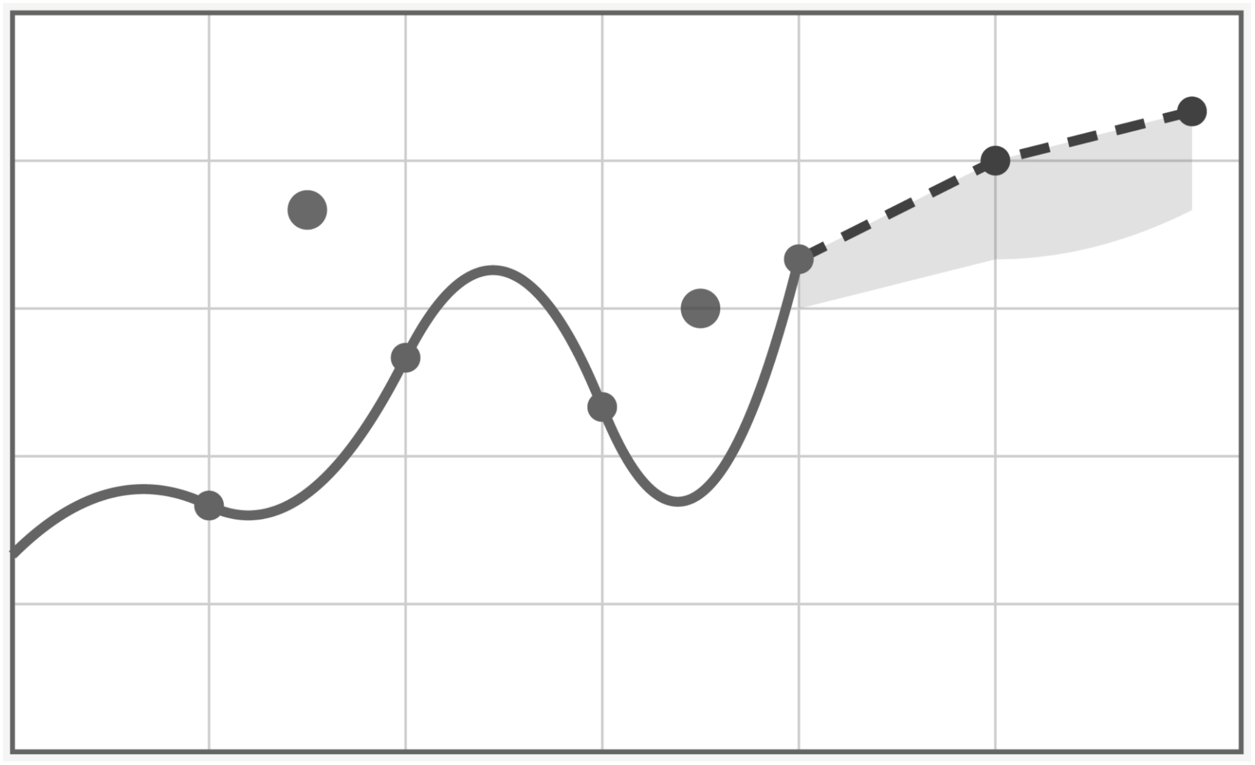
Exercises 5 (Time Series Prediction)

DATA.ML.450 Time Series Analysis using Machine Learning (Autumn 2025)



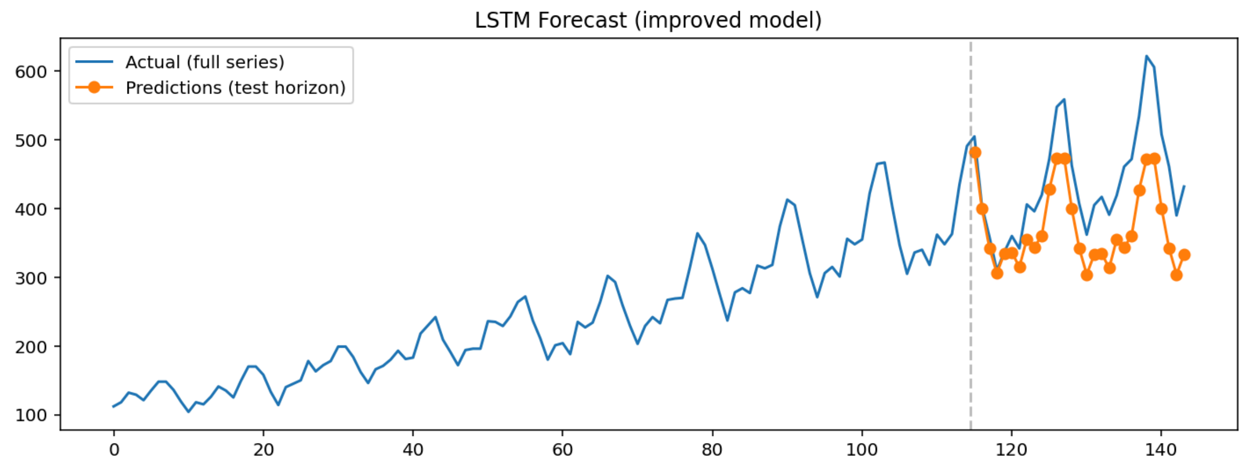
Anas Uddin

08.10.2025

MSc in Data Science

Exercise 1

I implemented StackAbuse's LSTM time series model. Fixed the hidden state initialization issue stated in the problem. I tried improving the architecture by adding a preprocessing linear layer before the LSTM and stacking multiple LSTM layers with dropout. I used an 80/20 train/test split to ensure proper evaluation. I can see that the improved model achieved smoother and more accurate predictions compared to the original. This shows that the architectural enhancements and proper handling of hidden states contributed to better learning and generalization.



Exercise 2

1) What does the multivariate data represent in input, and what does the LSTM network try to predict?

In the Charlie O'Neill "multivariate LSTM" example, the input at each time step is a vector of multiple features or covariates, for example, Open price, High, Low, and Volume for Bitcoin.

- Each time step has, for example, 4 values: Open, High, Low, Volume.
- The LSTM is fed a sequence of these multivariate vectors over some look-back window, for example, the past 100 days.
- Its task is to predict a target variable, typically the closing price over some future horizon, for example, the next 50 days.

2) Does this dataset have any internal correlation?

Yes, the dataset contains internal correlation:

- The features are correlated within each time step. For example, High, Low, Open, and Close prices are obviously related in financial data. Volume also often correlates with price movements.
- There is temporal correlation/autocorrelation across time: past prices influence future prices.
- Additionally, cross-correlations: features at past time steps can help predict the target; for example, price trends and volatility often precede price changes.

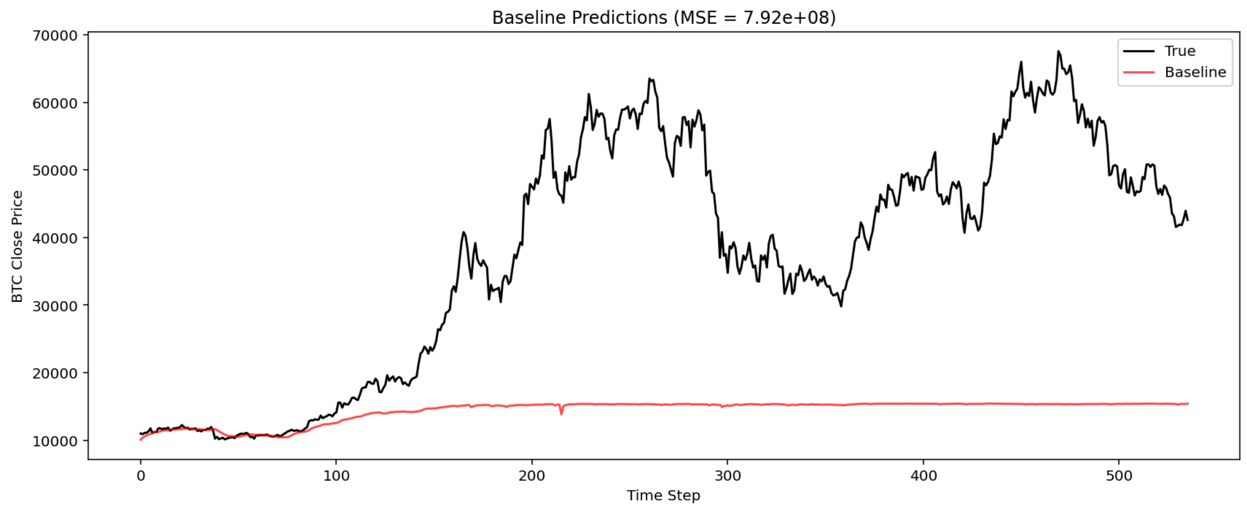
Output from the analysis:

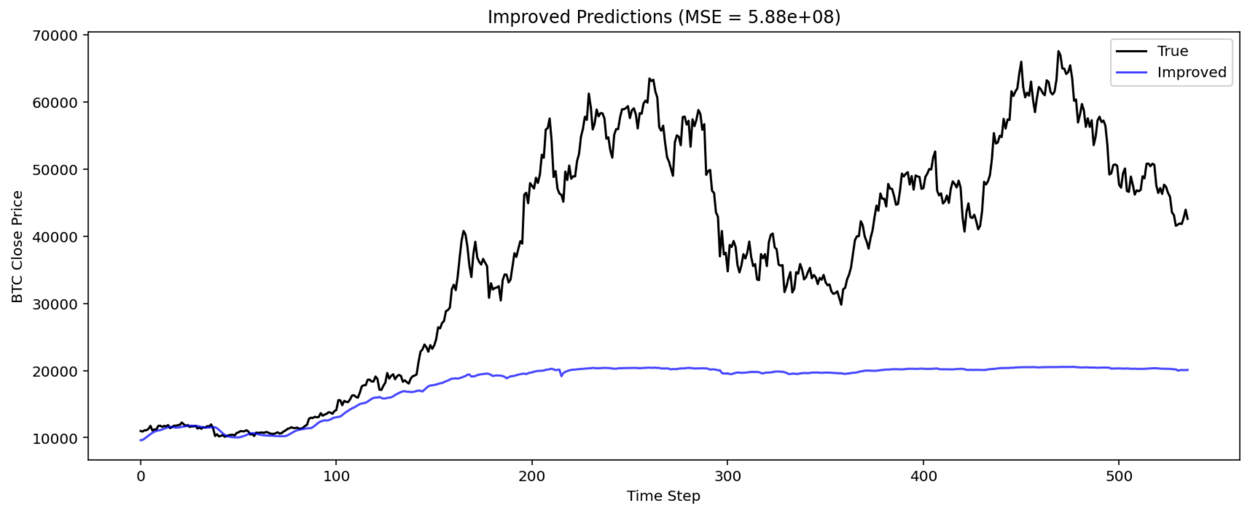
According to the Charlie O'Neill example, I implemented the multivariate LSTM model using Bitcoin OHLCV data from Yahoo Finance.

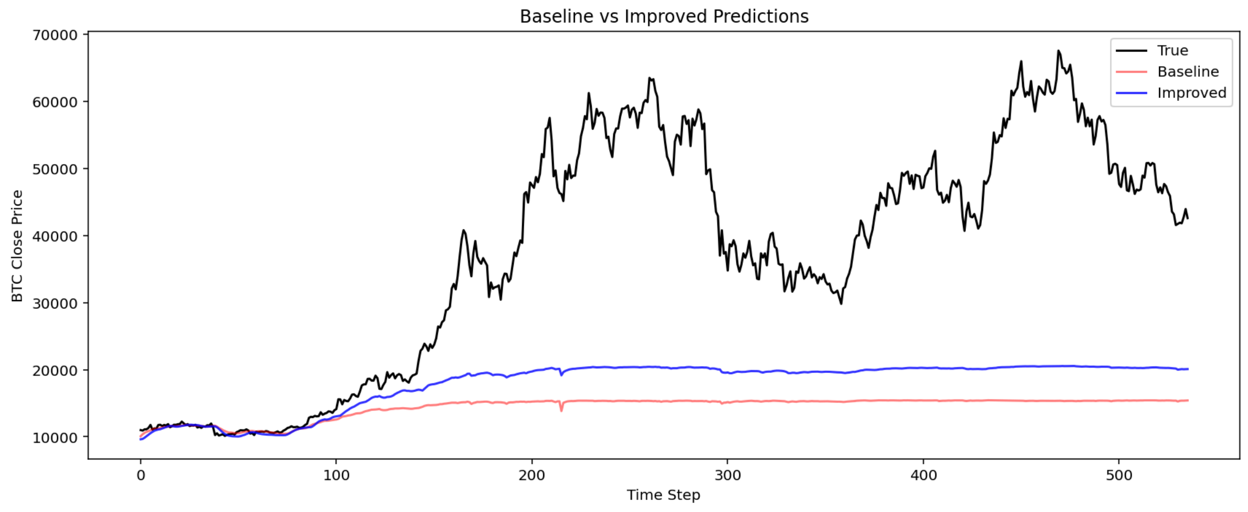
The input consisted of multiple correlated features (Open, High, Low, Volume, and Close), and the LSTM predicted the next-day closing price.

The dataset showed strong internal correlation among its price and volume variables, which the model leveraged for learning temporal relationships.

The baseline model achieved an MSE of approximately 7.92×10^8, while the improved model reduced the MSE to approximately 5.88×10^8, which shows a clear improvement in predictive accuracy.







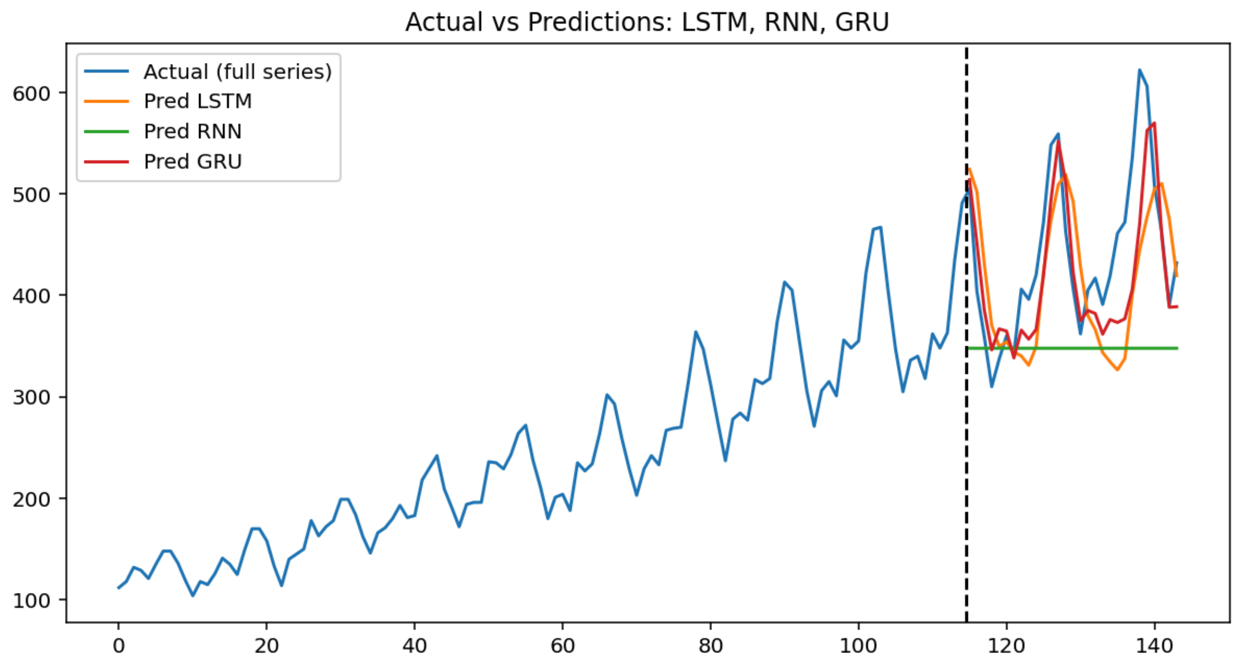
Exercise 3

I implemented the StackAbuse time series prediction model using three variants of the same architecture: LSTM, RNN, and GRU. I trained and tested on the same flight passenger dataset with an 80/20 split of data. The summed absolute errors on the test set were:

- LSTM: 1919.84
- RNN: 2791.13
- GRU: 1228.25

All three predictions followed similar seasonal patterns, but the GRU produced smoother and more accurate forecasts. The GRU performed best with the lowest total test error, then the LSTM, while the vanilla RNN showed the weakest predictive capability.

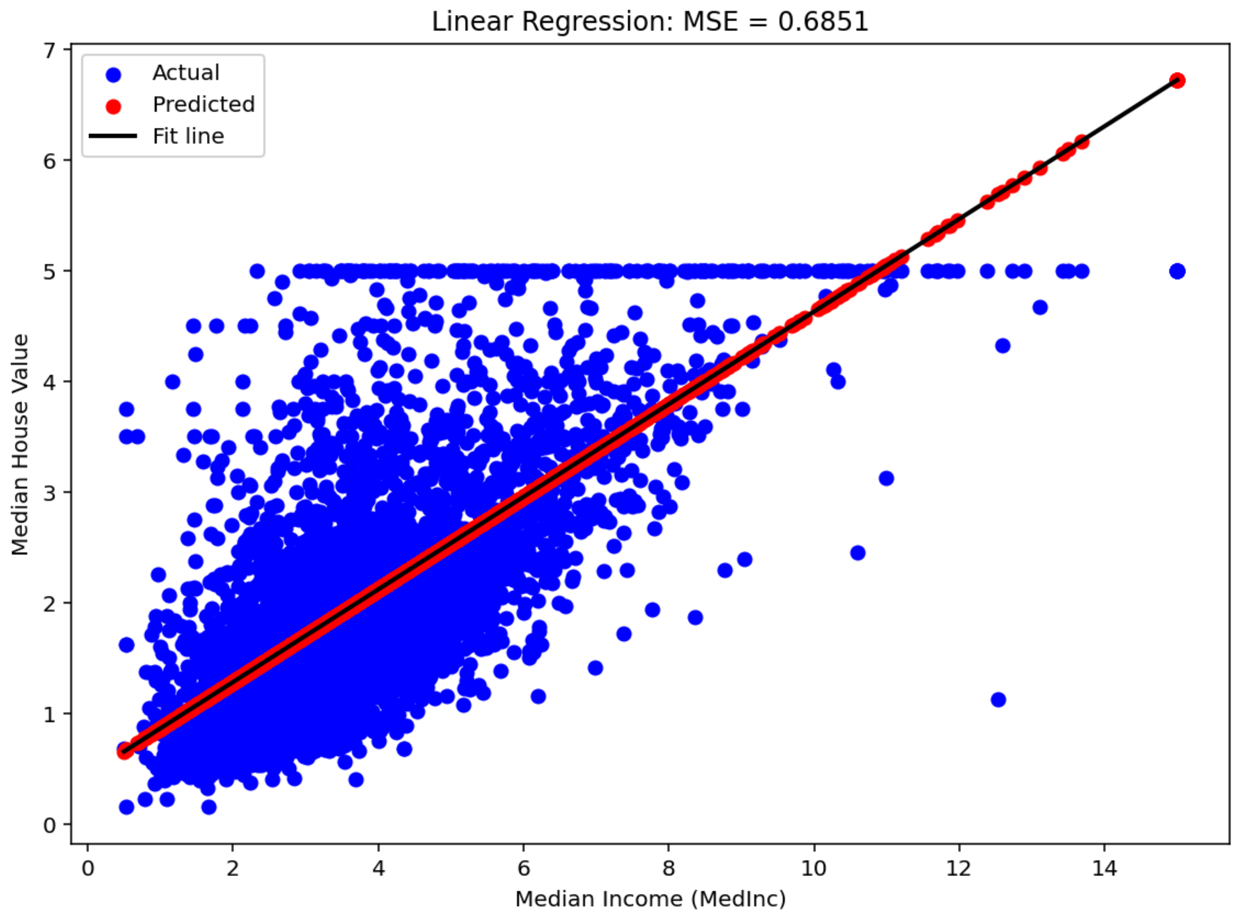
Below, the attached figures of all three predictions confirm that GRU tracks the actual series most closely.



Exercise 4a

I implemented Statology's linear regression example using the California housing dataset in order to predict median house value based on median income (MedInc). The data was split into an 80/20 train/test ratio. A linear regression model was fitted and evaluated on the test dataset.

The model achieved an MSE = 0.6851, which shows a reasonably good linear fit regarding the simplicity of using a single predictor (MedInc) as the Statology example shows. The plotted figure shows a clear upward trend where higher median income corresponds to higher median house value. It is consistent with the expected linear relationship. Although the model effectively captured the main pattern but some variance remains unexplained because of non-linear and multivariate factors.



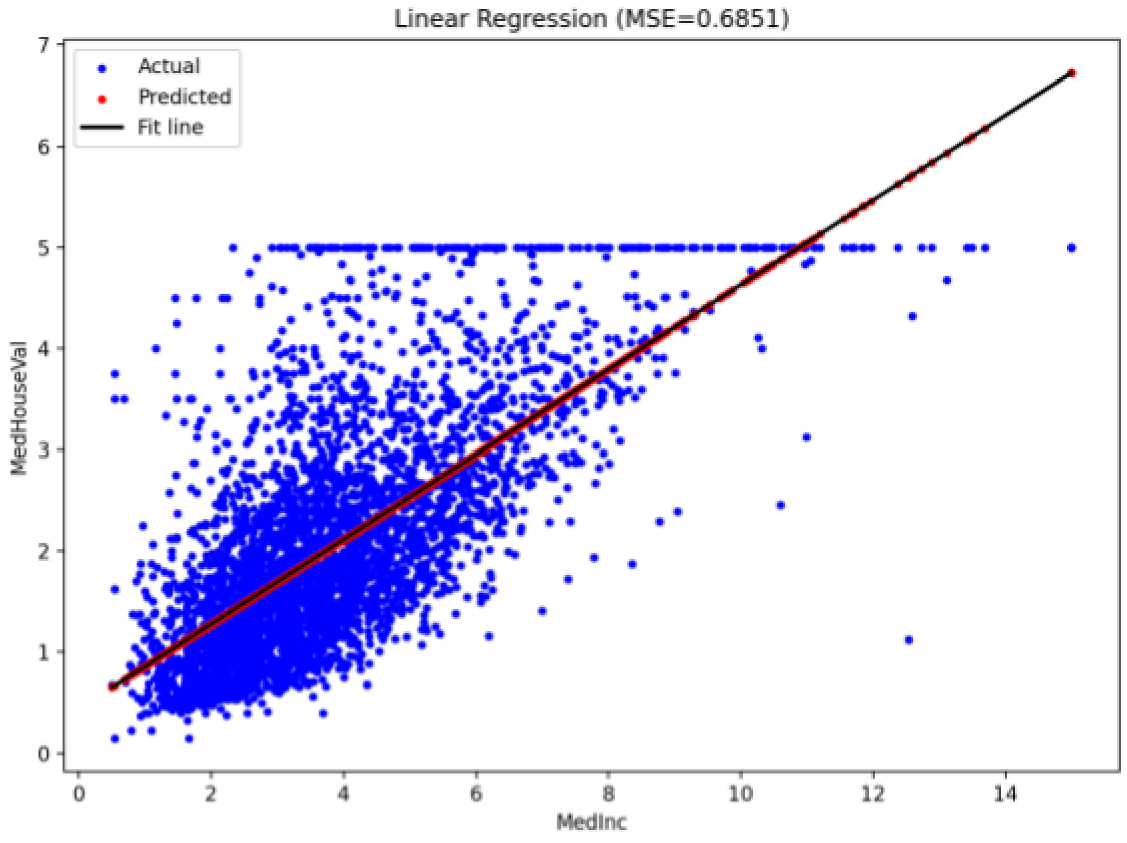
Exercise 4b

I applied both the Statology linear regression model and the multivariate LSTM model (from Exercise 2) to the California housing dataset to predict the median house value (MedHouseVal). Both models were trained and tested using an 80/20 data split.

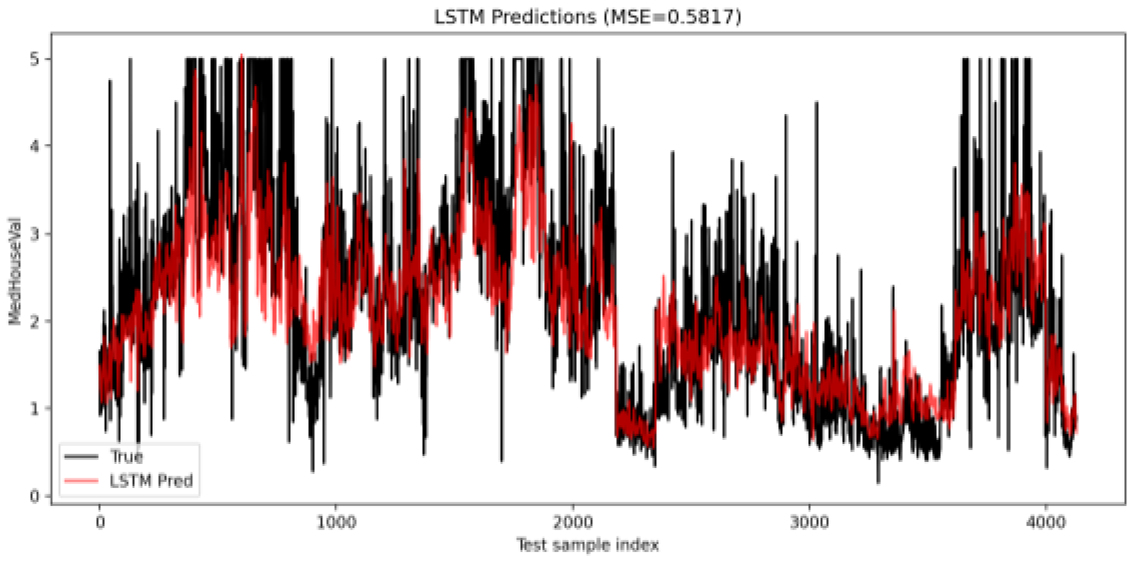
- Linear Regression MSE: 0.6851
- LSTM MSE: 0.5817

The LSTM achieved a lower mean squared error. It indicates that it captured slightly more variance in the target values compared to the Statology simple linear model.

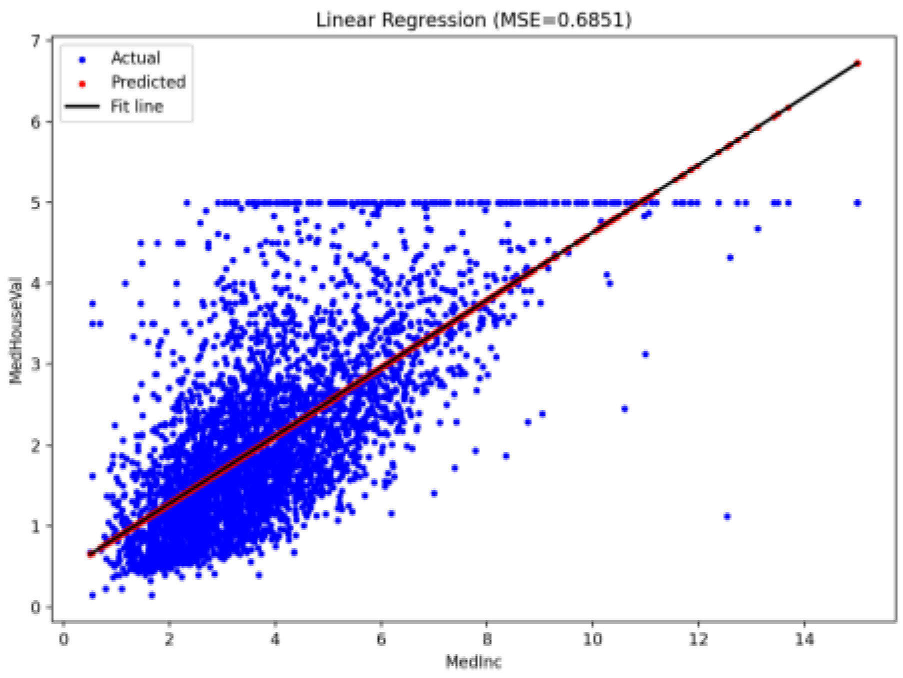
Based on the visual comparison, the LSTM predictions followed the true test values more closely; on the other hand, the linear regression fit was more rigid and unable to model nonlinearities between income and house value. As a whole, the LSTM predictor outperformed the linear regressor for this California housing dataset. It produced smoother and more accurate predictions.

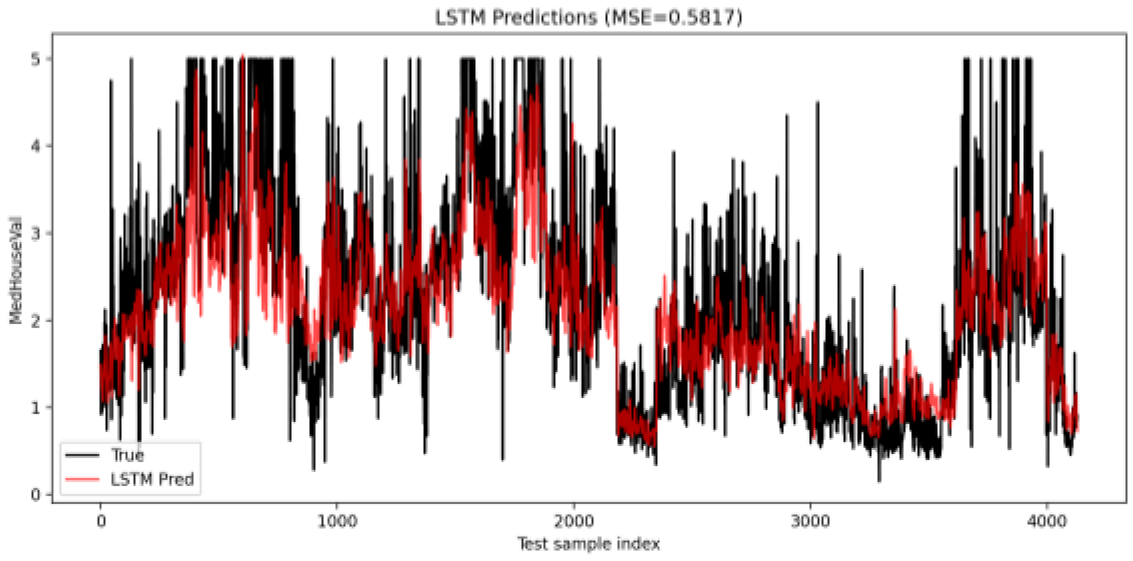


Actual vs predicted with fitted regression line



True vs predicted test sequence (LSTM)





Combined comparison

Exercise 5

I ran “anomally_detection_autoencoder.py” locally, and it is working.

We can make the following targeted changes to improve the anomaly classification performance of the original “anomally_detection_autoencoder.py”:

- Add Dropout layers inside both the Encoder and Decoder to reduce overfitting on normal data. In this way, the model is able to generalize better to unseen anomalous heartbeats.
- Change the loss function from nn.L1Loss (MAE) to a combination of L1 + small L2 term (MSE) in order to penalize larger reconstruction deviations more strongly, which improves anomaly separation.
- Using batch training, for example, mini-batches instead of one sequence at a time, like `DataLoader(train_dataset, batch_size=32, shuffle=True).` As a result, learning becomes more stable and gives smoother convergence.

What is the autoencoder doing in this code, and what is its purpose?

Here in this code, the LSTM autoencoder learns to compress and then reconstruct time-series signals (heartbeats). Because it is trained only on normal heartbeats, it learns typical patterns of normal behavior. When it sees a normal sequence, it reconstructs it accurately with low loss. When it sees an abnormal sequence, which is an anomaly, it fails to reconstruct it well, so it produces a higher reconstruction error. We classify each sequence as normal or anomalous by comparing this error to a threshold.

What is the best strategy to improve the code that is slow in the first place? In order to save time, is there any way to monitor the improvements on the fly?

The best strategy to improve the code that is slow in the first place is vectorization and batching:

- Using DataLoader with a reasonable batch size (for example, 32 or 64).
- Utilizing the GPU if available in the machine.
- Reducing sequence length or embedding dimension if it is possible.
- Avoiding .to(device) calls inside loops, which is redundant.

To monitor the improvements on the fly, we can plot or log the validation loss every few epochs. This allows us to visually confirm whether training is converging or if early stopping can be applied to save time.

Exercise 6a

I followed the DeepAR tutorial using the default synthetic dataset and implemented a clear 80/20 train-test split. I modified the code to compute and save mean squared error per individual test sequence as a CSV file (deepar_mse_per_sample.csv). (Check/Download here). The CSV file confirms that it contains MSE values for all 100 validation samples with errors ranging from near-zero (for example, series 19 with 0.00048) to higher outliers (for example, series 74 with 3.16 and series 6 with 2.23). This shows that the model performs well on most series but not as well with a few. It is expected in real-world scenarios.

Exercise 6b

I applied the recommended hint from the tutorial to improve the previous code and minimize MSE for all test sequences. Increased the number of training batches and epochs for better convergence. I also adjusted model capacity by increasing hidden size and number of RNN layers. These changes helped to lower the MSE for all test sequences.