Projet Machine Learning

BURKHART Aaron
CHACUN Baptiste
GUIDDIR Lucas
TARLIN Michel

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Table des matières

I. Introduction

Dans le cadre de notre UE “Introduction to the theory and practice of Machine Learning”, nous sommes amenés à créer un code capable de générer des régressions satisfaisantes pour des datasets semblables. Tout ceci en appliquant bien sûr de bonnes pratiques de code sur une forme plus "industrialisée" et en coopérant de manière simultanée grâce aux fonctionnalités de Github.

Vous trouverez donc dans ce rapport les éléments récapitulatifs de ce projet ainsi que nos différents résultats sur différents types d’approche de régression par machine learning.

II. Objectifs

Dans tout projet il est important de se fixer des objectifs et des métriques pour ainsi pouvoir évaluer l'efficacité de nos résultats. Nous pouvons ainsi nous figer 3 objectifs principaux:

- Le projet reste une démonstration de ce que nous avons appris au long de cette UE. Nous devons donc arriver à avoir des algorithmes efficaces et qui puissent vraiment être utilisés pour résoudre un problème de régression.

- D’autre part, nous pourrions être amenés à coder dans un futur des codes pour nos employeurs. Cela nécessite du respect des bonnes habitudes de code pour qu’ainsi une personne externe puisse l’utiliser en comprenant chaque fonction. Une grande importance sera donc donnée à la forme de nos différentes fonctions et fichiers. Nous devons aussi prévoir un fichier main utilisable par des personnes non expertes en machine learning.

- Finalement nos enseignants veulent également nous présenter le travail collaboratif à travers Github. Nous allons ainsi essayer d’exploiter au plus possible les fonctionnalités de cet outil.

III. Méthodologie

a) Etapes de l'étude d'un jeu de données en Machine learning

1. Visualisation préliminaire

Avant de se lancer dans un projet, il est important de voir l'état des données. S'assurer de leur format et aussi déterminer quelles sont les données qui peuvent ne pas être cohérente dans l'analyse (une colonne remplie de 0 par exemple). En regardant les premières lignes et en comprenant les données, nous pouvons donc déjà faire une première mise en forme pour le traitement.

2. Nettoyage des données

Nous allons réaliser une régression et il nous faudra donc traiter les données pour les rendre exploitables à nos algorithmes. D'abord, il faut remplir les données manquantes par la moyenne et binariser les colonnes catégoriques. Ensuite, il faut normaliser les features pour pouvoir faire des calculs de distances cohérents dans nos algorithmes. (Nous partons du principe que nous n'avons pas de connaissances au préalable sur l'importance d'une feature sur la régression. Nous n'allons donc pas pondérer nos algorithmes).

3. Visualisation intermédiaire des corrélations

Avant de nous lancer dans l'entraînement de nos algorithmes nous pouvons d'abord chercher des corrélations entre nos différentes features. Si nous trouvons des corrélations très élevées (proche de 1), nous pouvons davantage réduire les espaces de dimensions de nos données car une seule de ces deux variables sera nécessaire à l'apprentissage, par exemple. Nous allons donc observer les matrices de corrélations et les matrices de scatterplot de nos features.

4. Visualisation intermédiaire des corrélations

Avant de nous lancer dans l'entraînement des algorithmes, nous pouvons d'abord chercher des corrélations entre nos différentes features. Si nous trouvons des corrélations très élevées (proche de 1), nous pouvons davantage réduire les espaces de dimensions de nos données car une seule de ces deux variables sera nécessaire à l'apprentissage. Nous observerons donc les matrices de corrélations et les matrices de scatterplot de nos features.

5. Partition de notre set de données

Dans chaque méthode, il est important de garder une partie des données pour l'entraînement, et l'autre pour tester le modèle. Chaque algorithmes gardera ainsi ⅓ des données pour les tester et le reste servira d'entraînements pour les régressions. On utilisera des partitions aléatoires de ces ⅔ restants pour les méthodes d'entraînement itératives.
Nous allons également trouver les meilleurs paramètres par cross-validation pour des méthodes qui nous le permettent.

6. Entraînement des modèles

Cinq méthodes de régression seront appliquées pour les jeux de données. Il s'agit à chaque fois d'un entraînement supervisé puisque nous connaissons la valeur mesurée pour chaque vecteur de données. La fonction de coût sera l'erreur quadratique moyenne pour tous nos algorithmes.

7. Validation des modèles

Pour valider notre modèle, nous comparons des métriques obtenues par chacun des différents algorithmes. On se servira de l'erreur quadratique moyenne pour évaluer la performance. Mais celle-ci étant dépendante des valeurs de la fonction à régresser, elle sera normalisée par l'amplitude des valeurs de la fonction à régresser. Nous aurons ainsi un meilleur aperçu de ce que représente cette erreur par rapport à nos données.

D'autre part, nous souhaitons trouver un moyen d'évaluer la complexité de chacun de nos modèles. Nous avons donc décidé de passer par une métrique temporelle : le temps que met le modèle à s'entraîner. Ensuite, pour comparer les modèles et leurs complexité en fonction des datasets, le temps sera aussi normalisé par la taille des données.

b) Format de la solution proposée

Comme indiqué par les consignes, l'ensemble de nos fonctions d'entraînement et visualisation des données se trouve dans un fichier .py à charger. Nous avons cependant en plus un fichier main.ipynb. C'est un notebook sur lequel un utilisateur, en suivant des indications, pourra faire l'analyse d'un jeu de données nouveau en visualisant les performances des algorithmes pour faire un choix sur le modèle final. L'utilisateur devra tout de même fournir un jeu de données "convenable" en entrée.

IV. Présentation et analyse des résultats

a) Présentation des méthodes;

Nous utilisons 5 méthodes de régression, dont une (SVM) dérive d'une méthode de classification.

1. Utilisation de random forests

Bien que simple, l'algorithme des randoms forests donne souvent de très bon résultats, parfois même mieux que les modèles plus complexes comme des réseaux de neurones.

Nous allons donc effectuer d'abord un tuning de la profondeur maximale des arbres de décisions par cross-validation. Nous générons par la suite un modèle de forêt pour notre régression. (Nous ne proposerons pas de visualisation d'un arbre car nous perdrions cette information en les combinant dans la forêt).

Exemple de tuning de la profondeur:

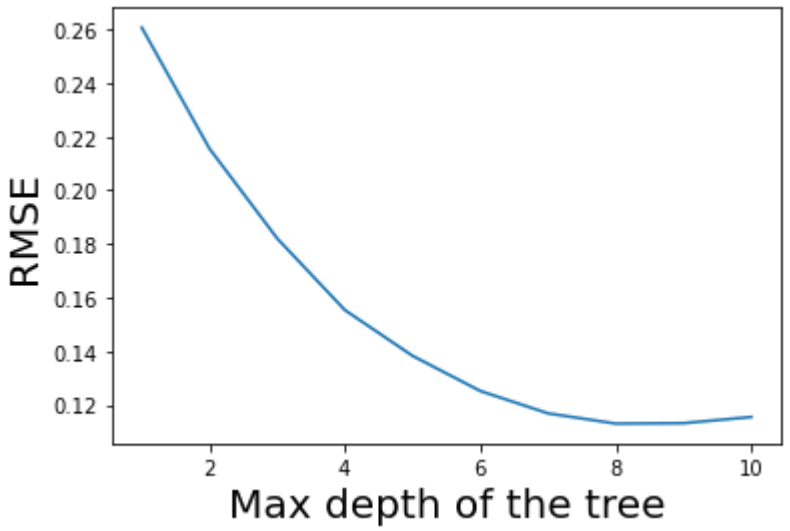

Figure 1 : Cross-validation de la profondeur max des arbres pour le dataset n°3

Une fois ce paramètre défini, nous pouvons entraîner le modèle et le tester.

2. Utilisation de réseaux de neurones

Les réseaux de neurones sont un outil souvent utilisé lorsque nous ne connaissons aucune structure des données et que nous voulons trouver un modèle non-linéaire qui s'approche le plus possible de la réalité. En fonction du nombre de paramètres, tel que le nombre de couches de neurones, l'entraînement peut prendre beaucoup de temps et incarne donc un outil puissant mais coûteux en complexité.

Réseau de 1 couche de neurones (Régression linéaire) :

Nous allons d'abord entraîner une seule couche de neurones avec autant de neurones que de features. Nous supposons ainsi que la fonction à régresser est une fonction affine des features. Nous arrivons ainsi à un modèle simple et interprétable s'il arrive à bien modéliser les données car nous disposons de l'ordonnée à l'origine et des coefficients.

Réseau de plusieurs couches de neurones :

Nous allons également utiliser des réseaux de neurones plus complexes, c'est-à-dire qu'ils auront plusieurs couches et des fonctions d'activation. En supposant que nos données ont toujours un bruit, nous utiliserons comme fonction d'activation des tangentes hyperboliques qui sont plus flexibles. De plus, le modèle utilisé est un modèle multilinéaire de perceptrons.

Ne connaissant pas les données auparavant, on déterminera par cross-validation le nombre de couches de neurones de notre modèle et donc ajuster sa complexité en sachant que la taille des données va beaucoup influencer son efficacité.

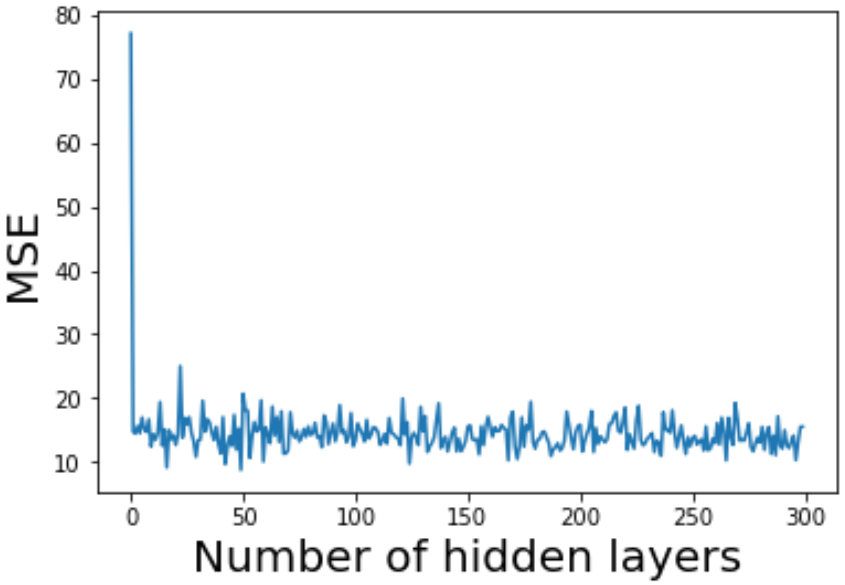

Figure 2 : Cross-validation du nombre de couches de neurones pour le dataset 1

Nous observons que hormis pour le perceptron simple, la MSE est assez équivalente dans ce cas là. Les oscillations sont dûes à la non-linéarité des modèles et donc au hasard du choix initial du set d'entraînement. Cependant, ce résultat pourrait varier pour d'autres datasets.

Nous laissons le choix à l'utilisateur de la profondeur du réseau vu les résultats un peu aléatoires que l'on peut obtenir avec cette méthode mais une valeur de 100 (celle par défaut dans scikit learn) donne usuellement de bons résultats.

3. Régression linéaire suite à une PCA

Parfois, il se trouve que les données ont structurellement de fortes variances suivant des directions de l'espace engendrées par les features, et qu'on peut ainsi considérablement réduire les dimensions de notre problème en trouvant ces directions. C'est le principe de la PCA qui est une manière analytique de transformer nos données pour réduire leurs dimensions.

Nous allons donc réaliser une PCA et observer les résultats qui s'obtiennent par régression linéaire dans ces espaces.

Nous donnons aussi les taux de variances expliquées cumulés par composantes, pour savoir à partir de combien de composantes nous retrouvons un pourcentage données des données.

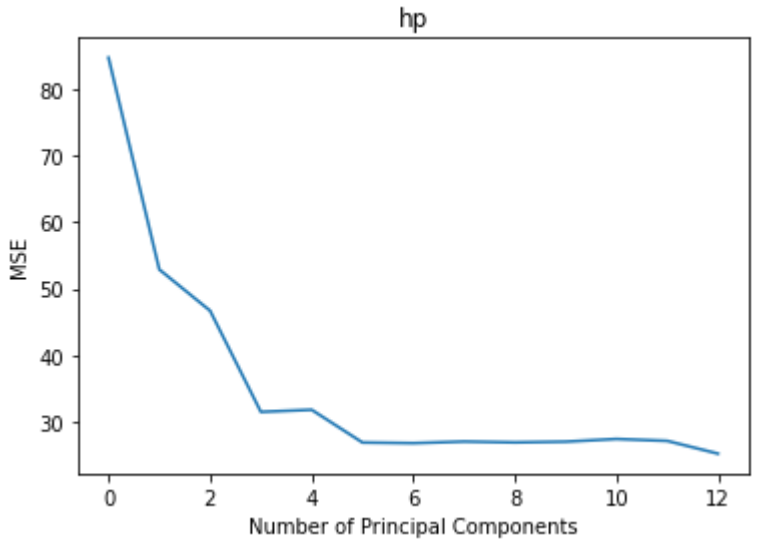

Figure 3 : EQM en fonction d'un modèle linéaire comportant n composantes pour le dataset n°1

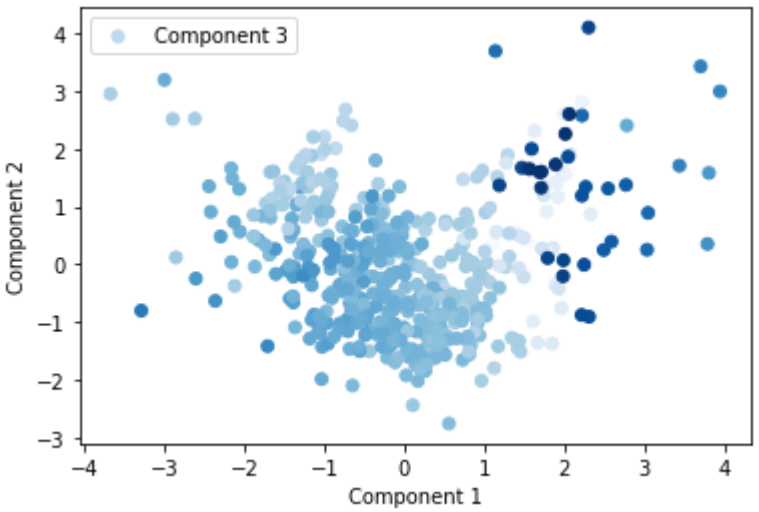

Figure 4 : Répartition des données suivant les trois premières composantes pour le dataset n°1

Nous pouvons observer qu'avec 3 composantes ici, il y a déjà des tendances par rapport à nos données.

4. Régression par SVM

Il était souhaitable de trouver un dernier moyen un peu plus original pour régresser les données. Du fait que des algorithmes très efficaces existent pour la classification, nous avons alors trouvé un module permettant de labelliser des données continues. Avec la fonction label_encoder de sk-learn, nous avons transformé notre problème de régression en classification et appliqué un modèle de SVM sur nos données.

Le label encoder est une sorte de fonction affine qui a une valeur continue associe un entier.

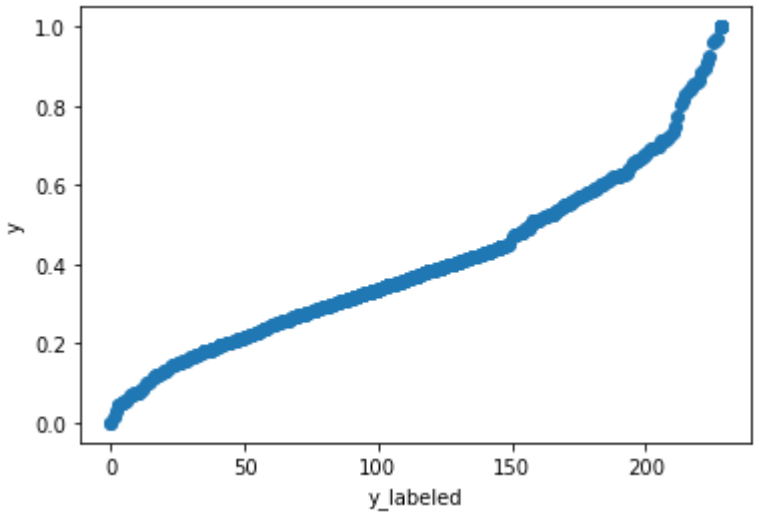

Figure 5 : Transformation de notre espace à régresser en labels pour le dataset 1

Cet algorithme peut vite devenir très coûteux en complexité car plus il y a de label, plus il y aura de frontières à calculer.

Nous avons voulu tuner le kernel des SVM en calculant le degré optimal pour un noyau polynomial. Cependant, les temps d'entraînement dépassant les 3h nous avons arrêté ce tuning et simplement choisi un polynôme de degré 2 qui donnait de bon résultats en TP.

b) Adaptabilité de nos algorithmes:

Pour confirmer la possibilité d'industrialisation du code, nous sommes passés par 3 dataset différents:

- Un dataset sur le prix des maisons à Boston
- Un dataset sur des patients pouvant avoir un cancer de prostate
- Un dataset extrait d'un projet des étudiants de la TAF Dasci avec une base de données beaucoup plus conséquente

Les 3 Datasets ont été traités par toutes les méthodes avec des résultats qui varient notamment pour le dernier duquel nous disposons de beaucoup plus de données.

Ils ont été légèrement modifiés avant d'être traités par notre fonction de nettoyage. Or, les algorithmes sont bien tous performants pour les 3 datasets.

c) Performance sur différents jeux de données

Comme indiqué précédemment nous allons travailler avec 3 métriques pour évaluer les performances de nos algorithmes:

L'erreur quadratique moyenne normalisée:

L'indicateur par défaut de la performance des algorithmes est l'erreur quadratique moyenne. C'est souvent la fonction à minimiser. Or, nos différents jeux de données ayant des fonctions à régresser de valeurs très différentes, nous allons donc normaliser ces valeurs par l'amplitude de cette fonction (le max - le min). Nous trouvons les résultats suivants:

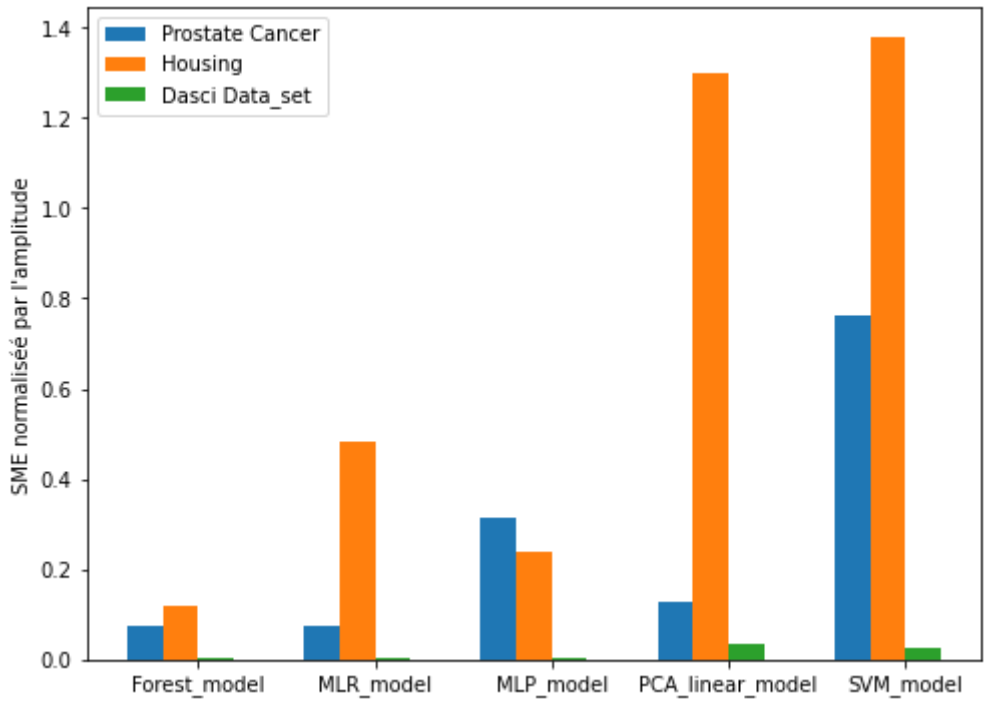

Figure 6 : EQMN des différents algos pour les 3 datasets

On constate que les EQM semblent suivre les mêmes tendances. Elles sont minimales pour les modèles de forêts de régression, plus importantes pour les réseaux de neurones et bien plus grandes pour la régression linéaire post-PCA et les SVM.

La PCA peut donc n'être efficace que dans certains cas, il est donc intéressant de la tester mais ce n'est pas une méthode à prioriser.

Les SVMs étant à la base une méthode de classification, il est compréhensible que les résultats soient si décevants pour une régression.

De plus, nous pouvons observer la plus au moins linéarité de nos données, pour des données plus linéaires comme le dataset du cancer de prostate, les méthode de MLR, PCA et SVM se comportent mieux que pour par exemple les données de Housing qui semblent bien moins linéairement liées. D'où une meilleure performance pour ce dataset sur un réseau de plus de couches.

Comme pronostiqués au début de ce document, les arbres de régression semblent être de loin les méthodes les plus précises.

Évaluons maintenant la complexité de ces différents modèles.

Le temps de calcul brut et normalisé:

Nous cherchions à évaluer en plus de la performance des différents modèles, leur complexité. Ainsi, nous pourrions trouver un compromis entre la précision des algorithmes et leur temps de calculs. Nous voulions aussi avoir une même base comparative pour ces temps. En effet, le troisième dataset étant bien plus grand que les deux premiers, il est normal que les entrainement prennent plus de temps. C'est pour cela que nous avons divisé ce temps par le nombre de donnés des datasets (nombre de colonnes x nombre de lignes).

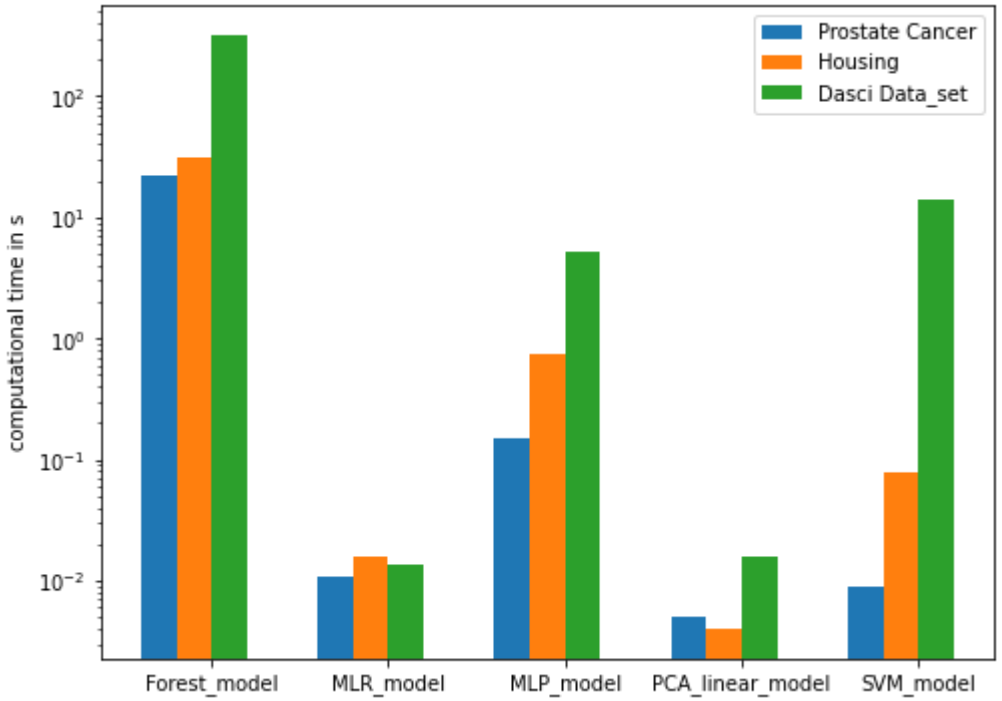

Figure 7 : Temps de calcul de l'entraînement des différents algorithmes pour les 3 datasets

Nous constatons avant la normalisation que le 3ème dataset met souvent plus de 10 à 100 fois plus de temps pour l'entraînement, ce qui est normal vu sa taille.

La méthode analytique par PCA reste la méthode la plus rapide, les SVM étant en complexité liés à la taille des données, sa croissance suit celle du nombre de données.

La méthode la plus longue reste celle donnant la meilleure performance : les arbres de décisions. Nous observons aussi que le temps d'entraînement des réseaux de neurones est bien supérieur et proportionnel à la taille des données.

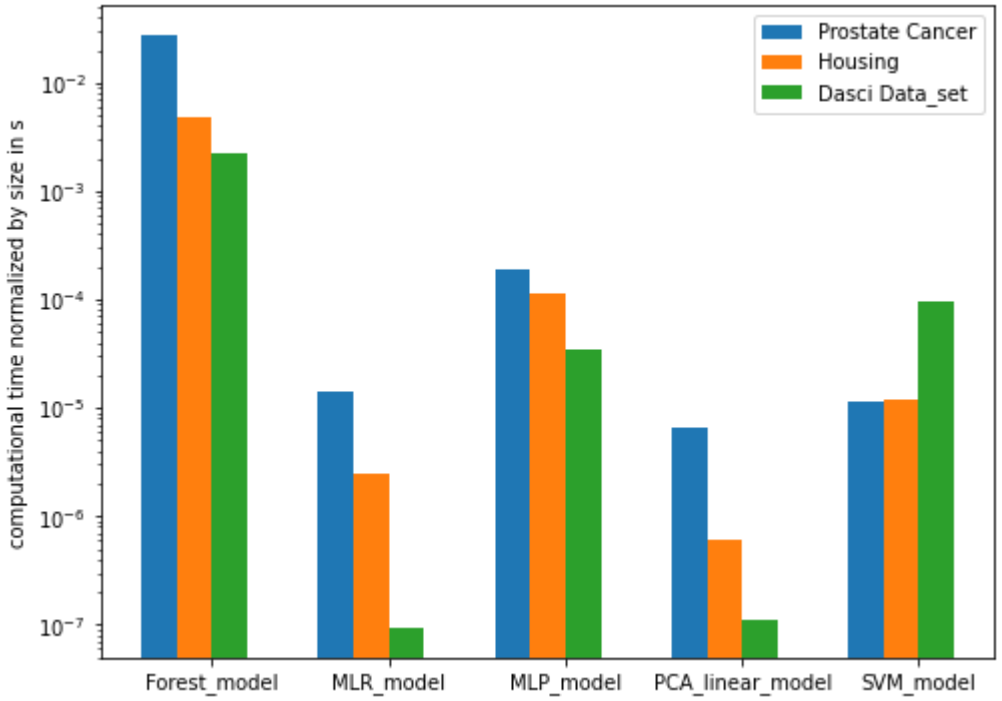

Figure 8 : Temps de calcul de l'entraînement normalisé des différents algos pour les 3 datasets

En normalisant le temps par la taille des données, nous observons des temps similaires pour les méthodes non-analytiques. La PCA par exemple, semble à nouveau être très rapide.

Les forêts de décisions bien que beaucoup plus justes mettent ici environs 100 fois plus de temps que les autres modèles.

Pour les réseaux de neurones, les temps sont similaires sauf pour le perceptron simple. Le temps de celui-ci pour une taille donnée est plutôt bas contrairement au réseau avec plus de couches où le temps semble rester plus ou moins constant.

V. Conclusion

Du point de vue algorithmique, nous pouvons retenir que les arbres bien que très coûteux en complexité vont au moins pour ces trois jeux de données et donc d'autres jeux similaire faire des modèles de régressions très satisfaisants.

Pour ce qui est de notre test avec les SVM, nous pouvons comprendre pourquoi ce n'est pas une méthode répandue, elle s'avère complexe et inefficace.

Finalement, pour les autres méthodes, leur performance varie en fonction de la linéarité du dataset, et donc les garder dans notre code est intéressant parce qu'elles pourront être efficaces.

De plus, la PCA pourrait nous donner une bonne visualisation de nos données.

Nous pouvons donc être satisfait de notre code du point de vue analytique.

D'autre part, nous avons pris plaisir à travailler sur Github. En répartissant le travail, nous avons pu tous contribuer à différents moments sans avoir besoin de clé USB ou d'être sur le même ordinateur. Il s'agit d'un logiciel que nous utiliserons maintenant car nous comprenons son fonctionnement.

Finalement du point de vue des bonnes habitudes du code, nous avons essayé d'exprimer clairement les fonctions et nous espérons que vous trouverez les fichiers clairs et intuitifs.