Volatility Forecasting Script Documentation: Initial Implementation (GARCH & LSTM)

This document provides a detailed explanation of the Python script prediction.py, designed for analyzing financial time series data and forecasting volatility using both traditional econometric models (GARCH family) and a deep learning approach (LSTM).

1. Introduction

The script aims to:

- Load and preprocess historical financial time series data.
- Analyze the stationarity and autocorrelation properties of returns and volatility.
- Split the data into training, validation, and test sets.
- Implement baseline forecasting models (Mean, Random Walk).
- Implement various GARCH-family models (GARCH, GJR-GARCH, EGARCH) with a robust rolling forecast methodology.
- Implement a Long Short-Term Memory (LSTM) neural network for volatility forecasting.
- Evaluate the performance of all models using RMSE and RMSPE metrics.
- Visualize actual vs. predicted volatility.

2. Setup and Library Imports

This section imports all necessary libraries. It's crucial to have these installed in your Python environment (pip install pandas numpy matplotlib seaborn scipy scikit-learn statsmodels arch tensorflow).

```
# Standard library imports
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from datetime import date, datetime, timedelta
from scipy import stats

# Scikit-learn for metrics
from sklearn.metrics import mean_squared_error as mse, r2_score

# Statsmodels for ADF test and ACF/PACF plots
from statsmodels.tsa.stattools import adfuller
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf

# ARCH library for GARCH models
from arch import arch_model
```

```
# TensorFlow for LSTM model
import tensorflow as tf

# Suppress warnings for cleaner output
import warnings

warnings.filterwarnings("ignore")
```

Explanation:

- **numpy**: Fundamental package for numerical operations, especially array manipulations.
- **pandas**: Used for data manipulation and analysis, particularly with DataFrames.
- **matplotlib.pyplot** & **seaborn**: For creating static, interactive, and animated visualizations.
- **datetime**: For handling date and time objects.
- **scipy.stats**: Provides statistical functions.
- **sklearn.metrics**: Offers various metrics for evaluating model performance.
- **statsmodels.tsa.stattools**: Contains statistical tests for time series, like the Augmented Dickey-Fuller (ADF) test.
- **statsmodels.graphics.tsaplots**: For plotting autocorrelation functions (ACF) and partial autocorrelation functions (PACF).
- **arch**: A specialized library for time series econometrics, particularly GARCH-type models.
- **tensorflow**: An open-source machine learning framework, used here for building and training the LSTM model.
- **warnings**: Used to suppress non-critical warnings for cleaner output.

3. Data Loading and Initial Visualization

This part of the script defines the date range for data, attempts to load a CSV file, and includes a fallback mechanism to generate dummy data if the file is not found. It then plots the historical closing prices.

```
# Define start and end dates
start_date = date(2010, 1, 1)
end_date = date(2023, 1, 1)

# Load data
try:
    df = pd.read_csv("datasets/abc.csv", index_col="Date", parse_dates=True)
except FileNotFoundError:
    print("File not found. Creating dummy data...")
    date_range = pd.date_range(start=start_date, end=end_date, freq="D")
    dummy_data = np.random.randn(len(date_range)).cumsum() + 100
    df = pd.DataFrame({"Close": dummy_data}, index=date_range)

# Plot closing prices
plt.figure(figsize=(10, 6))
plt.plot(df.index, df["Close"])
plt.title("Historical Closing Prices")
plt.xlabel("Date")
```

```
plt.ylabel("Price")
plt.grid(True)
plt.show()
```
Explanation:

- **start_date, end_date**: Define the period for which data is expected or generated.
- **try-except FileNotFoundError**: This robust block attempts to load datasets/abc.csv. If the file doesn't exist, it generates synthetic time series data mimicking a price series, ensuring the script can run even without the actual data file.
- **index_col='Date', parse_dates=True**: Important pandas arguments that set the 'Date' column as the DataFrame index and parse it as datetime objects, which is crucial for time series analysis.
- **plt.plot()**: Visualizes the Close prices over time, providing an initial overview of the data's trend and patterns.

4. Data Preprocessing

Financial time series analysis often requires transforming raw price data into returns and a measure of volatility. This section calculates simple returns, log returns, and absolute log returns as a proxy for realized volatility.

```
# Calculate returns
df["Simple_Return"] = df["Close"].pct_change()
df["Log_Return"] = np.log(df["Close"] / df["Close"].shift(1))

# Calculate realized volatility (absolute log returns)
df["Abs_Log_Return"] = np.abs(df["Log_Return"])

# Drop NA values
df.dropna(inplace=True)
```
Explanation:

- **Simple_Return**: Calculated as \((P_t - P_{t-1}) / P_{t-1}\). It represents the percentage change in price.
- **Log_Return**: Calculated as \(\ln(P_t / P_{t-1})\). Log returns are often preferred in financial modeling because they are additive over time and approximately normally distributed.
- **Abs_Log_Return**: This is a common proxy for *realized volatility*. Absolute log returns tend to exhibit volatility clustering (periods of high volatility followed by high volatility, and vice-versa), which is a key characteristic that GARCH and LSTM models aim to capture.
- **df.dropna(inplace=True)**: Removes any rows with NaN values, which typically occur in the first row after pct_change() or shift(1) operations.

5. Stationarity Testing

Stationarity is a critical assumption for many time series models. The Augmented Dickey-Fuller (ADF) test is used to check if a series is stationary.

```
def adf_test(series):
    result = adfuller(series)
    print(f"ADF Statistic: {result[0]}")
    print(f"p-value: {result[1]}")
    print("Critical Values:")
    for key, value in result[4].items():
        print(f"\t{key}: {value}")


print("\nADF Test for Log Returns:")
adf_test(df["Log_Return"])

print("\nADF Test for Absolute Log Returns:")
adf_test(df["Abs_Log_Return"])
```

Explanation:

- **adf_test(series) function**:
  - Takes a time series as input.
  - Calls adfuller() from statsmodels.tsa.stattools.
  - Prints the ADF Statistic, p-value, and critical values.
  - **Interpretation**: If the p-value is below a chosen significance level (e.g., 0.05), and the ADF Statistic is more negative than the critical values, the null hypothesis of non-stationarity is rejected, meaning the series is likely stationary.
- **Tests for Log_Return and Abs_Log_Return**: Both are tested because stationarity is desired for modeling. Financial returns are generally stationary, and a stationary volatility proxy is also beneficial for direct modeling.

6. Splitting Data

The data is split into training, validation, and test sets to properly evaluate model performance on unseen data and to tune hyperparameters (validation set).

```
train_size = int(len(df) * 0.7)
val_size = int(len(df) * 0.15)
test_size = len(df) - train_size - val_size

train = df[:train_size]
val = df[train_size : train_size + val_size]
test = df[train_size + val_size :]

# Plot data splits
plt.figure(figsize=(12, 6))
plt.plot(train.index, train["Abs_Log_Return"], label="Train")
plt.plot(val.index, val["Abs_Log_Return"], label="Validation")
plt.plot(test.index, test["Abs_Log_Return"], label="Test")
plt.title("Volatility Proxy Split")
plt.xlabel("Date")
```

```
plt.ylabel("Absolute Log Returns")
plt.legend()
plt.grid(True)
plt.show()

# ACF and PACF plots for squared returns
fig, ax = plt.subplots(1, 2, figsize=(16, 6))
plot_acf(df["Log_Return"] ** 2, lags=40, ax=ax[0], title="ACF of Squared
Returns")
plot_pacf(df["Log_Return"] ** 2, lags=40, ax=ax[1], title="PACF of Squared
Returns")
plt.show()
```

Explanation:

- **train_size, val_size, test_size**: Define the proportions for each split (70% train, 15% validation, 15% test).
- **train, val, test DataFrames**: The original DataFrame df is sliced into these three parts based on the calculated sizes.
- **Volatility Proxy Split Plot**: Visualizes how the Abs_Log_Return series is divided across the three sets. This helps confirm the chronological integrity of the split.
- **ACF and PACF plots for squared returns**:
    - Plotting the Autocorrelation Function (ACF) and Partial Autocorrelation Function (PACF) of *squared* log returns (df['Log_Return']**2) is crucial for identifying ARCH/GARCH effects (volatility clustering).
    - Significant spikes in these plots at various lags suggest that past squared returns can predict current squared returns, indicating the presence of conditional heteroskedasticity, which GARCH models are designed to capture.

7. Baseline Models

Two simple baseline models are implemented for comparison:

- **Mean Prediction**: Forecasts the historical mean of the training data.
- **Random Walk**: Forecasts the last observed value.

```
# Mean prediction
mean_vol = train["Abs_Log_Return"].mean()
mean_pred = np.full_like(test["Abs_Log_Return"], mean_vol)

# Random walk
rw_pred = test["Abs_Log_Return"].shift(1).dropna()
```

Explanation:

- **mean_pred**: Simply takes the average Abs_Log_Return from the training set and uses it as the prediction for all test set periods.
- **rw_pred**: Shifts the Abs_Log_Return series in the test set by one period, so the prediction for today is yesterday's actual value. dropna() removes the first NaN created by shift(1). Baselines provide a simple benchmark against which the more complex models can be judged.

8. GARCH Models

GARCH (Generalized Autoregressive Conditional Heteroskedasticity) models are econometric models specifically designed to capture volatility clustering. This script implements GARCH, GJR-GARCH, and EGARCH models using the arch library, employing a robust *rolling forecast with re-estimation*.

```
def rolling_forecast_garch(initial_train_data, test_data, vol_type="GARCH",
p=1, o=0, q=1):
    """
    Performs a rolling forecast for GARCH-type models by re-fitting the model
    on an expanding window of data at each step.

    Args:
        initial_train_data (pd.Series): The initial training data.
        test_data (pd.Series): The data to forecast over.
        vol_type (str): Type of volatility model ('GARCH', 'EGARCH', 'GARCH'
with o>0 for GJR).
        p (int): Order of the ARCH terms.
        o (int): Order of the asymmetric terms (for GJR-GARCH).
        q (int): Order of the GARCH terms.

    Returns:
        np.array: Array of forecasted volatilities.
    """
    forecasts = []
    # Create a copy to avoid modifying original train data, will be expanded
    current_data_for_fitting = initial_train_data.copy()

    for i in range(len(test_data)):
        # Define the ARCH model based on the current expanding window of data
        # For GJR-GARCH, vol='Garch' and o should be > 0
        if vol_type == "GARCH" and o > 0:  # GJR-GARCH specific
            model = arch_model(
                current_data_for_fitting, vol="Garch", p=p, o=o, q=q
            )
        else:  # Pure GARCH or EGARCH
            model = arch_model(
                current_data_for_fitting, vol=vol_type, p=p, q=q
            )

        # Fit the model on the current data
        # 'disp=off' suppresses the convergence output for each step
        model_fit = model.fit(disp="off")

        # Forecast 1 step ahead from the end of the current_data_for_fitting
        res = model_fit.forecast(horizon=1, reindex=False)

        # Extract the 1-step ahead variance forecast
        if not res.variance.empty:
            forecasts.append(res.variance.iloc[-1, 0])
        else:
            # This handles cases where forecast might fail (e.g., convergence
issues)
            forecasts.append(np.nan)
```

```
        # Append the actual observation from the test set to the data
        # for the next iteration's estimation. This creates an expanding
window.
        current_data_for_fitting = pd.concat(
            [current_data_for_fitting, test_data.iloc[i : i + 1]]
        )

    # Convert variances to volatilities, and handle any potential NaNs
    return np.sqrt(np.array(forecasts))


# Generate forecasts using the corrected rolling_forecast_garch function
# GARCH(1,1)
garch_pred = rolling_forecast_garch(
    train["Log_Return"], test["Log_Return"], vol_type="GARCH", p=1, o=0, q=1
)

# GJR-GARCH(1,1,1) - Note the 'o=1' for the leverage term
gjrgarch_pred = rolling_forecast_garch(
    train["Log_Return"], test["Log_Return"], vol_type="GARCH", p=1, o=1, q=1
)

# EGARCH(1,1) - 'o' parameter is ignored for EGARCH type
egarch_pred = rolling_forecast_garch(
    train["Log_Return"], test["Log_Return"], vol_type="EGARCH", p=1, o=0, q=1
)
```

Explanation:

- **rolling_forecast_garch function**:
  - This is the core of the GARCH forecasting. It implements an *expanding window* approach.
  - For each step in the test_data, the function:
    1. Initializes an arch_model with the current_data_for_fitting (which starts as the initial_train_data and expands).
    2. Fits (model.fit()) the GARCH model to this expanded dataset.
    3. Uses the fitted model (model_fit.forecast()) to predict the variance for the next step (horizon=1).
    4. Appends the actual observation from the test_data to current_data_for_fitting for the next iteration.
  - This re-fitting at each step is computationally intensive but provides the most robust form of rolling forecast as the model parameters adapt to new information.
  - The vol_type parameter allows specifying standard GARCH, or "Garch" with o>0 for GJR-GARCH (also known as TARCH), or "EGARCH" for Exponential GARCH.
  - Finally, the forecasted variances are converted to volatilities by taking the square root.
- **Model Types**:
  - **GARCH(1,1)**: A standard GARCH model where current conditional variance depends on the previous squared residual and the previous conditional variance.

- **GJR-GARCH(1,1,1)**: (also known as GARCH in mean with leverage, or TARCH). This model captures asymmetric effects in volatility, where negative shocks (bad news) can have a larger impact on future volatility than positive shocks (good news) of the same magnitude. This is achieved by setting o=1.
- **EGARCH(1,1)**: Exponential GARCH. This model also captures asymmetric effects, but it models the logarithm of the conditional variance, ensuring that forecasts of variance are always positive. It also avoids parameter constraints found in GARCH models.

9. LSTM Model

Long Short-Term Memory (LSTM) networks are a type of recurrent neural network (RNN) particularly well-suited for sequence prediction problems, making them viable for time series forecasting.

```
def create_lstm_model(input_shape):
    model = tf.keras.Sequential(
        [
            tf.keras.layers.LSTM(50, return_sequences=True,
input_shape=input_shape),
            tf.keras.layers.LSTM(50),
            tf.keras.layers.Dense(1),
        ]
    )
    model.compile(optimizer="adam", loss="mse")
    return model


# Prepare data for LSTM
def prepare_data(data, n_steps):
    X, y = [], []
    for i in range(len(data) - n_steps):
        X.append(data[i : i + n_steps])
        y.append(data[i + n_steps])
    return np.array(X), np.array(y)


n_steps = 10
X_train, y_train = prepare_data(train["Abs_Log_Return"].values, n_steps)
X_val, y_val = prepare_data(val["Abs_Log_Return"].values, n_steps)
X_test, y_test = prepare_data(test["Abs_Log_Return"].values, n_steps)

# Reshape data for LSTM [samples, timesteps, features]
X_train = X_train.reshape((X_train.shape[0], X_train.shape[1], 1))
X_val = X_val.reshape((X_val.shape[0], X_val.shape[1], 1))
X_test = X_test.reshape((X_test.shape[0], X_test.shape[1], 1))

# Create and train LSTM model
lstm_model = create_lstm_model((n_steps, 1))
history = lstm_model.fit(
    X_train, y_train, epochs=50, batch_size=32, validation_data=(X_val,
y_val), verbose=0
)
```

```
# Generate predictions
lstm_pred = lstm_model.predict(X_test).flatten()
```
Explanation:

- **create_lstm_model(input_shape) function**:
  - Defines a Sequential Keras model.
  - Uses two LSTM layers with 50 units each. The first layer has return_sequences=True to pass the full sequence output to the next LSTM layer. The second LSTM layer does not return sequences as it's followed by a Dense output layer.
  - The final Dense(1) layer produces a single output (the predicted volatility).
  - Compiled with adam optimizer and mse (Mean Squared Error) as the loss function, standard for regression tasks.
- **prepare_data(data, n_steps) function**:
  - Transforms the flat time series data into sequences (X) and corresponding target values (y).
  - n_steps defines the number of previous time steps to consider for predicting the next value (e.g., if n_steps=10, it uses the past 10 observations to predict the 11th).
- **Data Reshaping**: LSTM layers in Keras expect input in the format [samples, timesteps, features]. The X_train, X_val, X_test arrays are reshaped accordingly, adding a features dimension of 1.
- **Training**: The lstm_model.fit() method trains the network using the training and validation data. epochs=50 specifies 50 passes over the entire training dataset, and batch_size=32 processes 32 samples at a time. verbose=0 suppresses detailed output during training.
- **Prediction**: lstm_model.predict(X_test) generates forecasts for the test set. .flatten() converts the 2D output array into a 1D array.

10. Model Evaluation

This section defines metrics for evaluating model performance and then systematically calculates them for all implemented models. Finally, it visualizes the actual vs. predicted values.

```
def calculate_rmse(y_true, y_pred):
    return np.sqrt(mse(y_true, y_pred))


def calculate_rmspe(y_true, y_pred):
    # Filter out y_true == 0 to avoid division by zero
    non_zero_indices = y_true != 0
    if not np.any(non_zero_indices):  # Handle case where all true values are
zero
        return np.nan
    return (
        np.sqrt(
            np.mean(
                np.square(
```

```
                    (y_true[non_zero_indices] - y_pred[non_zero_indices])
                    / y_true[non_zero_indices]
                )
            )
        )
        * 100
    )


# Ensure all prediction arrays are series for proper indexing and alignment
garch_pred_series = pd.Series(garch_pred, index=test.index[:len(garch_pred)])
gjrgarch_pred_series = pd.Series(
    gjrgarch_pred, index=test.index[:len(gjrgarch_pred)]
)
egarch_pred_series = pd.Series(egarch_pred,
index=test.index[:len(egarch_pred)])
lstm_pred_series = pd.Series(
    lstm_pred, index=test.index[n_steps : n_steps + len(lstm_pred)]
)
mean_pred_series = pd.Series(mean_pred, index=test.index)
rw_pred_series = pd.Series(
    rw_pred, index=test.index[1:]
)  # rw_pred already shifted by 1

# Align all predictions and actuals to the common period
# The LSTM prediction length might be shorter than the GARCH ones by n_steps
# So, let's align everything to the LSTM's prediction length as it's the most
constrained.
test_aligned_lstm = test["Abs_Log_Return"].iloc[n_steps:]
common_index = test_aligned_lstm.index

models = {
    "Mean Prediction": mean_pred_series.reindex(common_index).dropna(),
    "Random Walk": rw_pred_series.reindex(common_index).dropna(),
    "GARCH": garch_pred_series.reindex(common_index).dropna(),
    "GJR-GARCH": gjrgarch_pred_series.reindex(common_index).dropna(),
    "EGARCH": egarch_pred_series.reindex(common_index).dropna(),
    "LSTM": lstm_pred_series.reindex(common_index).dropna(),
}

# The actual values for evaluation must also be reindexed to the common index
y_true_aligned = test_aligned_lstm.reindex(common_index).dropna()

results = []
for name, pred in models.items():
    # Only evaluate on indices where both y_true_aligned and pred have values
    eval_index = y_true_aligned.index.intersection(pred.index)

    if not eval_index.empty:
        rmse = calculate_rmse(y_true_aligned.loc[eval_index],
pred.loc[eval_index])
        rmspe = calculate_rmspe(y_true_aligned.loc[eval_index],
pred.loc[eval_index])
        results.append({"Model": name, "RMSE": rmse, "RMSPE": rmspe})
    else:
        results.append(
            {"Model": name, "RMSE": np.nan, "RMSPE": np.nan}
```

```
        )  # No common data to evaluate


results_df = pd.DataFrame(results)
print("\nModel Evaluation Results:")
print(results_df)

# Plot predictions vs actual
plt.figure(figsize=(15, 8))
plt.plot(y_true_aligned.index, y_true_aligned, label="Actual Volatility",
alpha=0.7)
for name, pred in models.items():
    # Plot only the part of the prediction that has corresponding actual
values
    plot_index = y_true_aligned.index.intersection(pred.index)
    if not plot_index.empty:
        plt.plot(plot_index, pred.loc[plot_index], label=name, alpha=0.7)
plt.title("Model Predictions vs Actual Volatility")
plt.xlabel("Date")
plt.ylabel("Volatility")
plt.legend()
plt.grid(True)
plt.show()
```

Explanation:

- **calculate_rmse(y_true, y_pred)**: Calculates the Root Mean Squared Error. This metric is sensitive to large errors and is in the same units as the target variable (volatility). A lower RMSE indicates better performance.
- **calculate_rmspe(y_true, y_pred)**: Calculates the Root Mean Squared Percentage Error. This metric expresses the error as a percentage of the actual value. It's useful for comparing models when the scale of the target variable varies significantly, but it can be highly sensitive to actual values close to zero.
    - **Robustness**: Includes checks to prevent division by zero errors if y_true contains zero values, which can sometimes occur with financial returns or volatility proxies.
- **Prediction Alignment**: This is a crucial and often overlooked step in multi-model evaluation for time series.
    - Different models might have different effective start points for their predictions in the test set (e.g., LSTM needs n_steps of past data, GARCH models forecast from the beginning of the test set, but their first valid forecast is effectively for the next period).
    - All predictions are converted to pd.Series with their correct time indices.
    - A common_index is established (based on the LSTM's prediction window, as it's the most restricted).
    - All prediction series and the actual Abs_Log_Return series are reindex()-ed to this common_index and dropna() is called to ensure only periods where *both* the actual value and the prediction are available are used for evaluation. This ensures fair comparison.
- **results DataFrame**: Stores the calculated RMSE and RMSPE for each model.

- **Prediction Plot**: Visualizes the actual volatility alongside the predictions from all models over the common test period. This allows for a quick visual assessment of how well each model tracks the true volatility fluctuations.