Bayesian ARIMA Project Documentation

This documentation guide will help you understand, set up, and effectively use the Bayesian ARIMA models for time series forecasting. This project has everything from helpful utilities to core functionality to model Bayesian-influenced ARIMA and SARIMA models. Each function has its own docstring which can be consulted if greater code-specific implementation details are needed.

Table of Contents

Project Overview

The **Bayesian ARIMA** project provides a robust framework for time series forecasting using Bayesian methods. Inspired by the mathematical foundations outlined in Columbia's MCMC Bayesian Lecture and Bayesian AutoRegressors, this project leverages PyMC3 for Bayesian inference, enabling probabilistic forecasting of stock prices.

Key Features:

- **Bayesian ARIMA and SARIMA Models**: Incorporate both non-seasonal and seasonal components.
- **Hierarchical Modeling**: Manage multiple timeframes (daily, hourly, minute) for comprehensive forecasting.
- **Ensemble Methods**: Combine forecasts from different models to improve prediction accuracy.

- **Utilities for Data Acquisition and Preprocessing**: Streamline data fetching and preparation.
- **Model Persistence**: Save and load trained models for future use without retraining.

Getting Started

This section will guide you through setting up the project on your local machine, installing necessary dependencies, and preparing your environment for training and forecasting.
This code has only been tested on my Windows 11 machine running Ubuntu 20.04.6 LTS on WSL2. Since Ubuntu 20.04 depends on Python 2.8, there are two options for the requirements file and its dependencies, with the `requirements3_8.txt` being the one for Python 3.8 and `requirements.txt` file being for later Python versions. If running a later Linux distribution, you can use the latter.
This system will work on on Windows as well, if you choose to use Anaconda to install PyMC or configure your Visual Studio C++ Build Tools on Windows to allow PyMC to call its underlying binaries. However, the following guide will continue under the assumption that you are using a WSL setup, or a Linux system

Installation

- **Install WSL or Linux** If you have a Linux machine, this *Step 1* can be skipped, but if not, then configure your Windows machine to use WSL2 to run a Unix-based distro.
- **Clone the Repository**
  Begin by cloning the project repository to your local machine:

```
git clone https://github.com/varun-un/Bayesian-ARIMA
cd Bayesian-ARIMA
```

- **Create a Virtual Environment**
  It's recommended to use a virtual environment to manage dependencies:

```
python3 -m venv venv
source venv/bin/activate
```

- **Install Dependencies**
  Install the required Python packages using `pip`:

```
pip install -r requirements.txt
```

- **Add Relative Import Path**
  Run the following command to add the `src/` folder to the list of paths that Python looks at. This will need to be done every time the console is re-opened, unless you add it to `.bashrc` or a global environment variable.

```
export PYTHONPATH="$PWD/src:$PYTHONPATH"
```

Once this is all setup, you're ready to skip to training the model.

Project Structure

```
bayesian-arima/
├── data/
│   ├── raw/
│   └── processed/
├── models/
│   ├── arima/
│   └── hierarchical/
├── src/
```

```
│   ├── ensemble/
│   │   ├── ensemble.py
│   │   ├── regression_ensemble.py
│   │   └── weighted_average.py
│   ├── model/
│   │   ├── bayesian_arima.py
│   │   ├── bayesian_sarima.py
│   │   ├── hierarchical_model.py
│   │   └── model_selection.py
│   └── utils/
│       ├── data_acquisition.py
│       ├── preprocessor.py
│       └── trading_time.py
├── train.py
├── predict.py
└── requirements.txt
```

Components:

- **data/**: Directory for storing datasets
  - **raw/**: Raw data fetched directly from data sources
  - **processed/**: Data after preprocessing steps
- **models/**: Directory for saving trained models.
  - **arima/**: Individual ARIMA and SARIMA models
  - **hierarchical/**: Hierarchical models managing multiple timeframes
- **src/**: Contains source code modules
  - **ensemble/**: Ensemble methods for combining forecasts
    - `ensemble.py`: Abstract base class for ensemble methods
    - `weighted_average.py`: Implements weighted average ensemble
    - `regression_ensemble.py`: Implements regression-based ensemble
  - **model/**: Core modeling classes and utilities.
    - `bayesian_arima.py`: Bayesian ARIMA model implementation
    - `bayesian_sarima.py`: Bayesian SARIMA model with seasonality
    - `hierarchical_model.py`: Manages hierarchical models across timeframes
    - `model_selection.py`: Utilities for selecting optimal model orders
  - **utils/**: Auxiliary utilities.
    - `data_acquisition.py`: Functions to fetch stock data
    - `preprocessor.py`: Data preprocessing functions
    - `trading_time.py`: Classes for handling trading time calculations
    - `metadata_manager.py`: Utilities to fetch metadata and sector data about a ticker
    - `postprocessor.py`: Functions for post-processing of data, which is mainly differencing inversion
- **train.py**: Script to train Bayesian ARIMA models
- **predict.py**: Script to make forecasts using trained models
- **requirements.txt**: Lists all Python dependencies

Core Components

This section delves into the core components of the project, explaining their functionalities and how they interrelate.

Bayesian ARIMA Models

Bayesian ARIMA models incorporate both autoregressive (AR) and moving average (MA) components within a Bayesian framework. This approach allows for probabilistic forecasting, capturing uncertainty in predictions.

bayesian_arima.py

Path: `src/model/bayesian_arima.py`
Description: Implements the `BayesianARIMA` class, facilitating the training and forecasting of ARIMA models using PyMC3.
Key Functionalities:

- Initialization: Define model order `(p, d, q)` and set up the model's name for saving and loading.
- Training: Train the ARIMA model using MCMC sampling to infer posterior distributions of parameters.
- Prediction: Generate forecasts based on posterior means of AR and MA coefficients.
- Model Persistence: Save and load trained models for future use.

Usage Example:

```
from model.bayesian_arima import BayesianARIMA
import pandas as pd

# load preprocessed data
data = pd.read_csv('data/processed/AAPL_processed.csv', index_col='Date',
        parse_dates=True)
y = data['Log_Returns']

# initialize and train the model
arima = BayesianARIMA(name='AAPL', p=2, d=1, q=1)
arima.train(y=y, draws=1000, tune=1000, target_accept=0.95)

# save the trained model
arima.save()

# generate forecasts
forecast = arima.predict(steps=5, last_observations=y.values[-2:])
print(forecast)
```

bayesian_sarima.py

Path: `src/model/bayesian_sarima.py`
Description: Extends the `BayesianARIMA` class to support Seasonal ARIMA (SARIMA) models, incorporating seasonal differencing and seasonal AR/MA terms. In terms of implementation, it is extremely similar, with just different inputs in for the constructor.
Key Functionalities:

- Seasonal Differencing: Remove seasonal patterns by differencing at seasonal lags.
- Seasonal AR/MA Components: Incorporate seasonal autoregressive and moving average terms.
- Hierarchical Structure: Manage multiple seasonal components across different timeframes.
- Seasonal Hyperparameters: The constructor can now take in values for the seasonal parameters:
    - `m`: Seasonality term
        - Data points before seasonal pattern repeats. For example, 12 in a monthly model to represent a new year
    - `P`: Autoregressive seasonality order
    - `D`: Seasonal differencing order
    - `Q`: Seasonal moving average order

hierarchical_model.py

Path: `src/model/hierarchical_model.py`
Description: Manages hierarchical models across different timeframes (daily, hourly, minute), allowing for comprehensive forecasting by integrating multiple ARIMA models.
Key Functionalities:

- Multiple Timeframes: Handle daily, hourly, and minute-level forecasts.
- Ensemble Integration: Combine forecasts from different models using ensemble methods.
- Model Training and Prediction: Train individual models and generate aggregated forecasts.

Usage Example:

```
from model.hierarchical_model import HierarchicalModel
from ensemble.weighted_average import WeightedAverageEnsemble
import pandas as pd

# initialize an ensemble method (optional)
ensemble_method = WeightedAverageEnsemble(weights=[0.3, 0.4, 0.3])

# initialize hierarchical model for a ticker
hier_model = HierarchicalModel(ticker='AAPL', ensemble=ensemble_method,
        memory_save=True)

# train all children sarima models
hier_model.train_models(num_draws=500, num_tune=200, target_accept=0.9)

# save the hierarchical model
hier_model.save()

# predict future value
future_time = pd.Timestamp("2025-01-15 10:30:00")
prediction = hier_model.predict_value(end_time=future_time)
print(f"Ensemble Prediction at {future_time}: {prediction}")
```

model_selection.py

Path: `src/model/model_selection.py`
Description: Provides utilities for selecting optimal ARIMA and SARIMA model orders based on statistical tests and information criteria. There are many hyperparameters that can be modified here to improve runtime, convergence speed, order exploration, etc. The exposed parameters are the max order to cap the `auto_arima` call to, however, within the function, you can change many of its aspects. The full documentation for `auto_arima` can be found here.
Key Functionalities:

- ADF Test: Check for stationarity in time series data.
- Auto ARIMA: Automatically determine the best `(p, d, q)` or `(p, d, q, P, D, Q)` order using `pmdarima`.
- SARIMA Order Determination: Extend ARIMA order selection to include seasonal components.

Ensemble Methods

Ensemble methods combine forecasts from multiple models to enhance prediction accuracy and robustness. This project includes both weighted average and regression-based ensemble techniques.

ensemble.py

Path: `src/ensemble/ensemble.py`
Description: Defines the abstract base class `Ensemble` that outlines the structure for ensemble methods.
Key Functionalities:

- Abstract Methods: `ensemble` and `train` methods must be implemented by subclasses.
- Interface Specification: Ensures consistency across different ensemble implementations.

weighted_average.py

Path: `src/ensemble/weighted_average.py`
Description: Implements the `WeightedAverageEnsemble` class, which combines forecasts using a predefined set of weights.
Key Functionalities:

- Weighted Combination: Multiply each forecast by its corresponding weight and sum the results.
- Weight Management: Allows updating weights post-initialization.

regression_ensemble.py

Path: `src/ensemble/regression_ensemble.py`
Description: Implements the `RegressionEnsemble` class, which uses a regression model (e.g., Linear Regression) to learn the optimal combination of forecasts.
Key Functionalities:

- Training: Fit a regression model using historical forecasts and actual values.
- Prediction: Generate a combined forecast by applying the trained regression model.

Utilities

Utility modules provide essential functions and classes that support the core modeling and ensemble processes. Not all of these are used by the core programs, yet they're left in for extra utility.

trading_time.py

Path: `src/utils/trading_time.py`
Description: The `TradingTimeDelta` class calculates time differences between two timestamps, considering only trading hours and trading days. This ensures that forecasts are aligned with actual trading periods.
Key Functionalities:

- Trading Days: Monday to Friday.
- Trading Hours: 9:30 AM to 4:00 PM.
- Delta Calculations: Compute time differences in various units (seconds, minutes, hours, days).
- Next Trading Time: Determine the next trading timestamp after a given time. These are static methods as they are interval-agnostic.
- Timestamp Generation: Create future trading timestamps based on increments. This is used to label time-series data with only valid time ranges that can be traded on (ignoring after hours trading).

Usage Example:

```python
from utils.trading_time import TradingTimeDelta
import pandas as pd

# Initialize TradingTimeDelta
start = "2024-10-15 10:30:00"
end = "2024-10-16 14:30:00"
delta = TradingTimeDelta(start_time=start, end_time=end)

# Get delta in different units
print("Delta Minutes:", delta.get_delta_minutes())  # Output: 630
print("Delta Hours:", delta.get_delta_hours())      # Output: 11
print("Delta Days:", delta.get_delta_days())        # Output: 1.615

# Generate future trading timestamps
future_time = pd.Timestamp("2025-01-15 10:30:00")
next_trading_time =
        TradingTimeDelta.get_next_trading_time(current_time=pd.Timestamp.now())
print(f"Next Trading Time: {next_trading_time}")
```

preprocessor.py

Path: `src/utils/preprocessor.py`
Description: The preprocessing module provides functions to prepare raw stock data for modeling, ensuring it meets the requirements for Bayesian ARIMA models.
Key Functionalities:

- Data Loading: Load stock data from CSV files.
- Log Returns Calculation: Normalize data by computing logarithmic returns.
- Stationarity Checks: Use the Augmented Dickey-Fuller (ADF) test to ensure the time series is stationary.
- Differencing: Apply differencing to stabilize the mean of the time series, a prerequisite for ARIMA modeling.
- Data Saving: Save processed data for subsequent model training.

data_acquisition.py

Path: `src/utils/data_acquisition.py`
Description: Contains functions to fetch historical stock data using the `yfinance` API and save it locally.
Key Functionalities:

- Data Fetching: Retrieve stock data for specified intervals and date ranges.
- Data Saving: Save fetched data to CSV files for later use.

Training the Model

Training the Bayesian ARIMA models involves processing the stock data, selecting optimal model orders, and performing Bayesian inference to estimate model parameters.

train.py

Path: `src/train.py`
Description: Script to train hierarchical Bayesian ARIMA models for a specified stock ticker.
Usage:
Run the script via the command line, specifying the stock ticker and optional hyperparameters.
Command Syntax:

```
python3 train.py <TICKER> [--num_draws NUM_DRAWS] [--num_tune NUM_TUNE] [--
        target_accept TARGET_ACCEPT]
```

Parameters:

- `<TICKER>`: **(Required)** Stock ticker symbol (e.g., `AAPL`, `MSFT`).
- `--num_draws`: **(Optional)** Number of samples to draw from the posterior distribution (default: `300`).
- `--num_tune`: **(Optional)** Number of tuning steps for the MCMC sampler (default: `100`).
- `--target_accept`: **(Optional)** Target acceptance rate for the sampler (default: `0.95`).

Example Commands:

1. **Default Training**:
   Train the model for Apple Inc. (`AAPL`) with default hyperparameters.

   ```
   python3 train.py AAPL
   ```

2. **Custom Hyperparameters**: Train the model with 500 draws, 200 tuning steps, and a target acceptance rate of 0.9. `bash python3 train.py AAPL --num_draws 500 --num_tune 200 --target_accept 0.9`

What the Script Does:

1. Initialize Ensemble Method: Uses a `WeightedAverageEnsemble` with predefined weights.
2. Initialize Hierarchical Model: Creates a `HierarchicalModel` instance for the specified ticker.
3. Train Models: Trains individual `BayesianSARIMA` models for daily, hourly, and minute intervals.
4. Save Models: Persists the trained models to the `models/hierarchical/` directory.

Output:

- Trained models saved as `.pkl` files in the `models/hierarchical/` directory.
- Console logs indicating training progress and completion.

Hyperparameters

Understanding and tuning hyperparameters is crucial for optimizing model performance. Below are the key hyperparameters that can be adjusted:

1. **MCMC Sampling Parameters**:
   - **`num_draws`** (`draws` in code):
     - Description: Number of posterior samples to draw. Higher values improve the accuracy of the posterior distribution, but increase computation time.
     - Default: `300`
   - **`num_tune`** (`tune` in code):
     - Description: Number of tuning steps to allow the sampler to adapt. Higher values can improve sampler performance, especially for complex models.
     - Default: `100`
   - **`target_accept`** (`target_accept` in code):
     - Description: Target acceptance rate for the sampler.
     - Default: `0.95`
2. **Model Order Parameters**:
   - **ARIMA Orders** (`p`, `d`, `q`):
     - Description: Define the autoregressive (`p`), differencing (`d`), and moving average (`q`) components.

        - Selection: Determined automatically via `auto_arima` in `model_selection.py`. Users can override or specify manually if needed.
    - **SARIMA Orders** (`P`, `D`, `Q`, `m`):
        - Description: Define the seasonal autoregressive (`P`), seasonal differencing (`D`), seasonal moving average (`Q`), and seasonal period (`m`).
        - Selection: Also determined automatically. Adjust based on known seasonal patterns (e.g., `m=12` for monthly data).
3. **Ensemble Weights**:
    - **Weights** (`weights` in `WeightedAverageEnsemble`):
        - Description: Define the contribution of each model (daily, hourly, minute) to the final prediction.
        - Default: `[0.3, 0.4, 0.3]`
4. **Other Parameters**:
    - **`noise_scale`** (`noise_scale` in prediction methods):
        - Description: Scale factor for noise in predictions. Set to `0` for deterministic forecasts or adjust to control uncertainty.
        - Default: `0.1`

Making Predictions

Once the models are trained, you can generate forecasts for specific future timestamps using the `predict.py` script.

predict.py

Path: `src/predict.py`
Description: Script to generate forecasts using the trained hierarchical Bayesian ARIMA models for a specified stock ticker and future time.
Usage:
Run the script via the command line, specifying the stock ticker and the desired future timestamp.
Command Syntax:

```
python3 predict.py <TICKER> <END_TIME>
```

Parameters:

- `<TICKER>`: **(Required)** Stock ticker symbol (e.g., `AAPL`, `MSFT`).
- `<END_TIME>`: **(Required)** Future timestamp for the prediction in the format `"YYYY-MM-DD HH:MM:SS"` (e.g., `"2025-01-15 10:30:00"`).

Example Command:

```
python3 predict.py AAPL "2025-01-15 10:30:00"
```

What the Script Does:

1. Load Trained Model: Loads the hierarchical model for the specified ticker from the `models/hierarchical/` directory.
2. Calculate Time Delta: Determines the time difference between the current time and the specified `END_TIME`, considering trading hours.
3. Generate Forecasts: Uses individual ARIMA models to forecast future values for daily, hourly, and minute intervals.
4. Combine Forecasts: Aggregates forecasts using the ensemble method (e.g., weighted average).
5. Visualization: Plots historical data along with forecasted values and the ensemble prediction.
6. Output: Displays the ensemble prediction value in the console.

Output:

- Plot: Visual representation of historical data, individual model forecasts, and the ensemble prediction.
- Console Log: Numerical value of the ensemble prediction at the specified `END_TIME`.

Model Persistence

Ensuring that trained models are saved and can be reloaded without retraining is essential for efficiency and scalability.

Saving Models

Both individual `BayesianSARIMA` models and the `HierarchicalModel` are saved using Python's `dill` library. Models are stored as `.pkl` files in designated directories. You can save these by calling `save()` on either an individual model or a hierarchical model.

- **Individual Models**: Saved in `models/arima/`. If saved by the model, it will be named by the following convention, by default: `{name}-{p}-{d}-{q}-{P}-{D}-{Q}`. If saved by the hierarchical model, its name will be `{ticker_symbol}_{data_interval}`.
- **Hierarchical Models**: Saved in `models/hierarchical/`. It will save all instance variables except for the ARIMA models themselves, which will be saved under the `arima/` folder. This includes the trained ensemble model.

Loading Models

Models can be loaded back into the environment for making predictions or further analysis. `bayesian_arima`, `bayesian_sarima`, and `hierarchical_model` all have a `load()` method that can take in an optional filename parameter. If not passed in, it will use the default name that it would've saved the `.pkl` file to originally.
Requires that the object be initialized at first, before calling `load()`, which will populate all the instance variables appropriately. If the file does not exist, it will raise a `FileNotFound` error.
Example:

```
from model.hierarchical_model import HierarchicalModel

# Initialize hierarchical model
hier_model = HierarchicalModel(ticker='AAPL')

# Load the saved model
hier_model.load()  # Automatically loads from models/hierarchical/AAPL.pkl

# Generate prediction
future_time = pd.Timestamp("2025-01-15 10:30:00")
prediction = hier_model.predict_value(end_time=future_time)
print(f"Ensemble Prediction at {future_time}: {prediction}")
```