

Entrée [1]:

```
import pandas as pd
import numpy as np
%matplotlib inline
import matplotlib.pyplot as plt
import matplotlib
from sklearn.preprocessing import MinMaxScaler
from keras.layers import LSTM, Dense, Dropout
from sklearn.model_selection import TimeSeriesSplit
from sklearn.metrics import mean_squared_error, r2_score
import matplotlib.dates as mandates
from sklearn.preprocessing import MinMaxScaler
from sklearn import linear_model
from keras.models import Sequential
from keras.layers import Dense
import keras.backend as K
from keras.callbacks import EarlyStopping
from keras.optimizers import Adam
from keras.models import load_model
from keras.layers import LSTM
from keras.utils.vis_utils import plot_model

import warnings

# Ignorer tous les avertissements
warnings.filterwarnings("ignore")
```

```
C:\Users\bouch\anaconda3\lib\site-packages\numpy\_distributor_init.py:30:
UserWarning: loaded more than 1 DLL from .libs:
C:\Users\bouch\anaconda3\lib\site-packages\numpy\.libs\libopenblas.GK7GX5K
EQ4F6UY03P26ULGBQYHGQ07J4.gfortran-win_amd64.dll
C:\Users\bouch\anaconda3\lib\site-packages\numpy\.libs\libopenblas.XWYDX2I
KJW2NMTWSFYNGFUWKQU3LYTCZ.gfortran-win_amd64.dll
    warnings.warn("loaded more than 1 DLL from .libs:")
<frozen importlib._bootstrap>:228: RuntimeWarning: scipy._lib.messagestream.MessageStream size changed, may indicate binary incompatibility. Expected 56 from C header, got 64 from PyObject
```

```
Entrée [2]: # Charger le fichier Excel
excel_file_path = 'Historiques_cours_boursiers.xlsx'
xls = pd.ExcelFile('Historiques_cours_boursiers.xlsx')

# Lire chaque onglet dans une boucle
dataframes = {} # Utiliser un dictionnaire pour stocker les DataFrames de

for sheet_name in xls.sheet_names:
    # Lire les données de l'onglet en cours
    df = pd.read_excel(excel_file_path, sheet_name)

    # Stocker le DataFrame dans le dictionnaire
    dataframes[sheet_name] = df

# Accéder à un DataFrame spécifique (par exemple, le premier onglet)
premier_onglet = dataframes[xls.sheet_names[0]]
```

Entrée [3]: #6 choix

```
list_choice= {'MSFT_df':dataframes[xls.sheet_names[9]],
              'AMZN_df' :dataframes[xls.sheet_names[12]],
              'TSLA_df':dataframes[xls.sheet_names[4]],
              'RMSPA_df':dataframes[xls.sheet_names[3]],
              'GE_df':dataframes[xls.sheet_names[2]],
              'TTE_df':dataframes[xls.sheet_names[1]]}
```

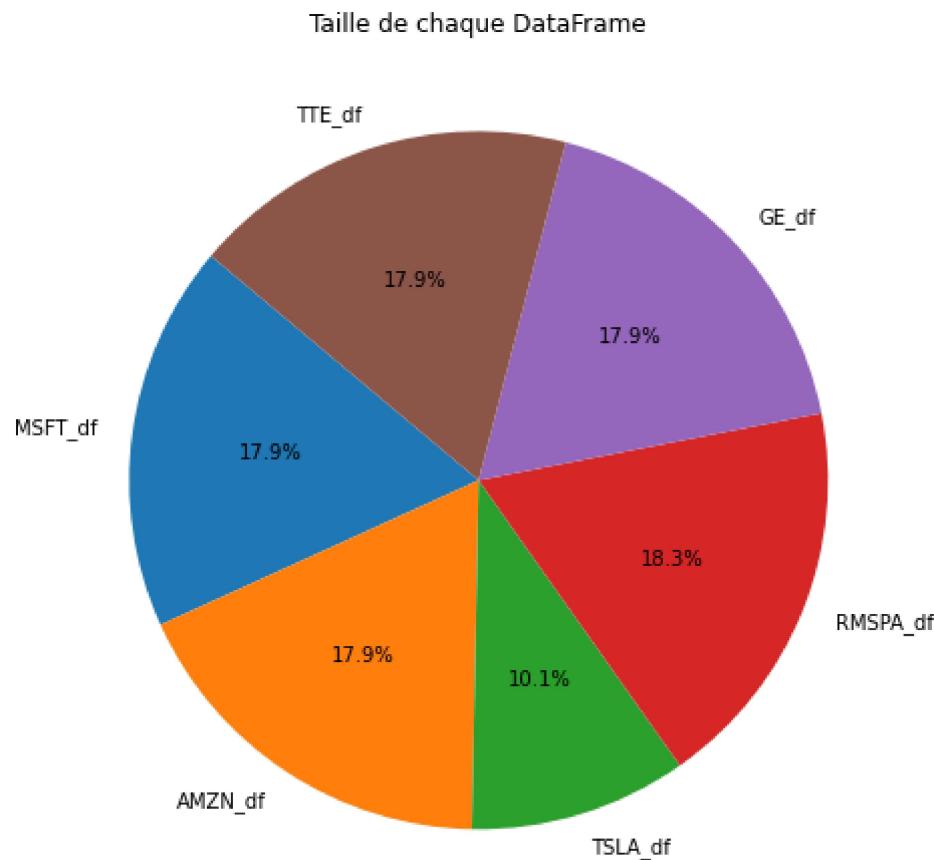
```
Entrée [4]: # Récupérer les noms des DataFrames et leurs tailles
df_sizes = {name: df.shape[0] for name, df in list_choice.items()}

# Afficher la taille de chaque DataFrame en texte
for name, size in df_sizes.items():
    print(f"La taille de {name} est {size} lignes.")
```

```
La taille de MSFT_df est 6031 lignes.
La taille de AMZN_df est 6031 lignes.
La taille de TSLA_df est 3394 lignes.
La taille de RMSPA_df est 6165 lignes.
La taille de GE_df est 6031 lignes.
La taille de TTE_df est 6031 lignes.
```

```
Entrée [5]: df_names = list(list_choice.keys())
df_sizes = [df.shape[0] for df in list_choice.values()]

# Créer un diagramme circulaire
plt.figure(figsize=(8, 8))
plt.pie(df_sizes, labels=df_names, autopct='%1.1f%%', startangle=140)
plt.title('Taille de chaque DataFrame')
plt.show()
```



```
Entrée [ ]:
```

1. Data preparation

```
Entrée [6]: for key, df in list_choice.items():

    df['Day'] = df['Date'].dt.day
    df['Month'] = df['Date'].dt.month
    df['Year'] = df['Date'].dt.year
```

2. Exploratory Data Analysis

```
Entrée [7]: import matplotlib.pyplot as plt

# Nombre total de sous-plots
num_plots = len(list_choice)

# Définir le nombre de lignes et de colonnes dans la grille
num_rows = 3
num_cols = 2

# Créer la grille de sous-plots
fig, axes = plt.subplots(num_rows, num_cols, figsize=(15, 10))

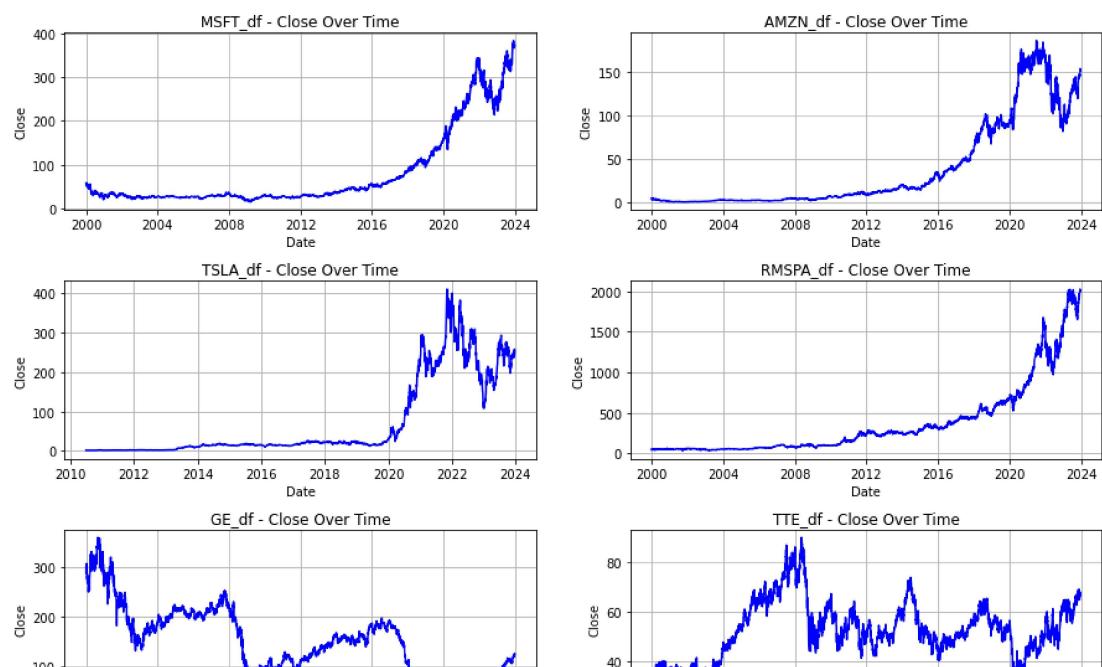
# Ajuster l'espacement entre les sous-plots
plt.subplots_adjust(hspace=0.4)

# Parcourir chaque DataFrame dans la liste
for idx, (df_name, df) in enumerate(list_choice.items()):
    # Convertir la colonne 'Date' en un tableau NumPy
    dates = df['Date'].to_numpy()

    # Convertir la colonne 'Adj Close' en un tableau NumPy
    close = df['Close'].to_numpy()
    opens=df['Open'].to_numpy()
    # Calculer les indices de ligne et de colonne
    row_idx = idx // num_cols
    col_idx = idx % num_cols

    # Tracer le graphique pour le DataFrame actuel dans le sous-plot correspondant
    axes[row_idx, col_idx].plot(dates, close, label='Close', color='blue')
    # axes[row_idx, col_idx].plot(dates, opens, label='Open', color='blue')
    axes[row_idx, col_idx].set_title(f'{df_name} - Close Over Time')
    axes[row_idx, col_idx].set_xlabel('Date')
    axes[row_idx, col_idx].set_ylabel('Close')
    axes[row_idx, col_idx].grid(True)

# Afficher la grille de sous-plots
plt.show()
```



Entrée []:

```
Entrée [8]: import matplotlib.pyplot as plt

# Nombre total de sous-plots
num_plots = len(list_choice)

# Définir le nombre de lignes et de colonnes dans la grille
num_rows = 3
num_cols = 2

# Créer la grille de sous-plots
fig, axes = plt.subplots(num_rows, num_cols, figsize=(15, 10))

# Ajuster l'espacement entre les sous-plots
plt.subplots_adjust(hspace=0.4)

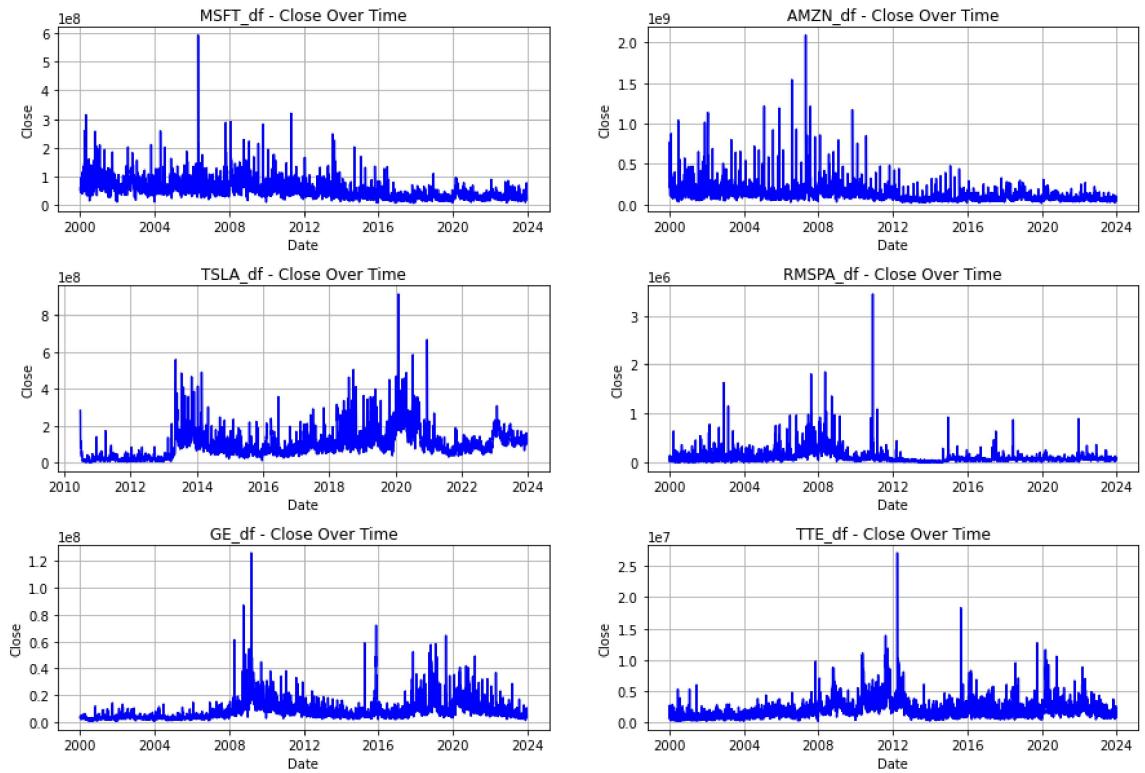
# Parcourir chaque DataFrame dans la liste
for idx, (df_name, df) in enumerate(list_choice.items()):
    # Convertir la colonne 'Date' en un tableau NumPy
    dates = df['Date'].to_numpy()

    # Convertir la colonne 'Adj Close' en un tableau NumPy
    volume = df['Volume'].to_numpy()

    # Calculer les indices de ligne et de colonne
    row_idx = idx // num_cols
    col_idx = idx % num_cols

    # Tracer le graphique pour le DataFrame actuel dans le sous-plot correspondant
    axes[row_idx, col_idx].plot(dates, volume, label='Volume', color='blue')
    # axes[row_idx, col_idx].plot(dates, opens, label='Open', color='blue')
    axes[row_idx, col_idx].set_title(f'{df_name} - Close Over Time')
    axes[row_idx, col_idx].set_xlabel('Date')
    axes[row_idx, col_idx].set_ylabel('Close')
    axes[row_idx, col_idx].grid(True)

# Afficher la grille de sous-plots
plt.show()
```



var à expliquer

Entrée [9]:

```
def calculate_trend(close_prices):
    # Initialize the target column with neutral values
    target = pd.Series(index=close_prices.index, dtype=int).fillna(0)

    # Calculate the percentage change between the current closing price and the
    # previous 5 closing prices
    pct_change = 100 * (close_prices.shift(-5) - close_prices) / close_prices

    # Determine the trend
    target[pct_change > 2] = 1 # Bullish trend
    target[pct_change < -2] = -1 # Bearish trend

    # Return the target series
    return target.iloc[:-5].astype(int)
```

Entrée [10]:

```

for key, df in list_choice.items():
    # Extraire la colonne 'Close' comme une série
    close_prices = df['Close']

    # Utiliser la fonction calculate_trend
    trend = calculate_trend(close_prices)

    # Ajouter la colonne 'Trend' au DataFrame
    df['Trend'] = trend

    # Afficher le DataFrame avec la colonne 'Trend'
    # print(f"\nDataFrame {key} avec la colonne 'Trend':")
    # print(df)

```

Entrée [11]: list_choice['MSFT_df'].head(15)

Out[11]:

	Date	Open	High	Low	Close	Adj Close	Volume	Day	Month	Year
0	2000-01-03	58.687500	59.31250	56.00000	58.28125	36.132240	53228400	3	1	2000
1	2000-01-04	56.781250	58.56250	56.12500	56.31250	34.911697	54119000	4	1	2000
2	2000-01-05	55.562500	58.18750	54.68750	56.90625	35.279823	64059600	5	1	2000
3	2000-01-06	56.093750	56.93750	54.18750	55.00000	34.098007	54976600	6	1	2000
4	2000-01-07	54.312500	56.12500	53.65625	55.71875	34.543606	62013600	7	1	2000
5	2000-01-10	56.718750	56.84375	55.68750	56.12500	34.795467	44963600	10	1	2000
6	2000-01-11	55.750000	57.12500	54.34375	54.68750	33.904270	46743600	11	1	2000
7	2000-01-12	54.250000	54.43750	52.21875	52.90625	32.799969	66532400	12	1	2000
8	2000-01-13	52.187500	54.31250	50.75000	53.90625	33.419930	83144000	13	1	2000
9	2000-01-14	53.593750	56.96875	52.87500	56.12500	34.795467	73416400	14	1	2000
10	2000-01-18	55.906250	58.25000	55.87500	57.65625	35.744789	81483600	18	1	2000
11	2000-01-19	55.250000	55.75000	53.00000	53.50000	33.168060	97568200	19	1	2000
12	2000-01-20	53.531250	54.84375	52.93750	53.00000	32.858070	56349800	20	1	2000
13	2000-01-21	53.500000	53.62500	51.62500	51.87500	32.160625	68416200	21	1	2000
14	2000-01-24	51.898438	52.84375	50.40625	50.62500	31.385658	63597600	24	1	2000



```
Entrée [12]: import matplotlib.pyplot as plt

# Définir le nombre de colonnes dans la grille
num_cols = 3

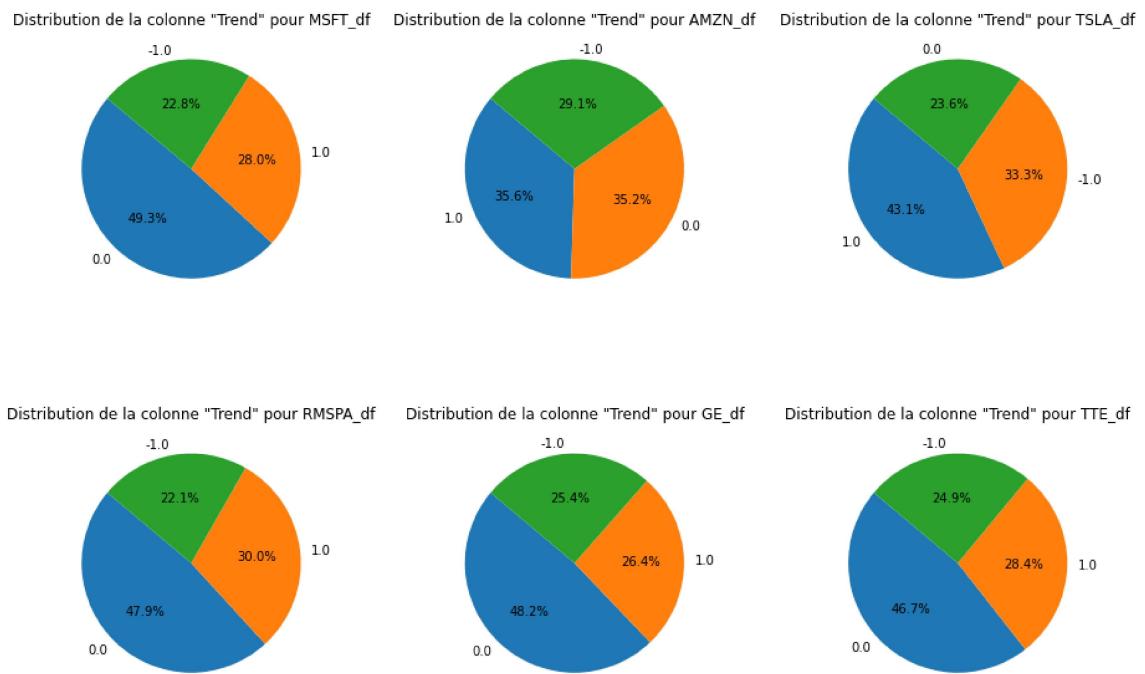
# Calculer le nombre de lignes nécessaire en fonction de la Longueur de Lis
num_rows = -(len(list_choice) // num_cols) # Utilisation de la division euclidienne

# Créer une grille de sous-graphiques
fig, axs = plt.subplots(num_rows, num_cols, figsize=(15, 5*num_rows))

# Ajuster l'espacement entre les sous-graphiques
plt.subplots_adjust(wspace=0.4, hspace=0.4)

# Parcourir chaque dataframe dans List_choice
for i, (key, df) in enumerate(list_choice.items()):
    # Calculer les indices de la ligne et de la colonne pour placer le pie chart
    row_idx = i // num_cols
    col_idx = i % num_cols
    # trend_counts = df['Trend'].to_numpy()
    # Sélectionner le sous-graphique correspondant dans la grille
    ax = axs[row_idx, col_idx] if num_rows > 1 else axs[col_idx]
    trend_counts = df['Trend'].value_counts()
    # Créer le pie chart pour la distribution de la colonne 'Trend'
    ax.pie(trend_counts, labels=trend_counts.index, autopct='%1.1f%%')
    ax.set_title(f'Distribution de la colonne "Trend" pour {key}')

# Afficher les sous-graphiques
plt.show()
```



Variables explicatives

Entrée [13]: # Définir les périodes N

```

N_14 = 14
N_21 = 21
N_ATR = 14
N_CMF_21 = 21
N_CMF_28 = 28

# Seuil pour CMF
seuil_haussier = 0.25
seuil_baissier = -0.25

# Fonction pour calculer RSI
def calculate_rsi(data, period):
    delta = data.diff()
    gain = delta.where(delta > 0, 0)
    loss = -delta.where(delta < 0, 0)
    avg_gain = gain.rolling(window=period).mean()
    avg_loss = loss.rolling(window=period).mean()
    rs = avg_gain / avg_loss
    rsi_column = 100 - (100 / (1 + rs))
    return rsi_column

# Fonction pour calculer RSI trend
def calculate_rsi_trend(rsi_column, overbought=70, oversold=30):
    return np.where((rsi_column > overbought), 1, np.where((rsi_column < oversold), -1, 0))

# Fonction pour calculer CMF
def calculate_cmf(data, period):
    mf_multiplier = ((data['Close'] - data['Low']) - (data['High'] - data['Close'])) / (data['High'] - data['Low'])
    mf_volume = mf_multiplier * data['Volume']
    cmf_column = mf_volume.rolling(window=period).sum() / data['Volume'].rolling(window=period).sum()
    return cmf_column

# Parcourir chaque dataframe dans list_choice
for key, df in list_choice.items():
    # Calculer MACD
    df['MACD_Line'] = df['Close'].ewm(span=12).mean() - df['Close'].ewm(span=26).mean()
    df['MACD_Signal'] = df['MACD_Line'].ewm(span=9).mean()
    df['MACD_Histogram'] = df['MACD_Line'] - df['MACD_Signal']
    df['MACD_trend'] = np.where((df['MACD_Line'] > 0) | (df['MACD_Histogram'] > 0), 1, np.where((df['MACD_Line'] < 0) | (df['MACD_Histogram'] < 0), -1, 0))

    # Calculer RSI et les tendances pour chaque période
    df['RSI_14'] = calculate_rsi(df['Close'], N_14)
    df['RSI_trend_14'] = calculate_rsi_trend(df['RSI_14'])

    df['RSI_21'] = calculate_rsi(df['Close'], N_21)
    df['RSI_trend_21'] = calculate_rsi_trend(df['RSI_21'])

    # Calculer ATR
    tr = pd.DataFrame(index=df.index)
    tr['HL'] = df['High'] - df['Low']
    tr['HC'] = abs(df['High'] - df['Close'].shift())
    tr['LC'] = abs(df['Low'] - df['Close'].shift())
    tr['TrueRange'] = tr[['HL', 'HC', 'LC']].max(axis=1)
    df['ATR'] = tr['TrueRange'].rolling(window=N_ATR).mean()

    # Calculer CMF
    df['CMF_21'] = calculate_cmf(df, N_CMF_21)
    df['CMF_28'] = calculate_cmf(df, N_CMF_28)

```

```
# Ajouter La colonne cmf_trend
df['cmf_trend_21'] = np.where(df['CMF_21'] > seuil_haussier, 1,
                                np.where(df['CMF_21'] < seuil_baissier,
                                         -1, 0))
df['cmf_trend_28'] = np.where(df['CMF_28'] > seuil_haussier, 1,
                                np.where(df['CMF_28'] < seuil_baissier,
                                         -1, 0))

# Calculer Daily Range
df['Daily_Range'] = df['High'] - df['Low']

# Calculer Gap
df['Gap'] = df['Open'] - df['Close'].shift(1)

df.dropna(inplace=True)
# Afficher le DataFrame avec les nouvelles colonnes
# print(f"\nDataFrame {key} avec les nouvelles colonnes:")
# print(df)
```

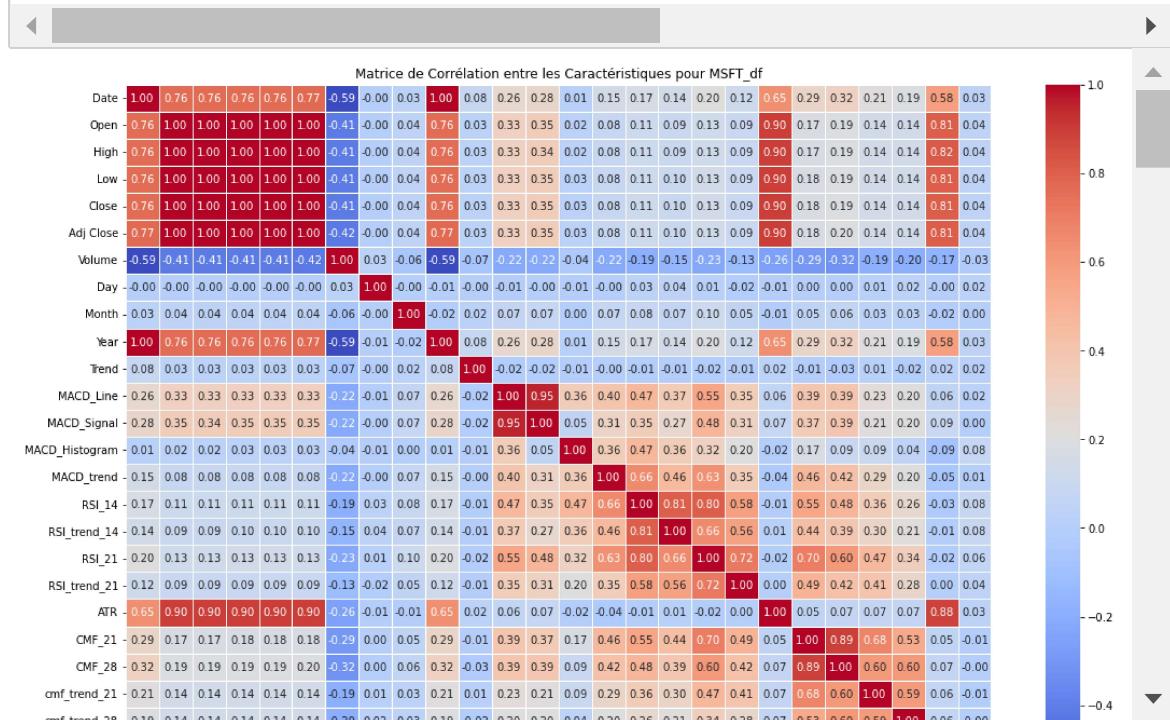
Entrée [14]: list_choice['MSFT_df'].columns

Out[14]: Index(['Date', 'Open', 'High', 'Low', 'Close', 'Adj Close', 'Volume', 'Day',
 'Month', 'Year', 'Trend', 'MACD_Line', 'MACD_Signal', 'MACD_Histogram',
 'MACD_trend', 'RSI_14', 'RSI_trend_14', 'RSI_21', 'RSI_trend_21',
 'ATR',
 'CMF_21', 'CMF_28', 'cmf_trend_21', 'cmf_trend_28', 'Daily_Range',
 'Gap'],
 dtype='object')

```
Entrée [15]: import seaborn as sns
import matplotlib.pyplot as plt

for key, df in list_choice.items():
    # Calculer la matrice de corrélation
    correlation_matrix = df.corr()

    # Afficher le heatmap de la matrice de corrélation
    plt.figure(figsize=(18, 12))
    sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt=".2f",
    plt.title(f"Matrice de Corrélation entre les Caractéristiques pour {key}")
    plt.show()
```



Entrée []:

Entrée [16]:

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.feature_selection import RFE
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
import pandas as pd

# Supposons que vos données soient dans un DataFrame 'df'
# Assurez-vous que 'df' contient toutes les colonnes nécessaires, sauf 'Trend'
for key, df in list_choice.items():
    # Définir x comme le DataFrame sans 'Trend' et 'Date'
    x = df.drop(['Trend', 'Date', 'CMF_21', 'Open', 'High', 'cmf_trend_21', 'Close'])

    # Définir y comme la colonne 'Trend'
    y = df['Trend']

    # Créer un objet StandardScaler
    scaler = StandardScaler()

    # Normaliser les données x
    x_scaled = scaler.fit_transform(x)

    # Exemple d'utilisation de RFE
    n_features_to_select = 9
    rf_model = RandomForestClassifier(n_estimators=100, random_state=42)
    rfe = RFE(estimator=rf_model, n_features_to_select=n_features_to_select)
    rfe.fit(x_scaled, y)

    # Obtenez les caractéristiques sélectionnées et leur classement
    selected_features = x.columns[rfe.support_]
    ranking = rfe.ranking_

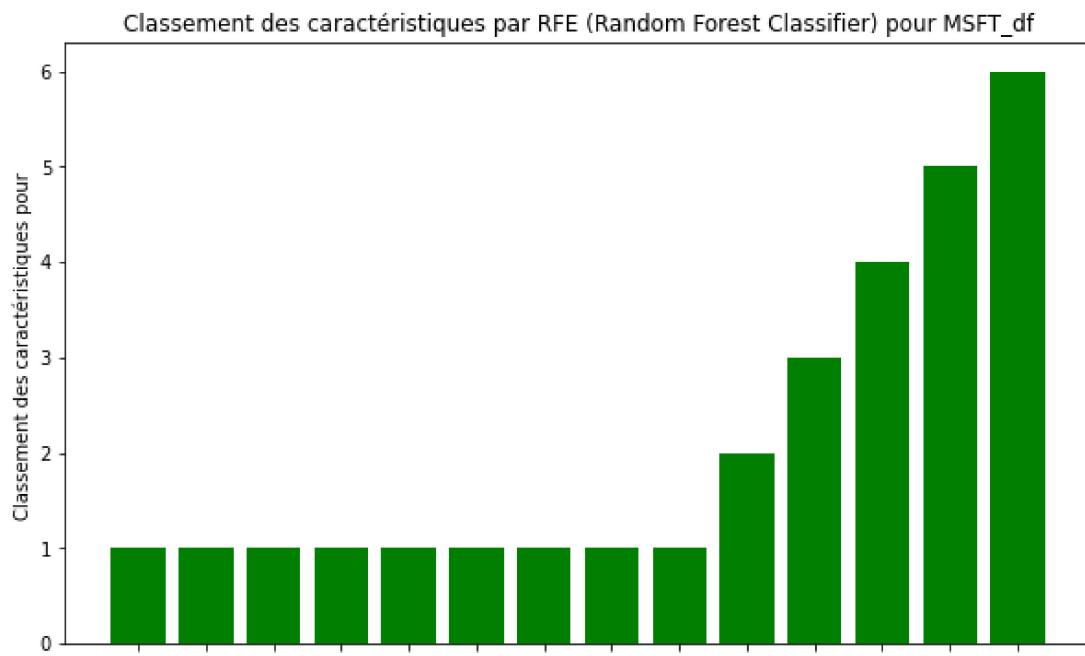
    # Trier les caractéristiques en fonction de leur classement
    sorted_features = [col for _, col in sorted(zip(ranking, x.columns))]

    # Trier les valeurs de classement
    sorted_ranking = sorted(ranking)

    # Créer un graphique à barres
    plt.figure(figsize=(10, 6))

    # Utilisez la fonction `bar` pour créer un graphique à barres verticales
    plt.bar(sorted_features, sorted_ranking, color='green')

    # Ajoutez une annotation pour chaque barre avec la valeur de classement
    plt.xlabel('Caractéristiques')
    plt.ylabel('Classement des caractéristiques pour ')
    plt.title('Classement des caractéristiques par RFE (Random Forest Classifier)')
    plt.xticks(rotation=45, ha='right') # Rotation des étiquettes sur l'axe
    plt.show()
```



D'après RFE , nous allons choisir [ATR , CMF_21 , CMF_28 , CLOSE , MACD_Histogram , MACD_Line , MACD_Signal , RSI_21, Volume]

Entrée [17]: `selected_features = ['ATR', 'CMF_28', 'Close', 'Daily_Range', 'MACD_Histogram', 'MACD_Line', 'MACD_Signal', 'RSI_21, Volume']
Sélectionnez la colonne 'Trend' comme votre variable cible
target_column = 'Trend'`

Entrée []:

8 Différentes modélisations de la variable à expliquer (target)

Entrée []:

Entrée []:

1. Régression logistique :

Entrée [18]:

```
import pandas as pd
import numpy as np
import statsmodels.api as sm
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report, confusion_matrix
import seaborn as sns
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, hamming_loss, confusion_matrix,
import seaborn as sns
import matplotlib.pyplot as plt
from tabulate import tabulate
from sklearn.metrics import recall_score, f1_score
```

Type *Markdown* and *LaTeX*: α^2

Entrée [19]:

```
!pip install prettytable
```

```
Requirement already satisfied: prettytable in c:\users\bouch\anaconda3\lib
\site-packages (3.9.0)
Requirement already satisfied: wcwidth in c:\users\bouch\anaconda3\lib\site-
packages (from prettytable) (0.2.5)
```



```

Entrée [23]: # Liste pour stocker les résultats de chaque modèle
from prettytable import PrettyTable
results_regression = []

# Parcourir chaque dataframe dans list_choice
for key, df in list_choice.items():

    # Créez un DataFrame avec les colonnes sélectionnées
    data = df[selected_features + [target_column]]
    # Supprimer les lignes avec des valeurs manquantes
    data.dropna(inplace=True)

    # Divisez les données en ensembles d'entraînement et de test
    X_train, X_test, y_train, y_test = train_test_split(data.drop(target_co

    # Standardisez les données
    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train)
    X_test_scaled = scaler.transform(X_test)

    # Entraînez le modèle de régression logistique
    model = LogisticRegression()
    model.fit(X_train_scaled, y_train)

    # Prédiction sur l'ensemble de test
    y_pred = model.predict(X_test_scaled)

    # Évaluation de la performance
    accuracy = accuracy_score(y_test, y_pred)
    conf_matrix = confusion_matrix(y_test, y_pred)
    class_report = classification_report(y_test, y_pred)
    hamming = hamming_loss(y_test, y_pred)
    recall = recall_score(y_test, y_pred, average='weighted')
    f1 = f1_score(y_test, y_pred, average='weighted')
    micro_recall = recall_score(y_test, y_pred, average='micro')
    micro_f1 = f1_score(y_test, y_pred, average='micro')
    macro_recall = recall_score(y_test, y_pred, average='macro')
    macro_f1 = f1_score(y_test, y_pred, average='macro')

    # Stocker les résultats dans la liste
    results_regression.append({
        'DataFrame': key,
        'Accuracy': accuracy,
        'Hamming': hamming,
        'Confusion Matrix': conf_matrix,
        'Classification Report': class_report,
        'Recall': recall,
        'F1 Score': f1,
        'Macro-average Recall': macro_recall,
        'Macro-average F1 Score': macro_f1,
        'Micro-average Recall': micro_recall,
        'Micro-average F1 Score': micro_f1,
    })

    # Affichage des résultats pour chaque dataframe
    for result in results_regression:
        print('\n' + '*' * 70)
        print(result['DataFrame'])
        print('*' * 70)

```

```

field_names = ['Metric', 'Value']
    # Create a PrettyTable object
table = PrettyTable(field_names)

# Add each metric value to the table

table.add_row(['Accuracy', f'{result['Accuracy']:.2%}'])
table.add_row(['Hamming', f'{result['Hamming']:.2%}'])
table.add_row(['Macro-average Recall', f'{result['Macro-average Recall']}'])
table.add_row(['Macro-average F1 Score', f'{result['Macro-average F1 Score']}'])
table.add_row(['Micro-average Recall', f'{result['Micro-average Recall']}'])
table.add_row(['Micro-average F1 Score', f'{result['Micro-average F1 Score']}'])

# Style the table
table.align['Metric'] = 'l'
table.align['Value'] = 'r'

# Print the table
print(table)

```

```

# Affichage de la matrice de confusion
plt.figure(figsize=(8, 6))
sns.heatmap(result['Confusion Matrix'], annot=True, fmt='d', cmap='Blues',
            xticklabels=model.classes_, yticklabels=model.classes_)
plt.xlabel('Prédit')
plt.ylabel('Réel')
plt.title(f'Matrice de Confusion pour DataFrame {result["DataFrame"]}')
plt.show()

# Affichage des résultats pour chaque dataframe sous forme de tableau
# table_headers = ['DataFrame', 'Accuracy', 'Hamming', 'Macro-average Recall',
#                  'Macro-average F1 Score', 'Micro-average Recall', 'Micro-average F1 Score']

# ## Afficher le tableau
# print(tabulate(table_rows, headers=table_headers, tablefmt='pretty'))

```

```
*****
MSFT_df
*****
+-----+-----+
| Metric | Value |
+-----+-----+
| Accuracy | 51.92% |
| Hamming | 48.08% |
| Macro-average Recall | 36.91% |
| Macro-average F1 Score | 30.39% |
| Micro-average Recall | 51.92% |
| Micro-average F1 Score | 51.92% |
+-----+-----+
```

Matrice de Confusion pour DataFrame MSFT_df

Entrée []:

1. **Accuracy (Précision)** : L'accuracy est la proportion d'observations correctement classées par le modèle. Dans votre cas, elle est d'environ 49%, ce qui signifie que le modèle a correctement prédit la classe de l'action (housse, baisse, stable) pour environ 49% des exemples dans l'ensemble de test.
2. **Confusion Matrix (Matrice de Confusion)** : La matrice de confusion montre le nombre d'observations correctement classées (diagonale principale) et les erreurs de classification. Les trois valeurs [23, 542, 25] sur la diagonale principale représentent les prédictions correctes pour les classes -1, 0, et 1, respectivement. Les autres valeurs représentent les erreurs.
3. **Classification Report (Rapport de Classification)** : Le rapport de classification fournit des mesures telles que la précision, le rappel (recall), et le f1-score pour chaque classe. Ces métriques donnent des informations sur la performance du modèle pour chaque classe individuellement. En général, la classe 0 (stable) semble mieux prédictive que les classes -1 (baisse) et 1 (housse).

En analysant ces résultats, vous pouvez identifier les points forts et faibles de votre modèle. Par exemple, vous pourriez explorer des techniques d'optimisation, ajuster les hyperparamètres, ou essayer d'autres algorithmes pour améliorer la performance du modèle, en particulier pour les classes -1 et 1 où la performance est actuellement plus faible.

Entrée []:

La matrice de corrélation que vous avez fournie montre les corrélations entre les différentes variables explicatives (features) ainsi que la variable cible Trend. Les valeurs de la matrice de corrélation varient entre -1 et 1, où :

- 1 indique une corrélation positive parfaite.
- -1 indique une corrélation négative parfaite.
- 0 indique aucune corrélation.

Quelques observations à partir de la matrice de corrélation :

1. Les variables Open, High, Low, Close, et Adj Close sont fortement corrélées entre elles, ce qui est attendu car ce sont toutes des mesures du prix d'une action.
2. Les indicateurs techniques tels que MACD_Line, MACD_Signal, MACD_Histogram, RSI_14, RSI_21, ATR, CMF_21, CMF_28, cmf_trend_21, cmf_trend_28 semblent avoir des corrélations significatives entre eux et avec les prix d'action.

3. La variable cible Trend ne semble pas avoir une corrélation très forte avec la plupart des variables explicatives, du moins en termes de corrélation linéaire.

Lors de l'utilisation de la régression logistique avec la technique de backward elimination, vous devriez éventuellement examiner la VIF (variance inflation factor) pour identifier les variables qui peuvent présenter une multicollinéarité. Cela peut être important pour améliorer la robustesse de votre modèle.

Entrée []:

2. Arbre de décision

Entrée [37]: `from sklearn.tree import DecisionTreeClassifier`

Entrée [38]: `N=14`

Entrée []:

```

# from sklearn.metrics import hamming_loss
# from sklearn.model_selection import GridSearchCV
# # Liste pour stocker les résultats de chaque modèle avec Grid Search, y compris les métriques supplémentaires
# results_tree_grid_search_hamming = []

# # Définir la grille des hyperparamètres à rechercher
# param_grid = {
#     'max_depth': [3, 5, 7, 10],
#     'min_samples_split': [2, 5, 10],
#     'min_samples_leaf': [1, 2, 4]
# }

# # Parcourir chaque dataframe dans list_choice
# for key, df in list_choice.items():
#     # Créez un DataFrame avec les colonnes sélectionnées
#     data = df[selected_features + [target_column]]
#     # Supprimer les lignes avec des valeurs manquantes
#     data.dropna(inplace=True)

#     # Divisez les données en ensembles d'entraînement et de test
#     X_train, X_test, y_train, y_test = train_test_split(data.drop(target_column), data[target_column], test_size=0.2, random_state=42)

#     # Création d'un arbre de décision
#     tree_model = DecisionTreeClassifier()

#     # Utiliser GridSearchCV pour trouver les meilleurs hyperparamètres
#     grid_search = GridSearchCV(tree_model, param_grid, cv=5, scoring='accuracy')
#     grid_search.fit(X_train, y_train)

#     # Prédictions sur l'ensemble de test
#     y_pred_tree = grid_search.predict(X_test)

#     # Évaluation du modèle
#     accuracy_tree = accuracy_score(y_test, y_pred_tree)
#     class_report_tree = classification_report(y_test, y_pred_tree)
#     conf_matrix_rf = confusion_matrix(y_test, y_pred_tree)
#     # Calculer la mesure de similarité de Hamming
#     hamming = hamming_loss(y_test, y_pred_tree)
#     recall = recall_score(y_test, y_pred_tree, average='weighted')
#     f1 = f1_score(y_test, y_pred_tree, average='weighted')
#     macro_recall = recall_score(y_test, y_pred_tree, average='macro')
#     macro_f1 = f1_score(y_test, y_pred_tree, average='macro')
#     micro_recall = recall_score(y_test, y_pred_tree, average='micro')
#     micro_f1 = f1_score(y_test, y_pred_tree, average='micro')

#     # Stocker les résultats dans la liste
#     results_tree_grid_search_hamming.append({
#         'DataFrame': key,
#         'Best Hyperparameters': grid_search.best_params_,
#         'Accuracy': accuracy_tree,
#         'Classification Report': class_report_tree,
#         'Hamming': hamming,
#         'Confusion Matrix': conf_matrix_rf,
#         'Classification Report': class_report_tree,
#         'Recall': recall,
#         'F1 Score': f1,
#         'Macro-average Recall': macro_recall,
#         'Macro-average F1 Score': macro_f1,
#         'Micro-average Recall': micro_recall,
#         'Micro-average F1 Score': micro_f1,
#     })

```

```
#     })

# # Affichage des résultats pour chaque dataframe avec Grid Search et Hamming
# for result in results_tree_grid_search_hamming:
#     print(f"\nResults for Decision Tree in DataFrame {result['DataFrame']}")
#     print(f"Best Hyperparameters: {result['Best Hyperparameters']}")  

#     print(f"Accuracy: {result['Accuracy']}")  

#     print(f"Hamming Loss: {result['Hamming Loss']}")  

#     print("\nClassification Report:")
#     print(result['Classification Report'])
#     print("*****")

#     print('\n' + '*' * 70)
#     print(result['DataFrame'])
#     print('*' * 70)

#     field_names = ['Metric', 'Value']
#         # Create a PrettyTable object
#     table = PrettyTable(field_names)

#     # Add each metric value to the table

#     table.add_row(['Accuracy', f"{result['Accuracy']:.2%}"])
#     table.add_row(['Hamming', f"{result['Hamming']:.2%}"])
#     table.add_row(['Macro-average Recall', f"{result['Macro-average Recall']:.2%}"])
#     table.add_row(['Macro-average F1 Score', f"{result['Macro-average F1 Score']:.2%}"])
#     table.add_row(['Micro-average Recall', f"{result['Micro-average Recall']:.2%}"])
#     table.add_row(['Micro-average F1 Score', f"{result['Micro-average F1 Score']:.2%}"])

#     # Style the table
#     table.align['Metric'] = 'l'
#     table.align['Value'] = 'r'

#     # Print the table
#     print(table)
```



Entrée []:

Entrée []:

Entrée []:

3. Random forest

Entrée [40]:

```

from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import accuracy_score, classification_report, hamming_
import matplotlib.pyplot as plt
from sklearn.tree import plot_tree

# Liste pour stocker les résultats de chaque modèle avec Grid Search, y com
results_rf_grid_search_hamming = []

# Définir la grille des hyperparamètres à rechercher pour Random Forest
param_grid_rf = {
    'n_estimators': [50, 100, 200],
    'max_depth': [None, 5, 10],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4]
}
key, df = next(iter(list_choice.items()))
# Parcourir chaque dataframe dans list_choice
for key, df in list_choice.items():

    # Créez un DataFrame avec les colonnes sélectionnées
    data = df[selected_features + [target_column]]
    # Supprimer les lignes avec des valeurs manquantes
    data.dropna(inplace=True)

    # Divisez les données en ensembles d'entraînement et de test
    X_train, X_test, y_train, y_test = train_test_split(data.drop(target_co

    # Création d'un modèle Random Forest
    rf_model = RandomForestClassifier()

    # Utiliser GridSearchCV pour trouver les meilleures hyperparamètres
    grid_search_rf = GridSearchCV(rf_model, param_grid_rf, cv=5, scoring='a
    grid_search_rf.fit(X_train, y_train)

    # Prédictions sur l'ensemble de test
    y_pred_rf = grid_search_rf.predict(X_test)

    # Évaluation du modèle
    accuracy_rf = accuracy_score(y_test, y_pred_rf)
    class_report_rf = classification_report(y_test, y_pred_rf)
    hamming_rf = hamming_loss(y_test, y_pred_rf)
    macro_recall_rf = recall_score(y_test, y_pred_rf, average='macro')
    macro_f1_rf = f1_score(y_test, y_pred_rf, average='macro')
    micro_recall_rf = recall_score(y_test, y_pred_rf, average='micro')
    micro_f1_rf = f1_score(y_test, y_pred_rf, average='micro')

    # Add the macro and micro averages to your results dictionary
    results_rf_grid_search_hamming.append({
        'DataFrame': key,
        'Best Hyperparameters': grid_search_rf.best_params_,
        'Accuracy': accuracy_rf,
        'Classification Report': class_report_rf,
        'Hamming Loss': hamming_rf,

        'Macro Recall': macro_recall_rf,
        'Macro F1 Score': macro_f1_rf,

        'Micro Recall': micro_recall_rf,
        'Micro F1 Score': micro_f1_rf,
    })

```

```
 })
```

Evaluation

Entrée [41]:

```
for result_rf in results_rf_grid_search_hamming :  
  
    print(f"\n{'*' * 70}\nResults for {result_rf['DataFrame']}\n{'*' * 70}"  
  
    # Afficher les résultats sous forme de tableau  
    #     table_headers1 = [ 'Accuracy', 'Hamming Loss', 'Macro Recall', 'Macro  
    #     table_headers2 = [ 'Best Hyperparameters'  
    table_rows = []  
    field_names = ['Metric', 'Value']  
        # Create a PrettyTable object  
    table = PrettyTable(field_names)  
    #     t=PrettyTable(field_names)  
    # Add each metric value to the table  
    print(result_rf['Best Hyperparameters'])  
    table.add_row(['Accuracy', f"{result_rf['Accuracy']:.2%}"])  
    table.add_row(['Hamming', f"{result_rf['Hamming Loss']:.2%}"])  
    table.add_row(['Macro-average Recall', f"{result_rf['Macro Recall']:.2%}"])  
    table.add_row(['Macro-average F1 Score', f"{result_rf['Macro F1 Score']:.2%}"])  
    table.add_row(['Micro-average Recall', f"{result_rf['Micro Recall']:.2%}"])  
    table.add_row(['Micro-average F1 Score', f"{result_rf['Micro F1 Score']:.2%}"]  
  
    # Style the table  
    table.align['Metric'] = 'l'  
    table.align['Value'] = 'r'  
  
    # Print the table  
    print(table)
```

```
*****
Results for MSFT_df
*****
{'max_depth': 10, 'min_samples_leaf': 2, 'min_samples_split': 2, 'n_estimators': 200}
+-----+-----+
| Metric | Value |
+-----+-----+
| Accuracy | 58.92% |
| Hamming | 41.08% |
| Macro-average Recall | 48.45% |
| Macro-average F1 Score | 48.93% |
| Micro-average Recall | 58.92% |
| Micro-average F1 Score | 58.92% |
+-----+-----+
```

```
*****
Results for AMZN_df
*****
{'max_depth': 10, 'min_samples_leaf': 1, 'min_samples_split': 5, 'n_estimators': 200}
+-----+-----+
| Metric | Value |
+-----+-----+
| Accuracy | 53.33% |
| Hamming | 46.67% |
| Macro-average Recall | 51.98% |
| Macro-average F1 Score | 52.07% |
| Micro-average Recall | 53.33% |
| Micro-average F1 Score | 53.33% |
+-----+-----+
```

```
*****
Results for TSLA_df
*****
{'max_depth': 10, 'min_samples_leaf': 1, 'min_samples_split': 5, 'n_estimators': 200}
+-----+-----+
| Metric | Value |
+-----+-----+
| Accuracy | 52.60% |
| Hamming | 47.40% |
| Macro-average Recall | 46.01% |
| Macro-average F1 Score | 42.62% |
| Micro-average Recall | 52.60% |
| Micro-average F1 Score | 52.60% |
+-----+-----+
```

```
*****
Results for RMSPA_df
*****
{'max_depth': 10, 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 50}
+-----+-----+
| Metric | Value |
+-----+-----+
| Accuracy | 55.06% |
| Hamming | 44.94% |
| Macro-average Recall | 45.79% |
| Macro-average F1 Score | 45.39% |
+-----+-----+
```

```
| Micro-average Recall | 55.06% |
| Micro-average F1 Score | 55.06% |
+-----+-----+
```

Results for GE_df

```
{'max_depth': 10, 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 50}
```

```
+-----+-----+
| Metric | Value |
+-----+-----+
| Accuracy | 56.00% |
| Hamming | 44.00% |
| Macro-average Recall | 48.51% |
| Macro-average F1 Score | 48.66% |
| Micro-average Recall | 56.00% |
| Micro-average F1 Score | 56.00% |
+-----+-----+
```

Results for TTE_df

```
{'max_depth': 10, 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 200}
```

```
+-----+-----+
| Metric | Value |
+-----+-----+
| Accuracy | 53.58% |
| Hamming | 46.42% |
| Macro-average Recall | 45.02% |
| Macro-average F1 Score | 44.87% |
| Micro-average Recall | 53.58% |
| Micro-average F1 Score | 53.58% |
+-----+-----+
```

4. SVM

```
Entrée [42]: # from sklearn.model_selection import train_test_split
# from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
# from sklearn.metrics import accuracy_score, confusion_matrix, classificat

# Liste pour stocker les résultats de chaque modèle
result_SVM = []

# Parcourir chaque dataframe dans list_choice
for key, df in list_choice.items():
    # Créez un DataFrame avec les colonnes sélectionnées
    data = df[selected_features + [target_column]]
    # Supprimer les lignes avec des valeurs manquantes
    data.dropna(inplace=True)

    # Divisez les données en ensembles d'entraînement et de test
    X_train, X_test, y_train, y_test = train_test_split(data.drop(target_co

    # Standardisez les données
    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train)
    X_test_scaled = scaler.transform(X_test)

    # Entraînez le modèle SVM
    model = SVC(random_state=42)
    model.fit(X_train_scaled, y_train)

    # Prédiction sur l'ensemble de test
    y_pred = model.predict(X_test_scaled)

    # Évaluation de la performance
    accuracy = accuracy_score(y_test, y_pred)
    conf_matrix = confusion_matrix(y_test, y_pred)
    class_report = classification_report(y_test, y_pred)
    hamming = hamming_loss(y_test, y_pred)
    macro_recall = recall_score(y_test, y_pred, average='macro')
    macro_f1 = f1_score(y_test, y_pred, average='macro')
    micro_recall = recall_score(y_test, y_pred, average='micro')
    micro_f1 = f1_score(y_test, y_pred, average='micro')

    # Stocker les résultats dans la liste
    result_SVM.append({
        'DataFrame': key,
        'Accuracy': accuracy,
        'Hamming': hamming,
        'Confusion Matrix': conf_matrix,
        'Classification Report': class_report,
        'macro_recall': macro_recall,
        'macro_f1': macro_f1,
        'micro_recall': micro_recall,
        'micro_f1': micro_f1
    })
}
```

Entrée []:

Entrée [54]:

```
for result_rf in result_SVM :  
  
    print(f"\n{'*' * 70}\nResults for {result_rf['DataFrame']}\n{'*' * 70}"  
  
    # Afficher les résultats sous forme de tableau  
    #     table_headers1 = [ 'Accuracy', 'Hamming Loss', 'Macro Recall', 'Macro  
    #     table_headers2 = [ 'Best Hyperparameters'  
    table_rows = []  
    field_names = ['Metric', 'Value']  
        # Create a PrettyTable object  
    table = PrettyTable(field_names)  
    #     t=PrettyTable(field_names)  
    # Add each metric value to the table  
  
    table.add_row(['Accuracy', f"{result_rf['Accuracy']:.2%}"])  
    table.add_row(['Hamming', f"{result_rf['Hamming']:.2%}"])  
    table.add_row(['Macro-average Recall', f"{result_rf['macro_recall']:.2%}"])  
    table.add_row(['Macro-average F1 Score', f"{result_rf['macro_f1']:.2%}"])  
    table.add_row(['Micro-average Recall', f"{result_rf['micro_recall']:.2%}"])  
    table.add_row(['Micro-average F1 Score', f"{result_rf['micro_f1']:.2%}"]  
  
    # Style the table  
    table.align['Metric'] = 'l'  
    table.align['Value'] = 'r'  
  
    # Print the table  
    print(table)
```

```
*****
Results for MSFT_df
*****
+-----+-----+
| Metric | Value |
+-----+-----+
| Accuracy | 53.33% |
| Hamming | 46.67% |
| Macro-average Recall | 38.50% |
| Macro-average F1 Score | 33.10% |
| Micro-average Recall | 53.33% |
| Micro-average F1 Score | 53.33% |
+-----+-----+*****
```

```
*****
Results for AMZN_df
*****
+-----+-----+
| Metric | Value |
+-----+-----+
| Accuracy | 43.67% |
| Hamming | 56.33% |
| Macro-average Recall | 41.09% |
| Macro-average F1 Score | 39.82% |
| Micro-average Recall | 43.67% |
| Micro-average F1 Score | 43.67% |
+-----+-----+*****
```

```
*****
Results for TSLA_df
*****
+-----+-----+
| Metric | Value |
+-----+-----+
| Accuracy | 44.13% |
| Hamming | 55.87% |
| Macro-average Recall | 36.60% |
| Macro-average F1 Score | 28.33% |
| Micro-average Recall | 44.13% |
| Micro-average F1 Score | 44.13% |
+-----+-----+*****
```

```
*****
Results for RMSPA_df
*****
+-----+-----+
| Metric | Value |
+-----+-----+
| Accuracy | 49.73% |
| Hamming | 50.27% |
| Macro-average Recall | 37.85% |
| Macro-average F1 Score | 32.65% |
| Micro-average Recall | 49.73% |
| Micro-average F1 Score | 49.73% |
+-----+-----+*****
```

```
*****
Results for GE_df
*****
+-----+-----+
```

Metric	Value
Accuracy	52.17%
Hamming	47.83%
Macro-average Recall	42.74%
Macro-average F1 Score	40.95%
Micro-average Recall	52.17%
Micro-average F1 Score	52.17%

Results for TTE_df

Metric	Value
Accuracy	51.17%
Hamming	48.83%
Macro-average Recall	39.65%
Macro-average F1 Score	35.86%
Micro-average Recall	51.17%
Micro-average F1 Score	51.17%

Entrée []:

[]

Entrée []:

[]

Entrée [45]: df.columns

```
Out[45]: Index(['Date', 'Open', 'High', 'Low', 'Close', 'Adj Close', 'Volume', 'Day',
       'Month', 'Year', 'Trend', 'MACD_Line', 'MACD_Signal', 'MACD_Histogram',
       'MACD_trend', 'RSI_14', 'RSI_trend_14', 'RSI_21', 'RSI_trend_21',
       'ATR',
       'CMF_21', 'CMF_28', 'cmf_trend_21', 'cmf_trend_28', 'Daily_Range',
       'Gap', 'SVM Signal'],
      dtype='object')
```

5. K plus proches voisins (KNN)

Entrée [56]:

```

from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import GridSearchCV, train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix

# Liste pour stocker les résultats de chaque modèle k-NN
result_knn = []

def train_knn_model(X_train, y_train, X_test, y_test, key):
    # Standardisation des features
    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train)
    X_test_scaled = scaler.transform(X_test)

    # Définition des hyperparamètres à rechercher
    param_grid = {'n_neighbors': [3, 5, 7, 9, 11]}

    # Recherche par grille pour trouver les meilleurs hyperparamètres
    grid_search = GridSearchCV(KNeighborsClassifier(), param_grid, cv=5)
    grid_search.fit(X_train_scaled, y_train)

    # Affichage des meilleurs hyperparamètres et du meilleur score
    print("Meilleurs hyperparamètres : ", grid_search.best_params_)
    print("Meilleur score : ", grid_search.best_score_)

    # Entraînement du modèle k-NN avec les meilleurs hyperparamètres
    best_knn_model = grid_search.best_estimator_
    best_knn_model.fit(X_train_scaled, y_train)

    # Prédictions sur l'ensemble de test
    y_pred_knn = best_knn_model.predict(X_test_scaled)

    # Évaluation du modèle
    accuracy = accuracy_score(y_test, y_pred_knn)
    conf_matrix = confusion_matrix(y_test, y_pred_knn)
    class_report = classification_report(y_test, y_pred_knn)
    hamming = hamming_loss(y_test, y_pred_knn)
    macro_recall = recall_score(y_test, y_pred_knn, average='macro')
    macro_f1 = f1_score(y_test, y_pred_knn, average='macro')
    micro_recall = recall_score(y_test, y_pred_knn, average='micro')
    micro_f1 = f1_score(y_test, y_pred_knn, average='micro')

    # Stocker les résultats dans la liste
    result_knn.append({
        'DataFrame': key,
        'Accuracy': accuracy,
        'Hamming': hamming,
        'Confusion Matrix': conf_matrix,
        'Classification Report': class_report,
        'Macro Recall': macro_recall,
        'Macro F1': macro_f1,
        'Micro Recall': micro_recall,
        'Micro F1': micro_f1
    })

# Parcourir chaque dataframe dans list_choice
for key, df in list_choice.items():
    # Créez un DataFrame avec les colonnes sélectionnées
    data = df[selected_features + [target_column]]
    # Supprimer les lignes avec des valeurs manquantes
    data.dropna(inplace=True)

```

```
# Divisez les données en ensembles d'entraînement et de test
X_train, X_test, y_train, y_test = train_test_split(data.drop(target_co
print(f"\nTraining k-NN model for DataFrame {key}:")
train_knn_model(X_train, y_train, X_test, y_test, key)
```

◀ ▶

```
Training k-NN model for DataFrame MSFT_df:
Meilleurs hyperparamètres : {'n_neighbors': 11}
Meilleur score : 0.4842676833507126
```

```
Training k-NN model for DataFrame AMZN_df:
Meilleurs hyperparamètres : {'n_neighbors': 3}
Meilleur score : 0.45884428223844276
```

```
Training k-NN model for DataFrame TSLA_df:
Meilleurs hyperparamètres : {'n_neighbors': 3}
Meilleur score : 0.4897807591396509
```

```
Training k-NN model for DataFrame RMSPA_df:
Meilleurs hyperparamètres : {'n_neighbors': 5}
Meilleur score : 0.5246496408211263
```

```
Training k-NN model for DataFrame GE_df:
Meilleurs hyperparamètres : {'n_neighbors': 5}
Meilleur score : 0.5536555005213764
```

```
Training k-NN model for DataFrame TTE_df:
Meilleurs hyperparamètres : {'n_neighbors': 3}
Meilleur score : 0.5148978971150504
```

Entrée [58]:

```
for result_rf in result_knn :  
  
    print(f"\n{'*' * 70}\nResults for {result_rf['DataFrame']}\n{'*' * 70}"  
    table_rows = []  
    field_names = ['Metric', 'Value']  
    table = PrettyTable(field_names)  
    table.add_row(['Accuracy', f'{result_rf['Accuracy']:.2%}'])  
    table.add_row(['Hamming', f'{result_rf['Hamming']:.2%}'])  
    table.add_row(['Macro-average Recall', f'{result_rf['Macro Recall']:.2%}'])  
    table.add_row(['Macro-average F1 Score', f'{result_rf['Macro F1']:.2%}'])  
    table.add_row(['Micro-average Recall', f'{result_rf['Micro Recall']:.2%}'])  
    table.add_row(['Micro-average F1 Score', f'{result_rf['Micro F1']:.2%}'])  
  
    # Style the table  
    table.align['Metric'] = 'l'  
    table.align['Value'] = 'r'  
  
    # Print the table  
    print(table)
```

```
*****
Results for MSFT_df
*****
+-----+-----+
| Metric | Value |
+-----+-----+
| Accuracy | 52.08% |
| Hamming | 47.92% |
| Macro-average Recall | 43.80% |
| Macro-average F1 Score | 43.21% |
| Micro-average Recall | 52.08% |
| Micro-average F1 Score | 52.08% |
+-----+-----+*****
```

```
*****
Results for AMZN_df
*****
+-----+-----+
| Metric | Value |
+-----+-----+
| Accuracy | 47.08% |
| Hamming | 52.92% |
| Macro-average Recall | 47.94% |
| Macro-average F1 Score | 47.07% |
| Micro-average Recall | 47.08% |
| Micro-average F1 Score | 47.08% |
+-----+-----+*****
```

```
*****
Results for TSLA_df
*****
+-----+-----+
| Metric | Value |
+-----+-----+
| Accuracy | 48.59% |
| Hamming | 51.41% |
| Macro-average Recall | 44.88% |
| Macro-average F1 Score | 44.11% |
| Micro-average Recall | 48.59% |
| Micro-average F1 Score | 48.59% |
+-----+-----+*****
```

```
*****
Results for RMSPA_df
*****
+-----+-----+
| Metric | Value |
+-----+-----+
| Accuracy | 50.00% |
| Hamming | 50.00% |
| Macro-average Recall | 46.65% |
| Macro-average F1 Score | 46.85% |
| Micro-average Recall | 50.00% |
| Micro-average F1 Score | 50.00% |
+-----+-----+*****
```

```
*****
Results for GE_df
*****
+-----+-----+
```

Metric	Value
Accuracy	54.83%
Hamming	45.17%
Macro-average Recall	51.12%
Macro-average F1 Score	51.31%
Micro-average Recall	54.83%
Micro-average F1 Score	54.83%

Results for TTE_df

Metric	Value
Accuracy	54.67%
Hamming	45.33%
Macro-average Recall	53.39%
Macro-average F1 Score	52.98%
Micro-average Recall	54.67%
Micro-average F1 Score	54.67%

6. Gradient boosting

Entrée [50]:

```

from sklearn.model_selection import RandomizedSearchCV, train_test_split
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.metrics import accuracy_score, classification_report
from sklearn.preprocessing import StandardScaler

def optimize_gb_model(X_train, y_train, X_test, y_test):
    # Standardisation des features
    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train)
    X_test_scaled = scaler.transform(X_test)

    # Définir les hyperparamètres à optimiser
    param_dist = {
        'n_estimators': [50, 100, 150, 200],
        'learning_rate': [0.01, 0.1, 0.2, 0.3],
        'max_depth': [3, 4, 5, 6]
    }

    # Initialiser le modèle Gradient Boosting
    gb_model = GradientBoostingClassifier(random_state=42)

    # Initialiser la recherche d'hyperparamètres
    random_search = RandomizedSearchCV(gb_model, param_dist, n_iter=10, cv=5)

    # Effectuer la recherche d'hyperparamètres sur les données d'entraînement
    random_search.fit(X_train_scaled, y_train)

    # Afficher les meilleures hyperparamètres et le meilleur score
    print("Meilleurs hyperparamètres après recherche aléatoire : ", random_search.best_params_)
    print("Meilleur score après recherche aléatoire : ", random_search.best_score_)

    # Prédictions sur l'ensemble de test avec les meilleures hyperparamètres
    y_pred_gb_optimized = random_search.predict(X_test_scaled)

    # Évaluation du modèle optimisé
    print("\nAccuracy Score (optimized):", accuracy_score(y_test, y_pred_gb_optimized))
    print("\nClassification Report (optimized):\n", classification_report(y_test, y_pred_gb_optimized))

    # Parcourir chaque dataframe dans list_choice
    for key, df in list_choice.items():
        # Créez un DataFrame avec les colonnes sélectionnées
        data = df[selected_features + [target_column]]
        # Supprimer les lignes avec des valeurs manquantes
        data.dropna(inplace=True)

        # Divisez les données en ensembles d'entraînement et de test
        X_train, X_test, y_train, y_test = train_test_split(data.drop(target_column, axis=1), data[target_column], test_size=0.2, random_state=42)

        print(f"\nOptimizing Gradient Boosting model for DataFrame {key}:")
        optimize_gb_model(X_train, y_train, X_test, y_test)

```



```
Optimizing Gradient Boosting model for DataFrame MSFT_df:  
Meilleurs hyperparamètres après recherche aléatoire : {'n_estimators':  
100, 'max_depth': 6, 'learning_rate': 0.3}  
Meilleur score après recherche aléatoire : 0.5611598453249913
```

```
Accuracy Score (optimized): 0.5708333333333333
```

```
Classification Report (optimized):  
precision recall f1-score support  
  
-1.0      0.48    0.36    0.41    255  
 0.0      0.60    0.76    0.67    599  
 1.0      0.55    0.40    0.46    346  
  
accuracy          0.57    1200  
macro avg       0.54    0.51    0.51    1200  
weighted avg     0.56    0.57    0.56    1200
```

```
Entrée [ ]:
```

```
Entrée [ ]: # Parcourir chaque dataframe dans List_choice  
for key, df in list_choice.items():  
    # Créez un DataFrame avec les colonnes sélectionnées  
    data = df[selected_features + [target_column]]  
    # Supprimer les lignes avec des valeurs manquantes  
    data.dropna(inplace=True)  
  
    # Divisez les données en ensembles d'entraînement et de test  
    X_train, X_test, y_train, y_test = train_test_split(data.drop(target_co  
  
    print(f"\nOptimizing Gradient Boosting model for DataFrame {key}:")  
    optimize_gb_model(X_train, y_train, X_test, y_test)
```

7.LSTM

```
Entrée [ ]:
```



```

Entrée [60]: import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler, LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import hamming_loss, precision_recall_fscore_support
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout
from tensorflow.keras.utils import to_categorical

# Liste pour stocker les résultats de chaque modèle LSTM
results_LSTM = []

def train_lstm_model(df, features, key):
    # Définir Les caractéristiques (X) et La cible (y)
    X = df[features].values
    y = df['Trend'].values

    # Encoder Les étiquettes en utilisant LabelEncoder
    label_encoder = LabelEncoder()
    y_encoded = label_encoder.fit_transform(y)

    # Normaliser Les données
    scaler = MinMaxScaler(feature_range=(0, 1))
    X_scaled = scaler.fit_transform(X)

    # Diviser Les données en ensembles d'entraînement et de test avec strat
    X_train, X_test, y_train, y_test = train_test_split(X_scaled, y_encoded)

    # Remodeler Les données pour les rendre compatibles avec LSTM
    X_train = X_train.reshape((X_train.shape[0], 1, X_train.shape[1]))
    X_test = X_test.reshape((X_test.shape[0], 1, X_test.shape[1]))

    # Convertir Les étiquettes en encodage one-hot (si nécessaire)
    num_classes = len(np.unique(y))
    y_train_onehot = to_categorical(y_train, num_classes=num_classes)
    y_test_onehot = to_categorical(y_test, num_classes=num_classes)

    # Créer Le modèle LSTM
    model = Sequential()
    model.add(LSTM(300, input_shape=(X_train.shape[1], X_train.shape[2])))
    model.add(Dropout(0.5))
    model.add(Dense(num_classes, activation='softmax'))

    # Compiler Le modèle
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=[accuracy])

    # Entrainer Le modèle et enregistrer L'historique de L'entraînement
    history = model.fit(X_train, y_train_onehot, epochs=10, batch_size=32, verbose=1)

    # Évaluation sur L'ensemble de test
    test_loss, test_accuracy = model.evaluate(X_test, y_test_onehot)
    print(f'\nTest Accuracy: {test_accuracy}')

    # Prédictions sur L'ensemble de test
    y_pred = model.predict(X_test)

    # Convertir Les prédictions en classe binaire (0, 1, 2) plutôt qu'en entier
    y_pred_classes = np.argmax(y_pred, axis=1)
    y_test_classes = np.argmax(y_test_onehot, axis=1)

    results_LSTM.append({
        'key': key,
        'model': model,
        'history': history,
        'test_loss': test_loss,
        'test_accuracy': test_accuracy,
        'y_pred': y_pred,
        'y_pred_classes': y_pred_classes,
        'y_test': y_test,
        'y_test_classes': y_test_classes
    })

```

```

# Calcul de l'indice de Hamming
hamming = hamming_loss(y_test_classes, y_pred_classes)
print(f'\nHamming Loss: {hamming}')

# Calcul du micro-average (précision, rappel, f1-score)
micro_avg = precision_recall_fscore_support(y_test_classes, y_pred_classes)

# Calcul du macro-average (précision, rappel, f1-score)
macro_avg = precision_recall_fscore_support(y_test_classes, y_pred_classes)

results_LSTM.append({
    'DataFrame': key,
    'Test Accuracy': test_accuracy,
    'Hamming Loss': hamming,
    'Micro-average Precision': micro_avg[0],
    'Micro-average Recall': micro_avg[1],
    'Micro-average F1 Score': micro_avg[2],
    'Macro-average Precision': macro_avg[0],
    'Macro-average Recall': macro_avg[1],
    'Macro-average F1 Score': macro_avg[2]
})

# Appliquer la fonction à chaque dataframe de la liste
for key, df in list_choice.items():
    print(f'\nTraining LSTM Model for DataFrame {key}...')
    train_lstm_model(df, selected_features, key)

```

Training LSTM Model for DataFrame MSFT_df...

Epoch 1/10
150/150 [=====] - 5s 17ms/step - loss: 1.0432
- accuracy: 0.4922 - val_loss: 1.0331 - val_accuracy: 0.4958

Epoch 2/10
150/150 [=====] - 1s 10ms/step - loss: 1.0317
- accuracy: 0.4949 - val_loss: 1.0287 - val_accuracy: 0.4958

Epoch 3/10
150/150 [=====] - 1s 10ms/step - loss: 1.0314
- accuracy: 0.4959 - val_loss: 1.0317 - val_accuracy: 0.4967

Epoch 4/10
150/150 [=====] - 2s 10ms/step - loss: 1.0272
- accuracy: 0.4984 - val_loss: 1.0253 - val_accuracy: 0.4917

Epoch 5/10
150/150 [=====] - 1s 10ms/step - loss: 1.0264
- accuracy: 0.4974 - val_loss: 1.0236 - val_accuracy: 0.4933

Epoch 6/10
150/150 [=====] - 2s 11ms/step - loss: 1.0241
- accuracy: 0.4999 - val_loss: 1.0242 - val_accuracy: 0.4959

Entrée [65]:

```
for result_rf in results_LSTM :  
  
    print(f"\n{'*' * 70}\nResults for {result_rf['DataFrame']}\n{'*' * 70}"  
    table_rows = []  
    field_names = ['Metric', 'Value']  
    table = PrettyTable(field_names)  
    table.add_row(['Accuracy', f'{result_rf['Test Accuracy']:.2%}'])  
    table.add_row(['Hamming', f'{result_rf['Hamming Loss']:.2%}'])  
    table.add_row(['Macro-average Recall', f'{result_rf['Macro-average Recall']:.2%}'])  
    table.add_row(['Macro-average F1 Score', f'{result_rf['Macro-average F1 Score']:.2%}'])  
    table.add_row(['Micro-average Recall', f'{result_rf['Micro-average Recall']:.2%}'])  
    table.add_row(['Micro-average F1 Score', f'{result_rf['Micro-average F1 Score']:.2%}'])  
  
    # Style the table  
    table.align['Metric'] = 'l'  
    table.align['Value'] = 'r'  
  
    # Print the table  
    print(table)
```

```
*****
Results for MSFT_df
*****
+-----+-----+
| Metric | Value |
+-----+-----+
| Accuracy | 49.50% |
| Hamming | 50.50% |
| Macro-average Recall | 34.97% |
| Macro-average F1 Score | 27.72% |
| Micro-average Recall | 49.50% |
| Micro-average F1 Score | 49.50% |
+-----+-----+*****
```

```
*****
Results for AMZN_df
*****
+-----+-----+
| Metric | Value |
+-----+-----+
| Accuracy | 41.00% |
| Hamming | 59.00% |
| Macro-average Recall | 38.51% |
| Macro-average F1 Score | 32.13% |
| Micro-average Recall | 41.00% |
| Micro-average F1 Score | 41.00% |
+-----+-----+*****
```

```
*****
Results for TSLA_df
*****
+-----+-----+
| Metric | Value |
+-----+-----+
| Accuracy | 41.60% |
| Hamming | 58.40% |
| Macro-average Recall | 33.71% |
| Macro-average F1 Score | 27.29% |
| Micro-average Recall | 41.60% |
| Micro-average F1 Score | 41.60% |
+-----+-----+*****
```

```
*****
Results for RMSPA_df
*****
+-----+-----+
| Metric | Value |
+-----+-----+
| Accuracy | 47.65% |
| Hamming | 52.35% |
| Macro-average Recall | 34.46% |
| Macro-average F1 Score | 26.32% |
| Micro-average Recall | 47.65% |
| Micro-average F1 Score | 47.65% |
+-----+-----+*****
```

```
*****
Results for GE_df
*****
+-----+-----+
```

Metric	Value
Accuracy	51.50%
Hamming	48.50%
Macro-average Recall	41.22%
Macro-average F1 Score	38.19%
Micro-average Recall	51.50%
Micro-average F1 Score	51.50%

Results for TTE_df

Metric	Value
Accuracy	48.00%
Hamming	52.00%
Macro-average Recall	36.62%
Macro-average F1 Score	30.49%
Micro-average Recall	48.00%
Micro-average F1 Score	48.00%

Entrée []:

8. Réseaux de neurones récurrents (RNN)

```

Entrée [66]: import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler, LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import hamming_loss, precision_recall_fscore_support
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import SimpleRNN, Dense, Dropout
from tensorflow.keras.utils import to_categorical

# Liste pour stocker les résultats de chaque modèle RNN
results_RNN = []

def train_rnn_model(df, features, key):
    # Définir Les caractéristiques (X) et La cible (y)
    X = df[features].values
    y = df['Trend'].values

    # Encoder Les étiquettes en utilisant LabelEncoder
    label_encoder = LabelEncoder()
    y_encoded = label_encoder.fit_transform(y)

    # Normaliser Les données
    scaler = MinMaxScaler(feature_range=(0, 1))
    X_scaled = scaler.fit_transform(X)

    # Diviser Les données en ensembles d'entraînement et de test avec strat
    X_train, X_test, y_train, y_test = train_test_split(X_scaled, y_encoded)

    # Remodeler Les données pour les rendre compatibles avec RNN
    X_train = X_train.reshape((X_train.shape[0], 1, X_train.shape[1]))
    X_test = X_test.reshape((X_test.shape[0], 1, X_test.shape[1]))

    # Convertir Les étiquettes en encodage one-hot (si nécessaire)
    num_classes = len(np.unique(y))
    y_train_onehot = to_categorical(y_train, num_classes=num_classes)
    y_test_onehot = to_categorical(y_test, num_classes=num_classes)

    # Créer Le modèle RNN
    model = Sequential()
    model.add(SimpleRNN(300, input_shape=(X_train.shape[1], X_train.shape[2])))
    model.add(Dropout(0.5))
    model.add(Dense(num_classes, activation='softmax'))

    # Compiler Le modèle
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=[hamming_loss])

    # Entrainer Le modèle et enregistrer L'historique de L'entraînement
    history = model.fit(X_train, y_train_onehot, epochs=10, batch_size=32, verbose=1)

    # Évaluation sur L'ensemble de test
    test_loss, test_accuracy = model.evaluate(X_test, y_test_onehot)
    print(f'\nTest Accuracy: {test_accuracy}')

    # Prédictions sur L'ensemble de test
    y_pred = model.predict(X_test)

    # Convertir Les prédictions en classe binaire (0, 1, 2) plutôt qu'en entier
    y_pred_classes = np.argmax(y_pred, axis=1)
    y_test_classes = np.argmax(y_test_onehot, axis=1)

    results_RNN.append({
        'key': key,
        'model': model,
        'history': history,
        'test_loss': test_loss,
        'test_accuracy': test_accuracy,
        'y_pred_classes': y_pred_classes,
        'y_test_classes': y_test_classes
    })

```

```

# Calcul de l'indice de Hamming
hamming = hamming_loss(y_test_classes, y_pred_classes)

# Calcul du micro-average (précision, rappel, f1-score)
micro_avg = precision_recall_fscore_support(y_test_classes, y_pred_classes)

# Calcul du macro-average (précision, rappel, f1-score)
macro_avg = precision_recall_fscore_support(y_test_classes, y_pred_classes)
print(f'\nMacro-average Precision: {macro_avg[0]}')
print(f'Macro-average Recall: {macro_avg[1]}')
print(f'Macro-average F1 Score: {macro_avg[2]}')

results_RNN.append({
    'DataFrame': key,
    'Test Accuracy': test_accuracy,
    'Hamming Loss': hamming,
    'Micro-average Precision': micro_avg[0],
    'Micro-average Recall': micro_avg[1],
    'Micro-average F1 Score': micro_avg[2],
    'Macro-average Precision': macro_avg[0],
    'Macro-average Recall': macro_avg[1],
    'Macro-average F1 Score': macro_avg[2]
})

# Appliquer la fonction à chaque dataframe de la liste
for key, df in list_choice.items():
    print(f'\nTraining RNN Model for DataFrame {key}...')
    train_rnn_model(df, selected_features, key)

```

```

Training RNN Model for DataFrame MSFT_df...
Epoch 1/10
150/150 [=====] - 2s 7ms/step - loss: 1.0416 - accuracy: 0.4920 - val_loss: 1.0282 - val_accuracy: 0.4933
Epoch 2/10
150/150 [=====] - 1s 5ms/step - loss: 1.0321 - accuracy: 0.4976 - val_loss: 1.0265 - val_accuracy: 0.4942
Epoch 3/10
150/150 [=====] - 1s 6ms/step - loss: 1.0312 - accuracy: 0.4926 - val_loss: 1.0279 - val_accuracy: 0.4900
Epoch 4/10
150/150 [=====] - 1s 5ms/step - loss: 1.0265 - accuracy: 0.4951 - val_loss: 1.0265 - val_accuracy: 0.4992
Epoch 5/10
150/150 [=====] - 1s 5ms/step - loss: 1.0253 - accuracy: 0.4993 - val_loss: 1.0267 - val_accuracy: 0.4925
Epoch 6/10
150/150 [=====] - 1s 5ms/step - loss: 1.0245 - accuracy: 0.4999 - val_loss: 1.0242 - val_accuracy: 0.4950

```

Entrée [67]:

```
for result_rf in results_RNN :  
  
    print(f"\n{'*' * 70}\nResults for {result_rf['DataFrame']}\n{'*' * 70}")  
    table_rows = []  
    field_names = ['Metric', 'Value']  
    table = PrettyTable(field_names)  
    table.add_row(['Accuracy', f'{result_rf['Test Accuracy']:.2%}'])  
    table.add_row(['Hamming', f'{result_rf['Hamming Loss']:.2%}'])  
    table.add_row(['Macro-average Recall', f'{result_rf['Macro-average Recall']:.2%}'])  
    table.add_row(['Macro-average F1 Score', f'{result_rf['Macro-average F1 Score']:.2%}'])  
    table.add_row(['Micro-average Recall', f'{result_rf['Micro-average Recall']:.2%}'])  
    table.add_row(['Micro-average F1 Score', f'{result_rf['Micro-average F1 Score']:.2%}'])  
  
    # Style the table  
    table.align['Metric'] = 'l'  
    table.align['Value'] = 'r'  
  
    # Print the table  
    print(table)
```

```
*****
Results for MSFT_df
*****
+-----+-----+
| Metric | Value |
+-----+-----+
| Accuracy | 49.17% |
| Hamming | 50.83% |
| Macro-average Recall | 35.00% |
| Macro-average F1 Score | 28.39% |
| Micro-average Recall | 49.17% |
| Micro-average F1 Score | 49.17% |
+-----+-----+*****
```

```
*****
Results for AMZN_df
*****
+-----+-----+
| Metric | Value |
+-----+-----+
| Accuracy | 41.67% |
| Hamming | 58.33% |
| Macro-average Recall | 39.13% |
| Macro-average F1 Score | 32.63% |
| Micro-average Recall | 41.67% |
| Micro-average F1 Score | 41.67% |
+-----+-----+*****
```

```
*****
Results for TSLA_df
*****
+-----+-----+
| Metric | Value |
+-----+-----+
| Accuracy | 42.94% |
| Hamming | 57.06% |
| Macro-average Recall | 34.20% |
| Macro-average F1 Score | 25.99% |
| Micro-average Recall | 42.94% |
| Micro-average F1 Score | 42.94% |
+-----+-----+*****
```

```
*****
Results for RMSPA_df
*****
+-----+-----+
| Metric | Value |
+-----+-----+
| Accuracy | 47.29% |
| Hamming | 52.71% |
| Macro-average Recall | 34.32% |
| Macro-average F1 Score | 26.37% |
| Micro-average Recall | 47.29% |
| Micro-average F1 Score | 47.29% |
+-----+-----+*****
```

```
*****
Results for GE_df
*****
+-----+-----+
```

Metric	Value
Accuracy	50.83%
Hamming	49.17%
Macro-average Recall	40.10%
Macro-average F1 Score	35.19%
Micro-average Recall	50.83%
Micro-average F1 Score	50.83%

Results for TTE_df

Metric	Value
Accuracy	48.00%
Hamming	52.00%
Macro-average Recall	36.46%
Macro-average F1 Score	30.09%
Micro-average Recall	48.00%
Micro-average F1 Score	48.00%

Entrée []:

9.Réseaux de neurones convolutifs (CNN)

Entrée [70]:

```

import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler, LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import hamming_loss, precision_recall_fscore_support
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv1D, MaxPooling1D, Flatten, Dense, Dropout
from tensorflow.keras.utils import to_categorical

# Liste pour stocker les résultats de chaque modèle CNN
results_CNN = []

def train_cnn_model(df, features, key):
    # Définir les caractéristiques (X) et la cible (y)
    X = df[features].values
    y = df['Trend'].values

    # Encoder les étiquettes en utilisant LabelEncoder
    label_encoder = LabelEncoder()
    y_encoded = label_encoder.fit_transform(y)

    # Normaliser les données
    scaler = MinMaxScaler(feature_range=(0, 1))
    X_scaled = scaler.fit_transform(X)

    # Diviser les données en ensembles d'entraînement et de test avec stratification
    X_train, X_test, y_train, y_test = train_test_split(X_scaled, y_encoded, test_size=0.2, random_state=42)

    # Remodeler les données pour les rendre compatibles avec CNN
    X_train = X_train.reshape((X_train.shape[0], X_train.shape[1], 1))
    X_test = X_test.reshape((X_test.shape[0], X_test.shape[1], 1))

    # Convertir les étiquettes en encodage one-hot (si nécessaire)
    num_classes = len(np.unique(y))
    y_train_onehot = to_categorical(y_train, num_classes=num_classes)
    y_test_onehot = to_categorical(y_test, num_classes=num_classes)

    # Créer le modèle CNN
    model = Sequential()
    model.add(Conv1D(filters=64, kernel_size=3, activation='relu', input_shape=(X_train.shape[1], 1)))
    model.add(MaxPooling1D(pool_size=2))
    model.add(Flatten())
    model.add(Dense(50, activation='relu'))
    model.add(Dense(num_classes, activation='softmax'))

    # Compiler le modèle
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

    # Entrainer le modèle et enregistrer l'historique de l'entraînement
    history = model.fit(X_train, y_train_onehot, epochs=10, batch_size=32, verbose=1)

    # Afficher les courbes d'apprentissage (loss et accuracy) au fil des épisodes
    plt.figure(figsize=(12, 4))
    plt.subplot(1, 2, 1)
    plt.plot(history.history['loss'], label='Train Loss')
    plt.plot(history.history['val_loss'], label='Validation Loss')
    plt.title('Training and Validation Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()

```

```

plt.subplot(1, 2, 2)
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

plt.tight_layout()
plt.show()

# Évaluation sur l'ensemble de test
test_loss, test_accuracy = model.evaluate(X_test, y_test_onehot)
print(f'\nTest Accuracy: {test_accuracy}')

# Prédictions sur l'ensemble de test
y_pred = model.predict(X_test)

# Convertir les prédictions en classe binaire (0, 1, 2) plutôt qu'en en
y_pred_classes = np.argmax(y_pred, axis=1)
y_test_classes = np.argmax(y_test_onehot, axis=1)

# Calcul de l'indice de Hamming
hamming = hamming_loss(y_test_classes, y_pred_classes)

# Calcul du micro-average (précision, rappel, f1-score)
micro_avg = precision_recall_fscore_support(y_test_classes, y_pred_classes)

# Calcul du macro-average (précision, rappel, f1-score)
macro_avg = precision_recall_fscore_support(y_test_classes, y_pred_classes)

results_CNN.append({
    'DataFrame': key,
    'Test Accuracy': test_accuracy,
    'Hamming Loss': hamming,
    'Micro-average Precision': micro_avg[0],
    'Micro-average Recall': micro_avg[1],
    'Micro-average F1 Score': micro_avg[2],
    'Macro-average Precision': macro_avg[0],
    'Macro-average Recall': macro_avg[1],
    'Macro-average F1 Score': macro_avg[2]
})

# Appliquer la fonction à chaque dataframe de la liste
for key, df in list_choice.items():
    print(f'\nTraining CNN Model for DataFrame {key}... ')
    train_cnn_model(df, selected_features, key)

```

```
Training CNN Model for DataFrame MSFT_df...
Epoch 1/10
150/150 [=====] - 1s 6ms/step - loss: 1.0375 - accuracy: 0.4932 - val_loss: 1.0330 - val_accuracy: 0.4942
Epoch 2/10
150/150 [=====] - 0s 3ms/step - loss: 1.0321 - accuracy: 0.4936 - val_loss: 1.0309 - val_accuracy: 0.4942
Epoch 3/10
150/150 [=====] - 0s 3ms/step - loss: 1.0298 - accuracy: 0.4936 - val_loss: 1.0285 - val_accuracy: 0.4950
Epoch 4/10
150/150 [=====] - 0s 3ms/step - loss: 1.0277 - accuracy: 0.4966 - val_loss: 1.0272 - val_accuracy: 0.4975
Epoch 5/10
150/150 [=====] - 0s 3ms/step - loss: 1.0265 - accuracy: 0.5018 - val_loss: 1.0254 - val_accuracy: 0.4967
Epoch 6/10
150/150 [=====] - 0s 3ms/step - loss: 1.0240 -
```

Entrée [71]:

```
for result_rf in results_CNN :  
  
    print(f"\n{'*' * 70}\nResults for {result_rf['DataFrame']}\n{'*' * 70}")  
    table_rows = []  
    field_names = ['Metric', 'Value']  
    table = PrettyTable(field_names)  
    table.add_row(['Accuracy', f'{result_rf['Test Accuracy']:.2%}'])  
    table.add_row(['Hamming', f'{result_rf['Hamming Loss']:.2%}'])  
    table.add_row(['Macro-average Recall', f'{result_rf['Macro-average Recall']:.2%}'])  
    table.add_row(['Macro-average F1 Score', f'{result_rf['Macro-average F1 Score']:.2%}'])  
    table.add_row(['Micro-average Recall', f'{result_rf['Micro-average Recall']:.2%}'])  
    table.add_row(['Micro-average F1 Score', f'{result_rf['Micro-average F1 Score']:.2%}'])  
  
    # Style the table  
    table.align['Metric'] = 'l'  
    table.align['Value'] = 'r'  
  
    # Print the table  
    print(table)
```

```
*****
Results for MSFT_df
*****
+-----+-----+
| Metric | Value |
+-----+-----+
| Accuracy | 50.08% |
| Hamming | 49.92% |
| Macro-average Recall | 34.60% |
| Macro-average F1 Score | 25.53% |
| Micro-average Recall | 50.08% |
| Micro-average F1 Score | 50.08% |
+-----+-----+*****
```

```
*****
Results for AMZN_df
*****
+-----+-----+
| Metric | Value |
+-----+-----+
| Accuracy | 41.33% |
| Hamming | 58.67% |
| Macro-average Recall | 38.99% |
| Macro-average F1 Score | 33.72% |
| Micro-average Recall | 41.33% |
| Micro-average F1 Score | 41.33% |
+-----+-----+*****
```

```
*****
Results for TSLA_df
*****
+-----+-----+
| Metric | Value |
+-----+-----+
| Accuracy | 42.20% |
| Hamming | 57.80% |
| Macro-average Recall | 34.88% |
| Macro-average F1 Score | 29.48% |
| Micro-average Recall | 42.20% |
| Micro-average F1 Score | 42.20% |
+-----+-----+*****
```

```
*****
Results for RMSPA_df
*****
+-----+-----+
| Metric | Value |
+-----+-----+
| Accuracy | 47.65% |
| Hamming | 52.35% |
| Macro-average Recall | 34.31% |
| Macro-average F1 Score | 25.76% |
| Micro-average Recall | 47.65% |
| Micro-average F1 Score | 47.65% |
+-----+-----+*****
```

```
*****
Results for GE_df
*****
+-----+-----+
```

Metric	Value
Accuracy	51.75%
Hamming	48.25%
Macro-average Recall	42.41%
Macro-average F1 Score	41.00%
Micro-average Recall	51.75%
Micro-average F1 Score	51.75%

Results for TTE_df

Metric	Value
Accuracy	48.58%
Hamming	51.42%
Macro-average Recall	38.86%
Macro-average F1 Score	34.48%
Micro-average Recall	48.58%
Micro-average F1 Score	48.58%

Entrée [73]: results_regression

```
Out[73]: [ {'DataFrame': 'MSFT_df',
  'Accuracy': 0.5191666666666667,
  'Hamming': 0.4808333333333333,
  'Confusion Matrix': array([[ 19, 221, 15],
    [ 9, 584, 6],
    [ 23, 303, 20]], dtype=int64),
  'Classification Report': '              precision  recall  f1-score
support\n\n      -1.0      0.37      0.07      0.12      255\n
0.0      0.53      0.97      0.68      599\n      1.0      0.49
0.06     0.10      346\n\n      accuracy
1200\n      macro avg      0.46      0.37      0.30      1200\nweighted avg
0.48     0.52      0.40      1200\n',
  'Recall': 0.5191666666666667,
  'F1 Score': 0.39774126222924605,
  'Macro-average Recall': 0.3690905119675387,
  'Macro-average F1 Score': 0.30392784625737057,
  'Micro-average Recall': 0.5191666666666667,
  'Micro-average F1 Score': 0.5191666666666667},
 {'DataFrame': 'AMZN_df',
  'Accuracy': 0.3958333333333333,
  'Hamming': 0.6041666666666666,
  'Confusion Matrix': array([[ 36, 130, 154],
    [ 34, 208, 191],
    [ 50, 166, 231]], dtype=int64),
  'Classification Report': '              precision  recall  f1-score
support\n\n      -1.0      0.30      0.11      0.16      320\n
0.0      0.41      0.48      0.44      433\n      1.0      0.40
0.52     0.45      447\n\n      accuracy
1200\n      macro avg      0.37      0.37      0.35      1200\nweighted avg
0.38     0.40      0.37      1200\n',
  'Recall': 0.3958333333333333,
  'F1 Score': 0.37206138744834655,
  'Macro-average Recall': 0.3698826795004934,
  'Macro-average F1 Score': 0.35307312808603825,
  'Micro-average Recall': 0.3958333333333333,
  'Micro-average F1 Score': 0.3958333333333333},
 {'DataFrame': 'TSLA_df',
  'Accuracy': 0.4234769687964339,
  'Hamming': 0.5765230312035661,
  'Confusion Matrix': array([[ 37,  0, 199],
    [ 10,  0, 150],
    [ 29,  0, 248]], dtype=int64),
  'Classification Report': '              precision  recall  f1-score
support\n\n      -1.0      0.49      0.16      0.24      236\n
0.0      0.00      0.00      0.00      160\n      1.0      0.42
0.90     0.57      277\n\n      accuracy
673\n      macro avg      0.30      0.35      0.27      673\nweighted avg
0.34     0.42      0.32      673\n',
  'Recall': 0.4234769687964339,
  'F1 Score': 0.3167510306724386,
  'Macro-average Recall': 0.3506955067409084,
  'Macro-average F1 Score': 0.26822840266776193,
  'Micro-average Recall': 0.4234769687964339,
  'Micro-average F1 Score': 0.4234769687964339},
 {'DataFrame': 'RMSPA_df',
  'Accuracy': 0.4819168173598553,
  'Hamming': 0.5180831826401446,
  'Confusion Matrix': array([[ 12, 211, 16],
    [ 6, 492, 25],
    [ 8, 307, 29]], dtype=int64),
  'Classification Report': '              precision  recall  f1-score
```

```

support\n\n      -1.0      0.46      0.05      0.09      239\n
0.0      0.49      0.94      0.64      523\n      1.0      0.41
0.08     0.14      344\n\n      accuracy
1106\n      macro avg      0.45      0.36      0.29      1106\nweighted avg
0.46     0.48      0.37      1106\n',
'Recall': 0.4819168173598553,
'F1 Score': 0.3666736562071925,
'Macro-average Recall': 0.3584127026800581,
'Macro-average F1 Score': 0.2908471084564221,
'Micro-average Recall': 0.4819168173598553,
'Micro-average F1 Score': 0.4819168173598553},
{'DataFrame': 'GE_df',
'Accuracy': 0.5166666666666667,
'Hamming': 0.4833333333333334,
'Confusion Matrix': array([[ 31, 196, 67],
   [ 34, 490, 42],
   [ 37, 204, 99]], dtype=int64),
'Classification Report': '          precision    recall  f1-score
support\n\n      -1.0      0.30      0.11      0.16      294\n
0.0      0.55      0.87      0.67      566\n      1.0      0.48
0.29     0.36      340\n\n      accuracy
1200\n      macro avg      0.44      0.42      0.40      1200\nweighted avg
0.47     0.52      0.46      1200\n',
'Recall': 0.5166666666666667,
'F1 Score': 0.4581987973502573,
'Macro-average Recall': 0.4207810096948084,
'Macro-average F1 Score': 0.39698548275190615,
'Micro-average Recall': 0.5166666666666667,
'Micro-average F1 Score': 0.5166666666666667},
{'DataFrame': 'TTE_df',
'Accuracy': 0.4966666666666665,
'Hamming': 0.503333333333333,
'Confusion Matrix': array([[ 13, 241, 33],
   [ 15, 530, 27],
   [ 12, 276, 53]], dtype=int64),
'Classification Report': '          precision    recall  f1-score
support\n\n      -1.0      0.33      0.05      0.08      287\n
0.0      0.51      0.93      0.65      572\n      1.0      0.47
0.16     0.23      341\n\n      accuracy
1200\n      macro avg      0.43      0.38      0.32      1200\nweighted avg
0.45     0.50      0.40      1200\n',
'Recall': 0.4966666666666665,
'F1 Score': 0.397449242872553,
'Macro-average Recall': 0.3757649379207208,
'Macro-average F1 Score': 0.3225720061833479,
'Micro-average Recall': 0.4966666666666665,
'Micro-average F1 Score': 0.4966666666666665}]

```

```
Entrée [87]: import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Liste pour stocker les résultats de chaque modèle
df_results = [results_regression, results_rf_grid_search_hamming, result_SV]

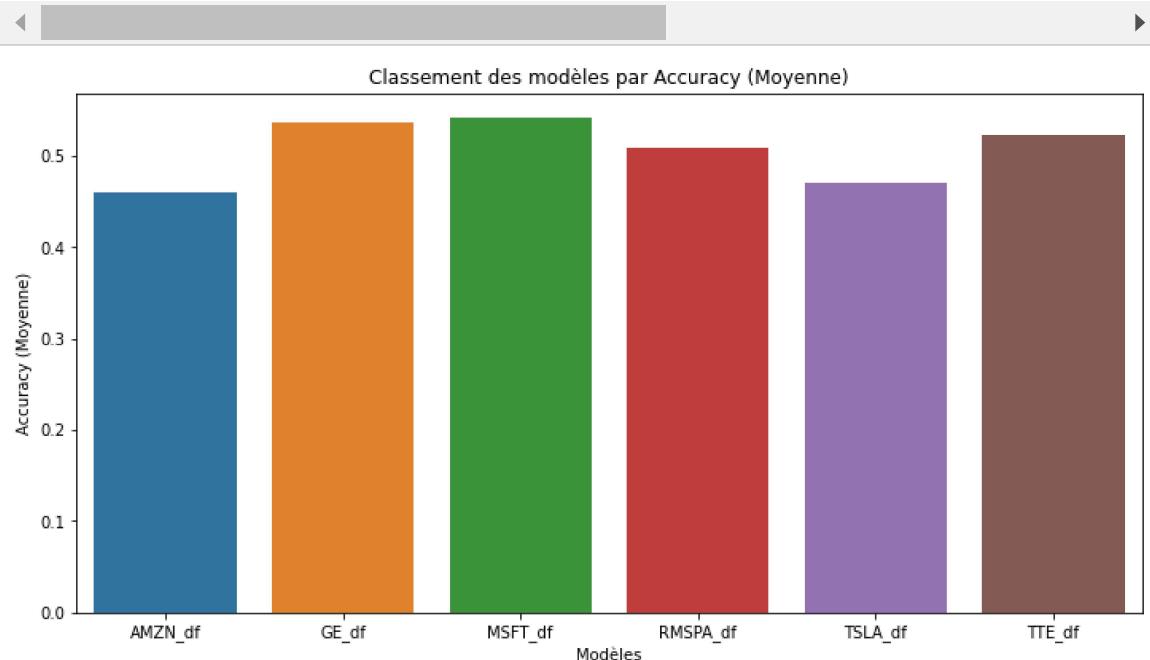
# Convertir les résultats en DataFrames
dfs = [pd.DataFrame(result) for result in df_results]

# Ajouter la colonne 'DataFrame' avec le nom du modèle à chaque DataFrame
for i, df in enumerate(dfs):
    df['DataFrame'] = f'Model_{i+1}'

# Concaténer les DataFrames en un seul
df_concatenated = pd.concat(dfs, ignore_index=True)

# Calculer la moyenne de l'Accuracy pour chaque modèle
df_model_mean = df_concatenated.groupby('DataFrame')['Accuracy'].mean().reset_index()

# Visualisation du classement
plt.figure(figsize=(12, 6))
sns.barplot(x='DataFrame', y='Accuracy', data=df_model_mean)
plt.title('Classement des modèles par Accuracy (Moyenne)')
plt.xlabel('Modèles')
plt.ylabel('Accuracy (Moyenne)')
plt.show()
```



Entrée []:

Entrée [93]: `for key in results_regression[0]:
 print(key)`

```
DataFrame  
Accuracy  
Hamming  
Confusion Matrix  
Classification Report  
Recall  
F1 Score  
Macro-average Recall  
Macro-average F1 Score  
Micro-average Recall  
Micro-average F1 Score
```

Entrée [94]: `# results_rf_grid_search_hamming, result_SVM, result_knn, results_LSTM, res`
`for key in results_rf_grid_search_hamming[0]:
 print(key)`

```
DataFrame  
Best Hyperparameters  
Accuracy  
Classification Report  
Hamming Loss  
Macro Recall  
Macro F1 Score  
Micro Recall  
Micro F1 Score
```

Entrée [96]: `for key in result_SVM[0]:
 print(key)`

```
DataFrame  
Accuracy  
Hamming  
Confusion Matrix  
Classification Report  
macro_recall  
macro_f1  
micro_recall  
micro_f1
```

Entrée [97]: `for key in result_knn[0]:
 print(key)`

```
DataFrame  
Accuracy  
Hamming  
Confusion Matrix  
Classification Report  
Macro Recall  
Macro F1  
Micro Recall  
Micro F1
```

Entrée [98]: `for key in results_LSTM[0]:
 print(key)`

```
DataFrame  
Test Accuracy  
Hamming Loss  
Micro-average Precision  
Micro-average Recall  
Micro-average F1 Score  
Macro-average Precision  
Macro-average Recall  
Macro-average F1 Score
```

Entrée [102]: `for elem in results_RNN:`

```
elem['Accuracy']=elem.pop('Test Accuracy')  
elem['Hamming']=elem.pop('Hamming Loss')  
elem['Macro Recall']=elem.pop('')  
elem['Micro Recall']=elem.pop('')  
elem['']=elem.pop('')  
elem['']=elem.pop('')  
elem['']=elem.pop('')
```

Entrée [105]: `results_RNN[0]`

```
Out[105]: {'DataFrame': 'MSFT_df',  
           'Hamming Loss': 0.5083333333333333,  
           'Micro-average Precision': 0.49166666666666664,  
           'Micro-average Recall': 0.49166666666666664,  
           'Micro-average F1 Score': 0.49166666666666664,  
           'Macro-average Precision': 0.40389206153212937,  
           'Macro-average Recall': 0.3500265240848024,  
           'Macro-average F1 Score': 0.2839277997033929,  
           'Accuracy': 0.49166667461395264}
```

Entrée [101]: `for key in results_CNN[0]:
 print(key)`

```
DataFrame  
Test Accuracy  
Hamming Loss  
Micro-average Precision  
Micro-average Recall  
Micro-average F1 Score  
Macro-average Precision  
Macro-average Recall  
Macro-average F1 Score
```

Entrée []:

10. Modèles de séquence temporelle (comme ARIMA)

Entrée []: MSFT_df=list_choice['MSFT_df']

```
# Installer la bibliothèque prophet si elle n'est pas déjà installée
# pip install prophet

import pandas as pd
from fbprophet import Prophet
from sklearn.metrics import accuracy_score, classification_report

# Supposons que MSFT_df contienne votre DataFrame avec la colonne 'Trend'
# et d'autres colonnes temporelles pertinentes.

# Renommer les colonnes pour Prophet (doit être ds et y)
prophet_df = MSFT_df[['Date', 'Trend']].rename(columns={'Date': 'ds', 'Tren

# Créer un modèle Prophet
model = Prophet()

# Adapter le modèle sur les données
model.fit(prophet_df)

# Créer un DataFrame pour les dates futures que vous souhaitez prédire
future = model.make_future_dataframe(periods=365) # Ajoutez ici le nombre

# Prédire les valeurs
forecast = model.predict(future)

# Examinez la prédiction
forecast[['ds', 'yhat', 'yhat_lower', 'yhat_upper']].tail()

# Convertir les valeurs prédites en classes (-1, 0, 1)
predicted_classes = pd.cut(forecast['yhat'], bins=[float('-inf'), -0.5, 0.5

# Supprimer les valeurs NaN (en particulier pour les dernières dates ajouté
predicted_classes = predicted_classes.dropna()

# Évaluer les performances du modèle
true_classes = MSFT_df['Trend'].astype('category').cat.codes # Convertir L
accuracy = accuracy_score(true_classes, predicted_classes)
print(f'Accuracy: {accuracy}')

# Afficher le rapport de classification
print(classification_report(true_classes, predicted_classes))
```

Entrée []: !pip install fbprophet

Entrée []:

