Rapport Projet Machine Learning

Damien et Geri

Teams RAMP: minatto_sama et damsss

Le but du projet était d'améliorer la prédiction sur RAMP. La mission consiste à prédire au mieux le volume de passager pour les vols US en ajoutant des nouvelles données.

Le meilleur RMSE obtenu par notre équipe est le suivant :

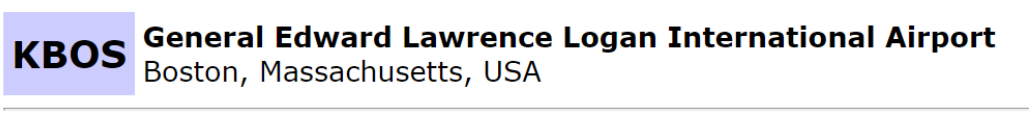
rank	team	submission	rmse	train time [s]	validation time [s]	submitted at (UTC)
7	damsss	(pass_0,302-><0,3)	0.299	3210.101359	170.642146	2020-03-13 10:17:54

Etape 1 : Collecte des données

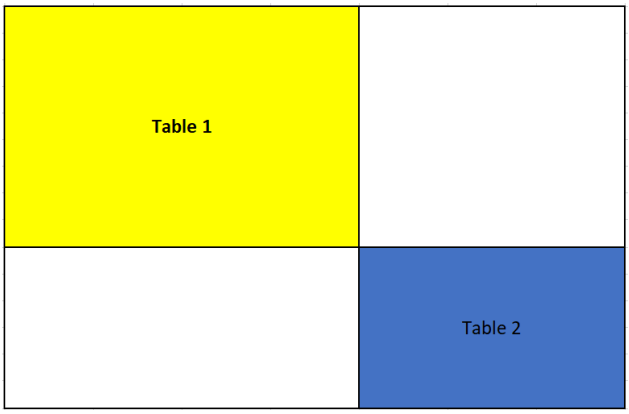
Contrainte : on pouvait ajouter que des données sur external_data.csv. Nous pensions initialement webscrapper sur le RAMP nos données, mais nous avons dû les rattacher directement au csv external_data. Voici nos données que nous avons ajouté :

- Variable « N_BD » binaire 0 ou 1 :
  Montre si la date de départ de l'avion est un jour férié aux USA ou un weekend. On suppose que pour ces jours-là, le volume va être supérieur à la normale. Nous avons créé une fonction qui donne 1 lorsque la date de départ est un de ces jours ou 0 sinon.
- Variable pour le prix du pétrole :
  Effectivement, on a commencé par prendre le prix du pétrole et on a trouvé une grande corrélation avec notre variable Y. Malheureusement ce n'était pas bon car, les avions utilisent du kérosène et les prix évoluent positivement entre les deux avec une corrélation de 70%.
  On a pris finalement le prix du kérosène à partir de la date départ et on l'a laggé : « offsetList = [1,3,6,12] » mois, puisque les compagnies aériennes achètent leur pétrole en avance.
  Malheureusement, dans tous les traitements que nous avons faits, elles n'ont jamais été significatives. Il y a en effet une trop grande corrélation avec d'autres variables tel que dateofdeparture.
  Le site par laquelle le prix du kérosène est pris est le suivant :
  https://fred.stlouisfed.org/series/MJFUELUSGULF

- On avait besoin de la ville de chaque aéroport pour trouver d'autres informations tel que la population. A partir du site suivant : https://www.airnav.com/airport/KBOS, on récupére l'information précise contenu dans le HTML en utilisant comme input le IATA des aéroport (ex : BOS). On s'est rendu compte en le faisant que dans certains cas, le nom de l'aéroport n'était rattaché à aucune ville. (Pour DFW, nous avons choisi de lui assigner Dallas)



- A partir de la ville, on récupère la population qui est un facteur important de notre analyse. Nous pensions que des villes de tailles importantes (en termes de populations) auraient plus de voyageurs que de petites villes. On les récupère via le dataset suivant : https://public.opendatasoft.com/api/records/1.0/search/?dataset=worldcitiespop&q=%s&sort=population&facet=country&refine.country=%s
- La partie la plus compliquée était de trouver les distances entre chaque aéroport et de les ajouter sur le csv. En effet, il est difficile d'ajouter une distance à un csv ne comprenant qu'une position géographique. (Dans le csv il n'y avait qu'une colonne du nom d'aéroport). Donc la solution qu'on a trouvée était de combiner deux CSV. Le premier étant l'external_data, et le second était de créer une liste de combinaisons d'aéroports de départ et d'arrivée à valeur unique. Dans notre cas on avait 20 aéroports uniques donc 400 combinaisons possibles. Grâce au site suivant : https://www.prokerala.com/travel/airports/distance/from-bos/to-ord/ on a réussi a récupéré la distance en Km entre chaque aéroport et à la stocker dans un DataFrame qui ne pouvait être fusionner avec l'external_data directement. Donc on a trouvé une astuce pour garder les deux tables dans le même csv mais a des endroits différents, plus précisément de cette manière suivante (les blancs seront des NaN en réalité) :



Etape 2 : Pre-processing et transformation

Apres importation de toutes les colonnes external_data.csv et en faisant des jointures sur Date, Airport d'Arrivée et de Départ (les deux à la fois distinctement), nous avons décidé de traiter certaines variables. Le choix de joindre nos informations de l'external_data aux aéroports d'arrivée et de départ nous semblait logique car les informations peuvent fortement différer entre les deux. Par exemple, une population élevée dans une ville de départ n'impliquera pas beaucoup de départ vers une ville faiblement peuplée. Parmi les variables où nous avions le minimum, le maximum, et la moyenne, nous avons préféré conserver la moyenne, le maximum et le minimum de la journée ne reflétant pas la tendance journalière.

- En commençant par les variables continues comme Température, Wind, Humidity mais aussi Population, on a choisi d'utiliser la différence entre les villes de départs et d'arrivées plutôt que de conserver nos valeurs pour nos deux aéroports.
- Transformation de DateofDeparture en variable continue
- On n'a pas conservé les variables catégorielles avec des données manquantes (Events : trop de valeurs uniques et trop de valeurs manquantes)

- On a décidé de transformer en variables binaires : Départ, Arriver, et notre date par jour, mois et année car on n'est pas sûr des séries temporelles et on aimerait bien capter les effets des mois.
- Au final on a choisi de scaler nos variables continues car les échelles n'était pas les mêmes et beaucoup de variations, plus précisément on a choisi le MinMax().

On a du coup récupérer une matrice de features de 72 colonnes séparées en variables continues (« scaler ») et dummies.

[133]:
```
print(X_encoded.shape)
X_encoded.head()
```

```
(8902, 72)
```

[133]:

	DateOfDeparture	WeeksToDeparture	std_wtd	Distance	Mean Humidity	Mean Sea Level PressurehPa	Mean TemperatureC	Mean VisibilityKm	Mean Wind SpeedKm/h	MeanDew PointC	N_BD	Oil_Poffset1	Oil_Poffset12	Oil_Poffset3
0	0.529946	0.499068	0.558489	0.245664	0.454545	0.540541	0.577465	0.50000	0.478723	0.464789	0	0.512111	0.437988	1.000000
1	0.680581	0.567756	0.533244	0.175950	0.660606	0.527027	0.647887	0.46875	0.553191	0.774648	0	0.826990	0.842151	0.498270
2	0.725953	0.401136	0.501799	0.268924	0.521212	0.608108	0.309859	0.37500	0.595745	0.253521	0	0.887543	0.808326	0.707612
3	0.068966	0.431146	0.425483	0.167788	0.515152	0.486486	0.549296	0.50000	0.734043	0.507042	1	0.467128	0.000000	0.467128
4	0.313975	0.429686	0.536924	0.310809	0.284848	0.283784	0.408451	0.62500	0.755319	0.267606	0	0.707612	0.842151	0.498270

Etape 3 : Feature Selection

Dans cette étape on a décidé de faire du Feature Sélection pour réduire la dimension de notre X_feature car on a ajouté beaucoup de variables qui peuvent ajouter beaucoup du bruit. L'algorithme qu'on a pensé à tourner est le Boruta, une dérivation du RandomForest. De base créé dans une librairie pour R, il a été récemment transféré sur Python aussi, il est plus rapide, grâce au langage C.

Mise en place :

Cet algorithme n'a pas besoin d'une spécification des variables importantes, le N optimal le trouve tout seul. Le principe reste similaire à un RandomForest mais à une difference, il mélange les données pour éliminer le bruit. Incovenient, il est très lent mais efficace et très facile à utiliser.

Voici le type de resultat :

[210]:

	column_name	feature_importance
0	DateOfDeparture	1
1	Distance	1
66	m_12	1
50	a_ORD	1
4	Mean TemperatureC	1
45	a_LAX	1
43	a_JFK	1
40	a_DTW	1
8	N_BD	1
30	d_ORD	1
25	d_LAX	1
23	d_JFK	1
13	Population	1
20	d_DTW	1

Donc le Boruta trouve les N premier feature optimal pour notre jeu de donnée. Dans ce cas là, il nous donne ces variables comme étant significatives (en fonction du X_train). Comme la base peut changer, nous avons jugé bon d'intégrer plus de critères, tel que l'ensemble des aéroports de départs et d'arrivées, il ne faut pas en effet se coincer avec que certains types de voyages, tous les aéroports sont importants à nos yeux.

Etape 4: Parameter Tuning and Model Selection

On a décidé de faire une rotation de modèle et trouver les modèles qui marchent bien avec notre jeu de données. Les modèles les plus performants sans Parameter Tuning était sont le XGBRegressor et le LGB. Donc on a choisi de garder les deux pour faire du tuning de Parameter.

- **LGBMRegressor** :

```
[71]: from sklearn.model_selection import RandomizedSearchCV
from xgboost import XGBRegressor
import lightgbm as lgbm
from sklearn.metrics import mean_squared_error
from sklearn.base import BaseEstimator
from sklearn.model_selection import RandomizedSearchCV

param_test ={'num_leaves': sp_randint(6, 50),
             'min_child_samples': sp_randint(100, 500),
             'min_child_weight': [1e-5, 1e-3, 1e-2, 1e-1, 1, 1e1, 1e2, 1e3, 1e4],
             'subsample': sp_uniform(loc=0.2, scale=0.8),
             'colsample_bytree': sp_uniform(loc=0.4, scale=0.6),
             'reg_alpha': [0, 1e-1, 1, 2, 5, 7, 10, 50, 100],
             'reg_lambda': [0, 1e-1, 1, 5, 10, 20, 50, 100]}

n_HP_points_to_test = 100
```

```
[*]: model = lgbm.LGBMRegressor(max_depth=-1, random_state=314, silent=True, metric='neg_root_mean_squared_error', n_jobs=4, n_estimators=5000)
searchLGB = RandomizedSearchCV(estimator = model, scoring = 'neg_root_mean_squared_error',
                               param_distributions = param_test,
                               n_iter=n_HP_points_to_test,
                               cv=3,
                               refit=True,
                               random_state=314,
                               verbose=True)
searchLGB.fit(X_train[important_features],y_train)
y_pred = searchLGB.predict(X_test[important_features])
rmse = mean_squared_error(y_test,y_pred)
rmse
```

```
Fitting 3 folds for each of 100 candidates, totalling 300 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 300 out of 300 | elapsed:  7.0min finished
```

```
[214]: 0.24306981153144241
```

```
[215]: searchLGB.best_params_
```

```
[215]: {'colsample_bytree': 0.55607546409401,
 'min_child_samples': 103,
 'min_child_weight': 10.0,
 'num_leaves': 22,
 'reg_alpha': 2,
 'reg_lambda': 1,
 'subsample': 0.8939112927620336}
```

En local, nous arrivions à descendre à des RMSE en-dessous de 0.24. Le probleme qu'on a eu était que on arrivait pas a avoir le meme score entre le local et le RAMP. Dans le RAMP, avec un copier coller du code, les paramètres sorties par best_params_ et le feature selection utilisé, on trouve généralement un RMSE deux fois supérieur. Le jeu de données dans le ramp différant de celui surlequel on s'est entrainé nous procure une grosse différence. Cela nous a énormement et longuement frustré.

- **XGBRegressor :**
  On a fait la meme chose pour le XGB : utilisation des variables sorties par le Boruta,utilisation du XGB avec RandomSearch pour capturer les paramètres optimisés. Le XGB arrive a donné un score de 0.16 en local vs 0.3 dans le RAMP.

```
[*]: param_XGBR = {'learning_rate' : [0.01, 0.025, 0.05, 0.075, 0.1, 0.15, 0.2],
                   'max_depth': [3, 5, 6, 7, 8, 9, 10],
                   'min_child_weight': [1, 3, 5, 7],
                   'gamma': [0 , 0.1, 0.2, 0.3, 0.4],
                   'colsample_bytree': [0.3, 0.4, 0.5, 0.7, 0.9],
                   'n_estimators':[10,25,50,100,200,300,400,500]}

modelXGB = XGBRegressor(random_state=42)

searchXGB = RandomizedSearchCV(estimator = modelXGB, scoring = 'neg_root_mean_squared_error',
                               param_distributions = param_XGBR,
                               cv = 5, n_iter=150, random_state = 42,verbose=0)
searchXGB.fit(X_train[important_features],y_train)
y_predXGB = searchXGB.predict(X_test[important_features])
rmseXGB = mean_squared_error(y_test,y_predXGB)
rmseXGB

[217]: 0.1623569581826509

[218]: searchXGB.best_params_

[218]: {'n_estimators': 300,
        'min_child_weight': 5,
        'max_depth': 10,
        'learning_rate': 0.2,
        'gamma': 0.1,
        'colsample_bytree': 0.4}
```

Etape final (RAMP) : Pour conclure, notre modele final est le suivant :

- **Feature important** : Séléction de nos variables que l'on conserve, Population et Mean Temperature sont bien la variation de ces variables entre l' aéroport de départ et d'arrivée.

```
        X_encoded[columns_to_scale] = scaler.fit_transform(X_encoded[columns_to_scal
e])
        important= ['DateOfDeparture','Distance',
                    'Mean TemperatureC',
                    'N_BD','Population','d_ATL','d_BOS',
                    'd_CLT','d_DEN','d_DFW','d_DTW','d_EWR','d_IAH','d_JFK','d_LAS',
                    'd_LAX','d_LGA','d_MCO','d_MIA','d_MSP','d_ORD','d_PHL','d_PHX','d
_SEA','d_SFO',
                    'a_ATL','a_BOS','a_CLT','a_DEN','a_DFW','a_DTW','a_EWR','a_IAH',
                    'a_JFK','a_LAS','a_LAX','a_LGA','a_MCO','a_MIA','a_MSP',
                    'a_ORD','a_PHL','a_PHX','a_SEA','a_SFO']
        X_encoded=X_encoded[important]
        X_array = X_encoded.values
        return X_array
```

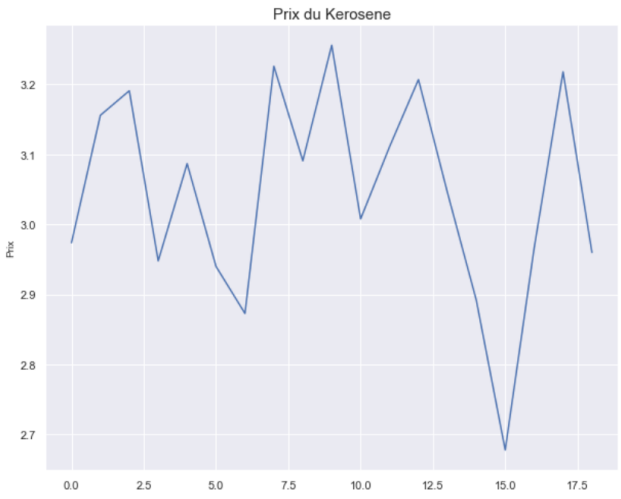
- **Regressor** : Vue que pour notre jeu de données, le XGB donne un bon résultat, on est resté sur lui.

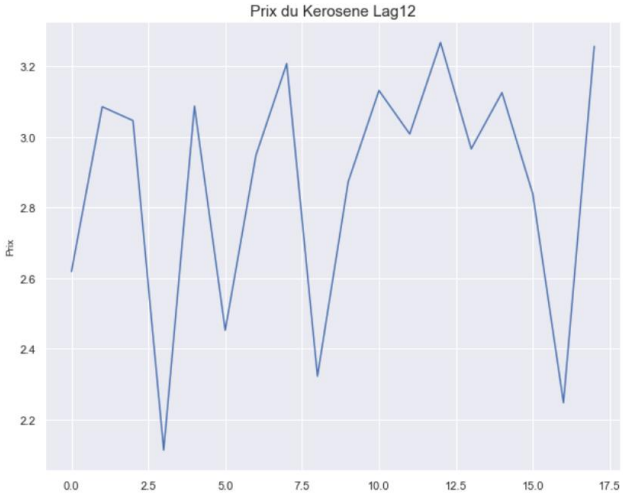
```
    class Regressor(BaseEstimator):
        def __init__(self):
            self.reg = XGBRegressor(n_estimators= 5000,min_child_weight= 5,max_depth= 30,l
earning_rate= 0.15,gamma= 0.2,colsample_bytree= 0.3)

        def fit(self, X, y):
            self.reg.fit(X, y)

        def predict(self, X):
            return self.reg.predict(X)
```

- **Score :** Bizzarement, on n'arrive pas a retrouver le meme score que notre local mais pour nous, ça reste encore un bon score. (Nous avons également obtenu 0.302, avec un train time inférieur à 700 en faisant varier les paramètres, et 0.298 avec un train time supérieur à 4500)

Analyse des données :

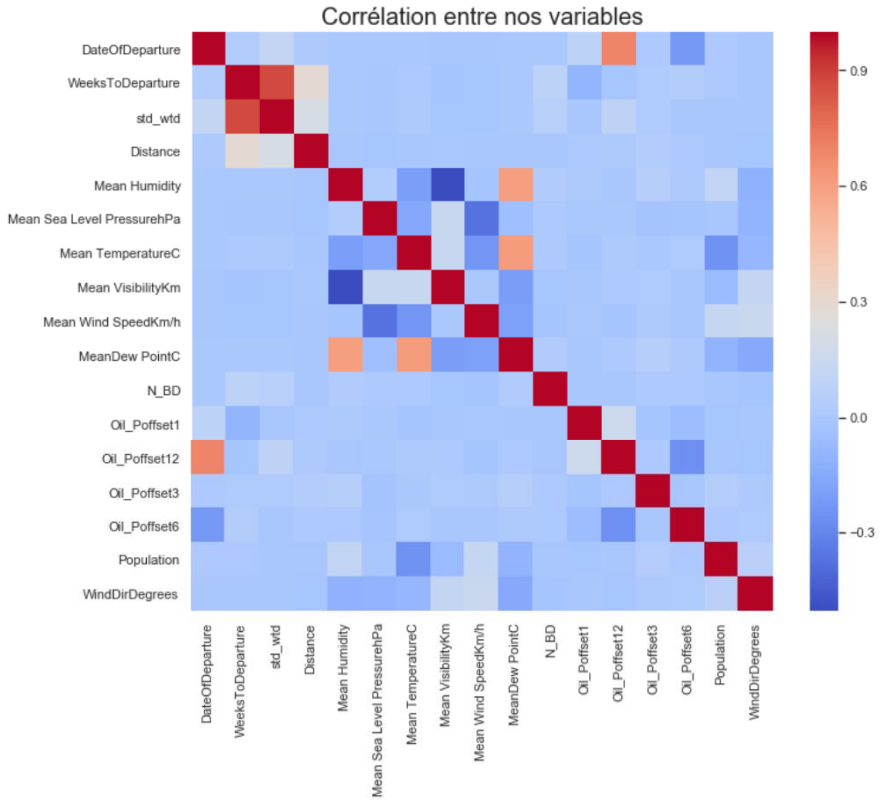
- On a une grande variations du prix de Kerosene pendant 2010 et 2013 :





- Correlation entre nos variables : Par exemple, WeekstoDeparture vs std_wtd

```
[60]: plt.figure(figsize=(11,9))
      cor = X_train.corr()
      sns.heatmap(cor, cmap='coolwarm')
      plt.title("Corrélation entre nos variables", fontsize= 20)
      plt.show()
```



Ci-dessous la matrice de corrélation entre nos variables. Nous avons choisi de retirer logiquement weekstodeparture, qui est semblable à datetodeparture (celle-ci étant toujours préférée dans nos premiers Boruta). Concernant, les variables weather de notre table initiale : Nous avons décidé initialement de conserver l'ensemble. Nous avons droppé Events, car trop de valeurs manquantes et trop de valeurs uniques. Pareil pour Précipitation qui avait des valeurs aberrantes. Pour les variables présentant un minimum, un maximum et une moyenne, nous ne conservons que la dernière. On suppose initialement que l'écart minimal entre ces variables entre l'aéroport de départ et d'arrivée est équivalent à l'écart maximal et moyen. Nous ne devons alors en garder qu'une. Ensuite, les variables Weather sont fortement corrélées entre-elles. Nous gardons Mean Temperature (qui est la différence entre la température de la ville de départ et d'arrivée) qui est toujours la variable météo la plus importante dans notre Boruta.

Ce graphique de corrélation peut être remplacé par un VIF qui sélectionnera les variables avec le moins de corrélation.

Conclusion

N'ayant pas le meilleur score RMSE (loin de là), notre approche n'est pas la meilleure. Mais nous avons cherché à appliquer une démarche pour trouver le meilleur modèle possible. Nous pensons que nous avons fait un erreur sur nos choix de variables et nos transformations, car même en appliquant le LGBM de l'équipe Paul (comme toutes les autres équipes ont fait) avec des paramètres ajustés pour notre modèle, nous n'avons jamais réussi à les atteindre.