

L2 MI - Mini Projet

Challenge **XPorter** - Groupe Scooter

Membres :

Fedy Ben Naceur, Hakim Kasdi,

Xuewen Guo, Kevin Tran,

Hamza Manita Jelidi, Hugo Penicand.

URL du challenge : [XPorter](#)

Repo Github du projet : [Notre Github](#)



Contexte et description du problème

Le challenge que nous avons choisie est le challenge XPorter. Notre mission est de prédire le nombre de voitures qui passeront à une date, une heure et des informations météorologiques supplémentaires.

Pour réaliser ce challenge, nous avons un dataset qui s'appelle "all-data", qui contient des informations indiquant la qualité de trafic entre différents jours avec des conditions météorologique, comme par exemple la neige, la pluie etc...

Nous avons séparé le travail à réaliser pour ce projet en 3 parties, chacune traitée par un binôme.

Les 3 parties sont :

- Le Pre-processing, c'est-à-dire la transformation de données brutes en données utilisables par l'étape suivante ;
- Le Model, qui sera chargé de trouver le plus performant model d'apprentissage mais aussi les hyper-paramètres correspondants pour notre proje ;
- La Visualisation, dont la mission est d'affichage des résultats de manière compréhensible et pertinente via une interface graphique claire.

Table des matières

1	<u>Team Pre-processing :</u>	
	Fedy Ben Naceur & Hakim Kasdi	3
1.1	Outliers Detection	3
1.1.1	Qu'est ce qu'une anomalie dans les données ?	3
1.1.2	Code	3
1.2	Réduction de dimension	3
1.2.1	T-SNE	3
1.2.2	PCA	4
1.3	Feature selection	5
1.3.1	Explication des methodes utilisées	5
2	<u>Team Model :</u>	
	Xuewen Guo & Kevin Tran	7
2.1	Les différents modèles testés	7
2.1.1	Cross-validation	8
2.2	RandomForest	8
2.3	Hyper-paramètre	9
2.4	La suite	9
3	<u>Team Visualisation :</u>	
	Hamza Manita Jelidi & Hugo Penicand	11
3.1	Scatterplot	12
3.2	Barplot (Hour,target)	12
3.3	Boxplot	13
3.4	Correlation Matrix	15

Chapitre 1

Team Pre-processing : Fedy Ben Naceur & Hakim Kasdi

Notre groupe était chargé de la tâche de "pre-processing". Notre rôle consiste à créer des méthodes qui nous aideront à avoir des meilleures performances. Nous avons procédé de la manière suivante :

- On utilise au début les méthodes fournies par sklearn pour détecter les anomalies dans les données et les éliminer ;
- On visualise les données à travers l'algorithme T-SNE (t-distributed stochastic neighbor embedding) ;
- On utilise PCA (Principal Component Regression) pour observer le nombre de "features" optimal qui représente le mieux nos données ;
- On sélectionne ensuite les features qui performant le mieux.

1.1 Outliers Detection

1.1.1 Qu'est ce qu'une anomalie dans les données ?

C'est une observation qui diffère du motif global de l'échantillon, qui peut être de deux types : univariées et multivariées. Outliers univariées peuvent être trouvées en examinant une distribution de valeurs dans un seul espace de caractéristiques. Les outliers multivariées peuvent être trouvées dans un espace à n dimensions (de n caractéristiques). Il peut être très difficile pour le cerveau humain d'examiner des distributions dans des espaces à n dimensions, c'est pourquoi nous devons former un modèle pour le faire à notre place.

1.1.2 Code

```
from sklearn.ensemble import IsolationForest
import numpy as np
clf = IsolationForest(behaviour = 'new',
max_samples='auto', random_state = 1,
contamination= 'auto')
preds = clf.fit_predict(data)
preds = np.where( preds<1)
for i in preds:
    data = data.drop(i, axis=0)
```

1.2 Réduction de dimension

1.2.1 T-SNE

Explication de l'algorithme

T-distributed stochastic neighbor embedding est une technique qui permet d'analyser des données décrites dans des espaces à forte dimensionnalité pour les représenter dans des espaces à deux ou trois dimensions.

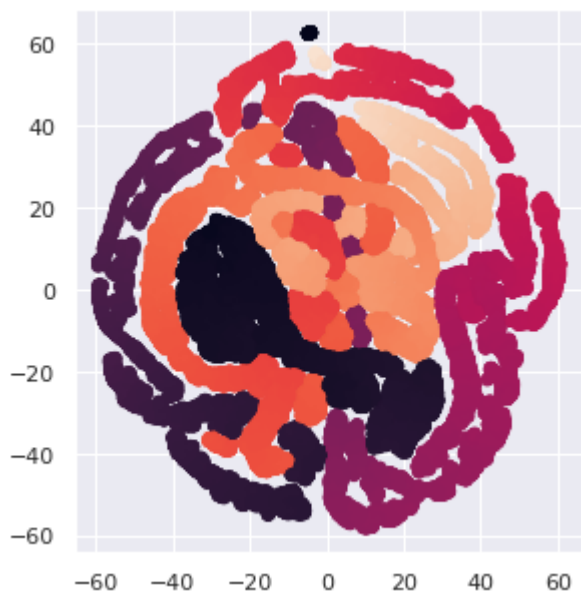
Le code

```

from sklearn.manifold import TSNE
import matplotlib.pyplot as plt
X_tsne = TSNE(learning_rate=100).fit_transform(data)
plt.figure(figsize=(10, 5))
plt.subplot(121)
plt.scatter(X_tsne[:, 0], X_tsne[:, 1], c=data.target)

```

Les résultats



1.2.2 PCA

Explication de l'algorithme

Principal Component Analysis pour des données numériques en n dimensions est un algorithme non supervisé d'identification des dimensions de variance décroissante et de changement de base pour ne conserver que les k dimensions de plus grande variance.

Le code

```

from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA

#features = data.columns[0:len(data.columns)-1]
#x = data.loc[:, features].values
#y = data.loc[:, ['target']].values

scaler = StandardScaler()
scaler.fit(data)
StandardScaler(copy=True, with_mean=True,
               with_std=True)
scaled_data = scaler.transform(data)

pca = PCA(0.99)
pca.fit(scaled_data)

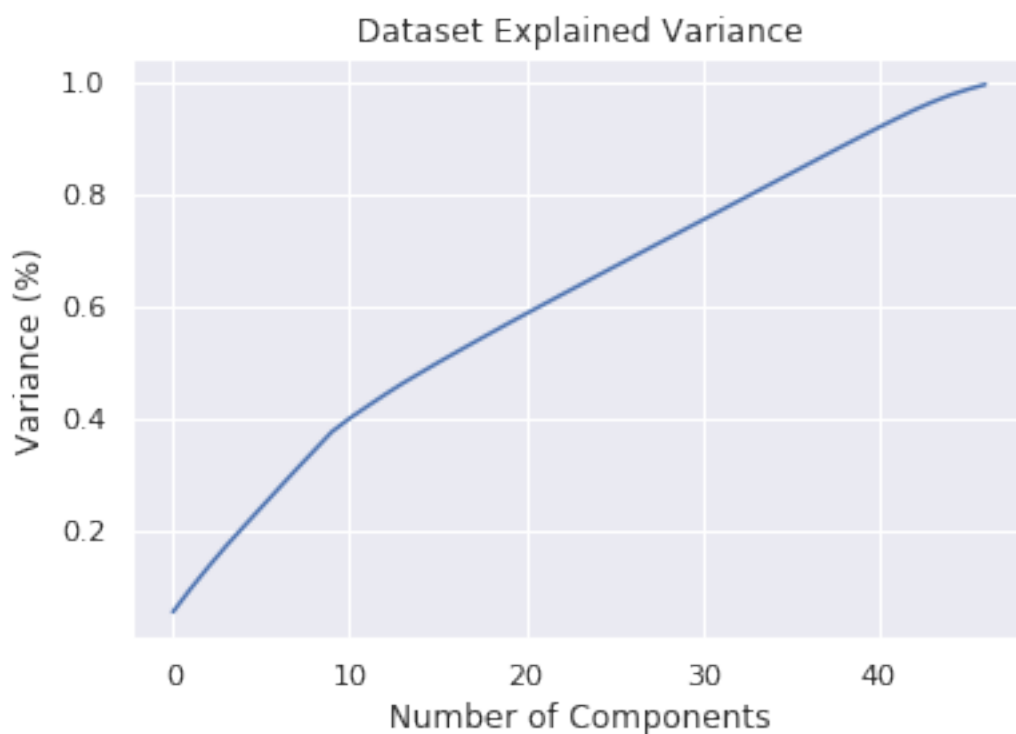
x_pca = pca.transform(scaled_data)
print(x_pca.shape)
pca.explained_variance_ratio_

```

Les résultats

En appliquant la méthode de PCA on a arrivé à observer la participation de chaque "feature" dans la representation des données. Pour concrétiser ce résultat on a fait un plot de nombre de "features" en fonction de la variance expliquée. Voici le code utilisé et le plot obtenu.

```
plt.figure()
plt.plot(np.cumsum(pca.explained_variance_ratio_))
plt.xlabel('Number_of_Components')
plt.ylabel('Variance_(%)') #for each component
plt.title('Dataset_Explained_Variance')
plt.show()
```



1.3 Feature selection

1.3.1 Explication des methodes utilisées

SelectKBest est une class qui note les caractéristiques à l'aide d'une fonction (f regression dans ce cas) et supprime ensuite toutes les caractéristiques sauf les k les plus élevées.

Le code

```
from sklearn.feature_selection import SelectKBest,
                                     f_regression
X = data[data.columns[0:len(data.columns)-1]].values
Y = data[data.columns[-1]].values
X=SelectKBest(score_func=f_regression,k=47).
    fit_transform(X,Y)
initial_feature_names = data.columns
for i in initial_feature_names:
    for j in range(47):
        a = np.equal(X[:,j],data[i].values)
        if a[j]:
            print(i)
X.shape()
```

Résultat sur le score final

Après l'utilisation de la méthode montré ci dessus sur le modèle créer par le groupe Model, on remarque que la performance pour certain modèle augmente et pour d'autres diminue. Voici les scores obtenu par le groupe Model :

- KNeighborsRegressor score : 0.77 (+/- 0.01)
- Decision Tree Regressor score : 0.91 (+/- 0.01)
- RandomForestRegressor score : 0.94 (+/- 0.01)
- GradientBoostingRegressor score : 0.92 (+/- 0.01)
- ExtraTreesRegressor score : 0.94 (+/- 0.00)
- Bagging Regressor score : 0.95 (+/- 0.00)
- AdaBoost Regressor score : 0.83 (+/- 0.02)

Voici les scores qu'on a obtenu après la sélection des "features" :

- KNeighborsRegressor score : 0.90 (+/- 0.01)
- Decision Tree Regressor score : 0.89 (+/- 0.01)
- RandomForestRegressor score : 0.94 (+/- 0.01)
- GradientBoostingRegressor score : 0.92 (+/- 0.00)
- ExtraTreesRegressor score : 0.92 (+/- 0.01)
- Bagging Regressor score : 0.93 (+/- 0.01)
- AdaBoost Regressor score : 0.83 (+/- 0.01)

Chapitre 2

Team Model :

Xuewen Guo & Kevin Tran

L'objectif de notre groupe était d'entraîner plusieurs modèles de régression, de les tester via la méthode de cross-validation. La première partie de notre tâche a été chercher dans la librairie de *scikit-learn* [1] les modèles que l'on voulait entraîner.

Les modèles sont des méthodes de cette librairie qui lors de son appel à des paramètres (qui sont les hyper-paramètres de notre modèle) prédéfinis. Pour améliorer les scores de notre modèle nous avons dû lire et sélectionner les hyper-paramètres que nous voulions optimiser.

Le choix a dû se faire en tenant en compte le temps d'entraînement de notre modèle, le risque de sur-apprentissage (overfitting), etc..

2.1 Les différents modèles testés

Le code

```
#KNeighborsRegressor
from sklearn.neighbors import KNeighborsRegressor

#DecisionTreeRegressor
from sklearn import tree

#RandomForestRegressor
from sklearn.ensemble import RandomForestRegressor

#GradientBoostingRegressor
from sklearn.ensemble import GradientBoostingRegressor

#ExtraTreesRegressor
from sklearn.ensemble import ExtraTreesRegressor

#BaggingRegressor
from sklearn.ensemble import BaggingRegressor

#AdaBoostRegressor
from sklearn.ensemble import AdaBoostRegressor

model_list = [
    KNeighborsRegressor(),
    tree.DecisionTreeRegressor(),
    RandomForestRegressor(),
    GradientBoostingRegressor(),
    ExtraTreesRegressor(),
    BaggingRegressor(),
    AdaBoostRegressor()
]
```

2.1.1 Cross-validation

code

```
from sklearn.metrics import make_scorer
from sklearn.model_selection import cross_val_score
for i in range(len(model_list)):
    scores = cross_val_score(model_list[i], X_train, Y_train, cv=5,
                              scoring=make_scorer(scoring_function))
    print('\n'+model_name[i]+' _score_: _%0.2f_(+/-_%0.2f)' %
          (scores.mean(), scores.std() ))
```

Résultat

Le score de ses différents modèles étant :

- KNeighborsRegressor score : 0.77 (+/- 0.01)
- Decision Tree Regressor score : 0.91 (+/- 0.01)
- RandomForestRegressor score : 0.94 (+/- 0.01)
- GradientBoostingRegressor score : 0.92 (+/- 0.01)
- ExtraTreesRegressor score : 0.94 (+/- 0.00)
- Bagging Regressor score : 0.95 (+/- 0.00)
- AdaBoost Regressor score : 0.83 (+/- 0.02)

2.2 RandomForest

Lors de notre test, le meilleur modèle de notre liste était le RandomForestRegressor [2]. Dans cette section nous vous expliquerons comment procède le modèle en vous présentant son algorithme [4].

Algorithm 1 Random Forest

Precondition: A training set $S := (x_1, y_1), \dots, (x_n, y_n)$, features F , and number of trees in forest B .

```
1 function RANDOMFOREST( $S, F$ )
2    $H \leftarrow \emptyset$ 
3   for  $i \in 1, \dots, B$  do
4      $S^{(i)} \leftarrow$  A bootstrap sample from  $S$ 
5      $h_i \leftarrow$  RANDOMIZEDTREELEARN( $S^{(i)}, F$ )
6      $H \leftarrow H \cup \{h_i\}$ 
7   end for
8   return  $H$ 
9 end function
10 function RANDOMIZEDTREELEARN( $S, F$ )
11   At each node:
12      $f \leftarrow$  very small subset of  $F$ 
13     Split on best feature in  $f$ 
14   return The learned tree
15 end function
```

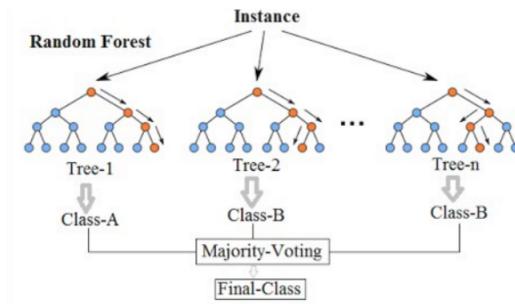
Cet algorithme permet de créer notre modèle selon un ensemble de données et de features.

Pour un nombre d'arbre fixé dans la forêt (B dans l'algorithme), l'algorithme va créer une liste d'arbres binaires de décision. Pour cela il sélectionne une partie des données $S^{(i)}$. Et appelle la fonction **Randomized-TreeLearn**, cette fonction va créer un arbre en plaçant à chaque noeuds de l'arbre une des meilleurs features testé sur la partie $S^{(i)}$.

Mini-Projets L2 2019-2020

Some classifiers...

Random Forest

Ensemble of
decision tree!

Hyper parameters:

- **n**, number of decision tree
- max depth of the trees

<https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html>

De cette façon chaque donnée parcourra les B arbres créés par l'algorithme. Chaque arbre aura la même longueur mais des nœuds différents car chaque partie des données sera différente. Cela est particulièrement efficace lorsque l'ensemble de données S est grand pour pouvoir définir des parties disjointes et ainsi avoir des arbres très différents. L'image ci-dessus donne une idée de ce qu'est notre modèle mais il s'agit ici du modèle non regressor.

2.3 Hyper-paramètre

Lors du TP4, nous avons vu les différents hyper-paramètres du modèle `RandomForestRegressor` [2]. Sur la version du modèle, il y a 18 paramètres, pour améliorer notre modèle nous n'avons pu chercher les meilleures valeurs de chaque paramètre pour nos données.

En effet, pour choisir de meilleur paramètre nous avons utilisé la méthode `GridSearchCV` [5]. Cette méthode prend les hyper paramètres et test chaque n -uplet. Par exemple pour nos paramètres, elle prendra (`max_depth = 5`, `n_estimators = 50`, `min_samples_split = 2`) puis avec les mêmes `max_depth` et `n_estimators` mais avec `min_samples_split = 3`, etc... Cette méthode prend énormément de temps, et il était à nous de choisir les ensembles où les hyper-paramètres se baladent. Exemple, '`max_depth`' : $[5, 20]$. Cette ensemble peut être très grand.

Nous avons finalement choisi les paramètres :

- `max_depth`
- `n_estimators`
- `min_samples_split`

`max_depth` nous permet de déterminer la longueur optimale des arbres de la forêt, `n_estimators` nous permet d'avoir le nombre d'arbre dans notre forêt et `min_samples_split` est le paramètre qui gère le nombre de données de chaque partie $S^{(i)}$ que nous avons vu dans l'algorithme.

code

```

parameters = { 'max_depth' : [5, 20], 'n_estimators' : [50, 100],
               'min_samples_split' : [2, 5] }
model = RandomForestRegressor()
clf = GridSearchCV(model, parameters)

```

2.4 La suite

Comme nous avons pu le voir dans ce chapitre, les features sont des acteurs importants sur la création de nos modèles comme nous avons pu le voir sur la présentation de l'algorithme du `RandomForest`.

Ces features peuvent à la fois réduire les biais et le temps d'apprentissage. Par la suite nous essayerons de tirer au mieux des deux parties pre-processing et model pour pouvoir présenter un meilleur score.

Bibliographie

- [1] Lien vers les différents modèles utilisés :
<https://scikit-learn.org/stable/index.html>
- [2] Lien de la méthode RandomForestRegressor utilisée :
<https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html#sklearn.ensemble.RandomForestRegressor>
- [3] Image du cours sur les différents classifieurs :
https://docs.google.com/presentation/d/1vfdS9ttwvK3mzGcoRBoe-n8iVzY0zSKUZsemXjDnPmA/edit#slide=id.g6e8cea48c5_0_65
- [4] Lien vers l'algorithme du modèle étudié :
<http://pages.cs.wisc.edu/~matthewb/pages/notes/pdf/ensembles/RandomForests.pdf>
- [5] Lien vers la méthode gridSearchCV :
https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html

Chapitre 3

Team Visualisation : Hamza Manita Jelidi & Hugo Penicand

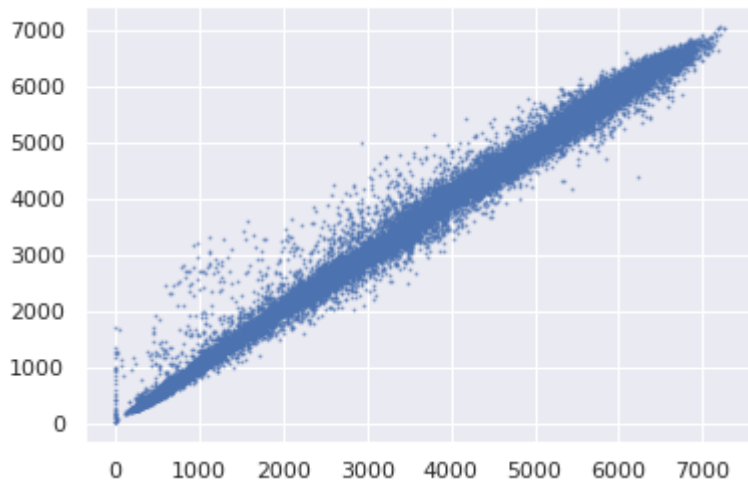
L'objectif de notre partie est de fournir une représentation lisible et précise des différentes étapes du projet afin qu'elles soient accessibles et compréhensibles par tous. Cela inclue la mise en évidence des données de départ et la représentation des résultats obtenues par notre modèle. Il s'agit également de proposer des explications concises et claires, ainsi qu'une interprétation de ce qui est observable à partir de nos interfaces graphiques. Pour cela, il est important de varier les types d'affichage et d'utiliser autant que possible des techniques permettant une visibilité accrue (couleur, forme, courbes, etc.).

Notre projet s'articule autour d'un modèle de régression appelé « régression linéaire ». Cette méthode d'analyse de données regroupe des algorithmes d'apprentissage supervisé adaptés aux données quantitatives. L'objectif est d'apprendre (autrement dit de trouver) la relation qui lie une variable d'intérêt, de type quantitatif, aux autres variables observées, éventuellement dans un but de prédiction. Pour cela, on utilise des paramètres liés à notre apprentissage (Target, qui représentent les données réelles) et on les compare avec des Meta-paramètres (les données théoriques). En termes d'interfaces graphiques, les barplots, les lineplots et les boxplots nous paraissent les plus indiqués. Nous avons utilisé les bibliothèques open-sources et celles intégrées à Python telles que Matplotlib, Numpy, Seaborn et Sklearn pour.

Remarque : De nombreuses visualisations nous étaient déjà proposées dans le TP4, nous avons donc repris en grande partie ces dernières que nous avons agrémenté d'explications de lectures et d'interprétation des résultats.

3.1 Scatterplot

Résultat



Le code

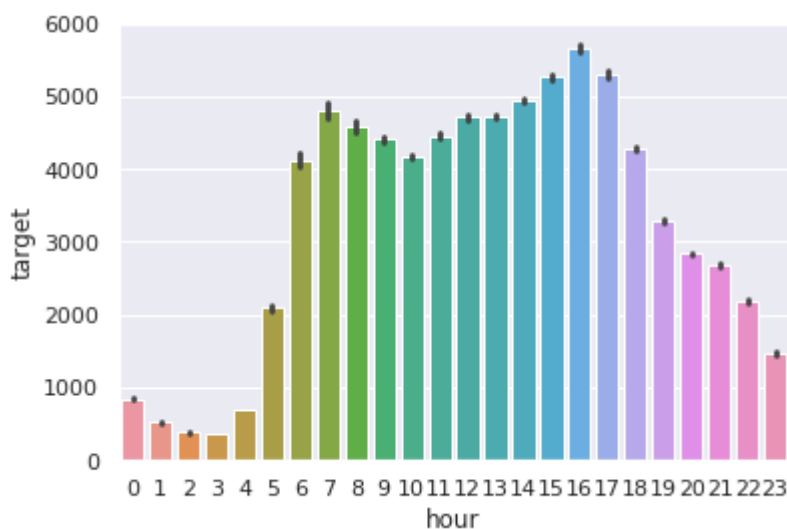
```
#graphe qui correspond valeurs theoriques en fonction des valeurs observees
plt.scatter(Y_train, Y_hat_train, alpha = '0.5', s = 1)
plt.show()
```

Explication

La régression linéaire simple permet d'évaluer la significativité du lien linéaire entre deux variables. Ci-dessus nous avons \hat{Y}_{train} qui correspond aux valeurs théoriques de nos données d'entraînement sur l'axe des abscisses (les valeur prédictive), et Y_{train} qui correspond aux vraies valeurs de nos données d'entraînement sur l'axe des ordonnées (les valeur observée). On peut voir très clairement qu'il y a une corrélation linéaire entre les variables. De plus la courbe observée croît ce qui indique le bon fonctionnement de la régression. Une telle visualisation permet une lecture immédiate des résultats de la régression mais également une bonne compréhension des erreurs éventuelles.

3.2 Barplot (Hour,target)

Résultat



Le code

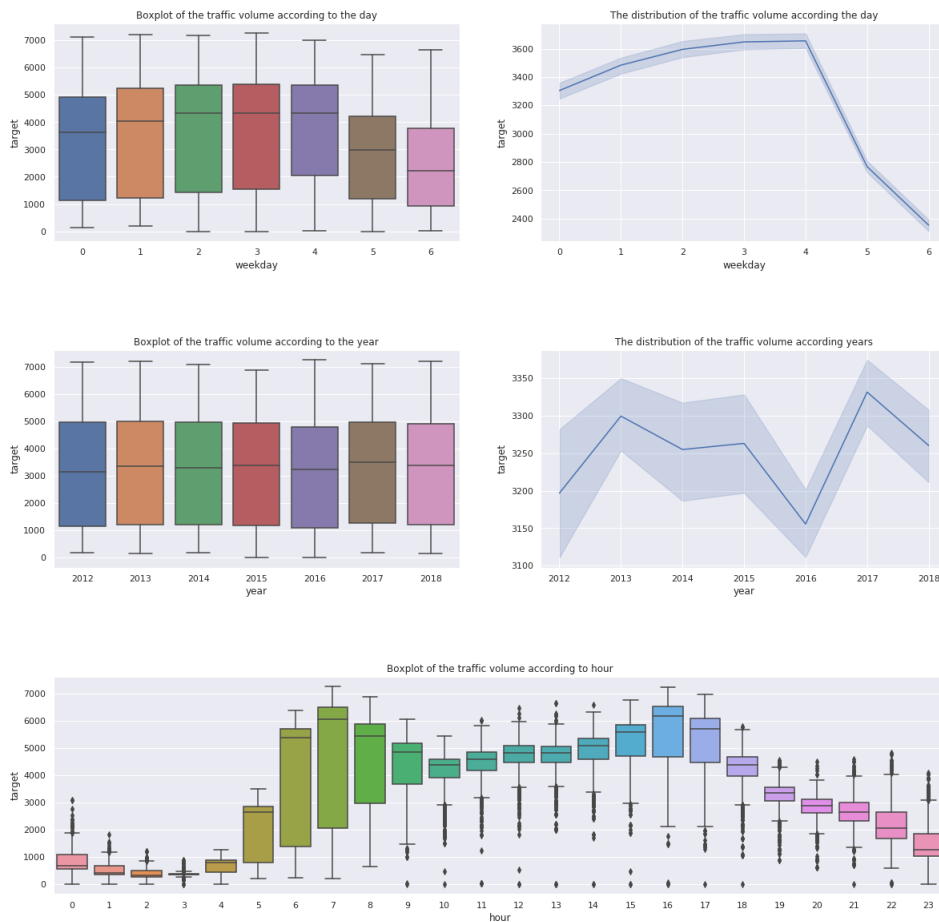
```
#graphe qui compare nos donnees en fonction des heures
ax = sns.barplot(x="hour", y="target", data=data)
```

Explication

On peut voir ici le volume du trafic en fonction des heures d'une journée analyse : On peut constater qu'il y a 2 pics d'affluences un autour de 7 heure ainsi qu'un autre autour de 16h. Entre ces 2 horaires le trafic restes très dense. Enfin entre minuit est 5 heure le trafic est très faible. On peut interpréter cela comme la période de travail (entre 6h et 18h) qui constitue une sources de déplacement constante.

3.3 Boxplot

Résultats



Le code

```
fig = plt.figure(figsize=(20,20)) //dimension du graphe
fig.subplots_adjust(hspace = 0.5)

#graphe qui compare le trafic par heure au donnee en boxplot
plt.subplot(313)
sns.boxplot('hour','target', data= data)
plt.title('Boxplot_of_the_traffic_volume_according_to_hour')

#graphe qui compare le trafic par jour au donnee en boxplot
plt.subplot(321)
sns.boxplot('weekday','target', data= data)
plt.title('Boxplot_of_the_traffic_volume_according_to_the_day')
```

```
#graphe qui compare le trafic par heure au donnee en line plot
plt.subplot(322)
sns.lineplot('weekday','target', data= data)
plt.title("The_distribution_of_the_traffic_volume_according_the_day")

#graphe qui compare le trafic par annee au donnee en boxplot
plt.subplot(323)
sns.boxplot('year','target', data= data)
plt.title('Boxplot_of_the_traffic_volume_according_to_the_year')

#graphe qui compare le trafic par heure au donnee en lineplot
plt.subplot(324)
sns.lineplot('year','target', data= data)
plt.title("The_distribution_of_the_traffic_volume_according_years")
plt.show()
```

Explication

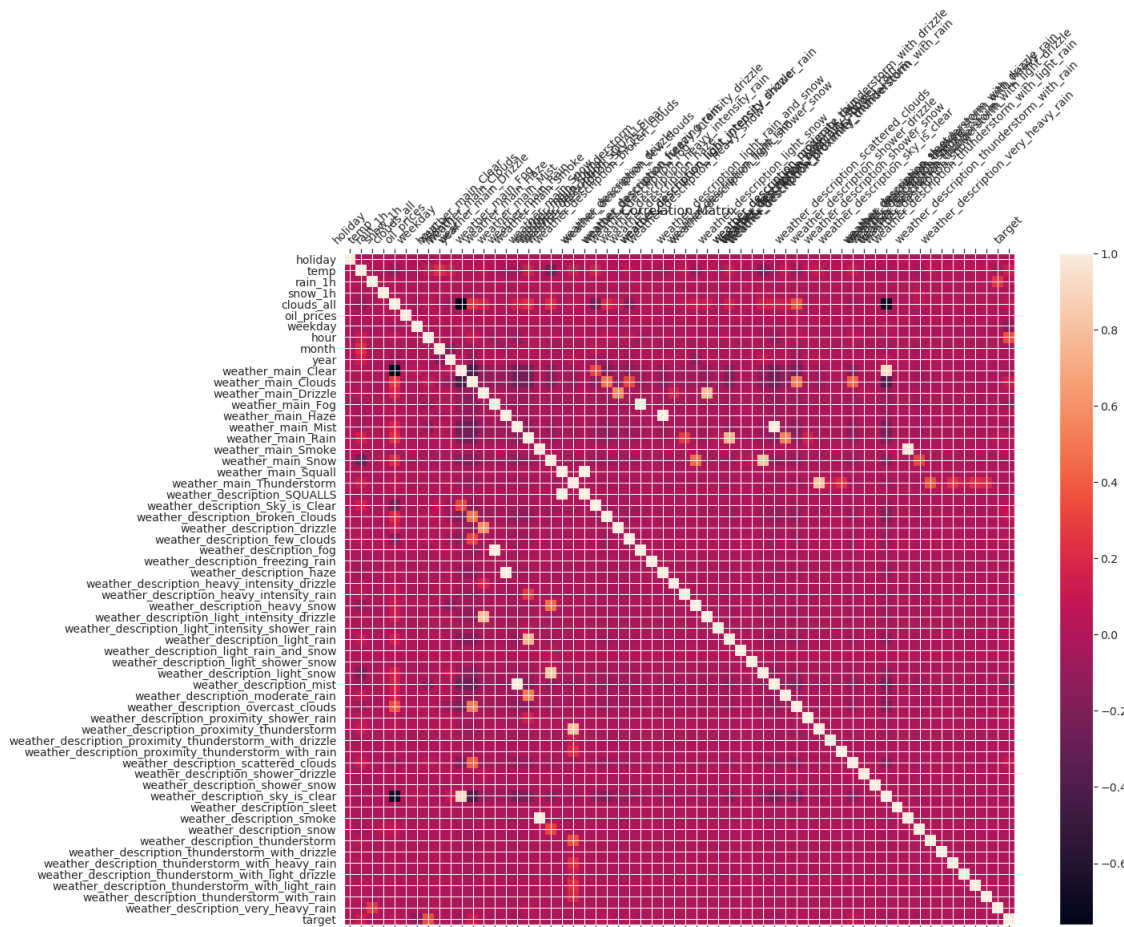
Nous avons ci-dessus plusieurs Boxplots représentant la Target en fonction d'un autre paramètre. Ce type de représentation indique pour chaque box :

- La médiane (la valeur centrale du graphique, il existe autant de valeur supérieure qu'inférieures à cette valeur dans l'échantillon);
- Les quartiles (qui sont es bords du rectangle est Pour le bord inférieur, un quart des observations ont des valeurs plus petites et trois quart ont des valeurs plus grandes, le bord supérieur suit le même raisonnement);
- Les extrémités des Box sont calculées en utilisant 1.5 fois l'espace interquartile (la distance entre le 1er et le 3ème quartile). Les box plots offrent plus d'information et de précision qu'un barplot classique mais cela diminue la lisibilité.

Ici les deux premiers boxplot sont accompagné d'un lineplot offrant une autre vision des mêmes données. En effet les lineplots mettent l'accent sur l'évolution des données.

3.4 Correlation Matrix

Résultat



Le code

```
f = plt.figure(figsize=(20, 15))

#on cree la matrice de confusion lie a nos donnee

plt.matshow(data.corr(), fignum=f.number)
plt.xticks(range(data.shape[1]), data.columns, fontsize=14, rotation=45)
plt.yticks(range(data.shape[1]), data.columns, fontsize=14)
cb = plt.colorbar()
cb.ax.tick_params(labelsize=14)
plt.title('Correlation_Matrix', fontsize=16);
```

Explication

Une matrice de corrélations attribue un coefficient de corrélation (un poids) à une paire de paramètres. Plus le coefficient est élevé plus les paramètres sont corrélés.

Analyse : Évidemment chaque paramètre est parfaitement corrélé à lui-même (anti-diagonale blanche). Cependant on peut observer quelque point clair en dehors de cette diagonale nous indiquant une corrélation entre deux paramètres différents.