ANSHUL VATS (104491909)

TASK-B.6 - REPORT

INTRODUCTION

This part of the project of stock prediction using two distinct models – ARIMA (AutoRegressive Integrated Moving Average) and LSTM (Long-Short Term M emory). Both models have their strengths and weaknesses in time series prediction, making them suitable for different scenarios.

MODEL IMPLEMENTATIONS

ARIMA MODEL

The ARIMA MODEL WAS CONFIGURED WITH THE ORDER (p,d,q) set to (1,1,1)

- **P**: The number of lag observations included in the model (Auto-Regressive part).
- **D**: The number of times that the raw observations are differenced (Integrated part).
- **Q**: The size of the moving average window (Moving Average part).

The ARIMA model was fitted using the training data, and predictions were generated for the test dataset.

LSTM Model

The LSTM model was built using keras with various configurations . Parameters were-

Units

Layers

Dropout

Bidirectional

Units means the number of neurons in each LSTM layer. While, layers defines the number of lstm layers in the model. Apart from this ,dropout and bidirectional represents a fraction of the neurons that will be dropped during training to prevent overfitting and, whether to use a bidirectional LSTM archietecture, respectively.

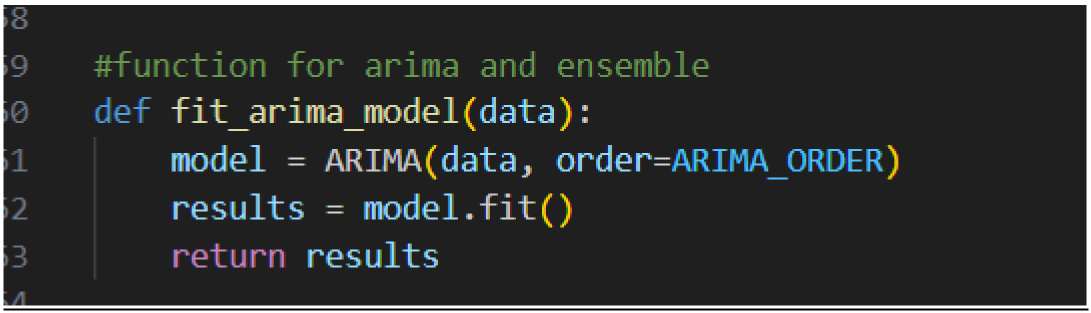
ENSEMBLE MODEL

An ensemble approach was utilized by combining the predictions of both the ARIMA and LSTM models. The predictions were weighted according to predefined weights:

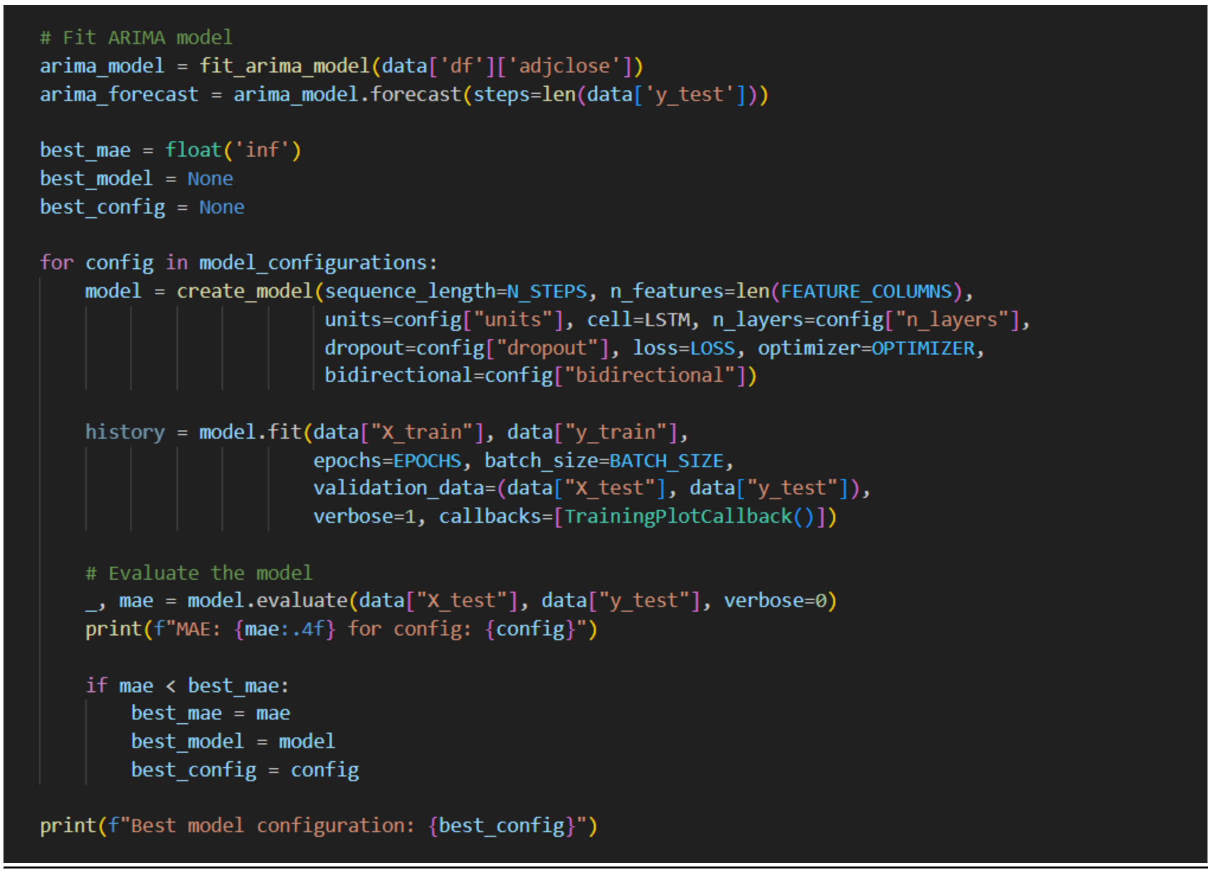
- **ARIMA Weight**: 0.3
- **LSTM Weight**: 0.7

This technique aims to leverage the strengths of both models to improve prediction accuracy.

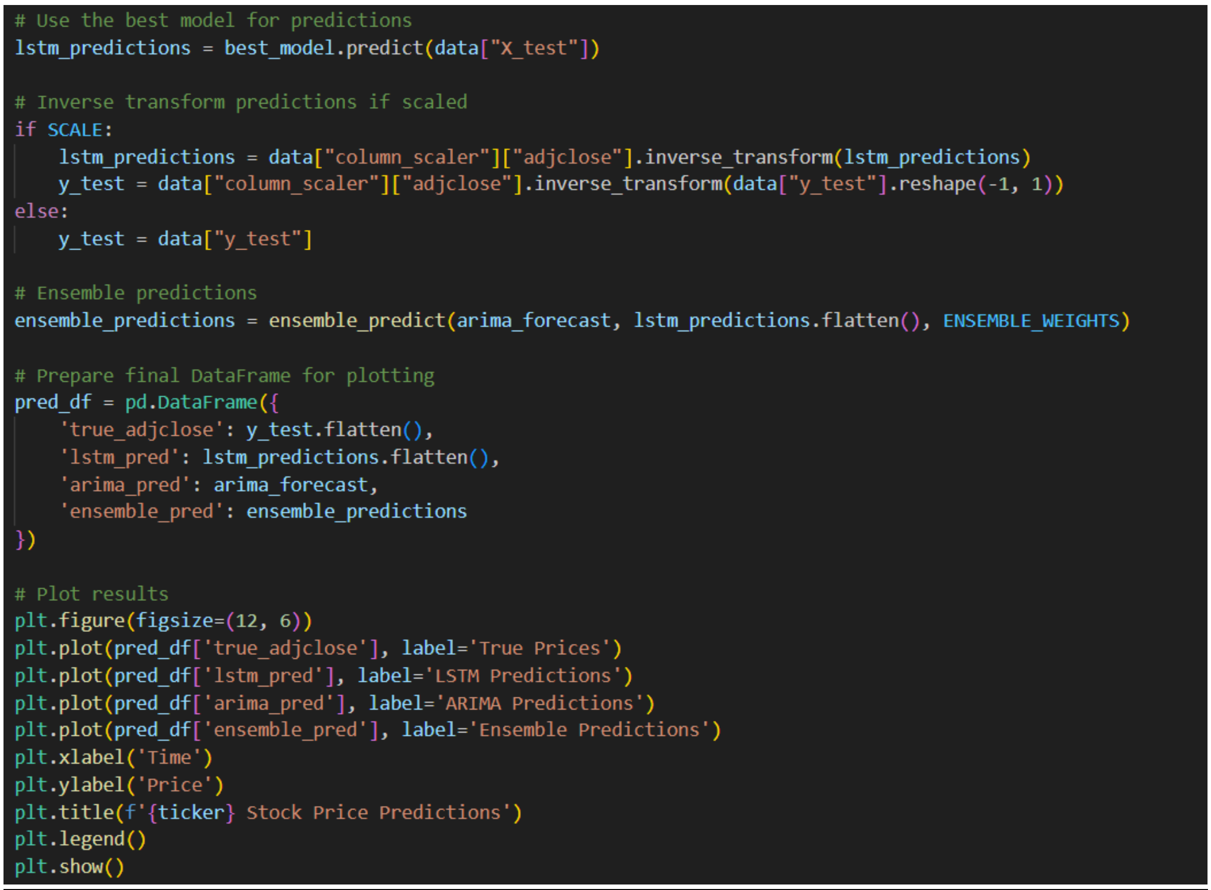
CODES EXPLANATION



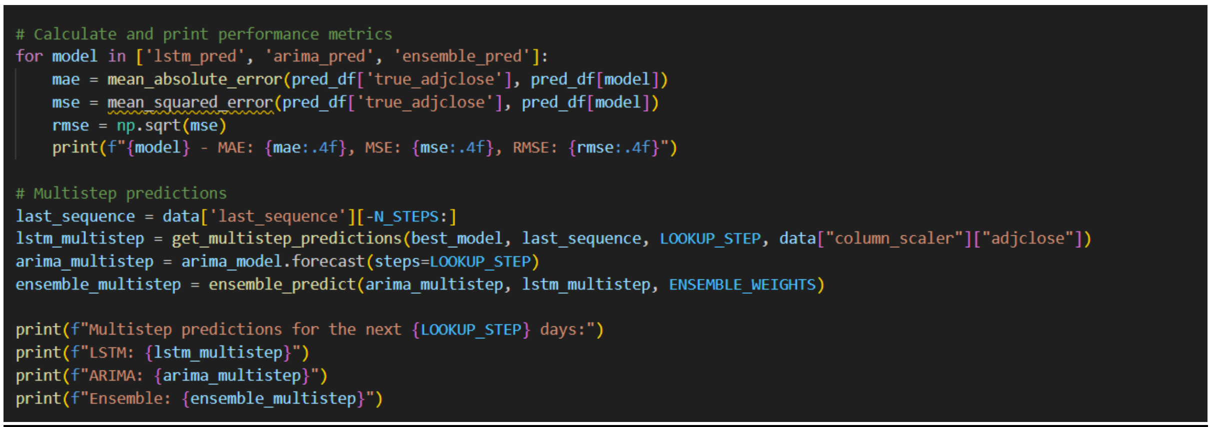
```python
#function for arima and ensemble
def fit_arima_model(data):
    model = ARIMA(data, order=ARIMA_ORDER)
    results = model.fit()
    return results
```



```python
# Fit ARIMA model
arima_model = fit_arima_model(data['df']['adjclose'])
arima_forecast = arima_model.forecast(steps=len(data['y_test']))

best_mae = float('inf')
best_model = None
best_config = None

for config in model_configurations:
    model = create_model(sequence_length=N_STEPS, n_features=len(FEATURE_COLUMNS),
                         units=config["units"], cell=LSTM, n_layers=config["n_layers"],
                         dropout=config["dropout"], loss=LOSS, optimizer=OPTIMIZER,
                         bidirectional=config["bidirectional"])

    history = model.fit(data["X_train"], data["y_train"],
                        epochs=EPOCHS, batch_size=BATCH_SIZE,
                        validation_data=(data["X_test"], data["y_test"]),
                        verbose=1, callbacks=[TrainingPlotCallback()])

    # Evaluate the model
    _, mae = model.evaluate(data["X_test"], data["y_test"], verbose=0)
    print(f"MAE: {mae:.4f} for config: {config}")

    if mae < best_mae:
        best_mae = mae
        best_model = model
        best_config = config

print(f"Best model configuration: {best_config}")
```



```python
# Use the best model for predictions
lstm_predictions = best_model.predict(data["X_test"])

# Inverse transform predictions if scaled
if SCALE:
    lstm_predictions = data["column_scaler"]["adjclose"].inverse_transform(lstm_predictions)
    y_test = data["column_scaler"]["adjclose"].inverse_transform(data["y_test"].reshape(-1, 1))
else:
    y_test = data["y_test"]

# Ensemble predictions
ensemble_predictions = ensemble_predict(arima_forecast, lstm_predictions.flatten(), ENSEMBLE_WEIGHTS)

# Prepare final DataFrame for plotting
pred_df = pd.DataFrame({
    'true_adjclose': y_test.flatten(),
    'lstm_pred': lstm_predictions.flatten(),
    'arima_pred': arima_forecast,
    'ensemble_pred': ensemble_predictions
})

# Plot results
plt.figure(figsize=(12, 6))
plt.plot(pred_df['true_adjclose'], label='True Prices')
plt.plot(pred_df['lstm_pred'], label='LSTM Predictions')
plt.plot(pred_df['arima_pred'], label='ARIMA Predictions')
plt.plot(pred_df['ensemble_pred'], label='Ensemble Predictions')
plt.xlabel('Time')
plt.ylabel('Price')
plt.title(f'{ticker} Stock Price Predictions')
plt.legend()
plt.show()
```



```python
# Calculate and print performance metrics
for model in ['lstm_pred', 'arima_pred', 'ensemble_pred']:
    mae = mean_absolute_error(pred_df['true_adjclose'], pred_df[model])
    mse = mean_squared_error(pred_df['true_adjclose'], pred_df[model])
    rmse = np.sqrt(mse)
    print(f"{model} - MAE: {mae:.4f}, MSE: {mse:.4f}, RMSE: {rmse:.4f}")

# Multistep predictions
last_sequence = data['last_sequence'][-N_STEPS:]
lstm_multistep = get_multistep_predictions(best_model, last_sequence, LOOKUP_STEP, data["column_scaler"]["adjclose"])
arima_multistep = arima_model.forecast(steps=LOOKUP_STEP)
ensemble_multistep = ensemble_predict(arima_multistep, lstm_multistep, ENSEMBLE_WEIGHTS)

print(f"Multistep predictions for the next {LOOKUP_STEP} days:")
print(f"LSTM: {lstm_multistep}")
print(f"ARIMA: {arima_multistep}")
print(f"Ensemble: {ensemble_multistep}")
```

<u>ARIMA Model Fitting:</u>

- fit_arima_model(data['df']['adjclose']):
    - This function call is expected to train an ARIMA (AutoRegressive Integrated Moving Average) model using the 'adjclose' column from the data DataFrame, which contains the adjusted closing prices of the stock.
    - The function likely preprocesses the data (e.g., checking for stationarity, determining ARIMA parameters (p, d, q)), fits the ARIMA model to the time series data, and returns the fitted model object.
- Purpose: The ARIMA model is designed to capture temporal dependencies in time series data, making it suitable for forecasting stock prices based on historical trends.

Forecasting with ARIMA:

- arima_forecast = arima_model.forecast(steps=len(data['y_test'])):
    - This line generates future predictions for the stock prices using the fitted ARIMA model. The forecast method computes predictions for the number of steps equal to the length of the test set y_test.
    - The result, stored in arima_forecast, contains the predicted values for the specified future time periods.

nitialization of Best Model Variables:

- **best_mae = float('inf')**: This initializes the variable best_mae to infinity, ensuring that any Mean Absolute Error (MAE) calculated during model evaluation will be lower than this initial value.
- **best_model = None and best_config = None**: These variables are set to None and will later store the best-performing model and its corresponding configuration.

Looping Through Model Configurations:

- **for config in model_configurations:**: This loop iterates through a list of different configurations for the LSTM model, which might include varying the number of units, layers, dropout rates, etc. Each configuration is a dictionary with parameters that define the structure and behavior of the model.

Model Creation:

- **create_model(...)**:
    - This function is called with several parameters:
        - **sequence_length=N_STEPS**: Specifies the length of the input sequences fed into the LSTM.
        - **n_features=len(FEATURE_COLUMNS)**: The number of features in the input data, determining the model's input shape.
        - **units=config["units"]**: Defines the number of units (neurons) in the LSTM layers based on the current configuration.
        - **cell=LSTM**: Specifies that the cell type used in the model is LSTM (Long Short-Term Memory).
        - **n_layers=config["n_layers"]**: The number of layers in the LSTM model.
        - **dropout=config["dropout"]**: The dropout rate used for regularization to prevent overfitting.
        - **loss=LOSS**: The loss function used to train the model, which could be Mean Squared Error (MSE) or another relevant metric.

        - **optimizer=OPTIMIZER**: The optimization algorithm used for training (e.g., Adam, RMSprop).
        - **bidirectional=config["bidirectional"]**: Indicates whether the LSTM layers are bidirectional, allowing the model to learn from both past and future contexts.

Model Training:

- **history = model.fit(...)**:
    - This line trains the model using the training data (data["X_train"] and data["y_train"]) for a specified number of epochs and batch size.
    - **validation_data=(data["X_test"], data["y_test"])**: This argument is used to validate the model on the test set during training, allowing for monitoring of the model's performance on unseen data.
    - **verbose=1**: This setting enables detailed output during training, showing progress and metrics.
    - **callbacks=[TrainingPlotCallback()]**: Custom callbacks, like plotting training metrics, can be utilized to visualize the training process in real-time.

Model Evaluation:

- **_, mae = model.evaluate(data["X_test"], data["y_test"], verbose=0)**:
    - After training, the model is evaluated on the test set to compute the Mean Absolute Error (MAE), a metric indicating the average absolute difference between the predicted and actual stock prices.
    - The underscore _ is used to ignore the first return value (usually loss), focusing on MAE for this context.

Tracking the Best Model:

- **if mae < best_mae:**: This conditional checks if the current MAE is lower than the previously recorded best MAE.
- If true, it updates:
    - **best_mae = mae**: Sets the new best MAE.
    - **best_model = model**: Stores the current model as the best-performing one.
    - **best_config = config**: Saves the configuration of the best model for future reference or analysis.