Names: Abdulmoiz & Wasif Mehboob

Roll No: 21I-2687 & 21I-1766 **Section:** M

Full System Documentation

Data Processing:

The project began with data preprocessing, where the primary focus was on preparing the dataset for modeling. The main steps included:

Data Loading:

The project utilized a dataset containing time-series data, with each row representing a specific timestamp and associated values. The dataset was loaded into a pandas DataFrame for further processing.

Data Cleaning:

Initial data cleaning involved handling missing values, outliers, and any inconsistencies in the dataset.

Feature Engineering:

The project involved creating additional features from the timestamp, such as extracting year and month information. These features were useful for understanding potential seasonality patterns in the data.

Resampling:

To work with monthly data, the DataFrame was resampled to aggregate the daily data into monthly intervals. The resampling was performed using the resample function in pandas, which grouped the data by month and retained the first value of each month. **Normalization:**

Normalization was applied to scale the data to a consistent range, ensuring that each feature contributed proportionally to the model's training process. The MinMaxScaler was used to scale the data to the range [0, 1].

Models Implementation

We implemented eight time-series forecasting models to analyze and predict future trends in the data. Each model was implemented based on its suitability for capturing different aspects of the data, including seasonality, trends, and irregularities. The implemented models include:

ARIMA (Autoregressive Integrated Moving Average):

Configured and tuned ARIMA models to capture non-stationary and seasonal patterns in the data. Model parameters were determined through statistical tests such as the ADF test for stationarity and ACF/PACF plots for parameter estimation.

ANN (Artificial Neural Networks):

Designed and trained ANN models to capture complex nonlinear relationships in the data that might be missed by ARIMA. Different architectures with varying numbers of layers and neurons were explored to find the best fit for the dataset.

SARIMA (Seasonal ARIMA):

Implemented SARIMA to model and forecast seasonal time series data. Determined the seasonal order of differencing and lagged forecast errors using ADF tests and seasonal ACF/PACF plots.

Exponential Smoothing (ETS):

Selected appropriate ETS models based on data characteristics and configured the models using error, trend, and seasonal components. Parameters were chosen based on model fit criteria such as AIC or BIC.

Prophet:

Utilized Prophet model to handle time series with strong seasonal effects and historical holidays. Defined the model with yearly, weekly, and daily seasonality components and adjusted the model's flexibility by tuning the seasonality mode.

Support Vector Regression (SVR):

Applied SVR to model nonlinear relationships in the data through the use of kernel functions. Tuned parameters such as C and gamma and utilized cross-validation to ensure generalization to unseen data.

Long Short-Term Memory (LSTM):

Designed LSTM networks suitable for sequence prediction problems. Defined network architectures with one or more LSTM layers, specified the number of neurons, and selected a backpropagation algorithm for training.

Hybrid Models Integration

We explored the integration of hybrid models, combining the strengths of ARIMA and ANN to enhance overall forecast accuracy. The integration process involved using the forecast results from the ARIMA model as input features to the ANN.

The ANN then modeled the residuals generated by the ARIMA model, allowing for correction and improvement of predictions based on the errors.

System Architecture

The system architecture comprises a Flask backend responsible for loading data from SQLite which was saved for each model, creating plots, and returning them to the frontend, built using React.js. The frontend solely focuses on rendering forecasted plots and interacting with users.

Backend (Flask)

The Flask backend serves as a bridge between the SQLite database and the frontend, handling data retrieval, plot generation, and response delivery.

1. **SQLite Interaction:** Flask interacts with the SQLite database to get forecast data, including historical data, forecasted values, and residual analysis results.

2. **Data Processing**: Upon receiving requests, Flask processes the retrieved data, including preprocessing for plot generation. It prepares the necessary data structures for visualization.

3. **Plot Generation:** Using library of Matplotlib, Flask generates forecasted plots based on the retrieved data. It creates plots for historical data, forecasted values, and residual analysis as requested by the frontend.

4. **HTTP Response:** Flask packages the generated plots into HTTP responses and sends them back to the frontend.

Frontend (React.js)

The React.js frontend focuses on presenting forecasted plots to users in an intuitive and interactive manner.

1. **User Interface:** The frontend provides a user-friendly interface where users can view forecasted plots, analyze data, and interact with visualization settings for forecasting models selection from drop down menu.

2. **HTTP Requests:** The frontend sends HTTP requests to the Flask backend to fetch forecast data and analysis results.

3. **Plot Rendering:** Upon receiving plot data from the backend, the frontend displays historical data, forecasted values, and residual analysis plots as requested.

4. **User Interaction:** Users can switch between forecast models and see their desired plots.

Workflow

1. **User Interaction:** Users interact with the frontend interface, selecting forecast models and initiating plot requests.

2. **HTTP Requests:** The frontend sends HTTP requests to the Flask backend, indicating the desired data and plot types.

3. **SQLite Data Retrieval:** The Flask backend interacts with the SQLite database to fetch forecast data and analysis results based on the received requests.

4. **Plot Generation:** Flask generates forecasted plots based on the retrieved data and analysis results.

5. **HTTP Response:** Flask packages the generated plots into HTTP responses and sends them back to the frontend.

6. **Plot Rendering:** The frontend receives the plot from the backend and users can interact with the plots and analyze forecasted values.

Contributions

1. **Team member 1:** Wasif Mehboob primarily focused on the implementation of the Flask backend and the React.js frontend to design the user interface of the application. This involved setting up API endpoints in Flask to handle requests for data and plotting, as well as creating the necessary components and routes in React.js to display the retrieved data and plots to the user. Additionally, Wasif implemented a method for future forecasting from models that did not have a forecast function but only had a predict function. This method involved using the

last explored data to initialize a future data array, which was then sent to the predict function to predict future values.

2. **Team member 2:** Abdulmoiz was responsible for data preprocessing, normalizing, and selecting data that had lower p-value. He explored three types of data which were (Difference between the Original and Rolling Mean), (Difference between the Original and Exponentially Weighted Mean), and differenced data by shift 1. (Difference between the Original and Rolling Mean) data was selected because it had lower p-value than others. Ahsan implemented models that had a forecast function and did testing of models on historical data with cross validation to check their performance using predict function. This involved configuring and training models. Ahsan ensured that these models were properly configured and trained to generate accurate forecasts.