Final Project - Data Mining

Made by - Deepesh Jami

Dataset :

Gold Price | 10 Years | 2013-2023
Website - https://www.kaggle.com/datasets/farzadnekouei/gold-price-10-years-20132023

Algorithms Used :

- Random Forest Classifier
- Decision Trees
- Support Vector Machines

Deep Learning Algorithm Used :

- Long Short-Term Memory

Goal :

Our main goal is to really get into the history of gold prices, see what patterns we can find, and figure out what's been driving these prices. We're using some pretty neat tools for this Random Forest, Support Vector Machines, Decision Trees, and even LSTM networks.

Installing the Libraries/Dependencies :

Importing Libraries and Functions: We start by bringing in all the libraries needed. Numpy and pandas are like the bread and butter for data handling. Matplotlib is super handy for making charts. Sklearn gives us tools to split our data and create models like RandomForest, DecisionTree, and SVM. For our deep learning needs, we use TensorFlow Keras, which lets us build more complex models like LSTM networks.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.dates as mdates

from sklearn.model_selection import train_test_split, GridSearchCV, KFold, TimeSeriesSplit
from sklearn.ensemble import RandomForestClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.preprocessing import StandardScaler, MinMaxScaler
```

```
from sklearn.metrics import classification_report, accuracy_score, confusion_matrix, roc_auc_score

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout, BatchNormalization, Bidirectional
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau

from keras_tuner.tuners import RandomSearch
from tabulate import tabulate

# Installations via pip (usually placed at the beginning of a script or in a requirements.txt file for clarity and best practice)
!pip install keras-tuner --upgrade
!pip install tabulate
```

Reading the Dataset :

```
data = pd.read_csv('Gold Price Dataset.csv')
```

Initial Data Look-up :

```
data_info = data.info()
data_head = data.head()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2583 entries, 0 to 2582
Data columns (total 7 columns):
 #   Column    Non-Null Count  Dtype 
---  ------    --------------  ----- 
 0   Date      2583 non-null   object
 1   Price     2583 non-null   object
 2   Open      2583 non-null   object
 3   High      2583 non-null   object
 4   Low       2583 non-null   object
 5   Vol.      2578 non-null   object
 6   Change %  2583 non-null   object
dtypes: object(7)
memory usage: 141.4+ KB
```

Data Type Conversion and Formatting:

```
data['Date'] = pd.to_datetime(data['Date'])

def convert_price(value):
    return float(value.replace(',', ''))

for column in ['Price', 'Open', 'High', 'Low']:
    data[column] = data[column].apply(convert_price)

data['Change %'] = data['Change %'].str.rstrip('%').astype(float) / 100

def convert_volume(value):
    if pd.isna(value):
        return None
    if 'K' in value:
        return float(value.replace('K', '')) * 1000
    return float(value)

data['Vol.'] = data['Vol.'].apply(convert_volume)
```

This segment of code prepares our dataset for analysis:

- Date Format: Converts the 'Date' column to DateTime format for easier manipulation.
- Price Conversion: A function convert_price strips commas and converts price-related columns to floats for numerical analysis.
- Change Percentage: Strips the '%' from the 'Change %' column, converts it to a float, and divides by 100 to express it as a decimal.
- Volume Handling: The convert_volume function checks for missing values, translates 'K' to thousands, and ensures all volume data is numeric.

Handling Missing Data :

```
data['Vol.'] = data['Vol.'].ffill()
```

This line of code uses the `ffill()` method, which stands for "forward fill", to handle missing values in the 'Vol.' column of your dataset. It fills any missing entries with the last non-null

value found in the column before the missing one. This is a common technique in time series data to maintain continuity without introducing bias from external values.

Transforming Data :

```
data['Date'] = pd.to_datetime(data['Date'])
data.sort_values(by='Date', ascending=True, inplace=True)
```

These two lines of code are used to manipulate the 'Date' column in your dataset:

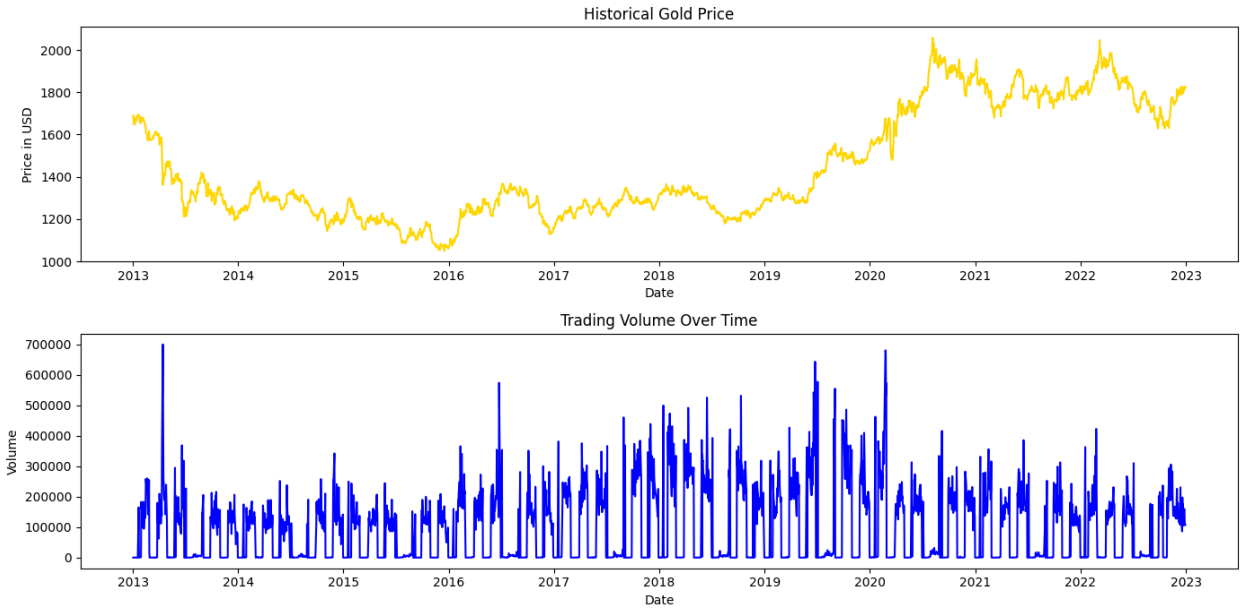
- data['Date'] = pd.to_datetime(data['Date']): This converts the 'Date' column from a string format to a datetime object, which allows for easier manipulation of dates within pandas, such as sorting, filtering, and time-based grouping.

- data.sort_values(by='Date', ascending=True, inplace=True): This sorts the dataset by the 'Date' column in ascending order, effectively organizing the data from the earliest to the latest date. The inplace=True parameter modifies the original DataFrame directly, saving the need to assign the sorted DataFrame to a new variable.

Data Visualization :

Visualisation 1 :

```
plt.figure(figsize=(14, 7))
plt.subplot(2, 1, 1)
plt.plot(data['Date'], data['Price'], label='Gold Price', color='gold')
plt.title('Historical Gold Price')
plt.xlabel('Date')
plt.ylabel('Price in USD')

plt.subplot(2, 1, 2)
plt.plot(data['Date'], data['Vol.'], label='Trading Volume', color='blue')
plt.title('Trading Volume Over Time')
plt.xlabel('Date')
plt.ylabel('Volume')

plt.tight_layout()
plt.show()
```



Historical Gold Price Chart (Top):

- This line chart illustrates the price of gold in USD over time.
- The gold price shows significant fluctuations with notable peaks and troughs. For example, there's a noticeable rise in prices around 2020, followed by periods of volatility. The trend appears to be a decline initially, stabilizing, and then showing an upward trend toward the later years.
- The visualization allows us to see how gold prices have changed, providing insights into periods of economic uncertainty or stability, which often correlate with increases or decreases in gold prices.

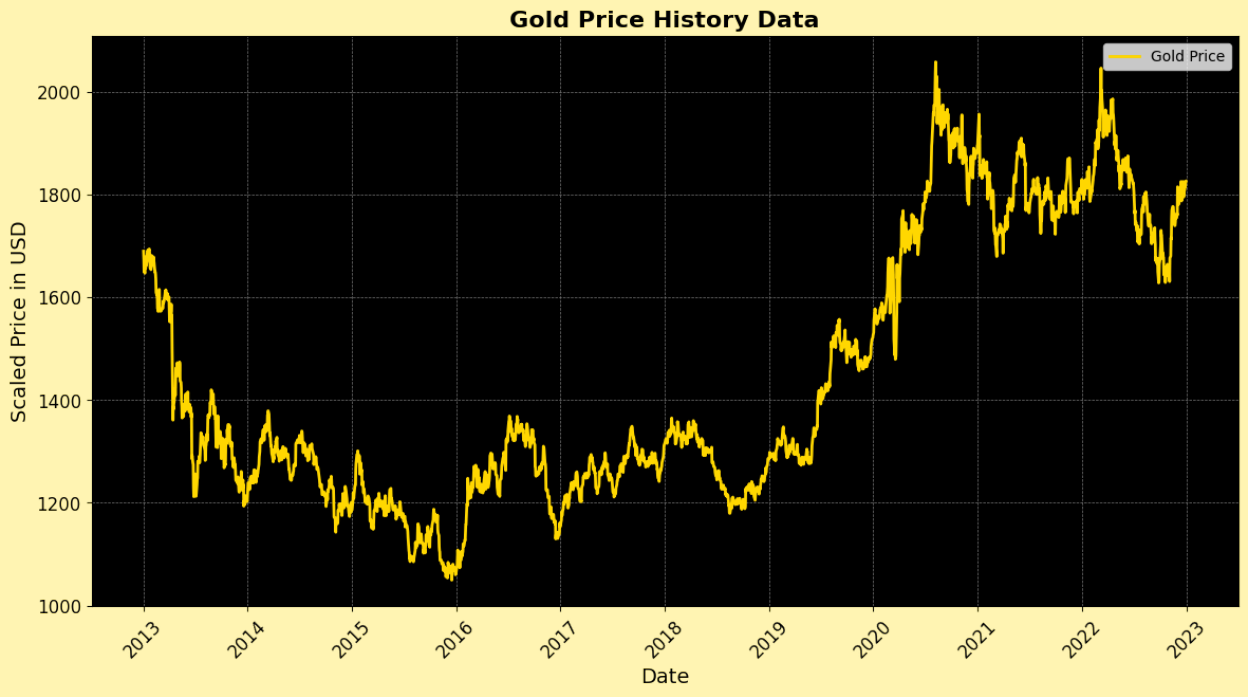
Trading Volume Over Time Chart (Bottom):

- This bar chart represents the trading volume of gold, measured in units (likely ounces or similar).
- The volume shows high variability, with some years experiencing exceptionally high trading volumes. The peaks might indicate higher market activity or significant events impacting gold trading.
- Lower bars interspersed with occasional spikes could reflect changes in market confidence or external economic events affecting liquidity and trading volume.

Together, these charts provide a comprehensive view of the gold market's dynamics, highlighting how external conditions can influence both the price and the amount of gold traded over time.

Visualisation 2 :

```
plt.figure(figsize=(14, 7))

plt.plot(data['Date'], data['Price'], label='Gold Price', color='gold',
linewidth=2)

plt.title('Gold Price History Data', fontsize=16, fontweight='bold')
plt.xlabel('Date', fontsize=14)
plt.ylabel('Scaled Price in USD', fontsize=14)

plt.grid(True, which='both', linestyle='--', linewidth=0.5, color='grey')

plt.gca().xaxis.set_major_locator(mdates.YearLocator())
plt.gca().xaxis.set_major_formatter(mdates.DateFormatter('%Y'))

plt.gca().set_facecolor('black')
plt.gca().figure.set_facecolor((255/255, 223/255, 0/255, 0.3))  # Proper RGBA
tuple
plt.xticks(fontsize=12, rotation=45)
plt.yticks(fontsize=12)

plt.legend()

plt.show()
```



The image displays a line chart titled "Gold Price History Data," which tracks the scaled price of gold in USD from 2013 to 2023:
Trend Analysis: The chart shows significant fluctuations in gold prices over the decade. Starting from 2013, prices initially see a sharp decline, reaching a low around 2015. This is followed by a period of relative stability with some minor fluctuations.

Key Movements:

- A notable increase begins around late 2018, with prices rising steeply through 2020, peaking around mid-2020. This spike could be associated with economic uncertainties, often a time when investors turn to gold as a 'safe haven.'
- After reaching this peak, prices slightly decline but then exhibit high volatility, indicating periods of economic stress or market speculation.

Recent Trends: Toward the end of the chart, there is a visible downturn in 2022 followed by a recovery phase as it heads into 2023, which might suggest a response to changing economic conditions or market reactions to global events.
This chart is valuable for analyzing how gold prices respond to various economic factors over time, providing insights into market sentiment and potential future trends.

Splitting Data For Test and Train :

```
data['Date'] = pd.to_datetime(data['Date'])

data.sort_values(by='Date', ascending=True, inplace=True)

cutoff_date = data['Date'].max() - pd.DateOffset(years=1)

train_data = data[data['Date'] <= cutoff_date]
test_data = data[data['Date'] > cutoff_date]

print(f'Training Data Size: {train_data.shape[0]}')
print(f'Testing Data Size: {test_data.shape[0]}')
```

Classification Algorithm 1 :

Random Forest Classifier

```
for lag in range(1, 3):
    data[f'Lag_{lag}'] = data['Price'].shift(lag)

data.dropna(inplace=True)

]data['Target'] = (data['Price'].shift(-1) > data['Price']).astype(int)
data.dropna(inplace=True)

]features = ['Open', 'High', 'Low', 'Vol.', 'Lag_1', 'Lag_2']
X = data[features]
y = data['Target']

]param_grid = {
    'n_estimators': [100, 200],
    'max_depth': [None, 10],
    'min_samples_split': [2, 10]
}

rf = RandomForestClassifier(random_state=42)
tscv = TimeSeriesSplit(n_splits=5)
```

```
grid_search = GridSearchCV(estimator=rf, param_grid=param_grid, cv=tscv,
scoring='accuracy', n_jobs=-1)
grid_search.fit(X, y)

best_model = grid_search.best_estimator_
print("Best model parameters:", grid_search.best_params_)
print("Best model score:", grid_search.best_score_)

y_pred = best_model.predict(X)

]print("Final Model Accuracy:", accuracy_score(y, y_pred))
print("Classification Report:")
print(classification_report(y, y_pred))

]kf = KFold(n_splits=10, shuffle=True, random_state=42)
metrics_list = []

for i, (train_index, test_index) in enumerate(kf.split(X), start=1):
    X_train, X_test = X.iloc[train_index], X.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

    best_model.fit(X_train, y_train)
    y_pred = best_model.predict(X_test)

    tn, fp, fn, tp = confusion_matrix(y_test, y_pred).ravel()
    accuracy = accuracy_score(y_test, y_pred)
    fpr = fp / (fp + tn) if (fp + tn) > 0 else 0
    fnr = fn / (fn + tp) if (fn + tp) > 0 else 0
    tss = (tp / (tp + fn)) - (fp / (fp + tn))
    hss = 2 * (tp * tn - fp * fn) / ((tp + fn) * (fn + tn) + (tp + fp) * (fp +
tn)) if ((tp + fn) * (fn + tn) + (tp + fp) * (fp + tn)) > 0 else 0

]    metrics_list.append({
        'Fold': i,
        'Accuracy': accuracy,
        'TP': tp,
        'TN': tn,
        'FP': fp,
        'FN': fn,
        'FPR': fpr,
```

```
        'FNR': fnr,
        'TSS': tss,
        'HSS': hss
    })

metrics_df = pd.DataFrame(metrics_list)
metrics_avg = metrics_df.mean(numeric_only=True).to_dict()

print("K-Fold Metrics Summary:")
print(metrics_df)

metrics_df.to_csv('RF_kfold_metrics_summary.csv', index=False)
print("K-Fold metrics saved to 'kfold_metrics_summary.csv'")

print("\nAverage Metrics across 10 folds:")
for metric, value in metrics_avg.items():
    print(f"{metric}: {value:.4f}")
```

The code snippet provided performs feature engineering, model training, and evaluation for a machine learning task using a dataset that presumably includes gold price data. Here's a breakdown of each part:

Feature Engineering:

- Lag Features: The code creates two lagged features of the 'Price' column (`Lag_1`, `Lag_2`) which represent the price values from one and two days prior, respectively. This helps in capturing trends or patterns in previous days that might influence future prices.

- Target Variable: It defines a binary target variable `Target` which indicates whether the price will increase (`1`) or not (`0`) the next day.

Data Preprocessing:

- Removes any rows with missing values after the new features are created to ensure the model trains on complete records only.

Model Training:

- Feature and Label Definition: Defines the features (`X`) used for model training, including 'Open', 'High', 'Low', 'Vol.', and the lagged prices. The labels (`y`) are the binary `Target`.
- Parameter Grid: Sets up a grid of hyperparameters for tuning the Random Forest model, including `n_estimators`, `max_depth`, and `min_samples_split`.
- Time Series Cross-Validation: Utilizes `TimeSeriesSplit` for cross-validation, which is appropriate for time series data as it respects the temporal order of observations.
- Grid Search: Conducts a grid search to find the best hyperparameters for the Random Forest model, maximizing the accuracy metric.

Model Evaluation:

- Best Model: Extracts the best Random Forest model from the grid search, displaying its parameters and score.
- Model Prediction and Report: Uses the best model to predict and generate a classification report and accuracy for the dataset, giving detailed insights into model performance.
- K-Fold Cross-Validation: Implements K-Fold cross-validation for further assessment, calculating various metrics per fold such as accuracy, false positive rate (FPR), false negative rate (FNR), true skill statistic (TSS), and Heidke skill score (HSS). These metrics provide a comprehensive view of model performance across different subsets of data.

Results and Output:

- Metrics Summary: Aggregates and prints a summary of metrics across folds and saves these metrics to a CSV file for future reference.
- Average Metrics: Calculates and displays the average of all evaluation metrics across the 10 folds to provide an overall assessment of the model's performance.

```
Best model parameters: {'max_depth': None, 'min_samples_split': 10, 'n_estimators': 200}
Best model score: 0.5075117370892018
Final Model Accuracy: 0.9804763764154627
Classification Report:
              precision    recall  f1-score   support
```

```
           0       0.98      0.98      0.98      1265
           1       0.98      0.98      0.98      1296

    accuracy                           0.98      2561
   macro avg       0.98      0.98      0.98      2561
weighted avg       0.98      0.98      0.98      2561

K-Fold Metrics Summary:
   Fold  Accuracy  TP  TN  FP  FN       FPR       FNR       TSS       HSS
0     1  0.529183  64  72  68  53  0.485714  0.452991  0.061294  0.060655
1     2  0.488281  71  54  68  63  0.557377  0.470149 -0.027526 -0.027577
2     3  0.519531  63  70  56  67  0.444444  0.515385  0.040171  0.040117
3     4  0.558594  72  71  53  60  0.427419  0.454545  0.118035  0.117834
4     5  0.460938  63  55  76  62  0.580153  0.496000 -0.076153 -0.075958
5     6  0.503906  62  67  63  64  0.484615  0.507937  0.007448  0.007449
6     7  0.523438  74  60  59  63  0.495798  0.459854  0.044348  0.044250
7     8  0.464844  60  59  66  71  0.528000  0.541985 -0.069985 -0.069921
8     9  0.523438  69  65  57  65  0.467213  0.485075  0.047712  0.047573
9    10  0.500000  74  54  72  56  0.571429  0.430769 -0.002198 -0.002202
K-Fold metrics saved to 'kfold_metrics_summary.csv'

Average Metrics across 10 folds:
Fold: 5.5000
Accuracy: 0.5072
TP: 67.2000
TN: 62.7000
FP: 63.8000
FN: 62.4000
FPR: 0.5042
FNR: 0.4815
TSS: 0.0143
HSS: 0.0142
```

Classification Algorithm 2 :

Decision Trees

```python
for lag in range(1, 5):
    data[f'Lag_{lag}'] = data['Price'].shift(lag)
data['Rolling_Mean_3'] = data['Price'].rolling(window=3).mean()
```

```
data['Rolling_Mean_7'] = data['Price'].rolling(window=7).mean()
data['Volatility_3'] = data['Price'].rolling(window=3).std()
data['Volatility_7'] = data['Price'].rolling(window=7).std()
data['Pct_Change'] = data['Price'].pct_change()

data.dropna(inplace=True)

price_diff = data['Price'].diff().shift(-1)
data['Target'] = (price_diff > 0.005).astype(int)
data.dropna(inplace=True)

features = ['Open', 'High', 'Low', 'Vol.', 'Lag_1', 'Lag_2', 'Lag_3', 'Lag_4',
            'Rolling_Mean_3', 'Rolling_Mean_7', 'Volatility_3', 'Volatility_7',
'Pct_Change']
X = data[features]
y = data['Target']

param_grid = {
    'max_depth': [5, 10, 20, 30],
    'min_samples_split': [2, 5, 10, 20],
    'min_samples_leaf': [1, 5, 10, 20],
    'min_impurity_decrease': [0.0, 0.01, 0.1],
    'criterion': ['gini', 'entropy']
}
dt = DecisionTreeClassifier(random_state=42, class_weight='balanced')
tscv = TimeSeriesSplit(n_splits=5)
grid_search = GridSearchCV(estimator=dt, param_grid=param_grid, cv=tscv,
scoring='accuracy', n_jobs=-1)
grid_search.fit(X, y)

best_model = grid_search.best_estimator_
print("Best model parameters:", grid_search.best_params_)
print("Best model score:", grid_search.best_score_)

kf = KFold(n_splits=10, shuffle=True, random_state=42)
metrics_list = []

for i, (train_index, test_index) in enumerate(kf.split(X), start=1):
    X_train, X_test = X.iloc[train_index], X.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]
```

```
    best_model.fit(X_train, y_train)
    y_pred = best_model.predict(X_test)

    tn, fp, fn, tp = confusion_matrix(y_test, y_pred).ravel()
    accuracy = accuracy_score(y_test, y_pred)
    fpr = fp / (fp + tn) if (fp + tn) > 0 else 0
    fnr = fn / (fn + tp) if (fn + tp) > 0 else 0
    tss = (tp / (tp + fn)) - (fp / (fp + tn))
     hss = 2 * (tp * tn - fp * fn) / ((tp + fn) * (fn + tn) + (tp + fp) * (fp +
tn)) if ((tp + fn) * (fn + tn) + (tp + fp) * (fp + tn)) > 0 else 0

    metrics_list.append({
        'Fold': i,
        'Accuracy': accuracy,
        'TP': tp,
        'TN': tn,
        'FP': fp,
        'FN': fn,
        'FPR': fpr,
        'FNR': fnr,
        'TSS': tss,
        'HSS': hss
    })

metrics_df = pd.DataFrame(metrics_list)
metrics_avg = metrics_df.mean(numeric_only=True).to_dict()

print("K-Fold Metrics Summary:")
print(metrics_df)

metrics_df.to_csv('DT_kfold_metrics_summary.csv', index=False)
print("K-Fold metrics saved to 'kfold_metrics_summary.csv'")

print("\nAverage Metrics across 10 folds:")
for metric, value in metrics_avg.items():
    print(f"{metric}: {value:.4f}")

y_pred = best_model.predict(X)
print("\nFinal Model Accuracy on Entire Dataset:", accuracy_score(y, y_pred))
```

```
print("\nClassification Report:")
print(classification_report(y, y_pred))
```

This code block describes an approach to preprocess financial data, engineer features, and use a Decision Tree Classifier for predicting whether gold prices will increase beyond a certain threshold the next day. Here's a detailed breakdown:

Feature Engineering:

- Lagged Features: Creates four lagged features (`Lag_1` to `Lag_4`) which capture the price of gold on previous days to include historical context in the model.
- Rolling Means: Calculates rolling means over 3 and 7 days to smooth out short-term fluctuations and highlight longer-term trends in price.
- Volatility: Computes the standard deviation over 3 and 7 days as 'Volatility_3' and 'Volatility_7', which measures the variability in price movements—a key factor in financial modeling.
- Percentage Change: Adds a feature for the percentage change in price from the previous day to capture the rate of price movement.

Target Variable:

- Target Creation: Determines if the difference in gold price for the next day is greater than 0.5% (`0.005`). This binary variable (`1` for yes, `0` for no) serves as the label for model training.

Data Preprocessing:

- Drops any rows with missing values resulting from the feature engineering steps to maintain data integrity.

Model Training:

- Feature and Label Definition: The dataset is split into features (`X`) and labels (`y`) based on the engineered columns.

- Parameter Grid: Establishes a grid for hyperparameter tuning using criteria like maximum depth, minimum samples per split, and criterion type (gini or entropy) among others, which helps in finding the optimal decision tree configuration.
- Time Series Cross-Validation: Implements **`TimeSeriesSplit`** for cross-validation, which is suited for sequential data, ensuring that validation follows temporal ordering.
- Grid Search CV: Utilizes a grid search on the decision tree model to optimize parameters based on accuracy.

Model Evaluation:

- Best Model Parameters and Score: After fitting, it prints the best model parameters and its score, providing insight into the model's effectiveness.
- K-Fold Cross-Validation: Conducts additional validation using **`KFold`** to assess model stability across multiple dataset splits, calculating metrics like accuracy, false positive rate (FPR), false negative rate (FNR), true skill statistic (TSS), and Heidke skill score (HSS).

Output and Metrics Summary:

- Generates a detailed classification report and accuracy metric for the final model on the entire dataset, offering a comprehensive view of its performance.
- Saves a CSV file with a summary of cross-validation metrics and prints an average of these metrics, facilitating an evaluation of the model's consistency and reliability.

```
Best model parameters: {'criterion': 'gini', 'max_depth': 5, 'min_impurity_decrease': 0.1,
'min_samples_leaf': 1, 'min_samples_split': 2}
Best model score: 0.5190588235294118
K-Fold Metrics Summary:
   Fold  Accuracy   TP   TN   FP   FN  FPR  FNR  TSS  HSS
0     1  0.472656    0  121    0  135  0.0  1.0  0.0  0.0
1     2  0.496094    0  127    0  129  0.0  1.0  0.0  0.0
2     3  0.507812  130    0  126    0  1.0  0.0  0.0  0.0
3     4  0.476562    0  122    0  134  0.0  1.0  0.0  0.0
4     5  0.496094  127    0  129    0  1.0  0.0  0.0  0.0
5     6  0.513725  131    0  124    0  1.0  0.0  0.0  0.0
6     7  0.529412    0  135    0  120  0.0  1.0  0.0  0.0
7     8  0.545098  139    0  116    0  1.0  0.0  0.0  0.0
8     9  0.529412    0  135    0  120  0.0  1.0  0.0  0.0
```

```
9    10  0.494118    0  126    0  129  0.0  1.0  0.0  0.0
K-Fold metrics saved to 'kfold_metrics_summary.csv'

Average Metrics across 10 folds:
Fold: 5.5000
Accuracy: 0.5061
TP: 52.7000
TN: 76.6000
FP: 49.5000
FN: 76.7000
FPR: 0.4000
FNR: 0.6000
TSS: 0.0000
HSS: 0.0000

Final Model Accuracy on Entire Dataset: 0.49354207436399217

Classification Report:
              precision    recall  f1-score   support

           0       0.49      1.00      0.66      1261
           1       0.00      0.00      0.00      1294

    accuracy                           0.49      2555
   macro avg       0.25      0.50      0.33      2555
weighted avg       0.24      0.49      0.33      2555

c:\Users\seera\AppData\Local\Programs\Python\Python311\Lib\site-packages\sklearn\metrics\_cla
ssification.py:1471: UndefinedMetricWarning: Precision and F-score are ill-defined and being
set to 0.0 in labels with no predicted samples. Use `zero_division` parameter to control this
behavior.
  _warn_prf(average, modifier, msg_start, len(result))
c:\Users\seera\AppData\Local\Programs\Python\Python311\Lib\site-packages\sklearn\metrics\_cla
ssification.py:1471: UndefinedMetricWarning: Precision and F-score are ill-defined and being
set to 0.0 in labels with no predicted samples. Use `zero_division` parameter to control this
behavior.
  _warn_prf(average, modifier, msg_start, len(result))
c:\Users\seera\AppData\Local\Programs\Python\Python311\Lib\site-packages\sklearn\metrics\_cla
ssification.py:1471: UndefinedMetricWarning: Precision and F-score are ill-defined and being
set to 0.0 in labels with no predicted samples. Use `zero_division` parameter to control this
behavior.
  _warn_prf(average, modifier, msg_start, len(result))
```

Classification Algorithm 3 :

Support Vector Machines

```
for lag in range(1, 5):
    data[f'Lag_{lag}'] = data['Price'].shift(lag)
data['Rolling_Mean_3'] = data['Price'].rolling(window=3).mean()
data['Rolling_Mean_7'] = data['Price'].rolling(window=7).mean()
data['Volatility_3'] = data['Price'].rolling(window=3).std()
data['Volatility_7'] = data['Price'].rolling(window=7).std()
data['Pct_Change'] = data['Price'].pct_change()

data.dropna(inplace=True)

price_diff = data['Price'].diff().shift(-1)
data['Target'] = (price_diff > 0.005).astype(int)
data.dropna(inplace=True)

features = ['Open', 'High', 'Low', 'Vol.', 'Lag_1', 'Lag_2', 'Lag_3', 'Lag_4',
            'Rolling_Mean_3', 'Rolling_Mean_7', 'Volatility_3', 'Volatility_7',
'Pct_Change']
X = data[features]
y = data['Target']

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

param_grid = {
    'C': [0.1, 1, 10],
    'gamma': ['scale', 0.1, 0.01],
    'kernel': ['rbf', 'linear']
}
svm = SVC(random_state=42)
kf = KFold(n_splits=10, shuffle=True, random_state=42)
grid_search    =    GridSearchCV(estimator=svm,    param_grid=param_grid,    cv=kf,
scoring='accuracy', n_jobs=-1)
grid_search.fit(X_scaled, y)

best_model = grid_search.best_estimator_
print("Best model parameters:", grid_search.best_params_)
print("Best model score:", grid_search.best_score_)
```

```
metrics_list = []

for i, (train_index, test_index) in enumerate(kf.split(X_scaled), start=1):
    X_train, X_test = X_scaled[train_index], X_scaled[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

    best_model.fit(X_train, y_train)
    y_pred = best_model.predict(X_test)

    tn, fp, fn, tp = confusion_matrix(y_test, y_pred).ravel()
    accuracy = accuracy_score(y_test, y_pred)
    fpr = fp / (fp + tn) if (fp + tn) > 0 else 0
    fnr = fn / (fn + tp) if (fn + tp) > 0 else 0
    tss = (tp / (tp + fn)) - (fp / (fp + tn))
    hss = 2 * (tp * tn - fp * fn) / ((tp + fn) * (fn + tn) + (tp + fp) * (fp +
tn)) if ((tp + fn) * (fn + tn) + (tp + fp) * (fp + tn)) > 0 else 0

    metrics_list.append({
        'Fold': i,
        'Accuracy': accuracy,
        'TP': tp,
        'TN': tn,
        'FP': fp,
        'FN': fn,
        'FPR': fpr,
        'FNR': fnr,
        'TSS': tss,
        'HSS': hss
    })

metrics_df = pd.DataFrame(metrics_list)
metrics_avg = metrics_df.mean(numeric_only=True).to_dict()

print("K-Fold Metrics Summary:")
print(metrics_df)

metrics_df.to_csv('SVM_kfold_metrics_summary.csv', index=False)
print("K-Fold metrics saved to 'kfold_metrics_summary.csv'")
```

```
print("\nAverage Metrics across 10 folds:")
for metric, value in metrics_avg.items():
    print(f"{metric}: {value:.4f}")

y_pred = best_model.predict(X_scaled)
print("\nFinal Model Accuracy on Entire Dataset:", accuracy_score(y, y_pred))
print("\nClassification Report:")
print(classification_report(y, y_pred))
```

This code block provides a comprehensive process for predicting whether gold prices will increase by more than 0.5% the next day using a Support Vector Machine (SVM) model. Below is a detailed breakdown of each section:

Feature Scaling:

- Standardization: Scales the features using **`StandardScaler`** to ensure that the SVM model, which is sensitive to the scale of the input features, performs optimally.

Model Training:

- Parameter Tuning: Uses **`GridSearchCV`** with **`KFold`** cross-validation to find the best hyperparameters (**`C`**, **`gamma`**, **`kernel`**) for the SVM model. This method ensures that the model's parameters are tuned to improve prediction accuracy.
- SVM Training: Trains the SVM model on scaled data using the best parameters identified.

Model Evaluation:

- Cross-Validation: Implements **`KFold`** cross-validation to evaluate model performance across different subsets of the dataset, ensuring that the results are robust and not dependent on a particular partition of the data.
- Performance Metrics: Calculates a range of metrics for each fold, including accuracy, true positive rate (TPR), false positive rate (FPR), true skill statistic (TSS), and Heidke skill score (HSS), providing a thorough assessment of model performance.

Results Output:

- Summary of Metrics: Aggregates and displays the metrics from each fold, providing insights into model consistency and effectiveness.
- Save Metrics: Outputs the metrics to a CSV file (**`SVM_kfold_metrics_summary.csv`**), allowing for further analysis or reporting.

- Average Metrics: Computes and prints the average of all metrics across the 10 folds to give an overall evaluation of model performance.
- Final Model Accuracy and Classification Report: Outputs the accuracy of the model on the entire dataset and a detailed classification report, offering a deep dive into the precision, recall, and F1-score of the model predictions.

```
Best model parameters: {'C': 10, 'gamma': 0.1, 'kernel': 'rbf'}
Best model score: 0.5182461015902423
K-Fold Metrics Summary:
   Fold  Accuracy  TP  TN  FP  FN       FPR       FNR       TSS       HSS
0     1  0.494118  64  62  62  67  0.500000  0.511450 -0.011450 -0.011438
1     2  0.501961  71  57  78  49  0.577778  0.408333  0.013889  0.013705
2     3  0.525490  71  63  73  48  0.536765  0.403361  0.059874  0.059098
3     4  0.470588  62  58  61  74  0.512605  0.544118 -0.056723 -0.056338
4     5  0.509804  57  73  46  79  0.386555  0.580882  0.032563  0.032008
5     6  0.545098  75  64  67  49  0.511450  0.395161  0.093388  0.093028
6     7  0.549020  75  65  55  60  0.458333  0.444444  0.097222  0.096998
7     8  0.541176  73  65  51  66  0.439655  0.474820  0.085525  0.084619
8     9  0.517647  61  71  58  65  0.449612  0.515873  0.034515  0.034537
9    10  0.527559  70  64  63  57  0.496063  0.448819  0.055118  0.055118
K-Fold metrics saved to 'kfold_metrics_summary.csv'

Average Metrics across 10 folds:
Fold: 5.5000
Accuracy: 0.5182
TP: 67.9000
TN: 64.2000
FP: 61.4000
FN: 61.4000
FPR: 0.4869
FNR: 0.4727
TSS: 0.0404
HSS: 0.0401

Final Model Accuracy on Entire Dataset: 0.6108277755982738

Classification Report:
              precision    recall  f1-score   support

           0       0.61      0.59      0.60      1256
           1       0.61      0.63      0.62      1293

    accuracy                           0.61      2549
```

```
   macro avg       0.61      0.61      0.61      2549
weighted avg       0.61      0.61      0.61      2549
```

Deep Learning Algorithm :

Long Short-Term Memory

```
for lag in range(1, 5):
    data[f'Lag_{lag}'] = data['Price'].shift(lag)
data['Rolling_Mean_3'] = data['Price'].rolling(window=3).mean()
data['Rolling_Mean_7'] = data['Price'].rolling(window=7).mean()
data['Volatility_3'] = data['Price'].rolling(window=3).std()
data['Volatility_7'] = data['Price'].rolling(window=7).std()
data.dropna(inplace=True)

scaler = MinMaxScaler()
scaled_data   =   scaler.fit_transform(data[['Price',   'Open',   'High',   'Low',
'Vol.', 'Lag_1', 'Lag_2',
                                            'Lag_3', 'Lag_4', 'Rolling_Mean_3',
'Rolling_Mean_7',
                                        'Volatility_3', 'Volatility_7']])

sequence_length = 10
X, y = [], []
for i in range(sequence_length, len(scaled_data)):
    X.append(scaled_data[i-sequence_length:i])
    y.append(data['Price'].iloc[i] > data['Price'].iloc[i-1])
X, y = np.array(X), np.array(y).astype(int)

def build_model(hp):
    model = Sequential()
    model.add(Bidirectional(LSTM(
        units=hp.Int('units', min_value=32, max_value=128, step=32),
        return_sequences=True),
        input_shape=(X.shape[1], X.shape[2])))
    model.add(BatchNormalization())
         model.add(Dropout(hp.Float('dropout_1',  min_value=0.1,  max_value=0.5,
step=0.1)))
```

```
    model.add(LSTM(units=hp.Int('units', min_value=32, max_value=128, step=32),
return_sequences=False))
        model.add(Dropout(hp.Float('dropout_2', min_value=0.1, max_value=0.5,
step=0.1)))
      model.add(Dense(units=hp.Int('dense_units', min_value=16, max_value=64,
step=16), activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
                model.compile(optimizer='adam',    loss='binary_crossentropy',
metrics=['accuracy'])
    return model

tuner = RandomSearch(
    build_model,
    objective='val_accuracy',
    max_trials=10,
    executions_per_trial=1,
    directory='lstm_tuning',
    project_name='gold_price_lstm')

kf = KFold(n_splits=10, shuffle=True, random_state=42)
metrics_list = []

for fold, (train_index, test_index) in enumerate(kf.split(X), start=1):
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]

    tuner.search(X_train, y_train, epochs=10, validation_data=(X_test, y_test),
batch_size=32, verbose=1)
    best_hps = tuner.get_best_hyperparameters(num_trials=1)[0]
    model = tuner.hypermodel.build(best_hps)

          early_stopping = EarlyStopping(monitor='val_loss', patience=5,
restore_best_weights=True)
     reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=3,
verbose=1)

        history = model.fit(X_train, y_train, epochs=20, batch_size=32,
validation_data=(X_test, y_test),
                        callbacks=[early_stopping, reduce_lr], verbose=1)
```

```
    y_pred_proba = model.predict(X_test)
    y_pred = (y_pred_proba > 0.5).astype(int)

    tn, fp, fn, tp = confusion_matrix(y_test, y_pred).ravel()
    accuracy = accuracy_score(y_test, y_pred)
    fpr = fp / (fp + tn) if (fp + tn) > 0 else 0
    fnr = fn / (fn + tp) if (fn + tp) > 0 else 0
    tss = (tp / (tp + fn)) - (fp / (fp + tn))
    hss = 2 * (tp * tn - fp * fn) / ((tp + fn) * (fn + tn) + (tp + fp) * (fp +
tn)) if ((tp + fn) * (fn + tn) + (tp + fp) * (fp + tn)) > 0 else 0

    metrics_list.append({
        'Fold': fold,
        'Accuracy': accuracy,
        'TP': tp,
        'TN': tn,
        'FP': fp,
        'FN': fn,
        'FPR': fpr,
        'FNR': fnr,
        'TSS': tss,
        'HSS': hss
    })

metrics_df = pd.DataFrame(metrics_list)
metrics_avg = metrics_df.mean(numeric_only=True).to_dict()

print("K-Fold Metrics Summary:")
print(metrics_df)

metrics_df.to_csv('LSTM_kfold_metrics_summary.csv', index=False)
print("\nMetrics saved to 'kfold_metrics_lstm.csv'")

print("\nAverage Metrics across 10 folds:")
for metric, value in metrics_avg.items():
    print(f"{metric}: {value:.4f}")
```

This code provides a comprehensive approach to modeling gold price movements using a LSTM (Long Short-Term Memory) network, with a focus on preprocessing, feature engineering, model building, tuning, and evaluation. Here's a detailed walkthrough:

LSTM Model Building:

- Hyperparameter Tuning: Uses `RandomSearch` from Keras Tuner to optimize LSTM architecture parameters such as number of units, dropout rate, and dense layer size.
- Model Architecture: Constructs a LSTM model with bidirectional layers, batch normalization, dropout for regularization, and dense layers for output.

Model Training and Evaluation:

- Cross-Validation Setup: Uses `KFold` cross-validation with data shuffling to assess model performance across different subsets of the dataset.
- Training and Hyperparameter Search: For each fold, performs a search for the best hyperparameters and then trains the LSTM model using these parameters.
- Callbacks: Implements `EarlyStopping` to halt training if validation loss doesn't improve, and `ReduceLROnPlateau` to reduce learning rate when validation loss plateaus, optimizing the training process.

Performance Metrics:

- Detailed Metrics: After training, computes a suite of metrics including accuracy, true positives, false positives, true negatives, false negatives, false positive rate (FPR), false negative rate (FNR), true skill statistic (TSS), and Heidke skill score (HSS).
- Metrics Summary: Collects metrics for each fold into a DataFrame, calculates averages, and saves to a CSV file for detailed analysis.

Outputs:

- K-Fold Metrics: Prints and saves a summary of metrics for each fold, providing insights into the consistency and effectiveness of the model across different partitions of the data.
- Average Performance: Displays the average performance metrics across all folds, offering a holistic view of model reliability and accuracy.

```
Reloading Tuner from lstm_tuning\gold_price_lstm\tuner0.json
Epoch 1/20
72/72 [==============================] - 11s 41ms/step - loss: 0.6998 - accuracy: 0.5046 -
val_loss: 0.6932 - val_accuracy: 0.5118 - lr: 0.0010
Epoch 2/20
72/72 [==============================] - 1s 20ms/step - loss: 0.6938 - accuracy: 0.5055 -
val_loss: 0.6934 - val_accuracy: 0.4803 - lr: 0.0010
Epoch 3/20
```

```
72/72 [==============================] - 1s 20ms/step - loss: 0.6923 - accuracy: 0.5265 -
val_loss: 0.6933 - val_accuracy: 0.4567 - lr: 0.0010
Epoch 4/20
71/72 [============================>.] - ETA: 0s - loss: 0.6942 - accuracy: 0.5123
Epoch 4: ReduceLROnPlateau reducing learning rate to 0.0005000000237487257.
72/72 [==============================] - 1s 21ms/step - loss: 0.6942 - accuracy: 0.5121 -
val_loss: 0.6936 - val_accuracy: 0.4764 - lr: 0.0010
Epoch 5/20
72/72 [==============================] - 1s 20ms/step - loss: 0.6919 - accuracy: 0.5191 -
val_loss: 0.6934 - val_accuracy: 0.4843 - lr: 5.0000e-04
Epoch 6/20
72/72 [==============================] - 2s 21ms/step - loss: 0.6911 - accuracy: 0.5323 -
val_loss: 0.6937 - val_accuracy: 0.4961 - lr: 5.0000e-04
8/8 [==============================] - 1s 8ms/step
Epoch 1/20
72/72 [==============================] - 10s 41ms/step - loss: 0.7042 - accuracy: 0.4906 -
val_loss: 0.6908 - val_accuracy: 0.5433 - lr: 0.0010
Epoch 2/20
72/72 [==============================] - 1s 20ms/step - loss: 0.6966 - accuracy: 0.5129 -
val_loss: 0.6902 - val_accuracy: 0.5276 - lr: 0.0010
Epoch 3/20
72/72 [==============================] - 2s 21ms/step - loss: 0.6933 - accuracy: 0.5143 -
val_loss: 0.6904 - val_accuracy: 0.5472 - lr: 0.0010
Epoch 4/20
72/72 [==============================] - 1s 19ms/step - loss: 0.6952 - accuracy: 0.5011 -
val_loss: 0.6908 - val_accuracy: 0.5118 - lr: 0.0010
Epoch 5/20
71/72 [============================>.] - ETA: 0s - loss: 0.6943 - accuracy: 0.5154
Epoch 5: ReduceLROnPlateau reducing learning rate to 0.0005000000237487257.
72/72 [==============================] - 1s 20ms/step - loss: 0.6943 - accuracy: 0.5147 -
val_loss: 0.6904 - val_accuracy: 0.5394 - lr: 0.0010
Epoch 6/20
72/72 [==============================] - 1s 20ms/step - loss: 0.6931 - accuracy: 0.5138 -
val_loss: 0.6904 - val_accuracy: 0.5709 - lr: 5.0000e-04
Epoch 7/20
72/72 [==============================] - 1s 20ms/step - loss: 0.6943 - accuracy: 0.5129 -
val_loss: 0.6896 - val_accuracy: 0.5551 - lr: 5.0000e-04
Epoch 8/20
72/72 [==============================] - 1s 20ms/step - loss: 0.6915 - accuracy: 0.5191 -
val_loss: 0.6908 - val_accuracy: 0.5197 - lr: 5.0000e-04
Epoch 9/20
72/72 [==============================] - 2s 21ms/step - loss: 0.6933 - accuracy: 0.5134 -
val_loss: 0.6891 - val_accuracy: 0.5591 - lr: 5.0000e-04
Epoch 10/20
```

```
72/72 [==============================] - 1s 20ms/step - loss: 0.6919 - accuracy: 0.5252 -
val_loss: 0.6895 - val_accuracy: 0.5512 - lr: 5.0000e-04
Epoch 11/20
72/72 [==============================] - 1s 20ms/step - loss: 0.6925 - accuracy: 0.5222 -
val_loss: 0.6897 - val_accuracy: 0.5315 - lr: 5.0000e-04
Epoch 12/20
72/72 [==============================] - 1s 20ms/step - loss: 0.6921 - accuracy: 0.5270 -
val_loss: 0.6885 - val_accuracy: 0.5551 - lr: 5.0000e-04
Epoch 13/20
72/72 [==============================] - 1s 20ms/step - loss: 0.6935 - accuracy: 0.5165 -
val_loss: 0.6872 - val_accuracy: 0.5472 - lr: 5.0000e-04
Epoch 14/20
72/72 [==============================] - 2s 21ms/step - loss: 0.6921 - accuracy: 0.5147 -
val_loss: 0.6893 - val_accuracy: 0.5472 - lr: 5.0000e-04
Epoch 15/20
72/72 [==============================] - 1s 20ms/step - loss: 0.6937 - accuracy: 0.5217 -
val_loss: 0.6894 - val_accuracy: 0.5512 - lr: 5.0000e-04
Epoch 16/20
71/72 [============================>.] - ETA: 0s - loss: 0.6916 - accuracy: 0.5264
Epoch 16: ReduceLROnPlateau reducing learning rate to 0.0002500000118743628.
72/72 [==============================] - 2s 23ms/step - loss: 0.6918 - accuracy: 0.5257 -
val_loss: 0.6895 - val_accuracy: 0.5197 - lr: 5.0000e-04
Epoch 17/20
72/72 [==============================] - 2s 21ms/step - loss: 0.6918 - accuracy: 0.5230 -
val_loss: 0.6901 - val_accuracy: 0.5157 - lr: 2.5000e-04
Epoch 18/20
72/72 [==============================] - 2s 21ms/step - loss: 0.6914 - accuracy: 0.5257 -
val_loss: 0.6883 - val_accuracy: 0.5512 - lr: 2.5000e-04
8/8 [==============================] - 2s 10ms/step
Epoch 1/20
72/72 [==============================] - 10s 41ms/step - loss: 0.7028 - accuracy: 0.4923 -
val_loss: 0.6922 - val_accuracy: 0.5630 - lr: 0.0010
Epoch 2/20
72/72 [==============================] - 1s 20ms/step - loss: 0.6953 - accuracy: 0.5112 -
val_loss: 0.6962 - val_accuracy: 0.4843 - lr: 0.0010
Epoch 3/20
72/72 [==============================] - 1s 20ms/step - loss: 0.6952 - accuracy: 0.5055 -
val_loss: 0.6928 - val_accuracy: 0.4921 - lr: 0.0010
Epoch 4/20
71/72 [============================>.] - ETA: 0s - loss: 0.6952 - accuracy: 0.5180
Epoch 4: ReduceLROnPlateau reducing learning rate to 0.0005000000237487257.
72/72 [==============================] - 1s 21ms/step - loss: 0.6953 - accuracy: 0.5182 -
val_loss: 0.7000 - val_accuracy: 0.4331 - lr: 0.0010
Epoch 5/20
```

```
72/72 [==============================] - 1s 20ms/step - loss: 0.6927 - accuracy: 0.5002 -
val_loss: 0.6924 - val_accuracy: 0.5394 - lr: 5.0000e-04
Epoch 6/20
72/72 [==============================] - 1s 20ms/step - loss: 0.6918 - accuracy: 0.5186 -
val_loss: 0.6909 - val_accuracy: 0.5551 - lr: 5.0000e-04
Epoch 7/20
72/72 [==============================] - 1s 20ms/step - loss: 0.6920 - accuracy: 0.5270 -
val_loss: 0.6886 - val_accuracy: 0.5512 - lr: 5.0000e-04
Epoch 8/20
72/72 [==============================] - 1s 20ms/step - loss: 0.6916 - accuracy: 0.5261 -
val_loss: 0.6897 - val_accuracy: 0.5433 - lr: 5.0000e-04
Epoch 9/20
72/72 [==============================] - 2s 22ms/step - loss: 0.6919 - accuracy: 0.5305 -
val_loss: 0.6930 - val_accuracy: 0.5433 - lr: 5.0000e-04
Epoch 10/20
70/72 [============================>.] - ETA: 0s - loss: 0.6927 - accuracy: 0.5201
Epoch 10: ReduceLROnPlateau reducing learning rate to 0.00025000000118743628.
72/72 [==============================] - 2s 22ms/step - loss: 0.6927 - accuracy: 0.5213 -
val_loss: 0.6900 - val_accuracy: 0.5433 - lr: 5.0000e-04
Epoch 11/20
72/72 [==============================] - 2s 22ms/step - loss: 0.6909 - accuracy: 0.5217 -
val_loss: 0.6891 - val_accuracy: 0.5551 - lr: 2.5000e-04
Epoch 12/20
72/72 [==============================] - 1s 20ms/step - loss: 0.6905 - accuracy: 0.5331 -
val_loss: 0.6907 - val_accuracy: 0.5512 - lr: 2.5000e-04
8/8 [==============================] - 1s 8ms/step
Epoch 1/20
72/72 [==============================] - 11s 41ms/step - loss: 0.7004 - accuracy: 0.5202 -
val_loss: 0.6919 - val_accuracy: 0.5534 - lr: 0.0010
Epoch 2/20
72/72 [==============================] - 1s 20ms/step - loss: 0.6979 - accuracy: 0.5167 -
val_loss: 0.6911 - val_accuracy: 0.5336 - lr: 0.0010
Epoch 3/20
72/72 [==============================] - 1s 20ms/step - loss: 0.6932 - accuracy: 0.5088 -
val_loss: 0.6914 - val_accuracy: 0.5336 - lr: 0.0010
Epoch 4/20
72/72 [==============================] - 1s 20ms/step - loss: 0.6929 - accuracy: 0.5311 -
val_loss: 0.6908 - val_accuracy: 0.5415 - lr: 0.0010
Epoch 5/20
72/72 [==============================] - 2s 22ms/step - loss: 0.6932 - accuracy: 0.5180 -
val_loss: 0.6931 - val_accuracy: 0.5178 - lr: 0.0010
Epoch 6/20
72/72 [==============================] - 2s 21ms/step - loss: 0.6940 - accuracy: 0.5158 -
val_loss: 0.6946 - val_accuracy: 0.4822 - lr: 0.0010
```

```
Epoch 7/20
71/72 [============================>.] - ETA: 0s - loss: 0.6919 - accuracy: 0.5286
Epoch 7: ReduceLROnPlateau reducing learning rate to 0.0005000000237487257.
72/72 [==============================] - 2s 22ms/step - loss: 0.6919 - accuracy: 0.5289 -
val_loss: 0.6992 - val_accuracy: 0.4822 - lr: 0.0010
Epoch 8/20
72/72 [==============================] - 2s 21ms/step - loss: 0.6939 - accuracy: 0.5211 -
val_loss: 0.6985 - val_accuracy: 0.4625 - lr: 5.0000e-04
Epoch 9/20
72/72 [==============================] - 2s 22ms/step - loss: 0.6908 - accuracy: 0.5272 -
val_loss: 0.6987 - val_accuracy: 0.4901 - lr: 5.0000e-04
8/8 [==============================] - 2s 11ms/step
Epoch 1/20
72/72 [==============================] - 10s 42ms/step - loss: 0.6988 - accuracy: 0.4904 -
val_loss: 0.6928 - val_accuracy: 0.5375 - lr: 0.0010
Epoch 2/20
72/72 [==============================] - 1s 20ms/step - loss: 0.6939 - accuracy: 0.5053 -
val_loss: 0.6932 - val_accuracy: 0.4941 - lr: 0.0010
Epoch 3/20
72/72 [==============================] - 1s 19ms/step - loss: 0.6923 - accuracy: 0.5250 -
val_loss: 0.6936 - val_accuracy: 0.5257 - lr: 0.0010
Epoch 4/20
72/72 [==============================] - ETA: 0s - loss: 0.6970 - accuracy: 0.5044
Epoch 4: ReduceLROnPlateau reducing learning rate to 0.0005000000237487257.
72/72 [==============================] - 2s 21ms/step - loss: 0.6970 - accuracy: 0.5044 -
val_loss: 0.6937 - val_accuracy: 0.5257 - lr: 0.0010
Epoch 5/20
72/72 [==============================] - 1s 20ms/step - loss: 0.6924 - accuracy: 0.5250 -
val_loss: 0.6949 - val_accuracy: 0.5059 - lr: 5.0000e-04
Epoch 6/20
72/72 [==============================] - 1s 20ms/step - loss: 0.6918 - accuracy: 0.5184 -
val_loss: 0.6930 - val_accuracy: 0.5336 - lr: 5.0000e-04
8/8 [==============================] - 1s 7ms/step
Epoch 1/20
72/72 [==============================] - 10s 42ms/step - loss: 0.7006 - accuracy: 0.4982 -
val_loss: 0.6971 - val_accuracy: 0.4506 - lr: 0.0010
Epoch 2/20
72/72 [==============================] - 1s 19ms/step - loss: 0.6954 - accuracy: 0.5031 -
val_loss: 0.6948 - val_accuracy: 0.4743 - lr: 0.0010
Epoch 3/20
72/72 [==============================] - 1s 19ms/step - loss: 0.6976 - accuracy: 0.5154 -
val_loss: 0.6945 - val_accuracy: 0.4941 - lr: 0.0010
Epoch 4/20
```

```
72/72 [==============================] - 1s 19ms/step - loss: 0.6953 - accuracy: 0.4996 -
val_loss: 0.6929 - val_accuracy: 0.5415 - lr: 0.0010
Epoch 5/20
72/72 [==============================] - 1s 20ms/step - loss: 0.6950 - accuracy: 0.5000 -
val_loss: 0.7010 - val_accuracy: 0.4625 - lr: 0.0010
Epoch 6/20
72/72 [==============================] - 1s 19ms/step - loss: 0.6933 - accuracy: 0.5237 -
val_loss: 0.6948 - val_accuracy: 0.5178 - lr: 0.0010
Epoch 7/20
72/72 [==============================] - 1s 19ms/step - loss: 0.6914 - accuracy: 0.5237 -
val_loss: 0.6925 - val_accuracy: 0.5336 - lr: 0.0010
Epoch 8/20
72/72 [==============================] - 1s 19ms/step - loss: 0.6931 - accuracy: 0.5215 -
val_loss: 0.6952 - val_accuracy: 0.4822 - lr: 0.0010
Epoch 9/20
72/72 [==============================] - 1s 20ms/step - loss: 0.6917 - accuracy: 0.5307 -
val_loss: 0.6980 - val_accuracy: 0.4941 - lr: 0.0010
Epoch 10/20
71/72 [============================>.] - ETA: 0s - loss: 0.6914 - accuracy: 0.5312
Epoch 10: ReduceLROnPlateau reducing learning rate to 0.0005000000237487257.
72/72 [==============================] - 1s 19ms/step - loss: 0.6915 - accuracy: 0.5307 -
val_loss: 0.7007 - val_accuracy: 0.4901 - lr: 0.0010
Epoch 11/20
72/72 [==============================] - 1s 19ms/step - loss: 0.6930 - accuracy: 0.5246 -
val_loss: 0.6965 - val_accuracy: 0.5059 - lr: 5.0000e-04
Epoch 12/20
72/72 [==============================] - 1s 19ms/step - loss: 0.6931 - accuracy: 0.5075 -
val_loss: 0.6972 - val_accuracy: 0.5217 - lr: 5.0000e-04
8/8 [==============================] - 1s 8ms/step
Epoch 1/20
72/72 [==============================] - 11s 52ms/step - loss: 0.6998 - accuracy: 0.4925 -
val_loss: 0.6954 - val_accuracy: 0.4743 - lr: 0.0010
Epoch 2/20
72/72 [==============================] - 1s 21ms/step - loss: 0.6951 - accuracy: 0.5136 -
val_loss: 0.6949 - val_accuracy: 0.4822 - lr: 0.0010
Epoch 3/20
72/72 [==============================] - 1s 20ms/step - loss: 0.6964 - accuracy: 0.5061 -
val_loss: 0.6949 - val_accuracy: 0.4941 - lr: 0.0010
Epoch 4/20
72/72 [==============================] - 2s 21ms/step - loss: 0.6947 - accuracy: 0.5237 -
val_loss: 0.6959 - val_accuracy: 0.4743 - lr: 0.0010
Epoch 5/20
71/72 [============================>.] - ETA: 0s - loss: 0.6948 - accuracy: 0.5255
Epoch 5: ReduceLROnPlateau reducing learning rate to 0.0005000000237487257.
```

```
72/72 [==============================] - 1s 20ms/step - loss: 0.6949 - accuracy: 0.5254 -
val_loss: 0.6969 - val_accuracy: 0.4783 - lr: 0.0010
Epoch 6/20
72/72 [==============================] - 2s 22ms/step - loss: 0.6915 - accuracy: 0.5268 -
val_loss: 0.6997 - val_accuracy: 0.4506 - lr: 5.0000e-04
Epoch 7/20
72/72 [==============================] - 1s 20ms/step - loss: 0.6913 - accuracy: 0.5355 -
val_loss: 0.7018 - val_accuracy: 0.4427 - lr: 5.0000e-04
Epoch 8/20
70/72 [============================>.] - ETA: 0s - loss: 0.6916 - accuracy: 0.5317
Epoch 8: ReduceLROnPlateau reducing learning rate to 0.0002500000118743628.
72/72 [==============================] - 1s 20ms/step - loss: 0.6917 - accuracy: 0.5311 -
val_loss: 0.7029 - val_accuracy: 0.4466 - lr: 5.0000e-04
8/8 [==============================] - 1s 8ms/step
Epoch 1/20
72/72 [==============================] - 10s 40ms/step - loss: 0.7048 - accuracy: 0.5123 -
val_loss: 0.6936 - val_accuracy: 0.4704 - lr: 0.0010
Epoch 2/20
72/72 [==============================] - 1s 20ms/step - loss: 0.6973 - accuracy: 0.5079 -
val_loss: 0.6940 - val_accuracy: 0.4625 - lr: 0.0010
Epoch 3/20
72/72 [==============================] - 1s 20ms/step - loss: 0.6963 - accuracy: 0.5035 -
val_loss: 0.6923 - val_accuracy: 0.5059 - lr: 0.0010
Epoch 4/20
72/72 [==============================] - 1s 20ms/step - loss: 0.6962 - accuracy: 0.5224 -
val_loss: 0.6962 - val_accuracy: 0.4941 - lr: 0.0010
Epoch 5/20
72/72 [==============================] - 1s 19ms/step - loss: 0.6961 - accuracy: 0.4904 -
val_loss: 0.7003 - val_accuracy: 0.4783 - lr: 0.0010
Epoch 6/20
70/72 [============================>.] - ETA: 0s - loss: 0.6950 - accuracy: 0.5031
Epoch 6: ReduceLROnPlateau reducing learning rate to 0.0005000000237487257.
72/72 [==============================] - 1s 20ms/step - loss: 0.6950 - accuracy: 0.5022 -
val_loss: 0.6937 - val_accuracy: 0.4545 - lr: 0.0010
Epoch 7/20
72/72 [==============================] - 2s 23ms/step - loss: 0.6927 - accuracy: 0.5079 -
val_loss: 0.6936 - val_accuracy: 0.4822 - lr: 5.0000e-04
Epoch 8/20
72/72 [==============================] - 1s 19ms/step - loss: 0.6913 - accuracy: 0.5154 -
val_loss: 0.6952 - val_accuracy: 0.4941 - lr: 5.0000e-04
8/8 [==============================] - 1s 8ms/step
Epoch 1/20
72/72 [==============================] - 10s 41ms/step - loss: 0.6958 - accuracy: 0.5105 -
val_loss: 0.6929 - val_accuracy: 0.5020 - lr: 0.0010
```

```
Epoch 2/20
72/72 [==============================] - 1s 20ms/step - loss: 0.6950 - accuracy: 0.5289 -
val_loss: 0.6918 - val_accuracy: 0.5296 - lr: 0.0010
Epoch 3/20
72/72 [==============================] - 1s 19ms/step - loss: 0.6954 - accuracy: 0.5162 -
val_loss: 0.6921 - val_accuracy: 0.5375 - lr: 0.0010
Epoch 4/20
72/72 [==============================] - 1s 19ms/step - loss: 0.6928 - accuracy: 0.5206 -
val_loss: 0.6902 - val_accuracy: 0.5534 - lr: 0.0010
Epoch 5/20
72/72 [==============================] - 1s 20ms/step - loss: 0.6935 - accuracy: 0.5167 -
val_loss: 0.6947 - val_accuracy: 0.4743 - lr: 0.0010
Epoch 6/20
72/72 [==============================] - 2s 21ms/step - loss: 0.6923 - accuracy: 0.5215 -
val_loss: 0.6982 - val_accuracy: 0.4901 - lr: 0.0010
Epoch 7/20
72/72 [==============================] - ETA: 0s - loss: 0.6932 - accuracy: 0.5184
Epoch 7: ReduceLROnPlateau reducing learning rate to 0.0005000000237487257.
72/72 [==============================] - 1s 20ms/step - loss: 0.6932 - accuracy: 0.5184 -
val_loss: 0.6962 - val_accuracy: 0.4743 - lr: 0.0010
Epoch 8/20
72/72 [==============================] - 1s 20ms/step - loss: 0.6930 - accuracy: 0.5202 -
val_loss: 0.7037 - val_accuracy: 0.4545 - lr: 5.0000e-04
Epoch 9/20
72/72 [==============================] - 1s 21ms/step - loss: 0.6911 - accuracy: 0.5303 -
val_loss: 0.6974 - val_accuracy: 0.4901 - lr: 5.0000e-04
8/8 [==============================] - 1s 8ms/step
Epoch 1/20
72/72 [==============================] - 10s 42ms/step - loss: 0.7005 - accuracy: 0.5035 -
val_loss: 0.6931 - val_accuracy: 0.4743 - lr: 0.0010
Epoch 2/20
72/72 [==============================] - 1s 20ms/step - loss: 0.6996 - accuracy: 0.4987 -
val_loss: 0.6928 - val_accuracy: 0.5178 - lr: 0.0010
Epoch 3/20
72/72 [==============================] - 1s 19ms/step - loss: 0.6961 - accuracy: 0.5136 -
val_loss: 0.6949 - val_accuracy: 0.4625 - lr: 0.0010
Epoch 4/20
72/72 [==============================] - 1s 19ms/step - loss: 0.6925 - accuracy: 0.5228 -
val_loss: 0.6969 - val_accuracy: 0.4704 - lr: 0.0010
Epoch 5/20
72/72 [==============================] - 1s 19ms/step - loss: 0.6951 - accuracy: 0.5018 -
val_loss: 0.6892 - val_accuracy: 0.5613 - lr: 0.0010
Epoch 6/20
```

```
72/72 [==============================] - 1s 20ms/step - loss: 0.6938 - accuracy: 0.5096 -
val_loss: 0.6968 - val_accuracy: 0.4625 - lr: 0.0010
Epoch 7/20
72/72 [==============================] - 1s 19ms/step - loss: 0.6938 - accuracy: 0.5061 -
val_loss: 0.6917 - val_accuracy: 0.5375 - lr: 0.0010
Epoch 8/20
71/72 [============================>.] - ETA: 0s - loss: 0.6939 - accuracy: 0.5075
Epoch 8: ReduceLROnPlateau reducing learning rate to 0.0005000000237487257.
72/72 [==============================] - 1s 19ms/step - loss: 0.6939 - accuracy: 0.5070 -
val_loss: 0.6915 - val_accuracy: 0.5336 - lr: 0.0010
Epoch 9/20
72/72 [==============================] - 1s 19ms/step - loss: 0.6931 - accuracy: 0.5114 -
val_loss: 0.6901 - val_accuracy: 0.5455 - lr: 5.0000e-04
Epoch 10/20
72/72 [==============================] - 1s 20ms/step - loss: 0.6927 - accuracy: 0.5083 -
val_loss: 0.6925 - val_accuracy: 0.5138 - lr: 5.0000e-04
8/8 [==============================] - 1s 10ms/step
K-Fold Metrics Summary:
```

	Fold	Accuracy	TP	TN	FP	FN	FPR	FNR	TSS	HSS
0	1	0.511811	24	106	21	103	0.165354	0.811024	0.023622	0.023622
1	2	0.547244	102	37	82	33	0.689076	0.244444	0.066480	0.068143
2	3	0.551181	88	52	78	36	0.600000	0.290323	0.109677	0.108827
3	4	0.541502	51	86	43	73	0.333333	0.588710	0.077957	0.078324
4	5	0.537549	96	40	89	28	0.689922	0.225806	0.084271	0.083475
5	6	0.533597	84	51	63	55	0.552632	0.395683	0.051685	0.052013
6	7	0.494071	6	119	3	125	0.024590	0.954198	0.021211	0.020507
7	8	0.505929	8	120	7	118	0.055118	0.936508	0.008374	0.008403
8	9	0.553360	51	89	44	69	0.330827	0.575000	0.094173	0.095142
9	10	0.561265	114	28	91	20	0.764706	0.149254	0.086040	0.089013

```
Metrics saved to 'kfold_metrics_lstm.csv'

Average Metrics across 10 folds:
Fold: 5.5000
Accuracy: 0.5338
TP: 62.4000
TN: 72.8000
FP: 52.1000
FN: 66.0000
FPR: 0.4206
FNR: 0.5171
TSS: 0.0623
HSS: 0.0627
```

Comparison of Results :

```
file_paths = {
    "Random Forest": "RF_kfold_metrics_summary.csv",
    "Decision Tree": "DT_kfold_metrics_summary.csv",
    "SVM": "SVM_kfold_metrics_summary.csv",
    "LSTM": "LSTM_kfold_metrics_summary.csv"
}

selected_metrics = ['Fold', 'Accuracy', 'TP', 'TN', 'FP', 'FN', 'FPR', 'FNR',
'TSS', 'HSS']

decimal_precision = 4


def collect_metrics(model_name, file_path, metrics=selected_metrics):
    try:
        metrics_df = pd.read_csv(file_path)
        metrics_df['Model'] = model_name

                                                          avg_metrics       =
metrics_df[metrics].mean(numeric_only=True).round(decimal_precision).to_dict()
        avg_metrics['Model'] = model_name

        fold_metrics = metrics_df[metrics + ['Model']]
        return avg_metrics, fold_metrics
    except FileNotFoundError:
        print(f"File not found for {model_name}: {file_path}")
        return None, None
    except Exception as e:
        print(f"Error processing {model_name}: {str(e)}")
        return None, None


def display_table(title, df):
    print(f"\n{title}")
    print(tabulate(df, headers='keys', tablefmt='pretty', showindex=False))


aggregated_results = []
```

```
foldwise_results = []

for model_name, file_path in file_paths.items():
    avg_metrics, fold_metrics = collect_metrics(model_name, file_path)
    if avg_metrics and fold_metrics is not None:
        aggregated_results.append(avg_metrics)
        foldwise_results.append(fold_metrics)

if aggregated_results and foldwise_results:
     comparison_df = pd.DataFrame(aggregated_results).sort_values(by='Accuracy',
ascending=False)
    display_table("Model Comparison Results (Average Metrics):", comparison_df)

    comparison_df.to_csv("model_comparison.csv", index=False)
    print("\nComparison results saved to 'model_comparison.csv'")

    all_folds_df = pd.concat(foldwise_results, ignore_index=True)
    display_table("Fold-Wise Metrics for All Models:", all_folds_df)

    all_folds_df.to_csv("fold_wise_metrics.csv", index=False)
    print("\nFold-wise metrics saved to 'fold_wise_metrics.csv'")

    with open("model_comparison.txt", "w") as f:
            f.write(tabulate(comparison_df, headers='keys', tablefmt='pretty',
showindex=False))
    with open("fold_wise_metrics.txt", "w") as f:
             f.write(tabulate(all_folds_df, headers='keys', tablefmt='pretty',
showindex=False))
    print("\nFormatted tables saved as text files.")
else:
       print("\nNo results to display. Ensure all files are available and
correctly formatted.")
```

This Python script is designed to aggregate, analyze, and compare performance metrics from multiple machine learning models used for predicting gold prices. Each model has its results stored in separate CSV files, which are processed to generate summary statistics and comparative insights. Here's a breakdown of the main components and functions:

Main Components:

- `file_paths` Dictionary: Maps model names to their corresponding CSV file paths containing k-fold cross-validation metrics.
- `selected_metrics` List: Specifies the metrics to be retrieved and analyzed for each model.
- `decimal_precision`: Sets the number of decimal places for rounding average metrics.

Functions:

1. `collect_metrics()`:
    - Purpose: Reads metrics from CSV files, calculates average metrics, and appends the model name for identification.
    - Error Handling: Includes checks for file existence and general exceptions to ensure robust processing.
    - Outputs: Returns a tuple containing average metrics and fold-wise metrics for each model. If there's an error, it returns `None`.
2. `display_table()`:
    - Purpose: Prints formatted tables using the `tabulate` library for clear presentation of data in the console.

Processing Flow:

- Aggregating Results: Iterates over each model's data, collects metrics using `collect_metrics()`, and appends results to lists for aggregated and fold-wise metrics.
- Dataframes Creation:
    - Comparison DataFrame: Aggregates average metrics across models and sorts them by accuracy for a high-level comparison.
    - All Folds DataFrame: Combines fold-wise metrics from all models for a detailed view.
- Display and Save Results:
    - Console Display: Uses `display_table()` to print results in a readable format.
    - CSV and Text Outputs: Saves the comparison and fold-wise results to CSV files for external analysis and text files for easy reference.

Output:

- Formatted Summaries: Provides formatted tables in the console for immediate review.
- Persistent Files:
    - CSV files for detailed numerical analysis and integration with other tools.
    - Text files with pretty-formatted tables for reports or documentation.

```
Model Comparison Results (Average Metrics):
+------+----------+------+------+------+------+--------+--------+--------+--------+---------------+
| Fold | Accuracy |  TP  |  TN  |  FP  |  FN  |  FPR   |  FNR   |  TSS   |  HSS   |     Model     |
+------+----------+------+------+------+------+--------+--------+--------+--------+---------------+
| 5.5  |  0.5338  | 62.4 | 72.8 | 52.1 | 66.0 | 0.4206 | 0.5171 | 0.0623 | 0.0627 |     LSTM      |
| 5.5  |  0.5182  | 67.9 | 64.2 | 61.4 | 61.4 | 0.4869 | 0.4727 | 0.0404 | 0.0401 |      SVM      |
| 5.5  |  0.5072  | 67.2 | 62.7 | 63.8 | 62.4 | 0.5042 | 0.4815 | 0.0143 | 0.0142 | Random Forest |
| 5.5  |  0.5061  | 52.7 | 76.6 | 49.5 | 76.7 |  0.4   |  0.6   |  0.0   |  0.0   | Decision Tree |
+------+----------+------+------+------+------+--------+--------+--------+--------+---------------+

Comparison results saved to 'model_comparison.csv'

Fold-Wise Metrics for All Models:
+------+--------------------+-----+-----+-----+-----+--------------------+--------------------+--------------------+---------------------+---------------+
| Fold |      Accuracy      | TP  | TN  | FP  | FN  |        FPR         |        FNR         |        TSS         |         HSS         |     Model     |
+------+--------------------+-----+-----+-----+-----+--------------------+--------------------+--------------------+---------------------+---------------+
|  1   | 0.5291828793774319 | 64  | 72  | 68  | 53  | 0.4857142857142857 | 0.452991452991453  | 0.0612942612942613 | 0.0606554901072345  | Random Forest |
|  2   |      0.48828125    | 71  | 54  | 68  | 63  | 0.5573770491803278 | 0.4701492537313433 | -0.0275263029116711| -0.0275769089349184 | Random Forest |
|  3   |      0.51953125    | 63  | 70  | 56  | 67  | 0.4444444444444444 | 0.5153846153846153 | 0.0401709401709401 | 0.0401170588952566  | Random Forest |
|  4   |      0.55859375    | 72  | 71  | 53  | 60  | 0.4274193548387097 | 0.4545454545454545 | 0.1180351906158357 | 0.1178336179555989  | Random Forest |
|  5   |      0.4609375     | 63  | 55  | 76  | 62  | 0.5801526717557252 |       0.496        | -0.0761526717557251| -0.0759578485716026 | Random Forest |
|  6   |      0.50390625    | 62  | 67  | 63  | 64  | 0.4846153846153846 | 0.5079365079365079 | 0.0074481074481074 | 0.0074490169739894  | Random Forest |
|  7   |      0.5234375     | 74  | 60  | 59  | 63  | 0.4957983193277311 | 0.4598540145985401 | 0.0443476660737287 | 0.0442499540975579  | Random Forest |
|  8   |      0.46484375    | 60  | 59  | 66  | 71  |       0.528        | 0.5419847328244275 | -0.0699847328244275| -0.0699206833435021 | Random Forest |
|  9   |      0.5234375     | 69  | 65  | 57  | 65  | 0.4672131147540984 | 0.4850746268656716 |  0.04771225838023  | 0.0475725786777262  | Random Forest |
|  10  |        0.5         | 74  | 54  | 72  | 56  | 0.5714285714285714 | 0.4307692307692308 | -0.0021978021978021| -0.0022021042329336 | Random Forest |
|  1   |      0.47265625    |  0  | 121 |  0  | 135 |        0.0         |        1.0         |        0.0         |         0.0         | Decision Tree |
|  2   |      0.49609375    |  0  | 127 |  0  | 129 |        0.0         |        1.0         |        0.0         |         0.0         | Decision Tree |
|  3   |      0.5078125     | 130 |  0  | 126 |  0  |        1.0         |        0.0         |        0.0         |         0.0         | Decision Tree |
|  4   |      0.4765625     |  0  | 122 |  0  | 134 |        0.0         |        1.0         |        0.0         |         0.0         | Decision Tree |
|  5   |      0.49609375    | 127 |  0  | 129 |  0  |        1.0         |        0.0         |        0.0         |         0.0         | Decision Tree |
|  6   | 0.5137254901960784 | 131 |  0  | 124 |  0  |        1.0         |        0.0         |        0.0         |         0.0         | Decision Tree |
|  7   | 0.5294117647058824 |  0  | 135 |  0  | 120 |        0.0         |        1.0         |        0.0         |         0.0         | Decision Tree |
|  8   | 0.5450980392156862 | 139 |  0  | 116 |  0  |        1.0         |        0.0         |        0.0         |         0.0         | Decision Tree |
|  9   | 0.5294117647058824 |  0  | 135 |  0  | 120 |        0.0         |        1.0         |        0.0         |         0.0         | Decision Tree |
|  10  | 0.4941176470588235 |  0  | 126 |  0  | 129 |        0.0         |        1.0         |        0.0         |         0.0         | Decision Tree |
|  1   | 0.4941176470588235 | 64  | 62  | 62  | 67  |        0.5         | 0.5114503816793893 | -0.0114503816793893| -0.0114380592196291 |      SVM      |
|  2   | 0.5019607843137255 | 71  | 57  | 78  | 49  | 0.5777777777777777 | 0.4083333333333333 | 0.0138888888888889 | 0.0137048880767473  |      SVM      |
|  3   | 0.5254901960784314 | 71  | 63  | 73  | 48  | 0.5367647058823529 | 0.4033613445378151 | 0.0598739495798319 | 0.0590979782270606  |      SVM      |
|  4   | 0.4705882352941176 | 62  | 58  | 61  | 74  | 0.5126050420168067 | 0.5441176470588235 | -0.0567226890756302| -0.056338028169014  |      SVM      |
|  5   | 0.5098039215686274 | 57  | 73  | 46  | 79  | 0.3865546218487395 | 0.5808823529411765 |  0.032563025210084 | 0.0320082601961796  |      SVM      |
|  6   | 0.5450980392156862 | 75  | 64  | 67  | 49  | 0.5114503816793893 | 0.3951612903225806 | 0.0933883279980301 | 0.0930275341877721  |      SVM      |
|  7   | 0.5490196078431373 | 75  | 65  | 55  | 60  | 0.4583333333333333 | 0.4444444444444444 | 0.0972222222222222 | 0.0969976905311778  |      SVM      |
|  8   | 0.5411764705882353 | 73  | 65  | 51  | 66  | 0.4396551724137931 | 0.4748201438848921 | 0.0855246837013147 | 0.0846193968029945  |      SVM      |
|  9   | 0.5176470588235295 | 61  | 71  | 58  | 65  | 0.4496124031007752 | 0.5158730158730159 | 0.0345145810262089 | 0.0345368916797488  |      SVM      |
|  10  | 0.5275590551181102 | 70  | 64  | 63  | 57  | 0.4960629921259842 | 0.4488188976377952 | 0.0551181102362204 | 0.0551181102362204  |      SVM      |
|  1   | 0.5118110236220472 | 24  | 106 | 21  | 103 | 0.1653543307086614 | 0.8110236220472441 | 0.0236220472440944 | 0.0236220472440944  |     LSTM      |
|  2   | 0.547244094488189  | 102 | 37  | 82  | 33  | 0.6890756302521008 | 0.2444444444444444 | 0.0664799253034547 | 0.0681426657308747  |     LSTM      |
|  3   | 0.5511811023622047 | 88  | 52  | 78  | 36  |        0.6         | 0.2903225806451613 | 0.1096774193548387 | 0.1088267881324633  |     LSTM      |
|  4   | 0.541501976284585  | 51  | 86  | 43  | 73  | 0.3333333333333333 | 0.5887096774193549 | 0.0779569892473118 | 0.0783242258652094  |     LSTM      |
|  5   | 0.5375494071146245 | 96  | 40  | 89  | 28  | 0.689922480620155  | 0.2258064516129032 | 0.0842710677669417 | 0.0834752453788277  |     LSTM      |
|  6   | 0.5335968379446641 | 84  | 51  | 63  | 55  | 0.5526315789473685 |  0.39568345323741  | 0.0516849678152214 | 0.0520132097040518  |     LSTM      |
|  7   | 0.4940711462450592 |  6  | 119 |  3  | 125 | 0.0245901639344262 | 0.9541984732824428 |  0.021211362783131 | 0.0205069263807392  |     LSTM      |
|  8   | 0.5059288537549407 |  8  | 120 |  7  | 118 | 0.0551181102362204 | 0.9365079365079364 |  0.008373953255843 | 0.0084030978584642  |     LSTM      |
```

```
|  9   | 0.5533596837944664 | 51  | 89  | 44  | 69  | 0.3308270676691729 |       0.575        |  0.094172932330827  | 0.0951416363348631  |     LSTM      |
|  10  | 0.5612648221343873 | 114 | 28  | 91  | 20  | 0.7647058823529411 | 0.1492537313432835 | 0.0860403863037753  | 0.0890128783209524  |     LSTM      |
+------+--------------------+-----+-----+-----+-----+--------------------+--------------------+---------------------+---------------------+---------------+
```

```
Fold-wise metrics saved to 'fold_wise_metrics.csv'

Formatted tables saved as text files.
```