Notice d’utilisation

Optimus Price

Arnaud BAZIN DE BEZONS
Sylvain BENISTAND
Jonathan BOUTAKHOT
Loïc CHEONG
Thomas HU
Cécile WANG

Mentor : Jae Yun JUN KIM

Projet de Fin d’Étude PFE
Dynamic Pricing Strategy

ECE PARIS
2020-2021

SOMMAIRE

0 - Les données

La partie des données est en réalité assez simple d'utilisation. L'objectif est de répertorier les liens des villes qui vous intéressent dans le fichier texte "urls.txt". Lorsqu'un nouveau mois s'est écoulé, allez sur le site : http://insideairbnb.com/get-the-data.html puis copier, selon la ville et la date, l'adresse du lien en .csv.gz qui vous intéresse. Mettez à jour le fichier urls.txt en collant le lien copié.

Nous n'avons pas spécifiquement prévu de dossier d'enregistrement des données, donc vous devrez mettre le fichier csv là où les fichier d'exécution python sont placés, mais libre à vous de modifier cela.

Une fois que le fichier répertoriant les urls est mis à jour, ouvrez le fichier airbnb_processing.py pour procéder au téléchargement des données.

a) airbnb_processing.py

Exécutez tout le code pour charger les données localement dans *airbnb_data.csv* sur votre PC.

Fonction get_url_list(path)

Paramètres:

path : chemin de l'emplacement du fichier *urls.txt*

Retourne: Renvoie une liste contenant tous les liens URL dans *urls.txt*

Détails du programme : Une boucle *for* lit chaque ligne de *urls.txt* et l'ajoute la chaîne de caractère (url) dans une liste

Fonction isolate_expected_urls(url_list,list_city)

Paramètres:

url_list : liste de liens URL, obtenu grâce à la fonction **get_url()**

list_city: liste de villes

Retourne: Applique un filtre sur **url_list.** Renvoie une liste de liens URL des villes spécifiées dans **list_city**

Détails du programme : Une boucle *for* parcourt toute la liste **url_list**, scinde le string de l'URL sur "/", récupère la ville et vérifie s'il correspond à une des villes dans **list_city.** S'il y a correspondance, le sting de l'URL est ajouté dans une nouvelle liste.

Fonction get_data(url_list,file_name, list_city)

Paramètres:

url_list : liste de liens URL, obtenu grâce à la fonction **get_url()**

file_name : chaîne de caractère. Nommage du fichier csv à télécharger

list_city: liste de villes

Retourne: Aucun. Traitement et téléchargement de toutes les données dans un fichier *airbnb_data.csv*

Détails du programme : Une boucle for lit chaque lien URL et définit un dataframe pour chaque URL. On applique un filtre pour récupérer les colonnes suivantes : *'id', 'property_type', 'room_type', 'accommodates', 'bedrooms', 'beds','price', 'availability_30', 'number_of_reviews', 'review_scores_rating','review_scores_accuracy',*. Sur certaines URL, il y a des colonnes manquantes comme 'square_feet','cleaning_fee'. Dans ce cas, on les crée nous-même et on complète avec des NaN.

On scinde le string de l'URL sur "/" (exemple : http://data.insideairbnb.com/united-states/ny/new-york-city/2020-12-10/data/listings.csv.gz), on récupère le nom du pays, la ville et la date. Ajouter les colonnes "country","city","date" au dataframe. Puis on ajoute chaque dataframe dans une liste de dataframes pour pouvoir les concaténer plus tard.

Nettoyer et transformer et les colonnes "price" et "cleaning_fee" en nombre flottant.

Ajouter les colonnes :

- "revenue_30": les revenues calculées du mois
- "booked" : nombre de jours réservés durant le mois écoulé

Trier le dataframe par "id" et "date". Supprimer les doublons et enfin télécharger le dataframe en CSV.

Fonction load_data(path)

Paramètres:

path : chemin de l'emplacement du fichier *airbnb_data.csv*

Retourne: Renvoie un dataframe de *airbnb_data.csv*

b) data_analysis.py

Fonction review_data(df_all,city)

Paramètres:

df : dataframe d'*airbnb_data.csv*

city *:* nom de la ville à filtrer dans le dataframe df

Retourne: Renvoie un nouveau dataframe avec la date, room_type, number_of_hosts, mean_price, mean_availability_30

Fonction multiplot_by_roomtype(df,y_data_name,y_label)

Paramètres:

df : dataframe de review_data(df_all,city)

y_data_name : nom de la colonne choisie dans df

y_label : nom de l'axe des abscisses du plot (y_data_name)

Retourne: Renvoie une courbe de y_data_name en fonction de la date pour chaque type de chambre

Fonction plot_by_squarefeet(df)

Paramètres:

df : dataframe d'*airbnb_data.csv*

Retourne: Renvoie un scatter plot, un nuage de prix ($0 - $1,500) en fonction de la surface (0 - 3,500 square feet) pour chaque type de chambre

Fonction get_trend_on_scatter(df)

Paramètres:

df : dataframe de review_data(df_all,city)

Retourne: Renvoie un scatter plot, un nuage de prix ($0 - $1,500) en fonction de la surface (0 - 3,500 square feet) avec une grossière régression linéaire

Fonction test()

Exécution de tout le programme des fonctions ci-dessus

1 - Côté DQN

Pour installer PyTorch, suivre les instructions sur https://pytorch.org/

1) Deep_Network.py

Définit la classe DeepQNetwork qui crée un réseau de neurones artificiel.

1. classe DeepQNetwork(nn.Module)

Fonction __init__(state_dim, action_dim, seed=2020, nb_node1=128, nb_node2=128)

Paramètres :

state_dim : dimension des états

action_dim : dimension des actions

seed : définit la graine aléatoire

nb_node1 : dimension des données que renvoie la couche 1 du réseau, par défaut = 128

nb_node2 : dimension des données que renvoie la couche 2 du réseau, par défaut = 128

Variables de la classe :

self.seed : récupère la graine en Paramètres

self.layer1 : couche 1 du réseau, reçoit les données d'input

self.layer2 : couche 2 du réseau, une couche cachée

self.layer3 : couche 3 du réseau, renvoie les données d'output

Détails du programme : Appel la fonction __init__() de nn.Module afin d'utiliser ses fonctions et propriétés.
`super(DeepQNetwork, self).__init__()`
Définit la graine aléatoire du générateur de nombre de pytorch
`self.seed = torch.manual_seed(seed)`
nn.Linear permet de créer des couches linéaires (linéaire car une transformation linéaire est appliquée aux données d'entrée). Nous avons créé 3 couches de réseau:

- couche 1: reçoit les données d'entrée de taille state_dim et renvoie des données de taille nb_node1
  `self.layer1 = nn.Linear(state_dim, nb_node1)`
- couche 2: reçoit les données de la couche 1 et renvoie des données de taille nb_node2
  `self.layer2 = nn.Linear(nb_node1, nb_node2)`
- couche 3: enfin la dernière couche récupère les données de la couche 2 et renvoie les données de sortie de taille action_dim
  `self.layer3 = nn.Linear(nb_node2,action_dim)`

Fonction forward(state)

Paramètres :
state : les données d'entrées
Retourne :
Renvoie les données en sortie du réseau
Détails du programme : ReLu est la fonction d'activation linéaire par morceau suivante : ReLU = max(0,x). Les fonctions d'activation usuelles en réseaux de neurones profonds étaient la sigmoid et la tangente hyperbolique, mais ont été remplacées par la ReLU pour sa simplicité de calcul (lors du calcul du gradient, la dérivé est constante/nul pour des valeurs positives/négatives) et donc sa rapidité de calcul.
Elle permet de traiter l'information qui arrive à une couche de neurone artificiel

- l'information 'state' arrive à la couche 1 et est traitée grâce à la fonction ReLU
  `l1 = F.relu(self.layer1(state))`
- l'information l1 arrive à la couche 2 et est traitée grâce à la fonction ReLU
  `l2 = F.relu(self.layer2(l1))`
- Enfin, l'information passe dans la couche 3 et on récupère les données en sortie
  `self.layer3(l2)`

2) Eps_Greedy_Policy.py

Définit la classe Epsilon Greedy Policy pour créer une politique. La politique recommande l'action à prendre à partir d'un état donné.
La politique epsilon greedy choisit l'action via 2 axes: l'exploration et l'exploitation. En particulier en début d'apprentissage, cette politique recommande d'explorer avant d'exploiter. Au fur et à mesure que des actions sont prises, la politique recommande de plus

en plus l'exploitation des données. Cette frontière qui sépare le choix entre l'exploitation et l'exploration est le seuil d'epsilon, que l'on va faire basculer d'un axe à l'autre au cours de l'apprentissage.

1. classe EpsilonGreedyPolicy

Fonction __init__(self, `eps_start=0.99, eps_end=0.01, eps_decay=1000`)

Paramètres :

eps_start : epsilon de départ, par défaut = 0.99

eps_end : epsilon de fin, par défaut = 0.01

eps_decay : facteur de décroissance d'epsilon

Variables de la classe :

self.eps_start : récupère l'epsilon de départ dans les Paramètres

self.eps_end : récupère l'epsilon de fin dans les Paramètres

self.decay : récupère le facteur de décroissance dans les Paramètres

self.steps_done : définit le nombre d'action qui ont déjà était prise, par défaut = 0

Fonction select_action(self, q_values)

Paramètres :

q_values : les données en sortie du réseau sont la valeur de Q pour chaque paire de (state, action) possible pour un état (= state) donné.

Retourne : Renvoie l'indice de l'action à prendre

Détails du programme : Un epsilon proche de 1 signifie une grande chance de prendre une action par exploration et inversement un epsilon proche de 0 signifie une grande probabilité de choisir l'action par exploitation.

Par défaut, le seuil d'epsilon varie entre les bornes [0.99, 0.01[selon la formule de décroissance d'epsilon suivante :

```
eps_threshold = self.eps_end + (self.eps_start - self.eps_end) \
      * np.exp(- 1. * self.steps_done / self.decay)
```

Ainsi pour un nombre aléatoire 'r' entre 0 et 1, si :

- r > seuil d'epsilon : exploitation des données
- r < seuil d'epsilon : exploration des données

Mathématiquement, l'exploration signifie prendre une action aléatoire tandis que l'exploitation signifie prendre l'action pour laquelle la 'q_values' est maximisée

Fonction select_action_test(self, q_values)

Paramètres :

q_values : les données en sortie du réseau sont la valeur de Q pour chaque paire de (state, action) possible pour un état (= state) donné.

Retourne : Renvoie l'indice de l'action à prendre

Détails du programme : Cette fonction permet de choisir l'action lors de la phase de test, lorsque le réseau est déjà entraîné et optimal. Ainsi le choix de l'action ce fera donc toujours par exploitation des données

3) Replay_memory.py

Définit une classe ReplayMemory permettant d'enregistrer en mémoire des expériences et d'en extraire un échantillon pour l'apprentissage.

1. classe ReplayMemory

Fonction __init__(self, capacity)

Paramètres :

capacity : la capacité de la mémoire, représente le nombre d'expérience que l'on peut enregistrer dans cette mémoire

Variables de la classe :

self.capacity : récupère la capacité en Paramètres

self.memory : définit la mémoire sous forme d'une liste

self.push_count : définit le nombre d'enregistrement d'expérience faite, initialisé à 0

Détails du programme : namedtuple permet de créer une nouvelle subclasse de tuple de type nommé `'Experience'` , la subclasse est nommé `experience`

```
self.experience = namedtuple('Experience', ('state', 'action', 'next_state', 'reward'))
```

Fonction push(self, state, action, next_state, reward)

Paramètres :

state : l'état actuel

action : l'action prise à l'état actuel

next_state : l'état suivant sachant l'état actuel et l'action prise

reward : la récompense due à l'action prise à l'état actuel

Détails du programme : On enregistre l'expérience associée au ('state', 'action', 'next_state' et 'reward')

```
e = self.experience(state, action, next_state, reward)
```

Dans le cas où la mémoire ne présente plus d'espace libre, la dernière expérience est supprimée et la nouvelle est sauvegardée, d'où le nom de la fonction 'push'.

```
self.memory[self.push_count % self.capacity] = e
```

Fonction sample(self, batch_size)

Paramètres :

batch_size : dimension de l'échantillon dont on veut extraire

Retourne : Renvoie un échantillon composé de 'batch_size' expériences

Fonction __len__(self)

Retourne : Renvoie la taille de la mémoire utilisée, permet de faire des contrôles sur la mémoire

4) Import_data.py

Importation des données en utilisant un fichier Excel.

Fonction get_proportion(b, proportion)

Paramètres:

b : nombre d'appartements totaux après filtrage

proportion : proportion pour l'entraînement

Retourne: le nombre de data utilisé pour l'entraînement

Détails du programme: En récupérant le nombre d'appartement nous multiplions par la proportion afin d'obtenir le nombre d'appartement pour l'apprentissage du réseau.

Fonction get_data(train_proportion, file_name = "airbnb_data_nyc.csv")

Paramètres :

file_name : le fichier contenant les datas provenant du fichier 'airbnb_processing.py'

train_proportion: nombre d'appartement pour l'entraînement

Retourne : Renvoie 2 dataframes : une dataframe des 'prix data_with_all_date' des appartements au cours du temps (2015 - 2020). Chaque ligne est un appartement identifié grâce à son "id". Chaque colonne représente une date mensuelle. Enfin, une deuxième dataframe 'booked_with_all_date' du même format que le premier mais avec la demande sur le mois (entre 0 et 30) à la place du prix. Et renvoie la proportion pour les données train.

Détails du programme : On récupère le fichier csv dans un dataframe pour une simplicité d'utilisation : `result = pd.read_csv(file_name)`

On supprime toutes les lignes en doublon : `result.drop_duplicates(inplace= True)`

On réalise un filtre sur les room_type de type "Entire home/apt", les aggrège par "id".

```
nb_id = result[result["room_type"] == "Entire home/apt"].groupby(["id"], as_index = False).size().reset_index(name="size")
nb_id.sort_values(by=["size","id"], ascending= False, inplace= True)
```

On garde ensuite dans la dataframe 'nb_id', les identifiants de tous les appartements qui présentent un prix sur toute la période 2015-2020

```
nb_id = nb_id[nb_id["size"] >=69 ]
```

On créer une nouvelle dataframe 'df' reprenant uniquement les appartements voulus

```
df = pd.merge(result[result["room_type"] == "Entire home/apt"], nb_id["id"], how = "right" ,on=["id"] )
```

On supprime les doublons potentiels

```
df.drop_duplicates(inplace= True)
```

On pivote la dataframe afin d'avoir par identifiant d'appartement unique les prix et les réservations (qui peuvent être vu comme la demande) au cours de la période. Ainsi chaque ligne est un identifiant, et chaque colonne est une date.

```
data_with_all_date = df.pivot(index="id", columns="date", values = "price")
booked_with_all_date = df.pivot(index="id", columns="date", values = "booked")
```

On crée une dataframe qui renvoie le prix moyen par appartement, et ne garde que les identifiants d'appartement qui ont un prix moyen entre 70 et 350 afin d'écarter les cas extraordinaires (grand prix).

```
mean_price_all_date = df.groupby("id", as_index = False).agg({"price" : ["mean"]})
mean_price_all_date.columns = ["id","mean_price"]
mean_price_all_date = mean_price_all_date[(mean_price_all_date["mean_price"] >= 70) & (mean_price_all_date["mean_price"]<=350)]
```

Grâce à ces identifiants, on filtre les 2 dataframes groupant par prix et par demande 'data_with_all_date' et 'booked_with_all_date'

Fonction training_data(df_price, df_booked, proportion)

Paramètres :

df_price : dataframe des prix par appartement, on le récupère de la fonction get_data ('data_with_all_date')

df_booked : dataframe des demandes par appartement, on le récupère de la fonction 'get_data ('booked_with_all_date')

proportion: proportion pour l'entraînement

Retourne : Renvoie un tableau de 3 colonnes, prix, demande et date.

Détails du programme : Sur la période 2015 à 2020, nous avons décidé de prendre 2015 - 2019 pour les données en training et 2020 pour le test. Nous récupérons les données entre 2015 et 2019 : `data_train = df_price[df_price.columns[0:proportion]]`

Pour chaque appartement, nous allons construire un tableau nommé 'price_grid_total'.

```
for k in range(len(data_train))
```

Ce tableau est constitué de 3 colonnes, prix, demande et date. Nous retrouvons dans les listes : price, demand et date les 3 colonnes que nous concatenons dans price_grid_total

```
price_grid_total.append(np.c_[price,demand,date])
```

Fonction create_data(nb_mois_test)

Paramètres:

nb_mois_test : nombre de mois de projection

Détail programme: création des données via notre modèle pour l'entraînement du réseau de neurone

Fonction create_data2(nb_mois_test)

Paramètres:

nb_mois_test : nombre de mois de projection

Détail programme: création des données plus déterministe via notre modèle pour l'entraînement du réseau de neurone. Le déterministe dans la création est représenté à travers différentes conditions à remplir.

Fonction load_data(path,train_proportion,strat_min,prop,step_prop)

Paramètres:

path: fichier excel

train_proportion: proportion de données pour l'entraînement

start_min_prop: proportion minimum de clients stratégiques

step_prop: proportion comprise entre 0 et 1 du maximum de clients stratégique

Retourne: tableau de données, tableau de données pour le test, proportion de données

Détail programme: Récupérer les données utiles en fonction de la proportion désirée.

5) Model_DQN.py

1. classe DQN

Fonction __init__(self, path, gamma ,learn_rate, train_proportion, strat_min_prop, step_prop, batch_size)

Paramètres :

path : nom du fichier de toutes les données nettoyées

gamma : discount factor, compris entre 0 et 1, valeur représentant l'importance du future par rapport au présent

learn_rate : learning rate, comprise entre 0 et 1, valeur représentant le taux d'apprentissage

train_proportion : pourcentage de donnée que nous souhaitons avoir en entraînement, comprise entre 0 et 1

strat_min_prop : proportion comprise entre 0 et 1 du minimum de clients stratégique

step_prop : proportion comprise entre 0 et 1 du maximum de clients stratégique

batch_size : dimension de l'échantillon

Variable de la classe :
self.price_grid : Tableau de prix, date et de demande, récupération du fichier Import_data et de la fonction training_data(). Ce tableau représente tous les états possibles et l'action se portera sur le choix d'un prix, donc il s'agit d'un indice parcourant les lignes de ce tableau.
self.price_grid_test : Le même tableau que price_grid mais sur la partie que nous souhaitons tester.
self.proportion : Index qui se réfère à la colonne de la dernière donnée dans le train.

self.state_dim : Dimension des états, équivalent au nombre de colonne du 'price_grid'

self.unit_cost : Frais de ménage, d'éléctricité et gaz.

self.device : Définit la machine sur laquelle torch.Tensor est ou sera attribuée. Il existe 2 types : "cuda" ou "cpu"

self.Transition : Définit une nouvelle subclasse de tuple de type nommé 'Transition'

self.policy_net : Définit un objet de la classe DeepQNetwork, avec en dimension des états 2*state_dim et en dimension des actions une fourchette définit.

self.target_net : Définit un objet de classe DeepQNetwork de même Paramètres que la policy_net. Il s'agit d'une copie de policy_net mais dont on ne mettra à jour les pondérations qu'après un lapse de temps, définie plus bas 'TARGET_UPDATE'.

self.policy : Définit la politique de choix d'action, c'est un objet de classe EpsilonGreedyPolicy.

self.memory : Définit la mémoire, c'est un objet de classe ReplayMemory dont la capacité de mémoire est définie par défaut à 100 000.

self.TARGET_UPDATE : Définit la fréquence à laquelle la target_network se met à jour, par défaut = 20

self.GAMMA : Définit le discount factor, détermine l'importance du future par rapport au présent

self.BATCH_SIZE : Définit la taille de l'échantillon d'expérience à prendre de la mémoire

self.optimizer : Définit le type d'optimiseur à l'algorithme d'Adam, une méthode d'optimisation stochastique.

self.T : Définit la période d'une saisonnalité de vente (une saisonnalité étant 1 épisode d'entraînement), définie à 12 mois, ainsi un prix est calculé par mois sur un an. Le nombre d'épisode N définira un entraînement sur N années.

--- Partie training ---

Fonction update_model(self, memory, policy_net, target_net)

Paramètres :

memory : Définit la mémoire, récupère la mémoire en variable de la class
policy_net : Définit la policy network , récupérée des variables de la classe.
target_net : Définit la target network, récupérée des variables de la classe.

Détails du programme : La policy_net est le réseau de politique, il renvoie en sortie les valeurs de q et permet ainsi de choisir l'action en fonction de la plus grande valeur de q. Quant à la target_net, il s'agit du réseau qui renvoie en sortie une approximation de la valeur optimale de q. Ce deuxième réseau permet de stabiliser la fonction d'approximation de la valeur de q.
La fonction update_model n'est appelée que lorsqu'il est possible d'extraire un échantillon de taille 'BATCH_SIZE'.

```
if self.BATCH_SIZE < len(memory):
```

On extrait un échantillon puis sépare dans plusieurs tensors les états, les actions, les récompenses et les états suivants.

```
non_final_next_states = torch.stack([s for s in batch.next_state if s is not None])
state_batch = torch.stack(batch.state)
action_batch = torch.cat(batch.action)
reward_batch = torch.stack(batch.reward)
```

Nous calculons la valeur de q en faisant passer les états au réseau de politique (policy_net)

```
state_action_values = policy_net(state_batch)[:,0].gather(1, action_batch)
```

Nous calculons la valeur de q pour l'état suivant par le réseau cible (target_net)

```
next_state_values = torch.zeros(self.BATCH_SIZE, device=self.device)
next_state_values[non_final_mask]=target_net(non_final_next_states)[:,0].max(1)[0].detach()
```

Et nous pouvons donc calculer la valeur de q espérée par la formule suivante :

```
expected_state_action_values = reward_batch[:, 0] + (self.GAMMA * next_state_values)
```

Nous pouvons maintenant établir la fonction de perte, nous avons choisi la fonction de perte de Huber présentant l'avantage d'être moins sensible aux cas extraordinaires par rapport à la fonction MSEloss et dans certain cas évite l'explosion du gradient.

Fonction env_initial_state(self)

Retourne : Renvoie l'état initial

Détails du programme : On initialise l'état comme un tableau de T lignes et state_dim colonnes. La première ligne correspond au prix, à la demande et la date. Chaque ligne correspond à un instant donné. Ce tableau est initialisé à 0.

Fonction env_step(self, state, action)

Paramètres :

state : état actuel

action : action associé à l'état

Retourne : Renvoie l'état de l'environnement à l'instant future, la récompense et la demande pour l'action prise à cet état donné : next_state, reward, demand_

Détails du programme :

Fonction profit_t_d(self, p_t, demand)

Paramètres :

p_t : price à l'instant t

demand : demande

Retourne : le profit selon la formule prix*exp(demand) - cout_totaux

Détails du programme : Pour calculer le profit nous récupérons la recette total = prix * nombre_de_nuits réservées, auquel nous soustrayons les charges liées à Airbnb unique_cost et share_cost qui prennent en compte le coût par nuit unit_cost.

Fonction demand(self, pt,date)

Paramètres :

pt : prix à l'instant t

date : période de réservation

Retourne : La demande

Détails du programme : Création d'une liste de demande vide. Parcours, la table des états, si le prix et la date passés en Paramètres sont identiques à celui dans la table des états, on récupère la demande associée, qui sera alors copiée dans la liste vide de demande. Après avoir tout parcouru, nous faisons une moyenne de cette demande qui sera alors retournée.

Fonction to_tensor(self, x)

Paramètres :

x : un array

Retourne : une matrice sous format PyTorch

Détails du programme : Nous récupérons un array que nous convertissons en matrice sous PyTorch sous format float avec l'outil torch.

Fonction to_tensor_long(self, x)

Paramètres :

x : un array

Retourne : une matrice sous format PyTorch

Détails du programme : Nous récupérons un array que nous convertissons en matrice sous PyTorch sous format long avec l'outil torch.

Fonction dqn_training(self, num_episodes)

Paramètres :

num_episodes : nombre d'itération pour chaque entraînement

Retourne : Renvoie les récompenses et les prix associés sur tous les épisodes

Détails du programme : Nous copions pour la 1ere fois les paramètres de la policy_net dans la target_net. Ceci permet de prévenir des cas où on entraînerait la policy_net sans avoir eu le temps de mettre à jour les paramètres de la target_net, cela peut arriver quand le nombre d'épisode n'est pas un multiple de TARGET_UPDATE.

```
# The target_net load the parmeters of the policy_net
# state_dict() maps each layer to its parameter tensor
self.target_net.load_state_dict(self.policy_net.state_dict())
```

Cette ligne permet de prévenir que target_net n'est pas en mode entraînement, ici c'est bien policy_net qui s'entraîne et target_net qui copie le fruit des résultats. `self.target_net.eval()`

Nous passons les états au réseau et celui-ci nous retourne les q_values, avec lesquelles nous allons choisir une action.

```
# Select and perform an action
with torch.no_grad():
  q_values = self.policy_net(self.to_tensor(state))
action = self.policy.select_action(q_values.detach().numpy())
```

Nous mettons à jour le modèle, c'est la phase d'apprentissage.

```
# Perform one step of the optimization (on the target network)
   self.update_model(self.memory, self.policy_net, self.target_net)
```

Nous copions ensuite les paramètres dans la target_net si l'épisode en cours est un multiple de TARGET_UPDATE.

```
# Update the target network, copying all weights and biases in DQN
    if i_episode % self.TARGET_UPDATE == 0:
      self.target_net.load_state_dict(self.policy_net.state_dict())
      clear_output(wait = True)
      print(f'Episode      {i_episode}      of      {num_episodes}
({i_episode/num_episodes*100:.2f}%)')
```

--- Partie test ---

Fonction env_initial_test_state(self, price, booked,date)

Paramètres :

price : prix

booked : le nombre de réservation

date: période

Retourne : un état
Détails du programme : Initialisation des états qui prennent en compte 3 features : prix, nombre_de_réservation , date

Fonction profit_t_d_test(p_t,demand)

Paramètres :
p_t : prix à un instant t
demand : demande
Retourne : profit généré par les réservations
Détails du programme : Récupérer les profits multiplier à la demande pour en soustraire les coûts totaux à la gestion d'un appartement

Fonction dqn_test(self, price_grid)

Paramètres :
price_grid : données (prix, date, nombre de réservation)
Retourne : récompenses de la partie test, les prix associés aux données du test et le nombres de réservation en fonction des données et action
Détails du programme : Réalise le même programme que le dqn_training mais cette fois ci sans de boucle sur les épisodes, en effet en test le dqn n'a pas besoin de réaliser la même chose plusieurs fois, sachant qu'il est optimisé, il ne vas plus prendre d'action aléatoire. On réalise donc le calcul une fois avec le dqn déjà optimal.

Fonction cumul_reward(self, seq_reward_all_apt, data_test, data_test_booked)

Paramètres :
seq_reward_all_apt : séquence de récompense pour tous les appartements
data_test : prix associés à chaque appartement
data_test_booked : nombre de réservation par appartement
Retourne : Renvoie le cumule des récompenses grâce au dqn et les récompenses cumulées à partir des données initiales.

--- Partie interaction ---

Fonction env_test_step(self, state, action)

Paramètres :
state : état
action : action prise en fonction de l'observation
Retourne : l'état suivant issu de l'action prise en fonction de l'état actuelle
Détails du programme : C'est le même programme que la fonction env_step() mais qui initialise la demande à 'False' car elle sera récupéré du client de notre modélisation plus tard, une fois qu'il aura pris connaissance du prix fourni par la dqn

Fonction dqn_interaction(self, initial_state)

Paramètres :

initial_state : état initiale

Retourne : prix et un état

Détails du programme : Réalise le même programme que dqn_test mais la fonction env_step() n'est pas la même, on reprend à la place env_test_step().

--- Partie plot train ---

Fonction plot_return_trace(self, returns, labelx, labely, smoothing_window=10, range_std=2)

Paramètres :

returns : récompenses

labelx : légende en abscisse

labely : légende en ordonnées

smoothing_window : fenêtre de lissage

range_std : fourchette de l'écart type

Détails du programme : Affiche un graphique représentant la moyenne de récompense ainsi qu'une fourchette à + ou - range_std * l'écart type.

Fonction plot_price_schedules(self, p_trace, sampling_ratio, last_highlights,T)

Paramètres :

p_trace : liste de liste de prix, représente la liste de prix généré pour chaque épisode

sampling_ratio : le pas ou le ratio sur laquelle on prend un échantillon des sequence de prix. En effet, ceci permet de ne pas se retrouver avec un graphique trop désordonné.

last_highlights : La séquence de prix que nous souhaitons lui mettre une couleur particulière

T : la période

Détails du programme : Affiche sur un graphique plusieurs courbes des séquences de prix.

Fonction plot_result(self, return_trace, p_trace)

Paramètres :

return_trace : récompenses

p_trace : prix

Détails du programme : Affiche la récompense moyenne pour chaque instant de T

--- Partie plot test ---

Fonction plot_price(self, seq_price_all_apt, data_test):

Paramètres :

seq_price : séquence de prix provenant du dqn

data_test : prix provenant des données initiales

Détails du programme : Plot les deux séquences de prix et permet de comparer lequel est le mieux .

Fonction plot_reward(self, cumul_reward_from_algo, cumul_reward_from_data)

Paramètres :

cumul_reward_from_algo : récompenses cumulés provenant des prix du DQN

cumul_reward_from_data : récompenses cumulés provenant des prix des données initiales

Détails du programme : Permet de comparer les sommes des récompenses des appartements entre les prix générés par le DQN et les prix initiaux.

Fonction plot_result_test(self, price_grid_test)

Paramètres :

price_grid_test: set de données (prix, nombre de réservation, date) provenant du test

Retourne : Renvoie la séquence de récompenses, séquences de nombre de réservation et récompenses issue des données initiales.

2 - Clients

Naiv_Clients.py :

1. classe Naiv_client

Fonction __init__(self, prix_min,prix_max,will_to_pay, echeance)

Paramètres:

`self :` L'objet Naiv Client
`prix_min :` Le prix minimum
`prix_max :` Le prix maximum
`will_to_pay :` La proba d'achat
`echeance :` L'échéance de voyage

Détails du programme :
Initialise l'objet de liste client naïf qui enregistre tous les clients naïf utilisé dans la modélisation marché.

1. classe list_naiv_client

Fonction __init__(self,prix_min,prix_max,nb_client,wtp,rate_to_assure)

Paramètres:

`self :` L'objet List Naiv Client
`prix_min :` Le prix minimum
`prix_max :` Le prix maximum
`nb_client :` Nombre de client dans la liste
`wtp :` La proba d'achat
`rate_to_assure :` taux de client qui doivent avoir un wtp élevé

Détails du programme :
Initialise l'objet client naïf.

Fonction del_client(self, list_i)

Paramètres:

`self :` L'objet List Naiv Client
`list_i:` Liste d'index de suppression

Détails du programme :
On inverse l'index et on parcourt la liste de clients pour les supprimer.

Fonction check_sales(self, price, list_resa, list_resa_scnd)

Paramètres:

`self :` L'objet List Naiv Client
`price :` Le prix envoyé par le vendeur
`list_de_resa:` Liste de réservation
`list_de_resa_scnd:` Liste de réservation spécifique au client naif

Détails du programme :
Fonction qui permet de vérifier quels clients achètent avec le prix transmis. L'index de ceux-ci est ensuite enregistré pour être supprimé via la fonction update_client en appelant la fonction del_client.

Fonction update_client(self,prix_min,prix_max,list_to_del,wtp,rate_to_assure,max_time,resting_time)

Paramètres:

`self :` L'objet List Naiv Client
`prix_min :` Le prix minimum
`prix_max :` Le prix maximum
`list_to_del :` Liste d'index à supprimer
`wtp :` wtp fixé à assurer
`rate_to_assure :` taux de client à wtp fixé à assurer
`max_time :` temps total
`resting_time :` temps restant

Détails du programme :
Fonction qui englobe l'update une fois que les vérification d'achat ont été effectué et se charge de supprimer et de remplacer les clients par de nouveau.

Strategic_Client.py :

Fonction wtp_actualisation(self, price, max_echeance,current_client) :

Paramètres:

`self :` L'objet Strategic Client
`price :` Le prix du produit
`max_echeance :` L'échéance maximale qu'il reste pour le mois à revenir
`current_client :` Le client actuel dont le WTP est à actualiser

Retourne:

`wtp :` Le WTP actualisé du client en fonction du prix du produit et de l'échéance

Détails du programme :

Fonction d'actualisation du Paramètres WTP (Willingness To Pay) du client en fonction du prix de l'objet à considérer et de l'échéance personnelle du client. Prend en Paramètres le client en question, le prix de l'objet, et la valeur

maximale de l'échéance. Le WTP est compris entre 0 et 1, et est composé intérieurement de 2 Paramètres de calcul : le prix de l'objet et l'échéance du client. Ainsi, chacune de ces deux variables (nommés wtp_echeance et wtp_price) peuvent atteindre une valeur maximale de 0.5, et donc leur somme maximale donne 1 au total. Il y a également des Paramètres de poids_price et poids_echeance qui sont les poids respectifs de ces deux Paramètres. Dans un premier temps, ces deux poids sont initialisés à 1, et dans le futur s'il y a besoin d'influencer un Paramètres plus que l'autre, le changement de valeur de ces poids est possible, mais une seule règle primordial est à respecter : la somme de ces 2 poids doivent être égales au nombre 2 (nombre de Paramètres engagés dans le calcul). Nous pouvons par exemple mettre les couples de valeurs de poids comme tels : (1, 1), (0.4, 1.6), (1.9, 0.1) etc.. Si cette condition n'est pas respectée, un petit message d'alerte est affiché pour vous prévenir que vos calculs peuvent être faussés ou incorrects.

Trois autres Paramètres sont également initialisés :

- range_price : qui est la différence entre le prix maximal et le prix minimum que le client est prêt à débourser
- pas_price : qui est le pas d'augmentation du Paramètres wtp_price, ainsi il est calculé par la formule (max_wtp_price / range_price), ce qui donne : (0.5 / range_price) arrondi au millième.
- pas_echeance : qui est le pas d'augmentation du Paramètres wtp_echeance, ainsi il est calculé par la formule (max_wtp_echeance / max_echeance), ce qui donne : (0.5 / max_echeance) arrondi au millième.

De là commencent les choses sérieuses, les fameux calculs. On rentre dans la première boucle for qui permet de déterminer le wtp_price. On parcourt la boucle for i in range(range_price), et si le prix de l'objet passé en Paramètres est égal au prix max du client - i, le wtp_price se voit attribuer la valeur du pas_price multiplié par i. Et ainsi de suite, plus le prix de l'objet sera petit, plus la valeur du wtp_price augmentera. Ce qui est logique, plus le prix d'un produit est bas, plus le client aura envie de l'acheter. Nous avons également déterminé la valeur limite du wtp_price, si le prix du produit est inférieur ou égal au prix minimum donné par le client, le wtp_price prend sa valeur maximum, c'est à dire 0.5.

Deuxième boucle, parcourant for i in range(max_echeance), si l'échéance donné par le client est égal à (max_echeance - i), le wtp_echeance se voit attribuer la valeur de pas_echeance multiplié par i. Et ainsi de suite, plus l'échéance du client sera petit, plus la valeur du wtp_echeance augmentera. Ce qui est logique, plus l'échéance du désir d'acheter du client sera petite, plus il aura envie d'acheter un produit. Nous avons également déterminé la valeur limite du wtp_echeance, si l'échéance du client est à 0 (il ne peut plus attendre pour acheter), le wtp_echeance se verra attribuer sa valeur maximale, c'est à dire 0.5. Ainsi, les Paramètres wtp_price et wtp_echeance étant calculés, nous pouvons calculer le wtp du client, qui n'est rien d'autre que la somme de ces deux derniers, multiplié par leur poids respectif. Et ainsi nous obtenons une valeur entre 0 et 1, qui représente le WTP du client concerné. La fonction retourne ainsi le WTP trouvé actualisé.

Fonction strategic_price(self, price_trace, current_client)

Paramètres:

`self :` L'objet client

`price_trace :` L'historique des prix rencontrés au fur et à mesure du produit

`current_client :` Le client actuel

Retourne:

`buy :` Un booléen True / False

Détails du programme :

Fonction de détermination si le client stratégique achète ou pas le produit, en fonction des prix qu'il rencontre au fur et à mesure au moment présent sans connaissance des prix futurs. Si le dernier prix (le prix que le client voit actuellement) est inférieur ou égal aux deux d'avant, il achète (retourne le booléen True), sinon il n'achète pas (retourne le booléen False).

Fonction check_sales(self, price, echeance, price_trace, list_resa)

Paramètres:

`self :` L'objet client

`price :` Le prix du produit

`max_echeance :` L'échéance maximale qu'il reste

`price_trace :` L'historique des prix rencontrés par le client

`list_resa :` Liste des réservations

Retourne:

`list_sales :` Liste des ventes

`buy_considered :` Nombre d'achats considérés

`buy_done :` Nombre d'achats réalisés

`buy_dropped :` Nombre d'achats abandonnés

`instant_buy :` Nombre d'achats instantanés

`buy_postponed :` Nombre d'achats repoussés

`list_resa :` Liste de réservations

Détails du programme :

Initialisation des variables de conditions d'achat considéré, d'achat réalisé, d'achat abandonné, d'achat instantané, d'achat repoussé, pour représenter les achats. Pour chaque client de la liste de clients, nous allons faire des tests et déterminer si le client stratégique achète le produit ou non. Pour qu'il achète, il faut que le prix du produit soit bien compris dans la tranche de prix du client, et que le prix soit au minimum plus petit ou égal aux deux prix rencontrés précédemment (cette partie est expliquée dans la fonction "strategic_price"). Néanmoins, son choix est très accentué sur le Paramètres de l'échéance, nous avons fait nos calculs et donné à l'échéance un poids de deux, et de 0.5 pour celui du prix et de l'aléatoire, pour que le client fasse son choix d'achat, il accordera ainsi bien plus d'importance à l'échéance. Si le prix du produit est inférieur au prix minimum que le client était prêt à débourser, il achète directement le produit. Également pareil si l'échéance du client est à 0 (c'est à dire qu'il n'a plus le temps d'attendre pour acheter), l'achat sera effectué. Dans les

cas contraires aux exemples ci-dessus, les achats sont soient abandonnés ou repoussés. Tous ces paramètres font de lui, un client stratégique, par rapport à un client naïf.

Détail précis du code :

Initialisation des variables de conditions d'achat considéré / d'achat réalisé / d'achat instantané / d'achat repoussé :

```
buy_considered = buy_done = buy_dropped = instant_buy = buy_postponed = 0
list_sales = []
```

Nous effectuons une boucle pour chaque client de notre liste de client :

```
for current_client in range(len(self.clients_list)):
```

Si le prix du produit est bien comprise dans la tranche de prix du client, la condition est validée :

```
if (price >= self.clients_list[current_client].prix_min and price <= self.clients_list[current_client].prix_max):
```

Et ainsi l'achat est considéré par l'incrémentation de la variable correspondante :

```
buy_considered += 1
```

Initialisation d'une variable aléatoire entre 0 et ⅓

```
actual_willingness = rnd.random()/3
```

Initialisation des poids des différentes variables, avec un poids total à 3 : nous avons mis le poids de l'échéance à 2 et le poids du prix et du le poids de l'aléatoire à 0.5, car le choix final du client sera bien plus influencé par le Paramètres de l'échéance pour les clients stratégiques :

```
poids_price, poids_echeance, poids_rnd = 0.5, 2, 0.5
```

Nous effectuons une boucle pour chaque prix dans la tranche de prix du client :

```
for i in range(range_price):
```

Si le prix du produit correspond au prix de la boucle, la condition est validée :

```
if(price == int (self.clients_list[current_client].prix_max - i)):
```

Le Paramètres de prix prend la valeur du pas multiplié par le tour de boucle :

```
wtp_price = pas_price * i
```

Si le prix du produit est inférieur au prix du client: le Paramètres du prix prend sa valeur max, c'est à dire ⅓ :

```
if(price <= self.clients_list[current_client].prix_min):
      wtp_price = 1/3
```

Nous effectuons une boucle pour chaque échéance :

```
for i in range(max_echeance):
```

Si échéance du client correspond à l'échéance de la boucle, la condtiion est validée :
```
if(self.clients_list[current_client].echeance == (max_echeance - i)):
```

Le Paramètres de l'échéance prend la valeur du pas multiplé par le tour de boucle:
```
wtp_echeance = pas_echeance * i
```

Sii l'échéance du client est égale à 0 : le Paramètres de l'échéance prend sa valeur maximale, c'est à dire ⅓ :
```
if(self.clients_list[current_client].echeance == 0):
      wtp_echeance = 1/3
```

Nous effectuons la somme total des Paramètres rentrant en compte dans le calcul du Willingness To Pay final (WTP), chacun respectivement multiplié par leur poids respectif :
```
wtp = wtp_price * poids_price + wtp_echeance * poids_echeance + actual_willingness * poids_rnd
```

Si la valeur la différence de 1 par le wtp trouvé est inférieure ou égale au wtp initial du client, la condition est validée :
```
if (1-wtp) <= self.clients_list[current_client].will_to_pay:
```

Nous actualisons une condition de prix, qui sera expliqué dans la fonction "strategic_price", mais le résultat retourné est "True" si la condition est remplie, et "False" dans le cas contraire :
```
condition_prix = self.strategic_price(price_trace, current_client)
```

Ainsi, si la condition de prix stratégique est respecté, l'achat est réalisé par l'incrémentation de la variable d'achat réalisé :
```
if( condition_prix == True):
      buy_done += 1
```

Sinon dans le cas contraire, achat abandonné:
```
else:
      buy_dropped += 1
```

Si au départ le prix est déjà inférieur au prix minimum du client, et que l'échéance est inférieure à 30, la condition est validée :
```
elif (price <= self.clients_list[current_client].prix_min) and list_resa[self.clients_list[current_client].echeance-1] < 30:
```

Et l'achat instantané est fait par l'incrémentation de la variable d'achat instantané :
```
instant_buy += 1
```

Ou sinon, l'achat est repoussé :

```
else:
    buy_postponed += 1
```

Fonction get_min_x_percent(self, x):

Paramètres:

`self :` L'objet client

`x :` Pourcentage de lignes à prendre pour calculer le minimum

Retourne:

`self.df['price'].head(int(x * row)).mean() :` Retourne la moyenne des prix les plus bas pour le pourcentage x

Détails du programme :

Avec un dataframe dans l'ordre croissant des prix, cette fonction va calculer la moyenne des prix les bas (en prenant donc les prix en haut de la dataframe), et il en prendra un pourcentage x passé en Paramètres pour calculer la moyenne qu'il retournera.

Fonction get_max_x_percent(self, x)

Paramètres:

`self :` L'objet client

`x :` Pourcentage de lignes à prendre pour calculer le maximum

Retourne:

`return self.df['price'].tail(int(x * row)).mean() :` Retourne la moyenne des prix les plus élevés pour le pourcentage x

Détails du programme :

Avec un dataframe dans l'ordre croissant des prix, cette fonction va calculer la moyenne des prix les élevés (en prenant donc les prix en bas de la dataframe), et il en prendra un pourcentage x passé en Paramètres pour calculer la moyenne qu'il retournera.

Fonction update_min_max(self, price, max_echeance)

Paramètres:

`self :` L'objet client

`price :` Le prix du produit

`max_echeance :` L'échéance maximale qu'il reste

Détails du programme :

Permet d'actualiser pour chaque client les Paramètres de prix maximum et minimum pour la tranche de prix du client, et également son Paramètres WTP (Willingness To Pay)

Fonction update_client(self,prix_min,prix_max,list_to_del,resting_time):

Paramètres:

`self :` L'objet client

`prix_min :` Le prix minimum

`prix_max :` Le prix maximale

`list_to_del :` Liste des clients à retirer du marché

`resting_time :` Augmente au fur et à mesure du temps, représente le temps passé dans le marché

Détails du programme : Permet de

3 - Marché

1. classe Market()

Fonction __init__(self, prix_min, prix_max, nb_clients, taux_naiv, wtp_to_assure, rate_to_assure, df_price, df_mean)

Paramètres:

`self :` L'objet Market
`prix_min :` Le prix minimum
`prix_max:` Le prix maximum
`nb_client:` Le nombre de client qui seront sur le marché
`taux_naiv :` Le taux de clients naïfs
`wtp_to_assure :` Le wtp a assurer (pour les clients naïfs)
`rate_to_assure :` Le taux de clients naïfs avec des wtp a assurer
`df_price :` dataframe des prix
`df_mean :` dataframe des moyennes

Détails du programme :
Initialisation du marché et de tous ses attributs.

Fonction check_sales_v2(self, price, echeance, price_trace)

Paramètres:

`self :` L'objet Market
`price:` Le prix fixé par le vendeur
`echeance :` L'échéance maximale initiale
`price_trade :` l'historique des prix précédents

Détails du programme :
Fonction qui englobe la vérification des ventes et l'enregistrement de l'exécution de celles-ci.

Fonction updates(self,price,p_trace,ite)

Paramètres:

`self :` L'objet Market
`price:` Le prix fixé par le vendeur
`p_trace :` Historique des prix pratiqué par le vendeur
`ite :` mois de parcours en cours d'exécution

Détails du programme :
Fonction qui exécute la vérification des ventes et l'update des clients.

4 - Main

Fonction test (nb_episode, learn_rate,nb_part,batch_size)

Paramètres:

`nb_episode: nombre de répétition qui réalise un entraînement`
`learn_rate : taux d'apprentissage`
`nb_part: nombre de partition`
`batch_size: taille de l'échantillon`

Retourne:

Détails du programme :
Récupération des données via Excel, puis filtrage des données pour obtenir ceux qui sont utiles. appel du DQN pour chaque partition afin de l'entraîner, après cela nous créons un marché qui va interagir avec les clients. Nous initialisons les états . Il a des achats qui sont faits par des clients naïfs avec un prix aléatoire et des clients stratégiques , ce qui permet d'actualiser le marché. Nous faisons le même processus pour tous les clients (naïfs et stratégiques) avec des prix compris entre 130 et 170. Les résultats sont comparés sur un graphique afin de voir quelle méthode est la plus efficace.

Fonction revenue_plot (df)

Paramètres:

`df : l'ensemble des données`

Retourne: graphique des revenues générés

Détails du programme : Superpose sur un graphique 6 courbes: 3 courbes en pointillé représentant les chiffre d'affaires (CA) total, CA généré par les clients stratégiques et CA généré par les clients naïfs avec un prix aléatoire. Les 3 autres courbes en traits pleins représentent le CA total généré, le CA généré par les clients stratégiques et le CA générés par les clients naïfs cette fois-ci avec un prix proposé par le DQN.

Fonction plot_ptrace (list_toplot)

Paramètres:

`list_toplot: list de prix à générer sur le graphique`

Retourne:

Détails du programme : Parcours la liste des listes à ploter, ce graphique représente un prix pour chaque mois (sur 12 mois).

Fonction plot_diff (df)

Paramètres:

`df: données`

Retourne:

Détails du programme : Plot la différence du CA généré par le DQN et les prix aléatoires en fonction de la part de clients stratégiques.

Fonction moyenne_plot ()

Paramètres:

Retourne: moyenne des différence entre les chiffres d'affaires

Détails du programme : Appel le programme précédent pour ploter la différence de CA généré par le DQN et les prix aléatoires en fonction de la part de clients stratégique et récupère la moyenne des différences de chiffres d'affaires.