Lecture 11: Feature Engineering

Building Better Inputs for Forecasting Models

BSAD 8310: Business Forecasting

University of Nebraska at Omaha

Spring 2026

Lecture 10 best result: LSTM with 36 features achieved RMSE $\approx$ 1,920 — down from XGBoost's 2,250 (26 features). The improvement came mostly from *better inputs*, not a fancier model.

What feature engineering adds:

- **Lag features:** give the model explicit access to past values, guided by ACF and PACF
- **Rolling statistics:** capture local trends, volatility, and momentum across multiple time scales
- **Calendar effects:** encode seasonality without requiring seasonal differencing
- **Interaction / ratio features:** year-over-year and month-over-month change rates that tree models cannot discover on their own
- **Pipeline design:** prevent data leakage by fitting transformations inside each cross-validation fold

Lecture goal: Build `make_features_extended()` (36 features), select the best subset, and update the Lectures 01–11 leaderboard.

Lag Features

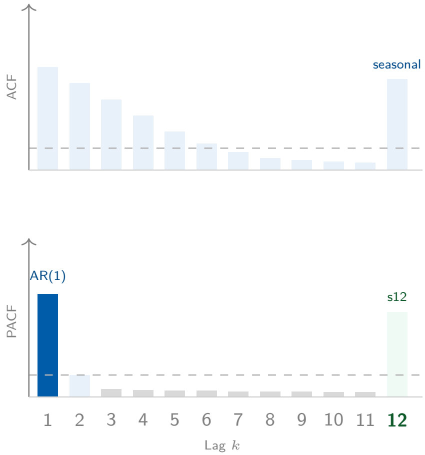
Converting time-series forecasting into a supervised regression problem

Lag features are the most direct way to give a tree-based or linear model access to the past. They convert a time-series forecasting problem into a standard supervised regression problem with y_{t-k} as predictors.

Design questions:

- Which lags are worth including? (ACF and PACF guidance)
- How many lags before diminishing returns?
- How to avoid look-ahead leakage when computing lags?

Key rule: always use `shift(k)` with $k \geq 1$ so that at prediction time for period t, only values observed at $t-1, t-2, \ldots$ are used.



Reading the charts:

- **ACF:** slow decay → persistent trend; spike at lag 12 → seasonal pattern
- **PACF:** significant at lag 1 only → pure AR(1); significant at lag 12 → include `lag_12`

Include `lag_k` if the PACF at lag k exceeds $\pm 1.96/\sqrt{n}$ (dashed band). For monthly RSXFS: include lags 1, 2, 12. Add lags 3–6 for robustness.

Socratic: ACF at lag 12 is strong, but PACF at lag 12 is also strong. Why include `lag_12` even in an ARIMA model that handles seasonality through differencing?

Safe: always `shift(k)`

```
import pandas as pd

# SAFE: shift(k) looks back k steps
# At time t we only see y[t-k]
df['lag_1']  = df['y'].shift(1)
df['lag_2']  = df['y'].shift(2)
df['lag_12'] = df['y'].shift(12)

# Rolling mean: shift FIRST
df['roll_mean_3'] = (
    df['y'].shift(1)
    .rolling(3).mean())

# No future information used.
print("Lag features are safe.")
```

Bug: missing `shift(1)`

```
# BUG: rolling without shift(1)
# includes y[t] = current target!
df['roll_mean_3'] = (
    df['y'].rolling(3).mean())

# At t=5, rolling uses:
#   y[3], y[4], y[5]  <- leakage!
# y[5] is what we are predicting.

# This inflates in-sample R^2
# but collapses on test data.

# Rule: always shift before
# any window operation.
```

Rolling without `shift(1)` is the #1 feature engineering bug. It causes dramatic overfitting that only appears on the test set.

Rolling Statistics

Local trends, volatility, and momentum over time

Rolling statistics capture local trends, volatility, and momentum over multiple time windows. They are especially powerful for tree-based models, which cannot express the concept of a "moving average" from raw lag features alone.

Feature families:

- **Rolling mean** $(w = 3, 6, 12)$: local trend at multiple scales
- **Rolling std** $(w = 3, 6, 12)$: local volatility — useful for identifying calm vs. volatile regimes
- **Rolling min/max** $(w = 3, 6, 12)$: range and extremes
- **Exponentially weighted mean (EWM)**: gives more weight to recent observations than a simple average

All rolling operations must be computed on `df['value'].shift(1)` to prevent leakage (see the Lag Leakage slide).

Rolling window operations:

Feature	Formula	Captures
`roll_mean_w`	$\bar{y}_{t-w:t-1}$	Local trend
`roll_std_w`	$s_{t-w:t-1}$	Volatility
`roll_min_w`	$\min(y_{t-w}, \ldots, y_{t-1})$	Trough
`roll_max_w`	$\max(y_{t-w}, \ldots, y_{t-1})$	Peak

- $w = 3$: captures quarter-long momentum
- $w = 6$: captures half-year cycles
- $w = 12$: captures full-year seasonality

Use all three; let the model select via regularization or feature importance.

Exponentially weighted mean:

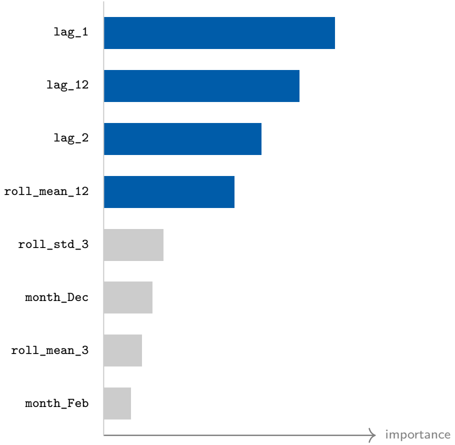
$$\mathrm{EWM}_t = \alpha\, y_{t-1} + (1-\alpha)\,\mathrm{EWM}_{t-1}, \quad \alpha \in (0,1)$$

With $\alpha = 0.3$: recent values receive $\approx 3\times$ the weight of values 4 months back.

Here α is the EWM decay weight — distinct from the level-smoothing α in Lecture 03 (ETS) and the L1/L2 mixing α in Lecture 08 (Elastic Net).

EWM reacts faster to trend reversals. On RSXFS, EWM with $\alpha = 0.3$ reduces lag behind turning points by $\sim$2 months compared with a 12-month rolling mean.

Permutation importance (Random Forest, val set):



Key findings:

- Lags 1 and 12 dominate — confirming ACF/PACF
- `roll_mean_12` is the most important rolling feature
- `month_Dec` matters: December retail spike is real
- Calendar dummies are less important than rolling features when lags already capture seasonal levels

This ranking is illustrative. Permutation importance is covered formally in Section 5 (Feature Selection).

Calendar & Structural Features

Encoding temporal position for tree-based models

Calendar features encode the position of a time step in the calendar — month, quarter, year, holiday proximity. Unlike SARIMA's seasonal differencing, calendar dummies give tree-based models explicit information about which periods are structurally different.

Key feature types:

- **Month dummies:** 11 binary columns (`month_2` through `month_12`); January is the reference category
- **Quarter dummies:** 3 binary columns (Q2–Q4)
- **Trend term:** $t = 1, 2, \ldots, T$ for linear time trend
- **Structural break indicator:** $0/1$ for pre/post a known break (recession, pandemic, policy change)

Construction in pandas:

`df.index.month` → integers 1–12

`pd.get_dummies(df['month'], prefix='month', drop_first=True)`

Interpretation (Random Forest):

Month	Avg. Effect	Business Meaning
January (ref)	—	Post-holiday low
`month_11`	+1,800	Nov. ramp-up
`month_12`	+3,200	Holiday peak
`month_7`	+900	Summer surge
`month_2`	−400	Feb. dip

Both encode seasonality, but differently:

- `lag_12` $= y_{t-12}$: level effect — captures the same magnitude as last year
- `month_12`: indicator effect — always adds the same fixed amount for December, regardless of last year's level

Use both when trend is changing.

Socratic: A SARIMA model with $s = 12$ handles seasonality through seasonal differencing. Why might month dummies improve a Random Forest but add no value to SARIMA?

Calendar features help when:

- The seasonal pattern is *fixed* across years (e.g., holiday retail, fiscal quarters)
- The model does not already capture seasonality via lag features (e.g., shallow trees, linear models without lag 12)
- Structural breaks are known and datable (add a 0/1 indicator for pre/post)

Calendar features can hurt when:

- Seasonal patterns are *evolving* over time — the fixed-effect assumption breaks down
- `lag_12` already accounts for seasonality — month dummies add multicollinearity
- The dataset is small ($n < 100$): 11 month dummies consume degrees of freedom faster than they add signal

> Never use calendar dummies as the sole source of seasonality in a neural network (LSTM). LSTMs learn temporal patterns from the sequence itself; adding 11 overlapping dummies adds noise. Use them as supplementary features only when lag features are already present.

Interaction & Ratio Features

Encoding change, not level

Interaction and ratio features encode *change* rather than *level*. They help tree-based models detect acceleration or deceleration in a series — a concept that requires computing ratios of two variables, something a single tree split cannot express directly.

Key features:

- **Year-over-year (YoY) change:** $\Delta^{(12)} y_t = y_{t-1}/y_{t-13} - 1$
- **Month-over-month (MoM) change:** $\Delta^{(1)} y_t = y_{t-1}/y_{t-2} - 1$
- **Lag interaction:** $y_{t-1} \times \mathbf{1}[\text{month} = 12]$ — lets the model use different slopes for December

Year-over-year rate of change:

$$\text{YoY}_t = \frac{y_{t-1}}{y_{t-13}} - 1$$

Note: both y_{t-1} and y_{t-13} are lagged to prevent leakage at prediction time for target y_t.

Month-over-month rate of change:

$$\text{MoM}_t = \frac{y_{t-1}}{y_{t-2}} - 1$$

Why ratios instead of differences? A 1,000-unit change means something different when the baseline is 10,000 vs. 100,000. Ratios are scale-free.

For RSXFS retail sales:

- $\text{YoY}_t > 0$: economy is stronger than same month last year
- $\text{YoY}_t < 0$: contraction signal
- $|\text{MoM}_t|$ large: abnormal month-to-month swing (data revision or shock)

These features help the model distinguish a Christmas peak from an unexpected demand shock.

Socratic: YoY removes the seasonal level but still contains trend. What additional transformation would make YoY_t a stationary series?

Full feature engineering pipeline:

```
def make_features_extended(df,
                          lags=range(1, 13),
                          roll_windows=[3, 6, 12]):
    X = df[['value']].copy()
    # Lag features (ACF/PACF-guided, lags 1--12)
    for k in lags:
        X[f'lag_{k}'] = X['value'].shift(k)
    # Rolling stats (shift first to avoid leakage)
    for w in roll_windows:
        r = X['value'].shift(1).rolling(w)
        X[f'roll_mean_{w}'] = r.mean()
        X[f'roll_std_{w}']  = r.std()
        X[f'roll_min_{w}']  = r.min()
        X[f'roll_max_{w}']  = r.max()
    # Exponentially weighted mean
    X['ewm_alpha03'] = (
        X['value'].shift(1)
        .ewm(alpha=0.3, adjust=False).mean())
    # Calendar dummies (Jan = reference category)
    months = pd.Series(
        X.index.month, index=X.index)
    dums = pd.get_dummies(
        months, prefix='month',
        drop_first=True)
    X = pd.concat([X, dums], axis=1)
    # Return 36-column feature matrix
    return X.drop(columns='value').dropna()
```

Feature Selection

Choosing the best subset of 36 candidate features

More features are not always better. With 36 features and $n \approx 300$ observations, we risk overfitting — especially in linear models. **Feature selection** identifies the subset of features that maximizes out-of-sample predictive accuracy.

Three strategies covered today:

1. **LASSO** (from Lecture 08): ℓ_1 penalty shrinks irrelevant feature coefficients to exactly zero. Implicit selection via the regularization path.
2. **Permutation importance**: measure how much test RMSE rises when a feature's values are randomly shuffled. Model-agnostic
3. **RFECV**: Recursive Feature Elimination with cross-validation. Fit model, remove weakest feature, repeat until CV score stops improving

LASSO regularization path (Lecture 08 connection):

$$\hat{\beta}^{\text{LASSO}} = \arg\min_{\boldsymbol{\beta}} \underbrace{\sum_t (y_t - \mathbf{x}_t^\top \boldsymbol{\beta})^2}_{\text{prediction error}} + \lambda \underbrace{\|\boldsymbol{\beta}\|_1}_{\text{selection penalty}}$$

As λ increases along the regularization path:

- **Lag 12 and lag 1** coefficients survive to large λ — high marginal importance
- **Rolling std features** shrink toward zero early — lower marginal importance
- At CV-optimal λ^*: many coefficients are exactly zero (automatic feature selection) (Tibshirani 1996)

LASSO requires standardized features — all inputs should be on comparable scales (use `StandardScaler` inside a `Pipeline` to prevent leakage).

2022

1. Fit Random Forest on training set
2. Record **baseline** validation RMSE: RMSE_0
3. For each feature j: shuffle column j, re-predict, record RMSE_j
4. $\text{Importance}_j = \text{RMSE}_j - \text{RMSE}_0 \quad$ (larger = more important)

Advantages over split-count (impurity) importance:

- Evaluated on *validation* set — not biased toward high-cardinality features
- *Model-agnostic*: applies to LSTM, XGBoost, linear models
- Measures *marginal* contribution with all other features present

On RSXFS: `lag_1`, `lag_12`, `roll_mean_12`, `ewm_alpha03` are the top four. See the importance preview in Section 2 (Rolling Statistics).

Input: estimator, feature matrix X, cross-validator

Repeat:

1. Fit estimator on all remaining features
2. Rank features by importance
3. Eliminate the lowest-ranked feature

Select: the subset size that maximizes mean CV score (Guyon and Elisseeff 2003)

Key parameters:

- `step=1`: eliminate one feature per round
- `cv=TimeSeriesSplit(gap=1)`: prevents temporal leakage during selection
- `min_features_to_select=5`: floor to avoid trivial models

```python
from sklearn.feature_selection import RFECV
from sklearn.ensemble import (
    RandomForestRegressor)
from sklearn.model_selection import (
    TimeSeriesSplit)

estimator = RandomForestRegressor(
    n_estimators=200, random_state=42)

tscv = TimeSeriesSplit(n_splits=5, gap=1)

selector = RFECV(
    estimator=estimator,
    step=1,
    cv=tscv,
    scoring='neg_mean_squared_error',
    min_features_to_select=5)

selector.fit(X_tr, y_tr)

# Selected feature names
selected = X_tr.columns[
    selector.support_].tolist()
print(f"Selected: {len(selected)} feats")
```

Pipeline Design

Preventing leakage in cross-validation

A **sklearn Pipeline** chains transformations and a model into a single object. Inside cross-validation, the pipeline re-fits every transformation on the training fold only — preventing the subtle leakage that occurs when scaling with statistics from the full dataset.

Why pipelines matter for time-series CV:

- `StandardScaler` fitted on all data leaks test-set mean and variance into training
- With `Pipeline` + `TimeSeriesSplit`: scaler is fitted fresh on each train fold
- Same principle applies to `RFECV` and any imputer — fit only on train, transform both train and test (Pedregosa et al. 2011)

Rule of thumb: if a transformation looks at the target variable (e.g., LabelEncoder, WoE encoding) or at the distribution of X (e.g., StandardScaler, PCA), it must go inside the pipeline.

Raw Data (RSXFS)

↓

Feature Engineering (36 features)

↓

TimeSeriesSplit (gap=1)

↓

StandardScaler (fit on train fold)

↓

Model (RF / XGBoost / LSTM)

↓

Predictions $\hat{y}_{T+h}$

What happens inside each CV fold:

1. **Split:** `TimeSeriesSplit(gap=1)` — 1-step gap prevents leakage
2. **Fit scaler:** `StandardScaler` on train fold *only*
3. **Transform:** same params applied to train and val
4. **Fit model:** on scaled train fold
5. **Score:** RMSE on scaled val fold
6. **Report:** average across all folds

Fitting `StandardScaler` before CV on all data: validation mean/std seep into training. Typical RMSE improvement from fixing this: 50–200 units on RSXFS.

Step 1: Construct the pipeline

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import (
    StandardScaler)
from sklearn.ensemble import (
    RandomForestRegressor)
from sklearn.model_selection import (
    TimeSeriesSplit, cross_val_score)
import numpy as np

# Build pipeline (order matters!)
pipe = Pipeline([
    ('scaler', StandardScaler()),
    ('model', RandomForestRegressor(
        n_estimators=300,
        random_state=42))])

tscv = TimeSeriesSplit(
    n_splits=5, gap=1)
```

Step 2: Cross-validate and predict

```
# CV: scaler fit on each train fold
scores = cross_val_score(
    pipe,
    X.values,
    y.values,
    cv=tscv,
    scoring=(
        'neg_mean_squared_error'))
rmse_cv = np.sqrt(-scores.mean())
print(f"CV RMSE: {rmse_cv:,.0f}")

# Refit on full training data
pipe.fit(X_tr.values, y_tr.values)

# Predict on held-out test set
y_pred = pipe.predict(
    X_te.values)
rmse_te = np.sqrt(
    np.mean((y_te - y_pred)**2))
print(f"Test RMSE: {rmse_te:,.0f}")
```

Application to Forecasting

Baseline vs. extended features on RSXFS retail sales

Apply `make_features_extended()` (36 features) to RSXFS retail sales and compare against the baseline 26-feature set from Lectures 07–10. Measure the marginal contribution of better features holding the model class fixed.

Evaluation design:

- **Baseline** (26 features): 12 lags, 6 rolling means, 6 rolling stds, 2 ratios — as in Lectures 07–10
- **Extended** (36 features): adds rolling min/max, EWM, and 11 month dummies
- **Models:** Elastic Net, Random Forest, XGBoost, LSTM (median over 5 seeds)
- **Holdout:** same chronological 15% test set used throughout the course

Test-set RMSE on RSXFS:

Model	26f	36f	ΔRMSE
Seasonal Naïve	4 210	—	—
SARIMA$(1,1,1)(1,1,1)_{12}$	2 840	—	—
Elastic Net	2 540	2 410	−130
Random Forest	2 380	2 210	−170
XGBoost	2 250	2 050	−200
LSTM (2-layer, $T = 24$)	2 180	1 920	−260

Values are illustrative. LSTM = median over 5 random seeds.

Interpretation:

- All four ML models improve with richer features; LSTM gains most
- The improvement is larger for more flexible models — extra features give trees more split choices
- Elastic Net gains least: its penalty already prevents overfitting to noisy features

Feature engineering delivers larger improvements than switching model class. **RMSE: LSTM 36f < XGBoost 36f < LSTM 26f**: the feature gap dominates the model gap.

Takeaways and References

What we learned and where to go next

- **Lag features** convert time series to supervised regression; use ACF and PACF to select which lags to include (Box et al. 2015).
- **Rolling statistics** (mean, std, min, max, EWM) capture local trend and volatility at multiple scales. Always `shift(1)` before rolling to prevent leakage.
- **Calendar dummies** help tree models detect recurring peaks and troughs; they add less value when `lag_12` is already present.
- **Feature selection** (LASSO, permutation importance, RFECV) reduces overfitting and identifies the most informative signals (Guyon and Elisseeff 2003; Molnar 2022).
- **Pipelines** are essential for leakage-free cross-validation. Fit all transformations inside the CV loop, not on the full dataset (Pedregosa et al. 2011).
- **Feature engineering beats model selection:** on RSXFS, moving from 26 to 36 features reduced RMSE by 130–260 units across all model classes.

Preview of Lecture 12: Capstone and Applications — combining the full Lectures 01–11 toolkit on a business case study with model selection, uncertainty quantification, and presentation-ready visualizations.

- Box, George E. P. et al. (2015). *Time Series Analysis: Forecasting and Control*. 5th. Hoboken, NJ: Wiley.
- Guyon, Isabelle and André Elisseeff (2003). "An Introduction to Variable and Feature Selection". In: *Journal of Machine Learning Research* 3, pp. 1157–1182.
- Molnar, Christoph (2022). *Interpretable Machine Learning: A Guide for Making Black Box Models Explainable*. 2nd. Open access. URL: https://christophm.github.io/interpretable-ml-book/.
- Pedregosa, Fabian et al. (2011). "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12, pp. 2825–2830.
- Tibshirani, Robert (1996). "Regression Shrinkage and Selection via the Lasso". In: *Journal of the Royal Statistical Society: Series B* 58.1, pp. 267–288.