne noire représente l'hyperplan optimal, qui aide à prédire la valeur cycle. De plus, les ligne en pointillé représente ligne de marge, présent à une distance ε de l'hyperplan. Idéalement, l'objectif de la régression à vecteurs de support est de positionner l'hyperplan de telle sorte à ce qu'il sépare les variables indépendantes d'une classe à l'autre.
La meilleure droite est celle qui maximise la marge, c'est à dire la distance entre la droite et le premier point.
Les modèles de SVR font partie des techniques les plus utilisées pour les prédictions des cours d'actions, puisque ces algorithmes sont maintenant capables d'anticiper les mouvements de marché et donc d'estimer de futures valeurs d'actif, comme par exemple des cours boursiers. De plus, elle est particulièrement efficace sur des problèmes non-linéaires, et peut être utilisée pour détecter les fraudes financières ou des anomalies de production en industrie.

Méthode :
Un SVR cherche une **fonction f(x)** qui colle aux données tout en restant **simple**. Pour ça, il trace autour de f(x) un **tube de largeur ε** : toutes les prédictions qui tombent **dans** ce tube ne sont **pas pénalisées**. Seules les observations **en dehors** du tube coûtent quelque chose à l'algorithme. On ajuste f(x) pour que le tube capture le plus de points possible, mais on évite de tortiller la courbe inutilement grâce à un **terme de régularisation**.
Le paramètre C est le ratio de l'erreur de classification par rapport à la régularisation.
- **plus C est grand,** plus on punit les écarts en dehors du tube, plus le modèle essaiera de passer au millimètre près par les points. Il y aura moins d'erreur de classification (autoroute plus fine). Mais si C est trop grand, il peut y avoir des problème de généralisation (tube trop petit).
- **plus C est petit**, plus on accepte des écarts au profit d'**une fonction plus lisse qui généralise mieux**. On aura donc un tube large, mais on peut avoir beaucoup d'erreurs de classification (on regarde alors les courbes d'apprentissage pour observer les erreurs de classification)
Tous les points n'ont pas le même rôle. Ceux qui sont **sur** la paroi du tube ou **en dehors** deviennent des **vecteurs de support**. Eux seuls "portent" le modèle; les points bien au chaud à l'intérieur du tube ne changent rien à la solution.
Le principe du SVR vient aussi du **kernel** : si une droite ne suffit pas, on mesure les similarités entre points avec une fonction noyau (le plus courant est **RBF**) qui permet au modèle d'être non linéaire sans construire explicitement des centaines de nouvelles colonnes.
Les kernels non linéaire comme le RBF, vont être sensible à **γ (gamma)** => principe du cône
- **petit gamma** donne une fonction **très lisse. Plus il sera proche de 0, plus le kernele st quasi euclidien**
- **grand gamma** on aura alors une multi-dimension.
Quand on varie gamma => toujours en puissance de 10.

Si on récapitule => 3 paramètres contrôle la sensibilité du SVR :
- **ε** est la largeur du tube et s'exprime dans **les unités de la cible** (si on prédit une année, ε=1 signifie "je ne punis pas les erreurs ≤ 1 an").
- **C** fixe le compromis biais/variance en dehors du tube.
- **γ** façonne les dimensions quand on utilise un noyau non linéaire (ex. RBF).

Comme ces trois-là interagissent, on les choisit **via validation croisée**, sur des **échelles log**. Le SVR est **sensible à l'échelle des features**, donc **on standardise systématiquement X;** et comme ε est en unités de y, on pense à **standardiser y** quand ses valeurs sont grandes ou très étalées, sinon ε devient difficile à régler.
En pratique, le SVR brille sur des jeux **taille petite à moyenne** avec un **nombre de variables moyen/élevé**, surtout quand la relation n'est pas purement linéaire. Sur des jeux énormes, l'entraînement peut devenir coûteux; on bascule alors vers **LinearSVR** (linéaire, bien plus rapide) ou vers des approches par sous-échantillonnage. Quand un SVR "rame" ou sur-apprend, je vérifie l'échelle des variables, je diminue C, j'augmente un peu ε pour réduire le nombre de vecteurs de support, et je redescends γ pour lisser la fonction.
quand on autorise des courbures, et on garde tout ce petit monde stable grâce à la standardisation et à la validation croisée. Sur des jeux de taille raisonnable avec des patterns non linéaires, c'est souvent redoutablement efficace.
Exemple dans le notebook => modèle non-linéaire

```
from sklearn.svm import SVR
svr = SVR(kernel="rbf", C=10.0, gamma="scale", epsilon=0.1)
svr.fit(X_train, y_train)
y_pred = svr.predict(X_test)
```



Meilleur: {'Classifier__C': 1000, 'Classifier__epsilon': 0.01, 'Classifier__kernel': 'rbf'}

Dans scikit-learn :
SVR = SVRegression
SVC = SMClassifier => avec kernel