BASKARAN

Amalan

N°12001105

19/11/2023

TP 5 : Réseau de neurones pfofonds pour la régression

Entrée [1]:

```
!pip install scikeras
import pandas as pd
from keras.models import Sequential
from keras.layers import Dense
from scikeras.wrappers import KerasClassifier, KerasRegressor
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold
from sklearn.pipeline import Pipeline
from sklearn.model_selection import train_test_split
```

```
Collecting scikeras
  Downloading scikeras-0.12.0-py3-none-any.whl (27 kB)
Requirement already satisfied: packaging>=0.21 in /usr/local/lib/python3.10/dist-packages (from scikeras) (23.2)
Requirement already satisfied: scikit-learn>=1.0.0 in /usr/local/lib/python3.10/dist-packages (from scikeras) (1.2.2)
Requirement already satisfied: numpy>=1.17.3 in /usr/local/lib/python3.10/dist-packages (from scikit-learn>=1.0.0->scikera
s) (1.23.5)
Requirement already satisfied: scipy>=1.3.2 in /usr/local/lib/python3.10/dist-packages (from scikit-learn>=1.0.0->scikeras)
(1.11.4)
Requirement already satisfied: joblib>=1.1.1 in /usr/local/lib/python3.10/dist-packages (from scikit-learn>=1.0.0->scikera
s) (1.3.2)
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/python3.10/dist-packages (from scikit-learn>=1.0.0->s
cikeras) (3.2.0)
Installing collected packages: scikeras
Successfully installed scikeras-0.12.0
```

EXO 1 : Lecture et Labelisation du jeu de données

Entrée [2]:

```
df = pd.read_csv("/content/housing_data_for_regression.csv",delim_whitespace=True,header=None)
print(df.head())
```

```
        0     1     2   3      4      5     6       7   8      9     10  \
0  0.00632  18.0  2.31   0  0.538  6.575  65.2  4.0900   1  296.0  15.3
1  0.02731   0.0  7.07   0  0.469  6.421  78.9  4.9671   2  242.0  17.8
2  0.02729   0.0  7.07   0  0.469  7.185  61.1  4.9671   2  242.0  17.8
3  0.03237   0.0  2.18   0  0.458  6.998  45.8  6.0622   3  222.0  18.7
4  0.06905   0.0  2.18   0  0.458  7.147  54.2  6.0622   3  222.0  18.7

       11    12    13
0  396.90  4.98  24.0
1  396.90  9.14  21.6
2  392.83  4.03  34.7
3  394.63  2.94  33.4
4  396.90  5.33  36.2
```

Entrée [3]:

```
feature_names = ['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS', 'RAD', 'TAX', 'PTRATIO', 'B', 'LSTAT', 'MEDV']
# le nom qui va être assigné à chaque colonne
df.columns = feature_names # assignez le frature_names à chaque colonne
```

Entrée [4]:

```
print(df.head()) #affichage de l'entete des données.
```

```
      CRIM    ZN  INDUS  CHAS    NOX     RM   AGE     DIS  RAD    TAX  \
0  0.00632  18.0   2.31     0  0.538  6.575  65.2  4.0900    1  296.0
1  0.02731   0.0   7.07     0  0.469  6.421  78.9  4.9671    2  242.0
2  0.02729   0.0   7.07     0  0.469  7.185  61.1  4.9671    2  242.0
3  0.03237   0.0   2.18     0  0.458  6.998  45.8  6.0622    3  222.0
4  0.06905   0.0   2.18     0  0.458  7.147  54.2  6.0622    3  222.0

   PTRATIO       B  LSTAT  MEDV
0     15.3  396.90   4.98  24.0
1     17.8  396.90   9.14  21.6
2     17.8  392.83   4.03  34.7
3     18.7  394.63   2.94  33.4
4     18.7  396.90   5.33  36.2
```

Entrée [5]:

```
df = df.rename(columns={'MEDV': 'PRICE'}) # changemnt de nom de MEDV par price
print(df.describe()) # analyse de chaque colonne
```

```
             CRIM          ZN       INDUS        CHAS         NOX          RM  \
count  506.000000  506.000000  506.000000  506.000000  506.000000  506.000000
mean     3.613524   11.363636   11.136779    0.069170    0.554695    6.284634
std      8.601545   23.322453    6.860353    0.253994    0.115878    0.702617
min      0.006320    0.000000    0.460000    0.000000    0.385000    3.561000
25%      0.082045    0.000000    5.190000    0.000000    0.449000    5.885500
50%      0.256510    0.000000    9.690000    0.000000    0.538000    6.208500
75%      3.677083   12.500000   18.100000    0.000000    0.624000    6.623500
max     88.976200  100.000000   27.740000    1.000000    0.871000    8.780000

              AGE         DIS         RAD         TAX     PTRATIO           B  \
count  506.000000  506.000000  506.000000  506.000000  506.000000  506.000000
mean    68.574901    3.795043    9.549407  408.237154   18.455534  356.674032
std     28.148861    2.105710    8.707259  168.537116    2.164946   91.294864
min      2.900000    1.129600    1.000000  187.000000   12.600000    0.320000
25%     45.025000    2.100175    4.000000  279.000000   17.400000  375.377500
50%     77.500000    3.207450    5.000000  330.000000   19.050000  391.440000
75%     94.075000    5.188425   24.000000  666.000000   20.200000  396.225000
max    100.000000   12.126500   24.000000  711.000000   22.000000  396.900000

            LSTAT       PRICE
count  506.000000  506.000000
mean    12.653063   22.532806
std      7.141062    9.197104
min      1.730000    5.000000
25%      6.950000   17.025000
50%     11.360000   21.200000
75%     16.955000   25.000000
max     37.970000   50.000000
```

L'analyse des données commence souvent par l'examen de quelques statistiques descriptives, qui résument les caractéristiques principales des variables. Ces mesures peuvent aider à avoir une vue d'ensemble des données, à identifier des anomalies éventuelles comme des données manquantes ou aberrantes, ou à comparer des groupes différents.

Entrée [6]:

```
X = df.drop('PRICE', axis = 1) # les features serons tous sauf price.
y = df['PRICE'] # le label sera price
```

EXO 2 : Costruction et entrainement du modèle

Entrée [7]:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 20) # séparation des données.
```

Entrée [8]:

```
from sklearn.preprocessing import StandardScaler
scaler=StandardScaler()
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

StandardScaler() est une méthode de prétraitement des données qui permet de normaliser les valeurs des caractéristiques. Elle consiste à soustraire la moyenne et à diviser par l'écart-type de chaque caractéristique, afin de réduire l'effet des variations d'échelle entre les caractéristiques. Ainsi, les caractéristiques ont une moyenne nulle et une variance unitaire, ce qui facilite la comparaison et l'apprentissage des modèles.

Entrée [9]:

```
model = Sequential() #crée un modèle de réseau de neurones séquentiel avec trois couches
model.add(Dense(128, input_dim=13, activation='relu'))
# cette couche a couche a 128 neurones, avec une dimension d'entrée de 13 et une fonction d'activation ReLU
model.add(Dense(64, activation='relu'))
# cette couche a couche a 64 neurones et une fonction d'activation ReLU
model.add(Dense(1, activation='linear'))
# cette couche a 1 neurone et utilise une activation linéaire qui produira une prédiction continue
```

Entrée [10]:

```
model.compile(loss='mean_squared_error', optimizer='adam', metrics=['mae']) #compilation du modele
model.summary() #affiche un résumé du modèle
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 128)               1792

 dense_1 (Dense)             (None, 64)                8256

 dense_2 (Dense)             (None, 1)                 65

=================================================================
Total params: 10113 (39.50 KB)
Trainable params: 10113 (39.50 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Lorsqu'on compile un modele, on passe par plusieurs étapes :

- la fonction loss permettre de mesurer de l'erreur entre les prédictions du modèle et les valeurs réelles
- on ajoute un optimiseur qui sera responsable de la mise à jour des poids du modèle afin de minimiser la fonction de perte
- des métriques sont utilisées pour évaluer les performances du modèle pendant l'entraînement et l'évaluation une fois ces parametres sont fixé, on aura une modele.

Entrée [11]:

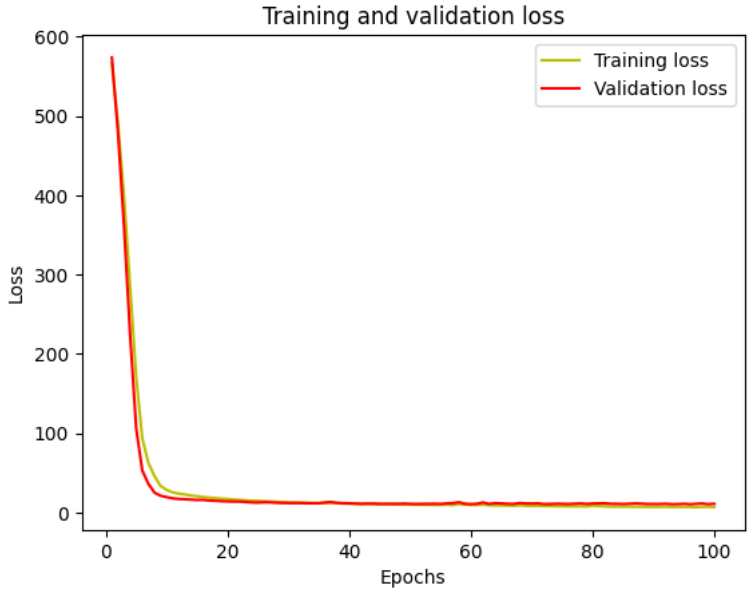
```
history = model.fit(X_train_scaled, y_train, validation_split=0.2, epochs =100) #entrainer
```

```
Epoch 1/100
11/11 [==============================] - 3s 58ms/step - loss: 566.1277 - mae: 21.8600 - val_loss: 573.8643 - val_mae: 2
1.8478
Epoch 2/100
11/11 [==============================] - 0s 9ms/step - loss: 490.5713 - mae: 20.0330 - val_loss: 480.3946 - val_mae: 19.
6672
Epoch 3/100
11/11 [==============================] - 0s 9ms/step - loss: 398.7499 - mae: 17.6417 - val_loss: 360.7146 - val_mae: 16.
5895
Epoch 4/100
11/11 [==============================] - 0s 8ms/step - loss: 285.3007 - mae: 14.5673 - val_loss: 224.0113 - val_mae: 12.
5901
Epoch 5/100
11/11 [==============================] - 0s 10ms/step - loss: 170.4811 - mae: 10.6310 - val_loss: 106.0184 - val_mae: 8.
3184
Epoch 6/100
11/11 [==============================] - 0s 10ms/step - loss: 93.4149 - mae: 7.4101 - val_loss: 52.7830 - val_mae: 5.794
3
Epoch 7/100
```

EXO 3 : Analyse des performances du modèle

Entrée [12]:

```
from matplotlib import pyplot as plt
loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(1, len(loss) + 1)
plt.plot(epochs, loss, 'y', label='Training loss')
plt.plot(epochs, val_loss, 'r', label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```



Nous effectuons plusieurs epochs(itérations). À chaque epochs, le modèle est entraîné sur les données d'entraînement et les prédictions sont comparées aux labels. La différence entre les prédictions et les valeurs attendues qui quantifie l'erreur du modèle.

La courbe de loss montre comment la perte sur l'ensemble d'entraînement diminue au fur et à mesure que le modèle apprend. Plus le modèle s'améliore, plus la perte diminue donc il faudra au moins 20 epochs pour que le modele stablise. L'objectif de l'entraînement est de minimiser cette perte jusqu'à ce qu'elle atteigne un minimum.

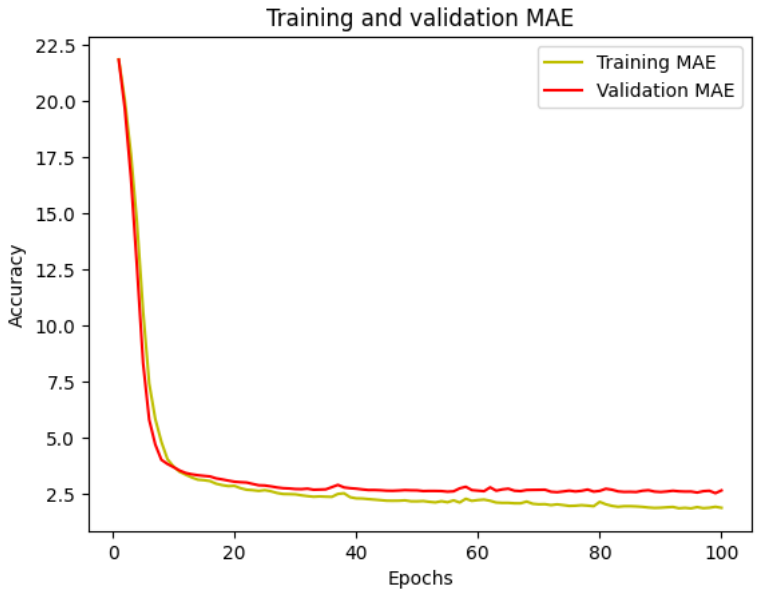
Entrée [13]:

```
print(history.history.keys()) # list all data in history
```

```
dict_keys(['loss', 'mae', 'val_loss', 'val_mae'])
```

Entrée [14]:

```
acc = history.history['mae']
val_acc = history.history['val_mae']
plt.plot(epochs, acc, 'y', label='Training MAE')
plt.plot(epochs, val_acc, 'r', label='Validation MAE')
plt.title('Training and validation MAE')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```



L'axe des y représente l'accuracy (MAE), qui est la mesure de la différence moyenne absolue entre les valeurs prédites par le modèle et les valeurs réelles cas de d'entainement et de validation. L'objectif est de minimiser le MAE afin d'avoir des prédictions plus précises.

Entrée [15]:

```
predictions = model.predict(X_test_scaled[:5])
print("Predicted values are: ", predictions)
print("Real values are: ", y_test[:5])
```

```
1/1 [==============================] - 0s 89ms/step
Predicted values are:  [[17.999893]
 [19.84415 ]
 [20.169617]
 [20.739223]
 [12.704486]]
Real values are:  498    21.2
94     20.6
150    21.5
221    21.7
423    13.4
Name: PRICE, dtype: float64
```

Ici nous comparons les vrai valeur et les valeurs predits, on peut voir que les prediction sont proches de valeur réel.ce qui indique une bonne précision du modèle.

Entrée [16]:

```
mse_neural, mae_neural = model.evaluate(X_test_scaled, y_test)
print('Mean squared error from neural net: ', mse_neural)
print('Mean absolute error from neural net: ', mae_neural)
```

```
4/4 [==============================] - 0s 4ms/step - loss: 14.5347 - mae: 2.8372
Mean squared error from neural net:  14.534743309020996
Mean absolute error from neural net:  2.837233066558838
```

On sait que MSE accorde plus d'importance aux grandes erreurs, tandis que le MAE traite toutes les erreurs de la même manière, qu'elles soient grandes ou petites. d'ou plus le mae est proche de 0 plus la modele est meilleur.

EXO 4 : Comparaison des performances

Régression linéaire

Entrée [17]:

```
from sklearn import linear_model
from sklearn.tree import DecisionTreeRegressor
from sklearn.metrics import mean_squared_error, mean_absolute_error
```

Entrée [18]:

```
lr_model = linear_model.LinearRegression()
lr_model.fit(X_train_scaled, y_train)
y_pred_lr = lr_model.predict(X_test_scaled)
```

Entrée [19]:

```
mse_lr = mean_squared_error(y_test, y_pred_lr)
mae_lr = mean_absolute_error(y_test, y_pred_lr)
print('Mean squared error from linear regression: ', mse_lr)
print('Mean absolute error from linear regression: ', mae_lr)
```

```
Mean squared error from linear regression:  16.49535197593168
Mean absolute error from linear regression:  3.0558941538909594
```

Arbres de décision

Entrée [20]:

```
tree = DecisionTreeRegressor()
tree.fit(X_train_scaled, y_train)
y_pred_tree = tree.predict(X_test_scaled)
```

Entrée [21]:

```
mse_dt = mean_squared_error(y_test, y_pred_tree)
mae_dt = mean_absolute_error(y_test, y_pred_tree)
print('Mean squared error using decision tree: ', mse_dt)
print('Mean absolute error using decision tree: ', mae_dt)
```

```
Mean squared error using decision tree:  19.16480392156863
Mean absolute error using decision tree:  3.1637254901960783
```

Random Forest

Entrée [22]:

```
from sklearn.ensemble import RandomForestRegressor
model = RandomForestRegressor(n_estimators = 30, random_state=30)
model.fit(X_train_scaled, y_train)
y_pred_RF = model.predict(X_test_scaled)
```

Entrée [23]:

```
mse_RF = mean_squared_error(y_test, y_pred_RF)
mae_RF = mean_absolute_error(y_test, y_pred_RF)
print('Mean squared error using Random Forest: ', mse_RF)
print('Mean absolute error Using Random Forest: ', mae_RF)
```

```
Mean squared error using Random Forest:  12.701104793028327
Mean absolute error Using Random Forest:  2.408235294117647
```

COMPARAISON entre les 3 algorithmes.

En comparent les 3 méthode, On a MSE qui varie beaucoup dans les 3 méthode car il est sensible au grosse variation. Comme on sait que si MAE est faible plus le modele est meilleur

- La régression linéaire est relativement simple, mais elle suppose une relation linéaire entre les fratures et les labels et ne peut modéliser que des relations linéaires.lorsque les données présentent des variations importantes. Dans ces situations, la régression linéaire risque de ne pas être adaptée et de fournir des prédictions peu précises. Dans notre probleme, on peut voir qu'il nest pas pire et n'est pas le meilleur non plus d'apres mae et mse.
- Arbres de décision peuvent modéliser des interactions complexes entre les variables. Cependant, les arbres de décision ont tendance à sur-ajuster (overfit) les données si les arbres sont trop profonds, ce qui réduit leur capacité de généralisation. comme il a sur ajuster sur nos données donc on voit qu'il est le pire modele pour notres cas.
- Random Forest combinent les avantages des arbres de décision en réduisant le sur-ajustement et en améliorant la précision prédictive. celle ci nous fourni une meilleur précision sur notres données.