REST API service: Predicting S&P 500's company Stock Prices using Neural Network Model

By Wong Tsz Yeung (20780319)

Objectives:

The primary objective of this project is to develop a cloud-native REST API service capable of predicting the future stock prices of companies listed in the S&P 500 index. The specific goals include:

- **Stock Price Prediction**: Utilize advanced neural network models to predict stock prices five days into the future for each S&P 500 company.
- **RESTful API Service**: Provide an accessible REST API for users to retrieve predictions, enabling easy integration with other applications or services.
- **Cloud-Native Architecture**: Implement the service using cloud-native features such as containerization to ensure scalability, portability, and efficient resource utilization.
- **Deploying:** Deploying the application on a cloud platform (AWS EC2) to take advantage of cloud services for load balancing, service discovery, and elastic scaling.

Design:

System Architecture

The application is designed as a microservices architecture, encapsulating the prediction logic and API service into separate, manageable components. This architecture promotes scalability, maintainability, and efficient resource utilization.

Microservice Components:

- **Model Training Service**: Responsible for training the neural network models for each S&P 500 company and saving the trained models.
- **Prediction API Service**: A Flask-based REST API that loads the trained models and provides endpoints for predictions.
- **Frontend Interface**: A simple web interface allowing users to select a ticker symbol and view predictions.

Cloud-Native Features:

- **Containerization**: Docker is used to containerize the application, ensuring consistency across different deployment environments and simplifying the deployment process.
- **Scalability and Deployment:** The application is deployed on AWS EC2 instances, leveraging AWS services for scalability and high availability.
- **Load Balancing and Service Discovery**: Cloud services can be utilized to distribute incoming requests and manage service instances efficiently.

Neural Network Model Design:

- **Neural Network Model**: The predictive model employs a hybrid neural network architecture combining Convolutional Neural Networks (CNN) and Long Short-Term Memory (LSTM) networks with attention mechanisms
- **Data Processing:** The application fetches historical stock data, performs feature engineering to calculate technical indicators, and prepares datasets suitable for training and prediction.

Implementation:

Model Training (train.py)

- **Data Preparation (feature_engineering.py):**
  - Fetches historical stock data and computes technical indicators.
  - Creates input sequences (traning_data) and corresponding labels (price_data) for model training.
- **Model Construction**:
  - Defines the neural network architecture with convolutional layers followed by bidirectional LSTM layers and an attention mechanism.
  - Applies regularization techniques to mitigate overfitting.
  - Compiles the model using the Mean Squared Error (MSE) loss function and the Adam optimizer.
- **Model Saving**:
  - Saves the trained model for each ticker symbol in the ./model/ directory in HDF5 format (.h5 files).
  - Checks if the model already exists to avoid retraining.

REST API Service (app.py)

- **Flask Application:**
  - Initializes a Flask app to serve RESTful API endpoints.
  - Implements the GET **/api/predict/<ticker_symbol>** endpoint to return stock price predictions.
  - Renders the frontend interface (index.html) at the root URL (/) which provides a search input for users to select an S&P 500 ticker symbol and then displays current price, predicted price, and percentage change after prediction.
- **Prediction Logic**:
  - Loads the pre-trained model corresponding to the requested ticker symbol.
  - Fetches recent stock data and technical indicators for the symbol.
  - Preprocesses the data to match the input format expected by the model.
  - Performs the prediction and computes the predicted stock prices.
  - Calculates the percentage change between the current price and the predicted price.
  - ➢ Example of GET **/api/predict/<ticker_symbol>**:

    ```
    {"current_price":128.42,
    "percentage_change":-0.26,
    "predicted_price":128.09,
    "success":true }
    ```

Dockerization (Dockerfile)

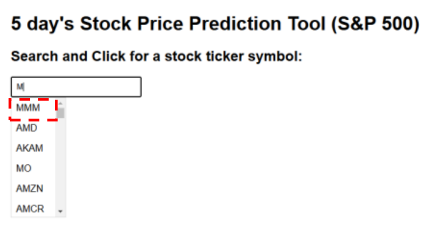
- **Base Image:** Utilizes the official Python 3.10 Docker image.
- **Environment Setup**:
  - Sets the working directory to /app.
  - Copies requirements.txt and installs the necessary Python packages without caching to reduce image size.
  - Copies the rest of the application code into the Docker image.
- **Exposure and Execution**:
  - Exposes port 5000 for the Flask web server.
  - Sets the command to run the Flask app, ensuring it listens on all interfaces (0.0.0.0) within the container.
    - `docker build -t flask-app .`
    - `docker run -p 8080:5000 flask-app`

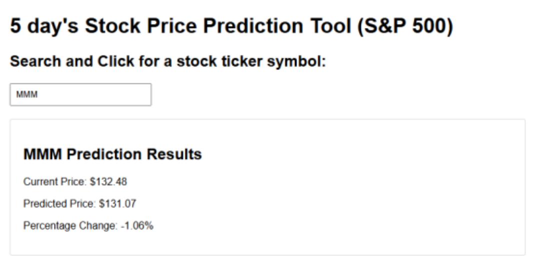
Deployment on AWS EC2

- **EC2 Instance Setup:**
  - **Instance Selection:** Chose an appropriate EC2 instance type (e.g., t2.medium) to balance performance and cost.
  - **Security Groups:** Configured security groups to allow inbound traffic on port 5000 for the Flask application and SSH access for management.
- **Environment Configuration:**
  - **Docker Installation:** Installed Docker on the EC2 instance to run the containerized application.
  - **Code Deployment:**
    - Cloned the project repository onto the EC2 instance.
    - Built the Docker image using the provided Dockerfile.
  - **Running the Container:**
    - Executed the Docker container, mapping the container's port 5000 to the host port 5000.
    - Used Docker commands to manage the container lifecycle (docker run, docker stop, etc.).

Result & Discussion:

The application successfully predicts stock prices for S&P 500 companies and provides these predictions through a RESTful API hosted on AWS EC2. Users can access predictions via the web interface or directly through the API.


5 day's Stock Price Prediction Tool (S&P 500)
Search and Click for a stock ticker symbol:
M
MMM
AMD
AKAM
MO
AMZN
AMCR


5 day's Stock Price Prediction Tool (S&P 500)

Search and Click for a stock ticker symbol:

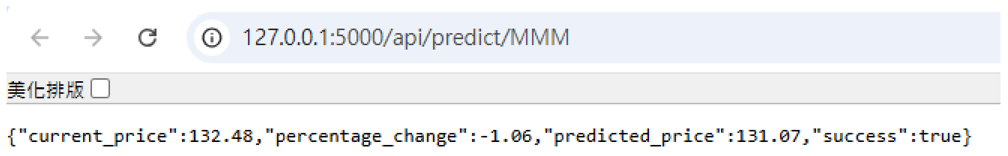
MMM

MMM Prediction Results

Current Price: $132.48

Predicted Price: $131.07

Percentage Change: -1.06%



```
{"current_price":132.48,"percentage_change":-1.06,"predicted_price":131.07,"success":true}
```

Advantages of Cloud-Native Architecture on AWS EC2

- **Scalability**:
  - **Elastic Scaling**: EC2 instances can be scaled vertically (larger instance types) or horizontally (adding more instances).
  - **Auto Scaling**: AWS Auto Scaling groups can automatically adjust the number of EC2 instances based on demand.
- **Flexibility**:
  - **Customization**: EC2 allows for deep customization of the instance environment, suitable for specific application needs.
  - **Integration with AWS Services**: Easy integration with other AWS services like S3 for storage, RDS for databases, and ELB for load balancing.
- **Cost-Efficiency**:
  - **Pay-as-You-Go**: Only pay for the compute time used, with options for reserved instances to reduce costs.
  - **Right-Sizing**: Ability to choose the appropriate instance type to match the application's performance requirements.

Performance Analysis

- **Response Time:** The REST API hosted on EC2 provides prompt responses, with low latency attributed to AWS's robust infrastructure.
- **Resource Utilization:** Monitoring indicates optimal CPU and memory usage, with the potential for scaling if demand increases.
- **Network Bandwidth:** Sufficient bandwidth to handle multiple simultaneous users without significant performance degradation.

Conclusion:

The project demonstrates the successful development of a cloud-native microservice-based application for predicting S&P 500 stock prices using a neural network model. Deploying the application on AWS EC2 showcases the advantages of cloud platforms in terms of scalability, flexibility, and integration with other services. While there are challenges associated with deployment and management in a cloud environment, the benefits in performance and the ability to handle varying loads make it a compelling approach for modern applications.