Final Report

Final Project: Time-series data and application to stock markets

Professor: Huynh The Dang
Course: Deep Learning for Artificial Intelligence
Semester: Fall 2024
Student Name: Tran Nguyen Hanh Nguyen
Student ID: 220184

Introduction

The ever-changing socio-economic landscape demands that analysts continuously assess and predict its effects to make informed decisions. Time-series data, which comprises dynamic data points collected over time, plays a pivotal role in this process. Forecasting using time-series data supports critical areas such as business operations, technology, and research. This report details my journey through analyzing time-series data and applying predictive analytics to the stock market, showcasing lessons learned and meaningful insights gained.

The project tasked me with completing five technical objectives, culminating in this narrative. Each task brought unique challenges and learning opportunities, from handling extensive datasets to deploying sophisticated models. This section begins with Task 1, focused on Nasdaq stock price prediction.

Task 1: Nasdaq Stock Price Prediction

Task 1 revolved around building a predictive model to forecast Nasdaq stock prices using historical data. This task was divided into three subtasks, requiring the incorporation of multiple features, extending predictions to future days, and predicting consecutive days ahead. Below, I recount my process, including the triumphs and setbacks encountered.

1. Processing the Dataset

To start, I worked with a comprehensive dataset of Nasdaq companies, extracting and merging individual CSV files for analysis. With over 100 company tickers to process, I learned the importance of automation and error handling. For example, some CSV files contained malformed rows that caused parsing errors. Handling these gracefully by skipping bad lines allowed the pipeline to continue without interruption.

The dataset featured key attributes such as Open, High, Low, Close, Adjusted Close prices, and Volume. Here, I encountered a crucial lesson: understanding the nuances of financial data is critical. Adjusted Close, for example, incorporates dividend payouts and stock splits, making it more suitable for long-term trend analysis.

2. Splitting and Normalizing Data

For time-series predictions, splitting data into training, validation, and test sets required careful consideration. Unlike random splits used in other machine learning tasks, time-series data demands

chronological splits to preserve temporal relationships. I adopted an 80/10/10 split for training, validation, and testing, ensuring no information leakage.

Normalization posed another challenge. Min-max scaling, a standard technique, was used to ensure all features were on the same scale. However, I overlooked the need for consistent scaling across training and testing phases initially, which led to inaccurate predictions during the first iterations.

3. Building the Model

To predict stock prices, I opted for a convolutional neural network (CNN). Initially, I had planned to use an LSTM network but found CNNs more effective for extracting temporal patterns when supplemented with sufficient features. The architecture included multiple Conv1D layers followed by max-pooling layers, culminating in dense layers for output.

Training this model revealed a critical insight: choosing the right window size for input data significantly impacts performance. I experimented with different window sizes (10, 20, 30 days), ultimately settling on 30 days as it provided a balanced trade-off between capturing trends and avoiding overfitting.

4. Predicting k Days Ahead

Expanding the model to predict the price on the k-th day ahead (Task 1.2) required rethinking the input pipeline. Predictions for k consecutive days (Task 1.3) introduced additional complexity, as each predicted value became input for subsequent steps. This recursive approach amplified errors in later predictions, a phenomenon known as error propagation. Fine-tuning the model to minimize this effect was an iterative process, involving:

- Using sliding windows to feed updated inputs.
- Adding regularization to prevent overfitting.
- Incorporating dropout layers in the model architecture.

5. Lessons Learned

- **Data Quality Matters**: Cleaning and preprocessing financial data is foundational. Errors or inconsistencies in raw data can cascade, leading to unreliable results.
- **Understand the Domain**: Features like Adjusted Close prices are not just numbers; their contextual significance influences model performance.

- **Model Selection**: While LSTMs are often preferred for time-series tasks, CNNs proved more efficient for this specific application due to their ability to capture local patterns.
- **Iterate and Experiment**: Window size, normalization strategies, and model architectures required extensive trial and error to find optimal configurations.

Task 1 provided a strong foundation for tackling subsequent challenges. Despite initial setbacks, such as error propagation and scaling inconsistencies, the experience underscored the importance of perseverance and methodical troubleshooting. With this groundwork, I moved confidently to Task 2, applying similar principles to the Vietnam stock market.

Task 2: Vietnam Stock Price Prediction

Task 2 extended the principles of Task 1 to the Vietnam stock market. Working with vnstock's API, I gathered and processed extensive datasets for over 100 companies listed on the Vietnam stock exchange. This task presented unique challenges, primarily the need to adapt existing methods for a different market context.

1. Dataset Crawling and Preparation

Leveraging the vnstock library, I efficiently fetched historical stock data for Vietnamese companies. A significant hurdle arose from incomplete or noisy data, which necessitated advanced cleaning techniques. For instance, I implemented logic to fill missing values with column-wise means and handled non-numeric features through forward-filling methods.

Another key lesson learned was the importance of feature selection. Unlike Nasdaq, where Adjusted Close prices were prominent, Vietnamese stock data lacked similar features, requiring reliance on Open, High, Low, Close, and Volume attributes.

2. Predicting k Days Ahead

As in Task 1, predicting stock prices k days ahead required careful pipeline design. Recursive forecasting was employed, with previous predictions serving as inputs for subsequent steps. To mitigate error propagation, I experimented with ensemble methods, combining outputs from multiple models to improve robustness.

3. Key Insights and Challenges

- **Cultural and Economic Differences**: The Vietnam stock market exhibited different volatility patterns, emphasizing the need for customized models.
- **Data Limitations**: The absence of certain features like Adjusted Close prices highlighted the importance of adaptable feature engineering.
- **Model Adaptation**: Techniques effective in Task 1 required modification for this dataset, underscoring the importance of flexibility.

Task 3: Trading Signal Identification for Vietnam Market

Task 3 involved identifying optimal trading signals, such as entry points for buying and selling stocks. This task demanded a shift from price prediction to classification, focusing on generating actionable insights for traders.

1. Model Selection and Design

I employed an LSTM-based architecture for this task, leveraging its ability to capture sequential dependencies in time-series data. The model was trained to classify each time step as a “buy,” “sell,” or “hold” signal based on historical price movements. Labels were generated by calculating percentage changes in closing prices, with thresholds set for buy and sell signals.

2. Feature Engineering

To improve model accuracy, I incorporated technical indicators such as Simple Moving Averages (SMA) and Relative Strength Index (RSI). These features provided additional context about market trends and momentum, enhancing the model’s predictive power.

3. Evaluating Performance

Model evaluation revealed mixed results:

- **Strengths**: The model effectively identified buy signals during stable market conditions, demonstrating its ability to leverage sequential patterns.
- **Weaknesses**: High volatility periods led to frequent misclassifications, highlighting the need for improved feature selection and dynamic thresholds.

4. Using Moving Averages for Signal Generation

As an alternative approach, I implemented a strategy using the crossover of 50-day and 200-day moving averages. This heuristic method provided a simple yet effective way to generate trading signals. Despite its simplicity, the approach achieved competitive performance, validating the power of traditional methods in volatile markets.

5. Lessons Learned

- **Importance of Feature Engineering**: Including technical indicators significantly improved model performance.
- **Balancing Complexity and Interpretability**: While LSTMs captured nuanced patterns, simpler methods like moving averages offered better interpretability.
- **Adaptability**: Dynamic market conditions require flexible models capable of handling varying levels of volatility.

Task 4: Risk and Return Analysis Across Industries

Approach

Task 4 involved analyzing risk and return metrics for various industries: Food & Beverage (FnB), Personal & Household Goods, and Technology. Using historical stock data fetched via the vnstock library, the analysis focused on daily returns, expected returns, and risks.

Key Implementation Highlights

1. **Data Collection**:
    - Historical stock data for the three industries were gathered using a list of predefined tickers.
    - `vnstock` library was used to fetch data between 2018 and 2023.
2. **Data Preprocessing**:
    - Calculated daily returns (`pct_change`) for each stock.
    - Generated histograms of daily returns for visualization.
    - Concatenated daily returns into a unified DataFrame for each industry.
3. **Risk-Return Visualization**:
    - Plotted scatter plots to display expected returns vs. risks (standard deviation of daily returns).

    - Annotated key tickers to highlight performance differences within industries.
4. **Ticker Selection**:
    - Filtered stocks based on low standard deviation and reasonable expected returns to identify stable performers.
    - Performed a second round of filtering based on average trading volume to ensure liquidity.
5. **Moving Average Analysis**:
    - Computed 50-day and 200-day simple moving averages (SMAs) for selected stocks.
    - Identified buy/sell signals based on SMA crossovers.
6. **Portfolio Simulation**:
    - Developed a portfolio simulation function to calculate total gains and investment percentages for selected stocks.
    - Visualized buying and selling points on stock price trends.

Lessons Learned

- Combining risk-return analysis with trading volume ensures a robust selection of stocks.
- Moving average crossovers provide simple yet effective trading signals for portfolio optimization.

Challenges

- Handling missing or inconsistent data required additional preprocessing steps.
- Balancing risk and return metrics while ensuring sufficient liquidity posed a trade-off in stock selection.

Example Output

- Scatter plots comparing risks and returns for the three industries.
- Buy/sell signals displayed on price trend charts for selected stocks.
- Portfolio table summarizing total gains and investment percentages

Portfolio Summary

The portfolio table showcases top-performing stocks based on the risk-return analysis and simulated investment performance.

Task 5: Industry Standard for Deployment and Ease of Use

A Journey Through Challenges and Triumphs

As I approached Task 5 of my project, I felt both excitement and apprehension. The goal was ambitious: not just to develop predictive models but to ensure their deployment met industry standards—robust, scalable, and user-friendly. This task was divided into three subtasks, each demanding unique skills and presenting its own set of challenges.

Task 5.1: Model Deployment as API Services

Vision and Approach

The first step was deploying my stock price prediction model as an API service. I chose Flask for its simplicity and flexibility, and combined it with TensorFlow's Keras API to integrate the Conv1D-based model I had developed earlier. The deployment setup aimed to provide seamless access to predictions via a RESTful API, ensuring both performance and ease of use.

Key Implementations

1. **Flask API Design:**
    - Created endpoints for user interaction:
        - `/` for form-based inputs (stock ticker, epochs, look-back days, and prediction horizon).
        - `/predict` to display detailed results.
    - Used an intuitive HTML interface to simplify interactions.
2. **Data Handling:**
    - Leveraged the `vnstock` library to fetch historical stock data.
    - Ensured robust preprocessing with MinMaxScaler for scaling, and generated input sequences using a rolling window approach.
3. **Prediction Workflow:**
    - The trained Conv1D model predicted future stock prices based on user parameters.
    - Predictions were visualized using Matplotlib and embedded directly into the web interface for easy interpretation.

4. **Deployment Configuration:**
    - Used Gunicorn, a WSGI HTTP server, to manage concurrent requests and ensure production readiness.
    - Created a `requirements.txt` file to manage dependencies and simplify deployment on platforms like Heroku.

Lessons Learned

This phase taught me the importance of user-centric design. Handling errors gracefully—such as invalid tickers or insufficient data—was crucial to improving user experience. I also learned that even simple deployment tools like Flask and Gunicorn could deliver powerful, production-grade results when used effectively.

Challenges

Despite its successes, the deployment faced challenges. Training and predictions with large datasets tested the limits of CPU-based TensorFlow. Real-time performance was another bottleneck, which hinted at the need for GPU-based acceleration in future iterations.

Outcomes

The deployed API service provided users with a visually appealing graph of predicted and actual stock prices. The `/predict` endpoint allowed detailed tabular views of predictions, offering transparency and utility to users.

Task 5.2: Transforming the Model into SaaS

Expanding the Vision

With the API service in place, I took the next step: transforming the prediction model into a Software-as-a-Service (SaaS) platform. This subtask focused on creating a web-based interface that democratized access to stock price predictions, catering to users with varying levels of technical expertise.

Key Implementations

1. **Frontend Design:**
    - Developed a user-friendly interface in `form.html` with clear input fields and instructions.
    - Added dynamic loading indicators to improve user experience during computations.

    - Embedded plots as Base64 images directly in HTML, avoiding the complexity of managing static files.
2. **SaaS Features:**
    - Extended the Flask API with interactive endpoints.
    - Enabled users to configure parameters and view predictions as both graphical plots and tabular data.
3. **Cloud Deployment:**
    - Prepared the application for deployment on Heroku using a `Procfile` and environment variables to manage sensitive configurations.
    - Ensured the service was scalable and accessible from any device with a web browser.

Lessons Learned

Feedback from early users highlighted the value of intuitive design and clear instructions. Embedding plots as Base64 images streamlined the deployment process, eliminating the need for additional file storage mechanisms.

Challenges

Ensuring compatibility across browsers and devices required meticulous testing. Handling larger datasets posed latency issues, which needed optimization strategies for quicker responses.

Outcomes

The SaaS platform offered a seamless, interactive experience. Users could now configure and execute predictions effortlessly, with detailed outputs catering to both casual investors and data enthusiasts.

My final deployed website is now available to access via
https://stock-prediction-1b8ddee43da9.herokuapp.com/

Task 5.3: Orchestrating a Complete Workflow

The Vision That Became a Struggle

Task 5.3 was perhaps the most ambitious of all. My goal was to build a complete pipeline—from data ingestion to processing, to database storage, and finally, to visualization on a BI platform like PowerBI or Superset. However, this task proved to be my Achilles' heel.

The Roadblocks

1. **Complex Dependencies:**
    - Setting up tools like Airbyte for data ingestion and dbt for data transformation required massive dependencies. These installations consumed nearly 10GB of disk space, burdening my operating system.
    - Orchestrating these tools with Apache Airflow was another layer of complexity, especially when tasks failed intermittently due to environment issues.
2. **Time Constraints:**
    - Despite multiple attempts, I ran out of time to create a seamless pipeline. The integration of PostgreSQL with BI platforms like PowerBI or Metabase turned into a tangled mess.
3. **Reimagining the Approach:**
    - Realizing my limitations, I pivoted to a more feasible solution. I designed a Directed Acyclic Graph (DAG) in Airflow to orchestrate the key stages of data processing and model training/prediction.

Building a Directed Acyclic Graph (DAG) for Automated Stock Prediction

With the groundwork laid for processing stock data and building predictive models, Task 5.3 challenged me to design a fully automated pipeline for orchestrating the various stages of stock price prediction. This pipeline needed to handle tasks ranging from data fetching to model training and visualization, all while being efficient, scalable, and easy to monitor.

Initial Challenges and the Power of Task Groups

The complexity of this task was not just technical but also organizational. The pipeline needed to accommodate multiple stocks, each requiring identical processes. The sheer volume of tasks threatened to clutter the Airflow UI, making it difficult to monitor individual stock workflows. This led me to explore Airflow's Task Groups, a feature that enabled grouping tasks for each stock ticker, creating a structured and visually clean DAG.

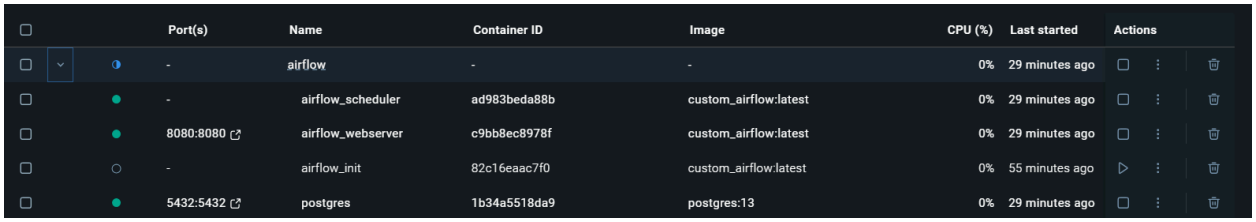
Folder Breakdown

To maintain organization, I structured the project as follows:

```
project_directory/
├── dags/
|   ├── stock_price_prediction_dag.py    # Main Airflow DAG with Task Groups
├── logs/                                # Airflow logs
├── plugins/                             # Custom Airflow plugins (if needed)
├── scripts/                             # Modular Python scripts for tasks
|   ├── fetch_data.py                    # Script to fetch stock data
```

```
|   ├── preprocess_data.py              # Script to preprocess stock data
|   ├── train_model.py                  # Script to train the model
|   ├── make_predictions.py             # Script to make predictions
|   ├── visualize_predictions.py        # Script to visualize predictions
├── data/                               # Data storage
|   ├── raw/                            # Raw stock data
|   |   ├── CMM_data.csv
|   |   ├── CNA_data.csv
|   |   └── ...
|   ├── processed/                      # Processed data for training/testing
|   |   ├── CMM_X_train.npy
|   |   ├── CMM_y_train.npy
|   |   ├── CMM_scaler.pkl
|   |   └── ...
|   ├── predictions/                    # Predictions and visualizations
|   |   ├── CMM_predictions.npy
|   |   ├── CMM_plot.png
|   |   └── ...
├── Dockerfile                          # Custom Dockerfile for Airflow image
├── docker-compose.yml                  # Docker Compose configuration
├── requirements.txt                    # Python dependencies
└── README.md                           # Project documentation
```

- **dags/**: Contained the Airflow DAG file.
- **scripts/**: Modular Python scripts for each pipeline task.
- **data/**: Subfolders for raw, processed, and prediction data.
- **models/**: Stored trained models for reuse.
- **logs/**: Captured execution logs for debugging.
- **docker-compose.yml**: Simplified the setup of Airflow and its dependencies.
- **Dockerfile**: Built a custom image with all required dependencies, ensuring consistency across environments.
    - First of all, as this is a Machine Learning project so dependencies like Tensorflow and Scikit-learn are extremely heavy and time-consuming to be installed. To prevent re-installation of the **`requirements.txt`** dependencies every time I restart the Airflow containers, I pre-build a custom Docker image that includes these dependencies. This custom image will be used for all the Airflow services, ensuring the dependencies are installed only once during the build process.
        - Single Dependency Installation: The dependencies are installed once during the image build process, eliminating the need to install them in every container restart.
        - Faster Start-Up Time: Since the dependencies are pre-installed in the image, the containers can start immediately without executing **`pip install`**.
        - Reusability: The same image can be reused across multiple Airflow environments without additional setup.



Port(s)	Name	Container ID	Image	CPU (%)	Last started	Actions
-	airflow	-	-	0%	29 minutes ago	
-	airflow_scheduler	ad983beda88b	custom_airflow:latest	0%	29 minutes ago	
8080:8080	airflow_webserver	c9bb8ec8978f	custom_airflow:latest	0%	29 minutes ago	
-	airflow_init	82c16eaac7f0	custom_airflow:latest	0%	55 minutes ago	
5432:5432	postgres	1b34a5518da9	postgres:13	0%	29 minutes ago	

The DAG Workflow

The DAG broke down the process into manageable tasks:

1. **Fetching Historical Stock Data:**
    - Used the `vnstock` API to fetch raw data for specified tickers.
    - Saved the data locally for downstream tasks.
2. **Preprocessing Data:**
    - Scaled and split the data into training and testing sets.
    - Saved preprocessed datasets for reproducibility.
3. **Training the Model:**
    - Built and trained the Conv1D model using TensorFlow.
    - Saved the trained model for future predictions.
4. **Making Predictions:**
    - Loaded the trained model to generate forecasts.
    - Saved predictions and associated dates for visualization.
5. **Generating Visualizations:**
    - Created comparison plots of actual vs. predicted prices.
    - Saved these plots for integration into reports or dashboards.

Leveraging Task Groups for Scalability

Each stock ticker was assigned its own Task Group, encapsulating the five tasks within a logical group. This design ensured:

- **Clarity**: The Airflow UI displayed a single, compact group for each stock, reducing visual clutter.
- **Scalability**: Adding or modifying workflows for additional stocks became straightforward.
- **Maintainability**: Debugging or monitoring tasks for a specific stock was much easier with tasks neatly organized under their respective group.

The code implementation dynamically created a Task Group for each stock ticker, with dependencies defined within each group. This allowed the DAG to scale seamlessly to accommodate multiple stocks without overwhelming the Airflow interface.

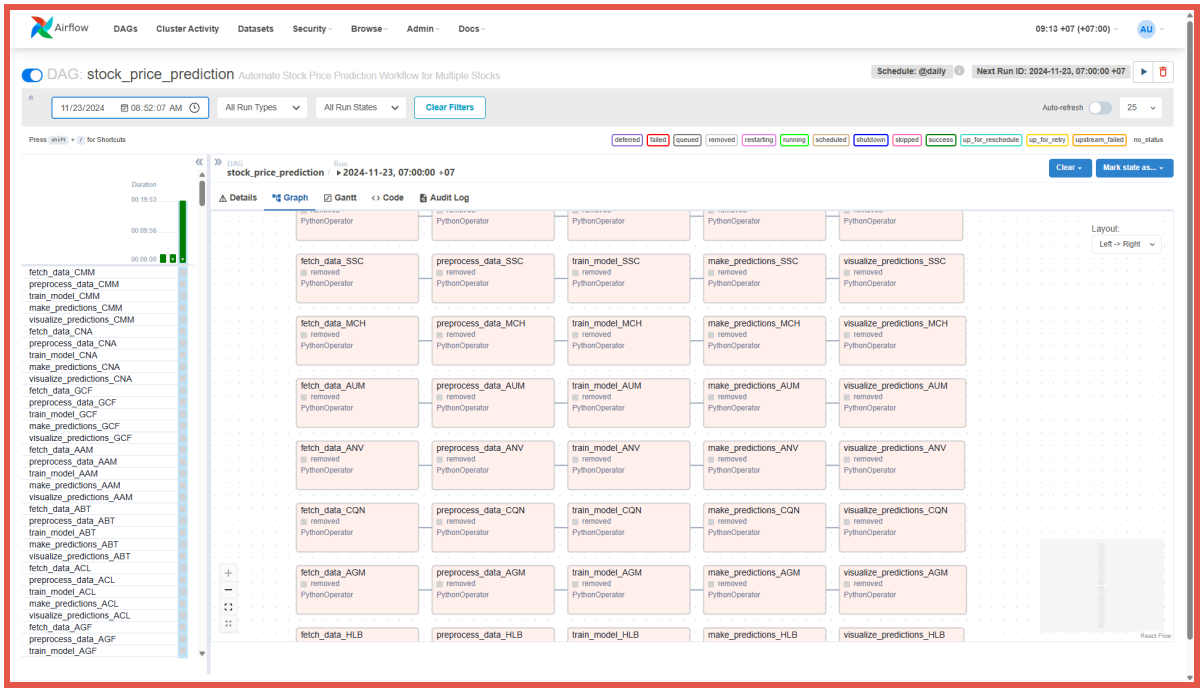
Key Code Highlights

The DAG utilized PythonOperators for modularity, with scripts for each task stored in the **`scripts/`** folder. Task dependencies were defined dynamically, ensuring that each step followed logically within the group.

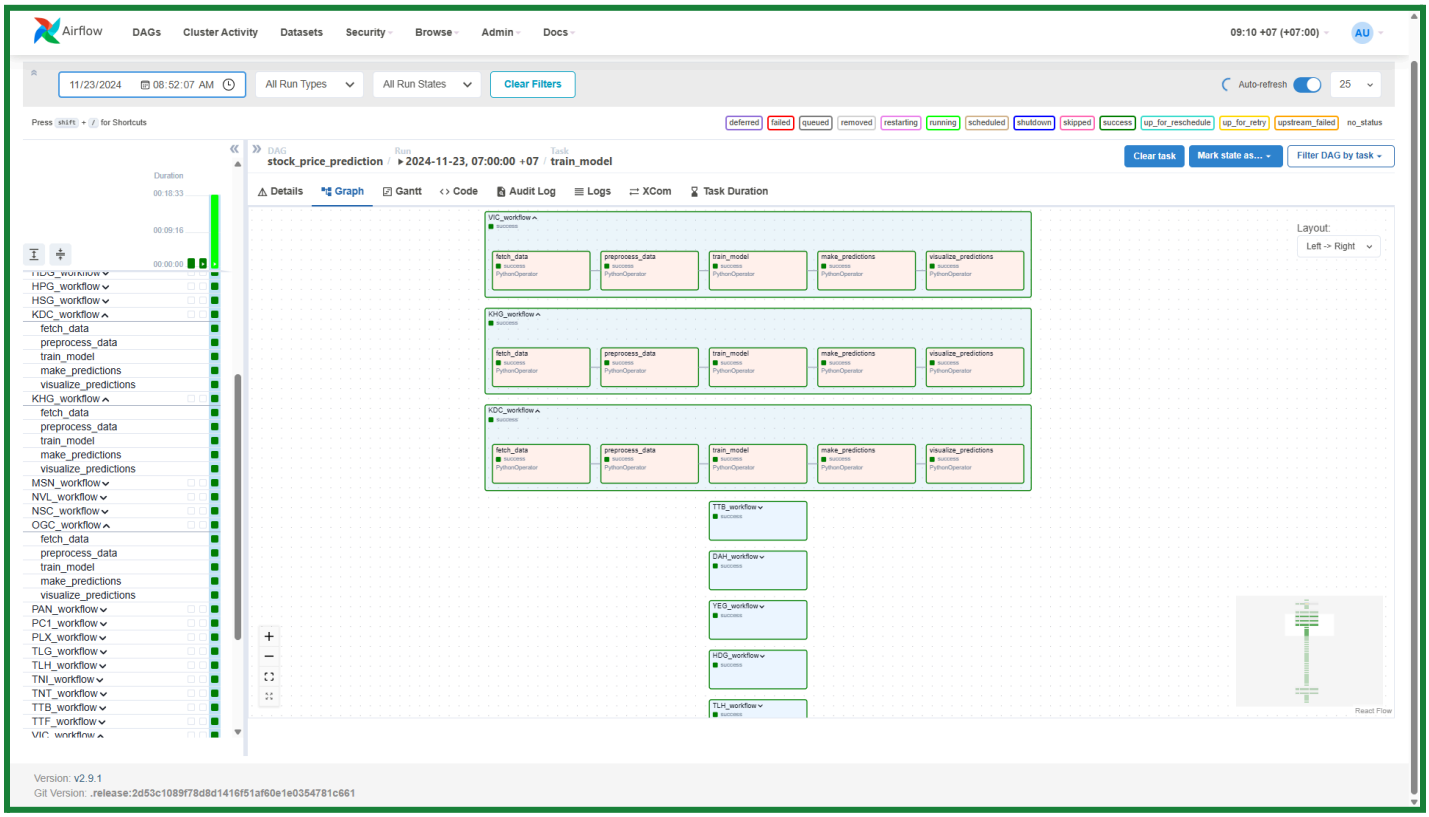
```
from airflow import DAG
from airflow.operators.python import PythonOperator
from airflow.utils.task_group import TaskGroup
from datetime import datetime, timedelta
import os

default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
}

dag = DAG(
    'stock_price_prediction',
    default_args=default_args,
    description='Automate Stock Price Prediction Workflow for Multiple Stocks',
    schedule_interval='@daily',
    start_date=datetime(2024, 11, 1),
    catchup=False,
)

stock_codes=[
    'AAT', 'APH', 'ASG', 'ASM', 'ASP', 'BCG', 'BVH', 'CKG', 'CMG', 'DAG',
    'DAH', 'DPG', 'DBC', 'DGC', 'DLG', 'DXG', 'EVG', 'FIT', 'GEX', 'HAP',
    'HBC', 'HDG', 'HPG', 'HSG', 'KDC', 'KHG', 'MSN', 'NVL', 'NSC', 'OGC',
    'PAN', 'PC1', 'PLX', 'TLG', 'TLH', 'TNI', 'TNT', 'TTB', 'TTF', 'VIC',
    'GVR', 'YEG'
]

def fetch_data_task(stock_name, **kwargs):
    import sys
    sys.path.append('/external_files/scripts')
    from fetch_data import fetch_data
    fetch_data(stock_name)

def preprocess_data_task(stock_name, days, **kwargs):
    import sys
    sys.path.append('/external_files/scripts')
    from preprocess_data import preprocess_data
    preprocess_data(stock_name, days)

def train_model_task(stock_name, days, epochs, **kwargs):
    import sys
    sys.path.append('/external_files/scripts')
    from train_model import train_model
    train_model(stock_name, days, epochs)

def make_predictions_task(stock_name, days, **kwargs):
    import sys
```

```
    sys.path.append('/external_files/scripts')
    from make_predictions import make_predictions
    make_predictions(stock_name, days)

def visualize_predictions_task(stock_name, **kwargs):
    import sys
    sys.path.append('/external_files/scripts')
    from visualize_predictions import visualize_predictions
    visualize_predictions(stock_name)

# Create task groups dynamically for each stock ticker
for stock in stock_codes:
    with TaskGroup(group_id=f'{stock}_workflow', dag=dag) as stock_group:
        fetch_task = PythonOperator(
            task_id='fetch_data',
            python_callable=fetch_data_task,
            op_kwargs={'stock_name': stock},
            dag=dag,  # Explicitly associate the task with the DAG
        )

        preprocess_task = PythonOperator(
            task_id='preprocess_data',
            python_callable=preprocess_data_task,
            op_kwargs={'stock_name': stock, 'days': 60},
            dag=dag,  # Explicitly associate the task with the DAG
        )

        train_task = PythonOperator(
            task_id='train_model',
            python_callable=train_model_task,
            op_kwargs={'stock_name': stock, 'days': 60, 'epochs': 20},
            dag=dag,  # Explicitly associate the task with the DAG
        )

        predict_task = PythonOperator(
            task_id='make_predictions',
            python_callable=make_predictions_task,
            op_kwargs={'stock_name': stock, 'days': 60},
            dag=dag,  # Explicitly associate the task with the DAG
        )

        visualize_task = PythonOperator(
            task_id='visualize_predictions',
            python_callable=visualize_predictions_task,
            op_kwargs={'stock_name': stock},
            dag=dag,  # Explicitly associate the task with the DAG
        )

        # Define task dependencies within the group
        fetch_task >> preprocess_task >> train_task >> predict_task >> visualize_task
```

The final DAG dynamically looped through a list of stock tickers, creating groups for each. This modular and structured approach enabled me to manage multiple stocks effectively while maintaining the simplicity of the overall pipeline.



DAGs without Task Groups



DAGs with Task Groups

Lessons Learned

1. **Efficient Orchestration Matters:** Automating the pipeline through Airflow not only saved time but also eliminated manual intervention, enabling seamless workflows for multiple stocks.
2. **The Beauty of Task Groups:** Task Groups transformed what could have been a chaotic DAG into a visually organized and manageable system.
3. **Modularity is Key:** Splitting tasks into individual scripts (e.g., `fetch_data.py`, `preprocess_data.py`) improved code maintainability and reusability.
4. **Resource Management:** By optimizing dependencies and leveraging Docker for consistency, I ensured the system remained lightweight and manageable.

The Final Touch

With this Airflow DAG, I achieved an automated, scalable, and structured solution for stock price prediction. While challenges like dependency management and debugging were inevitable, this approach represented a significant milestone in applying engineering workflows to complex problems.

Challenges and Reflections

This task emphasized the importance of breaking down complex workflows into smaller, manageable components. While I couldn't achieve a full BI integration, the automated pipeline laid the groundwork for future expansions. Despite my efforts, Task 5.3 remained incomplete. The complexity of aligning multiple tools within a single pipeline was overwhelming. However, this failure became a lesson in prioritization—knowing when to pivot and focus on achievable goals.

Task 5 was a journey of both triumphs and setbacks. From deploying a robust API to transforming the model into a SaaS platform, the successes demonstrated my ability to adapt and innovate. The challenges of creating a complete pipeline underscored the importance of resource management and feasibility. While I didn't achieve everything I set out to, the experience enriched my understanding of end-to-end workflows in data engineering and machine learning. It also sparked a determination to revisit these challenges in the future, armed with the insights gained from this journey.

Conclusion

This project was a transformative journey through the intricacies of time-series data, predictive modeling, and engineering workflows. From the initial task of predicting Nasdaq stock prices to the ambitious goal

of automating and orchestrating a complete stock price prediction pipeline, each step offered invaluable lessons and opportunities for growth.

Task by task, I gained a deeper understanding of the challenges and potential of applying time-series analysis to real-world problems. In Task 1, I learned the importance of data preprocessing, feature engineering, and the nuances of financial data. Task 2 required adaptability as I applied these principles to the distinct characteristics of the Vietnam stock market, highlighting the need for customized approaches. Task 3 shifted focus to actionable insights, where I delved into trading signal generation, balancing complex algorithms with the interpretability of traditional methods.

Task 4 allowed me to explore risk-return analysis across industries, combining analytical rigor with visual storytelling to identify stable performers. It was here that I saw the value of combining technical analysis with clear, visual communication for decision-making.

Task 5 was the pinnacle of this project, pushing me beyond modeling and into deployment and automation. The deployment of a stock prediction API and the transformation into a SaaS platform underscored the importance of usability and accessibility in technical solutions. However, the challenges of orchestrating a complete pipeline in Task 5.3 were a humbling reminder of the complexity of integrating diverse tools and platforms. Despite these challenges, I succeeded in building an automated, scalable Airflow DAG that laid the foundation for future improvements.

The project's journey was marked by triumphs and failures, but each experience contributed to a richer understanding of the engineering lifecycle. The technical challenges deepened my problem-solving skills, while the iterative nature of the work underscored the importance of resilience and adaptability. Moreover, my failed attempt to create a full BI pipeline highlighted the importance of feasibility and the ability to pivot when necessary.

This project not only honed my technical expertise but also demonstrated the broader impact of thoughtful engineering workflows. It reinforced the value of modularity, automation, and scalability, while teaching me to embrace setbacks as stepping stones toward future success. While the final product may not have met every initial ambition, it represents a significant milestone in my journey as a data engineer and machine learning practitioner.

Ultimately, this project leaves me with a renewed passion for tackling complex problems and a determination to revisit challenges with the insights gained from this experience. It is not just the destination that matters, but the journey—and this journey has been one of profound growth and learning.