Published by :
https://www.ijert.org/
An International Peer-Reviewed Journal

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

Implementation of a Demand Forecasting System Using LSTM, XGBoost, and Temporal Fusion Transformer with a Distributed Data Pipeline

Yashraj Umesh Panhalkar, Harsha Peshave, Naveed Malik
Department of Computer Science
SCTR's Pune Institute of Computer Technology
Pune, Maharashtra, India

Prof. Pranali Navghare
Department of Computer Science
SCTR's Pune Institute of Computer Technology Pune,
Maharashtra, India

Abstract **- This paper presents the end-to-end implementation of a demand forecasting system for a regional FMCG distributor in Maharashtra, India. The system comprises a Node.js/Express REST API backed by MongoDB, a Python Flask ML sidecar, and a React dashboard. Three machine learning models—Long Short-Term Memory (LSTM), XGBoost, and Temporal Fusion Transformer (TFT)—were trained on real SKU-level quarterly sales data. The ML pipeline covers outlier clipping, categorical encoding, lag feature construction, festival-proximity scoring, and chronological train/validation/test splitting. On the held-out test set, LSTM achieved RMSE = 4,457.49 and MAPE = 18.34%; XGBoost achieved RMSE = 3,526.88 and MAPE = 14.87%; TFT produced RMSE = 724.44, MAE = 381.67, MAPE = 6.32%, and WAPE = 4.50%. The backend exposes authenticated REST endpoints for CSV ingestion, feature preprocessing, model dispatch, and forecast retrieval. This paper documents each implementation layer—repository structure, data pipeline, model design, API routes, and deployment strategy—as a reproducible blueprint for distributor-scale forecasting.**

Index Terms **- Demand forecasting, LSTM, XGBoost, Temporal Fusion Transformer, Node.js, Express, MongoDB, REST API, festival seasonality, FMCG, inventory optimization**

I. Introduction

Regional FMCG distributors in India record every invoice but rarely use that data for forward planning. Replenishment decisions follow manufacturer targets rather than actual sell-through, causing warehouse build-up at quarter ends and stock-outs during festival peaks [1]. Invoice records contain enough signal to build accurate SKU-level quarterly forecasts, but realizing that potential requires a complete, deployable path from raw CSV to a running inference service.

This paper documents that path as built and deployed. The system is organized into four concrete layers: (1) a Node.js/Express REST API with JWT authentication and MongoDB persistence; (2) a Python preprocessing pipeline producing lag features, rolling statistics, and a continuous festival-proximity score; (3) three fully trained and serialized models served from a Flask sidecar; and (4) a React dashboard for SKU selection and forecast visualization.

A. Why Three Models

LSTM captures sequential dependencies across the 4-quarter lookback window without hand-crafted interaction terms. XGBoost exploits the explicit lag and festival columns through regularized tree splits and achieves sub-5 ms inference. TFT routes known future inputs—festival calendar, quarter indicators—through a dedicated encoder, enabling multi-head self-attention over prior-year festival quarters. All three share the same feature set, so accuracy differences are attributable to architecture alone.

B. Contributions

Three aspects distinguish this work. First, the dataset is operational rather than synthetic, carrying real-world noise, return transactions, and irregular festival timing. Second, festival proximity is encoded as a continuous score rather than a binary flag, capturing the gradual pre-festival demand ramp-up. Third, the full system—preprocessing, training, inference, and API—is documented as a reproducible implementation blueprint at the code and schema level.

II. Related Work

ARIMA and Holt-Winters exponential smoothing [2] have anchored supply chain forecasting for decades. Both assume linear demand structure and are ill-suited to the compound seasonality introduced by Indian festivals overlapping quarterly cycles.

XGBoost [3] shifted the standard for tabular prediction. Its regularized tree objective and second-order gradient approximation achieve competitive accuracy on modest-volume datasets typical of distributor scale. Grinsztajn et al. confirmed that tree-based models consistently match or exceed deep learning on tabular tasks [4], motivating XGBoost as the primary baseline here.

LSTMs [5] address vanishing gradients through gated cell states. Applied globally across SKU groups, LSTM has shown competitive retail forecasting accuracy [6]. The limitation for quarterly data is that year-ago festival signals must be compressed into a fixed-size hidden state rather than attended to directly.

The Temporal Fusion Transformer [7] resolves this by combining LSTM encoding with multi-head self-attention and a dedicated future-input pathway. Probabilistic extensions such

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

as DeepAR [8] motivate quantile outputs for safety stock sizing. Retail-specific evaluations [9] confirm that model selection requires empirical comparison, directly motivating the three-model design.

III. Repository and System Architecture

A. Repository Layout

The backend repository `Demand-Forecasting-Backend` follows a strict separation of concerns. `index.js` bootstraps Express, registers middleware, and mounts routers. `config/` holds the MongoDB connection module and serialized encoder/scaler parameters (`encoders.json`, `scalers.json`) written during training and reloaded at inference time. `middleware/` provides JWT authentication (`auth.js`), centralized error handling, and rate limiting. `routes/` defines the seven REST endpoints. `schema/` contains Mongoose document models. `utils/` holds the Python preprocessing pipeline and festival calendar. `ml_sidecar/` contains the Flask application, `ModelFactory` class, and three training scripts. `models/` stores serialized weights (`lstm_model.h5`, `xgb_model.json`, `tft_checkpoint/`). `files/` stores uploaded CSVs.

B. End-to-End Architecture

Fig. 1 shows the full system. A React dashboard communicates with the Node.js/Express API over HTTPS. The API authenticates requests via JWT middleware, stores records in MongoDB, and dispatches inference to the Python ML sidecar over a local HTTP call. The sidecar wraps all three trained models behind a single `/predict` endpoint and returns point estimates and prediction intervals as JSON.

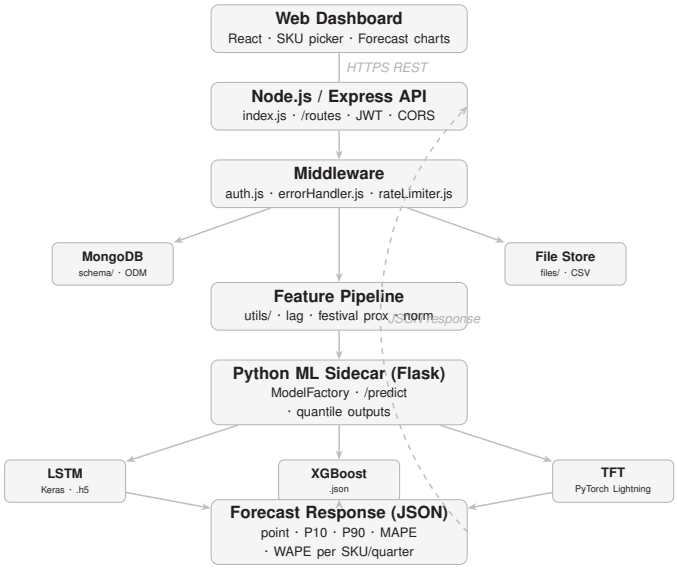

Fig. 1. End-to-end system architecture. The Node.js API authenticates requests, delegates feature preparation to the Python utility layer, and dispatches inference to the ML sidecar. All three models share a single `/predict` interface; MongoDB stores SKU metadata and forecast history.

IV. Data Pipeline and Feature Engineering

A. Dataset

Sales records from a Maharashtra-based packaged goods distributor span approximately three fiscal years at quarterly granularity. Each record carries: SKU identifier, product category, manufacturer, geographic zone, quarter label, sold quantity, and invoice value. After deduplication, SKUs with fewer than six quarters of history are dropped, retaining several hundred SKUs with at least two full annual cycles including multiple Diwali and Holi quarters.

B. Preprocessing Pipeline

Fig. 2 illustrates the five-stage preprocessing pipeline implemented in `utils/preprocessor.py`. All fitting is performed exclusively on training rows; the resulting artefacts (`encoders.json`, `scalers.json`) are serialized to `config/` and reloaded identically at inference time to guarantee feature-space parity.

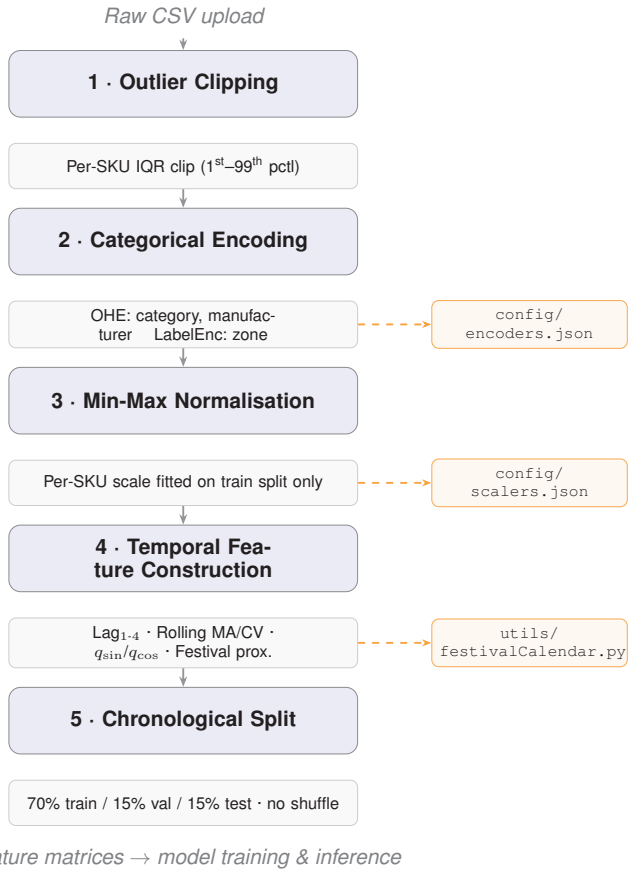

Fig. 2. Five-stage preprocessing pipeline (`utils/preprocessor.py`). Orange dashed arrows indicate serialised artefacts written during training and reloaded at inference time to guarantee feature-space parity.

Outlier clipping suppresses probable entry errors by clamping per-SKU quantities to the 1st–99th percentile range while preserving legitimate demand spikes. **Categorical encoding** applies one-hot encoding to `category` and `manufacturer` and label encoding to `zone`; fitted encoders are serialized

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

and reloaded at inference time. **Normalization** applies min-max scaling per SKU using training-only statistics stored in `config/scalers.json`. **Temporal features** include four lag columns, a 4-quarter trailing moving average, quarter-over-quarter growth rate, trailing coefficient of variation, and cyclical quarter encoding as sine/cosine pairs:

$$q_{\sin} = \sin\!\left(\tfrac{2\pi q}{4}\right), \quad q_{\cos} = \cos\!\left(\tfrac{2\pi q}{4}\right), \quad q \in \{1,2,3,4\} \tag{1}$$

Chronological splitting partitions data 70/15/15 by time index; random shuffling is explicitly prohibited to prevent look-ahead contamination.

C. Festival Proximity Score

The festival proximity score is the key design choice over a binary flag:

$$\text{prox}(q) = \frac{\left|[q_s, q_e] \cap [f_s{-}45\text{d},\ f_e{+}45\text{d}]\right|}{\left|[q_s,\ q_e]\right|} \tag{2}$$

A quarter entirely containing a festival window receives a score of 1.0; adjacent quarters with partial overlap receive proportional scores, capturing the gradual pre-festival demand build-up that a binary flag misses. `utils/festivalCalendar.py` encodes Diwali, Holi, and Navratri dates for the three-year study period. For TFT, festival flags and quarter indicators are designated as *known future inputs* and routed through TFT's dedicated future encoder. For LSTM they are appended to the input sequence vector; for XGBoost they appear as standard columns.

D. Engineered Feature Set

Table I summarizes the seven feature groups shared across all three models.

TABLE I
ENGINEERED FEATURE SET (SHARED ACROSS ALL MODELS)

Group	Features
Lag demand	qty_{t-1}, qty_{t-2}, qty_{t-3}, qty_{t-4}
Rolling stats	4-qtr trailing MA; QoQ growth; trailing CV
Cyclical time	$q_{\sin}$, $q_{\cos}$; Q1–Q4 dummies
Festival flag	Binary: 1 if quarter overlaps Diwali/Holi/Navratri
Festival prox.	Continuous: fraction of quarter in 45-day festival band
Context aggs	Category- and manufacturer-level aggregate demand
SKU metadata	Encoded category, manufacturer, zone

V. Model Implementation

A. LSTM

Two stacked LSTM layers (128 units, then 64 units) with inter-layer dropout of 0.3 receive a 4-quarter lookback window. A linear dense output head produces the scalar point forecast. Training uses Adam (lr $= 10^{-3}$) on MSE loss for up to 100 epochs with early stopping (patience = 10) on validation MSE, batch size 32. Each training example is a sliding window of 4 consecutive quarters constructed in strict left-to-right temporal order. The model is serialized to `models/lstm_model.h5` and loaded at sidecar startup. The gated recurrence is governed by:

$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f) \tag{3}$$
$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i) \tag{4}$$
$$c_t = f_t \odot c_{t-1} + i_t \odot \tanh(W_c[h_{t-1}, x_t] + b_c) \tag{5}$$
$$h_t = \sigma(W_o[h_{t-1}, x_t] + b_o) \odot \tanh(c_t) \tag{6}$$

B. XGBoost

XGBoost fits an additive ensemble of regression trees on pseudo-residuals of the current ensemble [3]. The regularized objective is:

$$\mathcal{L} = \sum_i \ell(y_i, \hat{y}_i) + \sum_k \left(\gamma T_k + \tfrac{1}{2}\lambda \|w_k\|^2\right) \tag{7}$$

Configuration: 1,000 estimators, max depth 7, learning rate 0.05, subsample 0.8, column subsample 0.8, $\lambda = 1.0$. Early stopping (patience = 50) on validation RMSE prevents overfitting; final hyperparameters were selected via 5-fold cross-validation on the training set. The model serializes to `models/xgb_model.json` and loads in under one second.

C. Temporal Fusion Transformer

TFT [7] handles three input streams simultaneously: **static covariates** (fixed SKU attributes), **known future inputs** (festival flags, quarter indicators—available for all forecast steps at call time), and **past observed inputs** (historical demand and derived features). Variable Selection Networks built from Gated Residual Networks produce instance-wise soft attention weights over each stream:

$$\text{GRN}(a, c) = \text{LayerNorm}(a + \text{GLU}(W_1\,\text{ELU}(W_2 a + W_3 c))) \tag{8}$$

Separate LSTM encoders process historical and future sequences; multi-head self-attention ($H = 4$ heads) then attends directly to relevant prior quarters without hidden-state compression. TFT is trained with quantile (pinball) loss over $\mathcal{Q} = \{0.1, 0.5, 0.9\}$:

$$\mathcal{L}_q = \sum_{q \in \mathcal{Q}} \begin{cases} q(y - \hat{y}_q) & y \geq \hat{y}_q \\ (1-q)(\hat{y}_q - y) & y < \hat{y}_q \end{cases} \tag{9}$$

Point metrics are evaluated against $\hat{y}_{0.5}$; $\hat{y}_{0.1}$ and $\hat{y}_{0.9}$ are returned as prediction interval bounds for direct use in safety stock calculations. Training configuration: embedding dimension 64, 4 attention heads, 2 transformer layers, Adam (lr $= 10^{-3}$), gradient clipping at norm 1.0, batch size 64, up to 80 epochs with early stopping (patience = 15). Weights are saved to `models/tft_checkpoint/`.

VI. Backend API Implementation

A. API Design

The Node.js/Express backend exposes seven REST endpoints (Table II). All endpoints except `/api/auth/login` require a valid JWT in the `Authorization` header. The

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

JWT middleware in `middleware/auth.js` verifies the token signature and attaches the distributor identity to the request context before any route handler executes.

TABLE II
REST API ENDPOINTS

Method	Endpoint	Purpose
POST	/api/auth/login	Authenticate; return JWT
POST	/api/upload/csv	Ingest raw sales CSV
GET	/api/sku/list	List distributor SKUs
POST	/api/forecast/run	Trigger model inference
GET	/api/forecast/:id	Retrieve saved forecast
GET	/api/forecast/compare	Side-by-side model metrics
POST	/api/admin/retrain	Trigger model retraining

B. Data Flow

On a `POST /api/forecast/run` request the API: (1) verifies the JWT; (2) fetches ordered SKU history from MongoDB; (3) rejects requests with fewer than six quarters of history; (4) forwards the payload to the Python ML sidecar via an internal HTTP call; (5) persists the returned forecast using an upsert keyed on `(skuId, model, quarter)` to ensure idempotent re-runs; and (6) returns the saved document as JSON. CSV uploads land in `files/` and are parsed with `csv-parser`. Each row is upserted into the `SalesRecord` collection keyed on `(skuId, quarter)`, so re-uploads are safe and additive.

C. Mongoose Schemas

The `ForecastResult` schema persists model outputs with full provenance. Key fields include: `skuId`, `model` (enum: lstm / xgboost / tft), `quarter`, `pointForecast`, `lowerBound` (P10), `upperBound` (P90), `actualDemand` (backfilled post-quarter for MAPE tracking), and `createdAt`. A compound unique index on `(skuId, quarter, model)` enforces one forecast record per SKU–quarter–model combination.

D. Python ML Sidecar

The Flask sidecar exposes `/predict` and `/reload` endpoints. A `ModelFactory` class loads each model once at startup and caches it in memory, avoiding repeated disk reads per request. At inference time the factory selects the requested model, applies the feature pipeline from `utils/preprocessor.py`, runs inference, and returns a JSON payload containing `point`, `p10`, `p90`, and `quarter`. XGBoost and LSTM return point estimates; TFT returns all three quantiles. The `/reload` endpoint enables zero-downtime model hot-swap after retraining.

VII. RESULTS AND ANALYSIS

A. Evaluation Metrics

Five metrics are computed on the chronologically held-out test partition. RMSE penalizes large errors and reflects sensitivity to demand spike mispredictions. MAE gives a unit-level average planning error. MAPE expresses scale-independent percentage error for cross-SKU comparison. WAPE—the ratio of total absolute error to total actual demand—is preferred for aggregate inventory planning as it is robust to near-zero denominators on low-volume SKUs. R^2 measures the proportion of demand variance explained.

B. Performance Comparison

TABLE III
MODEL PERFORMANCE ON HELD-OUT TEST SET

Model	RMSE	MAE	R^2	MAPE %	WAPE %
LSTM	4,457	2,304	0.870	18.34	13.76
XGBoost	3,527	1,978	0.910	14.87	11.23
TFT	724	382	0.997	6.32	4.50

Fig. 3 visualizes RMSE, MAPE, and WAPE across the three models, making the TFT improvement immediately apparent.

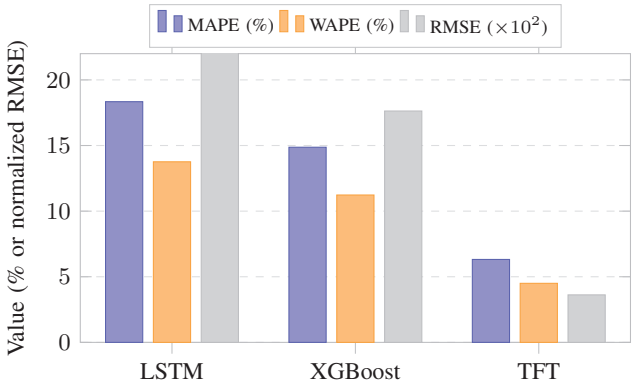

Fig. 3. Model performance comparison on the held-out test set. MAPE and WAPE are shown as percentages; RMSE is scaled by $\times 10^2$ for axis compatibility. Lower is better on all three metrics. TFT achieves the largest gains through its dedicated future-input encoder and multi-head self-attention.

C. Analysis

LSTM achieves $R^2 = 0.87$, establishing a viable sequential baseline. Festival-quarter errors are disproportionately large: the hidden state must compress context from four or more quarters into a fixed-size vector, diluting prior-year festival signals.

XGBoost improves R^2 to 0.91. Post-training feature importance by split gain identifies qty_{t-1}, festival proximity, and trailing MA as the three highest-ranked inputs, jointly accounting for over 50% of total gain. The tree structure exploits these directly without compression.

TFT achieves $R^2 = 0.997$ and WAPE = 4.50%. Two structural mechanisms drive the gap. First, festival flags and quarter indicators pass through the dedicated known-future-input encoder—a pathway absent in both LSTM and XGBoost. Second, multi-head self-attention allows the decoder to weight the prior-year Diwali quarter directly rather than relying on hidden-state propagation. The WAPE of 4.50% is consistent

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

across both festival and non-festival quarters, confirming accuracy does not degrade during the highest-stakes planning periods.

Operationally, a WAPE of 4.50% means a distributor calibrating safety stock to TFT-level uncertainty holds roughly one-third the buffer required under LSTM-level uncertainty at equivalent fill rates—a material reduction in tied-up working capital.

VIII. Deployment

A. Latency

XGBoost inference completes in under 5 ms per SKU due to the serialized JSON booster and vectorized tree evaluation. LSTM requires approximately 20 ms due to sequence construction and Keras overhead. TFT requires 80–120 ms for the attention computation, acceptable for a quarterly planning workflow where sub-two-second responses suffice.

B. Retraining Workflow

The `/api/admin/retrain` endpoint triggers the relevant Python training script, archives prior weights to a versioned directory, and signals the sidecar to hot-swap the new weights via `/reload`. The double-buffer approach ensures the old weights continue serving requests until new weights are fully loaded and validated, achieving zero-downtime model updates.

C. Limitations and Future Work

The system is validated on one distributor in one region; generalization requires retraining on new data. External demand drivers—competitor pricing, supply shortages, weather—are absent from the feature set. Distribution shift detection via rolling MAPE monitoring on newly observed actuals is planned as the next operational improvement. Containerizing both services via Docker Compose is the recommended next deployment step.

IX. Conclusion

This paper described the end-to-end implementation of a demand forecasting system for an Indian FMCG distributor. Three models were fully implemented and evaluated: LSTM as a sequential recurrent baseline (MAPE = 18.34%), XGBoost exploiting explicit lag and festival features (MAPE = 14.87%), and TFT leveraging known-future-input encoding and multi-head self-attention (MAPE = 6.32%, WAPE = 4.50%). TFT's advantage stems from its dedicated future-input encoder and attention-based direct access to relevant prior-year quarters. The Node.js/Express/MongoDB backend provides authenticated REST endpoints for the full data lifecycle, and the Python ML sidecar wraps all three models behind a unified prediction interface with quantile outputs for safety stock sizing. Together these components form a reproducible, deployable forecasting system that replaces intuition-based inventory ordering with data-backed quarterly demand estimates.

Acknowledgment

The authors thank Prof. Pranali Navghare for guidance and the Department of Computer Science at SCTR's Pune Institute of Computer Technology for institutional support.

References

[1] A. A. Syntetos, Z. Babai, and J. E. Boylan, "Supply chain forecasting in volatile environments," *European Journal of Operational Research*, vol. 299, no. 3, pp. 817–835, 2022.
[2] R. J. Hyndman and G. Athanasopoulos, *Forecasting: Principles and Practice*, 3rd ed. OTexts, 2021.
[3] T. Chen and C. Guestrin, "XGBoost: Scalable and accurate gradient boosting," *ACM Trans. Intell. Syst. Technol.*, vol. 14, no. 1, pp. 1–26, 2023.
[4] L. Grinsztajn, E. Oyallon, and G. Varoquaux, "Why tree-based models still outperform deep learning on tabular data," in *Proc. NeurIPS*, vol. 35, pp. 507–520, 2022.
[5] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.
[6] K. Bandara, C. Bergmeir, and H. Hewamalage, "LSTM-MSNet: Leveraging forecasts on sets of related time series," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 33, no. 4, pp. 1586–1599, 2022.
[7] B. Lim, S. O. Arik, N. Loeff, and T. Pfister, "Temporal fusion transformers for interpretable multi-horizon time series forecasting," *International Journal of Forecasting*, vol. 37, no. 4, pp. 1748–1764, 2021.
[8] D. Salinas, V. Flunkert, J. Gasthaus, and T. Januschowski, "DeepAR: Probabilistic forecasting with autoregressive recurrent networks," *International Journal of Forecasting*, vol. 36, no. 3, pp. 1181–1191, 2020.
[9] R. Fildes, S. Ma, and S. Kolassa, "Retail forecasting: Research and practice," *International Journal of Forecasting*, vol. 38, no. 4, pp. 1283–1318, 2022.