In [ ]:
```python
# Financial Data Analysis: S&P 500 and Bitcoin
# Data Collection and Preprocessing

import yfinance as yf
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from datetime import datetime
import warnings
import sklearn
import random
import tensorflow as tf
warnings.filterwarnings('ignore')

# Set display options for better output
pd.set_option('display.max_columns', None)
pd.set_option('display.width', 1000)

random.seed(42)
np.random.seed(42)
tf.random.set_seed(42)
```

Prepare Dataset

In [ ]:
```python
# Data Download Configuration
# S&P 500: January 1, 2002 to December 31, 2023
# Bitcoin: January 1, 2015 to December 31, 2023

# Define date ranges
sp500_start = "2002-01-01"
sp500_end = "2023-12-31"
bitcoin_start = "2015-01-01"
bitcoin_end = "2023-12-31"

print("Downloading S&P 500 data...")
sp500_data = yf.download("^GSPC", start=sp500_start, end=sp500_end, progress=False)

print("Downloading Bitcoin data...")
bitcoin_data = yf.download("BTC-USD", start=bitcoin_start, end=bitcoin_end, progress=False)
```

```
# Display basic information about downloaded data
print(f"\nS&P 500 Data Shape: {sp500_data.shape}")
print(f"S&P 500 Date Range: {sp500_data.index.min()} to {sp500_data.index.max()}")
print(f"Total S&P 500 observations: {len(sp500_data)}")

print(f"\nBitcoin Data Shape: {bitcoin_data.shape}")
print(f"Bitcoin Date Range: {bitcoin_data.index.min()} to {bitcoin_data.index.max()}")
print(f"Total Bitcoin observations: {len(bitcoin_data)}")
```

In [ ]:
```
# Change Close column to Adj Close (because S&P 500 and Bitcoin doesn't have adjusted close price)

sp500_data.columns = sp500_data.columns.set_levels(['Adj Close' if x == 'Close' else x for x in sp500_data.columns.levels[0]], l

bitcoin_data.columns = bitcoin_data.columns.set_levels(['Adj Close' if x == 'Close' else x for x in bitcoin_data.columns.levels[


print("\nVerification - S&P 500 has Adj Close:", ('Adj Close', '^GSPC') in sp500_data.columns)
print("Verification - Bitcoin has Adj Close:", ('Adj Close', 'BTC-USD') in bitcoin_data.columns)
```

In [ ]:
```
# Data Cleaning and Examination
# Checking for missing or incorrect observations

print("=== DATA QUALITY EXAMINATION ===\n")

# S&P 500 Data Examination
print("S&P 500 Data Quality Check:")
print("-" * 30)
print("Missing values by column:")
print(sp500_data.isnull().sum())
print(f"\nTotal missing values: {sp500_data.isnull().sum().sum()}")
print(f"Data completeness: {((1 - sp500_data.isnull().sum().sum() / (len(sp500_data) * len(sp500_data.columns))) * 100):.2f}%")

# Check for zero or negative prices (incorrect observations)
print("\nChecking for zero or negative prices:")
for col in ['Open', 'High', 'Low', 'Adj Close']:
    zero_negative = (sp500_data[col] <= 0).sum()
    print(f"{col}: {zero_negative} zero/negative values")

print(f"\nS&P 500 Basic Statistics:")
print(sp500_data[['Open', 'High', 'Low', 'Adj Close', 'Volume']].describe())

# Bitcoin Data Examination
print("\n" + "="*50)
print("Bitcoin Data Quality Check:")
```

```
print("-" * 30)
print("Missing values by column:")
print(bitcoin_data.isnull().sum())
print(f"\nTotal missing values: {bitcoin_data.isnull().sum().sum()}")
print(f"Data completeness: {((1 - bitcoin_data.isnull().sum().sum() / (len(bitcoin_data) * len(bitcoin_data.columns))) * 100):.2

# Check for zero or negative prices (incorrect observations)
print("\nChecking for zero or negative prices:")
for col in ['Open', 'High', 'Low', 'Adj Close']:
    zero_negative = (bitcoin_data[col] <= 0).sum()
    print(f"{col}: {zero_negative} zero/negative values")

print(f"\nBitcoin Basic Statistics:")
print(bitcoin_data[['Open', 'High', 'Low', 'Adj Close', 'Volume']].describe())
```

In [ ]:
```
# Calculate Logarithmic Returns (Dependent Variable)
# Log returns = ln(P_t / P_{t-1}) = ln(P_t) - ln(P_{t-1})

# Calculate logarithmic returns for S&P 500
sp500_data['Log_Returns'] = np.log(sp500_data['Adj Close'] / sp500_data['Adj Close'].shift(1))

# Calculate logarithmic returns for Bitcoin
bitcoin_data['Log_Returns'] = np.log(bitcoin_data['Adj Close'] / bitcoin_data['Adj Close'].shift(1))

# Remove the first row (NaN due to shift operation)
sp500_clean = sp500_data.dropna()
bitcoin_clean = bitcoin_data.dropna()
```

In [ ]:
```
# Create a comprehensive statistics DataFrame
statistics_data = {
    'Observations': [len(sp500_clean), len(bitcoin_clean)],
    'Mean_Daily_Return': [sp500_clean['Log_Returns'].mean(), bitcoin_clean['Log_Returns'].mean()],
    'Standard_Deviation': [sp500_clean['Log_Returns'].std(), bitcoin_clean['Log_Returns'].std()],
    'Minimum_Return': [sp500_clean['Log_Returns'].min(), bitcoin_clean['Log_Returns'].min()],
    'Maximum_Return': [sp500_clean['Log_Returns'].max(), bitcoin_clean['Log_Returns'].max()],
    'Skewness': [sp500_clean['Log_Returns'].skew(), bitcoin_clean['Log_Returns'].skew()],
    'Kurtosis': [sp500_clean['Log_Returns'].kurtosis(), bitcoin_clean['Log_Returns'].kurtosis()],
    'Infinite_Values': [np.isinf(sp500_clean['Log_Returns']).sum(), np.isinf(bitcoin_clean['Log_Returns']).sum()],
    'NaN_Values': [sp500_clean['Log_Returns'].isnull().sum(), bitcoin_clean['Log_Returns'].isnull().sum()]
}

# Create DataFrame with asset names as index
returns_statistics = pd.DataFrame(statistics_data, index=['S&P_500', 'Bitcoin'])
```

```
print("=== LOGARITHMIC RETURNS STATISTICS DATAFRAME ===")
print(returns_statistics.round(6))

# Display transposed version for better readability
print("\n=== STATISTICS TABLE (TRANSPOSED) ===")
print(returns_statistics.T.round(6))

# Save the statistics DataFrame for later analysis
print(f"\nDataFrame shape: {returns_statistics.shape}")
print(f"DataFrame columns: {list(returns_statistics.columns)}")
print(f"DataFrame index: {list(returns_statistics.index)}")

# Access specific statistics easily
print(f"\nS&P 500 Mean Return: {returns_statistics.loc['S&P_500', 'Mean_Daily_Return']:.6f}")
print(f"Bitcoin Mean Return: {returns_statistics.loc['Bitcoin', 'Mean_Daily_Return']:.6f}")
print(f"Volatility Ratio (Bitcoin/S&P500): {returns_statistics.loc['Bitcoin', 'Standard_Deviation'] / returns_statistics.loc['S&

# Display the DataFrame in different formats for export
returns_statistics.T
```

In [ ]:

```
# Data Visualization and Final Verification

# Verify observation counts match the study description
expected_sp500 = 5537
expected_bitcoin = 3286

# Create visualizations
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, figsize=(15, 10))

# S&P 500 Price Series
ax1.plot(sp500_clean.index, sp500_clean['Adj Close'])
ax1.set_title('S&P 500 Adjusted Close Price (2002-2023)')
ax1.set_ylabel('Price ($)')
ax1.grid(True, alpha=0.3)

# Bitcoin Price Series
ax2.plot(bitcoin_clean.index, bitcoin_clean['Adj Close'])
ax2.set_title('Bitcoin Price (2015-2023)')
ax2.set_ylabel('Price ($)')
ax2.grid(True, alpha=0.3)

# S&P 500 Log Returns
ax3.plot(sp500_clean.index, sp500_clean['Log_Returns'])
ax3.set_title('S&P 500 Logarithmic Returns')
```

```
ax3.set_ylabel('Log Returns')
ax3.set_xlabel('Date')
ax3.axhline(y=0, color='red', linestyle='--', alpha=0.5)
ax3.grid(True, alpha=0.3)

# Bitcoin Log Returns
ax4.plot(bitcoin_clean.index, bitcoin_clean['Log_Returns'])
ax4.set_title('Bitcoin Logarithmic Returns')
ax4.set_ylabel('Log Returns')
ax4.set_xlabel('Date')
ax4.axhline(y=0, color='red', linestyle='--', alpha=0.5)
ax4.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

# Display sample data
print("S&P 500 Data (First 5 rows):")
print(sp500_clean[['Open', 'High', 'Low', 'Adj Close', 'Volume', 'Log_Returns']].head())

print("\nBitcoin Data (First 5 rows):")
print(bitcoin_clean[['Open', 'High', 'Low', 'Adj Close', 'Volume', 'Log_Returns']].head())

print(bitcoin_clean.tail())
```

Cross Validation Data

In [ ]:

```
# Time Series Cross-Validation Implementation
# Novel 3-fold cross-validation scheme with rolling windows

import matplotlib.dates as mdates
from datetime import datetime, timedelta
from dateutil.relativedelta import relativedelta

# print("=== TIME SERIES CROSS-VALIDATION IMPLEMENTATION ===\n")

def create_sp500_cv_splits(data, start_date=None):
    if start_date is None:
        start_date = data.index.min()

    cv_splits = []
    window_start = start_date
```

```
    while True:
        # Define window boundaries
        train_start = window_start
        train_end = train_start + relativedelta(years=3) - timedelta(days=1)

        # Validation periods (8, 16, 24 months)
        val_start = train_end + timedelta(days=1)
        val1_end = val_start + relativedelta(months=8) - timedelta(days=1)  # 8 months
        val2_end = val_start + relativedelta(months=16) - timedelta(days=1) # 16 months
        val3_end = val_start + relativedelta(months=24) - timedelta(days=1) # 24 months (2 years)

        # Test period (1 year)
        test_start = val3_end + timedelta(days=1)
        test_end = test_start + relativedelta(years=1) - timedelta(days=1)

        # Check if we have enough data
        # if test_end > data.index.max():
        if test_end.year > 2024:
            break

        # Create splits for this window
        train_data = data[(data.index >= train_start) & (data.index <= train_end)]

        # Three validation folds
        val1_data = data[(data.index >= val_start) & (data.index <= val1_end)]
        val2_data = data[(data.index >= val_start) & (data.index <= val2_end)]
        val3_data = data[(data.index >= val_start) & (data.index <= val3_end)]

        test_data = data[(data.index >= test_start) & (data.index <= test_end)]

        cv_splits.append({
            'window_id': len(cv_splits) + 1,
            'train': {
                'data': train_data,
                'start': train_start,
                'end': train_end,
                'size': len(train_data)
            },
            'validation': [
                {
                    'fold': 1,
                    'data': val1_data,
                    'start': val_start,
                    'end': val1_end,
```

```
                    'size': len(val1_data),
                    'months': 8
                },
                {
                    'fold': 2,
                    'data': val2_data,
                    'start': val_start,
                    'end': val2_end,
                    'size': len(val2_data),
                    'months': 16
                },
                {
                    'fold': 3,
                    'data': val3_data,
                    'start': val_start,
                    'end': val3_end,
                    'size': len(val3_data),
                    'months': 24
                }
            ],
            'test': {
                'data': test_data,
                'start': test_start,
                'end': test_end,
                'size': len(test_data)
            }
        })

        # Move window forward by 1 year
        window_start += relativedelta(years=1)

    return cv_splits

def create_bitcoin_cv_splits(data, start_date=None):

    if start_date is None:
        # Start from a date that allows for proper window construction
        start_date = datetime(2015, 1, 1)

    cv_splits = []
    window_start = start_date

    # Define the testing period constraint
    test_period_start = datetime(2018, 1, 1)
    test_period_end = datetime(2023, 12, 31)
```

```
    while True:
        # Define window boundaries
        train_start = window_start
        train_end = train_start + relativedelta(years=2) - timedelta(days=1)

        # Validation periods (4, 8, 12 months)
        val_start = train_end + timedelta(days=1)
        val1_end = val_start + relativedelta(months=4) - timedelta(days=1)  # 4 months
        val2_end = val_start + relativedelta(months=8) - timedelta(days=1)  # 8 months
        val3_end = val_start + relativedelta(months=12) - timedelta(days=1) # 12 months

        # Test period (6 months)
        test_start = val3_end + timedelta(days=1)
        test_end = test_start + relativedelta(months=6) - timedelta(days=1)

        # Check constraints
        # if test_end > data.index.max() or test_end > test_period_end:
        #     break
        if test_end.year > 2023:
            break

        # Only include windows where test period is within 2018-2023
        if test_start < test_period_start:
            window_start += relativedelta(months=6)
            continue

        # Create splits for this window
        train_data = data[(data.index >= train_start) & (data.index <= train_end)]

        # Three validation folds
        val1_data = data[(data.index >= val_start) & (data.index <= val1_end)]
        val2_data = data[(data.index >= val_start) & (data.index <= val2_end)]
        val3_data = data[(data.index >= val_start) & (data.index <= val3_end)]

        test_data = data[(data.index >= test_start) & (data.index <= test_end)]

        cv_splits.append({
            'window_id': len(cv_splits) + 1,
            'train': {
                'data': train_data,
                'start': train_start,
                'end': train_end,
                'size': len(train_data)
            },
```

```
            'validation': [
                {
                    'fold': 1,
                    'data': val1_data,
                    'start': val_start,
                    'end': val1_end,
                    'size': len(val1_data),
                    'months': 4
                },
                {
                    'fold': 2,
                    'data': val2_data,
                    'start': val_start,
                    'end': val2_end,
                    'size': len(val2_data),
                    'months': 8
                },
                {
                    'fold': 3,
                    'data': val3_data,
                    'start': val_start,
                    'end': val3_end,
                    'size': len(val3_data),
                    'months': 12
                }
            ],
            'test': {
                'data': test_data,
                'start': test_start,
                'end': test_end,
                'size': len(test_data)
            }
        })

        # Move window forward by 6 months
        window_start += relativedelta(months=6)

    return cv_splits
```

In [ ]:

```
# Apply Cross-Validation Schemes to Data

print("=== GENERATING CROSS-VALIDATION SPLITS ===\n")

# Generate S&P 500 cross-validation splits
```

```python
sp500_cv_splits = create_sp500_cv_splits(sp500_clean)


# Generate Bitcoin cross-validation splits
bitcoin_cv_splits = create_bitcoin_cv_splits(bitcoin_clean)


# Create summary DataFrames
def create_cv_summary(cv_splits, asset_name):
    summary_data = []

    for split in cv_splits:
        # Add training data info
        summary_data.append({
            'Asset': asset_name,
            'Window_ID': split['window_id'],
            'Split_Type': 'Train',
            'Fold': 'N/A',
            'Start_Date': split['train']['start'].strftime('%Y-%m-%d'),
            'End_Date': split['train']['end'].strftime('%Y-%m-%d'),
            'Size': split['train']['size'],
            'Duration_Months': 'N/A'
        })

        # Add validation data info
        for val_fold in split['validation']:
            summary_data.append({
                'Asset': asset_name,
                'Window_ID': split['window_id'],
                'Split_Type': 'Validation',
                'Fold': val_fold['fold'],
                'Start_Date': val_fold['start'].strftime('%Y-%m-%d'),
                'End_Date': val_fold['end'].strftime('%Y-%m-%d'),
                'Size': val_fold['size'],
                'Duration_Months': val_fold['months']
            })

        # Add test data info
        summary_data.append({
            'Asset': asset_name,
            'Window_ID': split['window_id'],
            'Split_Type': 'Test',
            'Fold': 'N/A',
            'Start_Date': split['test']['start'].strftime('%Y-%m-%d'),
            'End_Date': split['test']['end'].strftime('%Y-%m-%d'),
```

```
            'Size': split['test']['size'],
            'Duration_Months': 'N/A'
        })

    return pd.DataFrame(summary_data)

# Create summary DataFrames
sp500_cv_summary = create_cv_summary(sp500_cv_splits, 'S&P_500')
bitcoin_cv_summary = create_cv_summary(bitcoin_cv_splits, 'Bitcoin')

# Combined summary
cv_summary_combined = pd.concat([sp500_cv_summary, bitcoin_cv_summary], ignore_index=True)


# Display first few windows for each asset
print("\n=== S&P 500 CV WINDOWS (First set) ===")
for i, split in enumerate(sp500_cv_splits[:1]):
    print(f"\nWindow {split['window_id']}:")
    print(f"  Train: {split['train']['start'].strftime('%Y-%m-%d')} to {split['train']['end'].strftime('%Y-%m-%d')} ({split['tra
    print(f"  Validation Folds:")
    for val_fold in split['validation']:
        print(f"    Fold {val_fold['fold']} ({val_fold['months']}mo): {val_fold['start'].strftime('%Y-%m-%d')} to {val_fold['end
    print(f"  Test: {split['test']['start'].strftime('%Y-%m-%d')} to {split['test']['end'].strftime('%Y-%m-%d')} ({split['test']

print("\n=== BITCOIN CV WINDOWS (First set) ===")
for i, split in enumerate(bitcoin_cv_splits[:1]):
    print(f"\nWindow {split['window_id']}:")
    print(f"  Train: {split['train']['start'].strftime('%Y-%m-%d')} to {split['train']['end'].strftime('%Y-%m-%d')} ({split['tra
    print(f"  Validation Folds:")
    for val_fold in split['validation']:
        print(f"    Fold {val_fold['fold']} ({val_fold['months']}mo): {val_fold['start'].strftime('%Y-%m-%d')} to {val_fold['end
    print(f"  Test: {split['test']['start'].strftime('%Y-%m-%d')} to {split['test']['end'].strftime('%Y-%m-%d')} ({split['test']

# Display summary statistics
print("\n=== CV SPLIT STATISTICS ===")
split_stats = cv_summary_combined.groupby(['Asset', 'Split_Type']).agg({
    'Size': ['mean', 'std', 'min', 'max'],
    'Window_ID': 'count'
}).round(0)
print(split_stats)
```

In [ ]:
```
# Visualize Cross-Validation Scheme

def plot_cv_timeline(cv_splits, asset_name, max_windows=8):
```

```python
fig, ax = plt.subplots(figsize=(16, max(6, len(cv_splits[:max_windows]) * 1.5)))

# Colors for different split types
colors = {
    'train': '#2E8B57',
    'val_fold1': '#4169E1',
    'val_fold2': '#1E90FF',
    'val_fold3': '#87CEEB',
    'test': '#DC143C'
}

y_positions = []

for i, split in enumerate(cv_splits[:max_windows]):
    y_pos = len(cv_splits[:max_windows]) - i - 1
    y_positions.append(y_pos)

    # Plot training period
    ax.barh(y_pos, (split['train']['end'] - split['train']['start']).days,
            left=split['train']['start'], height=0.6,
            color=colors['train'], alpha=0.8, label='Train' if i == 0 else "")

    # Plot validation periods
    val_colors = ['val_fold1', 'val_fold2', 'val_fold3']
    for j, val_fold in enumerate(split['validation']):
        ax.barh(y_pos + 0.1 + j*0.15, (val_fold['end'] - val_fold['start']).days,
                left=val_fold['start'], height=0.12,
                color=colors[val_colors[j]], alpha=0.8,
                label=f'Val Fold {j+1} ({val_fold["months"]}mo)' if i == 0 else "")

    # Plot test period
    ax.barh(y_pos, (split['test']['end'] - split['test']['start']).days,
            left=split['test']['start'], height=0.6,
            color=colors['test'], alpha=0.8, label='Test' if i == 0 else "")

    # Add window labels
    ax.text(split['train']['start'], y_pos, f'W{split["window_id"]}',
            verticalalignment='center', fontsize=9, fontweight='bold')

# Formatting
ax.set_ylim(-0.5, len(cv_splits[:max_windows]) - 0.5)
ax.set_ylabel('CV Windows (Newest to Oldest)', fontsize=12)
ax.set_xlabel('Time Period', fontsize=12)
ax.set_title(f'{asset_name} Cross-Validation Timeline\n({len(cv_splits)} Total Windows, Showing First {min(max_windows, len(
```

```
                fontsize=14, fontweight='bold')

    # Format x-axis
    ax.xaxis.set_major_formatter(mdates.DateFormatter('%Y-%m'))
    ax.xaxis.set_major_locator(mdates.YearLocator())
    plt.xticks(rotation=45)

    # Add legend
    ax.legend(loc='upper right', bbox_to_anchor=(1, 1), frameon=True, fancybox=True, shadow=True)

    # Add grid
    ax.grid(True, alpha=0.3, axis='x')

    plt.tight_layout()
    return fig, ax

print("=== CROSS-VALIDATION SCHEME VISUALIZATION ===\n")

# Create visualizations
fig1, ax1 = plot_cv_timeline(sp500_cv_splits, 'S&P 500', max_windows=8)
plt.show()

fig2, ax2 = plot_cv_timeline(bitcoin_cv_splits, 'Bitcoin', max_windows=8)
plt.show()

# Create a comprehensive comparison chart
def create_cv_comparison_chart():

    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(20, 8))

    # S&P 500 scheme visualization
    y_pos = 1

    # S&P 500 scheme
    ax1.barh(y_pos, 3*365, left=0, height=0.6, color='#2E8B57', alpha=0.8, label='Train (3yr)')
    ax1.barh(y_pos+0.1, 8*30, left=3*365, height=0.15, color='#4169E1', alpha=0.8, label='Val Fold 1 (8mo)')
    ax1.barh(y_pos+0.25, 16*30, left=3*365, height=0.15, color='#1E90FF', alpha=0.8, label='Val Fold 2 (16mo)')
    ax1.barh(y_pos+0.4, 24*30, left=3*365, height=0.15, color='#87CEEB', alpha=0.8, label='Val Fold 3 (24mo)')
    ax1.barh(y_pos, 1*365, left=5*365, height=0.6, color='#DC143C', alpha=0.8, label='Test (1yr)')

    ax1.set_xlim(0, 6*365)
    ax1.set_ylim(0.5, 1.8)
    ax1.set_xlabel('Days')
    ax1.set_title('S&P 500 CV Scheme\n(6-year windows)', fontweight='bold')
    ax1.legend()
```

```
    ax1.grid(True, alpha=0.3)

    # Bitcoin scheme
    ax2.barh(y_pos, 2*365, left=0, height=0.6, color='#2E8B57', alpha=0.8, label='Train (2yr)')
    ax2.barh(y_pos+0.1, 4*30, left=2*365, height=0.15, color='#4169E1', alpha=0.8, label='Val Fold 1 (4mo)')
    ax2.barh(y_pos+0.25, 8*30, left=2*365, height=0.15, color='#1E90FF', alpha=0.8, label='Val Fold 2 (8mo)')
    ax2.barh(y_pos+0.4, 12*30, left=2*365, height=0.15, color='#87CEEB', alpha=0.8, label='Val Fold 3 (12mo)')
    ax2.barh(y_pos, 6*30, left=3*365, height=0.6, color='#DC143C', alpha=0.8, label='Test (6mo)')

    ax2.set_xlim(0, 3.5*365)
    ax2.set_ylim(0.5, 1.8)
    ax2.set_xlabel('Days')
    ax2.set_title('Bitcoin CV Scheme\n(~3.5-year windows)', fontweight='bold')
    ax2.legend()
    ax2.grid(True, alpha=0.3)

    plt.suptitle('Time Series Cross-Validation Scheme Comparison', fontsize=16, fontweight='bold')
    plt.tight_layout()
    return fig

fig3 = create_cv_comparison_chart()
plt.show()

# Create summary table
cv_scheme_summary = pd.DataFrame({
    'Asset': ['S&P 500', 'Bitcoin'],
    'Total_Windows': [len(sp500_cv_splits), len(bitcoin_cv_splits)],
    'Window_Length': ['6 years', '~3.5 years'],
    'Train_Period': ['3 years', '2 years'],
    'Validation_Folds': ['8/16/24 months', '4/8/12 months'],
    'Test_Period': ['1 year', '6 months'],
    'Window_Shift': ['1 year', '6 months'],
    'Test_Coverage': [
        f"{sp500_cv_splits[0]['test']['start'].strftime('%Y-%m-%d')} to {sp500_cv_splits[-1]['test']['end'].strftime('%Y-%m-%d')
        f"{bitcoin_cv_splits[0]['test']['start'].strftime('%Y-%m-%d')} to {bitcoin_cv_splits[-1]['test']['end'].strftime('%Y-%m-
    ]
})

print("\n=== CROSS-VALIDATION SCHEME SUMMARY ===")
print(cv_scheme_summary.to_string(index=False))
```

In [ ]:

```
# Utility Functions for Cross-Validation Data Access

def get_cv_data(cv_splits, window_id, fold=None, return_type='data'):
```

```
    split = next((s for s in cv_splits if s['window_id'] == window_id), None)
    if split is None:
        raise ValueError(f"Window ID {window_id} not found")

    if return_type == 'train':
        return split['train']['data']
    elif return_type == 'test':
        return split['test']['data']
    elif return_type == 'validation':
        if fold is None:
            raise ValueError("Fold number must be specified for validation data")
        if fold not in [1, 2, 3]:
            raise ValueError("Fold must be 1, 2, or 3")
        return split['validation'][fold-1]['data']
    else:
        return split

# Example usage functions
def demonstrate_cv_usage():

    print("=== CROSS-VALIDATION DATA ACCESS EXAMPLES ===\n")

    # Example 1: Get training data from first S&P 500 window
    train_data_sp500 = get_cv_data(sp500_cv_splits, window_id=1, return_type='train')
    print(f"S&P 500 Window 1 - Training data shape: {train_data_sp500.shape}")
    print(f"Training period: {train_data_sp500.index.min()} to {train_data_sp500.index.max()}")

    # Example 2: Get validation fold 2 data from first S&P 500 window
    val_data_sp500 = get_cv_data(sp500_cv_splits, window_id=1, fold=2, return_type='validation')
    print(f"\nS&P 500 Window 1 - Validation Fold 2 shape: {val_data_sp500.shape}")
    print(f"Validation period: {val_data_sp500.index.min()} to {val_data_sp500.index.max()}")

    # Example 3: Get test data from first Bitcoin window
    test_data_bitcoin = get_cv_data(bitcoin_cv_splits, window_id=1, return_type='test')
    print(f"\nBitcoin Window 1 - Test data shape: {test_data_bitcoin.shape}")
    print(f"Test period: {test_data_bitcoin.index.min()} to {test_data_bitcoin.index.max()}")

    return train_data_sp500, val_data_sp500, test_data_bitcoin

# Run demonstration
sample_train, sample_val, sample_test = demonstrate_cv_usage()


# Save CV splits for later use (optional)
cv_implementation_summary = {
```

```
    'sp500_cv_splits': sp500_cv_splits,
    'bitcoin_cv_splits': bitcoin_cv_splits,
    'sp500_summary': sp500_cv_summary,
    'bitcoin_summary': bitcoin_cv_summary,
    'combined_summary': cv_summary_combined,
    'scheme_comparison': cv_scheme_summary
}
```

ARIMA

Test for stationarity

In [ ]:
```
from statsmodels.tsa.stattools import adfuller

sp500_adf_test = adfuller(sp500_clean['Log_Returns'])
# Output the results
print('ADF Statistic: %f' % sp500_adf_test[0])
print('p-value: %f' % sp500_adf_test[1])
```

A p-value < 0.05 indicates that the S&P500 dataset is stationary.

In [ ]:
```
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf

plot_acf(sp500_clean['Log_Returns'], lags=40)
plot_pacf(sp500_clean['Log_Returns'], lags=40)
plt.show()
```

In [ ]:
```
bitcoin_adf_test = adfuller(bitcoin_clean['Log_Returns'])
# Output the results
print('ADF Statistic: %f' % bitcoin_adf_test[0])
print('p-value: %f' % bitcoin_adf_test[1])

plot_acf(bitcoin_clean['Log_Returns'], lags=40)
plot_pacf(bitcoin_clean['Log_Returns'], lags=40)
plt.show()
```

A p-value < 0.05 indicates that the Bitcoin dataset is stationary.

In [ ]:
```
# ARIMA Model Implementation with AIC-based Selection
# Modern approach replacing traditional Box-Jenkins methodology
```

```
import itertools
from statsmodels.tsa.arima.model import ARIMA
from statsmodels.tsa.stattools import adfuller, kpss
from statsmodels.stats.diagnostic import acorr_ljungbox
import warnings
from sklearn.metrics import mean_squared_error, mean_absolute_error
from scipy import stats
import time

# Suppress convergence warnings for cleaner output
warnings.filterwarnings('ignore', category=UserWarning)
warnings.filterwarnings('ignore', category=RuntimeWarning)

print("=== AIC-BASED ARIMA MODEL SELECTION IMPLEMENTATION ===\n")

def find_optimal_arima_order(data, max_p=5, max_d=2, max_q=5, seasonal=False,
                             information_criterion='aic', verbose=False):

    best_ic = np.inf
    best_params = None
    best_model = None
    results_log = []

    # Create parameter grid
    if seasonal:
        # For seasonal ARIMA (not implemented in this study)
        param_grid = itertools.product(range(max_p+1), range(max_d+1), range(max_q+1),
                                       range(2), range(2), range(2), [12])
    else:
        # Standard ARIMA grid search
        param_grid = itertools.product(range(max_p+1), range(max_d+1), range(max_q+1))

    total_combinations = (max_p+1) * (max_d+1) * (max_q+1)

    if verbose:
        print(f"Testing {total_combinations} ARIMA parameter combinations...")
        start_time = time.time()

    for i, params in enumerate(param_grid):
        p, d, q = params[:3]

        # Skip if model is too simple (all parameters zero)
        if p == 0 and d == 0 and q == 0:
            continue
```

```
        try:
            # Fit ARIMA model
            model = ARIMA(data, order=(p, d, q))
            fitted_model = model.fit()

            # Get information criterion value
            if information_criterion.lower() == 'aic':
                ic_value = fitted_model.aic
            elif information_criterion.lower() == 'bic':
                ic_value = fitted_model.bic
            elif information_criterion.lower() == 'hqic':
                ic_value = fitted_model.hqic
            else:
                ic_value = fitted_model.aic

            # Store results
            results_log.append({
                'order': (p, d, q),
                'aic': fitted_model.aic,
                'bic': fitted_model.bic,
                'hqic': fitted_model.hqic,
                'llf': fitted_model.llf,
                'converged': fitted_model.mle_retvals['converged'] if hasattr(fitted_model, 'mle_retvals') else True
            })

            # Update best model if current is better
            if ic_value < best_ic:
                best_ic = ic_value
                best_params = (p, d, q)
                best_model = fitted_model

        except Exception as e:
            # Log failed fits
            results_log.append({
                'order': (p, d, q),
                'aic': np.nan,
                'bic': np.nan,
                'hqic': np.nan,
                'llf': np.nan,
                'converged': False,
                'error': str(e)
            })

            if verbose and i % 10 == 0:
```

```
                print(f"Failed to fit ARIMA{params}: {str(e)[:50]}...")

        if verbose and (i + 1) % 20 == 0:
            elapsed = time.time() - start_time
            progress = (i + 1) / total_combinations * 100
            print(f"Progress: {progress:.1f}% ({i+1}/{total_combinations}) | "
                  f"Best so far: ARIMA{best_params} ({information_criterion.upper()}={best_ic:.4f})")

    if verbose:
        total_time = time.time() - start_time
        print(f"\nGrid search completed in {total_time:.2f} seconds")
        print(f"Best model: ARIMA{best_params} with {information_criterion.upper()}={best_ic:.4f}")

    # Create results summary
    results_df = pd.DataFrame(results_log)
    successful_fits = results_df[results_df['converged'] == True]

    return {
        'best_order': best_params,
        'best_model': best_model,
        'best_ic_value': best_ic,
        'information_criterion': information_criterion,
        'results_df': results_df,
        'successful_fits': len(successful_fits),
        'total_attempts': len(results_log),
        'success_rate': len(successful_fits) / len(results_log) * 100
    }

def evaluate_arima_model(model, train_data, test_data, model_order):
    # Generate forecasts
    n_forecast = len(test_data)
    forecast_result = model.get_forecast(steps=n_forecast)
    forecasts = forecast_result.predicted_mean
    forecast_ci = forecast_result.conf_int()

    # Calculate performance metrics
    mse = mean_squared_error(test_data, forecasts)
    rmse = np.sqrt(mse)
    mae = mean_absolute_error(test_data, forecasts)
    mape = np.mean(np.abs((test_data - forecasts) / test_data)) * 100

    # Calculate R² score for fair comparison with LSTM and SVM
    ss_res = np.sum((test_data - forecasts) ** 2)
    ss_tot = np.sum((test_data - np.mean(test_data)) ** 2)
    r2 = 1 - (ss_res / ss_tot) if ss_tot > 0 else 0
```

```
    # Direction accuracy (for returns)
    direction_actual = np.sign(test_data.values[1:])
    direction_forecast = np.sign(forecasts.values[1:])
    direction_accuracy = np.mean(direction_actual == direction_forecast) * 100

    # Residual diagnostics
    residuals = model.resid

    # Ljung-Box test for serial correlation in residuals
    lb_test = acorr_ljungbox(residuals, lags=10, return_df=False)

    # Normality test (Jarque-Bera)
    jb_stat, jb_pvalue = stats.jarque_bera(residuals)

    # Heteroskedasticity test (simple approach)
    residuals_squared = residuals ** 2
    arch_stat, arch_pvalue = acorr_ljungbox(residuals_squared, lags=5, return_df=False)

    return {
        'model_order': model_order,
        'forecasts': forecasts,
        'forecast_ci': forecast_ci,
        'performance_metrics': {
            'mse': mse,
            'rmse': rmse,
            'mae': mae,
            'mape': mape,
            'r2': r2,
            'direction_accuracy': direction_accuracy
        },
        'diagnostic_tests': {
            'ljung_box_stat': lb_test['lb_stat'].iloc[-1],
            'ljung_box_pvalue': lb_test['lb_pvalue'].iloc[-1],
            'jarque_bera_stat': jb_stat,
            'jarque_bera_pvalue': jb_pvalue,
            'arch_stat': arch_stat[-1] if isinstance(arch_stat, np.ndarray) else arch_stat,
            'arch_pvalue': arch_pvalue[-1] if isinstance(arch_pvalue, np.ndarray) else arch_pvalue
        },
        'residuals': residuals
    }
```

In [ ]:

```
# Cross-Validation Integration for ARIMA Model Selection
# Implement the complete methodology with hyperparameter optimization
```

```python
def run_arima_cross_validation(cv_splits, data_clean, asset_name, max_p=3, max_d=2, max_q=3,
                               information_criterion='aic', verbose=True):

    print(f"\n=== {asset_name.upper()} ARIMA CROSS-VALIDATION ===")
    print(f"Running AIC-based model selection across {len(cv_splits)} windows...")
    print(f"Parameter search space: p∈[0,{max_p}], d∈[0,{max_d}], q∈[0,{max_q}]")
    print("-" * 80)

    all_results = []
    model_selection_summary = []

    for window_idx, split in enumerate(cv_splits):
        window_id = split['window_id']

        if verbose:
            print(f"\n Processing Window {window_id}/{len(cv_splits)}...")
            print(f"   Train: {split['train']['start'].strftime('%Y-%m-%d')} to {split['train']['end'].strftime('%Y-%m-%d')} ({s
            print(f"   Test:  {split['test']['start'].strftime('%Y-%m-%d')} to {split['test']['end'].strftime('%Y-%m-%d')} ({spl

        # Extract data
        train_data = split['train']['data']['Log_Returns']
        test_data = split['test']['data']['Log_Returns']

        # STEP 1: Model Selection using Training Data
        if verbose:
            print(f"    Model selection using {information_criterion.upper()} criterion...")

        selection_result = find_optimal_arima_order(
            train_data,
            max_p=max_p,
            max_d=max_d,
            max_q=max_q,
            information_criterion=information_criterion,
            verbose=False  # Keep individual window selection quiet
        )

        if selection_result['best_model'] is None:
            print(f"    Failed to find suitable model for Window {window_id}")
            continue

        best_order = selection_result['best_order']

        # STEP 2: Hyperparameter Validation using Validation Folds
        if verbose:
```

```
            print(f"    Validating ARIMA{best_order} across 3 validation folds...")

        validation_scores = []

        for val_fold in split['validation']:
            fold_num = val_fold['fold']
            val_data = val_fold['data']['Log_Returns']

            try:
                # Fit model on training data and evaluate on validation fold
                val_model = ARIMA(train_data, order=best_order).fit()
                val_forecasts = val_model.get_forecast(steps=len(val_data)).predicted_mean
                val_rmse = np.sqrt(mean_squared_error(val_data, val_forecasts))
                validation_scores.append(val_rmse)

            except Exception as e:
                if verbose:
                    print(f"      Validation fold {fold_num} failed: {str(e)[:50]}...")
                validation_scores.append(np.inf)

        avg_validation_rmse = np.mean(validation_scores)

        # STEP 3: Final Model Training and Out-of-Sample Evaluation
        if verbose:
            print(f"    Final evaluation on test data...")

        try:
            # Re-fit the model on training data
            final_model = ARIMA(train_data, order=best_order).fit()

            # Evaluate on test data
            evaluation = evaluate_arima_model(final_model, train_data, test_data, best_order)

            # Store comprehensive results
            window_result = {
                'window_id': window_id,
                'asset': asset_name,
                'train_period': f"{split['train']['start'].strftime('%Y-%m-%d')} to {split['train']['end'].strftime('%Y-%m-%d')}
                'test_period': f"{split['test']['start'].strftime('%Y-%m-%d')} to {split['test']['end'].strftime('%Y-%m-%d')}",
                'train_size': split['train']['size'],
                'test_size': split['test']['size'],
                'best_order': best_order,
                'model_selection': selection_result,
                'validation_scores': validation_scores,
                'avg_validation_rmse': avg_validation_rmse,
```

```
                'evaluation': evaluation,
                'final_model': final_model
            }

            all_results.append(window_result)

            # Summary for quick reference
            model_selection_summary.append({
                'Window': window_id,
                'Best_Order': f"ARIMA{best_order}",
                'AIC': selection_result['best_ic_value'],
                'Validation_RMSE': avg_validation_rmse,
                'Test_RMSE': evaluation['performance_metrics']['rmse'],
                'Test_MAE': evaluation['performance_metrics']['mae'],
                'Test_R2': evaluation['performance_metrics']['r2'],
                'Direction_Accuracy': evaluation['performance_metrics']['direction_accuracy'],
                'Ljung_Box_p': evaluation['diagnostic_tests']['ljung_box_pvalue']
            })

            if verbose:
                print(f"    ARIMA{best_order}: Test RMSE={evaluation['performance_metrics']['rmse']:.6f}, "
                      f"Direction Acc={evaluation['performance_metrics']['direction_accuracy']:.1f}%")

        except Exception as e:
            print(f"    Final evaluation failed for Window {window_id}: {str(e)}")
            continue

    # Create summary DataFrame
    summary_df = pd.DataFrame(model_selection_summary)

    # Calculate overall performance statistics
    if len(summary_df) > 0:
        performance_summary = {
            'total_windows': len(cv_splits),
            'successful_windows': len(summary_df),
            'success_rate': len(summary_df) / len(cv_splits) * 100,
            'avg_test_rmse': summary_df['Test_RMSE'].mean(),
            'std_test_rmse': summary_df['Test_RMSE'].std(),
            'avg_test_mae': summary_df['Test_MAE'].mean(),
            'avg_r2': summary_df['Test_R2'].mean(),
            'avg_direction_accuracy': summary_df['Direction_Accuracy'].mean(),
            'avg_validation_rmse': summary_df['Validation_RMSE'].mean(),
            'most_common_order': summary_df['Best_Order'].mode().iloc[0] if len(summary_df) > 0 else None
        }
    else:
```

```
        performance_summary = None

    print(f"\n{'='*80}")
    print(f"{asset_name.upper()} ARIMA CROSS-VALIDATION COMPLETE")
    print(f"{'='*80}")

    if performance_summary:
        print(f" Successfully processed {performance_summary['successful_windows']}/{performance_summary['total_windows']} windo
        print(f" Average Test RMSE: {performance_summary['avg_test_rmse']:.6f} ± {performance_summary['std_test_rmse']:.6f}")
        print(f" Average Direction Accuracy: {performance_summary['avg_direction_accuracy']:.2f}%")
        print(f" Most Common Model: {performance_summary['most_common_order']}")
    else:
        print(" No successful model fits achieved")

    return {
        'asset_name': asset_name,
        'all_results': all_results,
        'summary_df': summary_df,
        'performance_summary': performance_summary,
        'methodology': {
            'approach': 'AIC-based automated selection',
            'information_criterion': information_criterion,
            'parameter_space': f'p∈[0,{max_p}], d∈[0,{max_d}], q∈[0,{max_q}]',
            'cross_validation': '3-fold temporal validation',
            'evaluation_metric': 'Out-of-sample RMSE and direction accuracy'
        }
    }
```

In [ ]:

```
# Execute Complete ARIMA Analysis with AIC-based Model Selection

# Run ARIMA analysis for S&P 500
print("\n S&P 500 ANALYSIS")
sp500_arima_results = run_arima_cross_validation(
    cv_splits=sp500_cv_splits,
    data_clean=sp500_clean,
    asset_name='S&P 500',
    max_p=3,          # Conservative parameter space for computational efficiency
    max_d=1,          # Returns are already stationary (d=1 for potential overdifferencing test)
    max_q=3,
    information_criterion='aic',
    verbose=True
)

# Run ARIMA analysis for Bitcoin
```

```
print("\n BITCOIN ANALYSIS")
bitcoin_arima_results = run_arima_cross_validation(
    cv_splits=bitcoin_cv_splits,
    data_clean=bitcoin_clean,
    asset_name='Bitcoin',
    max_p=3,
    max_d=1,
    max_q=3,
    information_criterion='aic',
    verbose=True
)
```

In [ ]:

```
def create_arima_results_summary(sp500_results, bitcoin_results):

    print("\n" + "="*100)
    print("COMPREHENSIVE ARIMA ANALYSIS RESULTS")
    print("="*100)

    # Combined summary statistics
    summary_stats = {
        'Asset': ['S&P 500', 'Bitcoin'],
        'Total_Windows': [
            sp500_results['performance_summary']['total_windows'] if sp500_results['performance_summary'] else 0,
            bitcoin_results['performance_summary']['total_windows'] if bitcoin_results['performance_summary'] else 0
        ],
        'Avg_Test_RMSE': [
            sp500_results['performance_summary']['avg_test_rmse'] if sp500_results['performance_summary'] else np.nan,
            bitcoin_results['performance_summary']['avg_test_rmse'] if bitcoin_results['performance_summary'] else np.nan
        ],
        'Avg_Direction_Accuracy_%': [
            sp500_results['performance_summary']['avg_direction_accuracy'] if sp500_results['performance_summary'] else np.nan,
            bitcoin_results['performance_summary']['avg_direction_accuracy'] if bitcoin_results['performance_summary'] else np.n
        ],
        'Most_Common_Model': [
            sp500_results['performance_summary']['most_common_order'] if sp500_results['performance_summary'] else 'N/A',
            bitcoin_results['performance_summary']['most_common_order'] if bitcoin_results['performance_summary'] else 'N/A'
        ]
    }

    summary_df = pd.DataFrame(summary_stats)

    print("\n OVERALL PERFORMANCE SUMMARY")
    print("-" * 50)
    print(summary_df.to_string(index=False, float_format='%.4f'))
```

```
    # Detailed window-by-window results
    if len(sp500_results['summary_df']) > 0:
        print(f"\n S&P 500 DETAILED RESULTS ({len(sp500_results['summary_df'])} windows)")
        print("-" * 50)
        print(sp500_results['summary_df'].round(6).to_string(index=False))

    if len(bitcoin_results['summary_df']) > 0:
        print(f"\nBITCOIN DETAILED RESULTS ({len(bitcoin_results['summary_df'])} windows)")
        print("-" * 50)
        print(bitcoin_results['summary_df'].round(6).to_string(index=False))

    return summary_df

def plot_arima_performance_analysis(sp500_results, bitcoin_results):

    fig, axes = plt.subplots(2, 3, figsize=(18, 12))
    fig.suptitle('ARIMA Model Performance Analysis\nAIC-based Selection vs Traditional Box-Jenkins',
                 fontsize=16, fontweight='bold')

    # Extract data for plotting
    sp500_df = sp500_results['summary_df'] if len(sp500_results['summary_df']) > 0 else pd.DataFrame()
    bitcoin_df = bitcoin_results['summary_df'] if len(bitcoin_results['summary_df']) > 0 else pd.DataFrame()

    # Plot 1: Test RMSE Comparison
    ax1 = axes[0, 0]
    if len(sp500_df) > 0 and len(bitcoin_df) > 0:
        ax1.boxplot([sp500_df['Test_RMSE'], bitcoin_df['Test_RMSE']],
                    labels=['S&P 500', 'Bitcoin'], patch_artist=True,
                    boxprops=dict(facecolor='lightblue', alpha=0.7))
        ax1.set_title('Test RMSE Distribution')
        ax1.set_ylabel('RMSE')
        ax1.grid(True, alpha=0.3)

    # Plot 2: Direction Accuracy Comparison
    ax2 = axes[0, 1]
    if len(sp500_df) > 0 and len(bitcoin_df) > 0:
        ax2.boxplot([sp500_df['Direction_Accuracy'], bitcoin_df['Direction_Accuracy']],
                    labels=['S&P 500', 'Bitcoin'], patch_artist=True,
                    boxprops=dict(facecolor='lightgreen', alpha=0.7))
        ax2.set_title('Direction Accuracy Distribution')
        ax2.set_ylabel('Accuracy (%)')
        ax2.axhline(y=50, color='red', linestyle='--', alpha=0.5, label='Random (50%)')
        ax2.legend()
        ax2.grid(True, alpha=0.3)
```

```
# Plot 3: Model Order Distribution
ax3 = axes[0, 2]
combined_orders = []
combined_assets = []

if len(sp500_df) > 0:
    combined_orders.extend(sp500_df['Best_Order'].tolist())
    combined_assets.extend(['S&P 500'] * len(sp500_df))
if len(bitcoin_df) > 0:
    combined_orders.extend(bitcoin_df['Best_Order'].tolist())
    combined_assets.extend(['Bitcoin'] * len(bitcoin_df))

if combined_orders:
    order_counts = pd.Series(combined_orders).value_counts()
    order_counts.head(10).plot(kind='bar', ax=ax3, color='coral', alpha=0.7)
    ax3.set_title('Most Frequent ARIMA Orders')
    ax3.set_xlabel('ARIMA Order')
    ax3.set_ylabel('Frequency')
    ax3.tick_params(axis='x', rotation=45)
    ax3.grid(True, alpha=0.3)

# Plot 4: RMSE over Time (S&P 500)
ax4 = axes[1, 0]
if len(sp500_df) > 0:
    ax4.plot(sp500_df['Window'], sp500_df['Test_RMSE'], 'o-', color='blue', alpha=0.7, label='Test RMSE')
    ax4.plot(sp500_df['Window'], sp500_df['Validation_RMSE'], 's--', color='lightblue', alpha=0.7, label='Validation RMSE')
    ax4.set_title('S&P 500 RMSE Evolution')
    ax4.set_xlabel('Window')
    ax4.set_ylabel('RMSE')
    ax4.legend()
    ax4.grid(True, alpha=0.3)

# Plot 5: RMSE over Time (Bitcoin)
ax5 = axes[1, 1]
if len(bitcoin_df) > 0:
    ax5.plot(bitcoin_df['Window'], bitcoin_df['Test_RMSE'], 'o-', color='orange', alpha=0.7, label='Test RMSE')
    ax5.plot(bitcoin_df['Window'], bitcoin_df['Validation_RMSE'], 's--', color='gold', alpha=0.7, label='Validation RMSE')
    ax5.set_title('Bitcoin RMSE Evolution')
    ax5.set_xlabel('Window')
    ax5.set_ylabel('RMSE')
    ax5.legend()
    ax5.grid(True, alpha=0.3)

# Plot 6: Diagnostic Tests Summary
```

```
    ax6 = axes[1, 2]
    if len(sp500_df) > 0 and len(bitcoin_df) > 0:
        # Count models that pass Ljung-Box test (p > 0.05)
        sp500_pass_lb = (sp500_df['Ljung_Box_p'] > 0.05).sum()
        bitcoin_pass_lb = (bitcoin_df['Ljung_Box_p'] > 0.05).sum()

        categories = ['S&P 500\nPass LB Test', 'Bitcoin\nPass LB Test']
        values = [sp500_pass_lb/len(sp500_df)*100 if len(sp500_df) > 0 else 0,
                  bitcoin_pass_lb/len(bitcoin_df)*100 if len(bitcoin_df) > 0 else 0]

        bars = ax6.bar(categories, values, color=['blue', 'orange'], alpha=0.7)
        ax6.set_title('Residual Diagnostics\n(% Models Passing Ljung-Box)')
        ax6.set_ylabel('Percentage (%)')
        ax6.axhline(y=95, color='green', linestyle='--', alpha=0.5, label='Target (95%)')
        ax6.legend()
        ax6.grid(True, alpha=0.3)

        # Add value labels on bars
        for bar, value in zip(bars, values):
            height = bar.get_height()
            ax6.text(bar.get_x() + bar.get_width()/2., height + 1,
                    f'{value:.1f}%', ha='center', va='bottom')

    plt.tight_layout()
    return fig


# Execute comprehensive analysis
summary_df = create_arima_results_summary(sp500_arima_results, bitcoin_arima_results)
fig = plot_arima_performance_analysis(sp500_arima_results, bitcoin_arima_results)
plt.show()

print(f"Processed {len(sp500_cv_splits) + len(bitcoin_cv_splits)} total cross-validation windows")
```

LSTM

In [ ]:

```
# LSTM Model Implementation for Volatility Forecasting
# Deep learning approach to capture complex temporal patterns

import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.models import Sequential
```

```
from tensorflow.keras.layers import LSTM, Dense, Dropout
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau
from tensorflow.keras.optimizers import Adam
from sklearn.preprocessing import StandardScaler
import warnings
warnings.filterwarnings('ignore')

# Set random seeds for reproducibility
np.random.seed(42)
tf.random.set_seed(42)
```

In [ ]:

```
# Data Preparation Functions for LSTM
# Transform time series data into sequences for LSTM training

def create_sequences(data, lookback_window):
    X, y = [], []

    for i in range(lookback_window, len(data)):
        X.append(data[i-lookback_window:i])
        y.append(data[i])

    X = np.array(X)
    y = np.array(y)

    # Reshape X for LSTM input: (samples, timesteps, features)
    if len(X.shape) == 2:
        X = X.reshape(X.shape[0], X.shape[1], 1)

    return X, y


def calculate_realized_volatility(returns, window=5):
    # Calculate squared returns
    squared_returns = returns ** 2

    # Rolling sum of squared returns
    realized_vol = np.sqrt(squared_returns.rolling(window=window).sum())

    return realized_vol


def prepare_lstm_data(data, lookback_window=20, volatility_window=5,
                      forecast_horizon=1, target_type='volatility'):
    # Extract log returns
```

```
    returns = data['Log_Returns'].values

    if target_type == 'volatility':
        # Calculate realized volatility
        realized_vol = calculate_realized_volatility(data['Log_Returns'],
                                                     window=volatility_window)
        target_data = realized_vol.values

        # Remove NaN values from volatility calculation
        valid_idx = ~np.isnan(target_data)
        target_data = target_data[valid_idx]
        returns = returns[valid_idx]

    else:  # returns forecasting
        target_data = returns

    # Create sequences
    X, y = create_sequences(returns, lookback_window)

    # Adjust target based on forecast horizon
    if forecast_horizon > 1:
        X = X[:-forecast_horizon+1]
        y = y[forecast_horizon-1:]

    # For volatility forecasting, use future volatility as target
    if target_type == 'volatility':
        # Align volatility targets with returns sequences
        vol_start_idx = lookback_window + volatility_window - 1
        y_vol = target_data[vol_start_idx:vol_start_idx+len(y)]

        # Ensure alignment
        min_len = min(len(X), len(y_vol))
        X = X[:min_len]
        y = y_vol[:min_len]

    # Normalize features
    scaler_X = StandardScaler()
    X_scaled = scaler_X.fit_transform(X.reshape(-1, X.shape[-1])).reshape(X.shape)

    # Normalize targets
    scaler_y = StandardScaler()
    y_scaled = scaler_y.fit_transform(y.reshape(-1, 1)).flatten()

    return {
        'X': X_scaled,
```

```
        'y': y_scaled,
        'X_original': X,
        'y_original': y,
        'scaler_X': scaler_X,
        'scaler_y': scaler_y,
        'lookback_window': lookback_window,
        'volatility_window': volatility_window,
        'target_type': target_type,
        'n_samples': len(X_scaled)
    }
```

In [ ]:

```
# LSTM Model Architecture
# Build and configure neural network for time series forecasting

def build_lstm_model(lookback_window, lstm_units=[64, 32], dropout_rate=0.2,
                     learning_rate=0.001):

    model = Sequential(name='LSTM_Volatility_Forecasting')

    # First LSTM layer (returns sequences for next layer)
    model.add(LSTM(
        units=lstm_units[0],
        return_sequences=True,
        input_shape=(lookback_window, 1),
        name='LSTM_Layer_1'
    ))
    model.add(Dropout(dropout_rate, name='Dropout_1'))

    # Additional LSTM layers if specified
    for i, units in enumerate(lstm_units[1:], start=2):
        return_seq = i < len(lstm_units)  # Last LSTM layer doesn't return sequences
        model.add(LSTM(
            units=units,
            return_sequences=return_seq,
            name=f'LSTM_Layer_{i}'
        ))
        model.add(Dropout(dropout_rate, name=f'Dropout_{i}'))

    # Output layer
    model.add(Dense(1, name='Output_Layer'))

    # Compile model
    optimizer = Adam(learning_rate=learning_rate)
    model.compile(
```

```
        optimizer=optimizer,
        loss='mse',
        metrics=['mae']
    )

    return model


def get_callbacks(patience=15, min_delta=1e-6):

    # Early stopping to prevent overfitting
    early_stop = EarlyStopping(
        monitor='val_loss',
        patience=patience,
        restore_best_weights=True,
        verbose=0,
        min_delta=min_delta
    )

    # Reduce learning rate when plateau is reached
    reduce_lr = ReduceLROnPlateau(
        monitor='val_loss',
        factor=0.5,
        patience=5,
        min_lr=1e-7,
        verbose=0
    )

    return [early_stop, reduce_lr]


def train_lstm_model(model, X_train, y_train, X_val, y_val,
                     epochs=100, batch_size=32, verbose=0):

    callbacks = get_callbacks()

    history = model.fit(
        X_train, y_train,
        validation_data=(X_val, y_val),
        epochs=epochs,
        batch_size=batch_size,
        callbacks=callbacks,
        verbose=verbose,
        shuffle=False  # Important for time series data
    )
```

```
    return history
```

In [ ]:

```
# Evaluation and Prediction Functions
# Assess LSTM model performance on test data

def evaluate_lstm_predictions(model, X_test, y_test, scaler_y):
    # Generate predictions
    y_pred_scaled = model.predict(X_test, verbose=0).flatten()

    # Inverse transform to original scale
    y_pred = scaler_y.inverse_transform(y_pred_scaled.reshape(-1, 1)).flatten()
    y_true = scaler_y.inverse_transform(y_test.reshape(-1, 1)).flatten()

    # Calculate performance metrics
    mse = mean_squared_error(y_true, y_pred)
    rmse = np.sqrt(mse)
    mae = mean_absolute_error(y_true, y_pred)

    # Mean Absolute Percentage Error
    mape = np.mean(np.abs((y_true - y_pred) / (y_true + 1e-10))) * 100

    # R-squared
    ss_res = np.sum((y_true - y_pred) ** 2)
    ss_tot = np.sum((y_true - np.mean(y_true)) ** 2)
    r2 = 1 - (ss_res / ss_tot) if ss_tot > 0 else 0

    # Direction accuracy (for volatility changes)
    if len(y_true) > 1:
        direction_true = np.sign(np.diff(y_true))
        direction_pred = np.sign(np.diff(y_pred))
        direction_accuracy = np.mean(direction_true == direction_pred) * 100
    else:
        direction_accuracy = 0

    return {
        'predictions': y_pred,
        'true_values': y_true,
        'predictions_scaled': y_pred_scaled,
        'true_values_scaled': y_test,
        'metrics': {
            'mse': mse,
            'rmse': rmse,
            'mae': mae,
```

```
            'mape': mape,
            'r2': r2,
            'direction_accuracy': direction_accuracy
        }
    }
```

In [ ]:

```
# Cross-Validation Integration for LSTM
# Run LSTM training and evaluation across all CV windows

def run_lstm_cross_validation(cv_splits, data_clean, asset_name,
                              lookback_window=20, volatility_window=5,
                              lstm_units=[64, 32], dropout_rate=0.2,
                              learning_rate=0.001, epochs=100, batch_size=32,
                              verbose=True):

    print(f"Running LSTM across {len(cv_splits)} windows...")
    print(f"Architecture: {len(lstm_units)}-layer LSTM with {lstm_units} units")
    print(f"Lookback window: {lookback_window} days")
    print(f"Volatility window: {volatility_window} days")

    all_results = []
    model_summary = []

    for window_idx, split in enumerate(cv_splits):
        window_id = split['window_id']

        if verbose:
            print(f"\n Processing Window {window_id}/{len(cv_splits)}...")
            print(f"   Train: {split['train']['start'].strftime('%Y-%m-%d')} to "
                  f"{split['train']['end'].strftime('%Y-%m-%d')} ({split['train']['size']} obs)")
            print(f"   Test:  {split['test']['start'].strftime('%Y-%m-%d')} to "
                  f"{split['test']['end'].strftime('%Y-%m-%d')} ({split['test']['size']} obs)")

        try:
            # STEP 1: Prepare training data
            if verbose:
                print(f"   Preparing data with lookback={lookback_window}...")

            train_prepared = prepare_lstm_data(
                split['train']['data'],
                lookback_window=lookback_window,
                volatility_window=volatility_window,
                target_type='volatility'
            )
```

```
        # STEP 2: Prepare validation data (use fold 3 - longest validation period)
        val_fold = split['validation'][2]  # Use 24-month validation for S&P, 12-month for Bitcoin
        val_prepared = prepare_lstm_data(
            val_fold['data'],
            lookback_window=lookback_window,
            volatility_window=volatility_window,
            target_type='volatility'
        )

        # Use training scaler for validation data
        X_val_scaled = train_prepared['scaler_X'].transform(
            val_prepared['X_original'].reshape(-1, 1)
        ).reshape(val_prepared['X_original'].shape)
        y_val_scaled = train_prepared['scaler_y'].transform(
            val_prepared['y_original'].reshape(-1, 1)
        ).flatten()

        # STEP 3: Prepare test data
        test_prepared = prepare_lstm_data(
            split['test']['data'],
            lookback_window=lookback_window,
            volatility_window=volatility_window,
            target_type='volatility'
        )

        # Use training scaler for test data
        X_test_scaled = train_prepared['scaler_X'].transform(
            test_prepared['X_original'].reshape(-1, 1)
        ).reshape(test_prepared['X_original'].shape)
        y_test_scaled = train_prepared['scaler_y'].transform(
            test_prepared['y_original'].reshape(-1, 1)
        ).flatten()

        if verbose:
            print(f"  Building LSTM model...")

        # STEP 4: Build and train LSTM model
        model = build_lstm_model(
            lookback_window=lookback_window,
            lstm_units=lstm_units,
            dropout_rate=dropout_rate,
            learning_rate=learning_rate
        )
```

```
            if verbose:
                print(f"  Training model (max {epochs} epochs)...")

            history = train_lstm_model(
                model,
                train_prepared['X'], train_prepared['y'],
                X_val_scaled, y_val_scaled,
                epochs=epochs,
                batch_size=batch_size,
                verbose=0
            )

            # STEP 5: Evaluate on test data
            if verbose:
                print(f"  Evaluating on test data...")

            evaluation = evaluate_lstm_predictions(
                model,
                X_test_scaled,
                y_test_scaled,
                train_prepared['scaler_y']
            )

            # Store results
            window_result = {
                'window_id': window_id,
                'asset': asset_name,
                'train_period': f"{split['train']['start'].strftime('%Y-%m-%d')} to "
                                f"{split['train']['end'].strftime('%Y-%m-%d')}",
                'test_period': f"{split['test']['start'].strftime('%Y-%m-%d')} to "
                               f"{split['test']['end'].strftime('%Y-%m-%d')}",
                'train_size': train_prepared['n_samples'],
                'val_size': len(X_val_scaled),
                'test_size': len(X_test_scaled),
                'hyperparameters': {
                    'lookback_window': lookback_window,
                    'volatility_window': volatility_window,
                    'lstm_units': lstm_units,
                    'dropout_rate': dropout_rate,
                    'learning_rate': learning_rate
                },
                'training': {
                    'epochs_trained': len(history.history['loss']),
                    'final_train_loss': history.history['loss'][-1],
                    'final_val_loss': history.history['val_loss'][-1],
```

```
                'history': history.history
            },
            'evaluation': evaluation,
            'model': model
        }

        all_results.append(window_result)

        # Summary for quick reference
        model_summary.append({
            'Window': window_id,
            'Train_Samples': train_prepared['n_samples'],
            'Test_Samples': len(X_test_scaled),
            'Epochs': len(history.history['loss']),
            'Train_Loss': history.history['loss'][-1],
            'Val_Loss': history.history['val_loss'][-1],
            'Test_RMSE': evaluation['metrics']['rmse'],
            'Test_MAE': evaluation['metrics']['mae'],
            'Test_R2': evaluation['metrics']['r2'],
            'Direction_Accuracy': evaluation['metrics']['direction_accuracy']
        })

        if verbose:
            print(f"    Test RMSE={evaluation['metrics']['rmse']:.6f}, "
                  f"R²={evaluation['metrics']['r2']:.4f}, "
                  f"Direction Acc={evaluation['metrics']['direction_accuracy']:.1f}%")

    except Exception as e:
        print(f"    Window {window_id} failed: {str(e)}")
        continue

# Create summary DataFrame
summary_df = pd.DataFrame(model_summary)

# Calculate overall performance statistics
if len(summary_df) > 0:
    performance_summary = {
        'total_windows': len(cv_splits),
        'successful_windows': len(summary_df),
        'success_rate': len(summary_df) / len(cv_splits) * 100,
        'avg_test_rmse': summary_df['Test_RMSE'].mean(),
        'std_test_rmse': summary_df['Test_RMSE'].std(),
        'avg_test_mae': summary_df['Test_MAE'].mean(),
        'avg_r2': summary_df['Test_R2'].mean(),
        'avg_direction_accuracy': summary_df['Direction_Accuracy'].mean(),
```

```
            'avg_epochs': summary_df['Epochs'].mean()
        }
    else:
        performance_summary = None

    print(f"{asset_name.upper()} LSTM CROSS-VALIDATION COMPLETE")


    if performance_summary:
        print(f" Successfully processed {performance_summary['successful_windows']}/"
              f"{performance_summary['total_windows']} windows")
        print(f" Average Test RMSE: {performance_summary['avg_test_rmse']:.6f} ± "
              f"{performance_summary['std_test_rmse']:.6f}")
        print(f" Average R²: {performance_summary['avg_r2']:.4f}")
        print(f" Average Direction Accuracy: {performance_summary['avg_direction_accuracy']:.2f}%")
        print(f"  Average Epochs: {performance_summary['avg_epochs']:.1f}")
    else:
        print(" No successful model fits achieved")

    return {
        'asset_name': asset_name,
        'all_results': all_results,
        'summary_df': summary_df,
        'performance_summary': performance_summary,
        'methodology': {
            'model_type': 'LSTM Neural Network',
            'architecture': f'{len(lstm_units)}-layer LSTM',
            'lookback_window': lookback_window,
            'volatility_window': volatility_window,
            'target': 'Realized Volatility',
            'validation': '3-fold temporal validation'
        }
    }
```

LSTM: Test 3 windows

In [ ]:
```
# Execute LSTM Analysis on S&P 500
# Run complete analysis on first few windows to demonstrate methodology

print("LSTM VOLATILITY FORECASTING ANALYSIS")

# Configuration for LSTM
```

```
LSTM_CONFIG = {
    'lookback_window': 20,      # Use 20 days of past returns
    'volatility_window': 5,     # Calculate 5-day realized volatility
    'lstm_units': [64, 32],     # 2-layer LSTM with 64 and 32 units
    'dropout_rate': 0.2,        # 20% dropout for regularization
    'learning_rate': 0.001,     # Adam optimizer learning rate
    'epochs': 100,              # Maximum epochs (early stopping will apply)
    'batch_size': 32            # Training batch size
}

print("\n LSTM Configuration:")
for key, value in LSTM_CONFIG.items():
    print(f"   {key}: {value}")

# Run S&P 500 analysis (first 3 windows for demonstration)
print("\n S&P 500 ANALYSIS (First 3 Windows)")
sp500_lstm_results = run_lstm_cross_validation(
    cv_splits=sp500_cv_splits[:3],  # Limit to first 3 windows for demonstration
    data_clean=sp500_clean,
    asset_name='S&P 500',
    **LSTM_CONFIG,
    verbose=True
)
```

In [ ]:
```
print("\nBITCOIN ANALYSIS (First 3 Windows)")
bitcoin_lstm_results = run_lstm_cross_validation(
    cv_splits=bitcoin_cv_splits[:3],  # Limit to first 3 windows for demonstration
    data_clean=bitcoin_clean,
    asset_name='Bitcoin',
    **LSTM_CONFIG,
    verbose=True
)
```

LSTM: End test 3 windows

In [ ]:
```
# Visualization Functions for LSTM Results
# Create comprehensive visualizations of model performance

def plot_lstm_results(lstm_results, max_windows=3):
    asset_name = lstm_results['asset_name']
    all_results = lstm_results['all_results'][:max_windows]
    summary_df = lstm_results['summary_df']
```

```python
# Create figure with subplots
n_windows = len(all_results)
fig = plt.figure(figsize=(18, 4 * n_windows + 4))

# Create grid for plots
gs = fig.add_gridspec(n_windows + 1, 3, hspace=0.3, wspace=0.3)

# Plot each window's results
for idx, result in enumerate(all_results):
    window_id = result['window_id']
    evaluation = result['evaluation']
    history = result['training']['history']

    # Plot 1: Training History
    ax1 = fig.add_subplot(gs[idx, 0])
    ax1.plot(history['loss'], label='Training Loss', color='blue', alpha=0.7)
    ax1.plot(history['val_loss'], label='Validation Loss', color='orange', alpha=0.7)
    ax1.set_title(f'Window {window_id}: Training History')
    ax1.set_xlabel('Epoch')
    ax1.set_ylabel('Loss (MSE)')
    ax1.legend()
    ax1.grid(True, alpha=0.3)

    # Plot 2: Predictions vs Actual
    ax2 = fig.add_subplot(gs[idx, 1])
    true_vals = evaluation['true_values']
    pred_vals = evaluation['predictions']
    x_axis = range(len(true_vals))

    ax2.plot(x_axis, true_vals, label='True Volatility', color='black', alpha=0.7, linewidth=2)
    ax2.plot(x_axis, pred_vals, label='LSTM Prediction', color='red', alpha=0.7, linestyle='--')
    ax2.set_title(f'Window {window_id}: Predictions vs Actual')
    ax2.set_xlabel('Time Step')
    ax2.set_ylabel('Realized Volatility')
    ax2.legend()
    ax2.grid(True, alpha=0.3)

    # Add performance metrics as text
    metrics = evaluation['metrics']
    text_str = f"RMSE: {metrics['rmse']:.6f}\nMAE: {metrics['mae']:.6f}\nR²: {metrics['r2']:.4f}"
    ax2.text(0.02, 0.98, text_str, transform=ax2.transAxes,
            verticalalignment='top', bbox=dict(boxstyle='round', facecolor='wheat', alpha=0.5))

    # Plot 3: Scatter Plot (Predicted vs Actual)
```

```
        ax3 = fig.add_subplot(gs[idx, 2])
        ax3.scatter(true_vals, pred_vals, alpha=0.5, color='purple')

        # Add perfect prediction line
        min_val = min(true_vals.min(), pred_vals.min())
        max_val = max(true_vals.max(), pred_vals.max())
        ax3.plot([min_val, max_val], [min_val, max_val], 'r--', label='Perfect Prediction')

        ax3.set_title(f'Window {window_id}: Prediction Accuracy')
        ax3.set_xlabel('True Volatility')
        ax3.set_ylabel('Predicted Volatility')
        ax3.legend()
        ax3.grid(True, alpha=0.3)

    # Summary plots at the bottom
    if len(summary_df) > 0:
        # Plot 4: RMSE across windows
        ax4 = fig.add_subplot(gs[n_windows, 0])
        ax4.bar(summary_df['Window'], summary_df['Test_RMSE'], color='steelblue', alpha=0.7)
        ax4.set_title('Test RMSE Across Windows')
        ax4.set_xlabel('Window')
        ax4.set_ylabel('RMSE')
        ax4.grid(True, alpha=0.3, axis='y')

        # Plot 5: R² across windows
        ax5 = fig.add_subplot(gs[n_windows, 1])
        ax5.bar(summary_df['Window'], summary_df['Test_R2'], color='seagreen', alpha=0.7)
        ax5.axhline(y=0, color='red', linestyle='--', alpha=0.5)
        ax5.set_title('R² Score Across Windows')
        ax5.set_xlabel('Window')
        ax5.set_ylabel('R²')
        ax5.grid(True, alpha=0.3, axis='y')

        # Plot 6: Training epochs
        ax6 = fig.add_subplot(gs[n_windows, 2])
        ax6.bar(summary_df['Window'], summary_df['Epochs'], color='coral', alpha=0.7)
        ax6.set_title('Training Epochs (Early Stopping)')
        ax6.set_xlabel('Window')
        ax6.set_ylabel('Epochs')
        ax6.grid(True, alpha=0.3, axis='y')

    plt.suptitle(f'{asset_name} LSTM Volatility Forecasting Results',
                 fontsize=16, fontweight='bold', y=0.995)

    return fig
```

```python
def create_lstm_summary_table(sp500_results, bitcoin_results):

    # Overall comparison
    summary_data = {
        'Asset': ['S&P 500', 'Bitcoin'],
        'Windows_Tested': [
            sp500_results['performance_summary']['successful_windows'],
            bitcoin_results['performance_summary']['successful_windows']
        ],
        'Avg_RMSE': [
            sp500_results['performance_summary']['avg_test_rmse'],
            bitcoin_results['performance_summary']['avg_test_rmse']
        ],
        'Std_RMSE': [
            sp500_results['performance_summary']['std_test_rmse'],
            bitcoin_results['performance_summary']['std_test_rmse']
        ],
        'Avg_MAE': [
            sp500_results['performance_summary']['avg_test_mae'],
            bitcoin_results['performance_summary']['avg_test_mae']
        ],
        'Avg_R2': [
            sp500_results['performance_summary']['avg_r2'],
            bitcoin_results['performance_summary']['avg_r2']
        ],
        'Avg_Direction_Acc_%': [
            sp500_results['performance_summary']['avg_direction_accuracy'],
            bitcoin_results['performance_summary']['avg_direction_accuracy']
        ],
        'Avg_Epochs': [
            sp500_results['performance_summary']['avg_epochs'],
            bitcoin_results['performance_summary']['avg_epochs']
        ]
    }

    summary_df = pd.DataFrame(summary_data)

    print("\n📊 OVERALL PERFORMANCE COMPARISON")
    print("-" * 100)
    print(summary_df.to_string(index=False, float_format='%.6f'))

    # Detailed window results
    print(f"\n📈 S&P 500 DETAILED RESULTS")
```

```
    print("-" * 100)
    print(sp500_results['summary_df'].round(6).to_string(index=False))

    print(f"\n🪙 BITCOIN DETAILED RESULTS")
    print("-" * 100)
    print(bitcoin_results['summary_df'].round(6).to_string(index=False))

    # Methodology summary
    print(f"\n📋 METHODOLOGY SUMMARY")
    print("-" * 100)
    print(f"Model Type: {sp500_results['methodology']['model_type']}")
    print(f"Architecture: {sp500_results['methodology']['architecture']}")
    print(f"Lookback Window: {sp500_results['methodology']['lookback_window']} days")
    print(f"Volatility Window: {sp500_results['methodology']['volatility_window']} days")
    print(f"Target Variable: {sp500_results['methodology']['target']}")
    print(f"Validation Strategy: {sp500_results['methodology']['validation']}")

    return summary_df
```

In [ ]:

```
# Optional: Run Full Analysis on All Windows
# Uncomment and run this cell to process all CV windows (will take longer)


print("FULL S&P 500 Analysis")
sp500_lstm_full = run_lstm_cross_validation(
    cv_splits=sp500_cv_splits,  # All windows
    data_clean=sp500_clean,
    asset_name='S&P 500',
    **LSTM_CONFIG,
    verbose=True
)

# Full Bitcoin Analysis (all 11 windows)
print("\n Running FULL Bitcoin Analysis")
bitcoin_lstm_full = run_lstm_cross_validation(
    cv_splits=bitcoin_cv_splits,  # All windows
    data_clean=bitcoin_clean,
    asset_name='Bitcoin',
    **LSTM_CONFIG,
    verbose=True
)

# Generate complete summary
full_summary = create_lstm_summary_table(sp500_lstm_full, bitcoin_lstm_full)
```

In [ ]:
```
summary_table = create_lstm_summary_table(sp500_lstm_full, bitcoin_lstm_full)


fig1 = plot_lstm_results(sp500_lstm_full, max_windows=16)
plt.show()

# Visualize Bitcoin results
fig2 = plot_lstm_results(bitcoin_lstm_full, max_windows=11)
plt.show()
```

SVM

In [ ]:
```
# SVM Model Implementation for Volatility Forecasting
# Support Vector Machines with RBF kernel for non-linear pattern recognition

from sklearn.svm import SVR
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
from sklearn.model_selection import GridSearchCV
import warnings
warnings.filterwarnings('ignore')

# Set random seed for reproducibility
np.random.seed(42)
```

In [ ]:
```
# Data Preparation Functions for SVM
# Transform time series data into feature vectors

def create_svm_features(data, lookback_window=20):
    X, y = [], []

    for i in range(lookback_window, len(data)):
        X.append(data[i-lookback_window:i])
        y.append(data[i])

    X = np.array(X)
    y = np.array(y)

    return X, y
```

```python
def calculate_technical_features(returns, window=5):
    features = pd.DataFrame(index=returns.index)

    # Realized volatility (5-day)
    features['realized_vol'] = np.sqrt((returns ** 2).rolling(window=window).sum())

    # Moving average of returns
    features['ma_returns'] = returns.rolling(window=window).mean()

    # Rolling standard deviation
    features['rolling_std'] = returns.rolling(window=window).std()

    # Exponential moving average
    features['ema_returns'] = returns.ewm(span=window).mean()

    # Squared returns (proxy for volatility)
    features['squared_returns'] = returns ** 2

    # Return momentum
    features['momentum'] = returns.rolling(window=window).sum()

    return features


def prepare_svm_data(data, lookback_window=20, volatility_window=5,
                     use_technical_features=False, target_type='volatility'):

    # Extract log returns
    returns = data['Log_Returns'].values

    if target_type == 'volatility':
        # Calculate realized volatility as target
        realized_vol = calculate_realized_volatility(data['Log_Returns'],
                                                     window=volatility_window)
        target_data = realized_vol.values

        # Remove NaN values
        valid_idx = ~np.isnan(target_data)
        target_data = target_data[valid_idx]
        returns = returns[valid_idx]

    else:  # returns forecasting
        target_data = returns

    # Create basic features from returns
```

```
    X, y = create_svm_features(returns, lookback_window)

    # Adjust target for volatility forecasting
    if target_type == 'volatility':
        vol_start_idx = lookback_window + volatility_window - 1
        y_vol = target_data[vol_start_idx:vol_start_idx+len(y)]

        min_len = min(len(X), len(y_vol))
        X = X[:min_len]
        y = y_vol[:min_len]

    if use_technical_features and target_type == 'volatility':
        # This would require more complex feature engineering
        # For now, we keep it simple with lagged returns
        pass

    # Normalize features
    scaler_X = StandardScaler()
    X_scaled = scaler_X.fit_transform(X)

    # Normalize targets
    scaler_y = StandardScaler()
    y_scaled = scaler_y.fit_transform(y.reshape(-1, 1)).flatten()

    return {
        'X': X_scaled,
        'y': y_scaled,
        'X_original': X,
        'y_original': y,
        'scaler_X': scaler_X,
        'scaler_y': scaler_y,
        'lookback_window': lookback_window,
        'volatility_window': volatility_window,
        'target_type': target_type,
        'n_samples': len(X_scaled),
        'n_features': X_scaled.shape[1]
    }
```

In [ ]:

```
# SVM Model Architecture and Training
# Build and train Support Vector Regression models

def build_svm_model(kernel='rbf', C=1.0, epsilon=0.1, gamma='scale'):
    model = SVR(
        kernel=kernel,
```

```
        C=C,
        epsilon=epsilon,
        gamma=gamma,
        cache_size=500,  # Larger cache for faster training
        verbose=False
    )

    return model


def perform_hyperparameter_search(X_train, y_train, search_type='fast'):

    if search_type == 'fast':
        # Fast grid search with fewer options
        param_grid = {
            'C': [0.1, 1.0, 10.0],
            'epsilon': [0.01, 0.1],
            'gamma': ['scale', 'auto']
        }
        cv_folds = 3
    else:
        # Thorough search with more options
        param_grid = {
            'C': [0.1, 1.0, 10.0, 100.0],
            'epsilon': [0.01, 0.05, 0.1, 0.2],
            'gamma': ['scale', 'auto', 0.001, 0.01]
        }
        cv_folds = 5

    # Create base model
    base_model = SVR(kernel='rbf', cache_size=500)

    # Grid search with cross-validation
    grid_search = GridSearchCV(
        base_model,
        param_grid,
        cv=cv_folds,
        scoring='neg_mean_squared_error',
        n_jobs=-1,  # Use all CPU cores
        verbose=0
    )

    # Fit grid search
    grid_search.fit(X_train, y_train)
```

```python
    return {
        'best_params': grid_search.best_params_,
        'best_score': -grid_search.best_score_,  # Convert back to MSE
        'best_model': grid_search.best_estimator_,
        'cv_results': grid_search.cv_results_
    }


def train_svm_model(X_train, y_train, X_val, y_val,
                    use_hyperparameter_search=False,
                    C=1.0, epsilon=0.1, gamma='scale',
                    verbose=False):

    if use_hyperparameter_search:
        if verbose:
            print("hyperparameter search...")

        search_results = perform_hyperparameter_search(X_train, y_train, search_type='fast')
        model = search_results['best_model']
        best_params = search_results['best_params']

        if verbose:
            print(f"   Best parameters: C={best_params['C']}, "
                  f"epsilon={best_params['epsilon']}, gamma={best_params['gamma']}")
    else:
        # Use provided hyperparameters
        model = build_svm_model(kernel='rbf', C=C, epsilon=epsilon, gamma=gamma)
        model.fit(X_train, y_train)
        best_params = {'C': C, 'epsilon': epsilon, 'gamma': gamma}

    # Calculate training metrics
    train_pred = model.predict(X_train)
    train_mse = mean_squared_error(y_train, train_pred)

    # Calculate validation metrics
    val_pred = model.predict(X_val)
    val_mse = mean_squared_error(y_val, val_pred)

    return {
        'model': model,
        'best_params': best_params,
        'train_mse': train_mse,
        'val_mse': val_mse,
```

```
        'n_support_vectors': len(model.support_) if hasattr(model, 'support_') else 0
    }
```

In [ ]:

```
# Multi-Kernel Comparison Framework
# Compare Linear, Polynomial, and RBF kernels (Cocco et al., 2021)

def train_svm_multi_kernel(X_train, y_train, X_val, y_val, verbose=False):

    kernel_results = {}

    # Define kernel configurations
    kernel_configs = {
        'Linear': {'kernel': 'linear', 'C': [0.1, 1.0, 10.0]},
        'Polynomial (degree=2)': {'kernel': 'poly', 'degree': 2, 'C': [0.1, 1.0, 10.0]},
        'Polynomial (degree=3)': {'kernel': 'poly', 'degree': 3, 'C': [0.1, 1.0, 10.0]},
        'RBF': {'kernel': 'rbf', 'C': [0.1, 1.0, 10.0], 'gamma': ['scale', 'auto']}
    }

    if verbose:
        print("   Testing multiple kernel functions...")

    for kernel_name, config in kernel_configs.items():
        try:
            if verbose:
                print(f"      - Training {kernel_name} kernel...")

            # Prepare parameter grid
            if config['kernel'] == 'linear':
                param_grid = {
                    'kernel': ['linear'],
                    'C': config['C'],
                    'epsilon': [0.01, 0.1]
                }
            elif config['kernel'] == 'poly':
                param_grid = {
                    'kernel': ['poly'],
                    'degree': [config['degree']],
                    'C': config['C'],
                    'epsilon': [0.01, 0.1],
                    'gamma': ['scale']
                }
            else:  # RBF
                param_grid = {
                    'kernel': ['rbf'],
```

```
                'C': config['C'],
                'epsilon': [0.01, 0.1],
                'gamma': config['gamma']
            }

        # Grid search
        base_model = SVR(cache_size=500)
        grid_search = GridSearchCV(
            base_model,
            param_grid,
            cv=3,
            scoring='neg_mean_squared_error',
            n_jobs=-1,
            verbose=0
        )
        grid_search.fit(X_train, y_train)

        # Get best model
        best_model = grid_search.best_estimator_

        # Evaluate on validation set
        val_pred = best_model.predict(X_val)
        val_mse = mean_squared_error(y_val, val_pred)
        val_rmse = np.sqrt(val_mse)

        # Store results
        kernel_results[kernel_name] = {
            'model': best_model,
            'best_params': grid_search.best_params_,
            'val_mse': val_mse,
            'val_rmse': val_rmse,
            'best_cv_score': -grid_search.best_score_,
            'n_support_vectors': len(best_model.support_) if hasattr(best_model, 'support_') else 0
        }

        if verbose:
            print(f"        Validation RMSE: {val_rmse:.6f}, Support Vectors: {kernel_results[kernel_name]['n_support_vecto

    except Exception as e:
        if verbose:
            print(f"        Failed: {str(e)}")
        kernel_results[kernel_name] = None

# Find best kernel
valid_results = {k: v for k, v in kernel_results.items() if v is not None}
```

```
    if valid_results:
        best_kernel = min(valid_results.keys(), key=lambda k: valid_results[k]['val_rmse'])
        if verbose:
            print(f"  Best kernel: {best_kernel} (RMSE: {valid_results[best_kernel]['val_rmse']:.6f})")
    else:
        best_kernel = None

    return {
        'kernel_results': kernel_results,
        'best_kernel': best_kernel,
        'best_model': kernel_results[best_kernel]['model'] if best_kernel else None
    }


def compare_kernel_performance(kernel_comparison_results):

    comparison_data = []

    for kernel_name, results in kernel_comparison_results['kernel_results'].items():
        if results is not None:
            comparison_data.append({
                'Kernel': kernel_name,
                'Validation_RMSE': results['val_rmse'],
                'Validation_MSE': results['val_mse'],
                'CV_Score': results['best_cv_score'],
                'Support_Vectors': results['n_support_vectors'],
                'C': results['best_params']['C'],
                'Epsilon': results['best_params']['epsilon']
            })

    return pd.DataFrame(comparison_data).sort_values('Validation_RMSE')


# Evaluation and Prediction Functions for SVM
# Assess model performance on test data

def evaluate_svm_predictions(model, X_test, y_test, scaler_y):

    # Generate predictions
    y_pred_scaled = model.predict(X_test)

    # Inverse transform to original scale
    y_pred = scaler_y.inverse_transform(y_pred_scaled.reshape(-1, 1)).flatten()
    y_true = scaler_y.inverse_transform(y_test.reshape(-1, 1)).flatten()
```

```
    # Calculate performance metrics
    mse = mean_squared_error(y_true, y_pred)
    rmse = np.sqrt(mse)
    mae = mean_absolute_error(y_true, y_pred)
    r2 = r2_score(y_true, y_pred)

    # Mean Absolute Percentage Error
    mape = np.mean(np.abs((y_true - y_pred) / (y_true + 1e-10))) * 100

    # Direction accuracy (for volatility changes)
    if len(y_true) > 1:
        direction_true = np.sign(np.diff(y_true))
        direction_pred = np.sign(np.diff(y_pred))
        direction_accuracy = np.mean(direction_true == direction_pred) * 100
    else:
        direction_accuracy = 0

    # Calculate residuals
    residuals = y_true - y_pred

    return {
        'predictions': y_pred,
        'true_values': y_true,
        'predictions_scaled': y_pred_scaled,
        'true_values_scaled': y_test,
        'residuals': residuals,
        'metrics': {
            'mse': mse,
            'rmse': rmse,
            'mae': mae,
            'mape': mape,
            'r2': r2,
            'direction_accuracy': direction_accuracy
        }
    }
```

In [ ]:

```
# Cross-Validation Integration for SVM
# Run SVM training and evaluation across all CV windows

def run_svm_cross_validation(cv_splits, data_clean, asset_name,
                             lookback_window=20, volatility_window=5,
                             use_hyperparameter_search=True,
                             C=1.0, epsilon=0.1, gamma='scale',
                             verbose=True):
```

```python
all_results = []
model_summary = []

for window_idx, split in enumerate(cv_splits):
    window_id = split['window_id']

    if verbose:
        print(f"\n🔄 Processing Window {window_id}/{len(cv_splits)}...")
        print(f"   Train: {split['train']['start'].strftime('%Y-%m-%d')} to "
              f"{split['train']['end'].strftime('%Y-%m-%d')} ({split['train']['size']} obs)")
        print(f"   Test:  {split['test']['start'].strftime('%Y-%m-%d')} to "
              f"{split['test']['end'].strftime('%Y-%m-%d')} ({split['test']['size']} obs)")

    try:
        # STEP 1: Prepare training data
        if verbose:
            print(f"Lookback={lookback_window}...")

        train_prepared = prepare_svm_data(
            split['train']['data'],
            lookback_window=lookback_window,
            volatility_window=volatility_window,
            target_type='volatility'
        )

        # STEP 2: Prepare validation data (use fold 3 - longest validation period)
        val_fold = split['validation'][2]
        val_prepared = prepare_svm_data(
            val_fold['data'],
            lookback_window=lookback_window,
            volatility_window=volatility_window,
            target_type='volatility'
        )

        # Use training scaler for validation data
        X_val_scaled = train_prepared['scaler_X'].transform(val_prepared['X_original'])
        y_val_scaled = train_prepared['scaler_y'].transform(
            val_prepared['y_original'].reshape(-1, 1)
        ).flatten()

        # STEP 3: Prepare test data
        test_prepared = prepare_svm_data(
            split['test']['data'],
```

```
            lookback_window=lookback_window,
            volatility_window=volatility_window,
            target_type='volatility'
        )

        # Use training scaler for test data
        X_test_scaled = train_prepared['scaler_X'].transform(test_prepared['X_original'])
        y_test_scaled = train_prepared['scaler_y'].transform(
            test_prepared['y_original'].reshape(-1, 1)
        ).flatten()

        if verbose:
            print(f"Training ")

        # STEP 4: Train SVM model
        training_result = train_svm_model(
            train_prepared['X'], train_prepared['y'],
            X_val_scaled, y_val_scaled,
            use_hyperparameter_search=use_hyperparameter_search,
            C=C, epsilon=epsilon, gamma=gamma,
            verbose=verbose
        )

        model = training_result['model']

        # STEP 5: Evaluate on test data
        if verbose:
            print(f"Evaluating")

        evaluation = evaluate_svm_predictions(
            model,
            X_test_scaled,
            y_test_scaled,
            train_prepared['scaler_y']
        )

        # Store results
        window_result = {
            'window_id': window_id,
            'asset': asset_name,
            'train_period': f"{split['train']['start'].strftime('%Y-%m-%d')} to "
                            f"{split['train']['end'].strftime('%Y-%m-%d')}",
            'test_period': f"{split['test']['start'].strftime('%Y-%m-%d')} to "
                           f"{split['test']['end'].strftime('%Y-%m-%d')}",
            'train_size': train_prepared['n_samples'],
```

```
                'val_size': len(X_val_scaled),
                'test_size': len(X_test_scaled),
                'hyperparameters': training_result['best_params'],
                'training': {
                    'train_mse': training_result['train_mse'],
                    'val_mse': training_result['val_mse'],
                    'n_support_vectors': training_result['n_support_vectors']
                },
                'evaluation': evaluation,
                'model': model
            }

            all_results.append(window_result)

            # Summary for quick reference
            model_summary.append({
                'Window': window_id,
                'Train_Samples': train_prepared['n_samples'],
                'Test_Samples': len(X_test_scaled),
                'C': training_result['best_params']['C'],
                'Epsilon': training_result['best_params']['epsilon'],
                'Gamma': training_result['best_params']['gamma'],
                'N_Support_Vectors': training_result['n_support_vectors'],
                'Train_MSE': training_result['train_mse'],
                'Val_MSE': training_result['val_mse'],
                'Test_RMSE': evaluation['metrics']['rmse'],
                'Test_MAE': evaluation['metrics']['mae'],
                'Test_R2': evaluation['metrics']['r2'],
                'Direction_Accuracy': evaluation['metrics']['direction_accuracy']
            })

            if verbose:
                print(f"Test RMSE={evaluation['metrics']['rmse']:.6f}, "
                      f"R²={evaluation['metrics']['r2']:.4f}, "
                      f"Direction Acc={evaluation['metrics']['direction_accuracy']:.1f}%")
                print(f"Support Vectors: {training_result['n_support_vectors']}")

        except Exception as e:
            print(f"Window {window_id} failed: {str(e)}")
            import traceback
            traceback.print_exc()
            continue

    # Create summary DataFrame
    summary_df = pd.DataFrame(model_summary)
```

```python
    # Calculate overall performance statistics
    if len(summary_df) > 0:
        performance_summary = {
            'total_windows': len(cv_splits),
            'successful_windows': len(summary_df),
            'success_rate': len(summary_df) / len(cv_splits) * 100,
            'avg_test_rmse': summary_df['Test_RMSE'].mean(),
            'std_test_rmse': summary_df['Test_RMSE'].std(),
            'avg_test_mae': summary_df['Test_MAE'].mean(),
            'avg_r2': summary_df['Test_R2'].mean(),
            'avg_direction_accuracy': summary_df['Direction_Accuracy'].mean(),
            'avg_support_vectors': summary_df['N_Support_Vectors'].mean()
        }
    else:
        performance_summary = None

    print(f"\n{'='*80}")
    print(f"{asset_name.upper()} SVM CROSS-VALIDATION COMPLETE")
    print(f"{'='*80}")

    if performance_summary:
        print(f"Successfully processed {performance_summary['successful_windows']}/"
              f"{performance_summary['total_windows']} windows")
        print(f"Average Test RMSE: {performance_summary['avg_test_rmse']:.6f} ± "
              f"{performance_summary['std_test_rmse']:.6f}")
        print(f"Average R²: {performance_summary['avg_r2']:.4f}")
        print(f"Average Direction Accuracy: {performance_summary['avg_direction_accuracy']:.2f}%")
        print(f"Average Support Vectors: {performance_summary['avg_support_vectors']:.0f}")
    else:
        print("No successful model fits achieved")

    return {
        'asset_name': asset_name,
        'all_results': all_results,
        'summary_df': summary_df,
        'performance_summary': performance_summary,
        'methodology': {
            'model_type': 'Support Vector Regression',
            'kernel': 'RBF (Radial Basis Function)',
            'lookback_window': lookback_window,
            'volatility_window': volatility_window,
            'target': 'Realized Volatility',
            'hyperparameter_search': use_hyperparameter_search,
            'validation': '3-fold temporal validation'
```

```
        }
    }
```

In [ ]:

```
# Use first window for demonstration
demo_split = sp500_cv_splits[0]

print(f"Demo Window: {demo_split['train']['start'].strftime('%Y-%m-%d')} to {demo_split['test']['end'].strftime('%Y-%m-%d')}")

# Prepare data
print("\n📊 Preparing data...")
demo_train = prepare_svm_data(
    demo_split['train']['data'],
    lookback_window=20,
    volatility_window=5,
    target_type='volatility'
)

demo_val = prepare_svm_data(
    demo_split['validation'][2]['data'],
    lookback_window=20,
    volatility_window=5,
    target_type='volatility'
)

# Use training scaler for validation
X_val_demo = demo_train['scaler_X'].transform(demo_val['X_original'])
y_val_demo = demo_train['scaler_y'].transform(demo_val['y_original'].reshape(-1, 1)).flatten()

# Test multiple kernels
print("\n🔬 Testing kernel functions...")
kernel_comparison = train_svm_multi_kernel(
    demo_train['X'], demo_train['y'],
    X_val_demo, y_val_demo,
    verbose=True
)

# Display comparison table
print(f"\n{'='*100}")
print("KERNEL PERFORMANCE COMPARISON")
print("="*100)
kernel_summary = compare_kernel_performance(kernel_comparison)
print(kernel_summary.to_string(index=False, float_format='%.6f'))

print(f"\n🏆 Best Performing Kernel: {kernel_comparison['best_kernel']}")
```

```
print(f"   Validation RMSE: {kernel_comparison['kernel_results'][kernel_comparison['best_kernel']]['val_rmse']:.6f}")

# Visualize kernel comparison
fig, axes = plt.subplots(1, 2, figsize=(14, 5))

# Plot 1: Validation RMSE by kernel
valid_kernels = kernel_summary['Kernel'].tolist()
valid_rmse = kernel_summary['Validation_RMSE'].tolist()
colors = ['blue' if 'Linear' in k else 'orange' if 'Polynomial' in k else 'green' for k in valid_kernels]

axes[0].bar(range(len(valid_kernels)), valid_rmse, color=colors, alpha=0.7)
axes[0].set_xticks(range(len(valid_kernels)))
axes[0].set_xticklabels(valid_kernels, rotation=45, ha='right')
axes[0].set_title('Validation RMSE by Kernel Type')
axes[0].set_ylabel('RMSE')
axes[0].grid(True, alpha=0.3, axis='y')

# Plot 2: Support Vectors by kernel
valid_sv = kernel_summary['Support_Vectors'].tolist()
axes[1].bar(range(len(valid_kernels)), valid_sv, color=colors, alpha=0.7)
axes[1].set_xticks(range(len(valid_kernels)))
axes[1].set_xticklabels(valid_kernels, rotation=45, ha='right')
axes[1].set_title('Number of Support Vectors')
axes[1].set_ylabel('Support Vectors')
axes[1].grid(True, alpha=0.3, axis='y')

plt.suptitle('SVM Kernel Comparison', fontsize=14, fontweight='bold')
plt.tight_layout()
plt.show()




# Configuration for SVM
SVM_CONFIG = {
    'lookback_window': 20,              # Use 20 days of past returns as features
    'volatility_window': 5,             # Calculate 5-day realized volatility
    'use_hyperparameter_search': True,  # Enable grid search for optimal parameters
    'C': 1.0,                           # Default regularization (if search disabled)
    'epsilon': 0.1,                     # Default epsilon (if search disabled)
    'gamma': 'scale'                    # Default gamma (if search disabled)
}

print("\n SVM Configuration:")
for key, value in SVM_CONFIG.items():
```

```
    print(f"   {key}: {value}")

# Run S&P 500 analysis (first 3 windows for demonstration)
print("\n S&P 500 ANALYSIS (First 3 Windows)")
sp500_svm_results = run_svm_cross_validation(
    cv_splits=sp500_cv_splits[:3],  # Limit to first 3 windows for demonstration
    data_clean=sp500_clean,
    asset_name='S&P 500',
    **SVM_CONFIG,
    verbose=True
)
```

In [ ]:
```
# Execute SVM Analysis on Bitcoin
# Run analysis on Bitcoin data

print("\n PHASE 2: BITCOIN ANALYSIS (First 3 Windows)")
bitcoin_svm_results = run_svm_cross_validation(
    cv_splits=bitcoin_cv_splits[:3],  # Limit to first 3 windows for demonstration
    data_clean=bitcoin_clean,
    asset_name='Bitcoin',
    **SVM_CONFIG,
    verbose=True
)
```

In [ ]:
```
# Visualization Functions for SVM Results
# Create comprehensive visualizations of model performance

def plot_svm_results(svm_results, max_windows=3):

    asset_name = svm_results['asset_name']
    all_results = svm_results['all_results'][:max_windows]
    summary_df = svm_results['summary_df']

    # Create figure with subplots
    n_windows = len(all_results)
    fig = plt.figure(figsize=(18, 4 * n_windows + 4))

    # Create grid for plots
    gs = fig.add_gridspec(n_windows + 1, 3, hspace=0.3, wspace=0.3)

    # Plot each window's results
    for idx, result in enumerate(all_results):
        window_id = result['window_id']
        evaluation = result['evaluation']
```

```
        # Plot 1: Residuals Plot
        ax1 = fig.add_subplot(gs[idx, 0])
        residuals = evaluation['residuals']
        x_axis = range(len(residuals))

        ax1.scatter(x_axis, residuals, alpha=0.5, color='purple')
        ax1.axhline(y=0, color='red', linestyle='--', linewidth=2)
        ax1.set_title(f'Window {window_id}: Residuals')
        ax1.set_xlabel('Sample')
        ax1.set_ylabel('Residual (True - Predicted)')
        ax1.grid(True, alpha=0.3)

        # Plot 2: Predictions vs Actual
        ax2 = fig.add_subplot(gs[idx, 1])
        true_vals = evaluation['true_values']
        pred_vals = evaluation['predictions']
        x_axis = range(len(true_vals))

        ax2.plot(x_axis, true_vals, label='True Volatility', color='black', alpha=0.7, linewidth=2)
        ax2.plot(x_axis, pred_vals, label='SVM Prediction', color='green', alpha=0.7, linestyle='--')
        ax2.set_title(f'Window {window_id}: Predictions vs Actual')
        ax2.set_xlabel('Time Step')
        ax2.set_ylabel('Realized Volatility')
        ax2.legend()
        ax2.grid(True, alpha=0.3)

        # Add performance metrics as text
        metrics = evaluation['metrics']
        text_str = f"RMSE: {metrics['rmse']:.6f}\nMAE: {metrics['mae']:.6f}\nR²: {metrics['r2']:.4f}"
        ax2.text(0.02, 0.98, text_str, transform=ax2.transAxes,
                verticalalignment='top', bbox=dict(boxstyle='round', facecolor='lightgreen', alpha=0.5))

        # Plot 3: Scatter Plot (Predicted vs Actual)
        ax3 = fig.add_subplot(gs[idx, 2])
        ax3.scatter(true_vals, pred_vals, alpha=0.5, color='green')

        # Add perfect prediction line
        min_val = min(true_vals.min(), pred_vals.min())
        max_val = max(true_vals.max(), pred_vals.max())
        ax3.plot([min_val, max_val], [min_val, max_val], 'r--', label='Perfect Prediction')

        ax3.set_title(f'Window {window_id}: Prediction Accuracy')
        ax3.set_xlabel('True Volatility')
        ax3.set_ylabel('Predicted Volatility')
```

```
        ax3.legend()
        ax3.grid(True, alpha=0.3)

    # Summary plots at the bottom
    if len(summary_df) > 0:
        # Plot 4: RMSE across windows
        ax4 = fig.add_subplot(gs[n_windows, 0])
        ax4.bar(summary_df['Window'], summary_df['Test_RMSE'], color='steelblue', alpha=0.7)
        ax4.set_title('Test RMSE Across Windows')
        ax4.set_xlabel('Window')
        ax4.set_ylabel('RMSE')
        ax4.grid(True, alpha=0.3, axis='y')

        # Plot 5: R² across windows
        ax5 = fig.add_subplot(gs[n_windows, 1])
        ax5.bar(summary_df['Window'], summary_df['Test_R2'], color='seagreen', alpha=0.7)
        ax5.axhline(y=0, color='red', linestyle='--', alpha=0.5)
        ax5.set_title('R² Score Across Windows')
        ax5.set_xlabel('Window')
        ax5.set_ylabel('R²')
        ax5.grid(True, alpha=0.3, axis='y')

        # Plot 6: Number of support vectors
        ax6 = fig.add_subplot(gs[n_windows, 2])
        ax6.bar(summary_df['Window'], summary_df['N_Support_Vectors'], color='coral', alpha=0.7)
        ax6.set_title('Number of Support Vectors')
        ax6.set_xlabel('Window')
        ax6.set_ylabel('Support Vectors')
        ax6.grid(True, alpha=0.3, axis='y')

    plt.suptitle(f'{asset_name} SVM Volatility Forecasting Results',
                 fontsize=16, fontweight='bold', y=0.995)

    return fig


def create_svm_summary_table(sp500_results, bitcoin_results):

    print("\n" + "="*100)
    print("SVM VOLATILITY FORECASTING RESULTS SUMMARY")
    print("="*100)

    # Overall comparison
    summary_data = {
        'Asset': ['S&P 500', 'Bitcoin'],
```

```
        'Windows_Tested': [
            sp500_results['performance_summary']['successful_windows'],
            bitcoin_results['performance_summary']['successful_windows']
        ],
        'Avg_RMSE': [
            sp500_results['performance_summary']['avg_test_rmse'],
            bitcoin_results['performance_summary']['avg_test_rmse']
        ],
        'Std_RMSE': [
            sp500_results['performance_summary']['std_test_rmse'],
            bitcoin_results['performance_summary']['std_test_rmse']
        ],
        'Avg_MAE': [
            sp500_results['performance_summary']['avg_test_mae'],
            bitcoin_results['performance_summary']['avg_test_mae']
        ],
        'Avg_R2': [
            sp500_results['performance_summary']['avg_r2'],
            bitcoin_results['performance_summary']['avg_r2']
        ],
        'Avg_Direction_Acc_%': [
            sp500_results['performance_summary']['avg_direction_accuracy'],
            bitcoin_results['performance_summary']['avg_direction_accuracy']
        ],
        'Avg_Support_Vectors': [
            sp500_results['performance_summary']['avg_support_vectors'],
            bitcoin_results['performance_summary']['avg_support_vectors']
        ]
    }

    summary_df = pd.DataFrame(summary_data)

    print("\n OVERALL PERFORMANCE COMPARISON")
    print("-" * 100)
    print(summary_df.to_string(index=False, float_format='%.6f'))

    # Detailed window results
    print(f"\n S&P 500 DETAILED RESULTS")
    print("-" * 100)
    print(sp500_results['summary_df'].round(6).to_string(index=False))

    print(f"\n BITCOIN DETAILED RESULTS")
    print("-" * 100)
    print(bitcoin_results['summary_df'].round(6).to_string(index=False))
```

```
    # Methodology summary
    print(f"\n METHODOLOGY SUMMARY")
    print("-" * 100)
    print(f"Model Type: {sp500_results['methodology']['model_type']}")
    print(f"Kernel: {sp500_results['methodology']['kernel']}")
    print(f"Lookback Window: {sp500_results['methodology']['lookback_window']} days")
    print(f"Volatility Window: {sp500_results['methodology']['volatility_window']} days")
    print(f"Target Variable: {sp500_results['methodology']['target']}")
    print(f"Hyperparameter Search: {sp500_results['methodology']['hyperparameter_search']}")
    print(f"Validation Strategy: {sp500_results['methodology']['validation']}")

    return summary_df
```

In [ ]:
```
# Generate Comprehensive Results and Visualizations
# Display all SVM analysis results

# Create summary table
summary_table = create_svm_summary_table(sp500_svm_results, bitcoin_svm_results)

# Visualize S&P 500 results
print("\n" + "="*100)
print("GENERATING VISUALIZATIONS")
print("="*100)

fig1 = plot_svm_results(sp500_svm_results, max_windows=3)
plt.show()

# Visualize Bitcoin results
fig2 = plot_svm_results(bitcoin_svm_results, max_windows=3)
plt.show()
```

In [ ]:
```
# Optional: Run Full Analysis on All Windows
# Uncomment and run this cell to process all CV windows (will take longer)


# Full S&P 500 Analysis (all 16 windows)
print("Running FULL S&P 500 SVM Analysis (all windows)...")
sp500_svm_full = run_svm_cross_validation(
    cv_splits=sp500_cv_splits,  # All windows
    data_clean=sp500_clean,
    asset_name='S&P 500',
    **SVM_CONFIG,
    verbose=True
)
```

```
# Full Bitcoin Analysis (all 11 windows)
print("\\nRunning FULL Bitcoin SVM Analysis (all windows)...")
bitcoin_svm_full = run_svm_cross_validation(
    cv_splits=bitcoin_cv_splits,  # All windows
    data_clean=bitcoin_clean,
    asset_name='Bitcoin',
    **SVM_CONFIG,
    verbose=True
)

# Generate complete summary
full_summary = create_svm_summary_table(sp500_svm_full, bitcoin_svm_full)
print("\n✓ Full SVM analysis complete!")
```

Model Comparison: ARIMA vs LSTM vs SVM (error metrics)

In [ ]:

```
# Model Comparison: ARIMA vs LSTM vs SVM
# Compare all three forecasting approaches

def create_model_comparison_table(arima_results, lstm_results, svm_results, asset_name):

    print(f"\n{'='*100}")
    print(f"{asset_name.upper()}: COMPREHENSIVE MODEL COMPARISON")
    print(f"{'='*100}")

    comparison_data = {
        'Model': ['ARIMA', 'LSTM', 'SVM'],
        'Type': ['Statistical', 'Deep Learning', 'Machine Learning'],
        'Avg_RMSE': [
            arima_results['performance_summary']['avg_test_rmse'],
            lstm_results['performance_summary']['avg_test_rmse'],
            svm_results['performance_summary']['avg_test_rmse']
        ],
        'Avg_MAE': [
            arima_results['summary_df']['Test_MAE'].mean() if 'Test_MAE' in arima_results['summary_df'].columns else np.nan,
            lstm_results['performance_summary']['avg_test_mae'],
            svm_results['performance_summary']['avg_test_mae']
        ],
        'Avg_R2': [
            arima_results['performance_summary']['avg_r2'],
            lstm_results['performance_summary']['avg_r2'],
```

```
            svm_results['performance_summary']['avg_r2']
        ],
        'Direction_Acc_%': [
            arima_results['performance_summary']['avg_direction_accuracy'],
            lstm_results['performance_summary']['avg_direction_accuracy'],
            svm_results['performance_summary']['avg_direction_accuracy']
        ],
        'Windows': [
            arima_results['performance_summary']['successful_windows'],
            lstm_results['performance_summary']['successful_windows'],
            svm_results['performance_summary']['successful_windows']
        ]
    }

    comparison_df = pd.DataFrame(comparison_data)

    print("\n📊 PERFORMANCE METRICS COMPARISON")
    print("=" * 100)
    print(comparison_df.to_string(index=False, float_format='%.6f'))

    # Determine best model by RMSE
    best_idx = comparison_df['Avg_RMSE'].idxmin()
    best_model = comparison_df.loc[best_idx, 'Model']
    best_rmse = comparison_df.loc[best_idx, 'Avg_RMSE']

    print(f"\n🏆 BEST MODEL (Lowest RMSE): {best_model} with RMSE = {best_rmse:.6f}")

    # Create visualization
    fig, axes = plt.subplots(1, 3, figsize=(18, 5))

    models = comparison_df['Model']

    # Plot 1: RMSE Comparison
    axes[0].bar(models, comparison_df['Avg_RMSE'], color=['blue', 'red', 'green'], alpha=0.7)
    axes[0].set_title('Average Test RMSE')
    axes[0].set_ylabel('RMSE')
    axes[0].grid(True, alpha=0.3, axis='y')

    # Plot 2: Direction Accuracy Comparison
    axes[1].bar(models, comparison_df['Direction_Acc_%'], color=['blue', 'red', 'green'], alpha=0.7)
    axes[1].axhline(y=50, color='black', linestyle='--', label='Random (50%)')
    axes[1].set_title('Direction Accuracy')
    axes[1].set_ylabel('Accuracy (%)')
    axes[1].legend()
    axes[1].grid(True, alpha=0.3, axis='y')
```

```
    # Plot 3: R² Score Comparison (All Models)
    r2_models = ['ARIMA', 'LSTM', 'SVM']
    r2_values = [
        arima_results['performance_summary']['avg_r2'],
        lstm_results['performance_summary']['avg_r2'],
        svm_results['performance_summary']['avg_r2']
    ]
    axes[2].bar(r2_models, r2_values, color=['blue', 'red', 'green'], alpha=0.7)
    axes[2].axhline(y=0, color='black', linestyle='--', alpha=0.5)
    axes[2].set_title('R² Score (Coefficient of Determination)')
    axes[2].set_ylabel('R²')
    axes[2].grid(True, alpha=0.3, axis='y')

    plt.suptitle(f'{asset_name}: Model Performance Comparison', fontsize=16, fontweight='bold')
    plt.tight_layout()

    return comparison_df, fig


# Compare models for S&P 500
print("\n" + "="*100)
print("GENERATING MODEL COMPARISONS")
print("="*100)

sp500_comparison, sp500_fig = create_model_comparison_table(
    sp500_arima_results,
    sp500_lstm_full,
    sp500_svm_full,
    'S&P 500'
)
plt.show()

# Compare models for Bitcoin
bitcoin_comparison, bitcoin_fig = create_model_comparison_table(
    bitcoin_arima_results,
    bitcoin_lstm_full,
    bitcoin_svm_full,
    'Bitcoin'
)
plt.show()
```

Model Comparison: ARIMA vs LSTM vs SVM (performance indicators)

In [ ]:
```python
def volatility_predictions_to_returns_new(predictions, true_values, actual_returns, transaction_costs=0.0):
    min_len = min(len(predictions), len(true_values), len(actual_returns))
    predictions = predictions[:min_len]
    true_values = true_values[:min_len]
    actual_returns = (actual_returns.iloc[:min_len]
                      if isinstance(actual_returns, pd.Series)
                      else actual_returns[:min_len])

    actual_returns_array = (actual_returns.values
                            if isinstance(actual_returns, pd.Series)
                            else actual_returns)

    signal = np.where(predictions > transaction_costs, 1, -1)

    if transaction_costs > 0.0:
        signals = np.where(np.abs(actual_returns_array) > transaction_costs, signal, 0)

    strategy_returns = signals * actual_returns_array

    return pd.Series(strategy_returns)
```

In [ ]:
```python
def extract_model_returns_from_results_fixed(model_results, data_clean, model_type="S&P", window_indices=None):
    if model_type == "S&P":
        cost = 0.005
    elif model_type == "Bitcoin":
        cost = 0.065
    elif model_type == "EURUSD":
        cost = 0.0001
    all_strategy_returns = []

    windows_to_use = model_results['all_results']
    if window_indices is not None:
        windows_to_use = [w for w in windows_to_use if w['window_id'] in window_indices]

    for window_result in windows_to_use:
        try:
            # Get test period dates
            test_start = window_result['test_period'].split(' to ')[0]
            test_end = window_result['test_period'].split(' to ')[1]

            # Get actual returns during test period
            test_data = data_clean[test_start:test_end]
```

```
            # Get predictions - handle different model structures
            evaluation = window_result['evaluation']

            # ARIMA uses 'forecasts' (pandas Series), LSTM/SVM use 'predictions' (numpy arrays)
            if 'forecasts' in evaluation:
                # ARIMA model - predicts returns directly
                predictions = evaluation['forecasts'].values
                true_values = test_data['Log_Returns'].values[-len(predictions):]
            elif 'predictions' in evaluation:
                # LSTM or SVM model - predicts volatility
                predictions = evaluation['predictions']
                true_values = evaluation['true_values']
            else:
                print(f"⚠️  Warning: Unknown evaluation structure for window {window_result['window_id']}")
                continue

            # Align returns with predictions
            actual_returns = test_data['Log_Returns'].iloc[-len(predictions):]

            window_returns = volatility_predictions_to_returns_new(
                predictions, true_values, actual_returns.values, transaction_costs= cost
            )

            all_strategy_returns.append(window_returns)

        except Exception as e:
            print(f"⚠️  Warning: Failed to process window {window_result.get('window_id', '?')}: {str(e)}")
            continue

    if all_strategy_returns:
        return pd.concat(all_strategy_returns, ignore_index=True)
    else:
        return pd.Series([])
```

In [ ]:

```
def annualized_return(daily_returns):
    cumulative = (1 + daily_returns).prod()
    n = daily_returns.shape[0]
    return cumulative ** (TRADING_DAYS / n) - 1


def annualized_std(daily_returns):
    return daily_returns.std() * np.sqrt(TRADING_DAYS)
```

```python
def max_drawdown(daily_returns):
    equity = (1 + daily_returns).cumprod()
    peak = equity.cummax()
    drawdown = (equity - peak) / peak
    return np.abs(drawdown.min())  # Paper uses absolute value


def information_ratio(strategy_returns, benchmark_returns):
    arc = annualized_return(strategy_returns)
    asd = annualized_std(strategy_returns)

    if asd == 0:
        return np.nan
    return arc / asd


def modified_information_ratio(strategy_returns, benchmark_returns):
    arc = annualized_return(strategy_returns)
    asd = annualized_std(strategy_returns)
    md = max_drawdown(strategy_returns)

    if asd == 0 or md == 0:
        return np.nan

    return (arc * np.sign(arc) * arc) / (asd * md)


def sortino_ratio(daily_returns, risk_free_rate=0):
    negative_returns = daily_returns[daily_returns < 0]

    if len(negative_returns) == 0:
        return np.nan

    downside_std = np.std(negative_returns, ddof=1)
    asd_downside = downside_std * np.sqrt(TRADING_DAYS)

    arc = annualized_return(daily_returns)

    if asd_downside == 0:
        return np.nan

    return arc / asd_downside


def compute_performance_indicators(strategy_returns, benchmark_returns):
```

```
    return {
        "ARC": annualized_return(strategy_returns),
        "ASD": annualized_std(strategy_returns),
        "MD": abs(max_drawdown(strategy_returns)),
        "IR": information_ratio(strategy_returns, benchmark_returns),
        "IR*": modified_information_ratio(strategy_returns, benchmark_returns),
        "SR": sortino_ratio(strategy_returns)
    }
```

In [ ]:

```
# S&P 500 Evaluation - APPROACH A
TRADING_DAYS = 252

# Get benchmark returns (Buy-and-Hold)
sp500_bnh_returns = sp500_clean['Log_Returns'].loc["2007-01-01":"2023-12-29"].values


sp500_arima_strategy_returns = extract_model_returns_from_results_fixed(
    sp500_arima_results, sp500_clean
)
sp500_lstm_strategy_returns = extract_model_returns_from_results_fixed(
    sp500_lstm_full, sp500_clean
)
sp500_svm_strategy_returns = extract_model_returns_from_results_fixed(
    sp500_svm_full, sp500_clean
)

# Align lengths
min_len_lstmsvm = min(
    len(sp500_lstm_strategy_returns),
    len(sp500_svm_strategy_returns),
    len(sp500_bnh_returns)
)

min_len_arima = min(
    len(sp500_arima_strategy_returns),
    len(sp500_bnh_returns)
)


sp500_lstm_aligned = sp500_lstm_strategy_returns.iloc[:min_len_lstmsvm].values
sp500_svm_aligned = sp500_svm_strategy_returns.iloc[:min_len_lstmsvm].values
sp500_bnh_aligned = sp500_bnh_returns[-min_len_lstmsvm:]

sp500_arima_aligned = sp500_arima_strategy_returns[:min_len_arima]
```

```python
sp500_bnh_aligned2   = sp500_bnh_returns[-min_len_arima:]

results_sp500 = []

# ARIMA
arima_metrics = compute_performance_indicators(
    pd.Series(sp500_arima_aligned),
    pd.Series(sp500_bnh_aligned2)
)
arima_metrics['Model'] = 'ARIMA'
arima_metrics['Num_Trades'] = int(np.sum(np.abs(np.diff(sp500_arima_aligned > 0)) > 0))
results_sp500.append(arima_metrics)

TRADING_DAYS = 232
# LSTM
lstm_metrics = compute_performance_indicators(
    pd.Series(sp500_lstm_aligned),
    pd.Series(sp500_bnh_aligned)
)
lstm_metrics['Model'] = 'LSTM'
lstm_metrics['Num_Trades'] = int(np.sum(np.abs(np.diff(sp500_lstm_aligned > 0)) > 0))
results_sp500.append(lstm_metrics)

# SVM
svm_metrics = compute_performance_indicators(
    pd.Series(sp500_svm_aligned),
    pd.Series(sp500_bnh_aligned)
)
svm_metrics['Model'] = 'SVM'
svm_metrics['Num_Trades'] = int(np.sum(np.abs(np.diff(sp500_svm_aligned > 0)) > 0))
results_sp500.append(svm_metrics)

# Create TABLE 2
print("\nGenerating Table 2...")
table2_sp500 = pd.DataFrame(results_sp500)

print("\n" + "="*90)
print("TABLE 2: S&P 500 Long-Short Strategy Results")
print("="*90)
print(table2_sp500[['Model', 'ARC', 'ASD', 'MD', 'IR', 'IR*', 'SR']].to_string(index=False))

table2_sp500.to_csv('table2_sp500.csv', index=False)
```

In [ ]:

```
# BITCOIN Evaluation - APPROACH A

TRADING_DAYS = 365

# Get benchmark returns (Buy-and-Hold)
bitcoin_bnh_returns = bitcoin_clean['Log_Returns'].values

bitcoin_arima_strategy_returns = extract_model_returns_from_results_fixed(
    bitcoin_arima_results, bitcoin_clean, "Bitcoin"
)
bitcoin_lstm_strategy_returns = extract_model_returns_from_results_fixed(
    bitcoin_lstm_full, bitcoin_clean, "Bitcoin"
)
bitcoin_svm_strategy_returns = extract_model_returns_from_results_fixed(
    bitcoin_svm_full, bitcoin_clean, "Bitcoin"
)

# Align lengths
min_len_lstmsvm = min(
    len(bitcoin_lstm_strategy_returns),
    len(bitcoin_svm_strategy_returns),
    len(bitcoin_bnh_returns)
)

min_len_arima = min(
    len(bitcoin_arima_strategy_returns),
    len(bitcoin_bnh_returns)
)

bitcoin_lstm_aligned = bitcoin_lstm_strategy_returns.iloc[:min_len_lstmsvm].values
bitcoin_svm_aligned = bitcoin_svm_strategy_returns.iloc[:min_len_lstmsvm].values
bitcoin_bnh_aligned = bitcoin_bnh_returns[-min_len_lstmsvm:]

bitcoin_arima_aligned = bitcoin_arima_strategy_returns.iloc[:min_len_arima].values
bitcoin_bnh_aligned2 = bitcoin_bnh_returns[-min_len_arima:]


results_bitcoin = []

# ARIMA
arima_metrics = compute_performance_indicators(
    pd.Series(bitcoin_arima_aligned),
    pd.Series(bitcoin_bnh_aligned2)
)
```

```
arima_metrics['Model'] = 'ARIMA'
arima_metrics['Num_Trades'] = int(np.sum(np.abs(np.diff(bitcoin_arima_aligned > 0)) > 0))
results_bitcoin.append(arima_metrics)

TRADING_DAYS = 345

# LSTM
lstm_metrics = compute_performance_indicators(
    pd.Series(bitcoin_lstm_aligned),
    pd.Series(bitcoin_bnh_aligned)
)
lstm_metrics['Model'] = 'LSTM'
lstm_metrics['Num_Trades'] = int(np.sum(np.abs(np.diff(bitcoin_lstm_aligned > 0)) > 0))
results_bitcoin.append(lstm_metrics)

# SVM
svm_metrics = compute_performance_indicators(
    pd.Series(bitcoin_svm_aligned),
    pd.Series(bitcoin_bnh_aligned)
)
svm_metrics['Model'] = 'SVM'
svm_metrics['Num_Trades'] = int(np.sum(np.abs(np.diff(bitcoin_svm_aligned > 0)) > 0))
results_bitcoin.append(svm_metrics)

# Create TABLE 2
print("\nGenerating Table 2...")
table2_bitcoin = pd.DataFrame(results_bitcoin)

print("TABLE: Bitcoin Long-Short Strategy Results")

print(table2_bitcoin[['Model', 'ARC', 'ASD', 'MD', 'IR', 'IR*', 'SR']].to_string(index=False))

table2_bitcoin.to_csv('table2_bitcoin.csv', index=False)
```

EURUSD Dataset Analysis

Individual Model Performance (ARIMA, LSTM, SVM)

Dataset: EUR/USD Exchange Rate (2009-08-11 to 2019-08-11)

In [ ]:
```python
# Download EURUSD data
print("Downloading EURUSD data...")
eurusd_data = yf.download("EURUSD=X", start="2009-08-11", end="2019-08-11", progress=False)

print(f"\nEURUSD Data Shape: {eurusd_data.shape}")
print(f"EURUSD Date Range: {eurusd_data.index.min()} to {eurusd_data.index.max()}")
print(f"Total EURUSD observations: {len(eurusd_data)}")

# Calculate log returns
eurusd_data['Log_Returns'] = np.log(eurusd_data['Close'] / eurusd_data['Close'].shift(1))

# Clean data
eurusd_clean = eurusd_data.dropna()

print(f"\nAfter cleaning: {len(eurusd_clean)} observations")
print(f"Mean daily return: {eurusd_clean['Log_Returns'].mean():.6f}")
print(f"Standard deviation: {eurusd_clean['Log_Returns'].std():.6f}")
```

In [ ]:
```python
# Create cross-validation splits for EURUSD
# Using a simplified approach: 2-year training + 1-year validation (8,16,24 months) + 6-month test
from dateutil.relativedelta import relativedelta
from datetime import timedelta

def create_eurusd_cv_splits(data, start_date=None):
    if start_date is None:
        start_date = data.index.min()

    cv_splits = []
    window_start = start_date

    while True:
        # Define window boundaries
        train_start = window_start
        train_end = train_start + relativedelta(years=2) - timedelta(days=1)

        # Validation periods (8, 16, 24 months)
        val_start = train_end + timedelta(days=1)
        val1_end = val_start + relativedelta(months=8) - timedelta(days=1)
        val2_end = val_start + relativedelta(months=16) - timedelta(days=1)
        val3_end = val_start + relativedelta(months=24) - timedelta(days=1)

        # Test period (6 months)
        test_start = val3_end + timedelta(days=1)
        test_end = test_start + relativedelta(months=6) - timedelta(days=1)
```

```
        # Check if we have enough data
        if test_end > data.index.max():
            break

        # Create splits for this window
        train_data = data[(data.index >= train_start) & (data.index <= train_end)]

        # Three validation folds
        val1_data = data[(data.index >= val_start) & (data.index <= val1_end)]
        val2_data = data[(data.index >= val_start) & (data.index <= val2_end)]
        val3_data = data[(data.index >= val_start) & (data.index <= val3_end)]

        test_data = data[(data.index >= test_start) & (data.index <= test_end)]

        cv_splits.append({
            'window_id': len(cv_splits) + 1,
            'train': {
                'data': train_data,
                'start': train_start,
                'end': train_end,
                'size': len(train_data)
            },
            'validation': [
                {
                    'fold': 1,
                    'data': val1_data,
                    'start': val_start,
                    'end': val1_end,
                    'size': len(val1_data),
                    'months': 8
                },
                {
                    'fold': 2,
                    'data': val2_data,
                    'start': val_start,
                    'end': val2_end,
                    'size': len(val2_data),
                    'months': 16
                },
                {
                    'fold': 3,
                    'data': val3_data,
                    'start': val_start,
                    'end': val3_end,
```

```
                    'size': len(val3_data),
                    'months': 24
                }
            ],
            'test': {
                'data': test_data,
                'start': test_start,
                'end': test_end,
                'size': len(test_data)
            }
        })

        # Move window forward by 1 year
        window_start += relativedelta(years=1)

    return cv_splits

# Create EURUSD CV splits
eurusd_cv_splits = create_eurusd_cv_splits(eurusd_clean)

print(f"\n=== EURUSD Cross-Validation Setup ===")
print(f"Total CV windows: {len(eurusd_cv_splits)}")
print(f"\nFirst window details:")
print(f"  Train: {eurusd_cv_splits[0]['train']['start'].date()} to {eurusd_cv_splits[0]['train']['end'].date()} ({eurusd_cv_spli
print(f"  Val 1: {eurusd_cv_splits[0]['validation'][0]['start'].date()} to {eurusd_cv_splits[0]['validation'][0]['end'].date()}
print(f"  Test:  {eurusd_cv_splits[0]['test']['start'].date()} to {eurusd_cv_splits[0]['test']['end'].date()} ({eurusd_cv_splits
```

In [ ]:
```
# ARIMA Model for EURUSD
print("EURUSD - ARIMA MODEL")


eurusd_arima_results = run_arima_cross_validation(
    cv_splits=eurusd_cv_splits,
    data_clean=eurusd_clean,
    asset_name='EURUSD',
    max_p=3,
    max_d=1,
    max_q=3,
    information_criterion='aic',
    verbose=True
)
```

In [ ]:
```
if len(eurusd_arima_results['summary_df']) > 0:
    print(f"\n EURUSD DETAILED RESULTS ({len(eurusd_arima_results['summary_df'])} windows)")
```

```
    print("-" * 50)
    print(eurusd_arima_results['summary_df'].round(6).to_string(index=False))


print(f"Successfully processed {eurusd_arima_results['performance_summary']['successful_windows']}/{eurusd_arima_results['perfor
print(f"Average Test RMSE: {eurusd_arima_results['performance_summary']['avg_test_rmse']:.6f} ± {eurusd_arima_results['performan
print(f"Mean Test MAE:  {eurusd_arima_results['performance_summary']['avg_test_mae']:.6f}")
print(f"Average Direction Accuracy: {eurusd_arima_results['performance_summary']['avg_direction_accuracy']:.2f}%")
print(f"Most Common Model: {eurusd_arima_results['performance_summary']['most_common_order']}")
```

In [ ]:
```
# LSTM Model for EURUSD
print("\n" + "="*80)
print("EURUSD - LSTM MODEL")
print("="*80)

eurusd_lstm_results = run_lstm_cross_validation(
    cv_splits=eurusd_cv_splits,
    data_clean=eurusd_clean,
    asset_name='EURUSD',
    **LSTM_CONFIG,
    verbose=True
)


print(eurusd_lstm_results['summary_df'].round(6).to_string(index=False))
```

In [ ]:
```
# SVM Model for EURUSD
print("\n" + "="*80)
print("EURUSD - SVM MODEL")
print("="*80)

eurusd_svm_results = run_svm_cross_validation(
    cv_splits=eurusd_cv_splits,
    data_clean=eurusd_clean,
    asset_name='EURUSD',
    **SVM_CONFIG,
    verbose=True
)

print(eurusd_svm_results['summary_df'].round(6).to_string(index=False))
```

In [ ]:
```
# Comparative Summary: ARIMA vs LSTM vs SVM for EURUSD
print("\n" + "="*80)
print("EURUSD - MODEL COMPARISON SUMMARY")
```

```
print("="*80)

models_comparison = pd.DataFrame({
    'Model': ['ARIMA', 'LSTM', 'SVM'],
    'Mean RMSE': [
        eurusd_arima_results['performance_summary']['avg_test_rmse'],
        eurusd_lstm_results['performance_summary']['avg_test_rmse'],
        eurusd_svm_results['performance_summary']['avg_test_rmse']
    ],
    'Std RMSE': [
        eurusd_arima_results['performance_summary']['std_test_rmse'],
        eurusd_lstm_results['performance_summary']['std_test_rmse'],
        eurusd_svm_results['performance_summary']['std_test_rmse']
    ],
    'Mean MAE': [
        eurusd_arima_results['performance_summary']['avg_test_mae'],
        eurusd_lstm_results['performance_summary']['avg_test_mae'],
        eurusd_svm_results['performance_summary']['avg_test_mae']
    ],
    'Direction Accuracy': [
        eurusd_arima_results['performance_summary']['avg_direction_accuracy'],
        eurusd_lstm_results['performance_summary']['avg_direction_accuracy'],
        eurusd_svm_results['performance_summary']['avg_direction_accuracy']
    ]
})

print("\n" + models_comparison.to_string(index=False))

# Find best model
best_rmse_idx = models_comparison['Mean RMSE'].idxmin()
best_mae_idx = models_comparison['Mean MAE'].idxmin()

print(f"\n{'='*80}")
print(f"Best Model by RMSE: {models_comparison.iloc[best_rmse_idx]['Model']} ({models_comparison.iloc[best_rmse_idx]['Mean RMSE'
print(f"Best Model by MAE:  {models_comparison.iloc[best_mae_idx]['Model']} ({models_comparison.iloc[best_mae_idx]['Mean MAE']:.
print(f"{'='*80}")
```

In [ ]:

```
TRADING_DAYS = 252

# Get benchmark returns (Buy-and-Hold)
eurusd_bnh_returns = eurusd_clean['Log_Returns'].values
```

```
eurusd_arima_strategy_returns = extract_model_returns_from_results_fixed(
    eurusd_arima_results, eurusd_clean, "EURUSD"
)
eurusd_lstm_strategy_returns = extract_model_returns_from_results_fixed(
    eurusd_lstm_results, eurusd_clean, "EURUSD"
)
eurusd_svm_strategy_returns = extract_model_returns_from_results_fixed(
    eurusd_svm_results, eurusd_clean, "EURUSD"
)

# Align lengths
min_len_lstmsvm = min(
    len(eurusd_lstm_strategy_returns),
    len(eurusd_svm_strategy_returns),
    len(eurusd_bnh_returns)
)

min_len_arima = min(
    len(eurusd_arima_strategy_returns),
    len(eurusd_bnh_returns)
)

eurusd_lstm_aligned = eurusd_lstm_strategy_returns.iloc[:min_len_lstmsvm].values
eurusd_svm_aligned = eurusd_svm_strategy_returns.iloc[:min_len_lstmsvm].values
eurusd_bnh_aligned = eurusd_bnh_returns[-min_len_lstmsvm:]

eurusd_arima_aligned = eurusd_arima_strategy_returns.iloc[:min_len_arima].values
eurusd_bnh_aligned2 = eurusd_bnh_returns[-min_len_arima:]


results_eurusd = []

# ARIMA
arima_metrics = compute_performance_indicators(
    pd.Series(eurusd_arima_aligned),
    pd.Series(eurusd_bnh_aligned2)
)
arima_metrics['Model'] = 'ARIMA'
arima_metrics['Num_Trades'] = int(np.sum(np.abs(np.diff(eurusd_arima_aligned > 0)) > 0))
results_eurusd.append(arima_metrics)


TRADING_DAYS = 345
```

```python
# LSTM
lstm_metrics = compute_performance_indicators(
    pd.Series(eurusd_lstm_aligned),
    pd.Series(eurusd_bnh_aligned)
)
lstm_metrics['Model'] = 'LSTM'
lstm_metrics['Num_Trades'] = int(np.sum(np.abs(np.diff(eurusd_lstm_aligned > 0)) > 0))
results_eurusd.append(lstm_metrics)


# SVM
svm_metrics = compute_performance_indicators(
    pd.Series(eurusd_svm_aligned),
    pd.Series(eurusd_bnh_aligned)
)
svm_metrics['Model'] = 'SVM'
svm_metrics['Num_Trades'] = int(np.sum(np.abs(np.diff(eurusd_svm_aligned > 0)) > 0))
results_eurusd.append(svm_metrics)


table2_eurusd = pd.DataFrame(results_eurusd)

print("TABLE: EURUSD Long-Short Strategy Results")
print(table2_eurusd[['Model', 'ARC', 'ASD', 'MD', 'IR', 'IR*', 'SR']].to_string(index=False))

table2_eurusd.to_csv('table2_eurusd.csv', index=False)
```