Objectif

Création d'une application web permettant de prédire le prix d'un bien immobilier à partir de certaines caractéristiques.

Ressources

Pour réaliser ce TP vous aurez besoin d'avoir Python installé sur votre ordinateur, ainsi que de pouvoir ouvrir et exécuter un Jupyter notebook. Nous vous conseillons d'installer VsCode, un éditeur de code développé par Microsoft, afin de faciliter ce travail.

De plus vous aurez besoin d'être familier avec les éléments suivants:

- `Scikit-Learn` qui est une librairie Python permettant de facilement utiliser des modèles de Machine Learning.
- `Pickle` permet de sérialiser des objets Python (transformation d'une structure Python en un fichier pouvant être stocké puis reconstruit).
- `Pandas` une librairie permettant de manipuler des DataFrame très facilement.
- `Flask` est un framework de développement d'appication Web idéal pour créer des APIs.

Afin d'installer facilement tous les packages nécessaires vous pouvez utiliser la commande:

```
pip install -r requirements.txt
ou
python3 -m pip install -r requirements.txt
```

Données

Les données utilisées pour notre TP sont issues du jeu de données publique des valeurs foncières françaises de 2021. Ces données ont été retravaillées pour vous et se trouvent dans le fichier `data.csv` .

Déroulement du TP

Partie 1: création du modèle

Dans un premier temps nous supprimerons la variable `postcode`

1. Pour l'instant le preprocessing de nos données se limite à un encodage de la variable `house_type` . Complétez la fonction `transform_house_type` (de manière simple, on cherche à créer une fonction d'encodage de notre variable)

2. Lors de ce tp nous ne cherchons pas à optimiser notre modèle avec les meilleurs hyperparamètres. De ce fait, séparez vos prédicteurs de la valeur à prédire et entraînez un modèle linéaire sur toute les données. Vous pouvez utiliser la régression linéaire.

3. Lorsque votre modèle est entraîné, sauvegarder ce dernier au format `pickle` , appelez le `model.pkl` par exemple. Cette méthode vous permet de sauvegarder votre modèle entraîné dans un fichier qui pourra être utilisé dans notre application web.

Partie 2: création d'une webapp

Afin de développer le backend de notre application web nous utilisons le framework Flask. L'objectif de cette partie est de comprendre le fonctionnement des serveurs Web et de s'en servir pour appeler notre modèle et faire de l'inférence.

1. Lancez le fichier `app.py` avec la commande `python3 app.py` . Votre serveur web va se lancer et être accessible via des requêtes HTTP. Vérifiez que la connexion à votre serveur fonctionne (Linux/Mac users):

```
curl localhost:5678
```

ou directement depuis votre navigateur http://localhost:5678/hello. Il est possible qu'une URL différente de `localhost` soit affichée dans la console ayant servi à lancer votre programme, dans ce cas utilisez cette dernière.

Notes

- `curl` est un outil nous servant ici de client `HTTP`
- Comme vous pouvez le remarquer lorsque l'on utilise la commande `curl`, la donnée reçue est `<h1>Hello world</h1>`. Lorsque l'on ouvre notre navigateur on peut y voir apparaitre un magnifique **Hello world** formaté. Votre navigateur vous permet de formatter le HTML que vous recevez à l'écran mais ce dernier n'est rien de plus qu'un client HTTP aggrémenté de fonctions d'affichages.

2. Dans le dossier `templates` vous trouverez un fichier *index.html* qui contient un code HTML un peu plus complexe qu'un simple Hello World. Créer une nouvelle route `/app` permettant de renvoyer à l'utilisateur le contenu de la page *index.html*. Il est possible d'utiliser la fonction `render_template` de Flask. Vérifier que votre code fonctionne en vous rendant à l'adresse suivante sur votre navigateur: http://localhost:5678/app.

3. Créer une nouvelle route `/predict` permettant à l'utilisateur de passer des données via un `form` (requête POST ou GET à votre avis ?). Cette route devra effectuer dans l'odre:

- **Lecture des données**: nous transitons l'information de notre client vers notre backend via un formulaire. Flask permet de récupérer ces données dans le corps de la fonction grâce à l'objet request. Habituellement les formulaires sont utilisés pour les applications Web mais dans le cas de simples APIs il est préférable d'utiliser les formats JSON/XML/Protobuf.
- **Vérification de la donnée**: est-ce que la donnée contient tous les champs que l'on souhaite (house_type, nb_room, ...) ? Dans le cas où la donnée est mal formattée, renvoyer un code d'erreur 400.
- **Chargement du modèle**: après avoir sauvegardé votre modèle au format `pickle`, il est temps de le recréer et d'y faire appel.
- **Transformation des données**: appliquer le même preprocessing que lors de l'entrainement à vos nouvelles données. Vous pouvez directement intégrer la fonction `transform_house_type` que vous avez codé à la partie précédente.
- **Prédiction**: utilisez votre modèle chargé pour prédire le prix à partir des données d'entrée
- **Renvoi du résultat**: retournez la valeur prédite ainsi que le status 200.

4. Vérifiez que votre nouvelle route fonctionne:

```
curl -X POST -d "postcode=75&house_type=Maison&house_surface=130&nb_room=5&garden_area=300"  localhost
```

Ou bien testez directement depuis la page web en remplissant le formulaire !

Partie 3: Intégrez le code postal à votre model

Maintenant que vous avez réussi à déployer votre première webapp faisant appel à un modèle de Machine Learning, il est temps d'améliorer un peu tout ça ! L'objectif de cette partie est de rajouter le code postal à notre modèle !

1. Récupérez des données permettant d'enrichir votre jeu de données d'entrainement à partir des codes postaux. Refaire toute la première partie en prenant en compte votre/vos nouvelle(s) variable(s).

2. Retravaillez votre backend pour prendre en compte cette nouvelle variable. Il vous faudra revoir votre `preprocessing` pour y ajouter l'enrichissement de données, et votre étape de `validation` des données.

3. Modifier l'interface web pour ajouter un nouveau champ au formulaire. Un peu de HTML ça ne fait pas de mal 😀

4. Testez votre nouvelle application et vérifiez que les maisons à Paris coûtent très cher !

Partie 4: Pour aller plus loin

Cette section vous propose d'explorer différents sujets auxquels des ingénieurs dans le data sont confrontés. L'objectif est de vous exposer ces problématiques et libre à vous de creuser les sujets qui vous intéressent les plus.

- Aujourd'hui notre application nécessite Python et plusieurs dépendances pour fonctionner correctement. Dans un contexte de production il est courant de ne pas savoir exactement sur quelle machine tourne son programme (ex:

Kubernetes choisi parmi un cluster de machine). Il est impensable de devoir installer toutes les dépendances de tous nos programmes sur toutes les machines ! C'est pour répondre à cette problématique que Docker a été créé. Il permet de créer des images qui vont empaquêter toutes les dépendances et le code nécessaire pour faire fonctionner notre application. L'objectif est de créer une image Docker permettant de faire fonctionner notre application web correctement. Vous pouvez vous inspirer de ce tutoriel.

- `sklearn` a développé un objet `Pipeline` qui permet d'aggréger les étapes de preprocessing et de prediction au sein d'un même objet. Il est tout à fait possible (et même recommandé) de sérializer (avec `pickle` par exemple) le modèle accompagné de sa pipeline de traitement.
    1. Plutôt que d'implémenter plusieurs fonctions comme `transform_house_type`, implémenter une pipeline de traitement en utilisant les transformeurs de `sklearn`.
    2. Rajoutez une étape de standardisation des données numérique. L'objet `StandardScaler` le fera très bien pour vous et vous permettra de conserver l'écart-type et la variance de votre jeu d'entraînement (qui doit être réutilisée pour standardiser vos données de test !!)
    3. Sérialisez votre pipeline et modifier votre route de prédiction pour la simplifier.
- `pickle` est simple d'utilisation mais n'est pas vraiment adapté pour des cas d'usage en production. On préfèrera des formats adaptés à la sérialisation de pipeliene de ML comme `Open Neural Network Exchange` (ONNX). L'objectif est de transformer votre code pour exporter votre pipeline sous ce format.
- Dans notre exemple, à chaque appel à la route `/predict` on charge le modèle et on effectue l'inférence. Dans un exemple très simple comme le notre cela ne pose pas de problème, mais pour des applications plus gourmandes (ex: Deep Learning), le chargement des modèles peut prendre plusieurs minutes. Dans ces cas là comment faire ? Quelques idées à explorer:
    - Implémentez notre code en C++
    - `Quantization` : entrainer un réseau de neurones sur des float32 au lieu de float64 -> réduit la taille du modèle
    - `Pruning` : manière de retirer les couches les moins utiles
    - `Distillation` : entraîner un réseau de neurones plus petit à répliquer les décisions d'un gros réseau
    - Utilisation de GPU pour effectuer l'inférence (pour processer des images ou du texte c'est indispensable)
- Lorsqu'on cherche à améliorer notre modèle il est courant d'augmenter la taille de notre jeu de de données d'entrainement, de rechercher les meilleurs hyperparamètres, de repenser l'architecture, ... De fait, on ne peut pas se permettre de repasser par la phase exploratoire dans un Jupyter Notebook, de réentrainer notre modèle, de le packager et de remplacer le modèle existant par le nouveau. En pratique toutes ces étapes sont automatisées grâce à des pipelines spécifiques. Cette automatisation rentre dans le scope du `MLOps` (Machine Learning Operations). On vous invite à vous renseigner sur ces principes grâce à cet excellent site https://ml-ops.org/. Cette discipline est à cheval entre plusieurs métier: `Data Scientist`, `Data Engineer`, `Software Engineer` et `DevOps`.