Demand Forecasting

Multiplicative SARIMA from Scratch

Diego Coglievina Díaz   Juan Jesús Orihuela Torres

Universidad Anáhuac Querétaro

December 13, 2025

Repository and Deliverables

- Code repository: `https://github.com/diegocoglievina/Analitica-Avanzada--Proyecto-Final--SARIMA-`
- Reproducibility: notebooks for EDA, production scripts for training and inference, plus requirements.
- MLOps: MLflow for tracking experiments and serving artifacts.
- Deployment: Flask API for inference.

Agenda

Roadmap

Operating Landscape

- Multi-echelon supply chain with three channels: OEM, Dealer (Aftermarket), Professional End-Users.
- Dealer channel is the most volatile, OEM is contract-driven but penalizes stockouts.
- Forecast error has direct cost: overstock immobilizes capital, understock causes lost sales and penalties.

Problem and Value Proposition

- Objective: robust monthly forecasts that capture both seasonality and volatility.
- We forecast **Sales** and **Returns** separately to expose net cash-relevant flow.
- We implement multiplicative SARIMA with explicit interaction lags, plus L2 regularization.
- Model selection: grid search + rolling-origin cross-validation before deployment.

Objective

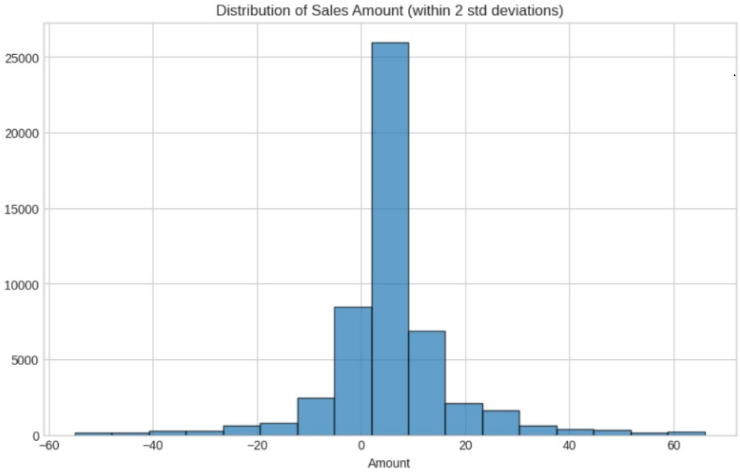
We do not only predict "how much will be sold". We also predict "how much comes back", and net it in a controlled way.

What We Built (End-to-End)

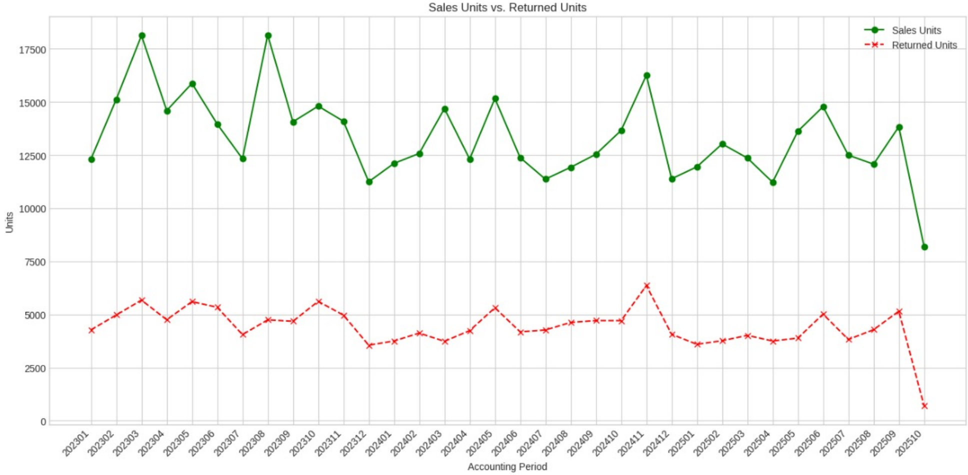
1. Convert daily transactions into a clean monthly panel (Sales vs Returns).
2. Transform series for stationarity (log, seasonal differencing, regular differencing).
3. Train multiplicative SARIMA from scratch with explicit polynomial expansion.
4. Validate using rolling-origin CV and compare to a seasonal naive baseline.
5. Register best models in MLflow and serve them via a Flask API.

Roadmap

Distribution of Transactions



Aggregated Monthly Series



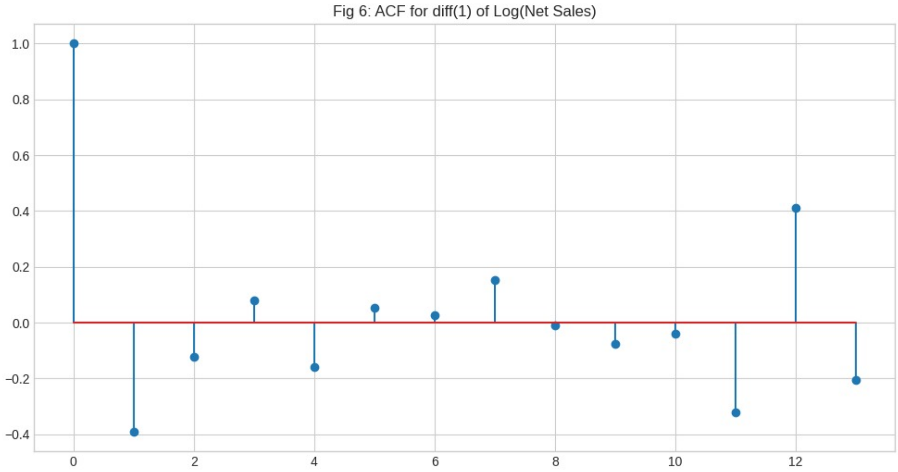
ACF by Segment (Seasonality Evidence)

- Dealer ACF: strong spikes at multiples of 12 plus slower decay (high persistence and volatility).
- OEM ACF: seasonal spikes with less noise (often lower-order models work).
- Professional End-Users ACF: seasonality present, lower variance than Dealer.

In simple terms

Spikes at 12, 24, 36 mean "this month is related to the same month last year".

ACF after Log Transform



- Variance becomes more stable, but seasonal memory persists.

Roadmap

From Transactions to Monthly Panel

- Filter to `Account == "Sales Units"` and remove zero `Amount`.
- Split by sign:
  - `net_sales_units`: sum of positive transactions.
  - `returns_units`: sum of absolute value of negative transactions.
- Aggregate monthly per segment and build a continuous month index.
- Fill missing months with zeros so the time axis is complete.

In simple terms

We convert a messy event stream into clean monthly signals that the model can interpret as "one value per month".

Transforms for Stationarity (Why and How)

- **Shifted log:** $y_t' = \log(y_t + c)$ to stabilize variance.
- **Seasonal differencing:** remove repeating annual level $(S = 12)$.
- **Regular differencing:** remove residual trend if present.
- **Diagnostics:** ACF/PACF + ADF/KPSS to avoid under or over differencing.

In simple terms

The goal is to transform the series into a form where its average and variability look stable over time.

ADF and KPSS (Complementary Diagnostics)

- **ADF**: null hypothesis = unit root (non-stationary). Small p suggests stationarity after differencing.
- **KPSS**: null hypothesis = stationary. Large p supports stationarity.
- Using both reduces the risk of wrong differencing order.

In simple terms

One test asks: “is it still non-stationary?” The other asks: “can we treat it as stationary now?”

Roadmap

Objects and Notation

- Observed series $\{y_t\}$ is one realization of a stochastic process $\{Y_t\}$.
- Backshift operator B: $BY_t = Y_{t-1}$, $B^k Y_t = Y_{t-k}$.
- Differencing is just subtraction with this operator: $\nabla = 1 - B$.

In simple terms

B means "go back in time". Differencing means "compare to the past".

White Noise and Innovations (The Driving Randomness)

- Innovations $\{w_t\}$ are modeled as i.i.d. with

$$\mathbb{E}[w_t] = 0, \qquad \mathrm{Var}(w_t) = \sigma_w^2, \qquad \mathrm{Cov}(w_t, w_{t-k}) = 0 \ (k \neq 0).$$

- Often assumed Gaussian for likelihood-based estimation, but the core idea is "uncorrelated shocks".

In simple terms

Mean 0 means shocks have no systematic direction. Constant variance means shocks have roughly the same typical size over time. Zero covariance means shocks do not persist by themselves.

AR, MA, ARMA (Why They Work)

- AR(p): today depends on past values plus a new shock.
- MA(q): today depends on recent shocks, which creates short memory.
- ARMA combines both to match richer autocorrelation shapes.

$$\phi_p(B)(X_t - \mu) = \theta_q(B)w_t$$

In simple terms

AR explains persistence using past values. MA explains persistence using delayed impact of shocks.

ARIMA: Making ARMA Work for Trend

- Real data often has trend, so ARMA on Y_t fails because mean is not stable.
- ARIMA applies differencing first:

$$X_t = \nabla^d Y_t = (1 - B)^d Y_t,$$

and then fits ARMA to X_t.

In simple terms

If the series keeps drifting upward or downward, we model the changes between months instead of raw levels.

SARIMA: Extending ARIMA with Seasonality

- Monthly seasonality means strong dependence at lag $S = 12$.
- Seasonal differencing:

$$\nabla_{12} Y_t = (1 - B^{12}) Y_t = Y_t - Y_{t-12}.$$

- Multiplicative SARIMA combines seasonal and non-seasonal polynomials:

$$\Phi_P(B^{12})\, \phi_p(B)\, (X_t - \mu) = \Theta_Q(B^{12})\, \theta_q(B)\, w_t.$$

In simple terms

Seasonal differencing asks: "how different is this month from the same month last year?"

Why Interaction Lags Appear (Key Implementation Detail)

$$(1 - \phi_1 B)(1 - \Phi_1 B^{12}) = 1 - \phi_1 B - \Phi_1 B^{12} + \phi_1 \Phi_1 B^{13}.$$

- Multiplication creates a cross-lag at $13 = 1 + 12$.
- Our implementation explicitly includes these induced lags instead of ignoring them.

In simple terms

"One month effect" and "one year effect" combine into "one month after one year" automatically.

Roadmap

Training Objective (What We Optimize)

- We estimate coefficients by minimizing one-step prediction error.
- Loss with L2 regularization:

$$J(\boldsymbol{\Omega}) = \frac{1}{2}\sum_t e_t^2 + \frac{\lambda}{2}\|\boldsymbol{\Omega}\|_2^2.$$

In simple terms

We want small prediction errors, but we also penalize overly large coefficients to avoid unstable forecasts.

SGD Update (How Parameters Move)

$$\mathbf{\Omega}^{(k+1)} = (1 - \eta\lambda)\mathbf{\Omega}^{(k)} + \eta\, e_{t_k}\, \mathbf{x}_{t_k}.$$

- $\mathbf{x}_{t_k}$: lagged inputs (including interaction lags).
- e_{t_k}: one-step residual $X_{t_k} - \widehat{X}_{t_k}$.
- η: learning rate, λ: shrinkage strength.

In simple terms

If we under-predict, coefficients move in the direction that increases the next prediction, but shrinkage keeps updates conservative.

Baseline and Evaluation Metric

- Baseline: seasonal naive (repeat last year's same month):

$$\hat{y}_{t+h}^{\text{baseline}} = y_{t+h-12}.$$

- Primary metric: RMSE, penalizes large errors:

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2}.$$

In simple terms

The baseline is "do what happened last year". If we cannot beat that, the model is not worth deploying.

Grid Search + Rolling-Origin Cross-Validation

- Fix $(d, D, S) = (1, 1, 12)$ from diagnostics, search over (p, q, P, Q).
- Rolling-origin CV: train on an expanding window, forecast $h = 3$, repeat, average error.

In simple terms

We test the model the same way it will be used: train on the past, predict the future, move forward, repeat.

Roadmap

MLOps with MLflow (Why It Matters)

- Track experiments: parameters, metrics, artifacts, and run metadata.
- Register best models and load them later for inference.
- This enables reproducible promotion to "Production" without manual file handling.

In simple terms

MLflow is our system of record. It answers: which model, trained when, with what settings, performed how.

Flask API (Production Path)

- Model discovery and loading:
  - `GET /models` lists available runs and metadata.
  - `POST /models/load` loads a specific run into memory for inference.
- Inference:
  - `POST /predict` forecasts a single segment and target.
  - `POST /predict/batch` forecasts Sales and Returns and computes total net forecast.
- Rate limiting per IP to prevent overload.

In simple terms

The API is the bridge: business systems send JSON, the server returns a forecast vector with dates.

Endpoint Example: `/predict` (Request and Response)

Request (JSON)

```
{
  "segment": "DEALER",
  "target": "net_sales",
  "num_periods": 12,
  "start_date": "2025-01-01"
}
```

Response (JSON)

```
{
  "segment": "DEALER",
  "target": "net_sales",
  "num_periods": 12,
  "forecasts": [
    {"month":"2025-01","forecast_value
        ":1234},
    {"month":"2025-02","forecast_value
        ":1189}
  ]
}
```

Endpoint Example: `/predict/batch` (Server-Side Netting)

- The server forecasts `net_sales` and `returns` for the same segment and horizon.
- Then it computes:

$$\texttt{total\_net}_t = \max(\texttt{net\_sales}_t - \texttt{returns}_t, 0).$$

In simple terms

Clients should not implement business arithmetic themselves. The server returns one consistent net series every time.

Rate Limiting (Why We Added It)

- Goal: keep latency stable and prevent a single client from exhausting resources.
- Mechanism: per-IP request limit over a short time window.
- On violation: return `HTTP 429` and instruct the client to retry later.

In simple terms

Even a good model can fail in production if the server is overloaded. Rate limiting protects the service.

Roadmap

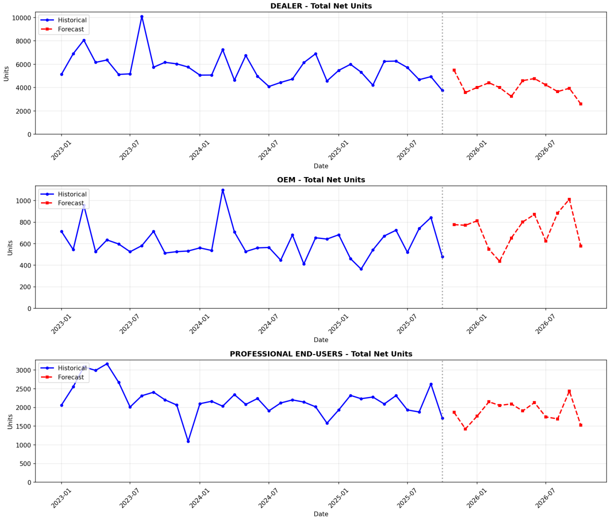
Forecast Highlights

- Model captures the recurring seasonal dip and recovery observed in history.
- Dealer segment, the most volatile, shows improved RMSE vs seasonal naive.
- Returns forecasts reduce fiscal surprises and improve net visibility.

In simple terms

Better forecasts are measured against a strong baseline and translated into business value: fewer surprises, better planning.

Forecast Visualization



Accuracy vs Baseline

Segment	Target	SARIMA RMSE	Baseline RMSE	Outcome
Dealer	Net Sales	3672.19	3685.09	**Better**
Dealer	Returns	2470.63	2699.63	**Better**
OEM	Net Sales	291.86	143.26	Baseline Better
OEM	Returns	96.91	104.72	**Better**
Pro Users	Net Sales	430.48	449.39	**Better**
Pro Users	Returns	96.62	101.76	**Better**

In simple terms

Beating the baseline is the minimum requirement. When baseline wins (OEM net sales), it signals either model mismatch or the need for exogenous drivers.

Roadmap

Training Script (02_ML_Flow.py) in One Slide

- Build lag sets including interaction lags (example: $1 + 12 = 13$).
- Fit with SGD on transformed series ($\log$ + seasonal diff + regular diff).
- Rolling-origin CV selects stable orders; best models logged to MLflow.
- Forecast inversion: undo differencing, inverse log, clamp to non-negative units.

In simple terms

The code is a controlled pipeline: transform, fit, validate, log, serve.

Inference Script (03_EndPoint.py) in One Slide

- Load models from MLflow artifacts at runtime.
- Validate JSON inputs (segment, target, horizon).
- Produce forecast vectors indexed by calendar months.
- Provide batch forecast and netting for downstream simplicity.

In simple terms

Training produces artifacts. The endpoint consumes artifacts and returns forecasts. This is the production boundary.

Roadmap

Conclusion

- Implemented multiplicative SARIMA from scratch with explicit seasonal interaction lags.
- Used rigorous evaluation: rolling-origin CV and a strong seasonal baseline.
- Deployed a reproducible, MLflow-backed Flask inference API.
- Separate Sales and Returns improves interpretability and net planning.

In simple terms

This is a complete, deployable forecasting system, not just a notebook result.

Limitations and Next Steps

- Add exogenous drivers (SARIMAX): promotions, price, macro indicators, supply constraints.
- Use Bayesian or smarter search for hyperparameters instead of small grids.
- Produce prediction intervals for risk-aware planning (not only point forecasts).
- Improve production hardening: persistent rate limits, caching, monitoring.

In simple terms

SARIMA captures internal time structure. Exogenous variables capture the outside world.

Live Demo Plan

- Call `/models` to show available MLflow runs.
- Load one model with `/models/load`.
- Run `/predict` for Dealer net sales and show the returned vector.
- Run `/predict/batch` to show server-side netting.

References

- Penn State STAT 510: Applied Time Series Analysis (ARIMA, SARIMA, diagnostics).
- Project repository (code, artifacts, and reproducibility assets).

Questions