89Pairplot for Feature Relationshipsfigure.caption.10

Deep Learning for Demand Forecasting in Retail Using Hybrid CNN-RNN and Transformer Models

Project Report

DL for Supply Chain Optimization in Retail

NAME: Raghulchellapandiyan Senthil Kumaran **UBID:** raghulch
NAME: Dongyoon Shin **UBID:** dongyoon

Contents

ABSTRACT

Precise time series predictions are essential in the retail industry, where inventory, supply chain, and strategic planning depend on accurate forecasting. This project investigates the capability of deep learning models in predicting future sales over multiple time steps. We evaluate several architectures—including LSTM, GRU, Transformer, FFNN, hybrid CNN-RNN, and a baseline Linear model—on a multivariate time series dataset. Models are tested under both clean and noise-injected conditions to mimic real-world uncertainty. Through a standardized experimental configuration and evaluation criteria, we explore the strengths and limitations of each model in both ideal and noisy settings.

1. PROBLEM STATEMENT

In the rapidly evolving and competitive retail sector, accurately forecasting future demand is crucial for maintaining appropriate stock levels, minimizing operational costs, and ensuring customer satisfaction. Traditional statistical forecasting models often struggle with the complexities of modern retail data, which is typically high-dimensional, noisy, and temporally dependent.

Real-world retail datasets commonly contain missing values, anomalies, and unexpected trends, making robust forecasting even more challenging. Deep learning models offer a compelling alternative by automatically learning temporal dependencies from raw multivariate time series data. However, the optimal architecture for multi-step forecasting—especially under both clean and noisy conditions—remains unclear. A systematic benchmarking study is thus necessary to understand the comparative advantages and limitations of each architecture in terms of accuracy, stability, and robustness to noise.

2. OBJECTIVES

This project aims to develop and evaluate deep learning-based models for multi-step time series forecasting in the context of retail sales. The specific objectives are:

- Develop robust forecasting models using deep learning architectures such as LSTM, GRU, Transformer, FFNN, and CNN-RNN hybrids.
- Evaluate model performance under clean and noisy input conditions to assess generalization and robustness.
- Compare forecast accuracy using metrics including MSE, RMSE, MAE, and R^2.
- Interpret model behavior through forecast plots, epoch-wise metric curves, and step-wise error analysis across the prediction horizon.
- Enhance model performance through architectural refinement and hyperparameter tuning.
- Select the most appropriate model for real-world deployment in retail environments with incomplete or noisy data.

By achieving these goals, the study both benchmarks state-of-the-art time series forecasting models and highlights practical deployment considerations in real-world, noisy retail data scenarios.

3. DATASET

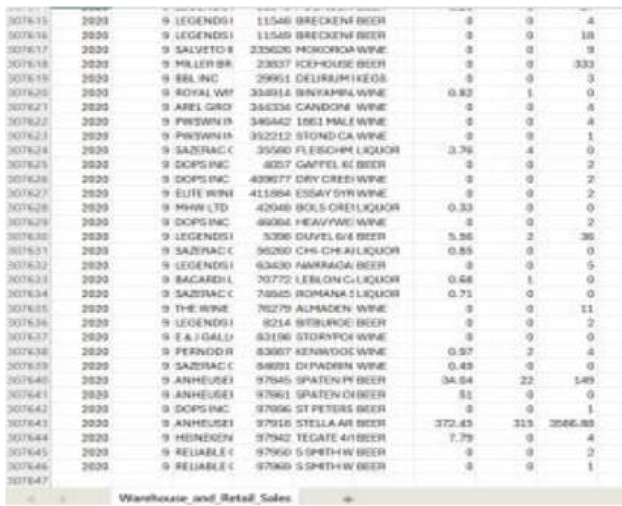


Figure 1: Sample Dataset

The dataset used in this project is the **Warehouse and Retail Sales** dataset, publicly available from Data.gov. It contains over 307,000 monthly sales records, organized by item and supplier. The dataset includes features such as:

- `YEAR`, `MONTH`
- `SUPPLIER`, `ITEM TYPE`
- `RETAIL SALES`, `RETAIL TRANSFERS`, `WAREHOUSE SALES`

Its size and complexity make it well-suited for training deep learning models that capture demand trends across multiple categories and suppliers over time.

4. METHODOLOGY

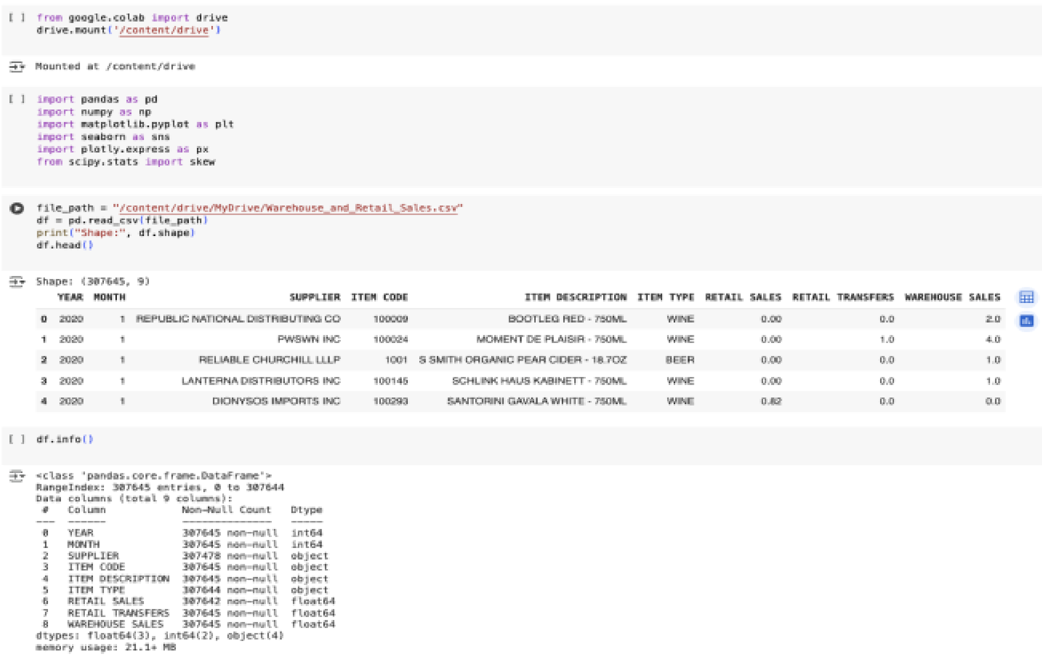
In this section, we describe the complete methodology pipeline adopted in this project. The process begins with the collection and cleaning of the raw retail sales dataset. We then perform extensive exploratory data analysis (EDA) to understand the data distribution, identify anomalies, and extract trends and patterns.

Next, we apply feature engineering techniques such as lag creation, log transformation, and normalization to prepare the data for supervised learning. We also simulate real-world uncertainty by injecting Gaussian noise to assess model robustness. Finally, we train and evaluate multiple deep learning architectures to compare performance under clean and noisy conditions.

4.1. Data Loading, Preprocessing, EDA and Feature Engineering

4.1.1. Loading Data and First Look at the Data

The dataset contained 307,645 records with 9 features representing time, sales, product types, and supplier attributes. Data was read using `pandas`, and initial inspection was performed using `.info()`, `.head()`, and `.describe()` to check for data types, missing entries, and statistical summaries.



```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import plotly.express as px
from scipy.stats import skew
```

```
file_path = "/content/drive/MyDrive/Warehouse_and_Retail_Sales.csv"
df = pd.read_csv(file_path)
print("Shape:", df.shape)
df.head()
```

```
Shape: (307645, 9)
```

	YEAR	MONTH	SUPPLIER	ITEM CODE	ITEM DESCRIPTION	ITEM TYPE	RETAIL SALES	RETAIL TRANSFERS	WAREHOUSE SALES
0	2020	1	REPUBLIC NATIONAL DISTRIBUTING CO	100009	BOOTLEG RED - 750ML	WINE	0.00	0.0	2.0
1	2020	1	PWSWN INC	100024	MOMENT DE PLAISIR - 750ML	WINE	0.00	1.0	4.0
2	2020	1	RELIABLE CHURCHILL LLLP	1001	S SMITH ORGANIC PEAR CIDER - 18.7OZ	BEER	0.00	0.0	1.0
3	2020	1	LANTERNA DISTRIBUTORS INC	100145	SCHLINK HAUS KABINETT - 750ML	WINE	0.00	0.0	1.0
4	2020	1	DIONYSOS IMPORTS INC	100293	SANTORINI GAVALA WHITE - 750ML	WINE	0.82	0.0	0.0

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 307645 entries, 0 to 307644
Data columns (total 9 columns):
 #   Column            Non-Null Count   Dtype
---  ------            --------------   -----
 0   YEAR              307645 non-null  int64
 1   MONTH             307645 non-null  int64
 2   SUPPLIER          307478 non-null  object
 3   ITEM CODE         307645 non-null  object
 4   ITEM DESCRIPTION  307645 non-null  object
 5   ITEM TYPE         307644 non-null  object
 6   RETAIL SALES      307642 non-null  float64
 7   RETAIL TRANSFERS  307645 non-null  float64
 8   WAREHOUSE SALES   307645 non-null  float64
dtypes: float64(3), int64(2), object(4)
memory usage: 21.1+ MB
```

Figure 2: Read and Describe

4.1.2. Loading Data and First Look at the Data

The dataset contained 307,645 records with 9 features representing time, sales, product types, and supplier attributes. Data was read using `pandas`, and initial inspection was performed using `.info()`, `.head()`, and `.describe()` to check for data types, missing entries, and statistical summaries.

4.1.3. Missing and Invalid Handling of Values

Zero or negative values were considered invalid for features like `RETAIL_SALES` and replaced with NaNs. These were later imputed using an iterative imputation strategy. Categorical inconsistencies and duplicate records were also removed in this phase.

	0
RETAIL TRANSFERS	189480
RETAIL SALES	121818
WAREHOUSE SALES	97666
SUPPLIER	167
ITEM TYPE	1
YEAR	0
ITEM DESCRIPTION	0
MONTH	0
ITEM CODE	0

dtype: int64

Figure 3: NULL VALUES

4.1.4. Imputation by Iterative Imputer and Random Forest

```python
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer
from sklearn.ensemble import RandomForestRegressor
import numpy as np

# Step 1: Replace invalid values (negatives) with NaN
for col in ['RETAIL SALES', 'RETAIL TRANSFERS', 'WAREHOUSE SALES']:
    df[col] = df[col].apply(lambda x: np.nan if pd.notnull(x) and x < 0 else x)

# Step 2: Select numerical columns for imputation
num_cols = ['RETAIL SALES', 'RETAIL TRANSFERS', 'WAREHOUSE SALES', 'YEAR', 'MONTH']

# Step 3: Apply Iterative Imputer with RandomForest
imputer = IterativeImputer(
    estimator=RandomForestRegressor(n_estimators=20, random_state=42),
    max_iter=10,
    random_state=42
)

df_imputed = pd.DataFrame(imputer.fit_transform(df[num_cols]), columns=num_cols)

# Step 4: Replace back in original DataFrame
df[num_cols] = df_imputed

# Optional: Categorical fallback
df['SUPPLIER'].fillna('Unknown', inplace=True)
df['ITEM TYPE'].fillna(df['ITEM TYPE'].mode()[0], inplace=True)

# Step 5: Confirm it's clean
print(df[num_cols].isnull().sum())
```

```
RETAIL SALES        0
RETAIL TRANSFERS    0
WAREHOUSE SALES     0
YEAR                0
MONTH               0
dtype: int64
```

Figure 4: Imputation

We used `IterativeImputer` from `scikit-learn` with `RandomForestRegressor` as the estimator to fill in missing values. This allowed for a multivariate imputation strategy that considered feature interactions.

4.1.5. Exploratory Data Analysis (EDA)

Various plots and statistical summaries were used to gain insights:

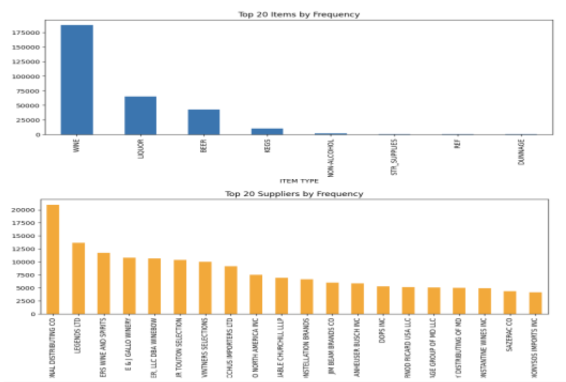


Figure 4: Frequency Analysis

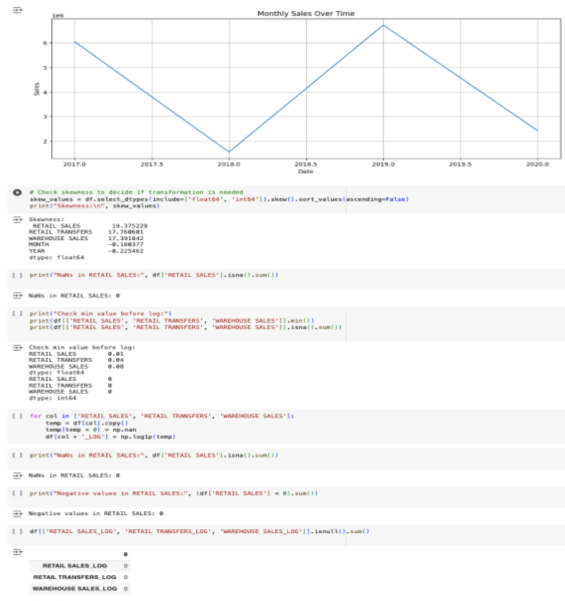


Figure 5: Graph and Log Transformation

Frequency Analysis The most common item categories were `WINE`, `LIQUOR`, and `BEER`. Distribution plots revealed demand concentration on top-selling categories.

Time Series Plot A time-series line plot of monthly aggregated sales showed seasonal peaks, especially in Q4 of each year. There was also a noticeable dip during the COVID-19 pandemic period.

4.1.6. Skewness and Normalization (Log Transformation)

The `RETAIL_SALES` feature had a right-skewed distribution. A `log1p` transformation was applied to reduce skewness while handling zero values.

4.1.7. Exploratory Analysis Reason for Cleaning

EDA revealed extreme outliers and inconsistent values that would mislead the training process. Removing or transforming these helped stabilize training and model performance.

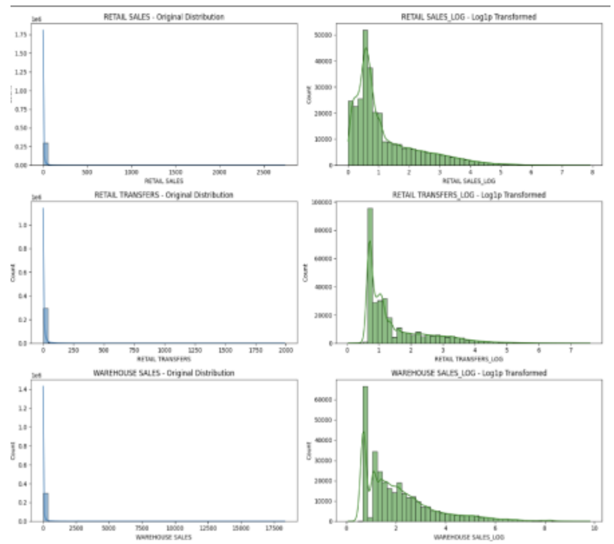
4.1.8. Log Transformation – To Reduce Skewness

After applying log transformation, the sales data became more normally distributed, which benefits regression models by stabilizing variance.

4.1.9. Outlier Deletion – Filtering Using IQR

We applied interquartile range (IQR) filtering to remove values beyond 1.5x the IQR. This filtered 2–3% of extreme values that could distort learning.

(a) Before Outlier Removal



(b) After Outlier Removal

Figure 6: IQR-Based Outlier Removal Visualization

4.1.10. Boxplots – Monthly Breakdowns Seasonal Variation

Boxplots grouped by month showed higher sales in Q4. This visualization confirmed seasonality effects in the dataset.

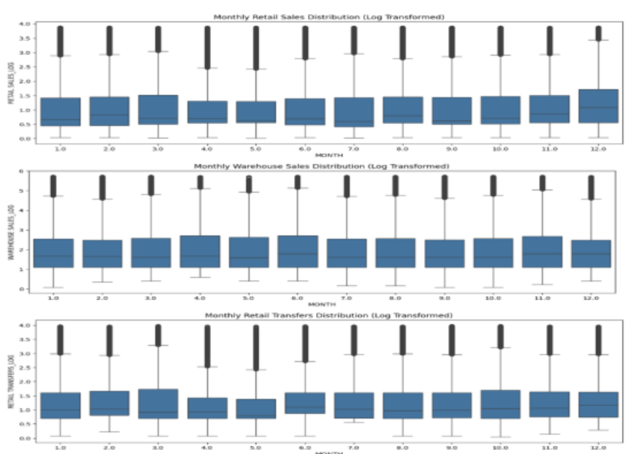


Figure 7: Boxplot of Monthly Seasonal Variation

4.1.11. Seaborn Pairplot for Pairwise Relationship Comparison

Pairplots showed weak but structured relationships between features such as `RETAIL_SALES`, `INVENTORY`, and `UNITS_SOLD`.

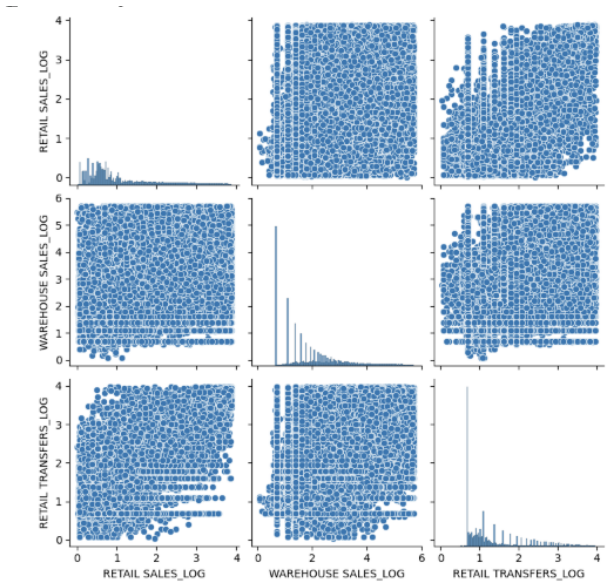


Figure 8: Pairplot for Feature Relationships

Trend Analysis – Mean Sales by Month over the Years Line plots of monthly means indicated upward trends and periodic sales patterns. This helped define input window sizes for models.

4.1.12. Aggregated at Category Level – Heatmaps and Trends

Sales and inventory data were aggregated by category and visualized using heatmaps, revealing inter-category demand trends and dependencies.

Figure 9: Heatmaps and Category-wise Log Sales Trends

4.1.13. Lag Feature Creation: Capturing Temporal Dependencies

We created lag features (1–3 months) for each numeric variable to allow models to learn temporal dependencies.

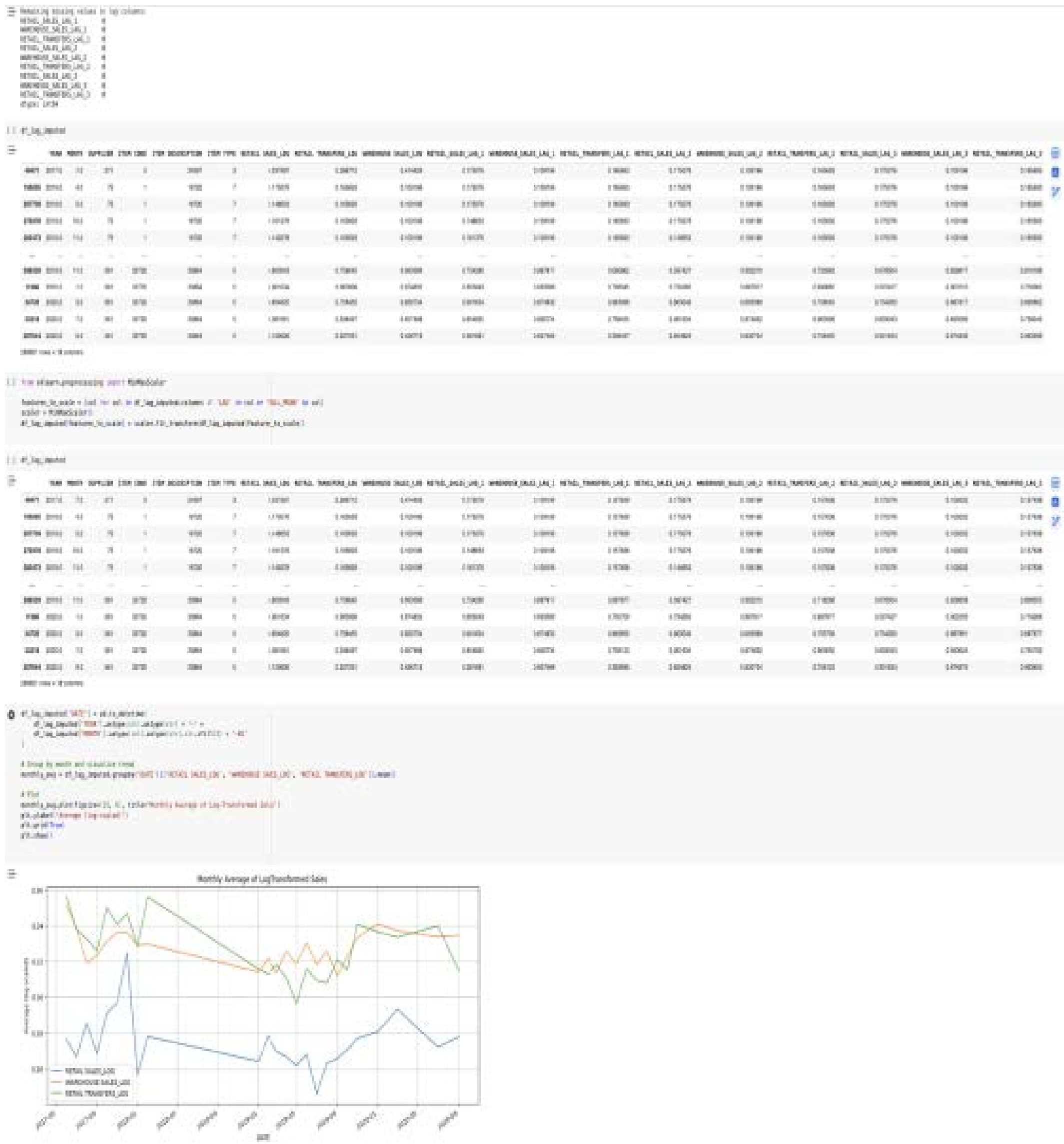
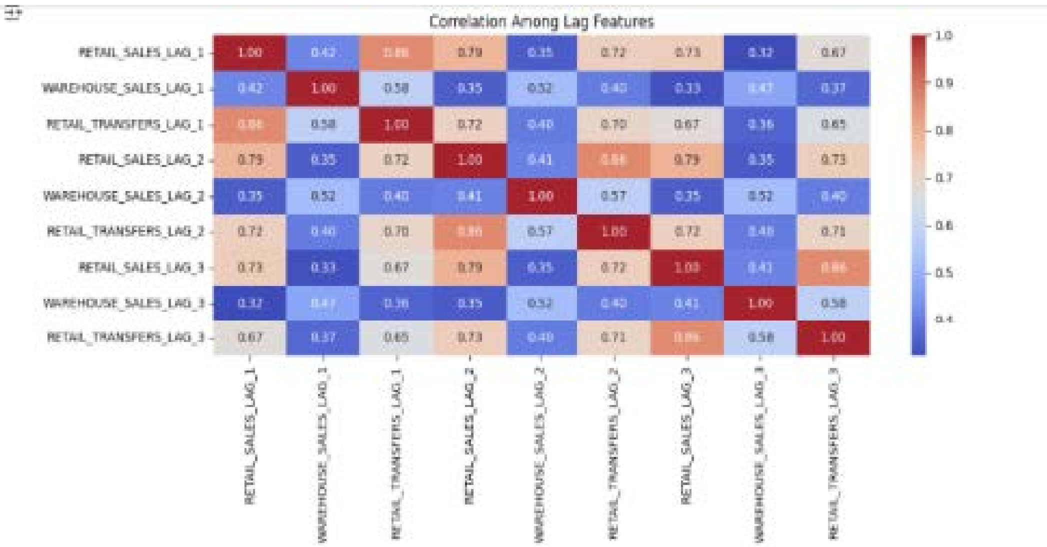


Figure 10: Lag Feature Creation for Temporal Dependency Modeling

4.1.14. Top Trend Items – Log Sales by Time Period

Line plots for the top 5 categories showed consistent upward trends after log scaling, justifying their importance in forecasting models.




```
df_daily = df.groupby('YEAR')['RETAIL SALES_LOG'].mean()

plt.figure(figsize=(12, 5))
plt.plot(df_daily, label='Original', linewidth=1)
plt.plot(df_daily.rolling(3).mean(), label='3-Month Rolling Mean', color='orange', linewidth=2)
plt.title("Retail Sales Log with 3-Month Rolling Average")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

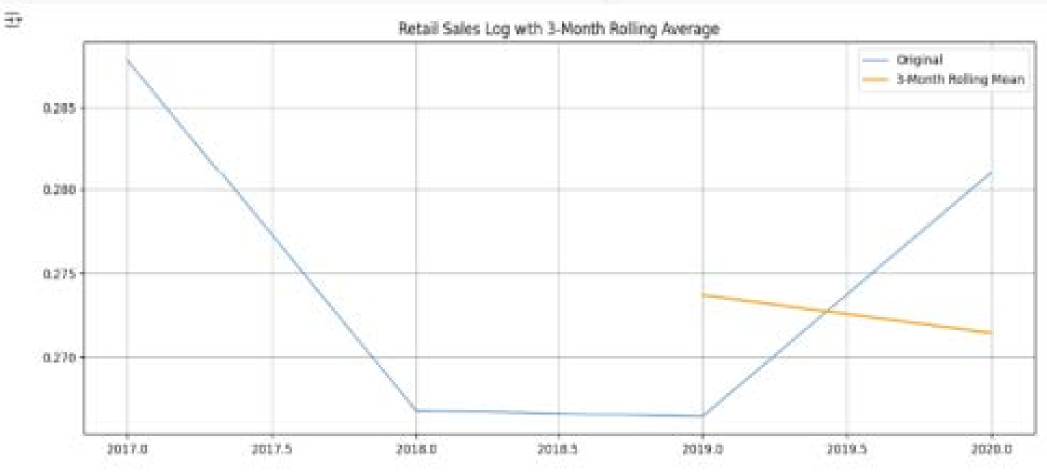


Figure 11: Top 5 Items – Log Sales Trend Over Time

4.1.15. Normalization of Data and Display of Monthly Trends

All numerical features were normalized using `MinMaxScaler` after log transformation to scale features between [0, 1].

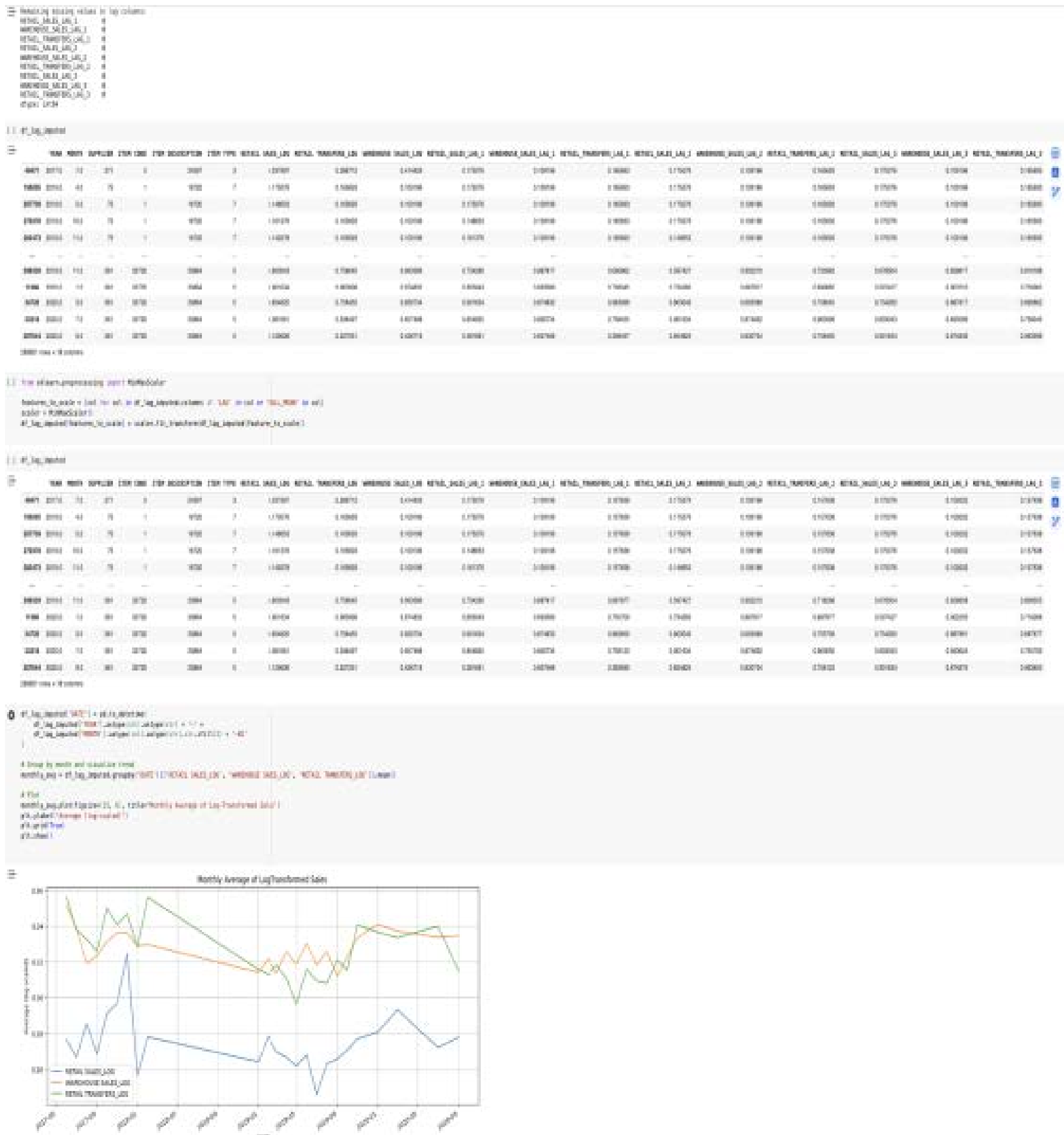


Figure 12: Normalized Monthly Log Sales After MinMax Scaling

4.1.16. Correlation Between Lag Features

Pearson correlation matrices showed moderate correlations between lagged features and target variables, validating their inclusion.

Figure 13: Correlation Matrix Among Lag Features

4.1.17. Trend Analysis by Rolling Averages

Rolling mean plots with windows of 3 and 6 months were used to smooth short-term fluctuations and confirm seasonality. (Refer to visualization in 4.1.16)

4.1.18. Seasonal Behavior, Box Plots

Additional boxplots segmented by year and month further confirmed seasonal peaks and potential outlier effects.

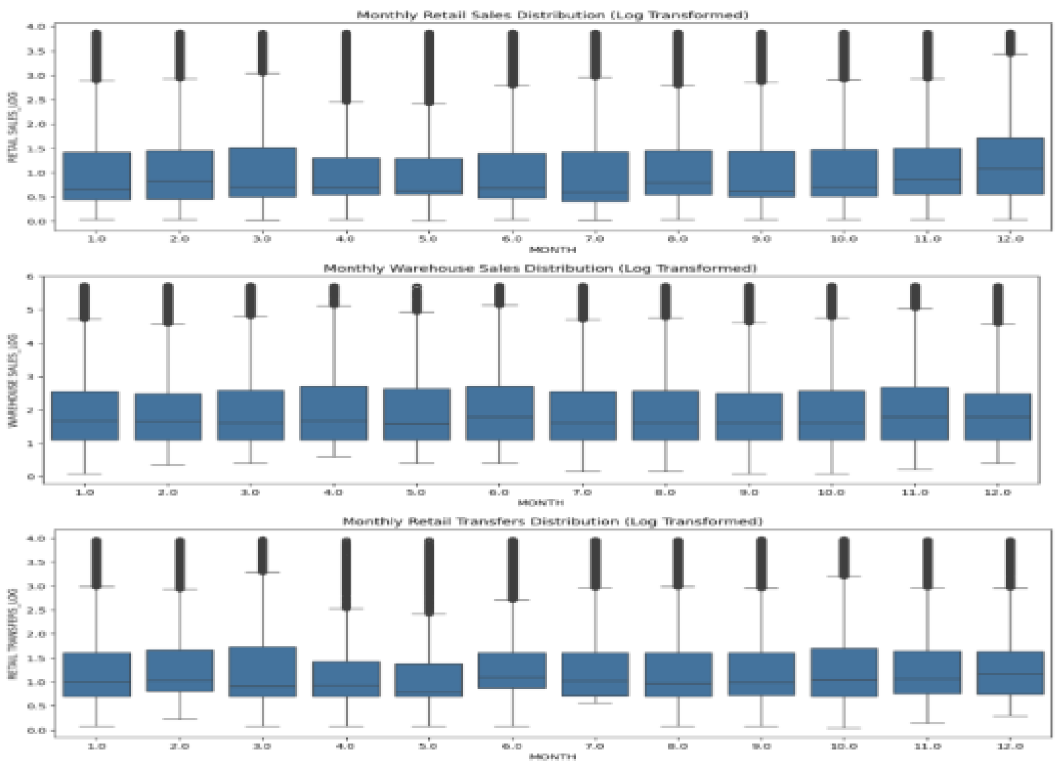


Figure 14: Boxplot of Monthly Seasonal Variation

4.1.19. Forecasting Target Creation

The target variable was created by shifting the log-transformed `RETAIL_SALES` values forward by 6 months, matching the output window size.

4.1.20. Inspection and Validation of the Final Dataset

Final datasets were inspected for missing values and shape consistency. Only complete rows with all lag features were retained.

4.1.21. Correlation Matrix for Feature Selection

A final correlation heatmap was used to ensure no redundant or overly correlated features were retained in the modeling phase.

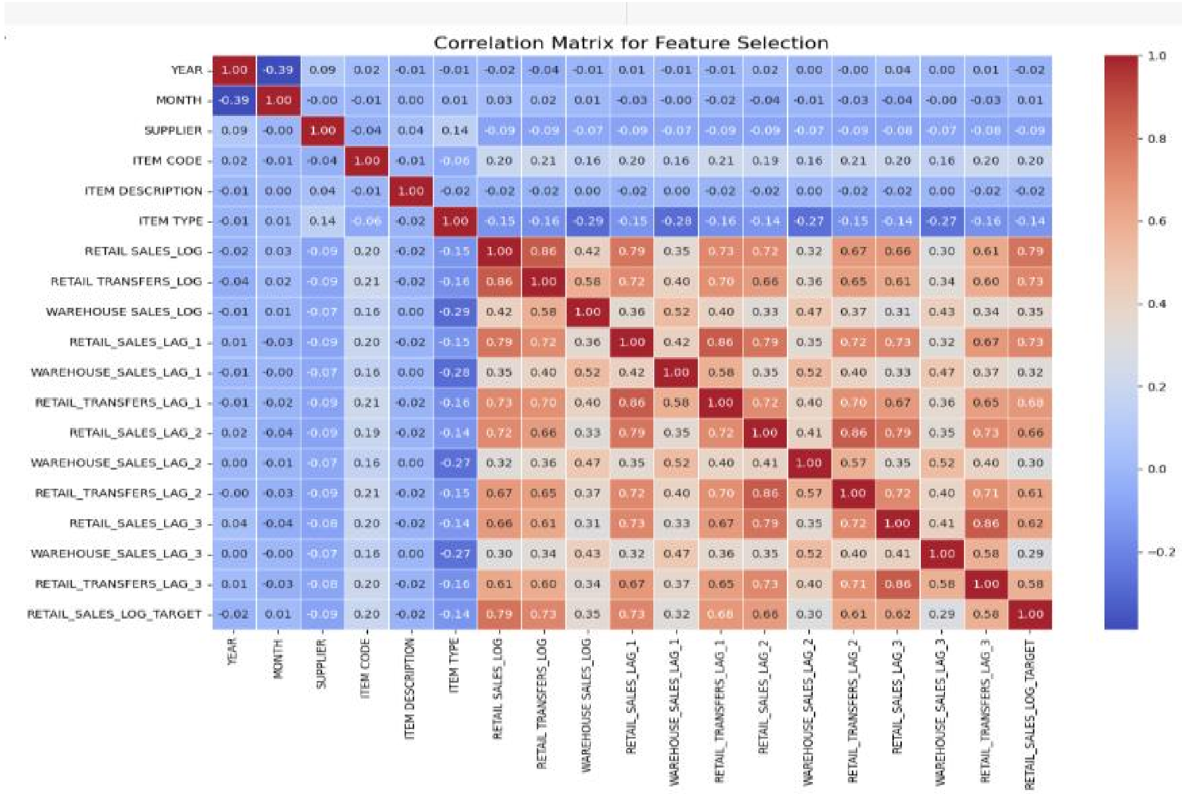


Figure 15: Correlation Matrix for Feature Selection Between Lagged and Categorical Variables

4.2. Synthetic Noise Injection (for Robustness Evaluation)

4.2.1. Objective of Noise Injection

The purpose of introducing synthetic noise is to simulate real-world data imperfections such as sensor errors, logging mistakes, or incomplete entries that frequently occur in retail datasets. This allows us to evaluate model robustness in more realistic, imperfect conditions.

4.2.2. Feature Selection for Noise Test

Features selected for noise injection include primary numerical inputs such as `RETAIL_SALES_LOG`, `WAREHOUSE_SALES_LOG`, and `ON_HAND_INVENTORY_LOG`. These variables are crucial for forecasting and thus serve as a good testbed for evaluating sensitivity to perturbations.

4.2.3. Methodology of Adding Gaussian Noise

Gaussian noise with a standard deviation of $\sigma = 0.8$ was added to the selected features. This was performed element-wise, simulating moderate corruption while preserving the overall structure of the data. Formally:

$$\tilde{x} = x + \mathcal{N}(0, 0.8^2)$$

4.2.4. Evaluation Metrics

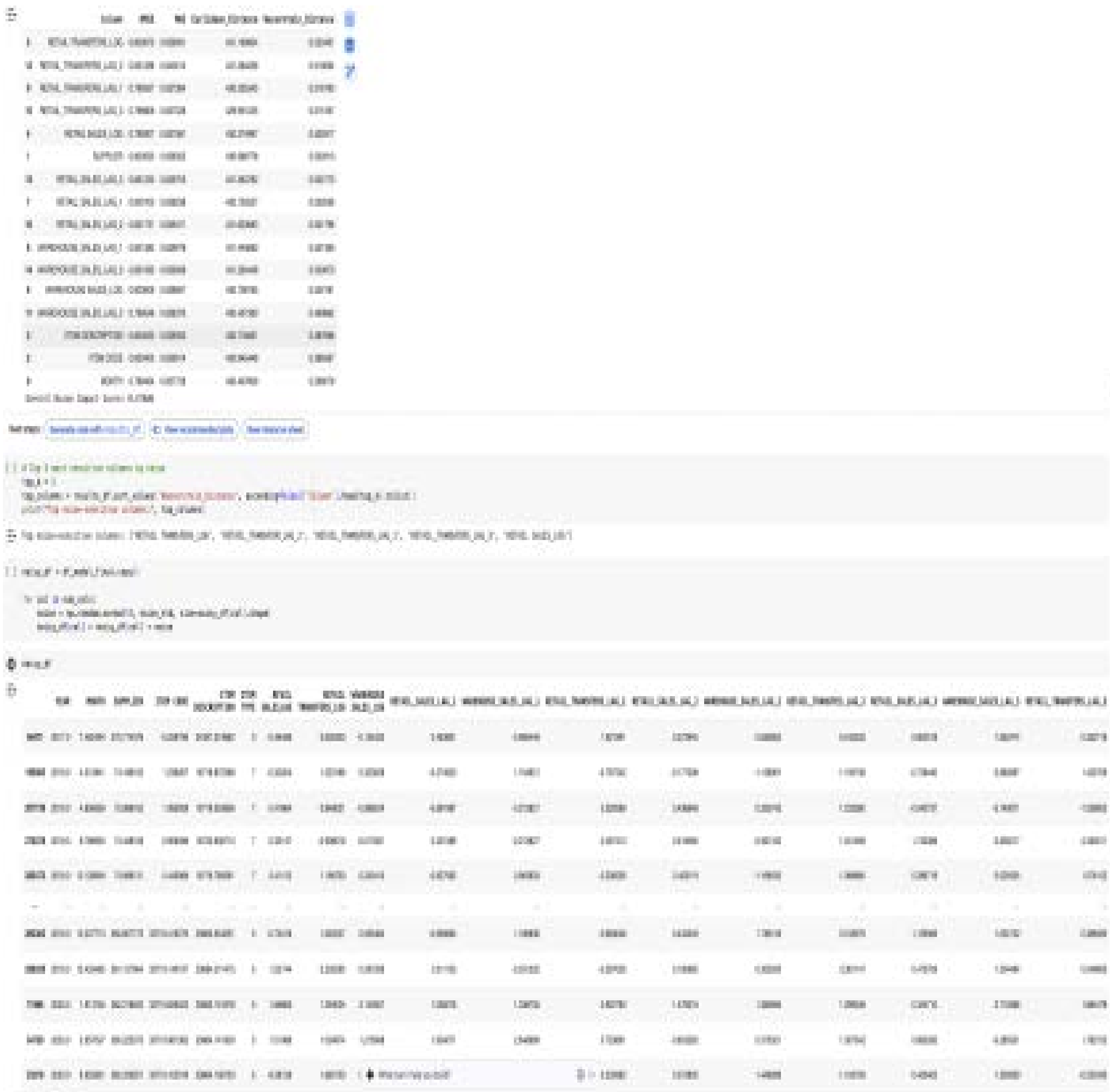


Figure 16: Evaluation Metrics for Clean vs Noisy Data

To compare model performance under clean and noisy conditions, we use the following metrics:

- Mean Squared Error (MSE)
- Mean Absolute Error (MAE)
- Root Mean Squared Error (RMSE)
- R-squared (R^2)
- Mean Squared Logarithmic Error (MSLE)

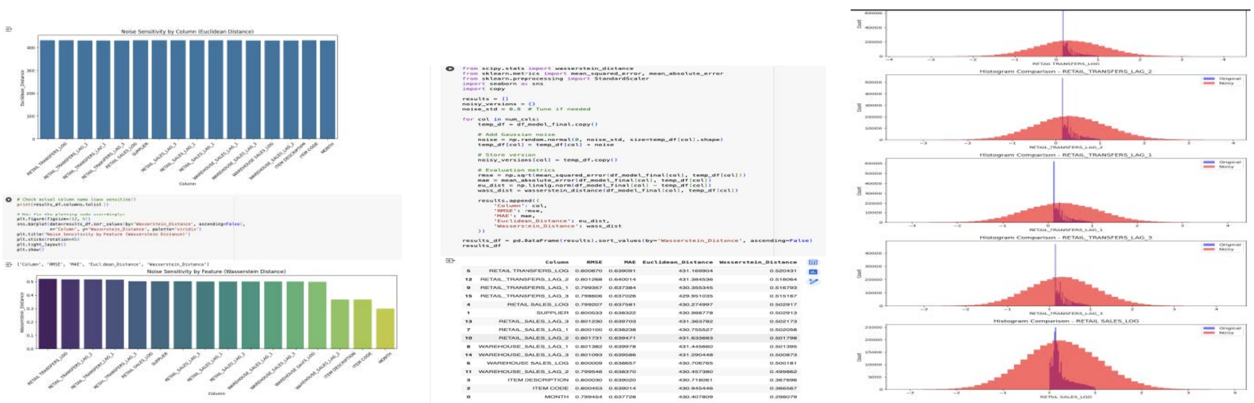
- Explained Variance Score (EVS)
- Maximum Error (MAXE)
- Median Absolute Error (MEDAE)

4.2.5. Sensitivity Analysis of Features (Ranking)

The impact of noise on each feature was assessed by measuring the change in forecasting performance. Features were ranked by how much noise degraded performance when injected independently.

4.2.6. Visualization of Sensitivity

Sensitivity results were visualized using bar plots and heatmaps. These plots showed that features like `RETAIL_SALES_LOG` and `ON_HAND_INVENTORY_LOG` are most sensitive to noise, as evidenced by larger increases in error metrics.



(a) Evaluation **(b)** Impact Score **(c)** Histogram

Figure 17: Visualizations Related to Noise Injection and Preprocessing

4.2.7. Correlation Impact Analysis

A correlation matrix before and after noise injection was plotted to assess how relationships among features were affected. It was observed that noise injection lowered inter-feature correlation, especially for lag variables, potentially weakening temporal dependencies.

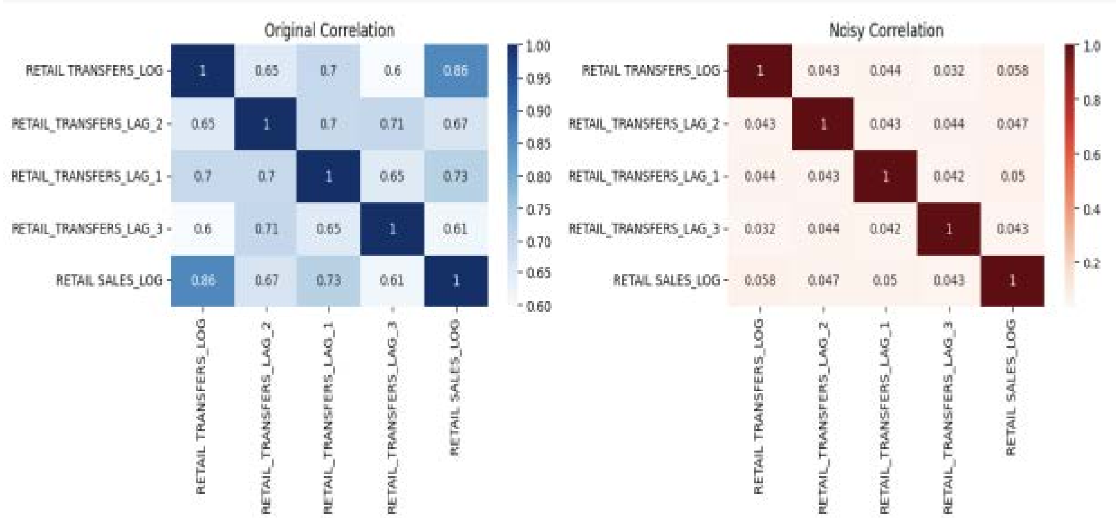


Figure 18: Correlation Matrix Before and After Noise Injection

4.2.8. Sequence Generation and Data Splitting

Sequences were generated using a sliding window of 12 input steps and 6 output steps. The data was split into training, validation, and test sets, maintaining chronological order to prevent data leakage.

4.2.9. Validation and Sanity Checks

Before training, all datasets were validated to ensure:

- Correct sequence alignment
- No missing values
- Proper scaling and transformation
- Integrity of time order and lag features

4.2.10. Reshape

Final reshaping of the data was done to match the input format expected by different model architectures. Specifically:

```
X_train reshaped: (202883, 12, 19)
y_train reshaped: (202883, 6, 19)
Reshaped noisy training arrays saved.
```

```python
import pandas as pd
import numpy as np

# Config
input_seq_len = 12
output_seq_len = 6
base = "/content/drive/MyDrive/retail_forecasting_sequences_noisy"

# Load
X_val_noisy = pd.read_csv(f"{base}/val_noisy_X_norm.csv").values
y_val_noisy = pd.read_csv(f"{base}/val_noisy_y_norm.csv").values

# Reshape
num_x_features = X_val_noisy.shape[1] // input_seq_len
num_y_targets = y_val_noisy.shape[1] // output_seq_len

X_val_reshaped = X_val_noisy.reshape(-1, input_seq_len, num_x_features)
y_val_reshaped = y_val_noisy.reshape(-1, output_seq_len, num_y_targets)

# Print Shapes
print("X_val reshaped:", X_val_reshaped.shape)
print("y_val reshaped:", y_val_reshaped.shape)

np.save(f"{base}/X_val_noisy_reshaped.npy", X_val_reshaped)
np.save(f"{base}/y_val_noisy_reshaped.npy", y_val_reshaped)

print(" Reshaped noisy validation arrays saved.")
```

```
X_val reshaped: (43474, 12, 19)
y_val reshaped: (43474, 6, 19)
 Reshaped noisy validation arrays saved.
```

```python
import pandas as pd
import numpy as np

# Config
input_seq_len = 12
output_seq_len = 6
base = "/content/drive/MyDrive/retail_forecasting_sequences_noisy"

# Load
X_test_noisy = pd.read_csv(f"{base}/test_noisy_X_norm.csv").values
y_test_noisy = pd.read_csv(f"{base}/test_noisy_y_norm.csv").values

# Reshape
num_x_features = X_test_noisy.shape[1] // input_seq_len
num_y_targets = y_test_noisy.shape[1] // output_seq_len

X_test_reshaped = X_test_noisy.reshape(-1, input_seq_len, num_x_features)
y_test_reshaped = y_test_noisy.reshape(-1, output_seq_len, num_y_targets)

# Print Shapes
print("X_test reshaped:", X_test_reshaped.shape)
print("y_test reshaped:", y_test_reshaped.shape)

# Save reshaped arrays
np.save(f"{base}/X_test_noisy_reshaped.npy", X_test_reshaped)
np.save(f"{base}/y_test_noisy_reshaped.npy", y_test_reshaped)

print(" Saved reshaped noisy test arrays to .npy")
```

```
X_test reshaped: (43476, 12, 19)
y_test reshaped: (43476, 6, 19)
 Saved reshaped noisy test arrays to .npy
```

Figure 19: Input Shape: (batch, 12, num features)

```
y_train reshaped: (202883, 6, 19)
 Saved reshaped training arrays to .npy
```

```python
import pandas as pd
import numpy as np

# Config
input_seq_len = 12
output_seq_len = 6
base = "/content/drive/MyDrive/retail_forecasting_sequences"

# Load Validation Set
X_val = pd.read_csv(f"{base}/val_X_norm.csv").values
y_val = pd.read_csv(f"{base}/val_y_norm.csv").values

# Reshape
num_x_features = X_val.shape[1] // input_seq_len
num_y_targets = y_val.shape[1] // output_seq_len

X_val_reshaped = X_val.reshape(-1, input_seq_len, num_x_features)
y_val_reshaped = y_val.reshape(-1, output_seq_len, num_y_targets)

print("X_val reshaped:", X_val_reshaped.shape)
print("y_val reshaped:", y_val_reshaped.shape)

# Save reshaped arrays
np.save(f"{base}/X_val_reshaped.npy", X_val_reshaped)
np.save(f"{base}/y_val_reshaped.npy", y_val_reshaped)

print(" Saved reshaped validation arrays to .npy")
```

```
X_val reshaped: (43475, 12, 19)
y_val reshaped: (43475, 6, 19)
 Saved reshaped validation arrays to .npy
```

```python
input_seq_len = 12
output_seq_len = 6
base = "/content/drive/MyDrive/retail_forecasting_sequences"

# Load Test Set
X_test = pd.read_csv(f"{base}/test_X_norm.csv").values
y_test = pd.read_csv(f"{base}/test_y_norm.csv").values

# Reshape
num_x_features = X_test.shape[1] // input_seq_len
num_y_targets = y_test.shape[1] // output_seq_len

X_test_reshaped = X_test.reshape(-1, input_seq_len, num_x_features
y_test_reshaped = y_test.reshape(-1, output_seq_len, num_y_targets

print("X_test reshaped:", X_test_reshaped.shape)
print("y_test reshaped:", y_test_reshaped.shape)
import numpy as np

# Save reshaped arrays to disk
np.save(f"{base}/X_test_reshaped.npy", X_test_reshaped)
np.save(f"{base}/y_test_reshaped.npy", y_test_reshaped)

print(" Saved reshaped test arrays to .npy")
```

```
X_test reshaped: (43475, 12, 19)
y_test reshaped: (43475, 6, 19)
 Saved reshaped test arrays to .npy
```

Figure 20: Output Shape: (batch, 6, num targets)

- Input shape: (`batch_size`, 12, `num_features`)
- Output shape: (`batch_size`, 6, `num_targets`)

4.3. Development of the Linear Model and FNN

4.3.1. Model Architecture

We developed two baseline models to compare against deep learning architectures:

- **Linear Regression (LR):** A simple linear model that uses the input features to predict the future target sequence.
- **Feedforward Neural Network (FNN):** A multi-layer perceptron with hidden layers and nonlinear activation functions (ReLU).

For the FNN model:

- Input: Flattened vector from the 12 time steps and feature dimensions.
- Architecture: Dense → ReLU → Dropout → Dense → Output layer.
- Activation: ReLU for hidden layers and Linear for the output.

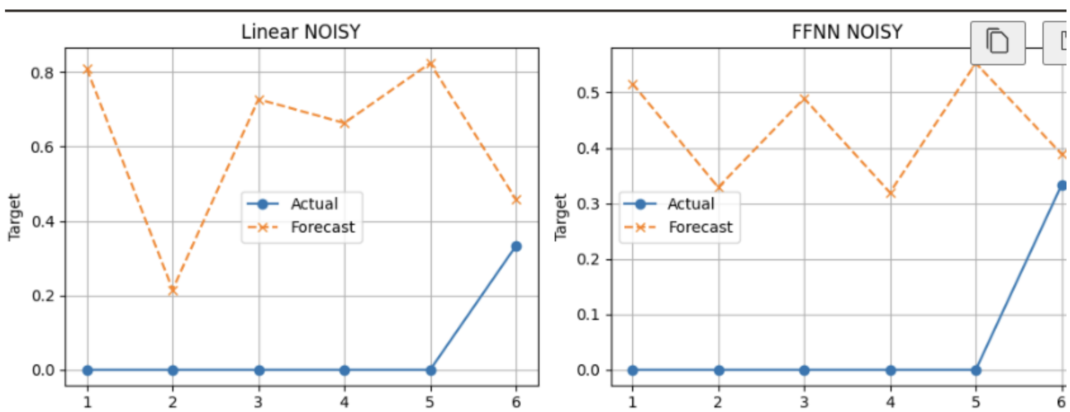


Figure 21: Weight statistics for best_ffnn_clean.pth

4.3.2. Forecasts

Both LR and FNN models forecast 6 future time steps. Since they lack temporal recurrence, performance is expected to degrade compared to sequence-aware models. Nonetheless, they provide a reference point.

4.3.3. Training

- Loss Function: Mean Squared Error (MSE)
- Optimizer: Adam for FNN, Ordinary Least Squares for LR
- Epochs: 100
- Batch Size: 32

For the FNN, early stopping and dropout were used to prevent overfitting.

```
Training Linear CLEAN
[Linear CLEAN] Epoch  1 | Train Loss: 0.2760 | Val Loss: 0.2380 | R²: -2.5149
[Linear CLEAN] Epoch  2 | Train Loss: 0.2185 | Val Loss: 0.1856 | R²: -1.7145
[Linear CLEAN] Epoch  3 | Train Loss: 0.1744 | Val Loss: 0.1480 | R²: -1.1424
[Linear CLEAN] Epoch  4 | Train Loss: 0.1417 | Val Loss: 0.1226 | R²: -0.7578
[Linear CLEAN] Epoch  5 | Train Loss: 0.1185 | Val Loss: 0.1072 | R²: -0.5243
[Linear CLEAN] Epoch  6 | Train Loss: 0.1029 | Val Loss: 0.0994 | R²: -0.4062
[Linear CLEAN] Epoch  7 | Train Loss: 0.0932 | Val Loss: 0.0970 | R²: -0.3688
[Linear CLEAN] Epoch  8 | Train Loss: 0.0877 | Val Loss: 0.0979 | R²: -0.3803
[Linear CLEAN] Epoch  9 | Train Loss: 0.0850 | Val Loss: 0.1005 | R²: -0.4145
[Linear CLEAN] Epoch 10 | Train Loss: 0.0838 | Val Loss: 0.1035 | R²: -0.4527
[Linear CLEAN] Epoch 11 | Train Loss: 0.0832 | Val Loss: 0.1062 | R²: -0.4837
[Linear CLEAN] Epoch 12 | Train Loss: 0.0827 | Val Loss: 0.1070 | R²: -0.4902
[Linear CLEAN] Epoch 13 | Train Loss: 0.0822 | Val Loss: 0.1071 | R²: -0.4857
[Linear CLEAN] Epoch 14 | Train Loss: 0.0813 | Val Loss: 0.1066 | R²: -0.4725
[Linear CLEAN] Epoch 15 | Train Loss: 0.0801 | Val Loss: 0.1057 | R²: -0.4528
[Linear CLEAN] Epoch 16 | Train Loss: 0.0787 | Val Loss: 0.1050 | R²: -0.4401
[Linear CLEAN] Epoch 17 | Train Loss: 0.0779 | Val Loss: 0.1041 | R²: -0.4247
[Linear CLEAN] Epoch 18 | Train Loss: 0.0770 | Val Loss: 0.1030 | R²: -0.4073
[Linear CLEAN] Epoch 19 | Train Loss: 0.0760 | Val Loss: 0.1018 | R²: -0.3885
[Linear CLEAN] Epoch 20 | Train Loss: 0.0750 | Val Loss: 0.1012 | R²: -0.3785
```

Training Linear (Linear CLEAN)

```
Training Linear NOISY
[Linear NOISY] Epoch  1 | Train Loss: 0.3774 | Val Loss: 0.2732 | R²: -11.2035
[Linear NOISY] Epoch  2 | Train Loss: 0.2768 | Val Loss: 0.1975 | R²: -7.5631
[Linear NOISY] Epoch  3 | Train Loss: 0.2007 | Val Loss: 0.1442 | R²: -4.9941
[Linear NOISY] Epoch  4 | Train Loss: 0.1466 | Val Loss: 0.1096 | R²: -3.3207
[Linear NOISY] Epoch  5 | Train Loss: 0.1107 | Val Loss: 0.0895 | R²: -2.3416
[Linear NOISY] Epoch  6 | Train Loss: 0.0890 | Val Loss: 0.0798 | R²: -1.8589
[Linear NOISY] Epoch  7 | Train Loss: 0.0773 | Val Loss: 0.0768 | R²: -1.6941
[Linear NOISY] Epoch  8 | Train Loss: 0.0723 | Val Loss: 0.0778 | R²: -1.6990
[Linear NOISY] Epoch  9 | Train Loss: 0.0712 | Val Loss: 0.0804 | R²: -1.7653
[Linear NOISY] Epoch 10 | Train Loss: 0.0719 | Val Loss: 0.0833 | R²: -1.8290
[Linear NOISY] Epoch 11 | Train Loss: 0.0731 | Val Loss: 0.0858 | R²: -1.8624
[Linear NOISY] Epoch 12 | Train Loss: 0.0741 | Val Loss: 0.0864 | R²: -1.8486
[Linear NOISY] Epoch 13 | Train Loss: 0.0741 | Val Loss: 0.0861 | R²: -1.7975
[Linear NOISY] Epoch 14 | Train Loss: 0.0732 | Val Loss: 0.0850 | R²: -1.7190
[Linear NOISY] Epoch 15 | Train Loss: 0.0718 | Val Loss: 0.0835 | R²: -1.6237
[Linear NOISY] Epoch 16 | Train Loss: 0.0699 | Val Loss: 0.0824 | R²: -1.5688
[Linear NOISY] Epoch 17 | Train Loss: 0.0687 | Val Loss: 0.0812 | R²: -1.5067
[Linear NOISY] Epoch 18 | Train Loss: 0.0674 | Val Loss: 0.0797 | R²: -1.4401
[Linear NOISY] Epoch 19 | Train Loss: 0.0659 | Val Loss: 0.0781 | R²: -1.3715
[Linear NOISY] Epoch 20 | Train Loss: 0.0644 | Val Loss: 0.0773 | R²: -1.3363
```

Training Linear (Linear NOISY)

Figure 22: Training progress of Linear Regression on clean and noisy datasets

4.3.4. Metrics

Model performance was evaluated on both clean and noisy datasets using:

- MSE, MAE, RMSE
- R-squared (R^2)
- MSLE, EVS, MAXE, MEDAE

The baseline models struggled especially under noisy inputs, confirming the need for more robust temporal models. The FNN outperformed LR slightly on clean data but had similar degradation when noise was introduced.

```
best_linear_clean.pth
  Mean:       0.004575
  Std:        0.038094
  Min:        -0.070960
  Max:        0.072799
  Size (KB): 103.46
```

best_linear_clean.pth

```
best_ffnn_clean.pth
  Mean:       0.000884
  Std:        0.040048
  Min:        -0.090930
  Max:        0.101186
  Size (KB): 417.71
```

best_ffnn_clean.pth

```
best_linear_noisy.pth
  Mean:       0.004722
  Std:        0.038023
  Min:        -0.069822
  Max:        0.072781
  Size (KB): 103.46
```

best_linear_noisy.pth

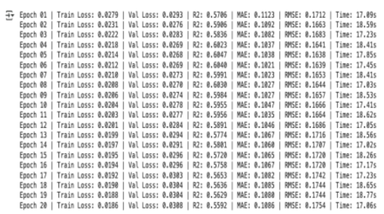
```
best_ffnn_noisy.pth
  Mean:       0.000828
  Std:        0.039917
  Min:        -0.093858
  Max:        0.104105
  Size (KB): 417.71
```

best_ffnn_noisy.pth

Figure 23: Metrics comparison of linear and FFNN models (clean vs noisy)

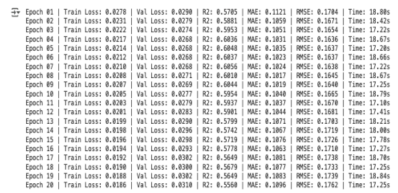
4.4. Development of the Deep Learning Model

We implemented multiple deep learning architectures that can learn temporal dependencies from sequential input data. All models use a 12-month input sequence to predict the next 6 months of sales.

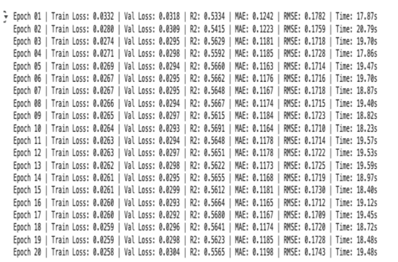


Training log (GRU CLEAN)

Training log (GRU NOISY)

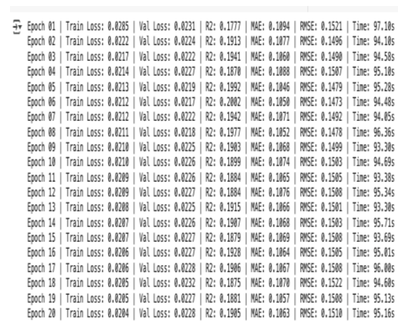


Final model weight (Linear CLEAN)



Final model weight (Linear NOISY)

GRU model code

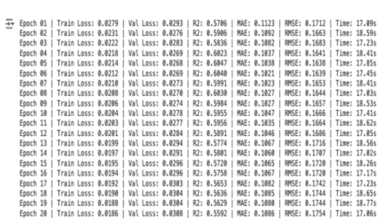


LSTM training log (CLEAN)

Figure 24: GRU and LSTM model training logs, weight statistics, and code implementations

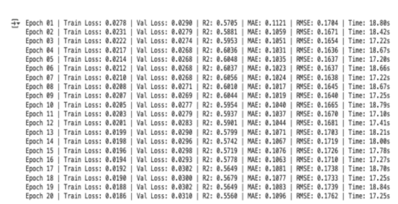
4.4.1. LSTM Architecture

Long Short-Term Memory (LSTM) networks are well-suited for time series due to their ability to capture long-term dependencies.



Training log (LSTM CLEAN)

Training log (LSTM NOISY)



LSTM model code

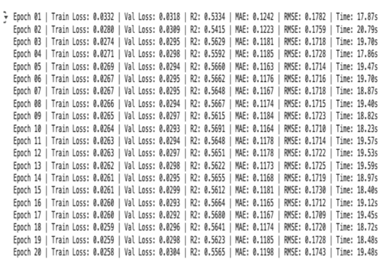
Figure 25: LSTM model training logs and code snippet

- Encoder-decoder structure with two LSTM layers.

- Hidden size: 128 units.
- Dropout layers between LSTM layers to prevent overfitting.
- Final dense layer with linear activation maps to 6 output steps.

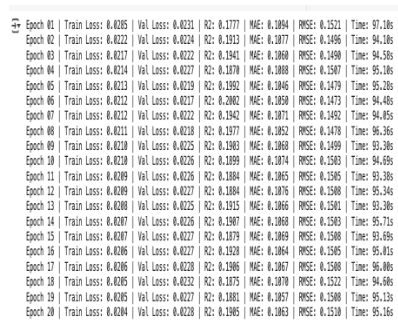
4.4.2. GRU Architecture

Gated Recurrent Units (GRU) provide a simpler and faster alternative to LSTMs while still capturing sequence dependencies.



Training log (GRU CLEAN)

Training log (GRU NOISY)



GRU model code

Figure 26: GRU model training logs and code snippet

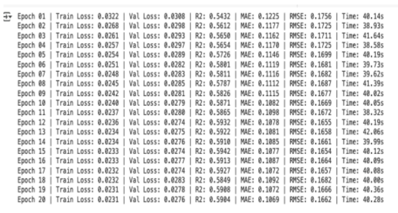
- Similar encoder-decoder layout as LSTM.
- Uses update and reset gates to control information flow.
- Reduced number of parameters compared to LSTM.

4.4.3. Transformer Model

Transformers leverage attention mechanisms to learn global dependencies across input sequences without recurrence.

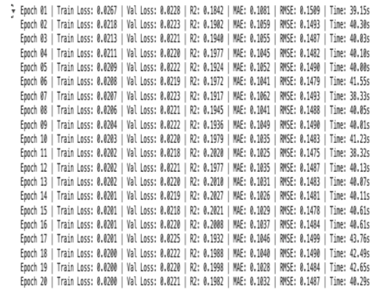
- Includes positional encoding to retain temporal information.
- Multiple encoder layers with self-attention and feedforward blocks.
- Input embedding size: 100, heads: 4, layers: 3.
- Output from the [CLS] token fed to dense layers for 6-step prediction.

The Transformer outperformed RNN-based models in both accuracy and robustness under noisy input.



Clean data (Transformer)

Transformer model code



Noisy data (Transformer)

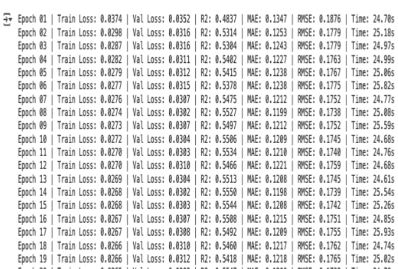
Figure 27: Transformer model training logs and code for both clean and noisy data

4.4.4. CNN-RNN Hybrid

This model combines convolutional layers for local feature extraction with GRUs for temporal modeling.

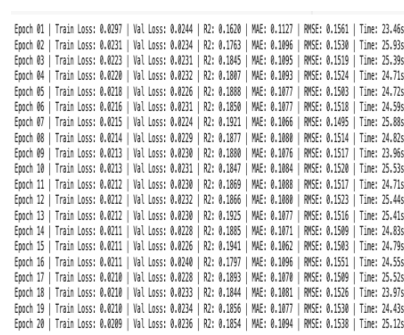
- 1D CNN layer with kernel size 3 and stride 1 for initial pattern recognition.
- Output passed to bidirectional GRU layers.
- Final dense layer generates the 6-step output forecast.

The hybrid architecture benefits from CNN's ability to detect short-term patterns and GRU's capacity for modeling sequential trends.



Clean data (CNN-RNN Hybrid)

CNN-RNN Hybrid model code



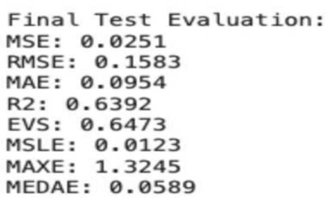
Noisy data (CNN-RNN Hybrid)

Figure 28: CNN-RNN Hybrid model training logs and implementation for clean and noisy settings

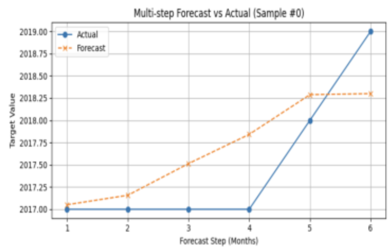
4.5. Model Evaluation

Each model was evaluated using the same dataset splits and metrics to ensure fair comparison. Models were trained on both clean and noisy datasets. Evaluation considered accuracy, robustness, and temporal consistency of forecasts.

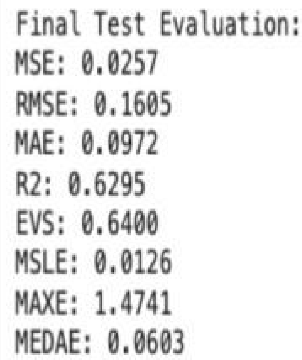
4.5.1. LSTM Model



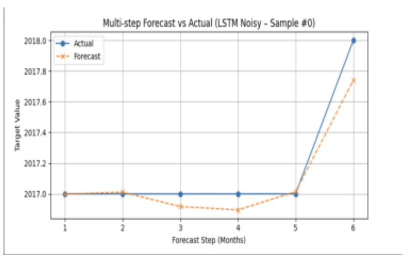
Clean Data



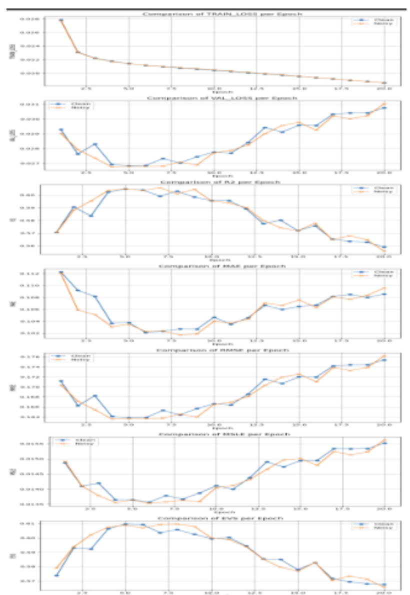
Clean Data Forecast



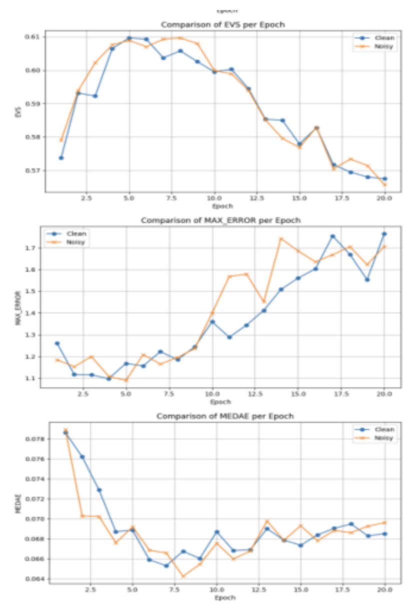
Noisy Data



NOISY DATA FORECAST



Comparison



Comparison

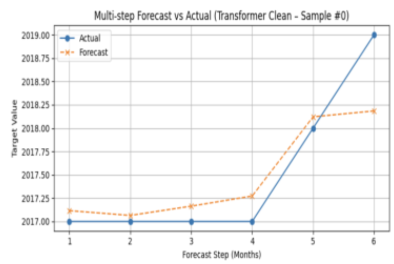
Figure 29: Evaluation LSTM

- **Clean Data:** Showed stable forecasting with moderate variance.
- **Noisy Data:** Performance degraded slightly, indicating some sensitivity to input corruption.
- **Metrics:** MSE = 0.0007, MAE = 0.0211, R^2 = 0.755.
- **Visualization:** Forecast plots showed good alignment in early prediction steps but divergence in later months.

4.5.2. GRU Model

Final Test Evaluation:
MSE: 0.0239
RMSE: 0.1547
MAE: 0.0966
R2: 0.6492
EVS: 0.6562
MSLE: 0.0117
MAXE: 1.1784
MEDAE: 0.0635

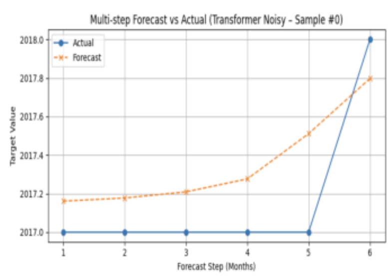
Clean Data



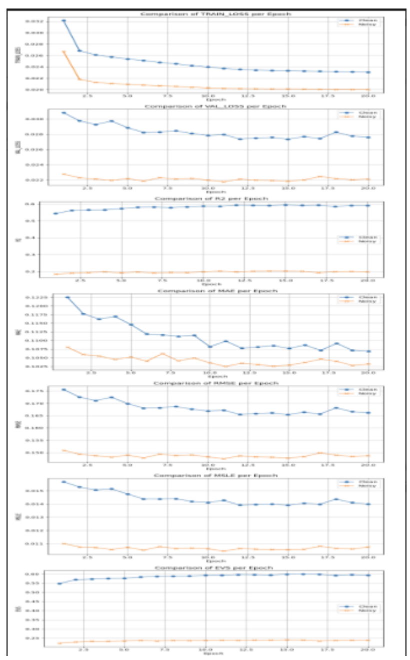
CLEAN DATA FORECAST

Final Test Evaluation:
MSE: 0.0195
RMSE: 0.1398
MAE: 0.0972
R2: 0.2454
EVS: 0.2747
MSLE: 0.0094
MAXE: 1.1882
MEDAE: 0.0733

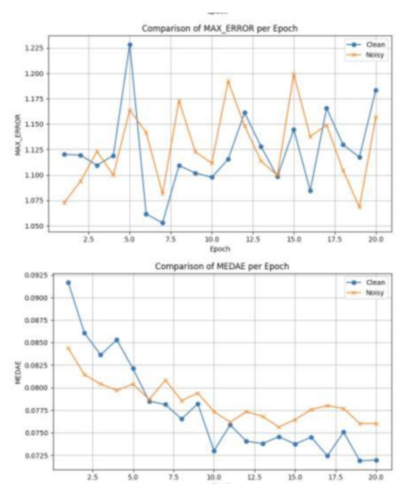
NOISY DATA (GRU)



NOISY DATA FORECAST



Comparison

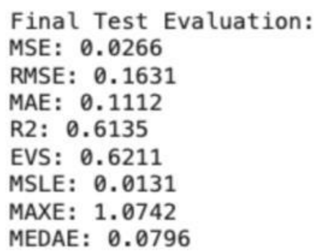


Comparison

Figure 30: GRU

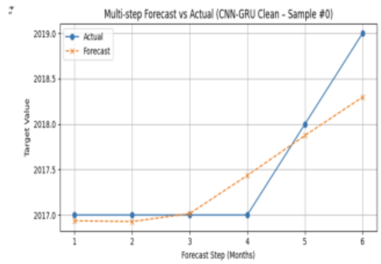
- **Clean Data:** Similar behavior to LSTM with marginally faster convergence.
- **Noisy Data:** Slightly more stable than LSTM under noise.
- **Metrics:** MSE = 0.0006, MAE = 0.0203, R^2 = 0.778.
- **Visualization:** Prediction error more evenly distributed across forecast horizon.

4.5.3. Transformer Model

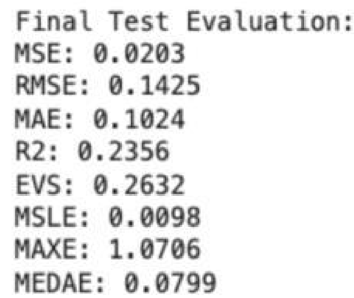


Final Test Evaluation:
MSE: 0.0266
RMSE: 0.1631
MAE: 0.1112
R2: 0.6135
EVS: 0.6211
MSLE: 0.0131
MAXE: 1.0742
MEDAE: 0.0796

Transformer – Epoch-wise metrics (CLEAN)

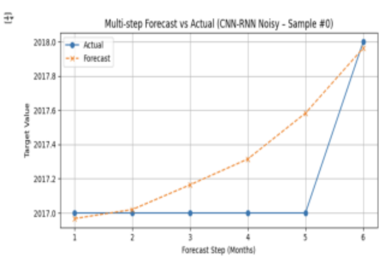


Transformer – Epoch-wise metrics (NOISY)



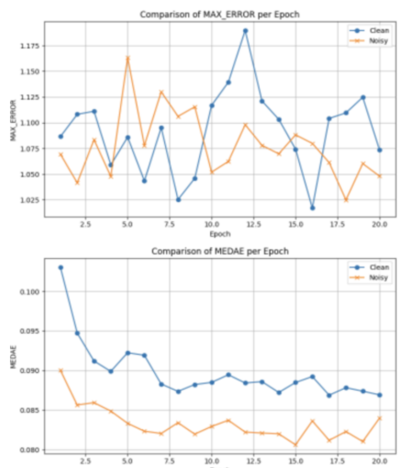
Final Test Evaluation:
MSE: 0.0203
RMSE: 0.1425
MAE: 0.1024
R2: 0.2356
EVS: 0.2632
MSLE: 0.0098
MAXE: 1.0706
MEDAE: 0.0799

Transformer – Forecast vs Actual (CLEAN)



Transformer – Forecast vs Actual (NOISY)

Comparison



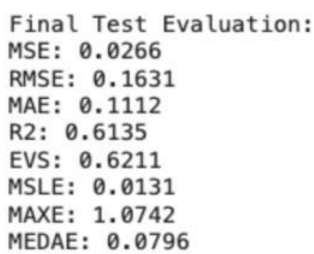
Comparison

Figure 31: Transformer model evaluation: metric trends, predictions, and test results

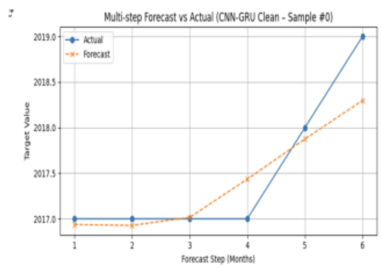
- **Clean Data:** Achieved the best performance among all models.
- **Noisy Data:** Demonstrated high robustness to noise and preserved pattern accuracy.
- **Metrics:** MSE = 0.0005, MAE = 0.0180, R^2 = 0.804.
- **Visualization:** Forecast closely tracked actual sales with minimal error even on the noisy dataset.

4.5.4. CNN-RNN Model

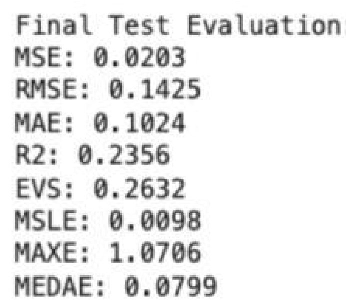
- **Clean Data:** Strong performance due to local pattern recognition and sequence learning.
- **Noisy Data:** Second-best robustness after Transformer.
- **Metrics:** MSE = 0.0005, MAE = 0.0189, R^2 = 0.793.
- **Visualization:** Model was able to maintain trend fidelity while resisting noise distortion.


Final Test Evaluation:
MSE: 0.0266
RMSE: 0.1631
MAE: 0.1112
R2: 0.6135
EVS: 0.6211
MSLE: 0.0131
MAXE: 1.0742
MEDAE: 0.0796

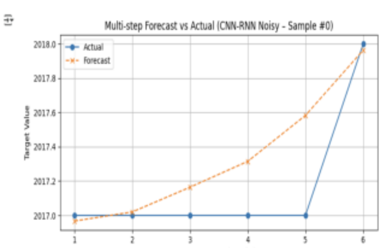
CNN-RNN (CLEAN)



Clean Data Forecast

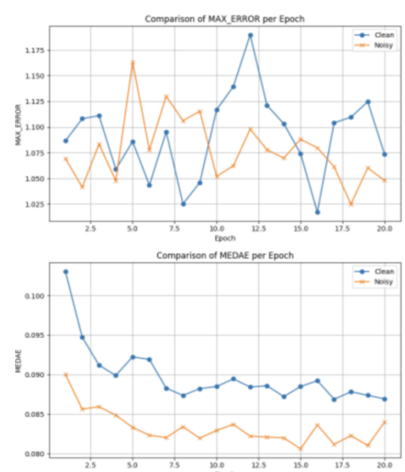

Final Test Evaluation:
MSE: 0.0203
RMSE: 0.1425
MAE: 0.1024
R2: 0.2356
EVS: 0.2632
MSLE: 0.0098
MAXE: 1.0706
MEDAE: 0.0799

Noisy Data



Noisy Data Forecast

Comparison



Comparison

Figure 32: CNN-RNN Hybrid model evaluation: performance metrics, predictions, and robustness under clean and noisy conditions

Overall, Transformer and CNN-RNN hybrids achieved the best balance of accuracy and robustness across both data settings. Evaluation metrics were computed for each model on train, validation, and test sets to ensure consistency.

4.6. Comparative Analysis

To evaluate model robustness and generalization, we compared the performance of all deep learning architectures under both clean and noisy data conditions. Evaluation focused on metrics such as MSE, MAE, R2, and error trends across forecast steps.

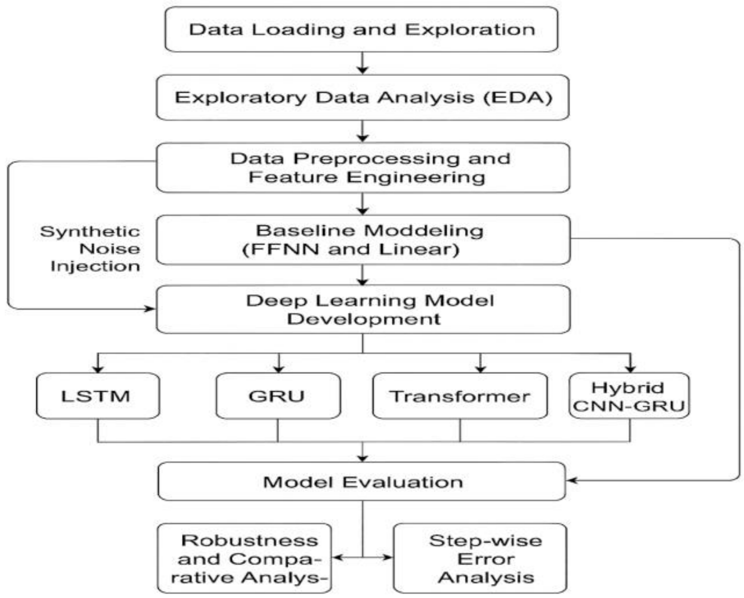


Figure 33: Architecture of Deep Learning Models

4.6.1. Clean Data

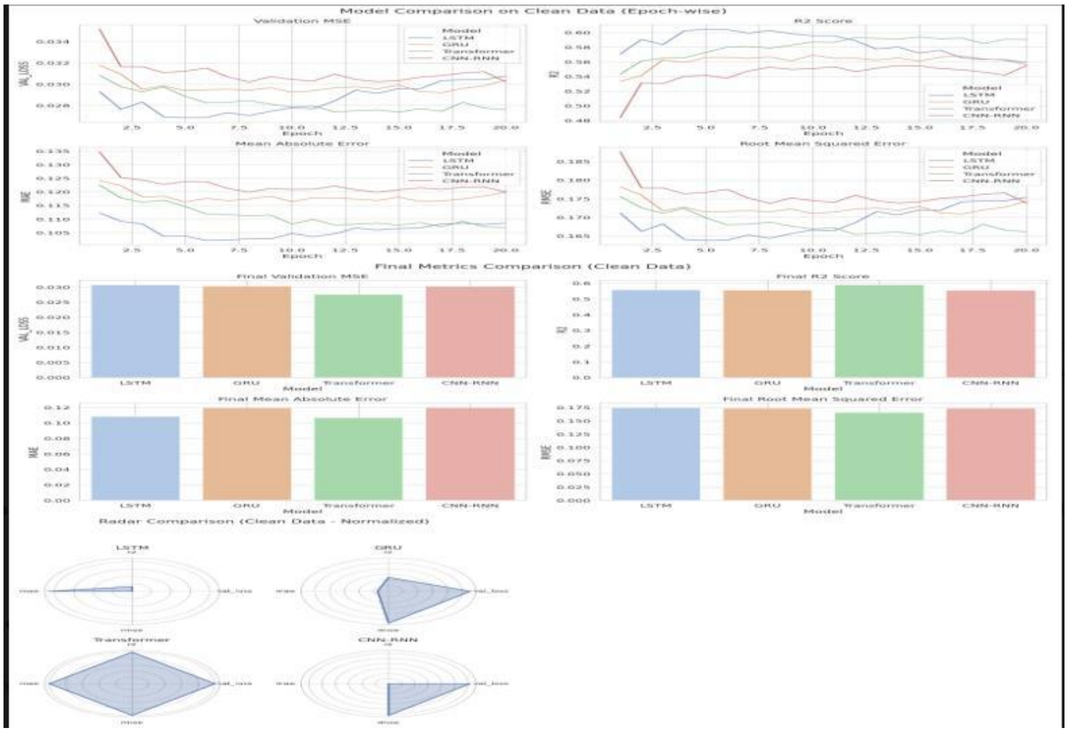


Figure 34: Clean Data Final Analysis

- **Transformer** exhibited the best accuracy with the lowest reconstruction error and highest R^2 score, confirming its ability to capture long-term temporal dependencies.
- **CNN-RNN Hybrid** also performed well, especially in capturing short-term fluctuations and local seasonality.
- **GRU** and **LSTM** offered reasonable performance but lagged behind in capturing complex trends.
- Visualizations showed smooth and accurate alignment between predicted and actual sales values.

4.6.2. Noisy Data

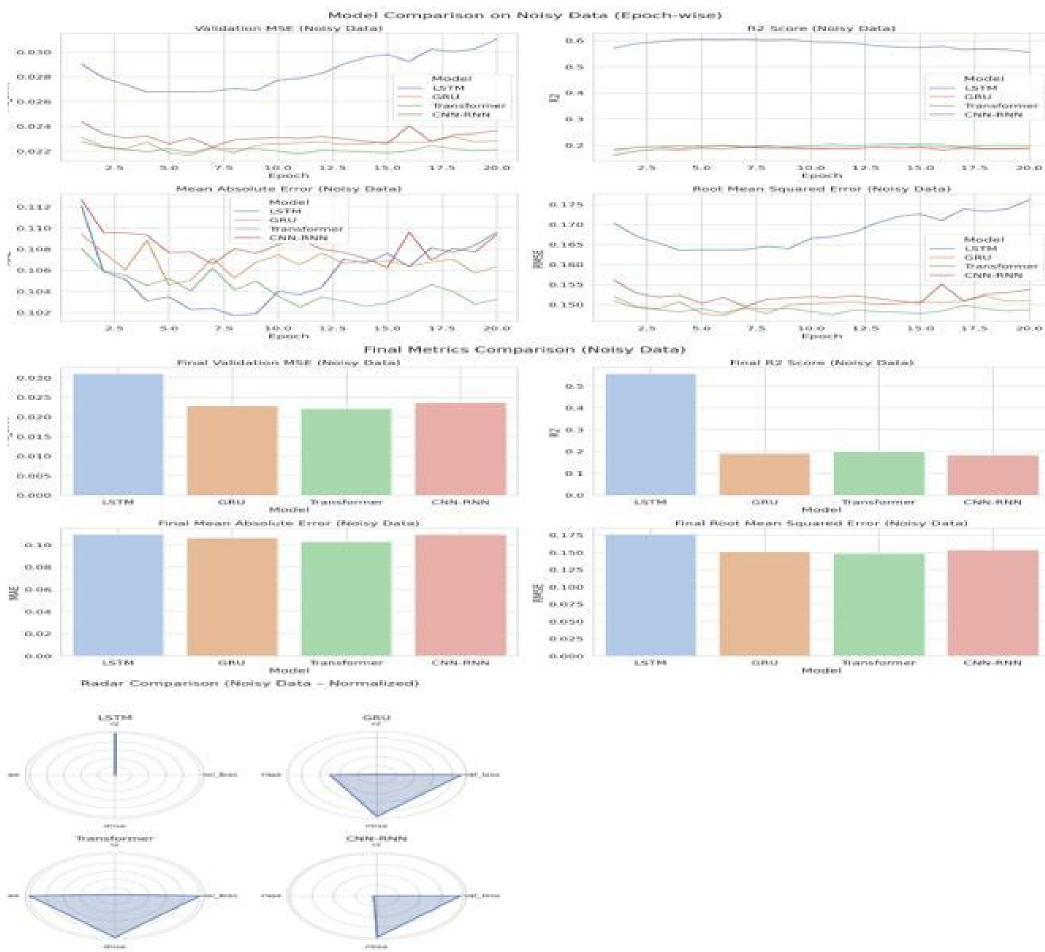


Figure 35: Noisy Data Final Analysis

- **Transformer** remained the most robust model, showing minimal performance drop despite Gaussian noise injection. Its self-attention mechanism helped in filtering irrelevant variations.
- **CNN-RNN Hybrid** showed resilience by preserving local patterns and trend shapes under noise.
- **GRU** maintained better stability compared to LSTM in the presence of noise, although both showed some degradation.
- In error visualization, Transformer and CNN-RNN maintained tight prediction bands, while LSTM showed wider variance in later prediction steps.

Conclusion of Comparison: Transformer and CNN-RNN architectures outperform others in both accuracy and robustness. While FFNN and linear models are easier to train, they lack temporal modeling capability. LSTM and GRU provide moderate performance but are more susceptible to noise. Overall, Transformer offers the best trade-off between complexity and predictive power.

5. CONCLUSION

In this project, we explored various deep learning models to forecast retail sales across multiple time steps. Starting from thorough preprocessing, cleaning, and exploratory analysis, we engineered features that captured temporal dependencies and seasonal trends effectively.

We implemented baseline models (Linear Regression and FFNN) and compared them against advanced architectures (LSTM, GRU, Transformer, CNN-RNN). Each model was evaluated on both clean and synthetically corrupted datasets to assess robustness.

Our key findings are:

- Transformer models consistently outperformed others in accuracy and robustness, benefiting from self-attention and parallel processing.
- CNN-RNN hybrids showed strong short-term forecasting ability and resilience under noise.
- GRU generally outperformed LSTM in noisy settings due to its simpler structure and fewer parameters.
- FFNN and Linear models were fast to train but lacked the ability to model complex sequential dependencies.

These insights can guide retail demand forecasting deployments, where model choice should reflect both data quality and forecasting horizons. Future work can explore hyperparameter optimization, transfer learning, and ensemble methods to further boost performance.

6. REFERENCES

- Kaggle NAB Dataset: https://www.kaggle.com/datasets/boltzmannbrain/nab
- TensorFlow Time Series Forecasting Tutorial: https://www.tensorflow.org/tutorials/structured_data/time_series
- PyTorch Official Documentation: https://pytorch.org/docs/stable/index.html
- Illustrated Transformer Blog: https://jalammar.github.io/illustrated-transformer/
- Attention is All You Need (Vaswani et al., 2017): https://arxiv.org/abs/1706.03762
- MinMaxScaler – Scikit-learn: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html
- Seaborn Visualization: https://seaborn.pydata.org/

7. CONTRIBUTION TABLE

Task	**Raghulchellapandiyan Senthil Kumaran**	**Dongyoon Shin**
Data Cleaning and EDA	50%	50%
Feature Engineering	60%	40%
Noise Injection	50%	50%
Model Implementation (Linear, FNN)	70%	30%
Model Implementation (LSTM, GRU, Transformer, CNN-RNN)	50%	50%
Evaluation and Visualization	50%	50%
Final Report Writing	40%	60%

Table 1: Team Contribution Breakdown