

ML TEMPS RÉEL SUR FLUX DE DONNÉES



ML TEMPS RÉEL SUR FLUX DE DONNÉES



Présenté par :

- **RAHMOUN Hayat**: FST Marrakech, Modélisation Calcul Scientifique pour L'ingénierie Mathématique
- **AGOUMI Achraf**: ENSA Al Hoceima, Ingénierie des données
- **EL BOUGRINI Nassim**: EMI, Génie industriel.
- **NACHOUR Ilham**: ENSA Al Hoceima, Ingénierie des données
- **HAFSI Siham** : ENSA KHOURIBGA , Informatique et ingénierie des données
- **CHAHIDI Khadija**: ENSA Al Hoceima, Ingénierie des données

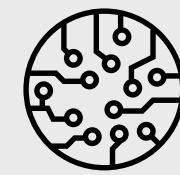
Encadré par :

- **Thierry BERTIN**

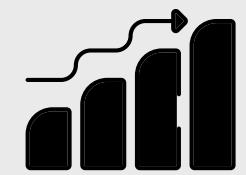
PLAN

- 1 Introduction
- 2 Random Forest
- 3 ARIMA
- 4 XGBoost
- 5 Conclusion

RANDOM FOREST



Features
Selection



Modeling

Random Forest:

Selection des Caracteristiques:



ANALYSE DE CORRÉLATION

Pour bien assurer une performance maximale de notre model il s'avère qu'il est fondamental de tester la corrélation et la dépendance de chaque feature avec notre cible et entre eux au même temps.

```
correlation_matrix = df.corr()  
correlation_with_target = correlation_matrix['close'].abs().sort_values(ascending=False) \  
    .to_frame().style.format({'SalePrice': '{:.03f}'}).background_gradient('Blues')  
  
correlation_with_target
```

ANALYSE DE CORRÉLATION

```
correlation_matrix = df.corr()  
correlation_with_target = correlation_matrix['close'].abs().sort_values(ascending=False) \  
.to_frame().style.format({'SalePrice': '{:.03f}'}) .background_gradient('Blues')
```

```
correlation_with_target
```

close	
close	1.000000
high	0.999544
low	0.999535
open	0.999068
close_lag_1	0.998444
low_lag_1	0.998082
high_lag_1	0.998075
open_lag_1	0.997601
close_lag_2	0.996959
low_lag_2	0.996626
high_lag_2	0.996601
open_lag_2	0.996156
close_lag_3	0.995476
low_lag_3	0.995163
high_lag_3	0.995125
open_lag_3	0.994689
sma_14	0.993942
volume_lag_1	0.385494
volume	0.385317
volume_lag_2	0.384685
volume_lag_3	0.383899
high_low_ratio	0.320624
rsi	0.093713
month	0.028376
close_pct_change	0.021933
volume_change	0.016115

IMPORTANCE DES CARACTÉRISTIQUES

```
x = df.drop(['date', 'close'], axis=1)
y = df['close']

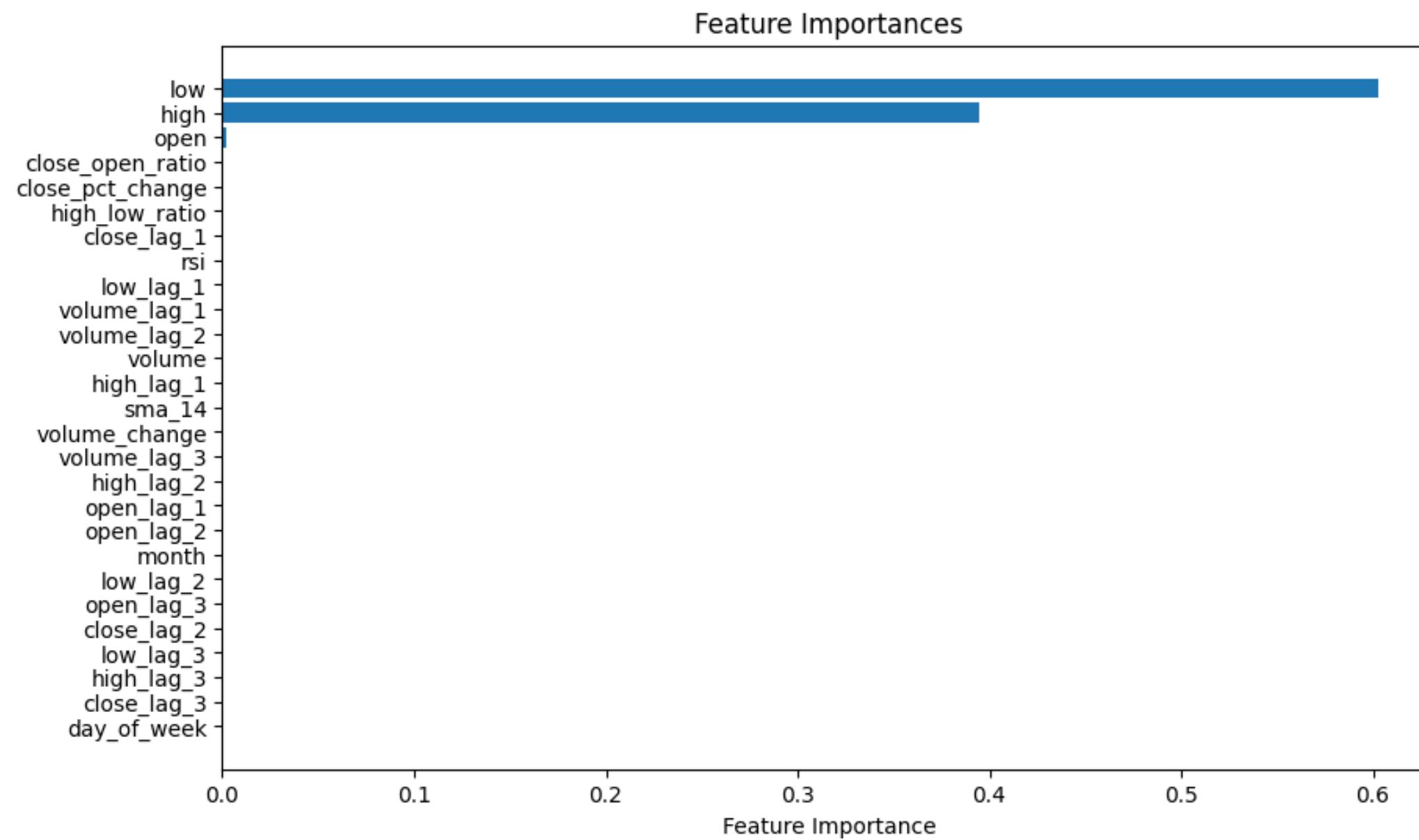
model = RandomForestRegressor()
model.fit(x, y)

feature_importances = model.feature_importances_
sorted_feature_importances = sorted(zip(X.columns, feature_importances),
                                     key=lambda x: x[1], reverse=True)

sorted_idx = np.argsort(feature_importances)

plt.figure(figsize=(10, 6))
plt.barh(range(len(sorted_idx)), feature_importances[sorted_idx], align='center')
plt.yticks(range(len(sorted_idx)), np.array(X.columns)[sorted_idx])
plt.xlabel('Feature Importance')
plt.title('Feature Importances')
plt.show()
```

IMPORTANCE DES CARACTÉRISTIQUES



ÉLIMINATION RÉCURSIVE DES CARACTÉRISTIQUES (RFE)

```
x = df.drop(['date', 'close'], axis=1)
y = df['close']

model = LinearRegression()
rfe = RFE(model, n_features_to_select=8)

rfe.fit(x, y)

selected_features = X.columns[rfe.support_]
print("Selected Features:", selected_features)
```

ÉLIMINATION RÉCURSIVE DES CARACTÉRISTIQUES (RFE)

```
x = df.drop(['date', 'close'], axis=1)
y = df['close']

model = LinearRegression()
rfe = RFE(model, n_features_to_select=8)

rfe.fit(x, y)

selected_features = X.columns[rfe.support_]
print("Selected Features:", selected_features)
```

```
Selected Features: Index(['open', 'high', 'low', 'high_lag_3', 'low_lag_3', 'close_pct_change',
   'high_low_ratio', 'close_open_ratio'],
   dtype='object')
```

LA MULTICOLLINEARITÉ

Sur la base de l'analyse de corrélation et l'importance des caractéristiques, nous choisissons de supprimer la variable '**open**'

Les variables prédictives essentielles: **low, high, close_lag_1, low_lag_3, high_lag_3, close_pct_change, high_low_ratio et close_open_ratio.**

Random Forest:

Modélisation:

ENTRAINEMENT DE MODÈLE:

```
x = df[['low', 'high', 'low_lag_3', 'high_lag_3', 'close_lag_1', 'close_pct_change', 'high_low_ratio', 'close_open_ratio']]  
y = df['close']  
  
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=1)
```

```
model = RandomForestRegressor(n_estimators=100, max_depth=20,  
                             min_samples_split=10, min_samples_leaf=1, random_state=1)
```

```
model.fit(x_train, y_train)
```

```
▼          RandomForestRegressor  
RandomForestRegressor(max_depth=20, min_samples_split=10, random_state=1)
```

```
y_pred = model.predict(x_test)  
  
mse = mean_squared_error(y_test, y_pred)  
r2 = r2_score(y_test, y_pred)  
  
print("Mean Squared Error:", mse)  
print("R-squared:", r2)
```

```
Mean Squared Error: 0.38006153958067057  
R-squared: 0.9996956056102598
```

VALIDATION CROISÉE:

```
cv_scores = cross_val_score(model, X, y, cv=5, scoring='neg_mean_squared_error')

avg_cv_mse = -cv_scores.mean()

print("Average Cross-Validated MSE:", avg_cv_mse)
```

```
Average Cross-Validated MSE: 4.07905302715922
```

ARIMA

Auto**R**egressive
Integrated
Moving
Average

ARIMA

DÉFINITION

L'ARIMA est une méthode statistique largement utilisée pour modéliser et prévoir les séries temporelles.

L'objectif principal de l'ARIMA est de capturer les tendances, les motifs saisonniers et les comportements de variation dans ces séries temporelles, afin de réaliser des prédictions futures ou d'effectuer une analyse approfondie.

L'ARIMA est un outil puissant pour la prévision de séries temporelles, mais il peut nécessiter une certaine expertise pour être correctement utilisé et interprété. De plus, il existe des variantes plus avancées de l'ARIMA, telles que SARIMA (ARIMA saisonnier) et ARIMAX (ARIMA avec variables exogènes), qui permettent de traiter des modèles temporels plus complexes.

ARIMA

LES ÉTAPES

Importations: Les bibliothèques nécessaires sont importées, notamment numpy, pandas, matplotlib, statsmodels, pmdarima, sklearn.metrics, pylab et math

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
plt.style.use('fivethirtyeight')
from pylab import rcParams
rcParams['figure.figsize'] = 10, 6
from statsmodels.tsa.stattools import adfuller
from statsmodels.tsa.seasonal import seasonal_decompose
from statsmodels.tsa.arima.model import ARIMA
from pmdarima.arima import auto_arima

from sklearn.metrics import mean_squared_error, mean_absolute_error
import math
```

ARIMA

LES ÉTAPES

Chargement des données : Les données de prix de clôture de l'action sont chargées à partir du site alphavantage . Les colonnes inutiles sont supprimées pour ne conserver que la colonne "close" (prix de clôture).

```
stock_data = df.drop(["open", "high", "low", "volume"], axis = 1)
```

stock_data	
	close
date	
1999-11-01	96.75
1999-11-02	94.81
1999-11-03	94.37
1999-11-04	91.56
1999-11-05	90.25
...	...
2023-08-14	141.91
2023-08-15	141.87
2023-08-16	140.64
2023-08-17	140.66
2023-08-18	141.41
5988 rows × 1 columns	

ARIMA

LES ÉTAPES

Visualisation des données : Les données de prix de clôture sont tracées dans un graphique pour visualiser la tendance.

```
#plot close price
plt.figure(figsize=(10,6))
plt.grid(True)
plt.xlabel('Date')
plt.ylabel('Close Prices')
plt.plot(stock_data['close'])
plt.title(' closing price')
plt.show()
```

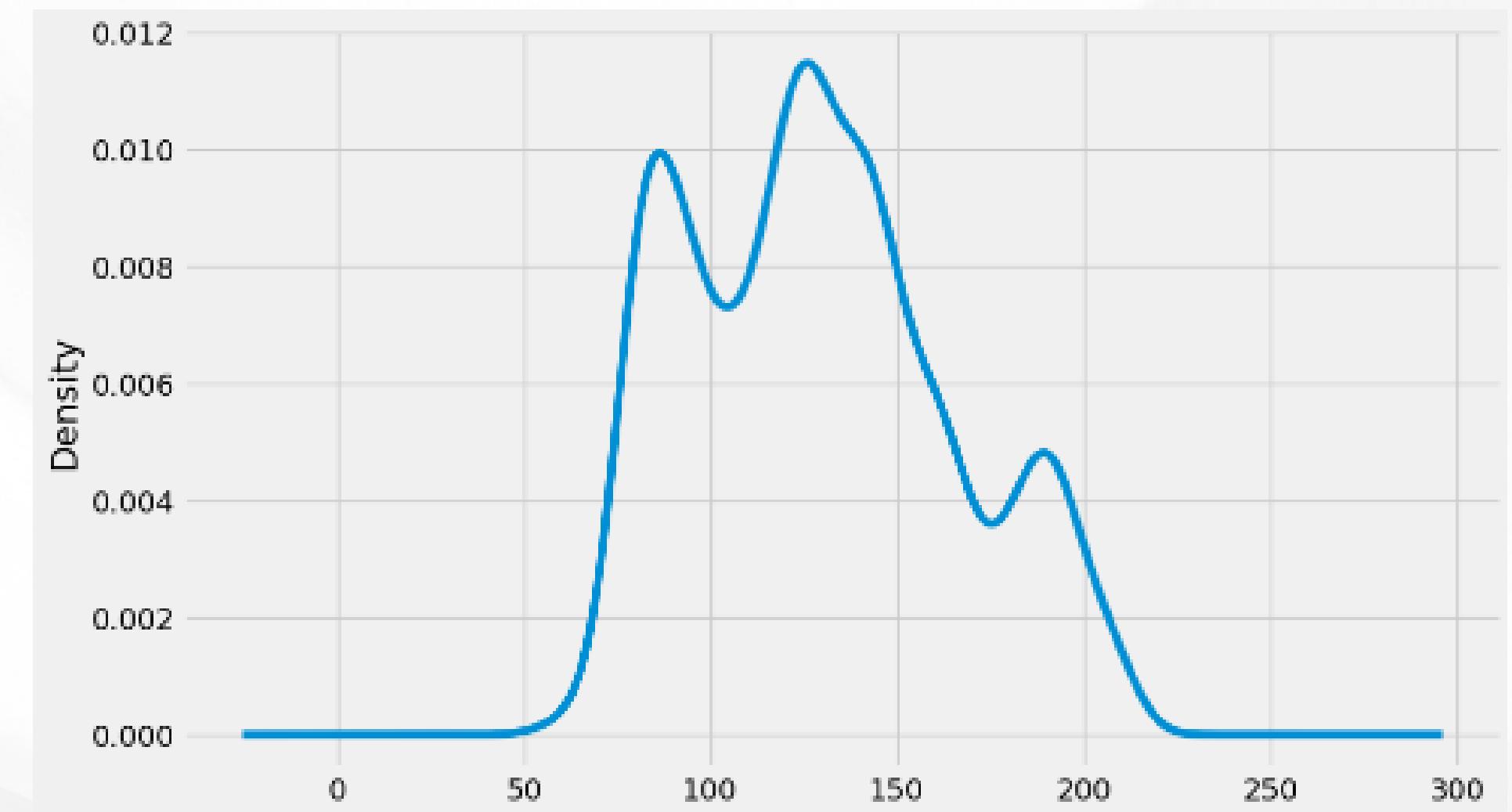


ARIMA

LES ÉTAPES

Visualisation des données : Les données de prix de clôture sont tracées dans un graphique pour visualiser la tendance.

```
#Distribution of the dataset  
df_close.plot(kind='kde')
```



ARIMA

LES ÉTAPES

Vérification de la Stationnarité : La stationnarité des données est vérifiée à l'aide du test Augmented Dickey-Fuller (ADF). Si les données ne sont pas stationnaires, des techniques de transformation peuvent être appliquées.

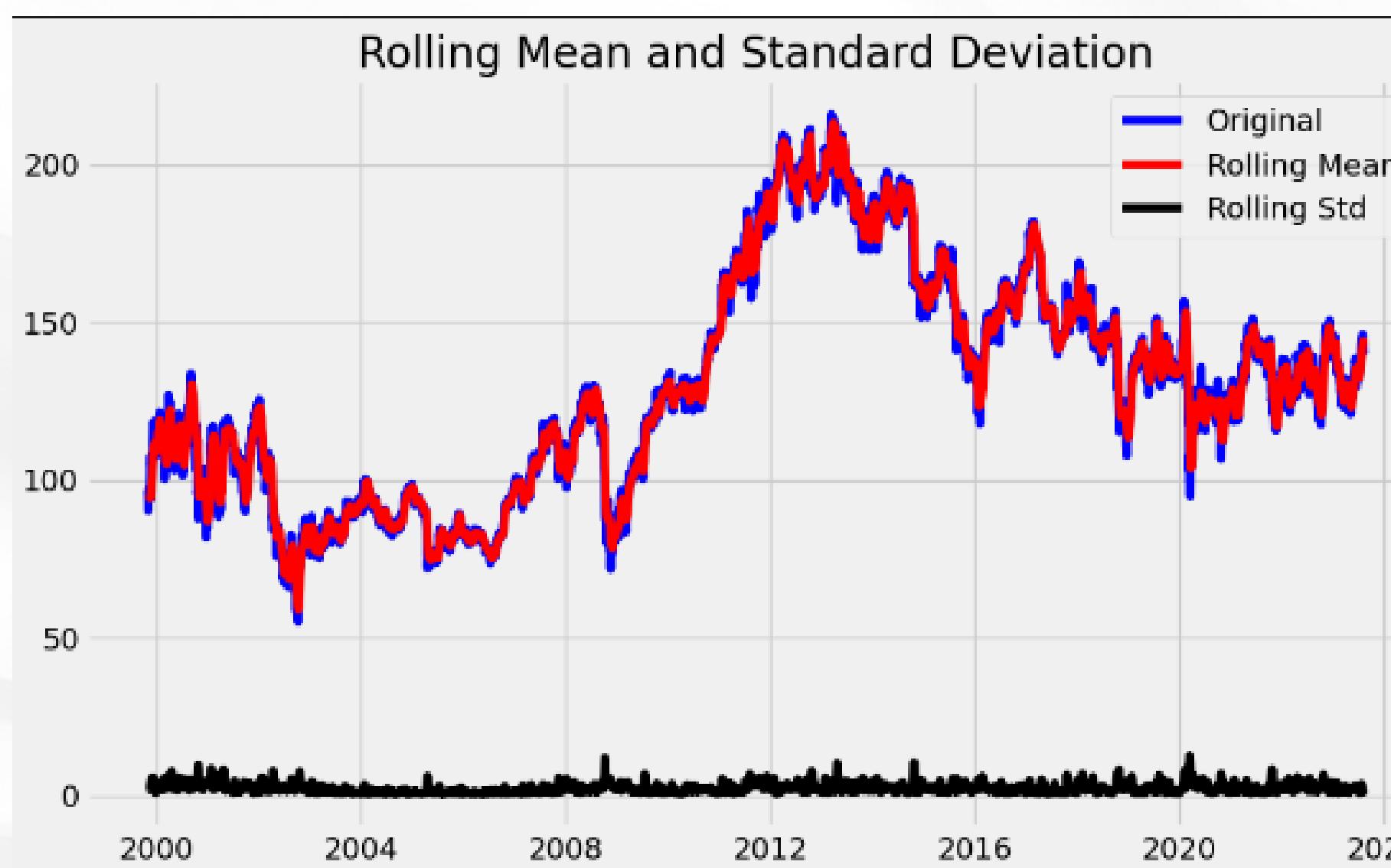
```
#Test for staionarity
def test_stationarity(timeseries):
    #Determining rolling statistics
    rolmean = timeseries.rolling(12).mean()
    rolstd = timeseries.rolling(12).std()
    #Plot rolling statistics:
    plt.plot(timeseries, color='blue',label='Original')
    plt.plot(rolmean, color='red', label='Rolling Mean')
    plt.plot(rolstd, color='black', label = 'Rolling Std')
    plt.legend(loc='best')
    plt.title('Rolling Mean and Standard Deviation')
    plt.show(block=False)

    print("Results of dickey fuller test")
    adft = adfuller(timeseries,autolag='AIC')
    # output for dft will give us without defining what the values are.
    #hence we manually write what values does it explains using a for loop
    output = pd.Series(adft[0:4],index=['Test Statistics','p-value','No. of lags used','Number of observations used'])
    for key,values in adft[4].items():
        output['critical value (%s)'%key] =  values
    print(output)

test_stationarity(df_close)
```

ARIMA: LES ETAPES

Vérification de la Stationnarité : La stationnarité des données est vérifiée à l'aide du test Augmented Dickey-Fuller (ADF). Si les données ne sont pas stationnaires, des techniques de transformation peuvent être appliquées.

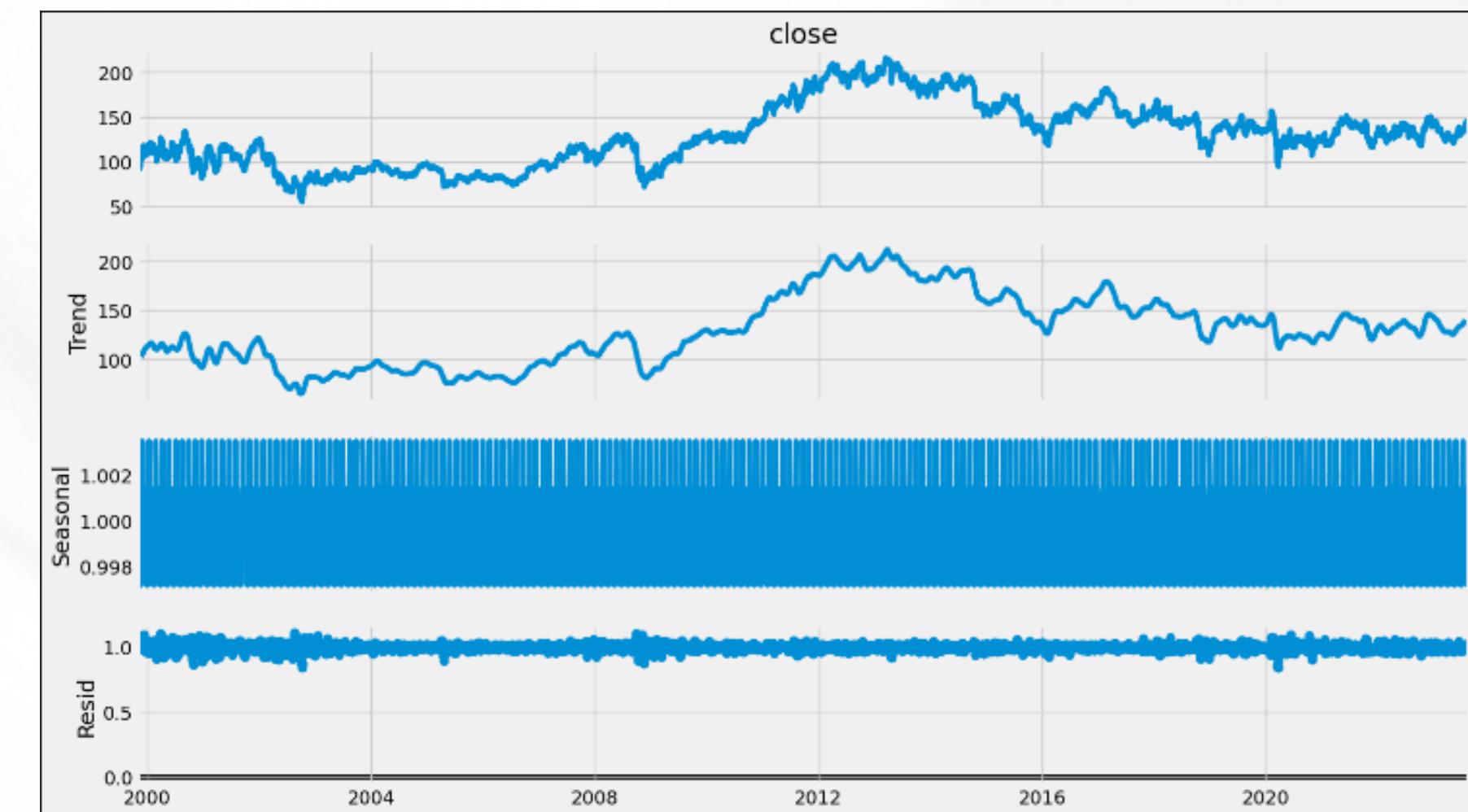


Results of dickey fuller test	
Test Statistics	-1.775288
p-value	0.392798
No. of lags used	25.000000
Number of observations used	5962.000000
critical value (1%)	-3.431447
critical value (5%)	-2.862025
critical value (10%)	-2.567028
dtype:	float64

ARIMA: LES ETAPES

Décomposition saisonnière : Les données sont décomposées en composantes de niveau, de tendance, de saisonnalité et de bruit pour mieux comprendre leur structure.

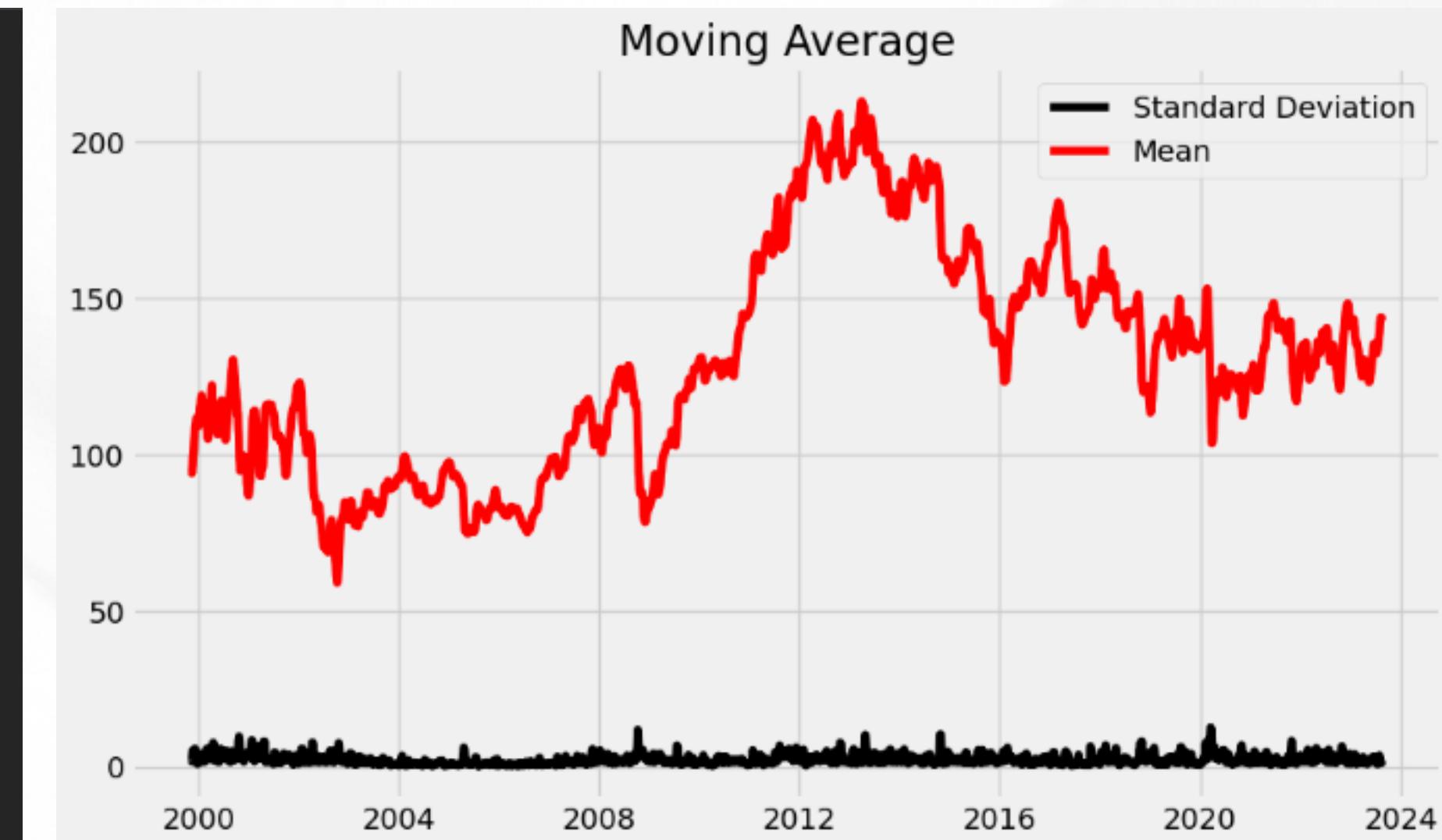
```
#To separate the trend and the seasonality from a time series,  
# we can decompose the series using the following code.  
result = seasonal_decompose(df_close, model='multiplicative', period = 30)  
fig = plt.figure()  
fig = result.plot()  
fig.set_size_inches(16, 9)
```



ARIMA: LES ETAPES

Élimination de la tendance : Une moyenne mobile est calculée pour éliminer la tendance et rendre les données stationnaires.

```
#if not stationary then eliminate trend
#Eliminate trend
from pylab import rcParams
rcParams['figure.figsize'] = 10, 6
#df_log = np.log(df_close)
df_log = df_close
moving_avg = df_log.rolling(12).mean()
std_dev = df_log.rolling(12).std()
plt.legend(loc='best')
plt.title('Moving Average')
plt.plot(std_dev, color ="black", label = "Standard Deviation")
plt.plot(moving_avg, color="red", label = "Mean")
plt.legend()
plt.show()
```



ARIMA: LES ETAPES

Division des données en ensembles d'entraînement et de test : Les données sont divisées en ensembles d'entraînement et de test pour la modélisation.

```
#split data into train and testing set
train_data, test_data = stock_data[3:int(len(stock_data)*0.9)], stock_data[int(len(stock_data)*0.9):]
plt.figure(figsize=(10,6))
plt.grid(True)
plt.xlabel('Dates')
plt.ylabel('Closing Prices')
plt.plot(stock_data['close'], 'green', label='Train data')
plt.plot(test_data['close'], 'blue', label='Test data')
plt.legend()
```

ARIMA: LES ETAPES

Division des données en ensembles d'entraînement et de test : Les données sont divisées en ensembles d'entraînement et de test pour la modélisation.



test_data	
	close
date	
2021-04-05	135.93
2021-04-06	134.22
2021-04-07	134.93
2021-04-08	135.12
2021-04-09	135.73
...	...
2023-08-14	141.91
2023-08-15	141.87
2023-08-16	140.64
2023-08-17	140.66
2023-08-18	141.41

599 rows × 1 columns

ARIMA: LES ETAPES

Modèle Auto ARIMA : La fonction auto_arima est utilisée pour déterminer automatiquement les paramètres optimaux du modèle ARIMA en effectuant des tests de différenciation et en ajustant les ordres p, d et q.

```
model_autoARIMA = auto_arima(train_data, start_p=0, start_q=0,
                               test='adf',      # use adftest to find optimal 'd'
                               max_p=3, max_q=3, # maximum p and q
                               m=1,             # frequency of series
                               d=None,          # let model determine 'd'
                               seasonal=False,   # No Seasonality
                               start_P=0,
                               D=0,
                               trace=True,
                               error_action='ignore',
                               suppress_warnings=True,
                               stepwise=True)
```

```
Performing stepwise search to minimize aic
ARIMA(0,1,0)(0,0,0)[0] intercept : AIC=22727.517, Time=0.07 sec
ARIMA(1,1,0)(0,0,0)[0] intercept : AIC=22724.019, Time=0.15 sec
ARIMA(0,1,1)(0,0,0)[0] intercept : AIC=22724.051, Time=0.17 sec
ARIMA(0,1,0)(0,0,0)[0]         : AIC=22725.598, Time=0.05 sec
ARIMA(2,1,0)(0,0,0)[0] intercept : AIC=22725.975, Time=0.20 sec
ARIMA(1,1,1)(0,0,0)[0] intercept : AIC=22725.926, Time=0.26 sec
ARIMA(2,1,1)(0,0,0)[0] intercept : AIC=22727.980, Time=0.29 sec
ARIMA(1,1,0)(0,0,0)[0]         : AIC=22722.105, Time=0.06 sec
ARIMA(2,1,0)(0,0,0)[0]         : AIC=22724.061, Time=0.11 sec
ARIMA(1,1,1)(0,0,0)[0]         : AIC=22724.012, Time=0.12 sec
ARIMA(0,1,1)(0,0,0)[0]         : AIC=22722.138, Time=0.07 sec
ARIMA(2,1,1)(0,0,0)[0]         : AIC=22726.066, Time=0.15 sec
```

```
Best model: ARIMA(1,1,0)(0,0,0)[0]
Total fit time: 1.741 seconds
```

ARIMA: LES ETAPES

Prédictions du modèle : Les prédictions du modèle sont générées en utilisant la méthode ARIMA avec les paramètres optimaux trouvés par auto_arima.

```
for time_point in range(N_test_observations):
    model = ARIMA(history, order=(1,1,0))
    model_fit = model.fit()
    output = model_fit.forecast()
    yhat = output[0]
    model_predictions.append(yhat)
    true_test_value = test_data[time_point]
    history.append(true_test_value)
```

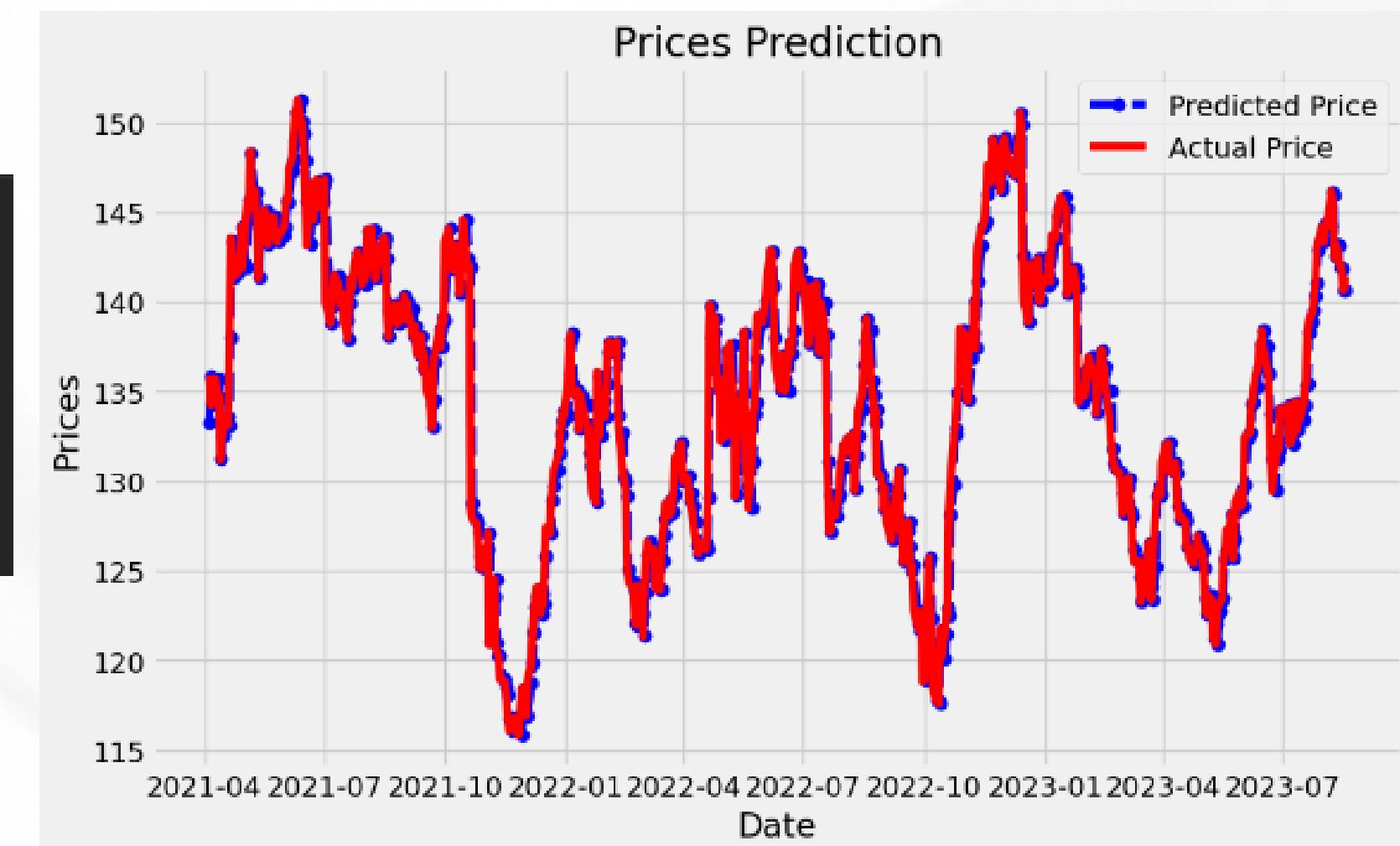
model_predictions

```
[133.23095747490808,
 135.84381706331337,
 134.2749319196386,
 134.90715494868797,
 135.11388787264025,
 135.71038017819376,
 134.62670285922067,
 131.28916232132465,
 132.58327328108192,
 132.5816112773876,
```

ARIMA: LES ETAPES

Visualisation des résultats : Les données réelles et les prévisions sont tracées dans un graphique pour comparer les performances du modèle.

```
test_set_range = stock_data[int(len(stock_data)*0.9):].index
plt.plot(test_set_range, model_predictions, color='blue', marker='o', linestyle='dashed',label='Predicted Price')
plt.plot(test_set_range, test_data, color='red', label='Actual Price')
plt.title('Prices Prediction')
plt.xlabel('Date')
plt.ylabel('Prices')
# plt.xticks(np.arange(881,1259,50), stock_data.Date[881:1259:50])
plt.legend()
plt.show()
```



ARIMA: LES ETAPES

Évaluation du modèle : Les performances du modèle sont évaluées en calculant l'erreur quadratique moyenne (MSE), l'erreur absolue moyenne (MAE), la racine de l'erreur quadratique moyenne (RMSE) et l'erreur moyenne absolue en pourcentage (MAPE).

```
# report performance
mse = mean_squared_error(test_data, model_predictions)
print('MSE: '+str(mse))
mae = mean_absolute_error(test_data, model_predictions)
print('MAE: '+str(mae))
rmse = math.sqrt(mean_squared_error(test_data, model_predictions))
print('RMSE: '+str(rmse))
mape = np.mean(np.abs(model_predictions - test_data)/np.abs(test_data))
print('MAPE: '+str(mape))
```

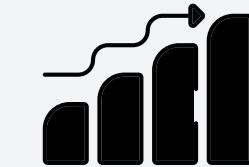
```
MSE: 3.3731382809271873
MAE: 1.3060508298377922
RMSE: 1.8366105414396345
MAPE: 0.009753250825299903
```

XGBOOST

eXtreme
Gradient
Boosting



Features
Selection



Modeling

FEATURE SELECTION

ANALYSE DE CORRÉLATION

Afin d'assurer une performance maximale de notre modèle, il est essentiel d'évaluer à la fois la relation et l'interconnexion de chaque caractéristique avec notre variable cible, ainsi qu'entre les différentes caractéristiques.

```
correlation_matrix = df.corr()
correlation_with_target = correlation_matrix["close"].abs().sort_values(ascending=False)\n.to_frame().style.background_gradient('Blues')
correlation_with_target
```

ANALYSE DE CORRÉLATION

```
correlation_matrix = df.corr()  
correlation_with_target = correlation_matrix["close"].abs().sort_values(ascending=False)\  
.to_frame().style.background_gradient('Blues')  
correlation_with_target
```

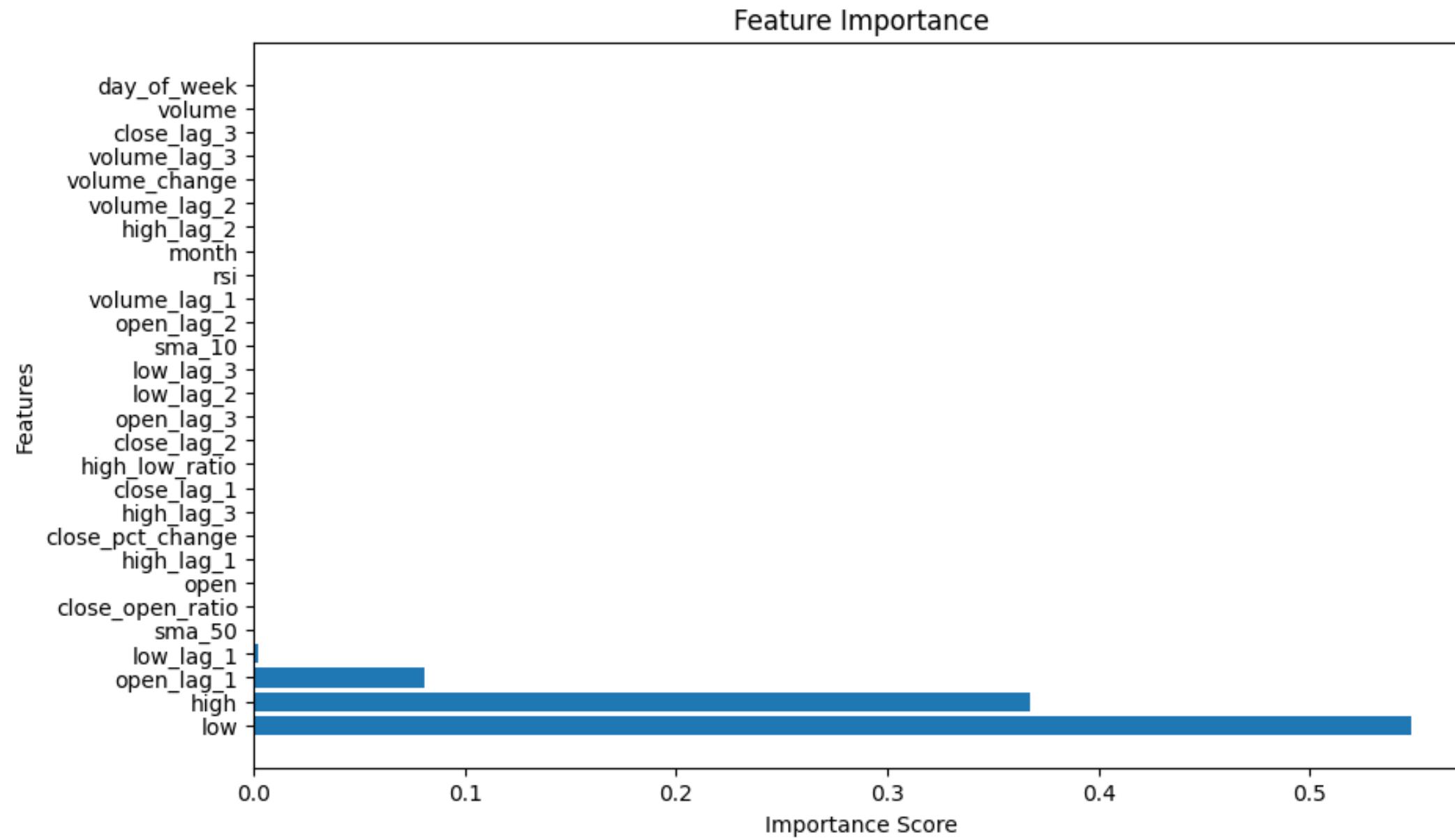
close	
close	1.000000
high	0.999544
low	0.999535
open_lag_1	0.999434
open	0.999066
high_lag_1	0.999000
low_lag_1	0.998903
close_lag_1	0.998444
open_lag_2	0.997987
high_lag_2	0.997543

volume	0.386643
volume_lag_1	0.382174
volume_lag_2	0.379336
volume_lag_3	0.377054
high_low_ratio	0.321943
rsi	0.085853
month	0.031117
close_pct_change	0.024944
volume_change	0.017805
close_open_ratio	0.014761
day_of_week	0.000905

IMPORTANCE DES CARACTÉRISTIQUES

```
x=df.drop(columns=['date','close'])  
y=df['close']  
  
xgb_model = xgb.XGBRegressor(  
    n_estimators=500,  
    max_depth=3,  
    learning_rate=0.5,  
    objective='reg:squarederror',  
    random_state=42  
)  
  
xgb_model.fit(x,y)  
  
feature_importance = xgb_model.feature_importances_  
feature_importance_dict = {feature: importance for feature, importance in zip(x.columns, feature_importance)}  
sorted_features = sorted(feature_importance_dict.items(), key=lambda x: x[1], reverse=True)  
  
# Print feature importance scores  
for feature, importance in sorted_features:  
    print(f"{feature}: {importance}")  
  
# Plot feature importance  
plt.figure(figsize=(10, 6))  
plt.barh([x[0] for x in sorted_features], [x[1] for x in sorted_features])  
plt.xlabel('Importance Score')  
plt.ylabel('Features')  
plt.title('Feature Importance')  
plt.show()
```

IMPORTANCE DES CARACTÉRISTIQUES



MODELING

ENTRAINEMENT DE MODÈLE:

```
x=df[['low','high','open_lag_1',
       'low_lag_1','sma_50','close_open_ratio',
       'open','high_lag_1','close_pct_change']]  
  
y=df['close']  
  
xgb_model = xgb.XGBRegressor(  
    n_estimators=500,  
    max_depth=3,  
    learning_rate=0.5,  
    objective='reg:squarederror',  
    random_state=42  
)  
  
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=42)  
xgb_model.fit(x_train, y_train)  
✓ 1.5s
```

XGBRegressor
XGBRegressor(base_score=0.5, booster='gbtree', callbacks=None,
 colsample_bylevel=1, colsample_bynode=1, colsample_bytree=1,
 early_stopping_rounds=None, enable_categorical=False,
 eval_metric=None, feature_types=None, gamma=0, gpu_id=-1,
 grow_policy='depthwise', importance_type=None,
 interaction_constraints='', learning_rate=0.5, max_bin=256,
 max_cat_threshold=64, max_cat_to_onehot=4, max_delta_step=0,
 max_depth=3, max_leaves=0, min_child_weight=1, missing=nan,
 monotone_constraints='()', n_estimators=500, n_jobs=0,
 num_parallel_tree=1, predictor='auto', random_state=42, ...)

ÉVALUATION DE MODÈLE:

```
y_pred = xgb_model.predict(x_test)
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print("Mean Squared Error:", mse)
print("R-squared:", r2)
```

✓ 0.0s

```
Mean Squared Error: 0.3139886254415497
R-squared: 0.9997445146390314
```

VALIDATION CROISÉE:

```
cv_scores = cross_val_score(xgb_model, X, y, cv=5, scoring='neg_mean_squared_error')

avg_cv_mse = -cv_scores.mean()

print("Average Cross-Validated MSE:", avg_cv_mse)

✓ 12.2s
```

Average Cross-Validated MSE: 4.358072244484283



**MERCI POUR VOTRE
ATTENTION**