Comprehensive Volatility Forecasting: Enhanced GARCH & LSTM Methodologies

This document provides a detailed explanation of the Python script designed for analyzing financial time series data and forecasting volatility. It encompasses various models, from traditional econometric approaches (GARCH family) to a deep learning model (LSTM), with a focus on robust rolling forecast techniques and comprehensive evaluation.

1. Helper Functions

This section defines several utility functions that streamline data processing and model evaluation throughout the script.

1.1 realized_volatility (series_log_return)

This function calculates a proxy for realized volatility. It has a conditional behavior based on whether the input is a single scalar or a series/array.

```
def realized_volatility(series_log_return):
    if np.isscalar(series_log_return):
        return np.abs(series_log_return)
    return np.sqrt(np.sum(series_log_return**2))
```

Explanation:

- If series_log_return is a single value (e.g., when applied element-wise to a pandas Series), it returns the absolute value of that log return. This is effectively calculating daily realized volatility as the absolute daily log return.
- If series_log_return is an array or series (e.g., a window of intraday returns), it calculates the square root of the sum of squared returns, which is the standard definition of realized volatility over a period. In this script's context, due to how it's called with log_returns.apply(), vols will store absolute daily log returns.

1.2 RMSE (y_true, y_pred)

Calculates the Root Mean Squared Error (RMSE) between true and predicted values, handling NaN values gracefully.

```
def RMSE(y_true, y_pred):
    y_true_arr, y_pred_arr = np.array(y_true), np.array(y_pred)
    mask = ~np.isnan(y_true_arr) & ~np.isnan(y_pred_arr)
    return np.sqrt(mse(y_true_arr[mask], y_pred_arr[mask])) if np.sum(mask) >
0 else np.nan
```

Explanation:

- Converts inputs to NumPy arrays.
- Creates a mask to identify positions where *both* y_true and y_pred are non-NaN. This ensures that only valid pairs are used for calculation.
- Returns the RMSE if there are valid pairs; otherwise, returns np.nan.

1.3 RMSPE(y_true, y_pred)

Calculates the Root Mean Squared Percentage Error (RMSPE), robustly handling NaN values and division by zero.

```
def RMSPE(y_true, y_pred):
    y_true_arr, y_pred_arr = np.array(y_true), np.array(y_pred)
    mask = ~np.isnan(y_true_arr) & ~np.isnan(y_pred_arr) & (y_true_arr != 0)
    if np.sum(mask) == 0: return np.nan
    percentage_errors = (y_true_arr[mask] - y_pred_arr[mask]) /
y_true_arr[mask]
    return np.sqrt(np.mean(np.square(percentage_errors)))
```

Explanation:

- Similar to RMSE, it masks for non-NaN values in both arrays.
- Crucially, it also adds a condition (y_true_arr != 0) to prevent division by zero errors in the percentage calculation.
- If no valid, non-zero y_true values exist, it returns np.nan.

1.4 windowed_dataset_tf(series, window_size, batch_size, shuffle_buffer)

Prepares a time series dataset in a windowed format suitable for TensorFlow models like LSTMs.

```
def windowed_dataset_tf(series, window_size, batch_size, shuffle_buffer):
    dataset = tf.data.Dataset.from_tensor_slices(series.values)
    dataset = dataset.window(window_size + 1, shift=1, drop_remainder=True)
    dataset = dataset.flat_map(lambda window: window.batch(window_size + 1))
    dataset = dataset.shuffle(shuffle_buffer).map(lambda window: (window[:-
1], window[-1]))
    return dataset.batch(batch_size).prefetch(1)
```

Explanation:

- **tf.data.Dataset.from_tensor_slices()**: Converts the input series into a TensorFlow Dataset.
- **.window(window_size + 1, shift=1, drop_remainder=True)**: Creates windows of data. Each window contains window_size + 1 elements. shift=1 means the next window starts one step after the previous one. drop_remainder=True ensures all windows are of the exact specified size.
- **.flat_map(lambda window: window.batch(window_size + 1))**: Flattens the dataset of windows into a dataset of batches of window elements.
- **.shuffle(shuffle_buffer).map(lambda window: (window[:-1], window[-1]))**: Shuffles the windows and then maps each window into an input-output pair (features, label), where features are the first window_size elements and label is the last element.

- **.batch(batch_size).prefetch(1)**: Batches the dataset into groups of batch_size and prefetches 1 batch to optimize data pipeline performance.

2. Data Loading and Preparation

This section handles loading the financial data, processing it into returns and volatility, and preparing it for time series analysis.

```
tckr = 'BTC-USD'
try:
    ticker_historical = pd.read_csv('datasets/abc.csv')
except FileNotFoundError:
    print("Error: 'datasets/abc.csv' not found. A dummy dataset will be
used.")
    num_dummy_points = 3000
    ticker_historical = pd.DataFrame({
        'Close': 25000 + np.random.randn(num_dummy_points).cumsum() * 50
    })

log_returns = np.log(ticker_historical.Close /
ticker_historical.Close.shift(1)).dropna()
vols = log_returns.apply(realized_volatility) # This will be absolute log
returns

# Stationarity Test
adfuller_results = adfuller(log_returns)
print(f'ADF Test on Log Returns: Statistic={adfuller_results[0]:.4f}, p-
value={adfuller_results[1]:.4f}')
if adfuller_results[1] <= 0.05:
    print("Conclusion: Log return series is likely stationary.")
else:
    print("Conclusion: Log return series is likely non-stationary.")

# Data Splitting
split_time_1, split_time_2 = 2000, 2500
if len(log_returns) <= split_time_2:
    raise ValueError("Dataset is too small for the defined
train/validation/test splits.")

train_idx, val_idx = log_returns.index[:split_time_1],
log_returns.index[split_time_1:split_time_2]
ts_train, ts_val = log_returns[train_idx], log_returns[val_idx]
vol_train, vol_val = vols[train_idx], vols[val_idx]
```

Explanation:

- **tckr**: A string variable to hold the ticker symbol, used for plot titles.
- **Data Loading with Fallback**: The script attempts to load abc.csv. If not found, it generates a dummy dataset of 3000 points, useful for testing the script's logic without a real data file.
- **log_returns**: Calculated as the natural logarithm of the ratio of current to previous closing prices. dropna() removes the first NaN value.
- **vols**: This series represents the daily volatility proxy. It's calculated by applying the realized_volatility function to each element of log_returns. Since realized_volatility

returns the absolute value for scalar inputs, vols effectively becomes the **absolute daily log returns**. This is the target variable for volatility forecasting in the LSTM model.
- **ADF Test**: Performs an Augmented Dickey-Fuller test on the log_returns series to check for stationarity. A low p-value (typically \(\le 0.05\)) indicates stationarity.
- **Data Splitting**: The dataset is split into training and validation sets using integer indices:
    - split_time_1: Divides the training set from the validation set.
    - split_time_2: Marks the end of the validation set. There is no explicit "test" set defined for this script's evaluation phase, as forecasting is done *on the validation set* after training on the training set.
    - ValueError is raised if the dataset is too small for the defined splits.
- **ts_train, ts_val**: These hold the log_returns for the training and validation periods, respectively. These are used for fitting GARCH models.
- **vol_train, vol_val**: These hold the vols (absolute log returns) for the training and validation periods, respectively. These are used for training and evaluating the LSTM model, and as the true values for all model evaluations.

3. Initial Analysis and Visualization

This section provides visual insights into the data and its properties.

```
plt.figure(figsize=(15, 5))
plt.plot(ticker_historical['Close'])
plt.title(f'Historical Close Prices for {tckr}')
plt.grid(True)
plt.show()

fig, axes = plt.subplots(1, 2, figsize=(15, 4))
plot_acf(ts_train**2, ax=axes[0], title='ACF of Squared Returns (Training)')
plot_pacf(ts_train**2, ax=axes[1], title='PACF of Squared Returns
(Training)')
plt.show()
```

Explanation:

- **Historical Close Prices Plot**: Shows the raw price movement over time.
- **ACF and PACF of Squared Returns**: These plots are critical for detecting **volatility clustering** and for identifying potential orders (p and q) for GARCH-type models.
    - Significant spikes in the Autocorrelation Function (ACF) and Partial Autocorrelation Function (PACF) of the *squared returns* suggest the presence of ARCH/GARCH effects, meaning that periods of high volatility tend to be followed by high volatility, and vice-versa.

4. Model Evaluation

This is the core section where different forecasting models are implemented, trained (if applicable), used for rolling predictions, and evaluated.

4.1 results_summary Dictionary

Initializes a dictionary to store the RMSE and RMSPE for each model.

```
results_summary = {}
```

4.2 Baseline Models

Two simple baseline models are used for comparison: Mean Prediction and Random Walk.

```
# Baseline 1: Mean Prediction
mean_pred = pd.Series(vol_train.mean(), index=vol_val.index)
results_summary['Baseline (Mean)'] = {'RMSE': RMSE(vol_val, mean_pred),
'RMSPE': RMSPE(vol_val, mean_pred)}

# Baseline 2: Random Walk
rw_pred = vols.shift(1).loc[val_idx]
results_summary['Baseline (Random Walk)'] = {'RMSE': RMSE(vol_val, rw_pred),
'RMSPE': RMSPE(vol_val, rw_pred)}
```

Explanation:

- **Mean Prediction**: Predicts the average of the vol_train (training volatility) for all periods in the validation set.
- **Random Walk**: Predicts that tomorrow's volatility will be today's actual volatility. vols.shift(1) shifts the entire vols series by one period. .loc[val_idx] then selects the values corresponding to the validation period.

4.3 GARCH Family Models with Rolling Forecast

This section iterates through different GARCH model configurations, performing a rigorous expanding-window rolling forecast for each.

```
garch_configs = [
    {'name': 'GARCH(1,1)', 'color': 'red', 'params': {'p': 1, 'o': 0, 'q': 1,
'vol': 'GARCH', 'dist': 'normal'}},
    {'name': 'GJR-GARCH(1,1,1)', 'color': 'darkgreen', 'params': {'p': 1,
'o': 1, 'q': 1, 'vol': 'GARCH', 'dist': 'normal'}},
    {'name': 'EGARCH(1,1,1)', 'color': 'magenta', 'params': {'p': 1, 'o': 1,
'q': 1, 'vol': 'EGARCH', 'dist': 'normal'}},
    {'name': "EGARCH(1,1,1)-t", 'color': 'cyan', 'params': {'p': 1, 'o': 1,
'q': 1, 'vol': 'EGARCH', 'dist': 'StudentsT'}}
]

for config in garch_configs:
    print(f"Starting {config['name']} rolling forecast (this may take
time)...")
    predictions = []
    for i in range(len(ts_val)):
        # Define expanding training window: all log_returns up to the current
validation point
        train_window =
log_returns.iloc[:log_returns.index.get_loc(val_idx[i])]
        if len(train_window) < 20: # Ensure enough data to fit the model
```

```
            predictions.append(np.nan)
            continue
        try:
            # Instantiate and fit the GARCH model on the current training
window
            model = arch_model(train_window, **config['params'], mean='Zero')
            model_fit = model.fit(disp='off')
            # Forecast 1 step ahead from the end of the current window
            pred_vol = np.sqrt(model_fit.forecast(horizon=1,
reindex=False).variance.iloc[-1, 0])
            predictions.append(pred_vol)
        except Exception: # Catch any fitting errors (e.g., convergence
issues)
            predictions.append(np.nan)
        if i % 100 == 0 or i == len(ts_val) - 1:
            print(f"  {config['name']} forecast step {i+1}/{len(ts_val)}
completed.")

    rolling_preds = pd.Series(predictions, index=ts_val.index)
    results_summary[config['name']] = {'RMSE': RMSE(vol_val, rolling_preds),
'RMSPE': RMSPE(vol_val, rolling_preds)}

    plt.figure(figsize=(20, 6))
    plt.plot(vol_val, label='Daily Realized Volatility (Validation)')
    plt.plot(rolling_preds, color=config['color'], linestyle='--',
label=f"Predicted Volatility ({config['name']})")
    plt.title(f"{config['name']} Rolling Forecast - {tckr} Validation Data",
fontsize=15)
    plt.legend()
    plt.grid(True)
    plt.show()
```

Explanation:

- **garch_configs**: A list of dictionaries, each defining a specific GARCH model variant (e.g., GARCH(1,1), GJR-GARCH(1,1,1), EGARCH(1,1,1), and EGARCH(1,1,1) with Student's T distribution for residuals).
- **Rolling Forecast Loop**: This is a critical part. For each step in the ts_val (validation period):
    1. **train_window**: Dynamically creates an *expanding training window*. This window includes all log_returns data from the beginning of the dataset up to the current point in the validation set being forecast. This means the model is re-trained with progressively more data at each step.
    2. **Minimum Data Check**: if len(train_window) < 20: ensures that there's sufficient data to fit the GARCH model, preventing errors early in the validation period.
    3. **Model Instantiation and Fitting**: arch_model() is called with the current train_window and parameters from config['params']. mean='Zero' assumes a zero conditional mean, common for high-frequency financial returns in volatility modeling. model.fit(disp='off') trains the model silently.
    4. **Forecasting**: model_fit.forecast(horizon=1) predicts the 1-step-ahead conditional variance, which is then square-rooted to get volatility (pred_vol).
    5. **Error Handling**: A try-except block catches potential errors during model fitting (e.g., convergence issues), appending np.nan if an error occurs.

6. **Progress Updates**: Prints messages every 100 steps or at the end for visibility during long computations.

- **Results and Plotting**: After all rolling predictions are made for a given GARCH config, the RMSE and RMSPE are calculated and stored. A plot comparing the model's predictions against the actual vol_val is generated, showing how well the model tracks volatility over the validation period.

4.4 LSTM Model with Rolling Prediction

This section trains and evaluates an LSTM neural network for volatility forecasting using a rolling prediction approach similar to the GARCH models.

```
print("Starting LSTM model training...")
window_size_lstm, batch_size_lstm, shuffle_buffer_size_lstm = 14, 32, 1000
train_dataset_lstm = windowed_dataset_tf(vol_train, window_size_lstm,
batch_size_lstm, shuffle_buffer_size_lstm)

lstm_model = tf.keras.models.Sequential([
    tf.keras.layers.Lambda(lambda x: tf.expand_dims(x, axis=-1),
input_shape=[None]),
    tf.keras.layers.LSTM(28, activation='tanh'),
    tf.keras.layers.Dense(1)
])
lstm_model.compile(optimizer='adam', loss='mse')
lstm_model.fit(train_dataset_lstm, epochs=100, verbose=0)
print("Finished LSTM training. Starting LSTM rolling prediction...")

lstm_predictions = []
for i in range(len(vol_val)):
    # Find the absolute index in the original `vols` series
    loc = vols.index.get_loc(val_idx[i])
    # Ensure enough historical data for the window
    if loc < window_size_lstm:
        lstm_predictions.append(np.nan)
        continue
    # Extract the input sequence from the `vols` series
    input_sequence = vols.iloc[loc-window_size_lstm:loc].values[np.newaxis,
:]
    # Predict the next volatility
    prediction = lstm_model.predict(input_sequence, verbose=0)[0, 0]
    lstm_predictions.append(prediction)
    if i % 100 == 0 or i == len(vol_val) - 1:
        print(f"  LSTM forecast step {i+1}/{len(ts_val)} completed.")

rolling_preds_lstm = pd.Series(lstm_predictions, index=ts_val.index)
results_summary['LSTM'] = {'RMSE': RMSE(vol_val, rolling_preds_lstm),
'RMSPE': RMSPE(vol_val, rolling_preds_lstm)}

plt.figure(figsize=(20, 6))
plt.plot(vol_val, label='Daily Realized Volatility (Validation)')
plt.plot(rolling_preds_lstm, color='sienna', linestyle='--', label='Predicted
Volatility (LSTM)')
plt.title(f'LSTM Rolling Forecast - {tckr} Validation Data', fontsize=15)
plt.legend()
plt.grid(True)
```

```
plt.show()
```
Explanation:

- **Hyperparameters**: window_size_lstm defines the number of past timesteps used to predict the next. batch_size_lstm and shuffle_buffer_size_lstm are for TensorFlow dataset optimization.
- **train_dataset_lstm**: Created using the windowed_dataset_tf helper function on vol_train (training volatility).
- **LSTM Model Architecture**:
    - tf.keras.layers.Lambda: Reshapes the input from [batch_size, window_size] to [batch_size, window_size, 1] as LSTMs expect a 3D input [samples, timesteps, features].
    - tf.keras.layers.LSTM(28, activation='tanh'): A single LSTM layer with 28 units and hyperbolic tangent activation.
    - tf.keras.layers.Dense(1): A final dense layer to output the single predicted volatility value.
    - Compiled with adam optimizer and mse loss.
- **Training**: lstm_model.fit() trains the model for 100 epochs on the train_dataset_lstm.
- **Rolling Prediction Loop**:
    1. For each point in vol_val (validation volatility), it identifies the loc (absolute index) in the original vols series.
    2. It extracts the input_sequence of window_size_lstm past volatility values from the vols series ending just before the current loc.
    3. This sequence is then fed into the *already trained*lstm_model to get a prediction.
    4. **Key Difference from GARCH**: Unlike the GARCH models which are re-fitted at each step, the LSTM model is trained once on vol_train. The "rolling" aspect here is that the *input window* for prediction slides forward, always using the most recent window_size_lstm actual values available from the combined train and *already observed*val data to make the next prediction. This is common for LSTMs after initial training.
- **Results and Plotting**: Similar to GARCH, the LSTM's performance is evaluated using RMSE and RMSPE, and its predictions are plotted against the actual validation volatility.

5. Final Results Summary

This concluding section consolidates the evaluation metrics for all models and highlights the best performers.

```
print("\n" + "="*50)
print("           MODEL PERFORMANCE SUMMARY")
print("="*50)

results_df = pd.DataFrame.from_dict(results_summary, orient='index')
results_df = results_df.sort_values(by='RMSE')
print(results_df)

print("\n--- Best Performing Models ---")
print(f"Lowest RMSE:  {results_df['RMSE'].idxmin()}
({results_df['RMSE'].min():.6f})")
```

```
print(f"Lowest RMSPE: {results_df['RMSPE'].idxmin()}
({results_df['RMSPE'].min():.6f})")
print("="*50)
```

Explanation:

- **results_summary to DataFrame**: The collected RMSE and RMSPE values from the results_summary dictionary are converted into a pandas DataFrame.
- **Sorting**: The DataFrame is sorted by RMSE in ascending order, making it easy to see which models performed best in terms of absolute error.
- **Best Performers**: The model with the lowest RMSE and the model with the lowest RMSPE are explicitly identified and printed, providing a quick summary of the top-performing approaches based on each metric.