Atelier 2 : Régression
Pr. Ali Idri
Master Quantitative Financial Modeling
2022-2023

Objectifs

Cet atelier a pour objectif de tester quatre types de modèles de régression à savoir la régression linéaire, Support Vector Regressor, Regression Trees, et le perceptron multi couches selon le plan suivant :

- Importation des modèles du package « Scikit-learn » et leurs tests et validation par les méthodes : Holdout, et validation croisée.
- Tuning des hyperparamètres par grid search.
- Feature selection.
- Etude de l'impact de normalisation sur l'apprentissage de chaque modèle.
- Visualisation des résultats.

1. Charger, explorer et prétraiter la base de données « Auto MPG »

La première étape consiste à définir et à explorer la base de données « Auto MPG » qui concerne la consommation de carburant en cycle urbain en miles par gallon (mpg), à prédire en termes de 1 attribut discret à valeurs multiples et de 6 attributs continus.
Pour plus de détails sur le dataset visitez : https://archive.ics.uci.edu/ml/datasets/auto+mpg
Les attributs sont :

- **mpg** : continu (target)
- **cylindres** : continue
- **déplacement** : continu
- **puissance** : continue
- **poids** : continu
- **accélération** : continue
- **année modèle** : continue
- **origine** : discrète à valeurs multiples
- **nom de la voiture** : chaîne (unique pour chaque instance)

1.1. Charger la base de données

Nous pouvons charger la base de données comme un DataFrame pandas directement à partir du ficher CSV.

```
#charger la dataset en utilisant pandas
import pandas as pd
df = pd.read_csv('autos_mpg.csv')
print(df.shape)
```

1.2. Explorer le dataset

```
#Explore dataset
df.info()
```

1.3. Prétraitements

La colonne HP est dans un format String, nous devrons donc la convertir en format INT ou float.

```
#changer le type de la variable HP
df['HP'] = pd.to_numeric(df['HP'], errors = 'coerce')
```

Puis il faut vérifier l'existence des valeurs nulles et les supprimer :

```
#Check the existence of null vaues
df.isna().sum()
#delete nan values
df.dropna(axis = 0, how = "any", inplace=True)
```

La colonne NAME contient les noms de chaque voiture dans un format de chaîne. nous allons explorer la colonne Nom pour voir combien de valeurs uniques il y a et si elle peut être utilisée comme fonctionnalité.

```
# name variable explore
df['NAME'].value_counts()
```

Il y a 301 valeurs de catégorie uniques dans la colonne NAME, ce qui représente 77 % des 392 enregistrements. Par conséquent, il est décidé de l'ignorer et de ne pas l'utiliser comme attribut.

```
# Supprimer le nom de voiture
df.drop('NAME', axis = 1, inplace= True)
```

Voyons la corrélation entre les autres variables et la colonne MPG!

```
# Correlations
import seaborn as sns
sns.heatmap(df.corr(), annot = True, cmap = 'coolwarm')
```

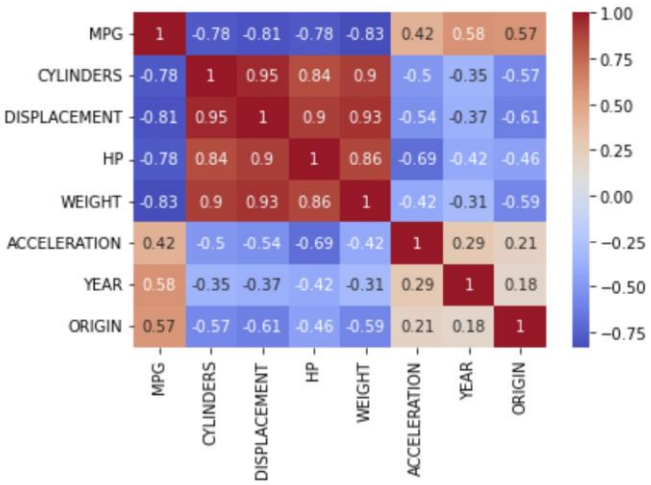

Fig 1. Corrélations entre les attributs du dataset Auto MPG.

Qst : Qu'est ce que vous remarquez de la figure 1?

Pour l'encodage. On a une seule variable qui est vraiment qualitative. C'est l'origine. Pour les autres variables qui concerne le nombre de cylindre et l'année on va les laisser dans leurs formats numériques.
On va utiliser deux types d'encodage : Ordinal Encoder et Dummies variables.
Tout d'abord, il faut importer la méthode « OrdinalEncoder ». La méthodes de « get_dummies » est inclue dans la librairie « pandas ».

```
from sklearn.preprocessing import OrdinalEncoder
```

Le code suivant présente comment encoder par la méthode de dummies :

```
# Dummies
df_dummies = pd.get_dummies(df,prefix=['ORIGIN','CYLINDERS','YEAR'], columns = ['ORIGIN'], drop_first=True)
df_dummies.head()
```

Pour créer un nouveau dataframe en utilisant Ordinal Encoder, on suit le code :

```
# Ordinal encoder
df_toenc = df[['ORIGIN']]
df_toenc = df_toenc.to_numpy()
enc = OrdinalEncoder()
enc_fitted = enc.fit(df_toenc)
encoded = enc_fitted.transform(df_toenc)
df_ordinal = df
df_ordinal['ORIGIN'] = encoded
df_ordinal.head()
```

2. Régression linéaire

Commençant par le plus simple modèle de régression : La régression linéaire.

2.1. Avec holdout

D'abord on va importer les packages nécessaires.

```
#import necessary packages
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from sklearn import linear_model
```

Puis on va faire le split du data. On le fait maintenant avec le dataframe encodé par méthode de dummies.

```
#split the data
X = df_dummies.drop(["MPG"], axis=1).values[:,:]
y = df_dummies["MPG"]  # label
X_train, X_test, y_train, y_test = train_test_split(X,y, test_size=0.25)
```

Et maintenant on va créer notre modèle de régression linéaire et le tester en utilisant la mesure R2.

```
Model = linear_model.LinearRegression()
Model.fit(X_train, y_train)
yhat_LR = model.predict(X_test)
#évaluer les predictions
score = mean_squared_error(y_test, yhat_LR)
print('MSE : %.3f' %score)
```

On peut refaire le split avec le dataframe résultat de la méthode get_dummies et comparer les résultats en remplissant le tableau suivant :

(Pour plus de détails sur la régression linéaire, visitez : https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html)

Régression linéaire	**Avec dummies**	**Avec Ordinal encoder**
MAE		
MSE		
R2		

2.2. Feature selection

Dans cette partie, on veut tester le modèle de régression linéaire avec la méthode de sélection backward et forward (Déjà vu dans le TP passé). Pour cela on aura besoin du package « mlxtend » qu'on peut installer par la commande : conda install mlxtend ou pip install mlxtend, puis l'importer la méthode de sélection comme suivant :

```
from mlxtend.feature_selection import SequentialFeatureSelector as sfs
```

a- Forward feature selection

Cette technique commence par un seul attribut et essaye de rajouter de plus en plus des autres jusqu'à ne plus avoir d'amélioration dans les résultats.

```
model_LR = linear_model.LinearRegression()

sfs = sfs(model_LR, k_features='best',scoring='r2', cv=5, forward = False)
sfs = sfs.fit(X_train, y_train) #trouver les meilleurs attributs
#transformer la matrice original et ne laisser que les attributs sélectionnés
X_train_sfs = sfs.transform(X_train)
# transforation de la matrice de test
X_test_sfs = sfs.transform(X_test)
# Entrainer le modèle de régression sur les données d'entrainements réduites
model_LR.fit(X_train_sfs, y_train)
#Prédiction sur les données de test
y_pred = model_LR.predict(X_test_sfs)
```

```
#évaluer les predictions
score =r2_score(y_test, y_pred)
print('R2 : %.3f' %score)
```

Vous pouvez aussi visualiser les résultats de sélection en utilisant le code suivant :

```
sfs_results = pd.DataFrame.from_dict(sfs.get_metric_dict()).T
sfs_results.sort_values(by='avg_score', ascending=False, inplace=True)
sfs_results
```

b- Backward feature selection

Cette technique commence par inclure tous les attributs et essaye d'éliminer les attributs un par un jusqu'à ne plus avoir d'amélioration dans les résultats.

Pour tester cette méthode il suffit de mentionner dans le code en haut « Forward=False ».

3. Tree Regressor

3.1. Avec Holdout

Commençant par importer les packages nécessaires.

```
from sklearn.tree import DecisionTreeRegressor
```

On va garder le même split qui a été déjà fait.
Et maintenant on va créer notre modèle de régression par arbre et le tester en utilisant la mesure R2.

```
model = DecisionTreeRegressor(random_state=0)
model.fit(X_train,y_train)
#prédire test set
yhat_tree = model.predict(X_test)
#évaluer les predictions
score = mean_squared_error(y_test, yhat_tree)
print('MSE : %.3f' %score)
```

3.2. Avec validation croisée

Tout d'abord on va importer les packages nécessaires.

```
from sklearn.model_selection import cross_val_score
import numpy as np
```

Maintenant on passe à la création, l'apprentissage et le test de notre modèle avant et après utilisation de 5 folds.

```
dt = DecisionTreeRegressor(random_state=0)
dt_fit = dt.fit(X, y)
dt_scores = cross_val_score(dt_fit, X, y, cv = 5,
scoring='neg_mean_squared_error')
print("mean cross validation score: {}".format(np.mean(dt_scores)))
```

3.3. Avec validation croisée et grid search

Ici on va entrainer notre arbre de régression en utilisant 5 folds et le grid search sur deux paramètres :

- **min_samples_split** : Le nombre minimum d'échantillons requis pour diviser un nœud interne.
- **max_depth** : La profondeur maximale de l'arbre. Si aucun, alors les nœuds sont étendus jusqu'à ce que toutes les feuilles soient pures ou jusqu'à ce que toutes les feuilles contiennent moins que les échantillons min_sample_split.

(Pour plus de détails sur les hyperparamètres de DecisionTreeRegressor, visitez : https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeRegressor.html)

```
#cv and grid search
from sklearn.model_selection import GridSearchCV

model = DecisionTreeRegressor()
parameter_space = {
    'min_samples_split': range(2, 10),
    'max_depth': (20, 30, 50)
}
reg = GridSearchCV(model, parameter_space, scoring =
'neg_mean_squared_error', cv=5, n_jobs=-1)
reg.fit(X, y)
# Afficher les resultats
print("Best score: %f using %s" % (abs(reg.best_score_), reg.best_params_))
print("Best estimator: ",reg.best_estimator_)
```

To do: Refaire le test en utilisant les deux autres mesures MAE et R2 et remplir le tableau.
(Pour plus de détails sur les métriques de régression, visitez : https://scikit-learn.org/stable/modules/model_evaluation.html)

Tree Regressor	Holdout	CV=5	CV=5 + Grid search
MAE			
MSE			
R2			

4. SVR

Dans cette partie, on va entrainer le modèle SVR en utilisant le même split.
D'abord, il faut implémenter les packages nécessaires :

```
from sklearn.svm import SVR
```

Puis on crée et on teste notre modèle :

```
from sklearn.model_selection import cross_val_score
model = SVR(C=1.0, epsilon=0.2)
model.fit(X_train, y_train)
yhat_SVR = model.predict(X_test)
#évaluer les predictions
score = mean_squared_error(y_test, yhat_SVR)
```

```
print('MSE : %.3f' %score)
```

To do: Refaire le même travail en utilisant la validation croisée et le grid search des hyperparamètres suivants et remplir la table.

parameter_space = {
'gamma': (1e-2, 1e-4),
'C': (1, 10),
'epsilon':[0.1,0.5,0.3]
}

(Pour plus de détails sur les hyperparamètres de SVR, visitez : https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVR.html)

SVR	Holdout	CV=5	CV=5 + Grid search
MAE			
MSE			
R2			

4.1. Feature selection

Vous pouvez aussi tester le modèle SVR avec une des méthodes de sélection. Ex : Sélection exhaustive (Déjà vu dans le TP passé) en utilisant le code suivant :

```
# with exhaustive fs and cv
from mlxtend.feature_selection import ExhaustiveFeatureSelector as EFS
lr = linear_model.LinearRegression()
efs = EFS(estimator=lr, # The Ml model
min_features=1,
max_features=6,
scoring='r2', cv=5)
efs = efs.fit(X, y)
print('Best accuracy score: %.2f' % efs.best_score_) # best_score_ shows
the best score
print('Best subset (corresponding names):', efs.best_feature_names_)
```

Le code suivant permet de visualiser les résultats de performances de chaque sous-ensembles d'attributs :

```
# Show the performance of each subset of features
efs_results = pd.DataFrame.from_dict(efs.get_metric_dict()).T
efs_results.sort_values(by='avg_score', ascending=False, inplace=True)
efs_results
```

Maintenant il faut transformer le dataset pour garder que les attributs sélectionnés :

```
#Transformer le dataset pour garder seulement les attributs sélectionnés
X_train_efs = efs.transform(X_train)
X_test_efs = efs.transform(X_test)
```

Et maintenant on peut entrainer notre modèle par le nouveau dataset :

```
# Entrainer le modèle de régression sur les données d'entrainements réduites
model_SVR = SVR()
model_SVR.fit(X_train_efs, y_train)
#Prédiction sur les données de test
y_pred = model_SVR.predict(X_test_efs)
#évaluer les predictions
score =r2_score(y_test, y_pred)
print('R2 : %.3f' %score)
```

Une autre manière plus simple pour faire les choses est d'utiliser les pipelines qui permet de faire des transformations d'une manière implicite. Le code de sélection avec SVR peut donc être réduit en code suivant :

```
from sklearn.pipeline import Pipeline
clf = Pipeline([
  ('feature_selection', EFS(linear_model.LinearRegression(), scoring='r2',
max_features=6, cv=5)),
  ('classification', SVR())
])
clf.fit(X_train, y_train)
clf.score(X_test, y_test)
```

5. MLP regressor

Dans cette partie, on va entrainer le modèle MLP regressor en utilisant le même split.
D'abord, il faut implémenter les packages nécessaires :

```
from sklearn.neural_network import MLPRegressor
```

Puis on crée et on teste notre modèle :

```
model = MLPRegressor(hidden_layer_sizes={100, 200,10}, activation='relu',
solver='adam', max_iter= 5000)
model.fit(X_train, y_train)
yhat_MLP = model.predict(X_test)
#évaluer les predictions
score = mean_squared_error(y_test, yhat_MLP)
print('MSE : %.3f' %score)
```

To do: Refaire le même travail en utilisant la validation croisée et le grid search des hyperparamètres suivants et remplir la table.

parameter_space = {
  'hidden_layer_sizes': [{100},{100, 200,50}, {100,100}],
  'solver': ['adam', 'sgd'],
  'batch_size':[200,50],
  'learning_rate':['0.001, 0.0001','adaptive']
}

Pour plus de détails sur les hyperparamètres de MLP Regressor, visitez :
https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPRegressor.html

MLP Regressor	Holdout	CV=5	CV=5 + Grid search
MAE			

MSE			
R2			

6. Normalization/Standarisation

Dans cette partie, on veut analyser l'impact de la normalisation et la standarisation sur l'apprentissage de nos modèles.

Min Max Normalisation :
Cet estimateur met à l'échelle et traduit chaque caractéristique individuellement de sorte qu'elle se trouve dans l'intervalle donnée, par ex. entre zéro et un.
La transformation est donnée par:

```
X_std = (X - X.min(axis=0)) / (X.max(axis=0) - X.min(axis=0))
X_scaled = X_std * (max - min) + min
```

où min, max = feature_range.
(Pour plus de détails, visitez :
https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html)

Standarisation :
Standardisez les caractéristiques en supprimant la moyenne et en mettant à l'échelle la variance unitaire.
Le score standard d'un échantillon x est calculé comme suit :

```
z = (x - u) / s
```

où u est la moyenne des échantillons d'apprentissage et s est l'écart type des échantillons d'apprentissage.
(Pour plus de détails, visitez :
https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html)

6.1. MLP Regressor en utilisant la normalisation et la standarisation

Dans cette partie on va appliquer la standarisation et la normalization de notre dataset. On va utiliser le package « make_pipeline » qui permet de faire un tunel d'aller-retour entre les valeurs originaux et les valeurs transformées.

Commençant par importer les packages nécessaires.

```
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler, MinMaxScaler
```

c- Standard Scaler

Avant d'utiliser la méthode de pipeline la plus simple, on veut voir d'abord comment la méthode de standarisation transforme nos données. Pour cela on va utiliser le code suivant :

```
# example of a standarization
# define min max scaler
scaler = StandardScaler()
# transform data
scaled = scaler.fit_transform(X)
scaled = pd.DataFrame(scaled)
scaled.describe()
```

Qst: Décrire les résultats obtenus.

Ici on va appeler le pipeline de standarisation du package « pipeline » appelé « StandardScaler » :

```
model = MLPRegressor(hidden_layer_sizes={100, 200,10}, activation='relu',
solver='adam', max_iter= 5000)
model = make_pipeline(StandardScaler(), model)
model.fit(X_train, y_train)
yhat = model.predict(X_test)
#évaluer les predictions
score = mean_squared_error(y_test, yhat)
print('MSE : %.3f' %score)
```

d- MinMax normalization

Pour voir comment la méthode MinMax transforme nos données, on utilise le code suivant :

```
# example of a normalization
# define min max scaler
scaler = MinMaxScaler()
# transform data
scaled = scaler.fit_transform(X)
scaled = pd.DataFrame(scaled)
scaled.describe()
```

Qst: Décrire les résultats obtenus.

Et maintenant on va appeler le pipeline:

```
model = MLPRegressor(hidden_layer_sizes={100, 200,10}, activation='relu',
solver='adam', max_iter= 5000)
model = make_pipeline(MinMaxScaler(), model)
model.fit(X_train, y_train)
yhat = model.predict(X_test)
#évaluer les predictions
score = mean_squared_error(y_test, yhat)
print('MSE : %.3f' %score)
```

To Do: Refaire le même travail pour LR, SVR and TreeRegressor en utilisant le holdout et compléter le tableau.

	No Stand/Norm			**StandardScaler**			**MinMaxScaler**		
Model/Metrics	**R2**	**MAE**	**MSE**	**R2**	**MAE**	**MSE**	**R2**	**MAE**	**MSE**
LR									
TreeRegressor									
SVR									
MLP Regressor									

Avec le pipeline

7. Visualisation

Pour visualizer les résidus de prédiction on utilise le code suivant :

```
fig, axs = plt.subplots(2, 2)
```

```
axs[0, 0].scatter(yhat_LR, yhat_LR-y_test)
axs[0, 0].plot([0,40],[0,0],color='black')
axs[0, 0].set_title("LR")
axs[0, 0].set_ylabel('Residuals (y_real-yhat)')
axs[1, 0].scatter(yhat_LR, yhat_tree-y_test)
axs[1, 0].plot([0,40],[0,0],color='black')
axs[1, 0].set_title("Tree Regressor")
axs[1, 0].set_ylabel('Residuals (y_real-yhat)')
axs[1, 0].sharex(axs[0, 0])
axs[0, 1].scatter(yhat_LR, yhat_SVR-y_test)
axs[0, 1].plot([0,40],[0,0],color='black')
axs[0, 1].set_title("SVR")
axs[0, 1].set_ylabel('Residuals (y_real-yhat)')
axs[1, 1].scatter(yhat_LR, yhat_MLP-y_test)
axs[1, 1].plot([0,40],[0,0],color='black')
axs[1, 1].set_title("MLP")
axs[1, 1].set_ylabel('Residuals (y_real-yhat)')
fig.tight_layout()
```

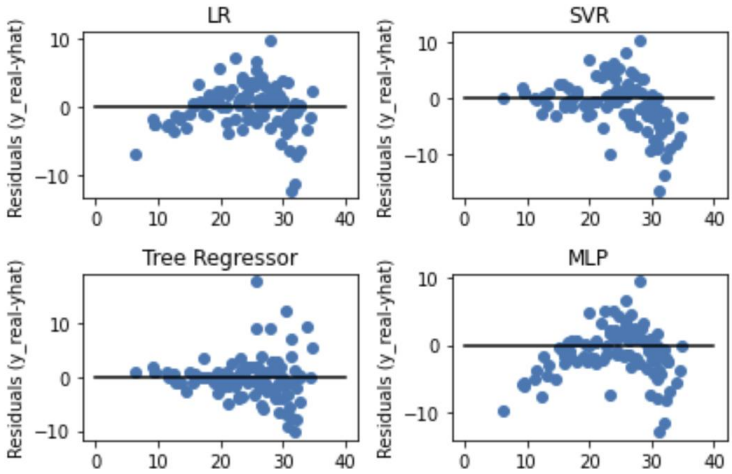

Fig 2. Visualisation des résidus pour chaque modèle.

On peut aussi visualiser les résidus par visualiser les valeurs réelles vs les valeurs prédites comme suit:

```
fig, axs = plt.subplots(2, 2)


axs[0, 0].scatter(y_test, yhat_LR, color = "g")
axs[0, 0].plot(y_test, y_test, color = "gray", label = "identity")
axs[0, 0].set_title("LR")
axs[0, 0].set_xlabel("y test(mpg)")
axs[0, 0].set_ylabel('y hat(mpg)')
axs[0, 0].legend()

axs[1, 0].scatter(y_test, yhat_tree, color = "g")
axs[1, 0].plot(y_test, y_test, color = "gray", label = "identity")
axs[1, 0].set_title("Tree Regressor")
axs[1, 0].set_xlabel("y test(mpg)")
axs[1, 0].set_ylabel('y hat(mpg)')
axs[1, 0].sharex(axs[0, 0])
axs[1, 0].legend()


axs[0, 1].scatter(y_test, yhat_SVR, color = "g")
```

```
axs[0, 1].plot(y_test, y_test, color = "gray", label = "identity")
axs[0, 1].set_title("SVR")
axs[0, 1].set_xlabel("y test(mpg)")
axs[0, 1].set_ylabel('y hat(mpg)')
axs[0, 1].legend()

axs[1, 1].scatter(y_test, yhat_MLP, color = "g")
axs[1, 1].plot(y_test, y_test, color = "gray", label = "identity")
axs[1, 1].set_title("MLP")
axs[1, 1].set_xlabel("y test(mpg)")
axs[1, 1].set_ylabel('y hat(mpg)')
axs[1, 1].legend()

fig.tight_layout()
```

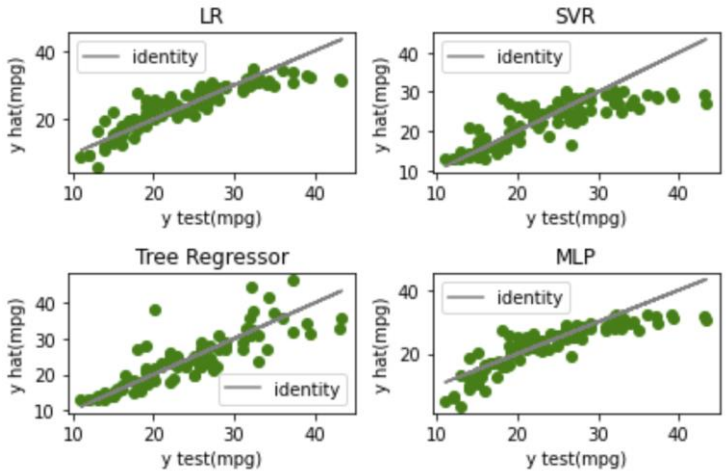

Fig 3. Visualisation des valeurs réelles vs valeurs prédites pour chaque modèle.

Pour la visualisation des prédictions de chaque modèle, on peut utiliser le package « Matplotlib ».

```
fig, axs = plt.subplots(2, 2)
axs[0, 0].plot(range(0,len(y_test)),yhat_LR, color= "red")
axs[0, 0].plot(range(0,len(y_test)),y_test, color= "green")
axs[0, 0].set_title("LR")
axs[1, 0].plot(range(0,len(y_test)),yhat_tree, color= "red")
axs[1, 0].plot(range(0,len(y_test)),y_test, color= "green")
axs[1, 0].set_title("Tree Regressor")
axs[1, 0].sharex(axs[0, 0])
axs[0, 1].plot(range(0,len(y_test)),yhat_SVR, color= "red")
axs[0, 1].plot(range(0,len(y_test)),y_test, color= "green")
axs[0, 1].set_title("SVR")
axs[1, 1].plot(range(0,len(y_test)),yhat_MLP, color= "red")
axs[1, 1].plot(range(0,len(y_test)),y_test, color= "green")
axs[1, 1].set_title("MLP")

fig.tight_layout()
```

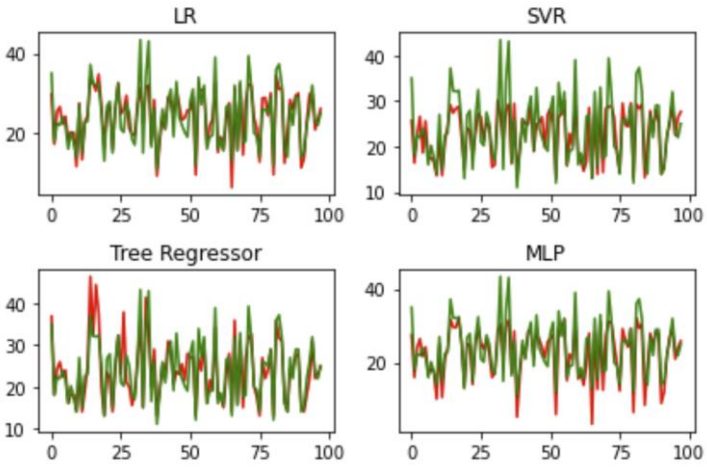

Fig 4. Visualisation des résultats de chaque modèle.