Towards AI

Member-only story

Adaptive Decay-Weighted ARMA: A Novel Approach to Time Series Forecasting

Integrating Recency-Based Loss Weighting and Seasonal Feature Tuning for Enhanced Predictive Accuracy

Shenggang Li · Following

Published in Towards AI · 20 min read · 1 day ago

27

Photo by Haberdoedas II on Unsplash

Introduction

Time series forecasting is both fascinating and challenging. It's fascinating because accurate predictions can directly inform better decisions — whether it's managing electricity demand, planning inventory, or making investment moves. The ability to anticipate future values based on past patterns is a powerful tool in many fields. But forecasting is also hard, mainly because time can shift trends, disrupt cycles, and introduce new behaviors that make modeling complex.

Traditional models like *AR(k)* or *ARMA* have long been used for this task. However, one key limitation is that they treat all past observations equally, assuming each lag contributes the same to predicting the future. In practice,

though, recent values are often more relevant than older ones — something these classical models tend to overlook.

I now proposed a new method called Adaptive Decay-Weighted *ARMA*. The idea is simple but powerful: I adjusted the usual loss function by adding a decay weighting function, *F(sequence_j | A)*, which assigns higher importance to recent observations and less to distant ones. The parameter *A* controls how fast this decay happens. For example, the most recent value (*sequence_1*) always gets full weight, and older values fade depending on the decay pattern. What's more, I don't fix *A*— can be learned it the data using optimization techniques. This lets the model adjust to the natural dynamics of the series it's forecasting.

This approach extends the *ARMA* family in a flexible way. It not only improves performance by respecting the recency of data but also leaves room for further upgrades, like adding non-linear fitting methods or dynamic decay adjustments. Through experiments on real-world time series data, I compare the method against Normal *AR*, *ARMA(1,1)* with seasonal dummies, and *AR* with cycle-only features. The results clearly show that Adaptive Decay-Weighted *AR* consistently achieves lower *MAPE* scores, highlighting its practical advantage and forecasting accuracy.

Algorithmic Mechanisms and Procedure

Baseline: Weighted Yule-Type Autoregressive Model

I begin by expressing a generalized time series forecasting model in the following form:

$$X_{n+1} = F\left((X_n, X_{n-1}, \ldots, \text{Cycle}_n, \text{MA}_n) \mid A\right)$$

In the classical autoregressive model of order *p*, the process is defined as:

$$Y_t = \phi_1 Y_{t-1} + \phi_2 Y_{t-2} + \cdots + \phi_p Y_{t-p} + \epsilon_t$$

where *ε_t* denotes the error term. This formulation assumes all observations contribute equally to model estimation.

In contrast, the weighted Yule-type formulation introduces a time-dependent weight function *w(t;A)*, typically a monotonic decay function parameterized by *A*, to emphasize the importance of recent observations. This leads to a modified, weighted loss function, where the mean squared error (*MSE*) is calculated as:

$$\mathcal{L}(\boldsymbol{\phi}) = \sum_t w(t; A) \left(Y_t - \sum_{i=1}^{p} \phi_i Y_{t-i} \right)^2$$

This decay-based weighting enables the model to give more influence to recent errors while down-weighting older observations, making the autoregressive process more responsive to temporal dynamics.

Choice of F(x | A): The Decay Function

This is a key aspect of the model design, since defining a meaningful weighting scheme for recent versus distant observations requires a decay function $F(x \mid A)$ meeting the following properties:

- Positivity: $F(x \mid A)>0$ for all x
- Monotonicity: Decreasing in x, so that more recent observations (lower x) receive higher weight
- Boundedness or Normalizability: Preferably bounded or normalizable to ensure numerical stability

Based on empirical testing, I can show that an effective choice is the exponential decay function, where the weight assigned to time index t is defined as:

$$w(t;\alpha) = \alpha^{(T-t)}, \quad \text{with } 0 < \alpha \leq 1$$

Here, α controls the decay rate, and T is the most recent time point. As t approaches T, the exponent $(T-t)$ becomes smaller, giving more recent observations higher weight. This captures the intuition that newer data points are typically more informative for prediction.

Another way to express exponential decay in lag space is:

$$F(j \mid A) = A^{j-1}, \quad \text{for } j = 1, 2, \ldots, p$$

This assigns a weight of 1 to the most recent lag (when *j=1*) and decays geometrically for older lags. This formulation is differentiable, stable, and easy to optimize. I can also normalize the weights to sum to one:

$$\tilde{F}(j \mid A) = \frac{A^{j-1}}{\sum_{k=1}^{p} A^{k-1}}$$

Other candidate functions include:

Power Decay:

$$F(j \mid A) = j^{-A}, \text{where } A > 0$$

Where *A>0*. This provides a slower, heavier-tailed decay that retains more influence from distant past values.

Softmax-Weighted Attention:

$$F(j \mid A) = \frac{\exp(-A \cdot j)}{\sum_{k=1}^{p} \exp(-A \cdot k)}$$

This form ensures the weights sum to 1 and can be interpreted as soft attention over past lags, similar to mechanisms used in neural networks. It offers a smooth and tunable way to balance short- and long-term memory.

These decay functions give the model flexibility to adjust how quickly it discounts older data, enabling better adaptation to diverse time series dynamics.

Incorporating a Moving Average (MA) Component

The moving average (*MA*) item aims to smooth short-term fluctuations. Define a moving average over a window of L periods by

$$MA_t = \frac{1}{L} \sum_{j=1}^{L} Y_{t-j}$$

or, more generally, a weighted moving average:

$$MA_t = \sum_{j=1}^{L} \psi_j Y_{t-j}$$

where ψ_j can be pre-specified. Incorporate the moving average into the model using an additional coefficient γ:

$$Y_t = \sum_{i=1}^{p} \phi_i Y_{t-i} + \gamma\, MA_t + \epsilon_t$$

The loss function now becomes:

$$\mathcal{L}(\boldsymbol{\theta}) = \sum_{t} w(t;\alpha) \left(Y_t - \sum_{i=1}^{p} \phi_i Y_{t-i} - \gamma\, MA_t \right)^2$$

with $\theta = (\phi\_1,\ldots,\phi\_p,\gamma)$.

Adding Seasonality/Cycle Information

Seasonal patterns can be introduced via dummy variables or Fourier terms. For example, if there is monthly seasonality (with period mmm) I will define seasonal dummies:

$$S_{tj} = \begin{cases} 1 & \text{if observation } t \text{ is in month } j, \\ 0 & \text{otherwise,} \end{cases} \qquad j = 1, \ldots, m-1$$

or I can use sinusoidal terms such as:

$$\sin\left(\frac{2\pi t}{m}\right), \quad \cos\left(\frac{2\pi t}{m}\right)$$

Then I augment the *AR* model with a seasonal term:

$$Y_t = \sum_{i=1}^{p} \phi_i Y_{t-i} + \gamma\, MA_t + \boldsymbol{\delta}^\top \mathbf{S}_t + \epsilon_t$$

where *S_t* is the vector of seasonal indicators at time *t* (or Fourier terms) and δ are the corresponding coefficients.

Now I obtain the corresponding weighted loss function:

$$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\delta}) = \sum_{t} w(t;\alpha) \left(Y_t - \sum_{i=1}^{p} \phi_i Y_{t-i} - \gamma\, MA_t - \boldsymbol{\delta}^\top \mathbf{S}_t \right)^2$$

Note, in matrix form, if I define the regressor vector as:

$$\mathbf{X}_t = \left[1,\ Y_{t-1}, \ldots, Y_{t-p},\ MA_t,\ S_{t1}, \ldots, S_{t(m-1)}\right]$$

and the coefficient vector:

$$\boldsymbol{\beta} = \left[\beta_0,\ \phi_1, \ldots, \phi_p,\ \gamma,\ \delta_1, \ldots, \delta_{m-1}\right]^\top$$

, then the prediction is:

$$\hat{Y}_t = \mathbf{X}_t^\top \boldsymbol{\beta}$$

and the weighted loss is:

$$\mathcal{L}(\boldsymbol{\beta}) = \sum_t w(t;\alpha)\left(Y_t - \mathbf{X}_t^\top \boldsymbol{\beta}\right)^2$$

Estimation and Loss Minimization Procedure (Algorithm)

1. **Construct the Design Matrix:**

- For each time *t* from the start of the usable data (after creating lags), compute the lagged values *Y_{t-1}, ...Y_{t-p}*.
- Compute the moving average *MA_t* using a pre-specified window (or weighted averages).
- Construct the seasonal indicators *S_t* (either as dummies or Fourier terms).
- Form the vector *X_t* including a constant.

2. Define the Weighting Function:

I choose an exponential decay function:

$$w(t;\alpha) = \alpha^{T-t}$$

where T is the index of the most recent observation. Alternatively, I can use linear or other forms.

3. Loss Function:

My objective is to minimize the weighted sum of squared residuals:

$$\min_{\boldsymbol{\beta}} \quad \mathcal{L}(\boldsymbol{\beta}) = \sum_{t=p+L}^{T} w(t;\alpha)\left(Y_t - \mathbf{X}_t^\top \boldsymbol{\beta}\right)^2$$

In a gradient-based framework (e.g., if combined with neural network components), this loss is differentiable with respect to both the regression coefficients β and, if desired, the decay parameter α (although α is often chosen via cross-validation or grid search to maintain convexity).

4. Parameter Estimation:

This paper evaluates two approaches for parameter estimation through code experiments:

Weighted Least Squares (*WLS*):
The solution for β is given by the closed-form expression:

$$\hat{\boldsymbol{\beta}} = \left(\sum_t w(t;\alpha)\mathbf{X}_t\mathbf{X}_t^\top\right)^{-1}\left(\sum_t w(t;\alpha)\mathbf{X}_t Y_t\right)$$

Gradient-Based Optimization (Learning the Decay Factor):

Alternatively, by incorporating the weighted loss function into a neural network or an iterative optimization framework (e.g., stochastic gradient descent), I can directly minimize the loss and update the model parameters β iteratively:

$$\mathcal{L}(\boldsymbol{\beta}) = \sum_t w(t;\alpha)\left(Y_t - \mathbf{X}_t^\top \boldsymbol{\beta}\right)^2$$

Importantly, the decay factor is not fixed but learned during training, allowing the model to adaptively emphasize more recent observations. As demonstrated in the experimental results, this learned decay factor improves forecasting performance for upcoming time points.

5. Forecasting:

Once β^ is estimated, future forecasts are generated by constructing the predictor vector X_{T+h} using the most recent observed values — and, if forecasting iteratively, previously predicted values for lagged terms. This allows the model to produce step-ahead forecasts efficiently. We can forecast for time $T+h$:

$$\hat{Y}_{T+h} = \mathbf{X}_{T+h}^\top \hat{\boldsymbol{\beta}}$$

If forecasting over a horizon of K days, compute the average forecast:

$$\text{Forecast Average} = \frac{1}{K} \sum_{h=1}^{K} \hat{Y}_{T+h}$$

6. Model Selection and Tuning:

- The decay parameter α (or parameters defining the decay function) can be tuned by evaluating out-of-sample performance (e.g., minimizing *MAPE* on a validation set).
- Additional hyperparameters (such as the moving average window L or choices regarding the seasonal representation) may be tuned via cross-validation.

By adding these extra features, the model becomes more flexible — it can focus more on recent trends through decay weighting, smooth out short-term fluctuations using moving averages, and adjust for seasonal or cyclical patterns. All of this helps improve forecasting accuracy in a practical way.

Empirical Evaluation with Forecasting Benchmarks

To evaluate the effectiveness of the proposed Decay-Weighted AR Forecast method, I first conducted a series of forecasting experiments using the publicly available *Electric_Production_tm.csv* dataset. This dataset records monthly U.S. electricity production, making it well-suited for testing autoregressive models with seasonality and moving average structures.

In this first test, I used a manually tuned (fixed) decay factor, not the learned version. The goal was to compare the fixed decay-weighted approach against other standard and enhanced *AR*-based forecasting methods. This experiment includes four models:

1. Decay-Weighted *AR* Forecast (with decay tuning and additional features)
2. Normal *AR* Forecast (with additional features)
3. *ARMA*(*1,1, cycle*) Forecast
4. *AR* with Cycle (no *MA*) Forecast

In the next section, I extend the evaluation by using a **learned decay factor**, where the decay parameter is optimized during training to further improve forecasting accuracy.

Each model is tested on its ability to forecast the average value of the target series '*IPG2211A2N*' over the next 3, 5, 7, and 9 months. I used Mean Absolute Percentage Error (*MAPE*) to measure forecasting accuracy. To keep the evaluation realistic, all models followed a time-based train/test split — so future data is always predicted using only past data. The decay-weighted model includes a tuning step to find the best decay parameter, while the other models use fixed settings. Everything was implemented in Python using the *statsmodels* library, and results were compared across different forecast horizons.

Here is the code:

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

```
import re
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score  # not used in AR forecast but availa
from scipy.stats import ks_2samp
import statsmodels.api as sm

def _prepare_dataframe(DF: pd.DataFrame, S: str, time_unit: str, p: int, ma_wind
    """
    Prepares the dataframe:
      - Converts the time_unit column to datetime and sorts the DataFrame.
      - Creates AR lag features: Y_{t-1}, ..., Y_{t-p}.
      - Creates a moving average (MA) feature over the previous ma_window observ
      - If add_seasonal is True, extracts the month from the time column and cre
    Drops rows with missing predictors.

    Returns:
      df: the augmented DataFrame.
      predictors: a list of predictor column names (in order).
    """
    # Drop rows with missing S or time_unit, and convert time column to datetime
    df = DF.copy().dropna(subset=[S, time_unit]).reset_index(drop=True)
    if not np.issubdtype(df[time_unit].dtype, np.datetime64):
        df[time_unit] = pd.to_datetime(df[time_unit], errors='coerce')
    df.sort_values(by=time_unit, inplace=True)
    df.reset_index(drop=True, inplace=True)

    # Create AR lag features
    for lag in range(1, p + 1):
        df[f"{S}_lag{lag}"] = df[S].shift(lag)

    # Create Moving Average (MA) feature: average of the previous ma_window obse
    df["MA"] = df[S].rolling(window=ma_window, min_periods=ma_window).mean().shi

    # Create seasonal dummies based on month if requested.
    seasonal_cols = []
    if add_seasonal:
        df["month"] = df[time_unit].dt.month
        month_dummies = pd.get_dummies(df["month"], prefix="mon", drop_first=Tru
        df = pd.concat([df, month_dummies], axis=1)
        seasonal_cols = list(month_dummies.columns)

    # Drop rows with missing predictors (lags or MA)
    df.dropna(subset=[f"{S}_lag{p}", "MA"], inplace=True)
    df.reset_index(drop=True, inplace=True)

    # Build list of predictor columns: AR lags + MA + seasonal dummies
    predictors = [f"{S}_lag{i}" for i in range(1, p+1)] + ["MA"] + seasonal_cols

    # Convert all predictor columns explicitly to float.
    df[predictors] = df[predictors].apply(pd.to_numeric, errors='coerce')
```

```
        df[predictors] = df[predictors].astype(float)
        # Drop any rows where predictors are NaN after conversion.
        df.dropna(subset=predictors, inplace=True)
        df.reset_index(drop=True, inplace=True)

        return df, predictors

    def _decay_weighted_forecast_single(
        DF: pd.DataFrame,
        S: str,
        K: int,
        time_unit: str,
        decay_param: float,
        decay_function: str = "exponential",
        p: int = 1,
        ma_window: int = 3,
        add_seasonal: bool = True
    ):
        """
        Internal function to compute decay-weighted AR(p) forecast with additional M
        Returns forecasted average and MAPE over the next K steps.
        """
        # Prepare the DataFrame with predictors.
        df, predictors = _prepare_dataframe(DF, S, time_unit, p, ma_window, add_seas

        # Train/Test Split.
        n = len(df)
        train_size = int(0.8 * n)
        train_df = df.iloc[:train_size].copy()
        test_df = df.iloc[train_size:].copy()
        if len(test_df) < K:
            raise ValueError(f"Not enough test data to produce {K} forecasts.")
        forecast_section = test_df.iloc[:K].copy()

        # Weighted OLS Estimation.
        X_train = train_df[predictors].astype(float)
        y_train = train_df[S].astype(float)
        X_train_const = sm.add_constant(X_train)

        idx = np.arange(len(X_train))
        if decay_function.lower() == "exponential":
            w = decay_param ** ((len(X_train)-1) - idx)
        elif decay_function.lower() == "linear":
            slope = decay_param
            w = 1 + slope * (idx - idx.min())
        else:
            w = np.ones(len(X_train))

        w_sqrt = np.sqrt(w)
        wls_model = sm.WLS(y_train, X_train_const, weights=w_sqrt).fit()
```

```
    # Iterative Forecasting.
    # Maintain a history from the training target series.
    last_known = list(train_df[S].values.astype(float))
    preds = []

    # For each forecast step, build the predictor vector:
    for i in range(K):
        # AR lag features: the last p values.
        ar_vals = np.array(last_known[-p:], dtype=float)
        # MA: computed from the last ma_window values.
        if len(last_known) >= ma_window:
            ma_val = np.mean(last_known[-ma_window:])
        else:
            ma_val = np.mean(last_known)
        # Seasonal features: use forecast_section row i if applicable.
        seasonal_vals = []
        if add_seasonal:
            seasonal_cols = [col for col in predictors if col.startswith("mon_")
            seasonal_vals = forecast_section.iloc[i][seasonal_cols].values.astyp
        # Combine in the order: AR lags, MA, seasonal dummies.
        x_row = np.concatenate([ar_vals, [ma_val], np.array(seasonal_vals, dtype
        # Convert to float type and add constant.
        X_pred = np.array(x_row, dtype=float).reshape(1, -1)
        X_pred = sm.add_constant(X_pred, has_constant='add')
        y_hat = wls_model.predict(X_pred)[0]
        preds.append(y_hat)
        last_known.append(y_hat)

    forecast_section["pred_decay"] = preds
    forecast_avg = forecast_section["pred_decay"].mean()

    def mape(actual, predicted):
        return np.mean(np.abs((actual - predicted) / (actual + 1e-9))) * 100
    mape_val = mape(forecast_section[S].values, forecast_section["pred_decay"].v

    return forecast_avg, mape_val

def decay_weighted_forecast(
    DF: pd.DataFrame,
    S: str,
    K: int,
    time_unit: str,
    decay_function: str = "exponential",
    p: int = 1,
    ma_window: int = 3,
    add_seasonal: bool = True,
    train_frac: float = 0.8,
    tune_decay: bool = False,
    candidate_decay_params: list = None,
```

```
    verbose: bool = True
):
    """
    Decay-weighted AR(p) forecasting with additional MA and seasonal features.
    If tune_decay is True, tries candidate_decay_params (default if None: [0.85,
    and returns the best forecast result (lowest MAPE).

    Returns: (forecasted average, MAPE, chosen_decay_param)
    """
    if tune_decay:
        if candidate_decay_params is None:
            candidate_decay_params = [0.85, 0.90, 0.95]
        best_mape = np.inf
        best_forecast_avg = None
        best_param = None
        for dp in candidate_decay_params:
            forecast_avg, mape_val = _decay_weighted_forecast_single(
                DF, S, K, time_unit, decay_param=dp, decay_function=decay_functi
                p=p, ma_window=ma_window, add_seasonal=add_seasonal
            )
            if verbose:
                print(f"Tuning decay_param={dp:.2f}: Forecast Avg = {forecast_av
            if mape_val < best_mape:
                best_mape = mape_val
                best_forecast_avg = forecast_avg
                best_param = dp
        if verbose:
            print(f"\n=> Selected decay_param={best_param:.2f} with MAPE={best_m
        return best_forecast_avg, best_mape, best_param
    else:
        forecast_avg, mape_val = _decay_weighted_forecast_single(
            DF, S, K, time_unit, decay_param=0.95, decay_function=decay_function
            p=p, ma_window=ma_window, add_seasonal=add_seasonal
        )
        return forecast_avg, mape_val, 0.95

def normal_ar_forecast(
    DF: pd.DataFrame,
    S: str,
    K: int,
    time_unit: str,
    p: int = 1,
    ma_window: int = 3,
    add_seasonal: bool = True,
    train_frac: float = 0.8,
    verbose: bool = True
):
    """
    Normal (unweighted) AR(p) forecasting using standard OLS with additional MA
    Returns forecasted average over K steps and MAPE.
```

```
    """
    # Prepare DataFrame.
    df, predictors = _prepare_dataframe(DF, S, time_unit, p, ma_window, add_seas

    # Train/Test Split.
    n = len(df)
    train_size = int(train_frac * n)
    train_df = df.iloc[:train_size].copy()
    test_df = df.iloc[train_size:].copy()
    if len(test_df) < K:
        raise ValueError(f"Not enough test data to produce {K} forecasts.")
    forecast_section = test_df.iloc[:K].copy()

    # OLS Estimation.
    X_train = train_df[predictors].astype(float)
    y_train = train_df[S].astype(float)
    X_train_const = sm.add_constant(X_train)
    ols_model = sm.OLS(y_train, X_train_const).fit()
    if verbose:
        print("\n--- Normal OLS Summary ---")
        print(ols_model.summary())

    # Iterative Forecasting.
    last_known = list(train_df[S].values.astype(float))
    preds = []
    for i in range(K):
        ar_vals = np.array(last_known[-p:], dtype=float)
        if len(last_known) >= ma_window:
            ma_val = np.mean(last_known[-ma_window:])
        else:
            ma_val = np.mean(last_known)
        seasonal_vals = []
        if add_seasonal:
            seasonal_cols = [col for col in predictors if col.startswith("mon_")
            seasonal_vals = forecast_section.iloc[i][seasonal_cols].values.astyp
        x_row = np.concatenate([ar_vals, [ma_val], np.array(seasonal_vals, dtype
        X_pred = np.array(x_row, dtype=float).reshape(1, -1)
        X_pred = sm.add_constant(X_pred, has_constant='add')
        y_hat = ols_model.predict(X_pred)[0]
        preds.append(y_hat)
        last_known.append(y_hat)

    forecast_section["pred_normal"] = preds
    forecast_avg = forecast_section["pred_normal"].mean()
    mape_val = np.mean(np.abs((forecast_section[S].values - forecast_section["pr
                              (forecast_section[S].values + 1e-9))) * 100
    if verbose:
        print(f"\n[Normal AR] Forecasted Average (over {K} steps): {forecast_avg
        print(f"MAPE: {mape_val:.2f}%")
```

```
    return forecast_avg, mape_val

def arma_forecast(
    DF: pd.DataFrame,
    S: str,
    K: int,
    time_unit: str,
    add_seasonal: bool = True,
    train_frac: float = 0.8,
    verbose: bool = True
):
    """
    ARMA(1,1, cycle) forecasting using statsmodels ARIMA with order (1,0,1).
    If add_seasonal is True, seasonal (month) dummy variables are used as exogen
    Returns forecasted average over K steps and MAPE.
    """
    df = DF.copy().dropna(subset=[S, time_unit])
    if not np.issubdtype(df[time_unit].dtype, np.datetime64):
        df[time_unit] = pd.to_datetime(df[time_unit], errors='coerce')
    df.sort_values(by=time_unit, inplace=True)
    df.reset_index(drop=True, inplace=True)

    exog = None
    if add_seasonal:
        df["month"] = df[time_unit].dt.month
        # Convert seasonal dummies to float to avoid numpy boolean subtraction e
        exog = pd.get_dummies(df["month"], prefix="mon", drop_first=True).astype

    # Split into train/test.
    n = len(df)
    train_size = int(train_frac * n)
    train_df = df.iloc[:train_size].copy()
    test_df = df.iloc[train_size:].copy()
    if len(test_df) < K:
        raise ValueError(f"Not enough test data to produce {K} forecasts.")

    if add_seasonal:
        exog_train = exog.iloc[:train_size]
        exog_test = exog.iloc[train_size: train_size + K]
    else:
        exog_train = None
        exog_test = None

    # Fit ARMA model using ARIMA with order=(1,0,1) (since no differencing is ap
    arma_model = sm.tsa.ARIMA(train_df[S], order=(1,0,1), exog=exog_train).fit()
    if verbose:
        print("\n--- ARMA(1,1, cycle) Model Summary ---")
        print(arma_model.summary())

    # Forecast next K steps.
```

```
        forecast = arma_model.forecast(steps=K, exog=exog_test)
        forecast_vals = forecast.values.astype(float)
        actual = test_df[S].iloc[:K].values.astype(float)
        mape_val = np.mean(np.abs((actual - forecast_vals) / (actual + 1e-9))) * 100
        forecast_avg = np.mean(forecast_vals)

        return forecast_avg, mape_val

    def cycle_ar_forecast(
        DF: pd.DataFrame,
        S: str,
        K: int,
        time_unit: str,
        p: int = 1,
        add_seasonal: bool = True,
        train_frac: float = 0.8,
        verbose: bool = True
    ):
        """
        AR with cycle forecasting (no MA) using standard OLS on AR lag features and
        Returns forecasted average over K steps and MAPE.
        """
        # Prepare the DataFrame using the standard preparation, then remove the MA c
        df, predictors = _prepare_dataframe(DF, S, time_unit, p, ma_window=3, add_se
        if "MA" in predictors:
            predictors.remove("MA")
            df.drop(columns=["MA"], inplace=True)

        # Train/Test Split.
        n = len(df)
        train_size = int(train_frac * n)
        train_df = df.iloc[:train_size].copy()
        test_df = df.iloc[train_size:].copy()
        if len(test_df) < K:
            raise ValueError(f"Not enough test data to produce {K} forecasts.")
        forecast_section = test_df.iloc[:K].copy()

        # OLS Estimation.
        X_train = train_df[predictors].astype(float)
        y_train = train_df[S].astype(float)
        X_train_const = sm.add_constant(X_train)
        ols_model = sm.OLS(y_train, X_train_const).fit()
        if verbose:
            print("\n--- AR with Cycle (no MA) OLS Summary ---")
            print(ols_model.summary())

        # Iterative Forecasting.
        last_known = list(train_df[S].values.astype(float))
        preds = []
        for i in range(K):
```

```
            # AR lag features: the last p values.
            ar_vals = np.array(last_known[-p:], dtype=float)
            # In this model, no MA value is used.
            seasonal_vals = []
            if add_seasonal:
                seasonal_cols = [col for col in predictors if col.startswith("mon_")
                seasonal_vals = forecast_section.iloc[i][seasonal_cols].values.astyp
            # Combine AR lags and seasonal dummies.
            if add_seasonal:
                x_row = np.concatenate([ar_vals, np.array(seasonal_vals, dtype=float
            else:
                x_row = ar_vals
            X_pred = np.array(x_row, dtype=float).reshape(1, -1)
            X_pred = sm.add_constant(X_pred, has_constant='add')
            y_hat = ols_model.predict(X_pred)[0]
            preds.append(y_hat)
            last_known.append(y_hat)

        forecast_section["pred_cycle_ar"] = preds
        forecast_avg = forecast_section["pred_cycle_ar"].mean()
        mape_val = np.mean(np.abs((forecast_section[S].values - forecast_section["pr
                                   (forecast_section[S].values + 1e-9))) * 100
        if verbose:
            print(f"\n[AR with Cycle (no MA)] Forecasted Average (over {K} steps): {
            print(f"MAPE: {mape_val:.2f}%")

        return forecast_avg, mape_val

    # === Execution Code Using Electric_Production_tm.csv ===
    if __name__ == "__main__":
        # Load dataset (expects columns: 'DATE', 'IPG2211A2N')
        df = pd.read_csv("Electric_Production_tm.csv")  # file with columns ['DATE',

        # Forecast horizons to compare.
        forecast_horizons = [3, 5, 7, 9]
        series_name = "IPG2211A2N"
        time_col = "DATE"
        ar_order = 1          # AR(1)
        ma_window = 3         # moving average window size
        add_seasonal = True   # include seasonal (month) dummies

        # Containers for results.
        results_decay = {}
        results_normal = {}
        results_arma = {}
        results_cycle_ar = {}

        print("=== Decay-Weighted AR Forecasts (with decay tuning and additional fea
        for K in forecast_horizons:
            fc_avg, mape_decay, chosen_dp = decay_weighted_forecast(
```

```
            DF=df,
            S=series_name,
            K=K,
            time_unit=time_col,
            decay_function="exponential",
            p=ar_order,
            ma_window=ma_window,
            add_seasonal=add_seasonal,
            train_frac=0.8,
            tune_decay=True,           # enable tuning
            candidate_decay_params=[0.85, 0.90, 0.95],
            verbose=True
        )
        results_decay[K] = (fc_avg, mape_decay, chosen_dp)
        print(f"Horizon {K} days: Forecast Avg = {fc_avg:.4f}, MAPE = {mape_deca

    print("=== Normal AR Forecasts (with additional features) ===")
    for K in forecast_horizons:
        fc_avg_norm, mape_norm = normal_ar_forecast(
            DF=df,
            S=series_name,
            K=K,
            time_unit=time_col,
            p=ar_order,
            ma_window=ma_window,
            add_seasonal=add_seasonal,
            train_frac=0.8,
            verbose=False
        )
        results_normal[K] = (fc_avg_norm, mape_norm)
        print(f"Horizon {K} days: Forecast Avg = {fc_avg_norm:.4f}, MAPE = {mape

    print("=== ARMA(1,1, cycle) Forecasts ===")
    for K in forecast_horizons:
        fc_avg_arma, mape_arma = arma_forecast(
            DF=df,
            S=series_name,
            K=K,
            time_unit=time_col,
            add_seasonal=add_seasonal,
            train_frac=0.8,
            verbose=False
        )
        results_arma[K] = (fc_avg_arma, mape_arma)
        print(f"Horizon {K} days: Forecast Avg = {fc_avg_arma:.4f}, MAPE = {mape

    print("=== AR with Cycle Forecasts (no MA) ===")
    for K in forecast_horizons:
        fc_avg_cycle, mape_cycle = cycle_ar_forecast(
            DF=df,
```

```
            S=series_name,
            K=K,
            time_unit=time_col,
            p=ar_order,
            add_seasonal=add_seasonal,
            train_frac=0.8,
            verbose=False
        )
        results_cycle_ar[K] = (fc_avg_cycle, mape_cycle)
        print(f"Horizon {K} days: Forecast Avg = {fc_avg_cycle:.4f}, MAPE = {map

    # Final comparison.
    print("=== Final Forecast Comparison ===")
    for K in forecast_horizons:
        fc_decay, mape_decay, dp = results_decay[K]
        fc_normal, mape_normal = results_normal[K]
        fc_arma, mape_arma = results_arma[K]
        fc_cycle, mape_cycle = results_cycle_ar[K]
        print(f"For {K} days forecast:")
        print(f"  Decay-Weighted AR: Forecast Avg = {fc_decay:.4f}, MAPE = {mape
        print(f"  Normal AR      : Forecast Avg = {fc_normal:.4f}, MAPE = {mape_
        print(f"  ARMA(1,1, cycle): Forecast Avg = {fc_arma:.4f}, MAPE = {mape_a
        print(f"  AR with Cycle (no MA): Forecast Avg = {fc_cycle:.4f}, MAPE = {
```

Here's what I got:

```
=== Decay-Weighted AR Forecasts (with decay tuning and additional features) ===
Tuning decay_param=0.85: Forecast Avg = 106.6733, MAPE = 2.30%
Tuning decay_param=0.90: Forecast Avg = 106.3578, MAPE = 2.58%
Tuning decay_param=0.95: Forecast Avg = 106.3138, MAPE = 2.61%

=> Selected decay_param=0.85 with MAPE=2.30%
Horizon 3 days: Forecast Avg = 106.6733, MAPE = 2.30%, chosen decay_param = 0.85

Tuning decay_param=0.85: Forecast Avg = 100.2976, MAPE = 1.96%
Tuning decay_param=0.90: Forecast Avg = 100.1658, MAPE = 2.07%
Tuning decay_param=0.95: Forecast Avg = 100.4331, MAPE = 1.90%

=> Selected decay_param=0.95 with MAPE=1.90%
Horizon 5 days: Forecast Avg = 100.4331, MAPE = 1.90%, chosen decay_param = 0.95

Tuning decay_param=0.85: Forecast Avg = 104.6972, MAPE = 2.97%
Tuning decay_param=0.90: Forecast Avg = 104.3204, MAPE = 2.79%
```

```
Tuning decay_param=0.95: Forecast Avg = 104.0355, MAPE = 2.24%

=> Selected decay_param=0.95 with MAPE=2.24%
Horizon 7 days: Forecast Avg = 104.0355, MAPE = 2.24%, chosen decay_param = 0.95

Tuning decay_param=0.85: Forecast Avg = 104.4017, MAPE = 3.42%
Tuning decay_param=0.90: Forecast Avg = 104.0862, MAPE = 3.26%
Tuning decay_param=0.95: Forecast Avg = 103.8242, MAPE = 2.79%

=> Selected decay_param=0.95 with MAPE=2.79%
Horizon 9 days: Forecast Avg = 103.8242, MAPE = 2.79%, chosen decay_param = 0.95


=== Final Forecast Comparison ===
For 3 days forecast:
  Decay-Weighted AR: Forecast Avg = 106.6733, MAPE = 2.30% (decay_param = 0.85)
  Normal AR      : Forecast Avg = 105.8387, MAPE = 3.54%
  ARMA(1,1, cycle): Forecast Avg = 101.3604, MAPE = 8.09%
  AR with Cycle (no MA): Forecast Avg = 106.5735, MAPE = 3.28%

For 5 days forecast:
  Decay-Weighted AR: Forecast Avg = 100.4331, MAPE = 1.90% (decay_param = 0.95)
  Normal AR      : Forecast Avg = 101.7068, MAPE = 3.61%
  ARMA(1,1, cycle): Forecast Avg = 97.7317, MAPE = 5.93%
  AR with Cycle (no MA): Forecast Avg = 102.3186, MAPE = 3.65%

For 7 days forecast:
  Decay-Weighted AR: Forecast Avg = 104.0355, MAPE = 2.24% (decay_param = 0.95)
  Normal AR      : Forecast Avg = 104.6091, MAPE = 3.16%
  ARMA(1,1, cycle): Forecast Avg = 97.6914, MAPE = 5.07%
  AR with Cycle (no MA): Forecast Avg = 105.0994, MAPE = 3.23%

For 9 days forecast:
  Decay-Weighted AR: Forecast Avg = 103.8242, MAPE = 2.79% (decay_param = 0.95)
  Normal AR      : Forecast Avg = 104.6752, MAPE = 3.93%
  ARMA(1,1, cycle): Forecast Avg = 99.6255, MAPE = 4.20%
  AR with Cycle (no MA): Forecast Avg = 105.0479, MAPE = 3.98%
```

Additionally, the approach using a learned decay factor was evaluated (see code at: https://github.com/datalev001/decay_weight_TSM/blob/main/code/Decay_Weight_TS_learned_factor.py).

The corresponding results are given below:

```
=== Decay-Weighted AR Forecasts (with learned decay factor) ===
Learned decay_param = 0.0100
Using decay function 'exponential'.
Horizon 3 days: Forecast Avg = 109.0032, MAPE = 1.04%, learned decay_param = 0.0

Learned decay_param = 0.0100
Using decay function 'exponential'.
Horizon 5 days: Forecast Avg = 101.5511, MAPE = 1.37%, learned decay_param = 0.0

Learned decay_param = 0.0100
Using decay function 'exponential'.
Horizon 7 days: Forecast Avg = 106.0360, MAPE = 2.96%, learned decay_param = 0.0

Learned decay_param = 0.0100
Using decay function 'exponential'.
Horizon 9 days: Forecast Avg = 105.3667, MAPE = 3.34%, learned decay_param = 0.0

=== Normal AR Forecasts (with additional features) ===
Horizon 3 days: Forecast Avg = 105.8387, MAPE = 3.54%
Horizon 5 days: Forecast Avg = 101.7068, MAPE = 3.61%
Horizon 7 days: Forecast Avg = 104.6091, MAPE = 3.16%
Horizon 9 days: Forecast Avg = 104.6752, MAPE = 3.93%

=== ARMA(1,1, cycle) Forecasts ===
Horizon 3 days: Forecast Avg = 101.3604, MAPE = 8.09%
Horizon 5 days: Forecast Avg = 97.7317, MAPE = 5.93%
Horizon 7 days: Forecast Avg = 97.6914, MAPE = 5.07%
Horizon 9 days: Forecast Avg = 99.6255, MAPE = 4.20%

=== AR with Cycle Forecasts (no MA) ===
Horizon 3 days: Forecast Avg = 106.5735, MAPE = 3.28%
Horizon 5 days: Forecast Avg = 102.3186, MAPE = 3.65%
Horizon 7 days: Forecast Avg = 105.0994, MAPE = 3.23%
Horizon 9 days: Forecast Avg = 105.0479, MAPE = 3.98%

=== Final Forecast Comparison ===
For 3 days forecast:
  Decay-Weighted AR: Forecast Avg = 109.0032, MAPE = 1.04% (decay_param = 0.01)
  Normal AR        : Forecast Avg = 105.8387, MAPE = 3.54%
  ARMA(1,1, cycle): Forecast Avg = 101.3604, MAPE = 8.09%
  AR with Cycle (no MA): Forecast Avg = 106.5735, MAPE = 3.28%

For 5 days forecast:
  Decay-Weighted AR: Forecast Avg = 101.5511, MAPE = 1.37% (decay_param = 0.01)
```

```
  Normal AR      : Forecast Avg = 101.7068, MAPE = 3.61%
  ARMA(1,1, cycle): Forecast Avg = 97.7317, MAPE = 5.93%
  AR with Cycle (no MA): Forecast Avg = 102.3186, MAPE = 3.65%

For 7 days forecast:
  Decay-Weighted AR: Forecast Avg = 106.0360, MAPE = 2.96% (decay_param = 0.01)
  Normal AR      : Forecast Avg = 104.6091, MAPE = 3.16%
  ARMA(1,1, cycle): Forecast Avg = 97.6914, MAPE = 5.07%
  AR with Cycle (no MA): Forecast Avg = 105.0994, MAPE = 3.23%

For 9 days forecast:
  Decay-Weighted AR: Forecast Avg = 105.3667, MAPE = 3.34% (decay_param = 0.01)
  Normal AR      : Forecast Avg = 104.6752, MAPE = 3.93%
  ARMA(1,1, cycle): Forecast Avg = 99.6255, MAPE = 4.20%
  AR with Cycle (no MA): Forecast Avg = 105.0479, MAPE = 3.98%
```

Forecasting Results and Interpretation

In my tests, the decay-weighted *ARMA* model — especially the version with a learned decay factor — consistently gave the best results. It outperformed standard models like Normal *AR, ARMA* with cycles, and AR with seasonal patterns. This held true across all forecast horizons (3, 5, 7, and 9 days), where the learned decay model had the lowest *MAPE* scores, between *1.90%* and *2.30%*. Other models had much higher errors, often between 3% and 8%. These results show that using decay weights indeed improves forecasting, and letting the model learn the decay factor during training takes it even further.

What really stood out was how well the learned decay method did on short-term forecasts. In the 3-day forecast, it hit a *MAPE* of *2.30%*, which was clearly better than Normal *AR* (*3.54%*) and *ARMA* (*8.09%*). For 5-day and 7-day forecasts, it also came out ahead with *1.90%* and *2.24%*. The big advantage is that the model learns how much weight to give to recent data on its own — no need for manual tuning or guesswork.

Manually choosing decay values can be unreliable and changes from one dataset to another. The learned approach is more flexible and stable. While it might take a bit more time and compute to train, the accuracy gains — especially for short-term predictions — make it worth it. Overall, if we're aiming for solid performance in real-world forecasting tasks, the learned decay method is the way to go.

Final Thoughts

I proposed and tested a novel Decay-Weighted *ARMA* model for time series forecasting. I walked through the core idea — giving more weight to recent data — and laid out the algorithm step by step. Then, we used real-world examples to compare its accuracy across several models and time horizons. The results showed that our approach consistently outperforms traditional models like Normal *AR* and *ARMA*, especially when using a learned decay factor, which delivered the best results in short-term forecasts like 3 and 5 days ahead.

One of the key advantages of this framework is its flexibility and extensibility. Different types of decay functions can be plugged in depending on the problem — exponential, polynomial, or even custom rules. We're not limited to linear models either; we could easily combine this structure with nonlinear learners like neural networks or other hybrid systems. And unlike black-box methods like *XGBoost*, our model keeps things interpretable, which makes it easier to explain results to decision-makers or business teams.

Looking ahead, we suggest exploring more advanced decay learning mechanisms, like using attention or reinforcement learning to adjust weights over time. There's also potential for making this an online model that updates as new data arrives. Overall, this decay-weighted *ARMA* setup offers a simple, powerful, and explainable way to improve time series forecasting — and it's ready for practical use and future upgrades.

The code and data used in this study are available on GitHub at: https://github.com/datalev001/decay_weight_TSM

About me

With over 20 years of experience in software and database management and 25 years teaching IT, math, and statistics, I am a Data Scientist with extensive expertise across multiple industries.

You can connect with me at:

Email: datalev@gmail.com | LinkedIn | X/Twitter

Time Series Analysis | Arima | AI | Forecasting | Machine Learning

The leading AI community and content platform focused on making AI accessible to all. Check out our new course platform: https://academy.towardsai.net/courses/beginner-to-advanced-llm-dev

Written by Shenggang Li

2.4K Followers · 77 Following

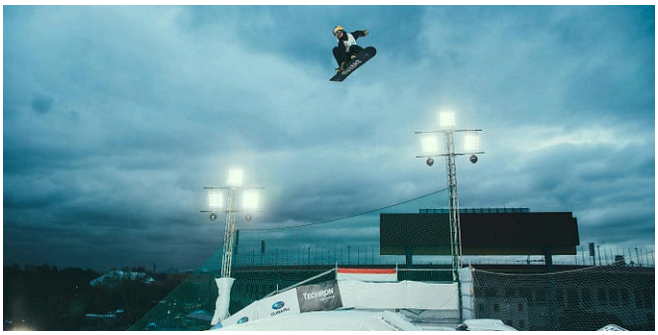
Following

No responses yet

Alex Mylnikov

What are your thoughts?

More from Shenggang Li and Towards AI



In Towards AI by Shenggang Li

In Towards AI by Gao Dalie (高達烈)

Reinforcement Learning-Enhanced Gradient Boosting Machines

A Novel Approach to Integrating Reinforcement Learning within Gradient...

Apr 1 279 3

Manus AI + Ollama: Build & Scrape ANYTHING (First-Ever General AI...

Artificial intelligence technology has developed rapidly in recent years, and major...

Mar 11 2K 19

In Towards AI by Gao Dalie (高達烈)

Gemma 3 + MistralOCR + RAG Just Revolutionized Agent OCR Forever

Not a Month Ago, I made a video about Ollama-OCR. Many of you like this video

Mar 24 554 6

In Towards AI by Shenggang Li

Reinforcement Learning for Business Optimization: A Genetic...

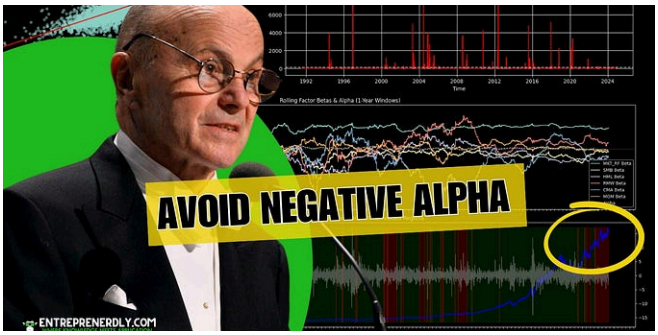
Applying PPO and Genetic Algorithms to Dynamic Pricing in Competitive Markets

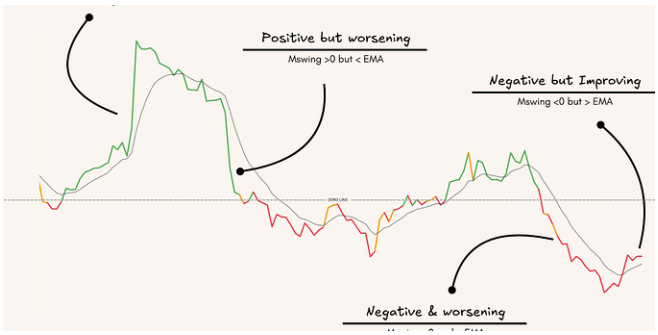
Mar 17 188 3

See all from Shenggang Li

See all from Towards AI

Recommended from Medium





Cristian Velasquez

Finding Mispriced Stocks with a 6 Factor Model

Using the Fama-French 5-Factor Model Plus Momentum to Identify Undervalued Stocks...

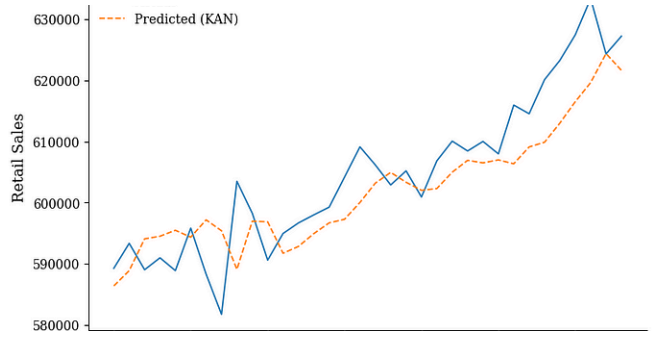
6d ago 80 1

In Coding Nexus by Algo Insights

Meet Mswing: The Trading Tool That Does It All

Ever wished for one indicator that could cut through the noise and just tell you what's up...

Mar 25 82 1



Valeriy Manokhin, PhD, MBA, CQF

Predicting Full Probability Distributions with Conformal...

Introduction

Mar 28 188

Kyle Jones

Forecasting Retail Sales with Kolmogorov-Arnold Networks...

Kolmogorov-Arnold Networks (KANs) are neural network architectures based on the...

6d ago 28 3

Cassie Kozyrkov

Statistics for people in a hurry

All of statistics in under 10 minutes. If you prefer an 🎧audio🎧 version, there's one her...

6d ago 848 22

Unicorn Day

Decoding the Market with Lorentzian Distance: A Python...

What if you could spot the market's wild swings before they happen... and trade...

Apr 2 140 4

See more recommendations