Time-series,
Spring, 2026

Time series as supervised learning

Faculty of DS & AI
Spring semester, 2026

Trong-Nghia Nguyen

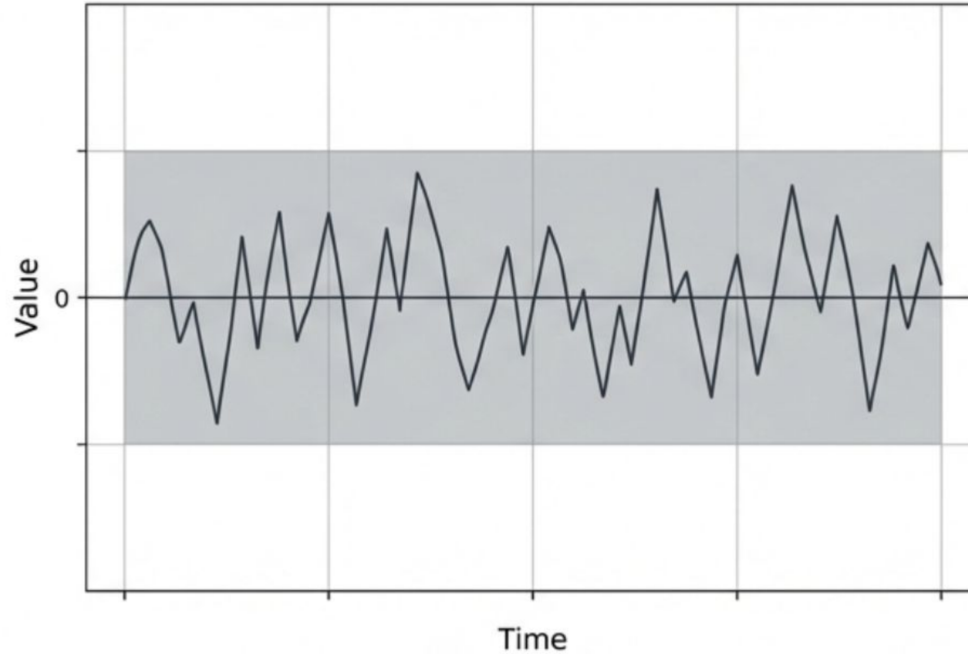
Content

- **ARCH/GARCH**
- Time Series as Supervised Learning
- ML Models for Forecasting
- Temporal Validation Strategies

ARCH/GARCH

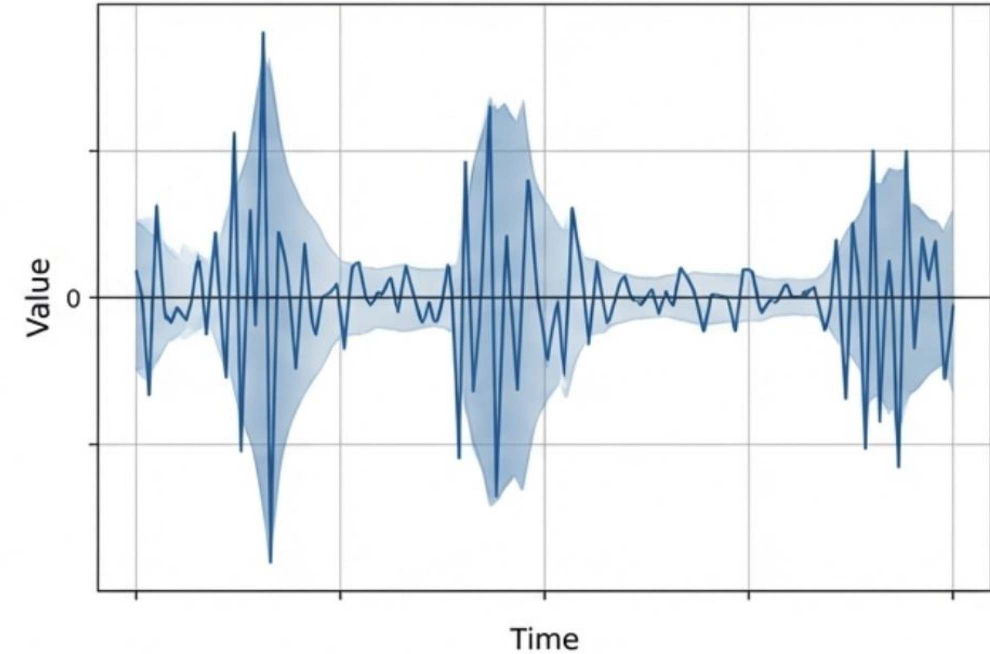
Definition

Traditional Models (e.g., ARIMA)



Homoskedasticity: Assumes the variance of the error term remains perfectly constant over time.

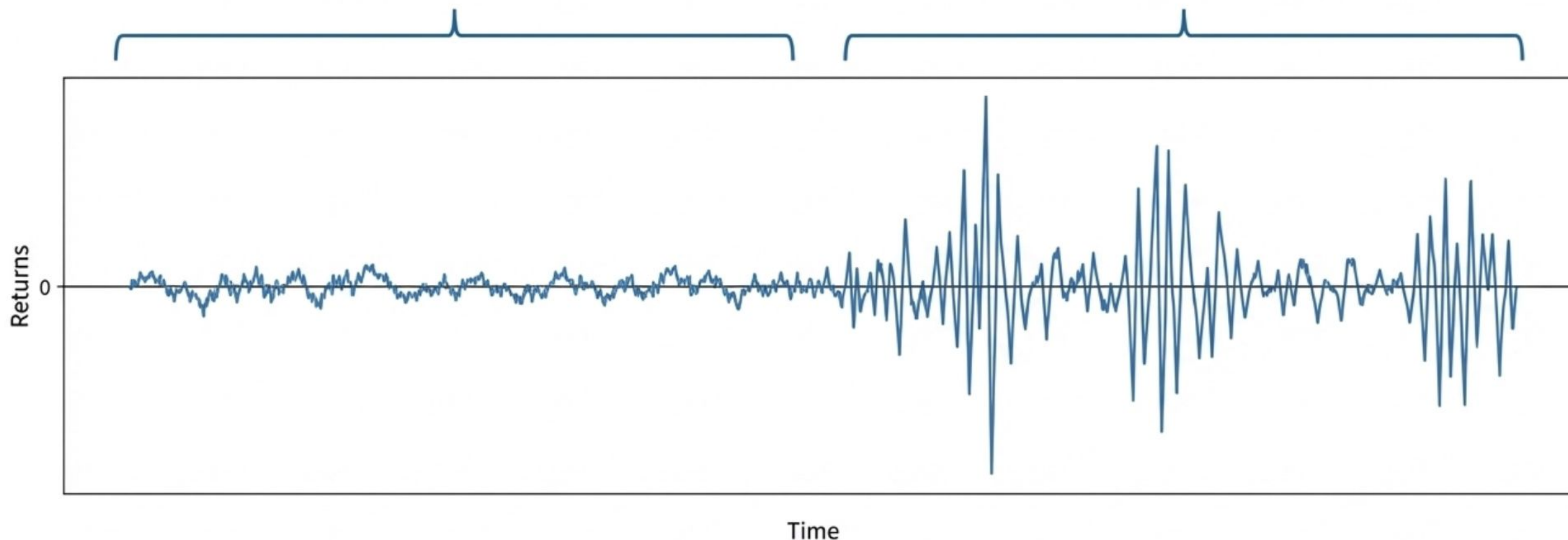
Market Data (Stocks, FX, Gold)



Heteroskedasticity: Prices exhibit violently shifting variance. The traditional model completely **fails** to capture these shifting risk profiles.

ARCH/GARCH

Definition



ARCH/GARCH

Definition

The **ARCH** Breakthrough (Engle, 1982)

Autoregressive Conditional Heteroskedasticity models current variance solely as a function of **past** squared errors.

$$\sigma_t^2 = \alpha_0 + \alpha_1 \varepsilon_{t-1}^2 + \ldots + \alpha_q \varepsilon_{t-q}^2$$

Today's Variance (Current Risk)

Constant baseline volatility

Market "shocks" from q periods ago

Limitation: Capturing long memory requires too many lags (q must be large), making the model overly complex and highly difficult to estimate.

ARCH/GARCH

Definition

The GARCH **Generalization** (Bollerslev, 1986)

Generalized ARCH resolves the parameter burden by introducing past variance back into the equation.

$$\sigma_t^2 = \omega + \sum \alpha_i \epsilon_{t-i}^2 + \sum \beta_j \sigma_{t-j}^2$$

ARCH **Term**: Reaction to short-term market shocks.

GARCH **Term**: Long-term volatility persistence (momentum).

Industry Standard: GARCH(1,1) is the dominant version. It explains most financial series efficiently with just 3 parameters $(\omega, \alpha_1, \beta_1)$.

ARCH/GARCH

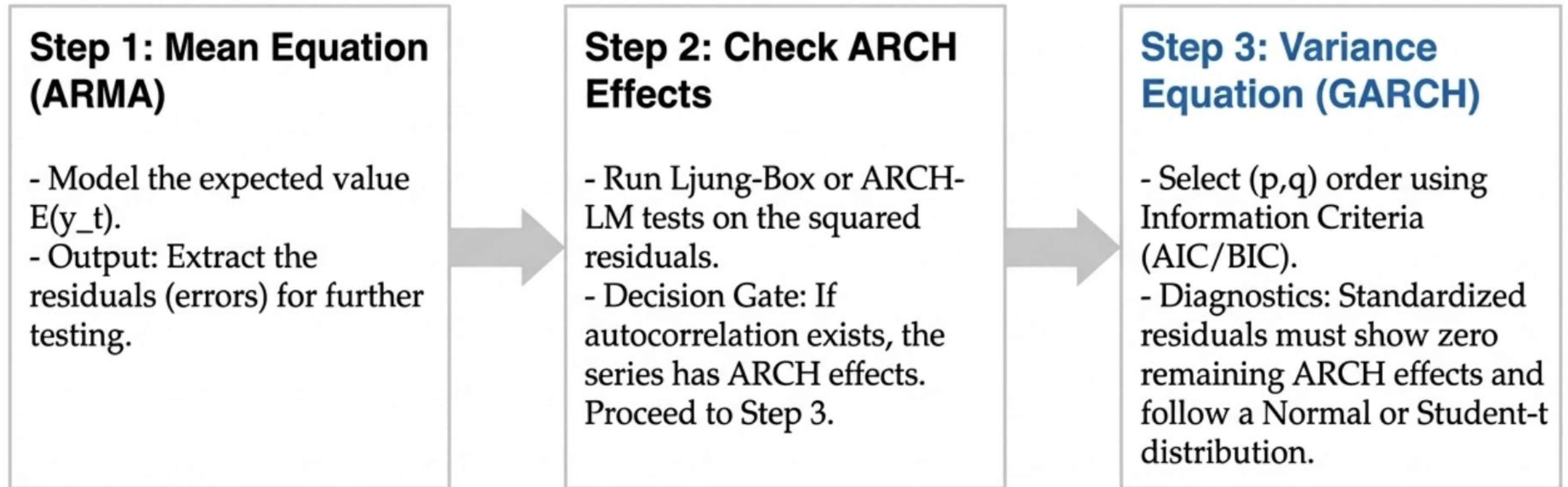
Definition

Architectural Comparison: ARCH vs. GARCH

Dimension	ARCH	GARCH
Origin	1982 (Robert Engle)	1986 (Tim Bollerslev)
Variance Structure	Past errors only	Past errors + Past variance
Complexity	High parameter burden (requires many lags to fit data well)	Parameter efficient (GARCH(1,1) is usually sufficient)
Primary Application	Short-term volatility analysis	Risk forecasting (Value-at-Risk), Option Pricing

ARCH/GARCH

Definition



ARCH/GARCH

Definition

Implementation via Python (arch library)

```
from arch import arch_model

# Define GARCH(1,1) model with a constant mean
model = arch_model(returns, vol='Garch', p=1, q=1)

# Fit the model
results = model.fit()

# Print summary diagnostics
print(results.summary())
```

This is the raw financial time series data (the shifting variance observed in Slide 2).

Instructs the engine to use Bollerslev's 1986 architecture.

Directly maps to the parameter efficiency of GARCH(1,1). p=1 captures one lag of past variance, q=1 captures one lag of past shocks.

Content

- ARCH/GARCH
- **Time Series as Supervised Learning**
- ML Models for Forecasting
- Temporal Validation Strategies

Time Series as Supervised Learning

Definition

Raw Time Series Reality:
Raw data is merely a list of sequential values over time (y_t).

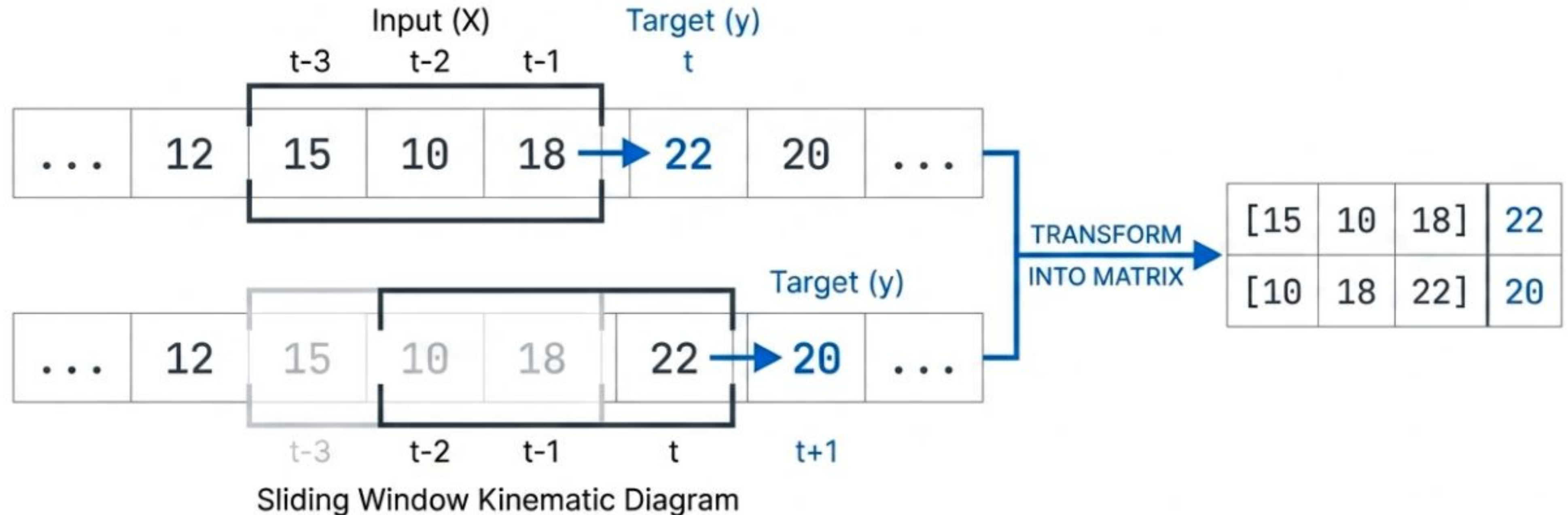
Date	Value (y)
01/01	10
02/01	15
03/01	12
04/01	18
05/01	20

Standard ML Expectation:
Models (XGBoost, Random Forest) require tabular data with distinct Features (X) and a Target (y).

Input (X_1)	Input (X_2)	Target (y)
?	?	10
?	?	15

Time Series as Supervised Learning

Method



Time Series as Supervised Learning

Method

Core technique: A "window" slides across the data. The past becomes the Input (X), the immediate future becomes the Target (y).

Window Step 1

Date	Value	
01/01	10	(X_1)
02/01	15	(X_2)
03/01	12	(y)
04/01	18	
05/01	20	

Window Step 2

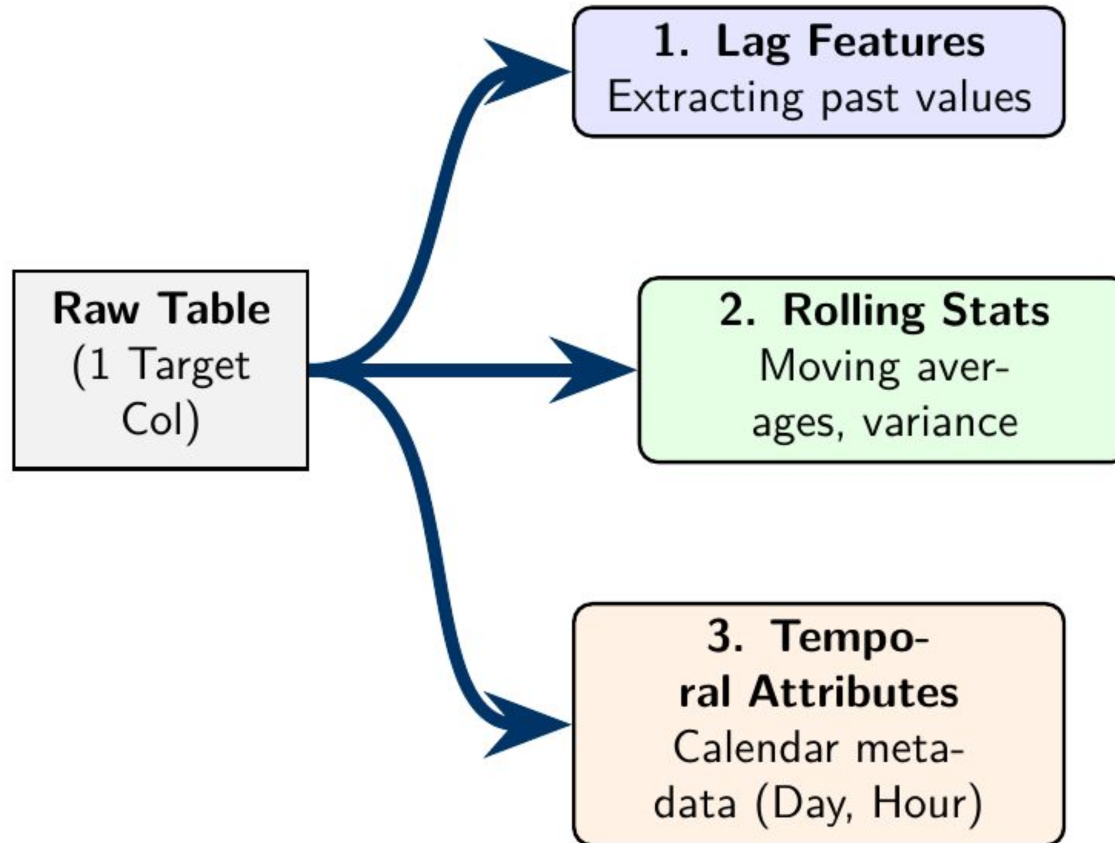
Date	Value	
01/01	10	
02/01	15	(X_1)
03/01	12	(X_2)
04/01	18	(y)
05/01	20	

Resulting Matrix:

Input (X)		Target(y)
X_1	X_2	y
10	15	12
15	12	18

Time Series as Supervised Learning

Method

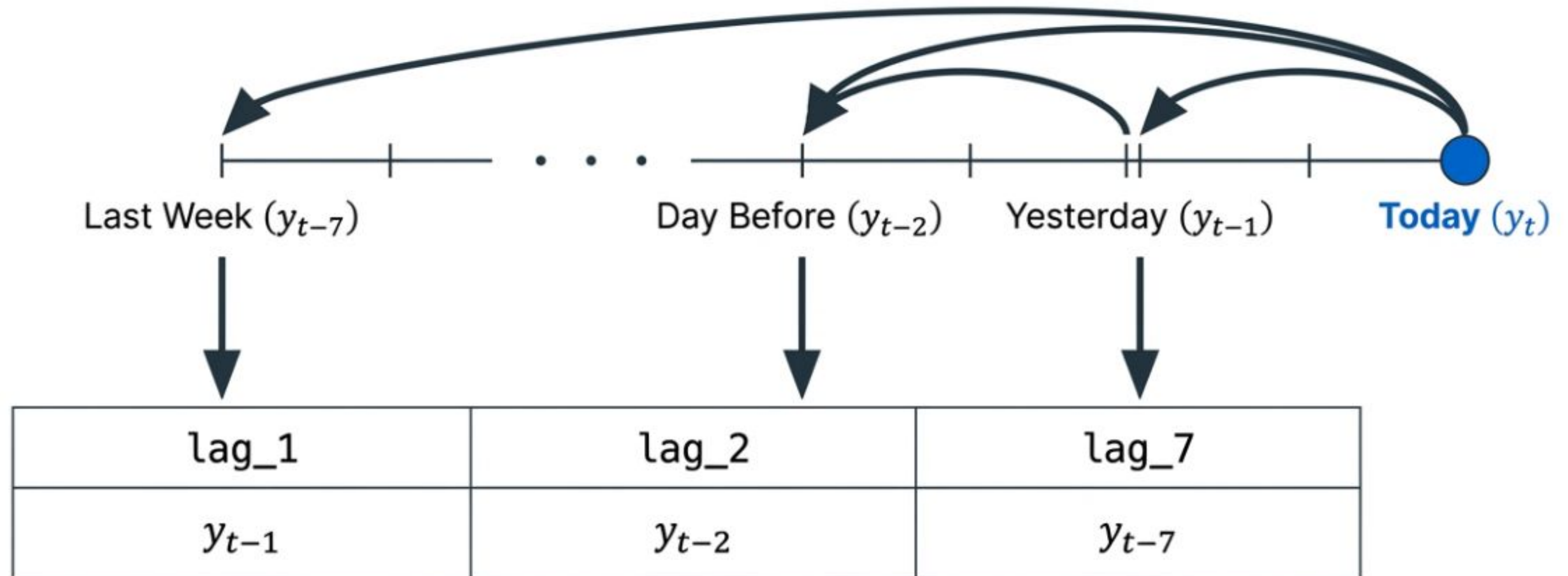


Time Series as Supervised Learning

Method

Pillar 1: Lag Features

Utilizing direct past values of the target variable as independent predictive variables.



Time Series as Supervised Learning

Method

Raw Table

Date	y
01/01	10
02/01	15
03/01	12
04/01	18
05/01	20

```python
import pandas as pd

# Create a copy
df_lag = df.copy()

# y from 1 day ago
df_lag['lag_1'] = df_lag['y'].shift(1)

# y from 2 days ago
df_lag['lag_2'] = df_lag['y'].shift(2)
```

New Table (with Features)

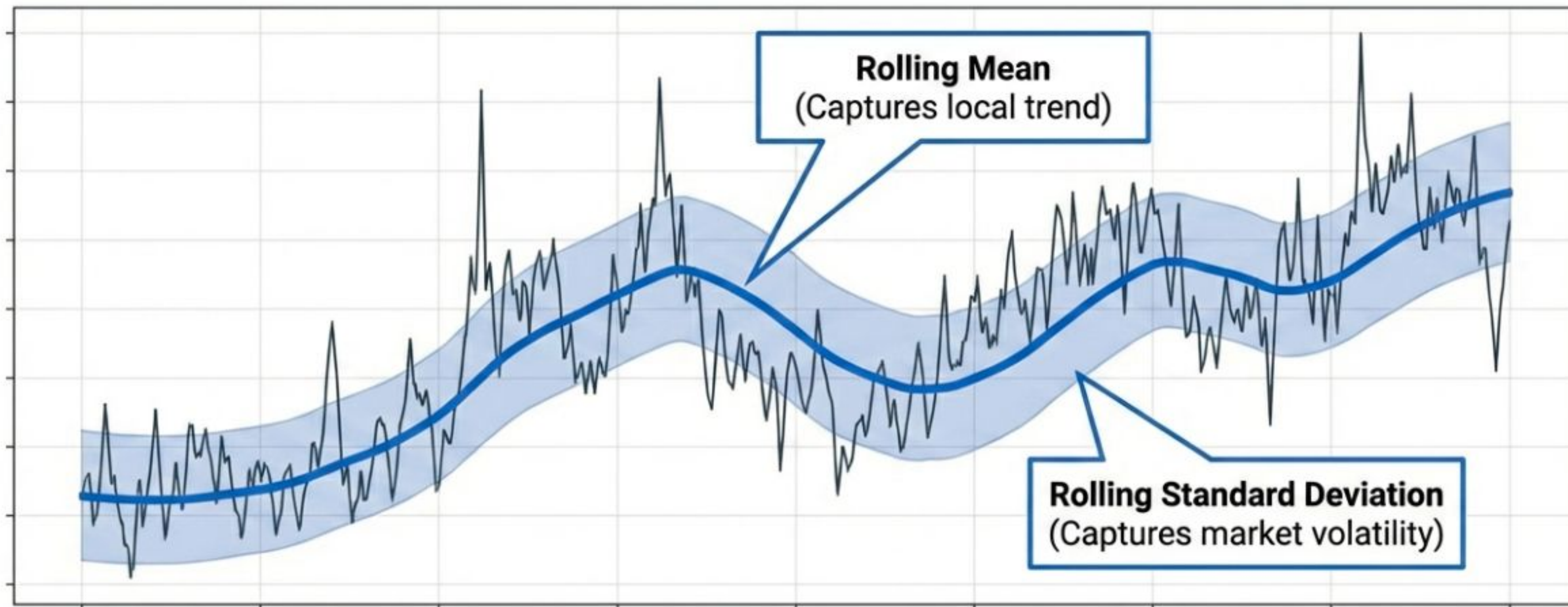
Date	y	lag_1	lag_2
01/01	10	NaN	NaN
02/01	15	10	NaN
03/01	12	15	10
04/01	18	12	15
05/01	20	18	12

Time Series as Supervised Learning

Method

Pillar 2: Rolling-Window Statistics

Replacing single isolated values with aggregated statistical behaviors calculated over a recent temporal interval to capture system state.



Time Series as Supervised Learning

Method

Raw Table

Date	y
01/01	10
02/01	15
03/01	12
04/01	18
05/01	20

```python
# Calculate moving average of
# the past 2 days (Shift 1)
df_roll['roll_mean_2'] = (
    df_roll['y'].shift(1)
                .rolling(2)
                .mean()
)
```

Note: Must use 'shift(1)' before 'rolling()' to prevent Data Leakage.

New Table (with Features)

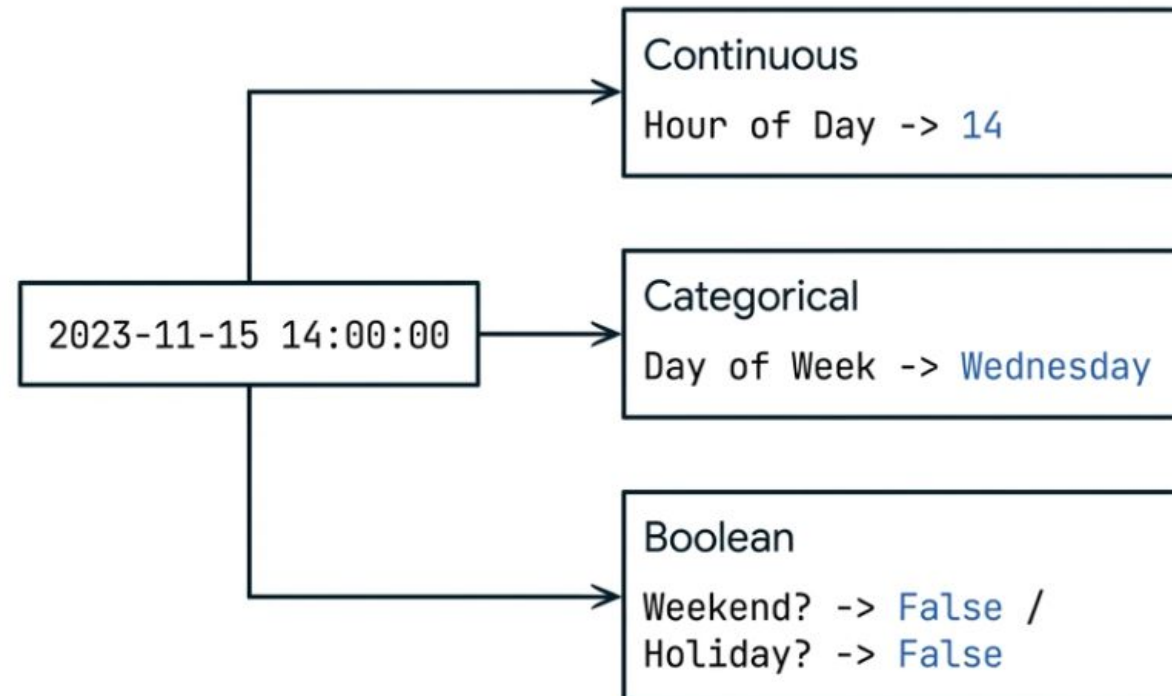
Date	y	roll_mean_2
01/01	10	NaN
02/01	15	NaN
03/01	12	12.5 *(10+15)/2*
04/01	18	13.5 *(15+12)/2*
05/01	20	15.0 *(12+18)/2*

Time Series as Supervised Learning

Method

Pillar 3: Temporal Attributes

Extracting metadata directly from the timestamp to explicitly teach the machine learning model about cyclicality and calendar events.



Time Series as Supervised Learning

Method

Raw Table

Date	y
01/01 (Mon)	10
02/01 (Tue)	15
03/01 (Wed)	12
04/01 (Thu)	18
05/01 (Fri)	20

```
# Extract day of week
# (0=Monday, 6=Sunday)
df_time['day_of_week'] = \
    df_time.index.dayofweek

# Create dummy variable (Boolean)
df_time['is_weekend'] = \
    (df_time['day_of_week'] >= 5)
    .astype(int)
```

New Table (with Features)

Date	y	day_of_week	is_
01/01	10	0	
02/01	15	1	
03/01	12	2	
04/01	18	3	
05/01	20	4	

Time Series as Supervised Learning

Method

Synthesis: The Complete Supervised Matrix

Ready for XGBoost / Random Forest

After creating individual feature tables (Slides 14, 15, 16), we merge them and drop rows containing NaN values caused by the shifting process.

```
# Concatenate all feature columns
final_df = pd.concat([df['y'], df_lag, df_roll, df_time], axis=1)

# Drop NaN rows so the model can train
final_df = final_df.dropna()
```

Final tabular dataset (X and y) ready for the algorithm:

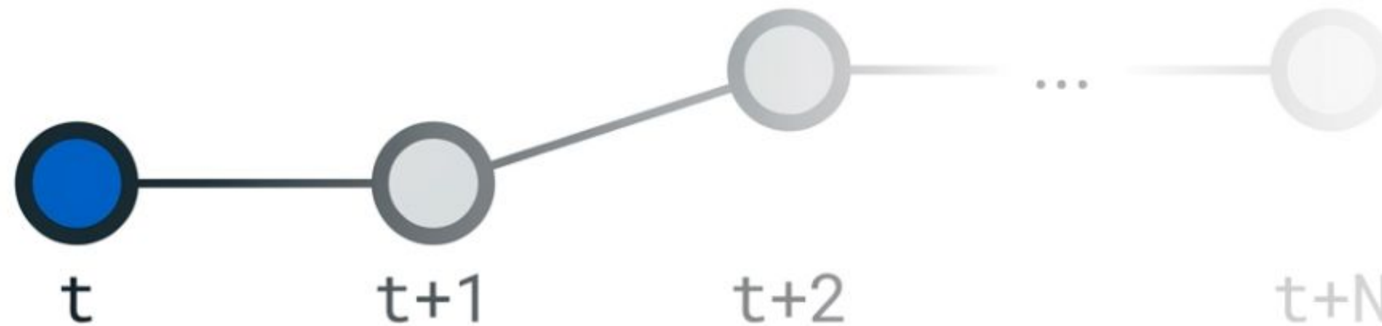
Date	Target (y)	lag_1	lag_2	roll_mean	day_of_week	is_weekend
03/01	**12**	15	10	12.5	2	0
04/01	**18**	12	15	13.5	3	0
05/01	**20**	18	12	15.0	4	0

Time Series as Supervised Learning

Definition

The Challenge of Multi-Step Forecasting

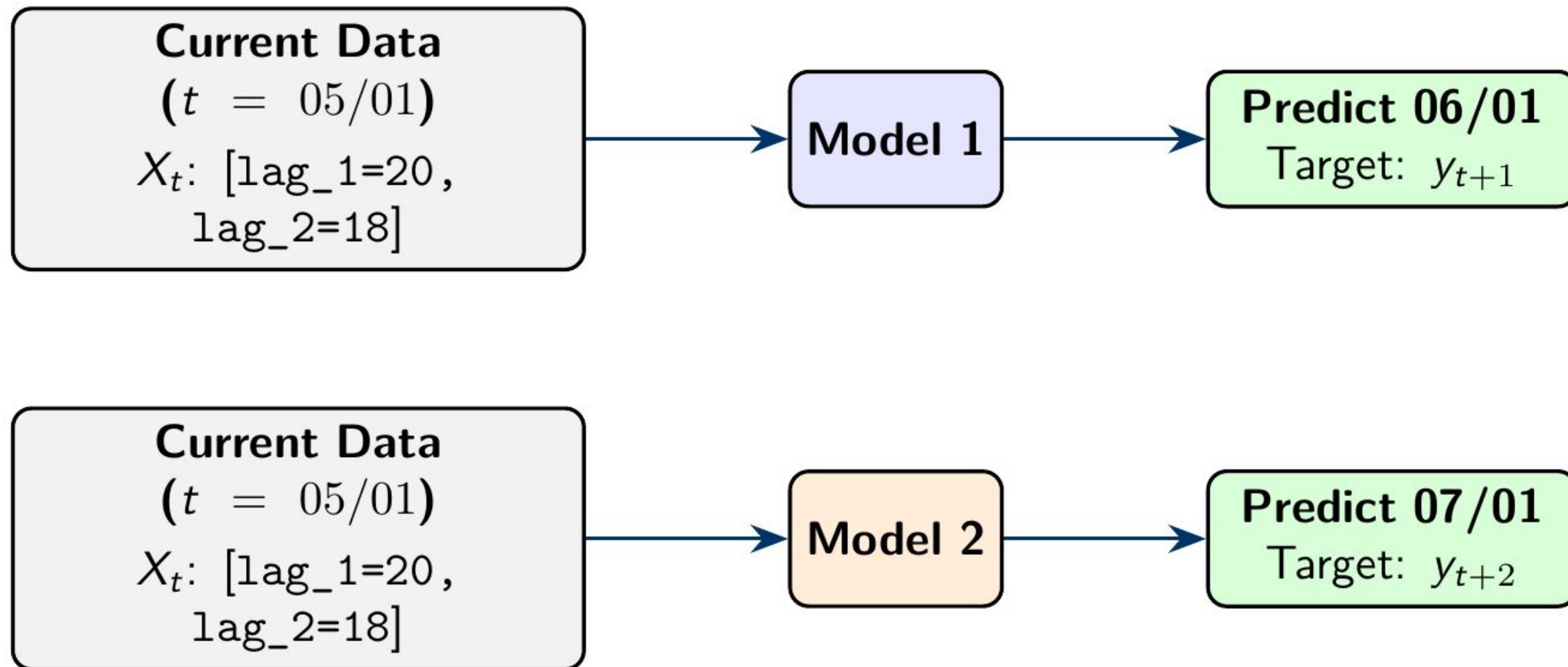
Real-world applications—such as predicting electricity prices for the next 24 hours—require predicting multiple consecutive future horizons, not just the immediate next step.



Time Series as Supervised Learning

Method

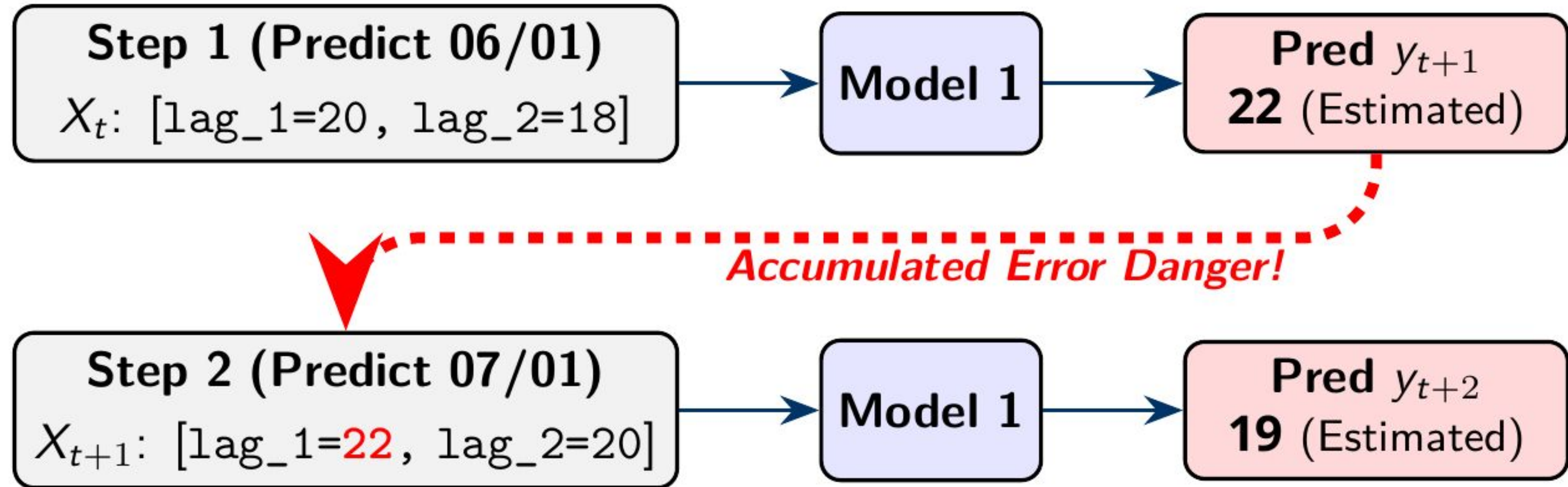
Scenario: Today is **05/01** ($y = 20$). We want to predict **06/01** ($t + 1$) and **07/01** ($t + 2$).



- **Key Insight:** The input features (X_t) remain exactly the same.

Time Series as Supervised Learning

Method



- **Key Insight:** The model's own output (**22**) replaces the true values in the next step's input.
- **Advantage:** Highly efficient (Requires only 1 model).
- **Drawback:** If the prediction at Step 1 is wrong, Step 2 is guaranteed to degrade.

Time Series as Supervised Learning

Definition

Diagnostic Matrix: Forecasting Strategies

	Direct Strategy	Recursive Strategy
Architecture	N distinct models	1 single model
Prediction Horizon	Model N predicts t+N	Model predicts 1 step, loops for N
Error Risk	Isolated (No accumulation)	High (Accumulated Error)
Compute Cost	Extremely High	Low

Time Series as Supervised Learning

Definition

Critical Implementation Principles

1.

Acknowledge Frequency

Feature design must strictly match the temporal frequency (hourly, daily) of the raw data.

2.

Beware the Recursive Loop

Monitor recursive architectures closely; accumulated error will degrade long-horizon predictions exponentially.

3.

Align Window to Reality

Window sizes and lag selections are not arbitrary; they must be dictated by the specific domain knowledge of the business problem.

Content

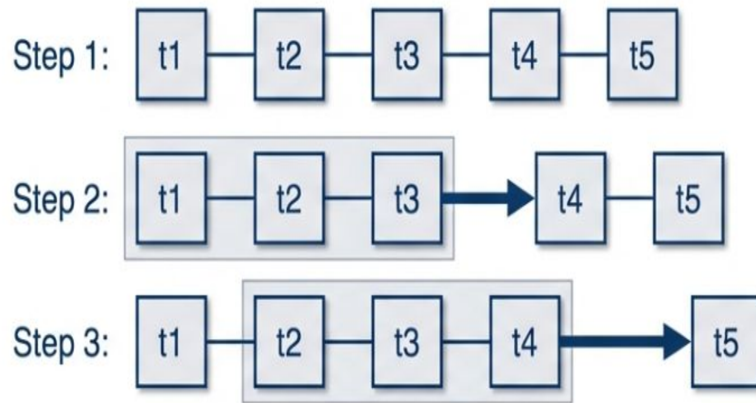
- ARCH/GARCH
- Time Series as Supervised Learning
- **ML Models for Forecasting**
- Temporal Validation Strategies

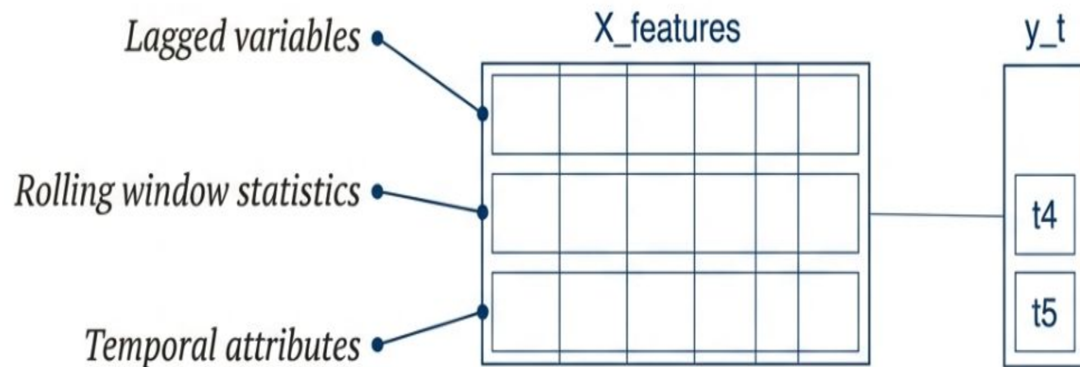
ML Models for Forecasting

Definition

Transforming sequences into tabular matrices via the Sliding Window

$$y_t = f(X_{\text{features}}) + \epsilon$$





Experimental Methodology: Feature extraction in Python

Shifts data downwards to align past observations with current targets.

```python
import pandas as pd

def create_features(df, target_col):
    # 1. Create Lag Variables (Historical states)
    df['lag_1'] = df[target_col].shift(1)
    df['lag_2'] = df[target_col].shift(2)

    # 2. Create Rolling Statistics (Sliding window)
    df['rolling_mean_3'] = df[target_col].rolling(window=3).mean()

    # Drop NaN values generated by shifting
    return df.dropna()

X_features = create_features(data, 'sales')
```

Calculates the local average over a 3-step sliding window.

ML Models for Forecasting

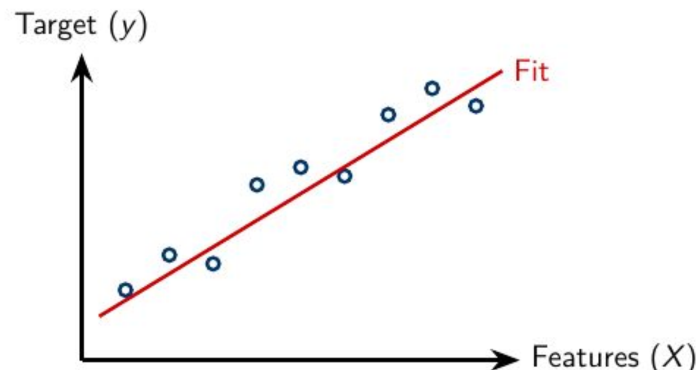
Definition

The Diagnostic Baseline: Linear Regression

Always start simple: Speed, Interpretability, and Native Extrapolation

Theory & Evaluation:

- $y = \beta_0 + \beta_1 X_{lag1} + \cdots + \epsilon$
- **Strengths:** Extremely fast. Highly interpretable (coefficients β directly indicate lag importance). Naturally extrapolates future trends.
- **Limitations:** Mathematically constrained to linear relationships. Fails on complex feature interactions.

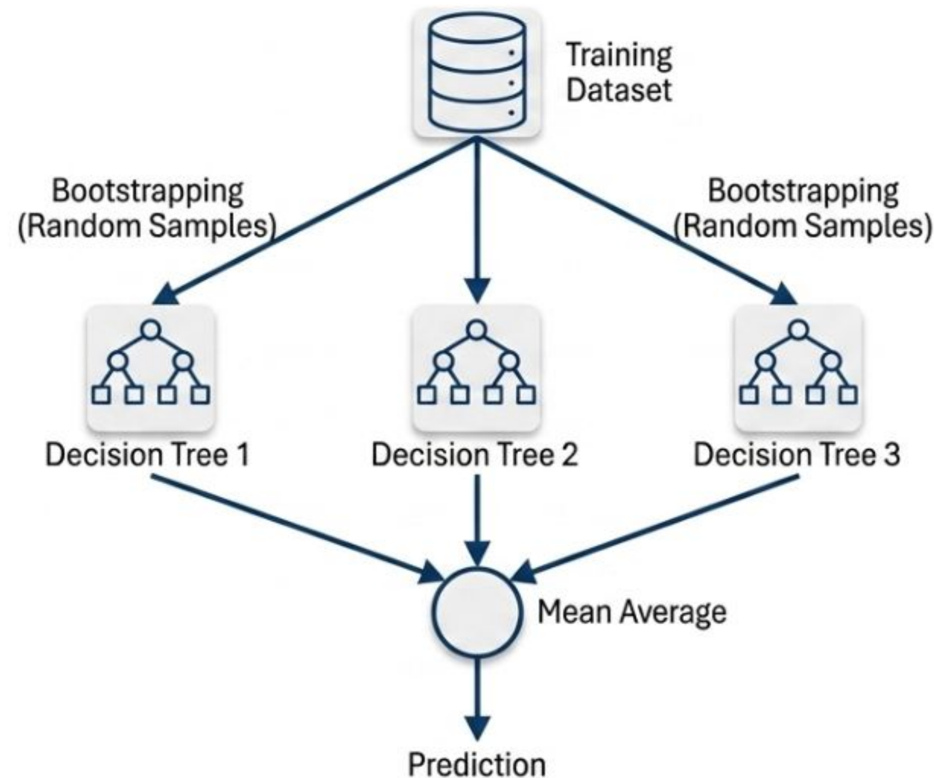



```python
from sklearn.linear_model import LinearRegression

# X: Matrix of Lags and Time features
# y: Target variable vector

# 1. Initialize the baseline model
model_lr = LinearRegression()

# 2. Train on historical data
model_lr.fit(X_train, y_train)

# 3. Predict the future
y_pred = model_lr.predict(X_test)

# 4. Interpretability (Check weights)
print("Lag Importance:", model_lr.coef_)
```

ML Models for Forecasting

Definition

Complexity evolution: Random Forest ensembles



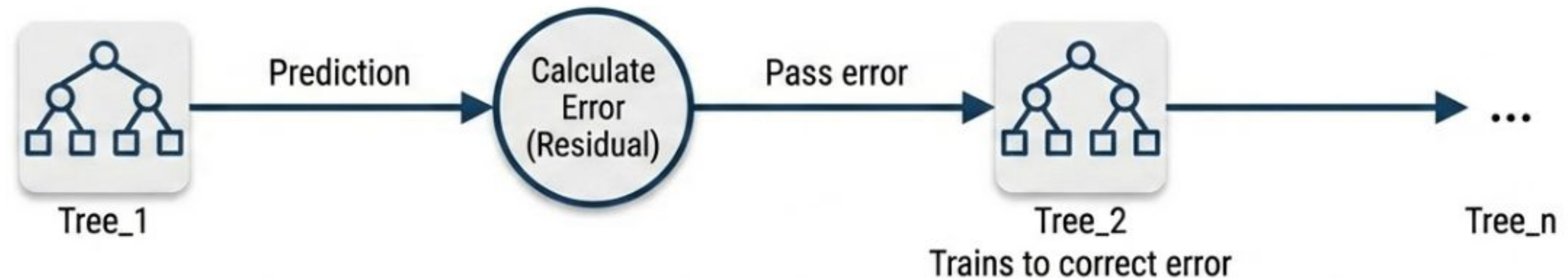
Mechanism: Builds multiple independent decision trees on randomized subsets of data, averaging their outputs to suppress overfitting.

Application Advantage: Highly robust when handling datasets containing dozens or hundreds of exogenous (external) variables.

ML Models for Forecasting

Definition

The dominant architecture: Gradient Boosting



Mechanism (XGBoost / LightGBM): Trees are built sequentially, not in parallel. Each new tree explicitly trains to correct the residual errors of the preceding sequence.

Application Advantage: Capable of mapping highly complex, non-linear relationships and feature interactions (e.g., capturing the specific synergistic effect of 'Monday' combined with 'Rush Hour').

ML Models for Forecasting

Method

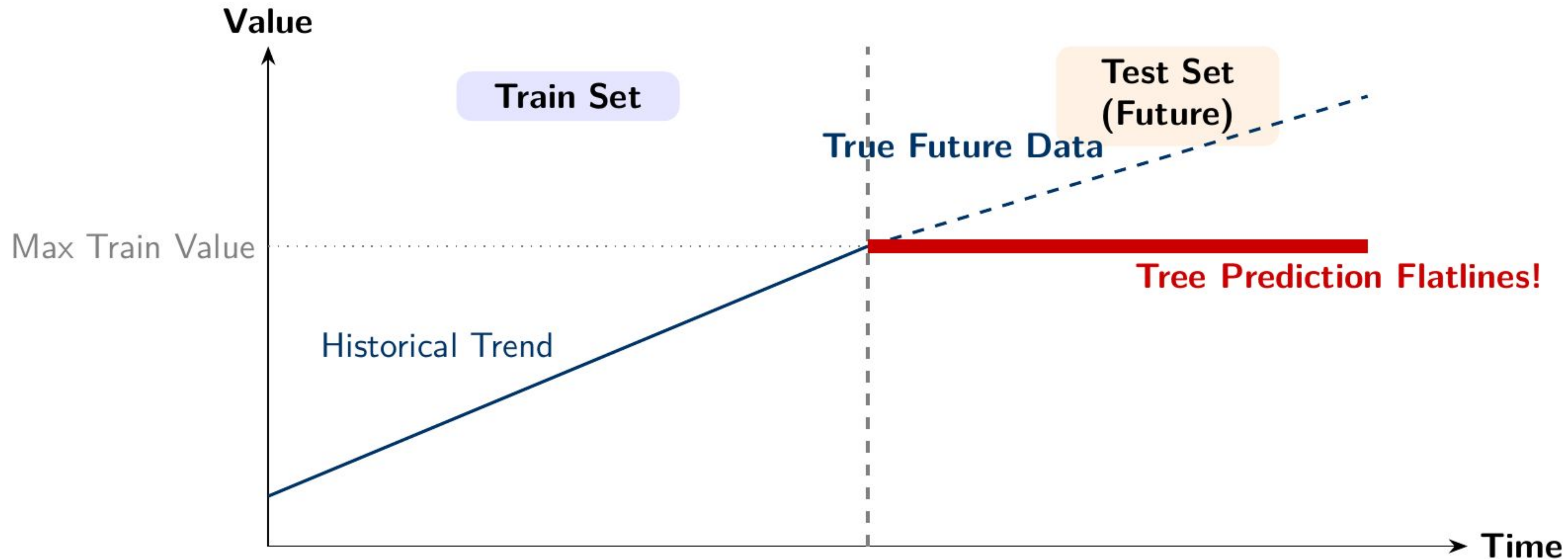
1. **Random Forest (RF)**
   - *Ensemble method:* Builds hundreds of independent decision trees in parallel.
   - Reduces overfitting by averaging predictions.
2. **Gradient Boosting (XGBoost, LightGBM)**
   - *Sequential method:* Trees are built one after another. Each new tree corrects the errors (residuals) of the previous ones.
   - State-of-the-art for tabular forecasting.

```python
import xgboost as xgb

# Define XGBoost Regressor
model_xgb = xgb.XGBRegressor(
    n_estimators=100,
    learning_rate=0.1,
    max_depth=5
)

# Train the model
model_xgb.fit(X_train, y_train)

# Predict
y_pred_xgb = model_xgb.predict(X_test)
```

ML Models for Forecasting

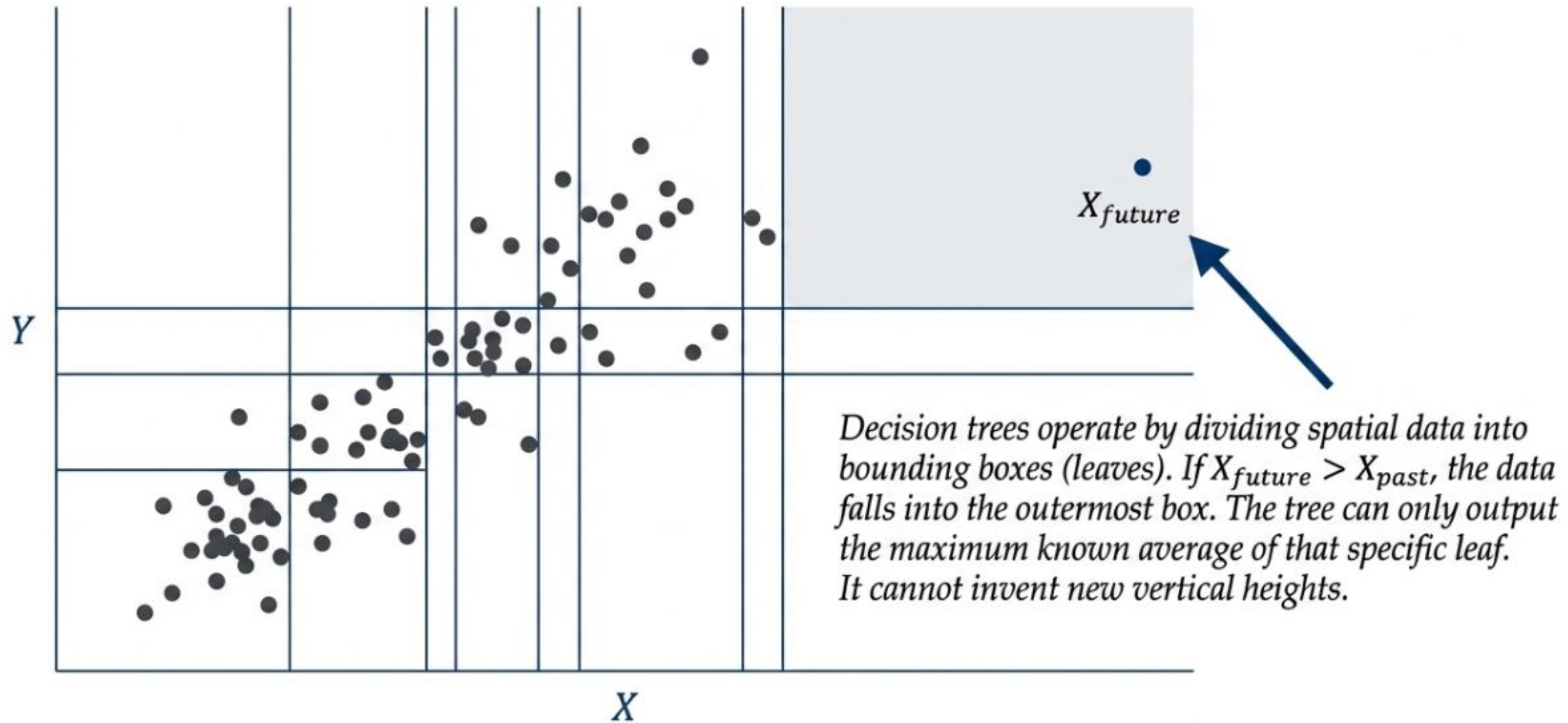
Definition

Decision trees work by splitting the data space based on historical thresholds. **They cannot predict values higher (or lower) than what they have seen in the training data.**



ML Models for Forecasting

Method



ML Models for Forecasting

Method

Experimental Methodology: Observing the flatline in production

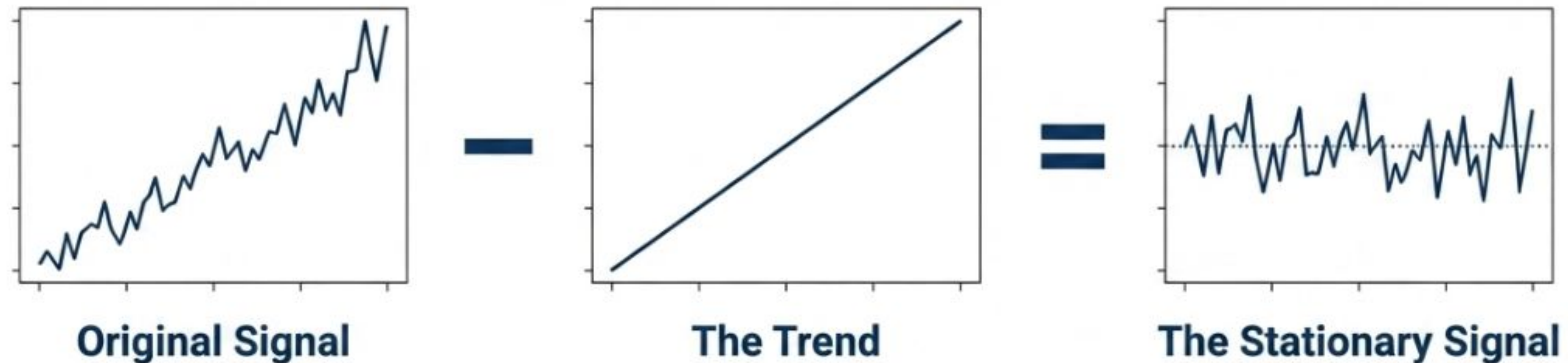
```
import numpy as np
from xgboost import XGBRegressor

# Generate a simple upward linear trend: y = 2x
X_train = np.array([1, 2, 3, 4, 5]).reshape(-1, 1)
y_train = np.array([2, 4, 6, 8, 10])

# Train the tree model
model = XGBRegressor().fit(X_train, y_train)

# Attempt to predict strictly future, unseen values
X_future = np.array([6, 7, 8, 9, 10]).reshape(-1, 1)
predictions = model.predict(X_future)
```

```
Output: [10. 10. 10. 10. 10.]
```

The model predicts 10 for all future time steps, simply repeating the highest y value it observed in the training set.

ML Models for Forecasting

Method



The Fix: Tree models require stationary targets. We must manually remove the trend (via detrending or differencing) before feeding data into the model.

The Reversal: After the model predicts the stationary future, the original trend is mathematically added back to the prediction to yield the final forecast.

ML Models for Forecasting

Method

Algorithmic Correction: Detrending & The Leakage Trap

CRITICAL RULE: Difference **FIRST**, Engineer Features **SECOND**

To use Tree models on trending data, we must remove the trend via Differencing. **Warning:** If you create lag features on raw data and difference them later, you risk severe **Data Leakage** and logic errors.

Step 1: The Correct Order

```
# 1. Take difference FIRST to make it stationary
df['y_diff'] = df['y'].diff()

# 2. Engineer features SECOND
# (Strictly use the DIFFERENCED data)
X_features = create_features(df[['y_diff']])
# DO NOT create lags on 'y' first!

# 3. Train XGBoost on y_diff
model_xgb.fit(X_train_diff, y_train_diff)

# 4. Predict the FUTURE DIFFERENCES
diff_pred = model_xgb.predict(X_test_diff)
```

Step 2: Reconstruct the Trend

Add the predicted differences cumulatively back to the *last known real value*.

```
import numpy as np

# Retrieve the anchor point
# (Last known real value from train set)
last_val = df['y'].iloc[train_end_idx]

# Cumulative sum of predicted changes
# y_t = y_{t-1} + diff_pred
y_pred_final = last_val + np.cumsum(diff_pred)
```

ML Models for Forecasting

Method

Architecture diagnostic: Model capability matrix

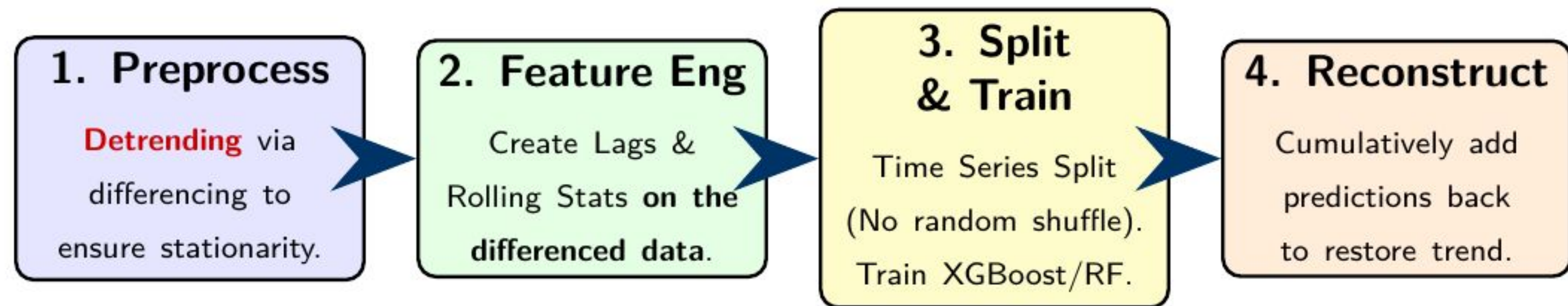
Dimension	Linear Regression	Random Forest	Gradient Boosting
Non-Linearity	Poor - Linear only	Excellent	Excellent - captures complex interactions
Exogenous Variables	Requires strict feature selection	Robust with hundreds of variables	Highly efficient feature utilization
Outlier Handling	Highly sensitive to noise	Extremely durable	Durable, depends on loss function
Extrapolation	Native capability	Fails - Flatlines	Fails - Flatlines

ML Models for Forecasting

Method

Deployment Architecture: The Production Pipeline

A strictly ordered blueprint for Time Series Machine Learning



Summary Code Execution:

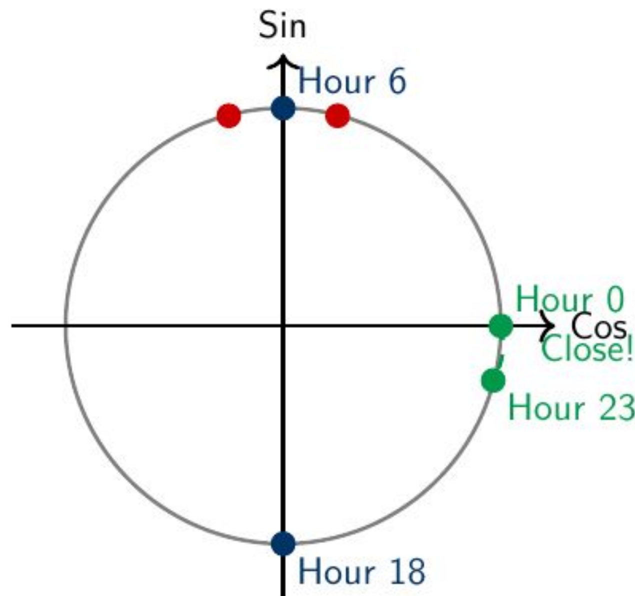
```
# 1. STATIONARITY FIRST
df['target_diff'] = df['target'].diff()

# 2. FEATURES SECOND
X_matrix = generate_lags_and_rolling(df['target_diff'])

# 3. CHRONOLOGICAL TRAINING
model = XGBRegressor().fit(X_train_diff, y_train_diff)
pred_diff = model.predict(X_test_diff)

# 4. RECONSTRUCTION
final_forecast = df['target'].iloc[last_train_index] + np.cumsum(pred_diff)
```

ML Models for Forecasting

Advanced Technique

Features Engine

1. The Cyclical Time Problem
To a standard model, Hour 23 and Hour 0 seem mathematically far apart (23 vs 0). In reality, they are 1 hour apart.
Solution: Map time to a circle using **Sine** and **Cosine** functions.



Python Implementation

```
import numpy as np

# 1. Cyclical Transformations
# Transform Hour (0-23) into Sin/Cos waves
df['hour_sin'] = np.sin(2 * np.pi * df['hour']/24)
df['hour_cos'] = np.cos(2 * np.pi * df['hour']/24)

# 2. Interaction Features
# Creating "Lags of Lags" (Acceleration)
df['momentum'] = df['lag_1'] - df['lag_2']

# Target encoding (Expanding mean to avoid leakage)
df['day_avg'] = df.groupby('day_of_week')['y']\
                 .transform(lambda x: x.expanding
                     ().mean())
```

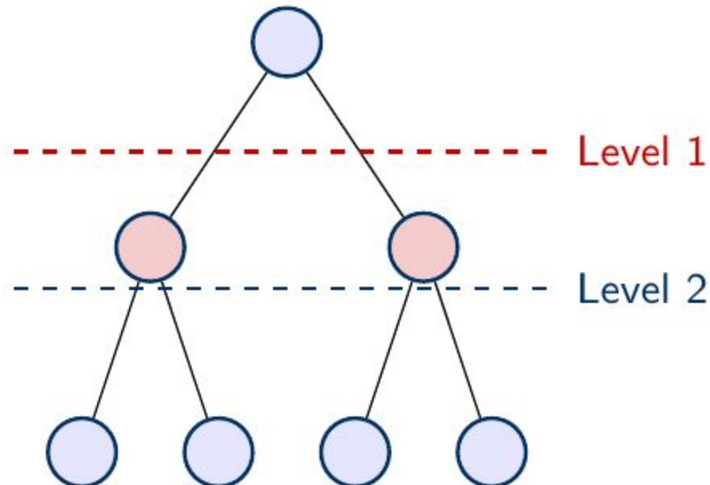
ML Models for Forecasting

Advanced Technique

Model

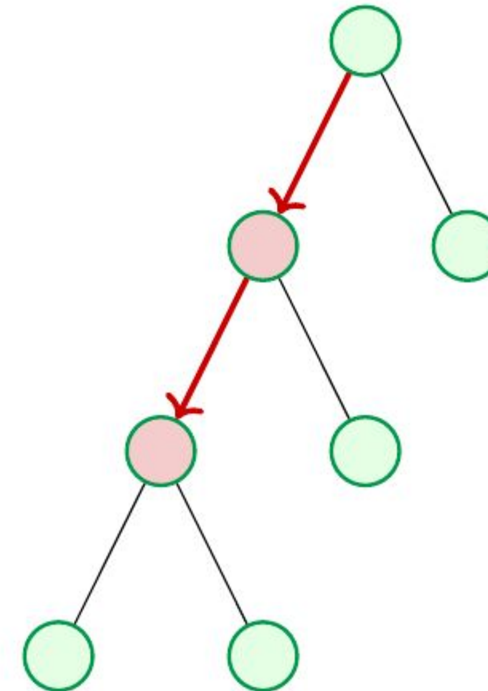
XGBoost: Level-wise Growth

Grows horizontally layer by layer. Highly stable, resists overfitting on smaller datasets.



LightGBM: Leaf-wise Growth

Splits the leaf with the *max loss reduction*. Faster, better for massive data, but needs `max_depth`.



ML Models for Forecasting

Advanced Technique

Model

Native Categorical Handling & Early Stopping

1. Native Categorical Handling
Unlike XGBoost which strictly requires One-Hot Encoding (making data huge), LGBM handles categoricals natively by grouping categories into optimal buckets (Exclusive Feature Bundling).
→ *Massive memory and speed boost.*

2. Early Stopping
Train until the model stops improving on the **Validation Set**. This completely automates the tuning of `n_estimators` (number of trees) and prevents overfitting.

```
import lightgbm as lgb
from lightgbm import early_stopping

# 1. Native Categoricals: No One-Hot needed
# Just tell LGBM which columns are categorical
cat_features = ['day_of_week', 'is_holiday']

train_data = lgb.Dataset(X_train, label=y_train,
                         categorical_feature=
                             cat_features)
val_data = lgb.Dataset(X_val, label=y_val)

# 2. Train with Early Stopping
model = lgb.train(
    params={'objective': 'regression',
            'learning_rate': 0.05},
    train_set=train_data,
    valid_sets=[train_data, val_data],
    # Stop if no improvement after 50 trees
    callbacks=[early_stopping(stopping_rounds=50)]
)
```

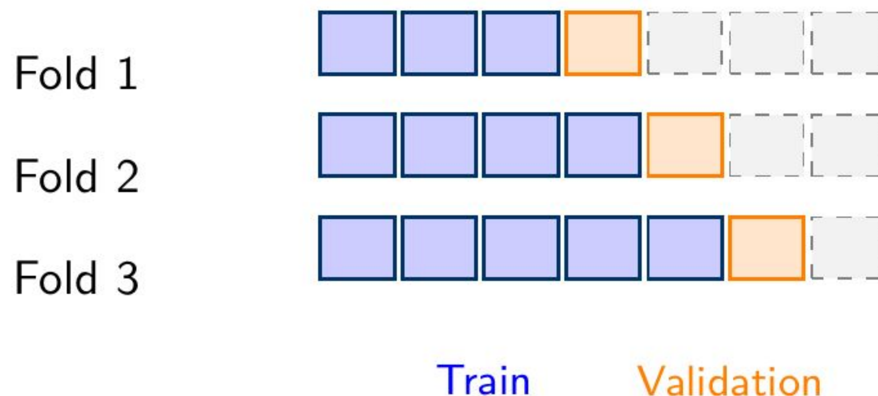
ML Models for Forecasting

Advanced Technique

Validation

The Leakage Problem:
Standard K-Fold Cross Validation shuffles data randomly. In time series, this means using **tomorrow** to predict **yesterday**.

Solution: Rolling-Origin
Strictly respect chronological order. Train on $[1 \ldots T]$, validate on $[T+1 \ldots T+H]$. Expand the training set, and repeat.




```python
from sklearn.model_selection import TimeSeriesSplit
import xgboost as xgb

# Define 5 rolling chronological splits
tscv = TimeSeriesSplit(n_splits=5)

model = xgb.XGBRegressor()
scores = []

# Manually loop to avoid leakage
for train_idx, val_idx in tscv.split(X):
    X_train, X_val = X.iloc[train_idx], X.iloc[
        val_idx]
    y_train, y_val = y.iloc[train_idx], y.iloc[
        val_idx]

    model.fit(X_train, y_train)
    preds = model.predict(X_val)
    scores.append(mean_absolute_error(y_val, preds)
        )

print("Average MAE:", np.mean(scores))
```

ML Models for Forecasting

Advanced Technique

Optimization

Bayesian Optimization vs. Grid Search

Why not Grid Search?
Testing every combination of 5 hyperparameters takes days. Grid Search is exhaustive but "blind".

The Kaggle Standard: Optuna
Optuna uses **Bayesian Optimization** (TPE algorithm). It looks at the history of previous trials to mathematically "guess" the most promising regions of the parameter space.

Key Parameters to Tune:

- `learning_rate` & `n_estimators`
- `max_depth` (controls complexity)
- `subsample` (adds randomness)

```python
import optuna
from sklearn.metrics import mean_squared_error

def objective(trial):
    # 1. Define search space
    params = {
        'max_depth': trial.suggest_int('max_depth', 3, 9),
        'learning_rate': trial.suggest_float('learning_rate', 0.01, 0.1),
        'n_estimators': trial.suggest_int('n_estimators', 100, 500)
    }

    # 2. Evaluate using TimeSeriesSplit
    mse_list = []
    for train_idx, val_idx in tscv.split(X):
        # ... split data ...
        model = xgb.XGBRegressor(**params).fit(X_train, y_train)
        mse_list.append(mean_squared_error(y_val, model.predict(X_val)))

    return np.mean(mse_list) # Optuna minimizes this return value

# 3. Run Optimization
study = optuna.create_study(direction='minimize')
study.optimize(objective, n_trials=30)
print(study.best_params)
```

ML Models for Forecasting

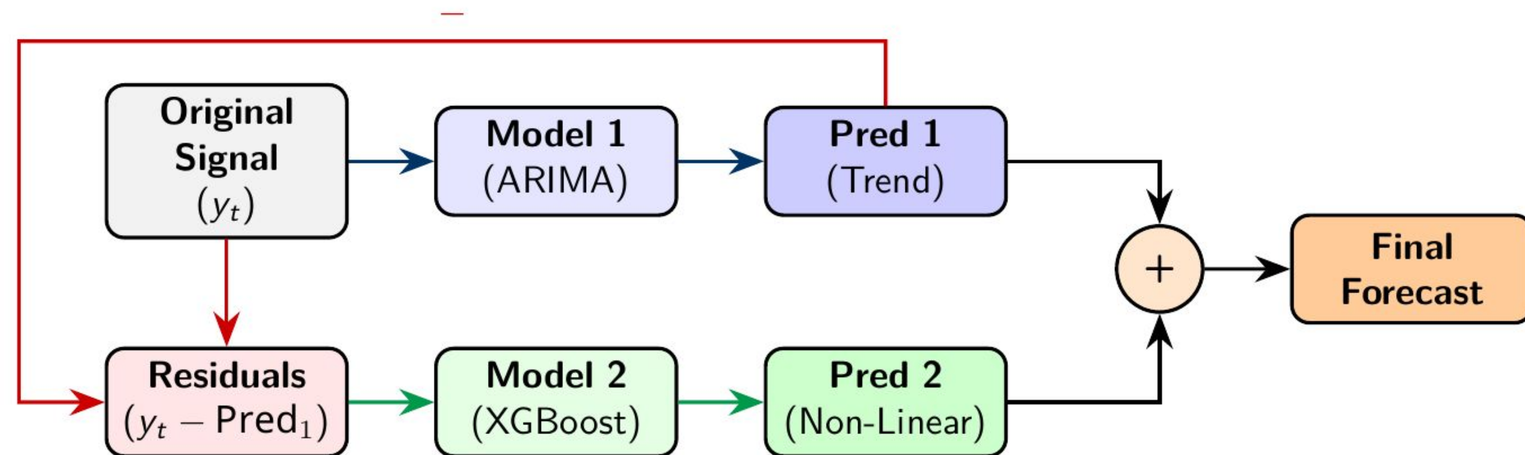
Advanced Technique

Optimization

Ensemble Strategies: Hybrid Residual Modeling

Combining Statistics (Linear) and Machine Learning (Non-Linear)

Statistical models (ARIMA) are excellent at predicting **Trend**. Tree models (XGBoost) are excellent at capturing complex **Non-linear interactions**. *Let's combine them.*



- **Alternative Strategy (Weighted Averaging):** Final $= (0.6\times$ LGBM$) + (0.4\times$ XGBoost$)$. Weights are determined based on Validation Set performance.

ML Models for Forecasting

Advanced Technique

Optimization

Summary Diagnostic Matrix: Choosing Your Weapon

A quick guide for production deployment

Dimension	XGBoost	LightGBM	CatBoost
Growth Strategy	Level-wise (Balanced)	Leaf-wise (Deep)	Symmetric (Oblivious)
Speed & RAM	Slower / High RAM	**Fastest / Low RAM**	Moderate / High RAM
Categorical Data	Needs One-Hot Encoding	Handled Natively	**Superior Native Handling**
Best Use Case	Smaller datasets, high stability required.	Massive datasets, need fast experimentation.	High cardinality data (many categories).

Final Takeaway for Kaggle / Industry:

1. Start with **LightGBM** for fast Feature Engineering iteration.
2. Use **Optuna** + **Time-Series Split** for tuning.
3. **Ensemble** (Average or Stack) multiple models for the final prediction.

Content

- ARCH/GARCH
- Time Series as Supervised Learning
- ML Models for Forecasting
- **Temporal Validation Strategies**

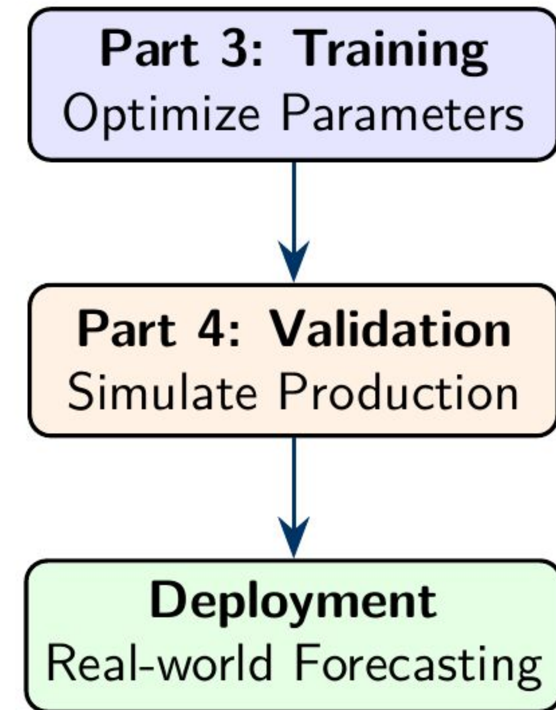
Temporal Validation Strategies

Optimization

What we have achieved:

- We avoided Data Leakage during hyperparameter tuning by replacing random K-Fold with **TimeSeriesSplit**.
- We automated the search for optimal tree depth and learning rates using **Optuna**.

The Next Challenge: A model that scores well during historical validation might still fail in real-world production. We need to simulate the *exact deployment environment* and measure the *direction of errors*, not just the magnitude.



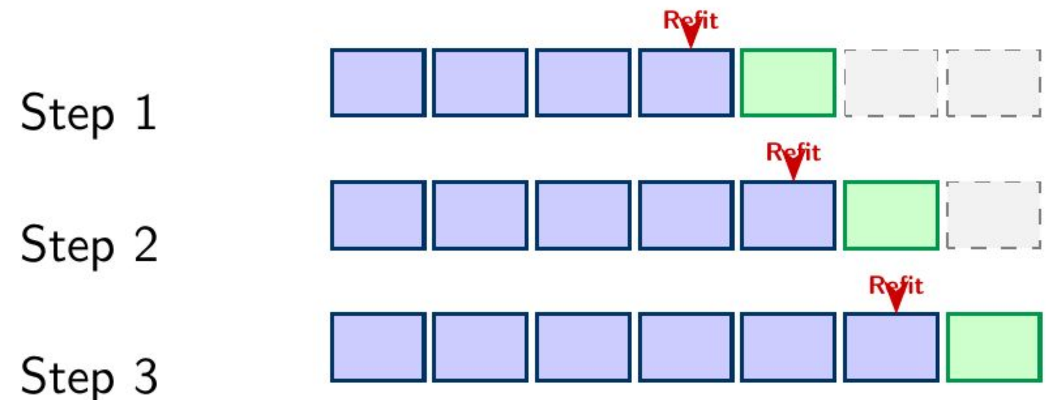
Temporal Validation Strategies

Optimization

The Production Reality:
In business, we don't train a model once and use it forever. Every day, new data arrives. We must **refit (retrain)** the model continuously to capture the latest market shocks.

Walk-Forward Mechanism:

1. Train on Day $1 \dots T$.
2. Predict Day $T + 1$.
3. *Observe true* $T + 1$. Add it to the training set. **Refit model.**
4. Predict Day $T + 2$. Repeat.



Temporal Validation Strategies

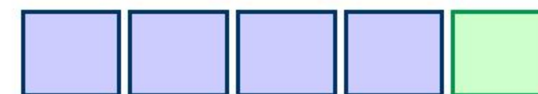
Optimization

The Operational Delay Problem:
If we predict Thursday's sales, standard CV assumes we have complete data up to Wednesday night. In reality, Wednesday's reports might not be processed until Friday.

Danger: Evaluating without a gap creates overly optimistic models.

The Solution: Gap Integration
Force a blind spot between the Train and Test sets to perfectly mimic the real-world operational delay.

Standard



Data Delay

With Gap

```
from sklearn.model_selection import TimeSeriesSplit

# Gap = 2 means skipping 2 time steps before testing
tscv = TimeSeriesSplit(n_splits=3, gap=2)

for train_idx, val_idx in tscv.split(X):
    # Train set ends. 2 steps skipped. Val set begins.
    print(f"Train end: {train_idx[-1]}, Val start: {val_idx[0]}")
```

Thank you