Report Task 5

# Multistep problem

# multstep predict function

def create\_sequences\_ms(data, seq\_length, n\_steps\_ahead):

xs, ys = [], []

for i in range(len(data)-(seq\_length+n\_steps\_ahead)+1):

x = data[i:(i+seq\_length)]

y = data[(i+seq\_length):(i+seq\_length+n\_steps\_ahead)]

xs.append(x)

ys.append(y)

return np.array(xs), np.array(ys)

The `create\_sequences\_ms` function addresses the challenge of multistep prediction. It facilitates forecasting a series of closing prices for a specified number of days ahead, defined by `n\_steps\_ahead`. This function requires three inputs: `data`, `seq\_length`, and `n\_steps\_ahead`. It generates two arrays, `xs` and `ys`, which contain the input and output sequences necessary for training a machine learning model. The function processes the data by iterating through it and appending subsequences of length `seq\_length` to the `xs` array. Subsequently, for the `ys` array, it extracts a subsequence of length `n\_steps\_ahead` from the data, beginning right after the end of the corresponding `x` subsequence. This extracted subsequence signifies the target closing prices that the model aims to predict.

# Predict the next n days

real\_data = [data['X\_test'][-1, :]]

real\_data = np.array(real\_data)

real\_data = np.reshape(real\_data, (real\_data.shape[0], real\_data.shape[1], n\_features))

prediction = model.predict(real\_data)

prediction=data["column\_scaler"][prediction\_column].inverse\_transform(prediction[:, :, closing\_price\_index])

# Loop over the prediction and each day predict price

for i, price in enumerate(prediction[0]):

    print(f"Prediction for day {i+1}: {price}")

I have also modified the code to effectively generate stock price predictions for the upcoming k days. The process begins with the preparation of input data, referred to as `real\_data`, which utilizes the most recent information from the test set. The dimensions of `real\_data` are altered to align with the model's required input format. Subsequently, the model forecasts stock prices for the next k days based on this prepared input. The predicted values are then converted back to their original scale using a column scaler. To conclude, a loop is employed to display each day's forecasted price.

# Multivariate problem

def create\_sequences\_mv(data, seq\_length):

    xs, ys = [], []

    for i in range(len(data) - (seq\_length + 1)):

        x = data[i:(i + seq\_length)]

        y = data[i + seq\_length]

        xs.append(x)

        ys.append(y)

    return np.array(xs), np.array(ys)

My function outlined above addresses the straightforward challenge of multivariate prediction. It generates sequences by taking into account various attributes of the same stock, such as the opening price, highest price, lowest price, closing price, adjusted closing price, and trading volume. These inputs are utilized to forecast the closing price for a specific future date. A critical aspect of this process involves supplying all relevant data features, not just the Closing column. Additionally, model creation must be tailored accordingly; this includes adjusting for feature count and restructuring the model output to incorporate this count effectively.

closing\_price\_index = FEATURE\_COLUMNS.index(prediction\_column)

# Get the actual prices

actual\_prices =data["column\_scaler"][prediction\_column].inverse\_transform(data["y\_test"][:, -1, closing\_price\_index].reshape(-1,1)).ravel()

# Predict the prices

predicted\_prices = model.predict(data['X\_test'])

predicted\_close\_prices = predicted\_prices[:, -1, closing\_price\_index].reshape(-1, 1)

predicted\_close\_prices =data["column\_scaler"][prediction\_column].inverse\_transform(predicted\_close\_prices).ravel()

As demonstrated, the model's predictions now generate data that incorporates the feature count. Therefore, it is essential to identify the index of the desired column—specifically, 'Close' in this instance—and utilize this index to extract only the relevant data from the prediction output.

# Multivariate problem + Multistep problem(Combine)

Integrating the two systems is straightforward; it merely requires incorporating the essential code for each, as outlined previously. The only aspect of the code that must be modified to accommodate both systems pertains to model creation, specifically where adjustments are needed for the dense layer and output shape.

model.add(TimeDistributed(Dense(n\_features)))

# Select the last n\_outputdays time steps from the sequence

model.add(Lambda(lambda x: x[:, -n\_outputdays:, :]))

model.compile(optimizer=optimizer, loss=loss)

return model

As depicted in this example, the inclusion of the dense layer necessitates the use of Time Distributed to extend its application across each time step. This approach supports both multi-step and multivariate considerations. Subsequently, it is essential to incorporate the Lambda layer to guarantee that the model's output conforms to the appropriate dimensions.

# Result

Firstly, for the multi-step, we can the prediction successfully displays for the specified number of days in the future.



A screenshot of a computer screen

Description automatically generated

Moving on to the multi-variate analysis, the output on the left corresponds to my version 0.4 code, while the output on the right is generated from my latest code. Both results lack high accuracy. This suggests that a more intricate model might be necessary, which would require additional training time to yield meaningful outcomes. Nonetheless, the current code meets the requirements outlined for this task.

