Loading Dataset

↳ 2 cells hidden

Steps to plot SMA and ES

[ ] ↳ 12 cells hidden

SMA

```
#It's important to note that SMA is a basic technical analysis tool and has inherent limitations. It tends to lag behind significant price
```

Plotings SMA for different terms

[ ] ↳ 21 cells hidden

Evaluation of SMA Model

```
#Simple Moving Averages (SMA) for each currency in the dataset, along with the Mean Squared Error (MSE), Mean Absolute Error (MAE), Root M
```

```
import pandas as pd
import numpy as np
from tabulate import tabulate

# Load the data
df = pd.read_csv('cryptocurrency.csv')

# Define the function to calculate Simple Moving Average
def calculate_sma(data, window_size):
    sma = data.rolling(window=window_size).mean()
    mape = np.mean(abs((data - sma) / data)) * 100
    return sma, mape

# Define a dictionary to store the results for each currency
results_30 = {}
results_60 = {}

# Calculate Simple Moving Averages, errors, and MAPE for each currency for both 30 and 60 day periods
for symbol in df['Symbol'].unique():
    df_curr = df[df['Symbol'] == symbol]
    sma_30, mape_30 = calculate_sma(df_curr['Close'], 30)
    mse_30 = np.mean((sma_30 - df_curr['Close']) ** 2)
    mae_30 = np.mean(abs(sma_30 - df_curr['Close']))
    rmse_30 = np.sqrt(mse_30)
    results_30[symbol] = {'MSE': mse_30, 'MAE': mae_30, 'RMSE': rmse_30, 'MAPE': mape_30}

    sma_60, mape_60 = calculate_sma(df_curr['Close'], 60)
    mse_60 = np.mean((sma_60 - df_curr['Close']) ** 2)
    mae_60 = np.mean(abs(sma_60 - df_curr['Close']))
    rmse_60 = np.sqrt(mse_60)
    results_60[symbol] = {'MSE': mse_60, 'MAE': mae_60, 'RMSE': rmse_60, 'MAPE': mape_60}

# Prepare the results for tabular display
table_headers = ['Currency', 'MSE (30 day)', 'MAE (30 day)', 'RMSE (30 day)', 'MAPE (30 day)', 'MSE (60 day)', 'MAE (60 day)', 'RMSE (60 c
table_rows = []
for symbol in results_30:
    row = [
        symbol,
        '{:.10f}'.format(results_30[symbol]['MSE']),
        '{:.10f}'.format(results_30[symbol]['MAE']),
        '{:.10f}'.format(results_30[symbol]['RMSE']),
        '{:.10f}'.format(results_30[symbol]['MAPE']),
        '{:.10f}'.format(results_60[symbol]['MSE']),
        '{:.10f}'.format(results_60[symbol]['MAE']),
```

```
        '{:.10f}'.format(results_60[symbol]['RMSE']),
        '{:.10f}'.format(results_60[symbol]['MAPE'])
    ]
    table_rows.append(row)

# Print out the results in a tabular format
table = tabulate(table_rows, headers=table_headers, tablefmt='grid')
print(table)

```

Currency	MSE (30 day)	MAE (30 day)	RMSE (30 day)	MAPE (30 day)	MSE (60 day)	MAE (60 day)	RMSE (60 day
BTC	3.53443e+06	739.617	1880.01	9.7985	8.71139e+06	1216.11	2951.51
ETH	20026.5	60.863	141.515	15.4647	40000	89.7577	200
XRP	0.0228524	0.0482692	0.15117	14.8806	0.0378636	0.0688217	0.194586
LTC	359.383	7.96584	18.9574	12.9566	650.763	11.6262	25.51
USDC	2.48877e-05	0.00267611	0.00498875	0.265654	2.63767e-05	0.00281594	0.0051358

```
#Directional Accuracy: Assess the directional accuracy of the SMA model by comparing the sign (positive or negative) of the SMA with the s
```

```
import pandas as pd
import numpy as np

# Load the data
df = pd.read_csv('cryptocurrency.csv')

# Define the function to calculate Simple Moving Average
def calculate_sma(data, window_size):
    sma = data.rolling(window=window_size).mean()
    mape = np.mean(abs((data - sma) / data)) * 100
    return sma, mape

# Define a dictionary to store the results for each currency
results_30 = {}
results_60 = {}

# Calculate Simple Moving Averages, errors, MAPE, and directional accuracy for each currency for both 30 and 60 day periods
for symbol in df['Symbol'].unique():
    df_curr = df[df['Symbol'] == symbol]
    sma_30, mape_30 = calculate_sma(df_curr['Close'], 30)
    mse_30 = np.mean((sma_30 - df_curr['Close']) ** 2)
    mae_30 = np.mean(abs(sma_30 - df_curr['Close']))
    rmse_30 = np.sqrt(mse_30)

    # Calculate directional accuracy
    price_changes_30 = np.diff(df_curr['Close']) > 0
    sma_direction_30 = np.diff(sma_30) > 0
    directional_accuracy_30 = np.mean(price_changes_30 == sma_direction_30) * 100

    results_30[symbol] = {'MSE': mse_30, 'MAE': mae_30, 'RMSE': rmse_30, 'MAPE': mape_30, 'Directional Accuracy (30 day)': directional_acc

    sma_60, mape_60 = calculate_sma(df_curr['Close'], 60)
    mse_60 = np.mean((sma_60 - df_curr['Close']) ** 2)
    mae_60 = np.mean(abs(sma_60 - df_curr['Close']))
    rmse_60 = np.sqrt(mse_60)

    # Calculate directional accuracy
    price_changes_60 = np.diff(df_curr['Close']) > 0
    sma_direction_60 = np.diff(sma_60) > 0
    directional_accuracy_60 = np.mean(price_changes_60 == sma_direction_60) * 100

    results_60[symbol] = {'MSE': mse_60, 'MAE': mae_60, 'RMSE': rmse_60, 'MAPE': mape_60, 'Directional Accuracy (60 day)': directional_acc

# Print out the results in a tabular format
table_headers = ['Currency', 'MSE (30 day)', 'MAE (30 day)', 'RMSE (30 day)', 'MAPE (30 day)', 'Directional Accuracy (30 day)', 'MSE (60 c
table_rows = []
for symbol in results_30:
    row = [
        symbol,
        '{:.10f}'.format(results_30[symbol]['MSE']),
```

```
        '{:.10f}'.format(results_30[symbol]['MAE']),
        '{:.10f}'.format(results_30[symbol]['RMSE']),
        '{:.10f}'.format(results_30[symbol]['MAPE']),
        '{:.2f}%'.format(results_30[symbol]['Directional Accuracy (30 day)']),
        '{:.10f}'.format(results_60[symbol]['MSE']),
        '{:.10f}'.format(results_60[symbol]['MAE']),
        '{:.10f}'.format(results_60[symbol]['RMSE']),
        '{:.10f}'.format(results_60[symbol]['MAPE']),
        '{:.2f}%'.format(results_60[symbol]['Directional Accuracy (60 day)'])
    ]
    table_rows.append(row)

# Calculate the maximum width for each column
column_widths = [max(len(header), max(len(str(row[i])) for row in table_rows)) for i, header in enumerate(table_headers)]

# Print the table
print(' | '.join([header.ljust(column_widths[i]) for i, header in enumerate(table_headers)]))
print('-' * (sum(column_widths) + len(table_headers) * 3 - 1))
for row in table_rows:
    print(' | '.join([str(row[i]).ljust(column_widths[i]) for i in range(len(row))]))

```

```
Currency | MSE (30 day)       | MAE (30 day)   | RMSE (30 day)   | MAPE (30 day) | Directional Accuracy (30 day) | MSE (60 day)       |
----------------------------------------------------------------------------------------------------------------------------------------
BTC      | 3534425.8360571079 | 739.6167777477 | 1880.0068712792 | 9.7984972069  | 56.69%                        | 8711392.3559497483 |
ETH      | 20026.4968330427   | 60.8630217434  | 141.5150056815  | 15.4647335284 | 56.42%                        | 39999.9947461019   |
XRP      | 0.0228523853       | 0.0482692373   | 0.1511700543    | 14.8806052628 | 55.22%                        | 0.0378635823       |
LTC      | 359.3830942502     | 7.9658429784   | 18.9574020965   | 12.9566421170 | 54.15%                        | 650.7625307051     |
USDC     | 0.0000248877       | 0.0026761081   | 0.0049887548    | 0.2656537991  | 58.94%                        | 0.0000263767       |
```

```
#Backtesting: Perform a historical backtest using the SMA model. Simulate trading decisions based on the SMA signals (e.g., buying when th
```

```
import pandas as pd
import numpy as np

# Load the data
df = pd.read_csv('cryptocurrency.csv')

# Define the function to calculate Simple Moving Average
def calculate_sma(data, window_size):
    return data.rolling(window=window_size).mean()

# Define the backtesting function
def backtest_sma_strategy(data, window_size):
    # Calculate the Simple Moving Average (SMA)
    sma = calculate_sma(data, window_size)

    # Simulate trading decisions based on SMA signals
    positions = np.where(data > sma, 1, -1)

    # Align positions with the price data
    positions = positions[sma.notna()]

    # Calculate daily returns
    returns = data.pct_change()

    # Adjust returns to match positions length
    returns = returns[sma.notna()]

    # Apply positions to calculate portfolio returns
    portfolio_returns = returns * positions

    # Calculate performance metrics
    total_return = portfolio_returns.sum()
    annualized_return = (1 + total_return) ** (252 / len(data)) - 1
    sharpe_ratio = np.sqrt(252) * portfolio_returns.mean() / portfolio_returns.std()
    cumulative_returns = (1 + portfolio_returns).cumprod()
    max_drawdown = 1 - cumulative_returns.div(cumulative_returns.cummax())
    max_drawdown = max_drawdown.max()

    return total_return, annualized_return, sharpe_ratio, max_drawdown

# Define the window size for Simple Moving Average (SMA)
window_size = 30

```

```
# Perform backtesting and evaluate performance for each currency
results = {}
for symbol in df['Symbol'].unique():
    df_curr = df[df['Symbol'] == symbol]
    closing_prices = df_curr['Close']

    # Perform backtesting
    total_return, annualized_return, sharpe_ratio, max_drawdown = backtest_sma_strategy(closing_prices, window_size)

    # Store the results
    results[symbol] = {
        'Total Return': total_return,
        'Annualized Return': annualized_return,
        'Sharpe Ratio': sharpe_ratio,
        'Max Drawdown': max_drawdown
    }

# Print the results for each currency
for symbol, metrics in results.items():
    print(f"Currency: {symbol}")
    print(f"Total Return: {metrics['Total Return']:.2f}")
    print(f"Annualized Return: {metrics['Annualized Return']:.2%}")
    print(f"Sharpe Ratio: {metrics['Sharpe Ratio']:.2f}")
    print(f"Max Drawdown: {metrics['Max Drawdown']:.2%}")
    print()

```

```
Currency: BTC
Total Return: 25.82
Annualized Return: 31.93%
Sharpe Ratio: 3.32
Max Drawdown: 41.50%

Currency: ETH
Total Return: 28.55
Annualized Return: 48.44%
Sharpe Ratio: 3.57
Max Drawdown: 40.70%

Currency: XRP
Total Return: 40.10
Annualized Return: 38.22%
Sharpe Ratio: 2.81
Max Drawdown: 54.04%

Currency: LTC
Total Return: 34.36
Annualized Return: 35.04%
Sharpe Ratio: 2.72
Max Drawdown: 62.41%

Currency: USDC
Total Return: 1.38
Annualized Return: 24.30%
Sharpe Ratio: 5.15
Max Drawdown: 2.79%
```

```
#Comparison with Other Models: Compare the performance of the SMA model with other technical indicators or predictive models. Use metrics
```

```
import pandas as pd
import numpy as np

# Load the data
df = pd.read_csv('cryptocurrency.csv')

# Define the function to calculate Simple Moving Average
def calculate_sma(data, window_size):
    return data.rolling(window=window_size).mean()

# Define another predictive model or technical indicator
# ...

# Define the evaluation function
def evaluate_model(data, window_size, model_name):
    # Calculate the Simple Moving Average (SMA)
    sma = calculate_sma(data, window_size)

```

```
    # Calculate the predictions of the other model or indicator
    # ...

    # Apply a trading strategy based on the predictions
    # ...

    # Calculate the accuracy of the SMA model
    sma_direction = np.sign(np.diff(sma))
    price_changes = np.sign(np.diff(data))
    accuracy = np.mean(sma_direction == price_changes) * 100

    # Calculate the accuracy of the other model or indicator
    # ...

    # Return the evaluation results
    return accuracy

# Define the window size for Simple Moving Average (SMA)
window_size = 30

# Define the name of the other model or indicator
model_name = 'Other Model'

# Evaluate the performance of the SMA model
sma_accuracy = evaluate_model(df['Close'], window_size, 'SMA')

# Evaluate the performance of the other model or indicator
# other_model_accuracy = evaluate_model(df['Close'], window_size, model_name)

# Print the results
print(f"SMA Accuracy: {sma_accuracy:.2f}%")
# print(f"{model_name} Accuracy: {other_model_accuracy:.2f}%")

```

```
SMA Accuracy: 55.72%
```

```
import pandas as pd
import numpy as np

# Load the data
df = pd.read_csv('cryptocurrency.csv')

# Define the function to calculate Simple Moving Average
def calculate_sma(data, window_size):
    return data.rolling(window=window_size).mean()

# Define another predictive model or technical indicator
# ...

# Define the evaluation function
def evaluate_model(data, window_size, model_name):
    # Calculate the Simple Moving Average (SMA)
    sma = calculate_sma(data, window_size)

    # Calculate the predictions of the other model or indicator
    # ...

    # Apply a trading strategy based on the predictions
    # ...

    # Calculate the accuracy of the SMA model
    sma_direction = np.sign(np.diff(sma))
    price_changes = np.sign(np.diff(data))
    accuracy = np.mean(sma_direction == price_changes) * 100

    # Calculate the accuracy of the other model or indicator
    # ...

    # Return the evaluation results
    return accuracy

# Define the window size for Simple Moving Average (SMA)
window_size = 30

# Define the name of the other model or indicator
model_name = 'Other Model'
```

```

# Define a dictionary to store the evaluation results for each coin
evaluation_results = {}

# Loop through each coin
for coin in df['Symbol'].unique():
    coin_data = df[df['Symbol'] == coin]
    coin_close_prices = coin_data['Close']

    # Evaluate the performance of the SMA model for the current coin
    sma_accuracy = evaluate_model(coin_close_prices, window_size, 'SMA')

    # Evaluate the performance of the other model or indicator for the current coin
    # other_model_accuracy = evaluate_model(coin_close_prices, window_size, model_name)

    # Store the evaluation results for the current coin
    evaluation_results[coin] = {'SMA Accuracy': sma_accuracy}
    # evaluation_results[coin] = {f'{model_name} Accuracy': other_model_accuracy}

# Print the evaluation results for each coin
for coin, results in evaluation_results.items():
    print(f"{coin}: {results['SMA Accuracy']:.2f}%")
    # print(f"{coin}: {results[f'{model_name} Accuracy']:.2f}%")

```

```
BTC: 56.22%
ETH: 55.67%
XRP: 54.74%
LTC: 53.61%
USDC: 57.54%
```

```
import pandas as pd
import numpy as np

# Load the data
df = pd.read_csv('cryptocurrency.csv')

# Define the function to calculate Simple Moving Average
def calculate_sma(data, window_size):
    return data.rolling(window=window_size).mean()

# Define another predictive model or technical indicator
# ...

# Define the evaluation function
def evaluate_model(data, window_size, model_name):
    # Calculate the Simple Moving Average (SMA)
    sma = calculate_sma(data, window_size)

    # Calculate the predictions of the other model or indicator
    # ...

    # Apply a trading strategy based on the predictions
    # ...

    # Calculate the accuracy of the SMA model
    sma_direction = np.sign(np.diff(sma))
    price_changes = np.sign(np.diff(data))
    accuracy = np.mean(sma_direction == price_changes) * 100

    # Calculate the precision and recall of the SMA model
    tp = np.sum((sma_direction == 1) & (price_changes == 1))
    fp = np.sum((sma_direction == 1) & (price_changes == -1))
    fn = np.sum((sma_direction == -1) & (price_changes == 1))
    precision = tp / (tp + fp)
    recall = tp / (tp + fn)

    # Calculate the accuracy, precision, and recall of the other model or indicator
    # ...

    # Return the evaluation results
    return accuracy, precision, recall

# Define the window size for Simple Moving Average (SMA)
window_size = 30

# Define the name of the other model or indicator
```

```
model_name = 'Other Model'

# Define a dictionary to store the evaluation results for each coin
evaluation_results = {}

# Loop through each coin
for coin in df['Symbol'].unique():
    coin_data = df[df['Symbol'] == coin]
    coin_close_prices = coin_data['Close']

    # Evaluate the performance of the SMA model for the current coin
    sma_accuracy, sma_precision, sma_recall = evaluate_model(coin_close_prices, window_size, 'SMA')

    # Evaluate the performance of the other model or indicator for the current coin
    # other_model_accuracy, other_model_precision, other_model_recall = evaluate_model(coin_close_prices, window_size, model_name)

    # Store the evaluation results for the current coin
    evaluation_results[coin] = {'SMA Accuracy': sma_accuracy, 'SMA Precision': sma_precision, 'SMA Recall': sma_recall}
    # evaluation_results[coin] = {f'{model_name} Accuracy': other_model_accuracy, f'{model_name} Precision': other_model_precision, f'{moc

# Print the evaluation results for each coin
for coin, results in evaluation_results.items():
    print(f"{coin}:")
    print(f"SMA Accuracy: {results['SMA Accuracy']:.2f}%")
    print(f"SMA Precision: {results['SMA Precision']:.2f}")
    print(f"SMA Recall: {results['SMA Recall']:.2f}")
    print()
    # print(f"{coin}:")
    # print(f"{model_name} Accuracy: {results[f'{model_name} Accuracy']:.2f}%")
    # print(f"{model_name} Precision: {results[f'{model_name} Precision']:.2f}")
    # print(f"{model_name} Recall: {results[f'{model_name} Recall']:.2f}")
    # print()
```

```
BTC:
SMA Accuracy: 56.22%
SMA Precision: 0.60
SMA Recall: 0.63

ETH:
SMA Accuracy: 55.67%
SMA Precision: 0.56
SMA Recall: 0.64

XRP:
SMA Accuracy: 54.74%
SMA Precision: 0.53
SMA Recall: 0.49

LTC:
SMA Accuracy: 53.61%
SMA Precision: 0.53
SMA Recall: 0.53

USDC:
SMA Accuracy: 57.54%
SMA Precision: 0.60
SMA Recall: 0.59
```

```
#Walk-Forward Analysis: Split the dataset into training and testing periods, then iteratively retrain the SMA model using a rolling window
```

```
import pandas as pd
import numpy as np

# Load the data
df = pd.read_csv('cryptocurrency.csv')

# Define the function to calculate Simple Moving Average
def calculate_sma(data, window_size):
    sma = data.rolling(window=window_size).mean()
    return sma

# Define the window size for Simple Moving Average (SMA)
sma_window = 30

# Define the number of periods for walk-forward analysis
n_periods = 10
```

```python

# Define lists to store the evaluation results
accuracy_list = []
precision_list = []
recall_list = []

# Perform walk-forward analysis
for i in range(n_periods):
    # Split the dataset into training and testing periods
    train_data = df[:i+1]
    test_data = df[i+1:i+2]

    # Calculate Simple Moving Average for training data
    train_sma = calculate_sma(train_data['Close'], sma_window)

    # Simulate trading decisions based on the SMA signals
    predicted_direction = np.where(train_sma.shift(1) < train_sma, 1, -1)
    actual_direction = np.where(train_data['Close'].shift(1) < train_data['Close'], 1, -1)

    # Calculate evaluation metrics
    accuracy = np.mean(predicted_direction == actual_direction)
    precision = np.mean(predicted_direction[actual_direction == 1] == 1)
    recall = np.mean(actual_direction[predicted_direction == 1] == 1) / np.mean(actual_direction == 1)

    # Store the evaluation results
    accuracy_list.append(accuracy)
    precision_list.append(precision)
    recall_list.append(recall)

# Calculate the average performance over the walk-forward analysis
average_accuracy = np.nanmean(accuracy_list)
average_precision = np.nanmean(precision_list)
average_recall = np.nanmean(recall_list)

# Print the evaluation results
print("Average Accuracy: {:.2%}".format(average_accuracy))
print("Average Precision: {:.2%}".format(average_precision))
print("Average Recall: {:.2%}".format(average_recall))

```

```
Average Accuracy: 87.75%
Average Precision: 0.00%
Average Recall: nan%
/usr/local/lib/python3.10/dist-packages/numpy/core/fromnumeric.py:3474: RuntimeWarning: Mean of empty slice.
  return _methods._mean(a, axis=axis, dtype=dtype,
/usr/local/lib/python3.10/dist-packages/numpy/core/_methods.py:189: RuntimeWarning: invalid value encountered in double_scalars
  ret = ret.dtype.type(ret / rcount)
<ipython-input-109-cdd1b1082488>:49: RuntimeWarning: Mean of empty slice
  average_recall = np.nanmean(recall_list)
```

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Rest of the code for calculations

# Plot the evaluation results
periods = range(1, n_periods + 1)

# Plot accuracy
plt.plot(periods, accuracy_list, label='Accuracy')
plt.axhline(y=average_accuracy, color='r', linestyle='--', label='Average Accuracy')
plt.xlabel('Periods')
plt.ylabel('Accuracy')
plt.title('Accuracy over Walk-Forward Analysis')
plt.legend()
plt.show()

# Plot precision
plt.plot(periods, precision_list, label='Precision')
plt.axhline(y=average_precision, color='r', linestyle='--', label='Average Precision')
plt.xlabel('Periods')
plt.ylabel('Precision')
plt.title('Precision over Walk-Forward Analysis')
plt.legend()
plt.show()

```

```
# Plot recall
plt.plot(periods, recall_list, label='Recall')
plt.axhline(y=average_recall, color='r', linestyle='--', label='Average Recall')
plt.xlabel('Periods')
plt.ylabel('Recall')
plt.title('Recall over Walk-Forward Analysis')
plt.legend()
plt.show()

```

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

# Load the data
df = pd.read_csv('cryptocurrency.csv')

# Convert 'Date' column to datetime
df['Date'] = pd.to_datetime(df['Date'])

# Convert numeric columns to float
numeric_columns = df.columns[1:]  # Exclude the 'Date' column
df[numeric_columns] = df[numeric_columns].apply(pd.to_numeric, errors='coerce')

```

```
# Define the function to calculate Simple Moving Average
def calculate_sma(data, window_size):
    sma = data.rolling(window=window_size).mean()
    return sma

# Define the window size for Simple Moving Average (SMA)
sma_window = 30

# Define the number of periods for walk-forward analysis
n_periods = 10

# Define lists to store the evaluation results for each coin
accuracy_dict = {}
precision_dict = {}
recall_dict = {}

# Perform walk-forward analysis for each coin
for coin in df.columns[1:]:  # Exclude the 'Date' column
    accuracy_list = []
    precision_list = []
    recall_list = []

    for i in range(n_periods):
        # Split the dataset into training and testing periods
        train_data = df[:i+1]
        test_data = df[i+1:i+2]

        # Calculate Simple Moving Average for training data
        train_sma = calculate_sma(train_data[coin], sma_window)

        # Simulate trading decisions based on the SMA signals
        predicted_direction = np.where(train_sma.shift(1) < train_sma, 1, -1)
        actual_direction = np.where(train_data[coin].shift(1) < train_data[coin], 1, -1)

        # Calculate evaluation metrics
        accuracy = np.mean(predicted_direction == actual_direction)
        precision = np.mean(predicted_direction[actual_direction == 1] == 1)
        recall = np.mean(actual_direction[predicted_direction == 1] == 1) / np.mean(actual_direction == 1)

        # Store the evaluation results
        accuracy_list.append(accuracy)
        precision_list.append(precision)
        recall_list.append(recall)

    # Store the evaluation results for the current coin
    accuracy_dict[coin] = accuracy_list
    precision_dict[coin] = precision_list
    recall_dict[coin] = recall_list

# Plot the evaluation results for each coin
periods = range(1, n_periods + 1)

for coin in df.columns[1:]:
    # Plot accuracy
    plt.plot(periods, accuracy_dict[coin], label='Accuracy')
    plt.axhline(y=np.nanmean(accuracy_dict[coin]), color='r', linestyle='--', label='Average Accuracy')
    plt.xlabel('Periods')
    plt.ylabel('Accuracy')
    plt.title(f'Accuracy over Walk-Forward Analysis - {coin}')
    plt.legend()
    plt.show()

    # Plot precision
    plt.plot(periods, precision_dict[coin], label='Precision')
    plt.axhline(y=np.nanmean(precision_dict[coin]), color='r', linestyle='--', label='Average Precision')
    plt.xlabel('Periods')
    plt.ylabel('Precision')
    plt.title(f'Precision over Walk-Forward Analysis - {coin}')
    plt.legend()
    plt.show()

    # Plot recall
    plt.plot(periods, recall_dict[coin], label='Recall')
    plt.axhline(y=np.nanmean(recall_dict[coin]), color='r', linestyle='--', label='Average Recall')
    plt.xlabel('Periods')
    plt.ylabel('Recall')
    plt.title(f'Recall over Walk-Forward Analysis - {coin}')
```

```
    plt.legend()
    plt.show()

```

```
#The Simple Moving Average (SMA) model itself does not inherently provide feature importance as it is a technique used for time-series ana

#However, if you are interested in identifying the important features that contribute to the SMA model's performance, you can explore addi

#1. Correlation Analysis: Calculate the correlation between the price series of the cryptocurrency and other relevant variables or indicat

#2. Technical Indicators: Apply various technical indicators (e.g., Relative Strength Index, Moving Average Convergence Divergence, Bollir

#3. Feature Engineering: Create additional features based on domain knowledge or specific characteristics of the cryptocurrency market. Fc

#4. Feature Selection Methods: Employ feature selection techniques such as recursive feature elimination, L1 regularization (Lasso), or ir

#By incorporating these techniques and analyzing the relationship between various features and the SMA model's predictions, you can gain i
```

SMA and ES

[ ] ↳ *1 cell hidden*

ES

[ ] ↳ 2 cells hidden