Prédiction du prix des AirBnB à Berlin

3A CS Mention IA

Vivien Conti, Tristan Basler, Mathian Akanati, Clément Boulay
Lien git, https://gitlab-student.centralesupelec.fr/clement.boulay/projetapprauto

I. Introduction

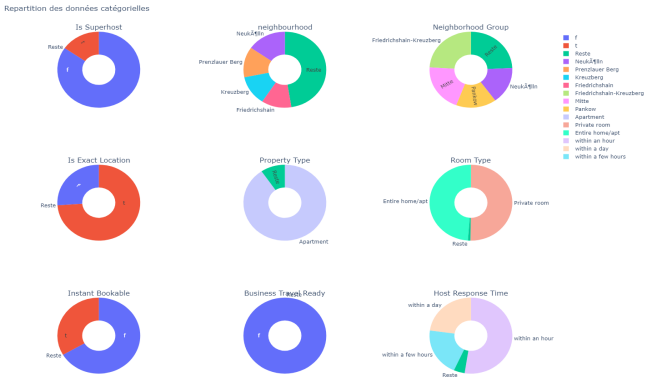
Ce rapport prend place dans l'évaluation du cours d'apprentissage automatique de troisième année à Centrale-Supélec. Il a pour but d'évaluer nos connaissances et compétences en apprentissage automatique.
Le sujet étudié est la prédiction de prix d'AirBnB à Berlin. Notre base de données provient du data challenge suivant. Notre objectif est de pré-traiter les données puis d'implémenter plusieurs modèles de régression sur ces dernières, puis de discuter les résultats obtenus.

II. Exploration des données

Nous proposons de commencer par la description des variables explicatives (*features*), groupées en colonnes (les individus sont disposés en lignes) dans la matrice de design que nous noterons X. Ensuite, nous nous pencherons sur l'analyse de la variable à prédire, notée y. Nous utiliserons l'écriture $\hat{f}(X) = y$
Notre jeu de données est composé de 15692 lignes et 39 colonnes. Les données à prédire sont contenues dans la colonne "Price". Les colonnes "Listing ID", "Listing Name", "Host ID", "Host Name" contiennent des informations macroscopiques, uniquement utiles à l'identification des individus, et ne possède pas de pouvoir prédictif. Elles seront donc mises de côté et gardées pour l'identification des individus au besoin. D'autre part, les colonnes "Country", "Country Code", "City" ne prennent chacune qu'une seule valeur. La variance de ces features étant nulle, elles ne possèdent pas non plus de pouvoir prédictif et on peut également les retirer du modèle.

A. Données d'entrées

La matrice X contient 19 features numériques, 3 features temporelles et 9 catégorielles (dont 2 ordinales, "Room Type" et "Host Response Time", et 7 nominales). Le jeu de données contient en tout 10.5% de valeurs manquantes. Ces dernières se trouvent majoritairement dans les colonnes "Host Response Time" et "Host Response Rate" (45%) ainsi que dans les colonnes relatives au *rating*, comme "First Review", "Last Review" ou "Value Rating" (18% cumulé). Il faudra par conséquent procéder à des *data imputation* lorsque c'est possible afin de pouvoir fournir ces données à certains modèles.
Voici la répartition des données catégorielles :



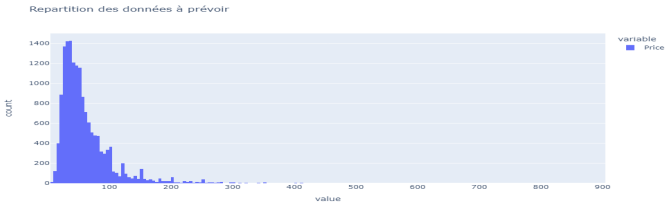
On remarque immédiatement donc la disparité dans les distributions. En effet, certaines variables sont équilibrées ("Neighborhood Group") quand d'autres sont dominées ("Property Type").

Les données numériques se répartissent comme suit :

D'une part, on distingue les variables telles que "Accomodates" ou "Host Since" qui se répartissent de façon lisse. D'autre part, les variables telles que "Guests Included" qui sont largement dominées par une valeur ou les variables telles que "Overall Rating" qui sont distribuées continuement à l'exception d'un pic (autour de 0.9 dans ce dernier cas). Il faudra prendre en compte ces disparités lors du découpage des données, par exemple en utilisant un découpage stratifié, comme abordé en TD, afin de préserver les caractéristiques du jeu de données.

B. Données à prédire

On fait ici le choix d'effectuer une *listwise deletion* lorsque la *target* d'un individu est manquante. La répartition de la variable à prédire est la suivante :



La moyenne est à 60.30 ($/nuit), l'écart-type à 48.90 ($/nuit). C'est une distribution à queue lourde, qui prend des valeurs extrêmes avec une probabilité non-nulle, ce qui complique la prédiction.
En conclusion, les données sont globalement inégalement réparties et de nombreuses données manquent. On pourra se reposer sur les distributions empiriques pour nous aider lors des *data imputation*. Nous avons fait le choix de conserver le maximum de features, quand bien même certaines possèdent de nombreuses données manquantes. Si nous nous apercevons que certaines features "imputées" nuisent à l'apprentissage, elles seront retirées naturellement (*via* PCA ou méthodes de "shrinkage", ou manuellement).

III. Preprocessing

Le jeu de données est structuré (en features et individus) mais n'est ni nettoyé ni pré-processé (features ne sont pas normalisées par exemple). La *data imputation* intervient après le découpage en jeux d'entraînement et de test.

A. Gestion des données manquantes

Pour les données numériques, nous avons implémenté deux méthodes pour inférer les données manquantes.

- Remplissage par moyenne :
  Les valeurs manquantes d'une colonne sont remplacées par la moyenne de la colonne (calculée en omettant ces dernières). L'*imputer* est fitté sur le jeu d'entraînement, c'est-à-dire que ce sont les moyennes des features de ce jeu d'entraînement qui sont insérées, à la fois pour les données manquantes du jeu d'entraînement et pour celles du jeu de test.

- Remplissage par médiane :
  Les valeurs manquantes d'une colonne sont remplacées par la médiane de la colonne (calculée en omettant ces dernières). Le commentaire est le même que celui de la section précédente.

B. Gestion des données numériques

Les variables prédictives numériques sont initialement encodées sous forme de chaînes de caractères. Il faut donc les convertir en float. Ensuite, la colonne "Host Response Rate" contient des pourcentages, que nous remplaçons par des valeurs dans [0-1]. Pour le reste des colonnes, après la *data imputation*, nous procédons à une normalisation.

C. Gestion des données catégorielles

Une seule méthode de *data imputation* est implémentée : celle de la valeur catégorique la plus présente dans la colonne. Cette méthode a l'avantage d'être simple (et nous manquions de temps) mais le désavantage de renforcer le déséquilibre des features déjà dominées. Trois méthodes d'encodage des données catégorielles sont implémentées :

- *HotOneEncoding* :
  Une colonne prenant N valeurs est transformée en N colonnes prenant des valeurs dans $\{0, 1\}$. Chacune des catégories (ou valeurs) d'une colonne donne donc naissance à une nouvelle feature qui prendra la valeur 1 si l'individu était de cette catégorie et 0 sinon.
  Cette méthode a pour avantage d'augmenter l'information contenue dans une colonne, mais fait fortement augmenter le nombre de features.

- *LabelEncoding* :
  Il s'agit de donner une valeur numérique unique à chaque catégorie rencontrée : la première catégorie rencontrée aura la valeur 1, la deuxième 2 et ainsi de suite.
  Une absence de données aura la valeur $n + 1$, avec n le nombre de classes.

- *OrdinalEncoding* : Il s'agit du même principe que le LabelEncoding. Cependant, à l'inverse du LabelEncoding, la valeur des classes est prédéfinie par un mapping que nous spécifions et qui respecte l'ordinalité.

D. Gestion des données temporelles

Dans notre jeu de données, trois colonnes contiennent des données temporelles au format $dd - mm - year$. Cependant, de nombreux modèles (comme les arbres) ne supportent pas ce format. Ainsi, nous avons choisi de linéariser ces colonnes en trois sous-colonnes : une pour l'année, une pour le mois et une pour le jour.
Cet encodage permet d'utiliser des arbres mais dilue l'information contenue dans une colonne temporelle dans trois colonnes.

E. Séparation en jeux d'entraînement/validation et de test

Pour cette partie, nous utilisons la fonction *train_test_split* de Scikit-Learn pour créer les datasets suivants X_train, X_test, y_train et y_test. On choisi de garder 80% des données pour la phase d'entraînment et 20% pour la phase de test. On renseigne donc le paramètre *test_size* avec la valeur 0.2. Pour éviter les biais de disposition, un pré-mélange des données est réalisé par un *stratify* selon la colonne "Overall Rating". Abordée dans le TD1, le *stratify* permet de s'assurer que les jeux d'entraînement et de test possèdent une même proportion d'individus pour chaque tranche de "Overall Rating", ce qui permet de maintenir une certaine consistance lors du passage de la phase d'entraînement à la phase de

test. Pour la reproductibilité, une *random_state=42* est passée afin de pouvoir comparer les modèles implémentés entre eux (ce qui est impossible si chaque modèle voit des données d'entraînement différentes !) et avec les autres personnes du challenge. Finalement, nous disposons de 10037 entrées d'entraînement et 2509 de test.

F. Scaling

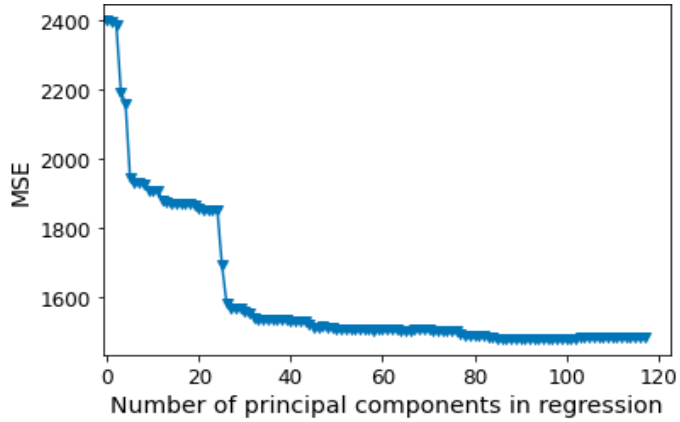
Pour diminuer le temps de calcul durant la phase d'entraînement, nous effectuons, de manière séparée, un scaling identique sur les datasets X_train et X_test. Les deux datasets doivent en effet être scalés séparément pour éviter de créer un biais et d'augmenter l'information dans le dataset d'entraînement. Deux méthodes de *scaling* sont implémentées :

- *StandardScaler* :
  Normalise les données par colonne en soustrayant à chaque individu la moyenne de la colonne, puis en divisant cette nouvelle valeur par la variance de la colonne. Particulièrement adapté quand la feature est déjà distribuée en gaussienne, comme "Longitude".

- *MinMaxScaler* :
  Normalise les données entre 0 et 1 en divisant l'écart de chaque donnée à la valeur la colonne par l'écart entre les données min et max de la colonne. Adapté lorsque le *StandardScaler* ne s'applique pas.

IV. Réduction de dimension : PCA

Parmi les différentes méthodes de réduction, nous avons choisi d'utiliser l'algorithme PCA. La scalibilité des données préprocessées permettent un bon fonctionnement de l'algorithme. Nous avons choisi M = 27 composantes principales.



Sélectionner 25 principales composantes réduit d'un quart le nombre de colonnes par rapport à ce que l'on obtenait à la fin du preprocessing.

V. Description des Modèles

Nous avons choisi d'implémenter de nombreux modèles afin de nous entraîner à leur mise en place et de pouvoir comparer leurs résultats. Ainsi, nous avons implémenté 8 modèles.

A. Arbres

Nous avons choisi d'implémenter le maximum de techniques d'arbres possibles. Dans notre cas, chaque feuille des arbres est associée à une fonction de régression qui prédit le prix de l'AirBnB. Voici les différents modèles que nous avons étudiés :

- Decision Tree, nous avons choisi d'effectuer un GridSearch :
  - splitter : ["best","random"],
  - max depth : [3,5,7,9,11],
  - min samples leaf : [3,4,5,6,7,8,9],
  - min weight fraction leaf : [.1,.2,.3,.4,.5,.6,.7,.8,.9],
  - max features : [auto , log2 , sqrt ,None],
  - max leaf nodes : [None,10,20,30,40,50,60,70,80,90]

  RMSE : 39.4

- Random Forest, nous avons choisi d'effectuer un GridSearch :
  - bootstrap : [True, False],
  - max depth : [10,20,30,40,50,60,70,80,90,100,None],
  - min samples leaf : [1,2,4],
  - min samples split: [2, 5, 10],
  - max features : [auto , sqrt ,None],
  - n estimators: [2, 4, 6, 8, 10, 12, 14, 16, 18, 20] (x100)

  RMSE : 37.4

- Bagging, nous avons choisi d'effectuer un GridSearch :
  - base estimator: [None,SVR(),LinearRegression(), KNeighborsRegressor()],
  - n estimators : [20,50,100],
  - max samples : [0.5,1.0, n samples//2,],
  - max features : [0.5,1.0, n features//2,],
  - bootstrap : [True, False],
  - bootstrap features: [True, False]

  RMSE : 36.8

- Boosting Tree, nous avons choisi d'effectuer un GridSearch :
  - learning rate : [0.01,0.02,0.03,0.04],
  - subsample : [0.9, 0.5, 0.2, 0.1],
  - n estimators : [100,500,1000, 1500],
  - max depth : [4,6,8,10]

  RMSE : 36.4

- Adaboost, nous avons choisi d'effectuer un GridSearch :
  - base estimator : [DecisionTreeRegressor (maxdepth = i) for i in range(4,11)],
  - learning rate: [i/10 for i in range(1,6)],
  - n estimators : range(50, 400, 50),

  RMSE : 36.3

- Xgboost est une libraire implémentant des arbres avec un gradient boosting similaire à Adaboost. Ell est à ma connaissance plus performante que sklearn. Nous avons choisi d'effectuer un GridSearch :
  - max depth : [i for i in range(4,11)],

- learning rate : [i/10 for i in range(1,6)],
- n estimators : range(50, 400, 50),
- sampling method : ["uniform","gradient based"],
- num parallel tree : [1,2,3]

RMSE : 34.2

B. Régression linéaire avec pénalisation L2 (Ridge)

Nous avons déjà utilisé une méthode de réduction de dimension à ce stade : PCA. Toutefois, on peut aussi utiliser les "shrinkage methods". L'une d'entre elles est la régression linéaire avec pénalisation L2 (ou régression "Ridge"). Son homologue pour la pénalisation L1 s'appelle la régression "Lasso" (pas implémentée faute de temps). Scikit-Learn propose une prise en compte de la pénalisation nativement avec *SGDRegressor()*. Nous avons choisi l'erreur quadratique comme fonction de perte et une pénalisation L2, puis implémenté une grid search pour le "learning_rate" et α, le paramètre contrôlant la force de la pénalisation. Les paramètres du Grid Search sont :
"learning_rate" $\in$ np.logspace(-5, 1, 10)
$\alpha \in$ np.logspace(-7, -3, 5)
Le minimum de RMSE est atteint pour "learning_rate" = 10^{-3} et α = 10^{-6} et vaut RMSE = 36.91 ($/nuit). La variable prédictive avec le plus grand coefficient dans la régression Ridge est : "Accomodates". Viennent ensuite "Bathrooms" puis "Bedrooms".

C. Support Vector Machines pour la régression

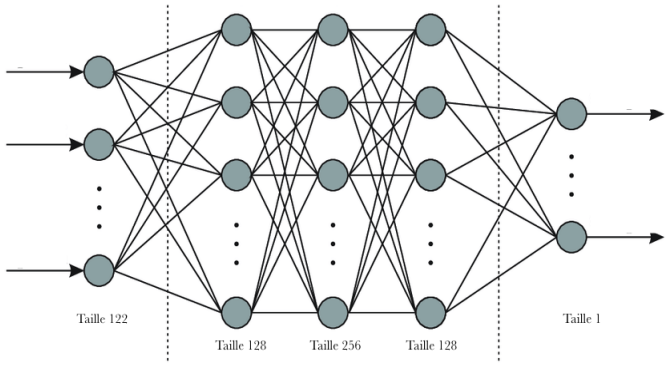
Nous avons ensuite implémenté un Support Vector Machine pour la régression (depuis Scikit-Learn, *SVR()*). D'abord, nous avons utilisé un noyeau linéaire (donc un SVM linéaire, correspondant de base vu en cours) et recherché la meilleure valeur de l'hyperparamètre C (qui contrôle la force de la pénalisation liée au relâchement de la contrainte de Hard-Margin) du SVM dans l'ensemble 0, 5, 10, 20, 100, 500, 1000. Puis avons procédé par dichotomie pour trouver le C optimal. Note : ici, nous ne cherchons à optimiser que C pour un type de noyau fixé. Techniquement, c'est donc un "line search" et pas un "grid search".
Le C optimal est trouvé pour $C = 10$ et correspond à un RMSE de 38 ($/nuit). Nous avons ensuite essayé un noyau de type polynomial et procédé par grid search pour le fine-tuning des hyperparamètres C et "coef0" (nous avons fixé le degré à 3, mais avec plus de temps il aurait fallu regarder plus de degrés. Un noyau polynomial apporte un léger incrément de performance. Nous avons déterminé que la meilleure combinaison de paramètres était $C = 100$ et "coef0" = . Le RMSE correspondant est $\$/nuit$.

D. Réseau de neurones

Afin de terminer notre exploration de modèles, nous avons implémenté un réseau de neurones grâce à la bibliothèque *sklearn*. Pour cela, nous allons tout d'abord utiliser l'objet *MLPRegressor* pour créer le réseau puis la fonction *GridSearchCV* afin d'optimiser ses paramètres. Intéressons nous donc à l'optimisation des paramètres suivants :

- Solver : Représente l'optimiseur chargé de l'optimisation des poids à chaque passage dans le réseau pour minimiser la loss. Les valeurs possibles sont {'lbfgs', 'sgd', 'adam'}.
- Alpha : Défini le facteur de régularisation afin de diminuer la complexité et ainsi l'overfitting du réseau. La technique utilisée ici est la $L2Regularisation$.
- Hidden_layer_sizes : Défini la taille des couches cachées dans le réseau.
- Activation : Représente la fonction d'activation utilisée pour casser la linéarité du problème. Les valeurs possibles sont {'identity', 'logistic', 'tanh', 'relu'}.
- Max_iter : Détermine le nombre d'époques dans le réseau.
- Learning_rate_init : Défini le pas d'apprentissage du réseau.

L'objectif de la fonction *GridSearchCV* va être de tester la performance du modèle en fonction des paramètres. Par défaut, 20% du dataset est réservé pour effectuer une cross validation. Les performances sont ensuite moyennées pour déterminer quel jeu de paramètres est optimal. Pour résoudre notre problème de régression, nous avons choisi la *RMSE* comme fonction de coût à minimiser.
Nous nous sommes d'abord concentrés sur l'optimisation du solver et de la fonction d'activation. Pour cela nous avons utilisé une structure en diamant avec peu de couches (trois couches cachées). La taille des couches est choisie en puissance de deux.



Pour cette première optimisation, les autres paramètres cités ci-dessus prennent leurs valeurs par défauts. Les résultats montrent que le solver $adam$ et la fonction d'activation $relu$ sont plus performants. Nous garderons donc ces paramètres fixés avec ces valeurs pour les optimisations suivantes. Aussi, la structure du réseau reste inchangée. Nous devons à présent renseigner un périmètre de recherche pour les paramètres à optimiser. Les différentes valeurs qu'ils peuvent prendre sont résumées dans le tableau suivant :

α	max_iter	$learning\_rate\_init$
$\mathbf{10^{-4}}$	10^3	10^{-3}
10^{-5}	10^4	$\mathbf{10^{-4}}$
10^{-6}	10^5	10^{-5}

Les meilleures performances sont alors obtenues lorsque le taux de régularisation $\alpha = 10^{-4}$ et le pas d'apprentissage $learning\_rate\_init = 10^{-4}$. Avec ces paramètres, nous obtenons un score $RMSE = 36.59$. On se rend compte que le paramètre max_iter n'a pas de poids dans l'optimisation de notre modèle et nous le fixerons donc à 10^3 par simplicité.

Finalement, il nous reste à nous intéresser au nombre de couches cachées dans le réseau. Pour cela, nous allons garder la même démarche et fournir un périmètre de recherche à la fonction *GridSearchCV* pour le paramètre $hidden\_layer\_sizes$. Le but ici, est de tester des structures plus ou moins complexes et donc de faire varier le nombre et la taille des couches. Nous considérerons la notation suivante $hidden\_layer\_sizes = (x, y, z)$ signifiant que le réseau possède cinq couches avec une d'entrée de taille nombre de features (122), une de sortie de taille 1 et trois couches cachées de taille respective x, y et z. Nous avons d'abord testé d'aborder des structures plus complexes avec quatre ou cinq couches cachées :

$hidden\_layer\_sizes$
$(\mathbf{128}, \mathbf{64}, \mathbf{16})$
$(128, 64, 32)$
$(128, 64, 64)$
$(128, 32, 16)$
$(128, 64, 32, 16)$
$(256, 128, 64, 16)$
$(256, 512, 256, 128, 16)$
$(256, 512, 256, 128, 64, 16)$

Nous remarquons que les modèles complexes avec un nombre plus élevé de couches sont moins performants que les modèles plus simplistes contenant trois couches. La $RMSE$ est la plus faible pour l'architecture $(128, 64, 16)$ pour laquelle on obtient $RMSE = 36.57$.
Pour conclure cette partie, nous pouvons dire que le modèle n'est pas très performant malgré son optimisation. En effet, sachant qu'une grande partie des AirBnB étaient distribués entre $20\$/nuit$ et $36\$/nuit$, l'incertitude est au moins de 100% pour cette échantillon.

VI. Analyse et conclusion

En conclusion, ce projet nous a démontré l'importance de l'exploration et du pré-traitement des données et la part majeure que ces derniers prennent dans la réussite d'un modèle de Machine Learning. En effet, la plupart des modèles que nous avons implémentés n'auraient pu fonctionner et s'entraîner sans des données nettoyées et formatées en entrée. Le pré-traitement est un partie chronophage mais primordiale pour la suite du schéma global du projet.
Ensuite, tester plusieurs modèles peut prendre du temps du fait du grid search et de la validation. Nous avons la chance d'avoir un panel de modèles disponibles et pré-implémentés dans Scikit-Learn qui facilite l'itération rapide et efficace de ces derniers sur notre jeu de données.
Finalement, malgré les divers modèles testés, nous restons sur des résultats plutôt médiocres. Nous expliquons cela par la présence d'outlayers dans les données à prédire. En effet, lorsque nous étudions les erreurs dans nos prédictions, nous observons des valeurs extrêmes à l'intérieur.
Une autre explication probable est le fait que notre problème est plus complexe que nos données. En effet, il n'est pas rare de choisir son AirBnB en fonction des photos disponibles, informations manquantes ici. De plus, de nombreuses études ont montré que les prix d'AirBnB ne respectent que très peu de régles et donc qu'il est difficile de les prédire et même de les justifier.

Enfin, nous tenions à remercier Mme. Tami pour l'ensemble des cours dispensés ce semestre ainsi que Mr. Quercini pour son encadrement lors des travaux dirigés.