COMP4651 Project Report

Price prediction of stocks in S & P 500 Index

Group member: 1. Chan Chun Kit, 20865705 2.Wong Tsz Yeung, 20780319

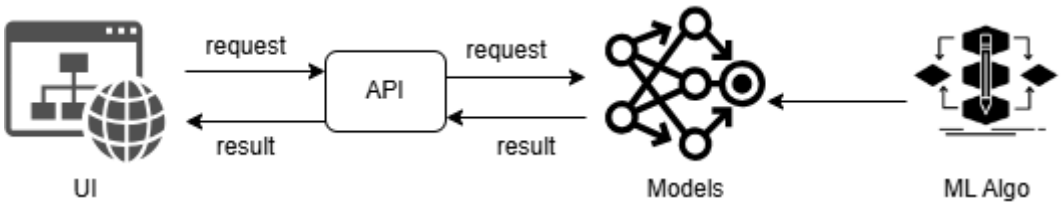
1. Introduction

The objective of this project is to design and implement a cloud-native microservice architecture for this project. We have built an api service which provides price prediction of stocks in the S & P 500 index. Stock prediction is a critical area in finance, where accurate forecasts can lead to significant economic benefits. Traditional monolithic applications often struggle to scale and adapt to changing requirements. This project aims to address these challenges by implementing a microservice architecture that enhances flexibility and scalability. By utilising cloud-native principles, the application can efficiently handle varying workloads and provide timely predictions to users.

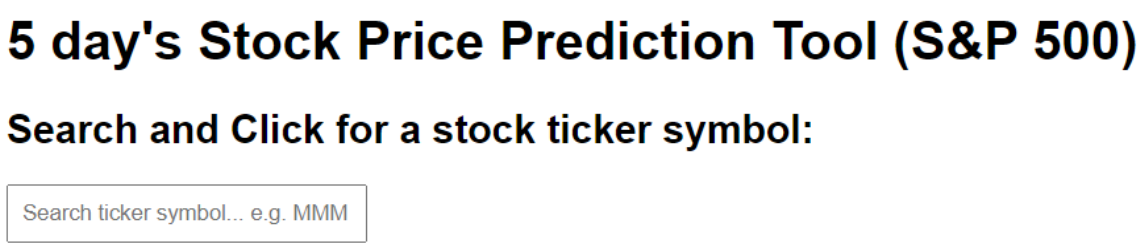
2. Architecture Overview

The service consists of three main components:

1. Frontend UI: A web application that allows users to search and click for a stock ticker symbol and request predictions.
2. Prediction service: A backend service that processes requests and returns predicted stock prices using api.
3. Data service: Machine Learning algorithm that trains models based on historical stock data.



2.1 Frontend UI



Users can enter the stock ticker symbols in the search box. Suggestions are available while the user is typing to help them find the desired stock. A loading message will be displayed while the prediction data is being fetched from the backend, providing feedback to the user. Once the prediction is finished, it is presented in a formatted manner, showing the current price, predicted price, and percentage change.

2.2 Prediction Service

The api receives the request from the frontend and obtains the stock symbol from the users. Then it will call the corresponding model of that stock to predict the price of that stock. Once the result is ready, it returns the result back to the frontend. If the stock symbol does not exist or any error occurs, it will return an error message.

2.3 Data Service

In order to predict the price of a specific stock, a model is trained based on the data of that stock for the last twenty years. The model is a neural network that consists of multiple convolutional layers to extract the feature, pooling layer to reduce the dimensionality of the feature maps and bidirectional LSTM layers with attention.

1. Convolutional Layers
    - Conv1D Layer 1: Applies 36 filters with a kernel size of 6 and ELU activation function.
    - Conv1D Layer 2: Applies 108 filters with a kernel size of 6 and ELU activation function.
    - Conv1D Layer 3: Applies 256 filters with a kernel size of 6 and ELU activation function.
    - These layers help in extracting features from the input time series data.

2. MaxPooling Layer
    - A MaxPooling1D layer with a pool size of 4 and stride of 1 is used to reduce the dimensionality of the feature maps, which helps in reducing the computational load and capturing the most important features.

3. Bidirectional LSTM Layers with Attention
    - Bidirectional LSTM Layer 1: This layer has 108 units and returns sequences, allowing the model to capture dependencies in both forward and backward directions.
    - Attention Layer: An attention mechanism is applied to the output of the first LSTM layer, helping the model to focus on the most relevant parts of the input sequence.
    - Bidirectional LSTM Layer 2: This layer has 36 units and does not return sequences. It includes L1 regularization to prevent overfitting.

3. Containerization

Our implementation also uses Docker to package the microservice into a container that includes all necessary dependencies and configurations. This approach ensures consistency across different environments as the containerized application behaves identically regardless of where it is deployed. The use of Docker allows for seamless integration with serverless platforms such as AWS Lambda, enabling automatic scaling and efficient resource utilization. Containers provide isolation, preventing conflicts between different services and simplifying the deployment process. This containerized setup enhances the portability and scalability of the microservice.

4. Deployment

The deployment of our microservice architecture on AWS leverages several cloud-native services to ensure high availability, scalability, and performance. We use Amazon EC2 instances to host our Docker containers, providing a flexible and scalable compute environment. By utilizing AWS's robust infrastructure, we can ensure that our application remains highly available and can scale automatically based on demand. This deployment strategy allows us to take full advantage of AWS's cloud-native features, ensuring a resilient and performant stock prediction service.

4.1 Procedure

1. Launch an EC2 instance
    - Instance Selection: Choose an appropriate EC2 instance type to balance performance and cost. Since it is a microservice, t2.medium or even small is capable of handling the workload and save some cost at the same time.
    - Security Groups: Configured security groups to allow inbound traffic on port 5000 for the Flask application and SSH access for management.

2. Environment Configuration

- Docker Installation: Install Docker on the EC2 instance to facilitate running the containerized application. Docker provides a consistent environment for the application, ensuring that it runs the same way in development, testing, and production.

```
sudo apt-get update  # For Ubuntu
sudo apt-get install -y docker.io
```

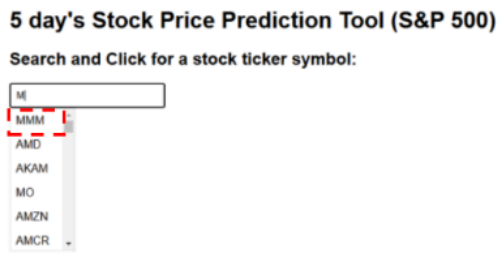
- Code Deployment:
  Clone the project repository onto the EC2 instance.
  Built the Docker image using the provided Dockerfile.

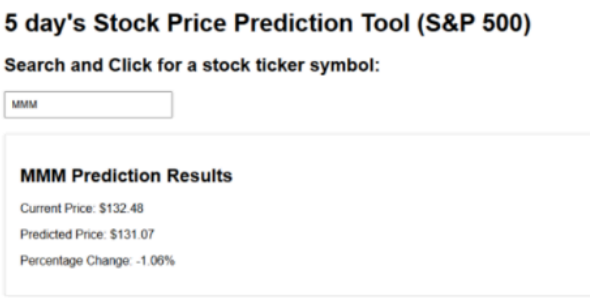
```
git clone
https://github.com/COMP4651-24fall/project-group-17/tree/mai
n
sudo systemctl start docker
sudo systemctl enable docker
docker build -t flask-app
```

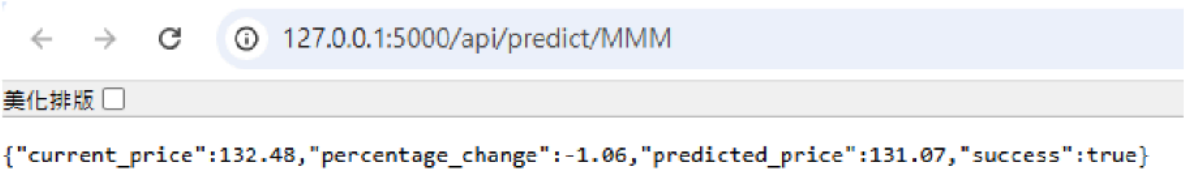
- Running the Container: Execute the Docker container, mapping the container's port 5000 to the host port 5000. Use Docker commands to manage the container lifecycle (docker run, docker stop, etc.).

```
docker run -p 8080:5000 flask-app
docker stop flask-app
```

4.2 Outcome







The above figure shows the successful execution of the service. As described in the previous sections, users can search for the desired stock symbol and the prediction results

of that stock will be returned through the RESTful API if it exists. Users can access predictions via the web interface or directly through the API.

4.3 Discussion

There are several advantages of deploying cloud native architecture on AWS EC2. First of all, it provides a large scalability which allows users to use larger instance types such as t2.large instead of t2.medium or add more instances to provide the services to more clients. Also, the AWS Auto Scaling groups can automatically adjust the number of EC2 instances based on demand.

This brings to the second advantage which is high cost efficiency. As the number of instances and the type of instances can be adjusted, you can only pay for the compute time used and the adjust enough amount of computing power without reserving the peak demand computing power for the whole day. In our case, the service could first launch at a single small instance and increase the size and number of instances if it got viral, fortunately.

Furthermore, the reliability of our services is also boosted up because of the robust infrastructure provided by Amazon. The EC2 provides prompt responses with low latency which powers up our services. We can monitor the optimal CPU and memory usage to decide if scaling is necessary to ensure all the users get qualitative services. The sufficient bandwidth can handle simultaneous users without significant performance degradation.

5. Possible Improvement

Besides all the things we have accomplished and mentioned above, there are also several things we want to experiment but are unable due to the time limitation. Currently, we deployed the service on an EC2 instance because we are more familiar with it. But we also want to try out other serverless technologies such as AWS Lambda that have been introduced. Also, the frontend interface could also be enhanced such that it shows a graph of the historical stock price along with the predicted price projection which is more intuitive to the users.

6. 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 reliability. 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.