DataChallenge

Can you predict the tide ? par INRIA

https://challengedata.ens.fr/participants/challenges/67/

Documents et jupyterNB au Github : https://github.com/kjousselin/M2_DataChallenge

Voici la démarche que nous avons adopté pour répondre au problème :

Table des matières

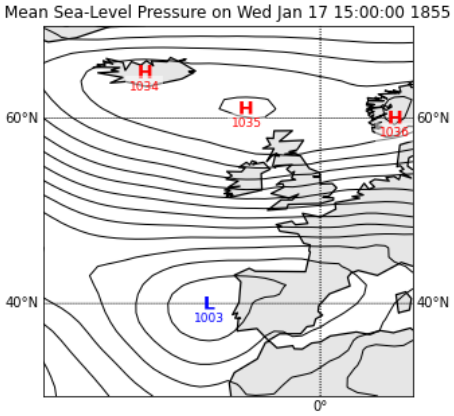
Partie I : Traiter les données

a) Importation et observation des données

Ouverture du fichier X_train_surge_new.npz

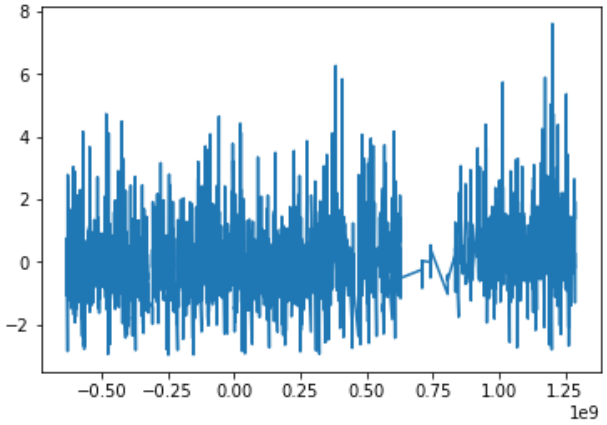
Le fichier contient 5599 observations, chaque observation comprend :

- id_**sequence** : Un numéro (index)
- 40 images **slp** (« Sea-Level Pressure fields **»**) 41x41 représentant la pression atmosphérique dans l'europe de l'ouest sur les 40 dates précédentes :

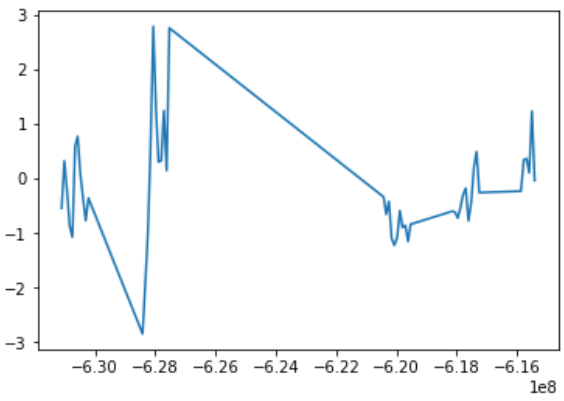


- **t_slp** représentant les dates des images précédentes,
- **surge1_input** et **surge2_input** : hauteur des marées sur les 10 dernières périodes (une période dure 12h) sur deux villes différentes (1 et 2),
- **t_surge1_input** et **t_surge2_input** : dates correspondantes,
- **t_surge1_output** et **t_surge2_output** : dates des marées à prévoir.

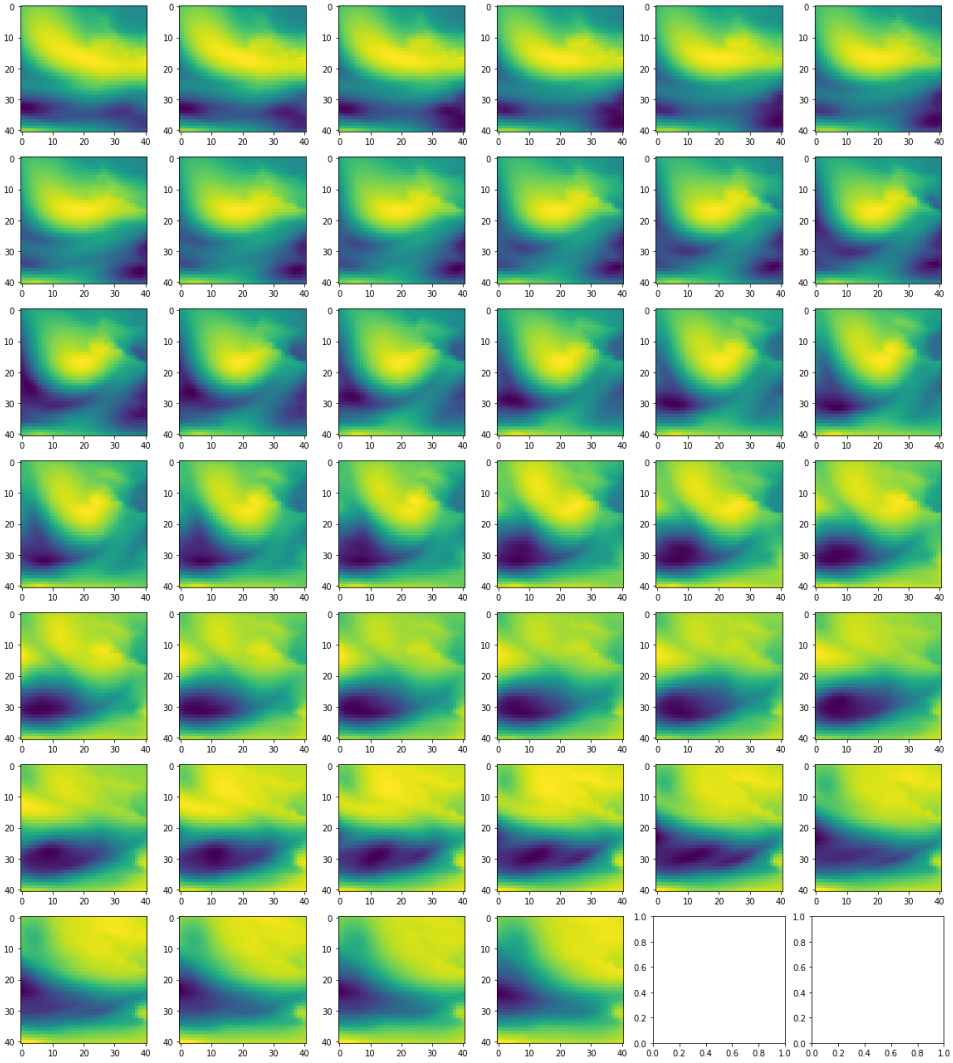
Voici ci-contre l'évolution de **surge1_input** (1ère période sur les 10) en fonction du temps (**t_surge1_input**) pour les 5599 observations :



Même chose pour les 50 premières dates (50 premières observations) :



Affichons les 40 images de pression atmosphérique (slp) liées à la première observation :



b) Traiter les données et séparer en deux jeux : un jeu d'entrainement et un jeu de validation, et transformation en un dataframe

Le jeu de données est important et n'est pas dans un dataframe.

On scinde les données originales en un set d'entrainement **df_train** et un set de validation **df_valid**, afin de faire mes propres tests.

60 % des données serviront à l'entrainement (taux = 0.6).

Ensuite, je construis un df en ne conservant que les données qui me paraissent intéressantes. Les étapes sont :

- Création d'un df vide
- **Ajout de la première valeur (uniquement) des t_slp** (colonne 't_slp_begin'). Garder les valeurs de t_slp suivantes est inutile car elles sont toutes corrélées !
- **Conservation de seulement 11 images** sur les 40 : en effet, d'une période à l'autre (3h) la pression atmosphérique varie peu (observation via plt.imshow())
  De plus, **les images sont moyennisées selon un pas de 4 pixels** : un carré de 16 pixels est fusionné en 1 valeur. On divise ainsi la taille d'une image (et donc le nombre de variables d'entrée) par 16 !
  Ces images sont mises à plat et les 11x10x10 colonnes « slp{k}_x_{i*pas}_y_{j*pas} » sont ajoutés au df.
- Ajout des deux colonnes « t_surge1 » et « t_surge2 » contenant uniquement la première date.
- Ajout des 2x10 « surge_input_i_j » contenant les 10 hauteurs de marées passées pour chaque ville.
- Ajout des deux colonnes « t_surge1_output » et « t_surge2_output » contenant les dates à prédire.

Ce travail est effectué pour chacun des deux sets, on obtient ainsi deux df **df_train** et **df_valid**.

Voici un aperçu des données :

```
df_train.head(2)
```
✓ 0.2s

	t_slp_begin	slp0_x_0_y_0	slp0_x_0_y_4	slp0_x_0_y_8	slp0_x_0_y_12	slp0_x_0_y_16	slp0_x_0_y_20	slp0_x_0_y_24	slp0_x_0_y_28	slp0_x_0_y_32	...
0	-631076416.0	1.025537	1.026974	1.026185	1.023688	1.020092	1.017873	1.017138	1.015019	1.013081	...
1	-630979200.0	1.019869	1.021717	1.021607	1.020457	1.019034	1.018357	1.019126	1.017700	1.016491	...

2 rows × 1125 columns

...

...	surge2_input_2	surge2_input_3	surge2_input_4	surge2_input_5	surge2_input_6	surge2_input_7	surge2_input_8	surge2_input_9	t_surge1_output	t_surge2_output
...	-1.202260	-1.193878	-1.143584	-0.816675	-0.757999	-0.129327	-0.179620	-0.372413	-630662400.0	-630658816.0
...	-1.143584	-0.816675	-0.757999	-0.129327	-0.179620	-0.372413	-0.053886	0.356847	-630576000.0	-630568832.0

Ces données sont ensuite à nouveau scindées en 2 : pour chaque ville. En effet : selon la localisation sur la carte, les pixels (pression atmosphérique) prédisant une hauteur de marée ne seront pas les mêmes !

On obtient : X_train_1, X_train_2, X_valid_1, X_valid2.

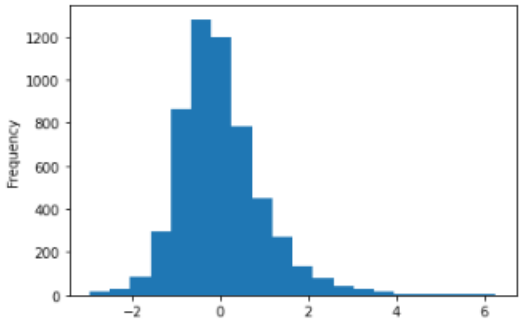
c) Extraction des données de sortie à prédire

On extrait « Y_train_surge.csv » et on obtient les 5599 lignes x 10 dates à prédire :

	id_sequence	surge1_t0	surge1_t1	surge1_t2	surge1_t3	surge1_t4	surge1_t5	surge1_t6	surge1_t7	surge1_t8	...	surge2_t0	surge2_t1	surge2_t2	surge2_t3	surge2_t4	surge2_t5	surge2_t6	surge2_t7	surge2_t8	surge2_t9
0	1	0.586936	1.069580	0.767928	-0.100162	0.070775	-0.244285	-0.354891	-0.928031	-0.773853	...	-0.053886	0.356847	0.348464	0.264641	0.901696	0.449052	0.113760	-0.422707	-0.456236	-0.825057
1	2	0.767928	-0.100162	0.070775	-0.244285	-0.354891	-0.928031	-0.773853	-0.375001	-0.361594	...	0.348464	0.264641	0.901696	0.449052	0.113760	-0.422707	-0.456236	-0.825057	-0.992703	-0.992703
2	3	0.070775	-0.244285	-0.354891	-0.928031	-0.773853	-0.375001	-0.361594	-0.210768	0.288635	...	0.901696	0.449052	0.113760	-0.422707	-0.456236	-0.825057	-0.992703	-0.992703	-0.322119	-0.883733
3	4	-0.354891	-0.928031	-0.773853	-0.375001	-0.361594	-0.210768	0.288635	-0.726929	-0.576103	...	0.113760	-0.422707	-0.456236	-0.825057	-0.992703	-0.992703	-0.322119	-0.883733	-0.473001	-0.422707

L'observation d'une colonne nous donne la répartition des valeurs suivante :

```
Y['surge1_t0'].plot(kind='hist', bins=20 )
plt.show()
```
✓ 2.9s



Ces données à prédire selon divisées en 2x10 colonnes « Y_train_{i}_{j} » et 2x10 colonnes « Y_valid_{i}_{j} ».

A l'aide du code suivant :

```
# Création des variables de sortie Y :
# 40 listes de la forme Y_{TYPE}_{i}_{j}
#       Y_train_1_0 à Y_train_2_9
# ainsi que :
#       Y_valid_1_0 à Y_valid_2_9
for TYPE in ['train', 'valid']:
    print("TYPE :")
    for i in [1,2]:                 # Parcours des lieux à prédir
        print("\tLieux", i, end=" : ")
        for j in range(10):         # Parcours des pas de temps à prédir
            print(j, end = " ; ")
            exec(f"Y_{TYPE}_{i}_{j} = list(Y['surge{i}_t{j}'][slice_{TYPE}])")
        print()
Y_train_1_1[:5]
```

Nous créerons ainsi un modèle pour chaque date à prédire, et chaque ville.

Partie II : Prévisions à l'aide de différentes méthodes

a) Méthode n°1 : K-plus proches voisins (KNN)

Pour commencer, j'ai voulu tester une méthode KNN.

```
from sklearn.neighbors import KNeighborsRegressor
KNR = KNeighborsRegressor(n_neighbors=10)
```

J'ai utilisé GridSearchCV afin de recherché les meilleurs paramètres pour la prédiction de X_train_1 :

```
# CV
parameters = {'n_neighbors':range(1,100,1)}
KNR = KNeighborsRegressor()
clf = GridSearchCV(KNR, parameters, scoring='neg_mean_absolute_error', cv=5, verbose =
clf.fit(X_train_1, Y_train_1_1)
```

J'ai utilisé le score de moyenne des valeurs absolue (MAE) afin d'avoir une idée de la bonne (ou mauvaise) représentation de la hauteur de vague. J'ai obtenu les résultats suivants :

```
print("meilleurs paramètres :", clf.best_params_)
print("meilleurs score (MAE) :", -clf.best_score_)
pred = clf.best_estimator_.predict(X_valid_1)
MAE_valid = np.sum(np.abs(pred - Y_valid_1_1))/n_valid
print("Score de validation (MAE) :", MAE_valid)
```

✓ 1.9s

```
meilleurs paramètres : {'n_neighbors': 68}
meilleurs score (MAE) : 0.7266264549607713
Score de validation (MAE) : 0.7027094938995766
```

Soit une distance moyenne en valeur absolu sur l'échantillon de validation, de 0,7. Ce qui est moyen au vu de la distribution des données (page précédente).

b) Méthode n°2 : Decision Tree

Avant de tester une forêt aléatoire, j'ai voulu tester un simple Arbre de Décision.

```
from sklearn.tree import DecisionTreeRegressor
model_tree = DecisionTreeRegressor(max_depth=5, criterion='absolute_error', splitter = 'random')
```

J'obtiens les résultats suivants sur le set de validation :

```
MAE : 0.5679757177214859
MSE : 0.6733362420064857
```

En utilisant GridSearchCv afin de choisir les meilleurs paramètres :

```
# CV
model_tree2 = DecisionTreeRegressor(criterion='absolute_error', splitter = 'random')
parameters = {'max_depth':range(2,10)}
clf = GridSearchCV(model_tree2, parameters, cv = 5, verbose = 4, scoring = 'neg_mean_squared_error')
clf.fit(X_train_1, Y_train_1_1)
```

Cette fois, j'ai choisi l'optimisation selon la MSE, en effet, le score final du DataChallenge sera construit sur une méthode similaire à la MSE (mais pondéré selon la date à prédire).

J'obtiens les résultats suivants :

```
Le meilleur paramètre est      {'max_depth': 4}
MAE : 0.5761382548441514
MSE : 0.693744253606538
```

Le R2 obtenu par clf.best_estimator_.score(X_valid_1, Y_valid_1_1) est de 0.33210.

Les MSE et MAE sont meilleurs, mais seuls 33% de la variance est expliqué par le modèle.

c) Méthode n°3 : Forêt aléatoire

J'ai ensuite testé une forêt aléatoire en effectuant une parallélisation.

```
from sklearn.ensemble import RandomForestRegressor
```

Après différentes tentatives pour choisir mes hyperparamètres, j'ai choisi les valeurs suivantes :

```
modele_rf = RandomForestRegressor(
     n_estimators=20,
     criterion='absolute_error',
     max_depth=3,
     min_samples_split=2,
     min_samples_leaf=1,
     n_jobs=4,
     verbose=4
     )
```

```
modele_rf.fit(X_train_1, Y_train_1_1)
```

Mes résultats sont les suivants :

```
MAE : 0.5129878260204572
MSE : 0.5444403354770355
```

Ces résultats m'ont paru intéressant, j'ai donc entrainé tous les modèles permettant d'obtenir les 20 colonnes à prédir « Y_valid_i_j », et j'ai testé la fonction « surge_prediction_metric » sur mes données de validations.

J'ai obtenu 0.568606895066, ce qui m'a incité à **soumettre ma première proposition.**

j'ai donc choisi pour faire ma première soumission. La suite a été traité en partie III

d) Méthode n°4 : Réseaux de neurones

Alors que le reste de mes travaux ont été fait sur le jupyter Notebook « **DataChallenge_etude.ipynb** », cette partie a été faite sur google colab (NoteBook « **DataChallenge_Cas_part_reseau_neurones.ipynb** » afin d'exploiter plus facilement PyTorch et les GPU de Google.

Après beaucoup de tests afin d'obtenir un réseau de neurones satisfaisant, j'ai créé une fonction permettant de tester différentes configurations, paramètres et hyperparamètres de mon réseau de neurones :

Voici la fonction permettant de créer mon réseau de neurones modulable :

```
from keras.models import Sequential
from keras.layers import Dense, Dropout, BatchNormalization, Activation
from keras.layers import Conv1D, MaxPooling1D, GlobalAveragePooling1D, Flatten

def un_model_sequentiel(first_relu = False, FONCT_ACT = 'relu', avec_conv = False, mon_batch = 32, df =
df_scale, y_true = Y_train_1_0, verbose = 0, end = False, sup = True):


    taille_entree = df.shape[1]

    model = Sequential()

    if avec_conv:
      # Add 1D convolutional layer with 32 filters, kernel size of 3 and 'relu' activation
      model.add(Conv1D(32, 3, input_shape=(taille_entree, 1))) #, activation='relu'))
      # Add a batch normalization layer
      model.add(BatchNormalization())
      # Add a max pooling layer
      model.add(MaxPooling1D(pool_size=2))
      # Add a dropout layer to reduce overfitting
      model.add(Dropout(0.5))

      model.add(Flatten())

      if first_relu:
          model.add(Dense(512, activation='relu'))
      else:
          model.add(Dense(512, activation=FONCT_ACT))

    else:
        if first_relu:
          model.add(Dense(512, input_shape=(taille_entree,), activation='relu'))
        else:
          model.add(Dense(512, input_shape=(taille_entree,), activation=FONCT_ACT))

    model.add(BatchNormalization())
    model.add(Dropout(0.5))

    if sup:     # Une couche de neurones supplémentaire
        model.add(Dense(512, activation=FONCT_ACT))
        model.add(BatchNormalization())
        model.add(Dropout(0.5))

    model.add(Dense(256, activation=FONCT_ACT))
    model.add(BatchNormalization())
    model.add(Dropout(0.5))

    model.add(Dense(128, activation=FONCT_ACT))
    model.add(BatchNormalization())
    model.add(Dropout(0.5))

    model.add(Dense(64, activation=FONCT_ACT))
    model.add(BatchNormalization())
    model.add(Dropout(0.5))

    if end:
        model.add(Dense(32, activation='relu'))
    else:
        model.add(Dense(32, activation=FONCT_ACT))

    model.add(BatchNormalization())
    model.add(Dropout(0.5))

    model.add(Dense(1))

    model.compile(optimizer='adam', loss='mean_squared_error')

    model.fit(df, y_true, epochs=20, batch_size = mon_batch, verbose = verbose)

    return(model)
```

Il est modulable dans le sens où un certain nombre de paramètres permettent de modifier sa configuration :

`first_relu :` Mettre en première fonction d'activation, la fonction 'relu'

`FONCT_ACT = 'relu' :` Choix de la fonction d'activation centrale

`avec_conv = False :` Commencer par une convolution

`mon_batch = 32 :` Taille du batch (nombre de données choisi aléatoireent pour entrainé notre modèle à chaque passage)

`end = False :` Finir par une fonction d'activation 'relu'

`sup = True :` Ajouter quelques couches de neuronnes

Une boucle me permet de tester toutes ces configurations et d'exporter les résultats. J'obtiens :

```
# AVEC deux nouvelles fonctionnalité + Scale loi normale
df_result_1_0_e = df_resultats.sort_values('MSE', ascending = True).copy()
df_result_1_0_e.head(10)
```

	fonc_act	first	Conv	end	sup	MAE	MSE	pourc_ok
60	softplus	False	True	False	False	0.329237	0.222531	28.9
93	elu	False	True	True	False	0.330602	0.224416	36.2
85	elu	True	True	True	False	0.327781	0.226558	33.0
58	softplus	False	False	False	True	0.338660	0.226590	28.2
87	elu	True	True	True	True	0.329850	0.229447	31.8
91	elu	False	False	True	True	0.331343	0.231181	34.9
95	elu	False	True	True	True	0.331530	0.232538	33.4
80	elu	True	False	False	False	0.337889	0.234531	35.6
81	elu	True	False	True	False	0.338343	0.235922	32.9
89	elu	False	False	True	False	0.331631	0.236000	31.8

J'ai donc choisi de tester les modèles suivants :

```
fonc_act, first, Conv, end, sup  = 'softplus', False, True, False, False
fonc_act, first, Conv, end, sup  = 'elu', False, True, True, False
fonc_act, first, Conv, end, sup  = 'elu', True, True, True, False
```

Par exemple pour le dernier, la configuration de mon réseau est la suivante :

- **Une couche de convolution** avec 32 filtres et un noyau de taille 3,
- Une couche de 512 neurones dont la fonction d'activation en sortie est une **ReLU** (« Rectified Linear Unit »),
- **3 couches de neurones** de respectivement 256, 128 et 64 neurones, avec des fonctions d'activation '**elu**' (« Exponential Linear Unit »),
- **1 couche de 32 neurones** dont la fonction d'activation est une ReLU,
- Entre chaque couche caché est appliqué une normalisation par lot (**Batchnormalisation**) afin d'augmenter la stabilité du modèle, ainsi qu'un **dropout** de 0.5 qui permet à chaque passage d' « annuler » 50 % des neurones et ainsi éviter le sur-apprentissage,
- **1 dernière couche de 1 neurone** sans fonction d'activation ($x \mapsto x$),
- La fonction de perte **Loss** est la **MSE** et **l'optimizer** 'Adam',
- Le modèle est ajusté avec 20 **epochs** (20 passages de l'ensemble des données).

J'ai exporté mes 4 résultats en csv afin de les soumettre au DataChallenge (voir partie IV)

Partie III : Application de nos « meilleurs modèles »

Après avoir obtenu des premiers résultats satisfaisants sur mes propres données de validation avec une **Random Forest** et avec un **Réseau de Neurones**, j'ai créé des scripts dans cette partie afin d'entrainer le modèle complet, récupérer et traiter et enfin prédire les données et exporter les résultats.

a) Importation de X_test et traitement

On importe le fichier « X_test_surge_new.npz » et on effectue le même traitement que sur X_train afin d'obtenir deux df "**X_test_1**" et "**X_test_2**" correspondant aux deux villes à prédir.

b) Concaténation des données d'entrainement et de validation

On concatène les données séparées en partie I, ayant servi à choisir le modèle afin d'avoir un dataset d'entrainement.

On concatène **df_train_1** avec **df_valid_1**, **df_train_2** avec **df_valid_2**, **Y_train_1_1** avec **Y_valid_1_1**, etc…

c) Construction du modèle Random Forest, entrainement, prédiction et export des résultats

Tout est dans le titre !

On obtient le df « `df_sortie_final.csv` » et on le soumet.

➔ **Ma première soumission auprès de DataChallenge a donné un score sur leurs données de test de 0,5985.**

d) Soumissions des prédictions obtenues via le réseau de neurones

Les prédictions obtenues par mes réseaux de neurones m'ont permis de faire 4 soumissions :

- 8693385243927780 : valeur aberrantes obtenues suite à une erreur de ma part !
- 0,572916734191184
- 0,565893029310509
- **0,554984184006569**

Ce qui me place aux rangs suivants :

73ème sur 137 sur le classement public
12ème sur 16 sur le classement académique privé

Conclusion

Le DataChallenge a été un bon moyen de mettre en application mes connaissances. Le format initial des données, leur complexité ainsi que leur grande dimension ont rendu les débuts difficiles mais en y consacrant du temps, j'ai pu les comprendre et mettre en application différentes modèles de prédiction. Mes 5 soumissions m'ont permis d'obtenir un résultat correct, mais je regrette que, malgré le temps passé, je n'ai pas pu obtenir un meilleur score.

Pour améliorer grandement ma démarche, il aurait fallu que je présente mes scripts sous la forme de fonctions ou de classe afin d'extraire les données, les tester, les concaténer, extraire les résultats (…) plus rapidement. Ce temps gagné m'aurait permis de tester davantage de modèles et de paramètres.

J'ai apprécié ce « challenge » et remercie mes camarades de promotion pour cette émulation, notamment Ann-Gaëlle, Léo, Sarah et Vahia.