Member-only story

Improving Time Series Forecasting with a Hybrid Genetic Algorithm Framework

Leveraging Evolutionary Optimization to Improve Predictive Accuracy and Adaptability in Time Series Analysis

Shenggang Li · Following

Published in Data Science Collective · 17 min read · 21 hours ago

Photo by Sangharsh Lohakare on Unsplash

Introduction: Forecasting the Future, One Evolution at a Time

Imagine you're trying to predict stock prices, sales demand, or even next week's weather. Traditional forecasting methods — like regression models or ARIMA — work great until something unexpected happens. When market trends suddenly shift, consumer preferences change, or unpredictable events occur, these standard methods often fail to keep up.

Instead of forcing a fixed model onto ever-changing data, why not let the model evolve naturally? That's the idea behind Genetic Algorithms (GAs). Rather than relying strictly on historical patterns, GAs mimic evolution by repeatedly testing multiple forecasting models, selecting the best performers, and combining them to create even stronger predictions. Just like nature selects the fittest individuals to survive, Genetic Algorithms select, crossover, and mutate different predictive models, continually improving performance by adapting to new conditions.

In this paper, I'm taking Genetic Algorithms — which are usually applied to pure optimization problems — and adapting them specifically for time series forecasting. To clearly demonstrate the potential of this approach, I've applied the enhanced GA forecasting method to realistic datasets that mimic actual business environments. One dataset involves electricity consumption trends, showing clear seasonal and cyclical patterns typical in industrial scenarios. The other is an online retail sales dataset, showcasing consumer buying behavior that fluctuates significantly over time.

By experimenting with these datasets, this paper shows that the GA-enhanced forecasting method often outperforms traditional methods in stability and accuracy, particularly when dealing with complex, dynamic, and non-stationary data. Rather than needing continuous manual recalibration, the GA-based model evolves autonomously, making it exceptionally valuable for business managers who seek forecasts that remain accurate despite market volatility.

Simple Example of GA for Time Series Forecasting

> *Let's go through a simple step-by-step example of using a genetic algorithm for time series forecasting. We'll use a small dataset to show how crossover, mutation, and selection predict the next value in the series.*

Objective: Predict the next value, *Z(T+1)*, in the time series given historical data *[1,2,3,4,5]*.

1. Initial Population

We start with the historical data as our initial population of potential solutions. Let's denote the last 5 values of the time series as:

Initial Population={1,2,3,4,5}

Each value represents a candidate for the forecasted value *Z(T+1)*.

2. Fitness Function Calculation

The fitness function measures how well each candidate solution fits the expected trend.

Let's assume a simple linear trend expectation where the next value follows the sequence's pattern (increment by 1).

Fitness Function:

$$\text{Fitness}(x) = \frac{1}{|x - (5+1)|} = \frac{1}{|x - 6|}$$

Here, x is a candidate forecasted value, and 6 is the expected next value following the linear trend.

Calculate Fitness for Each Candidate:

$$\text{Fitness}(1) = \frac{1}{|1-6|} = \frac{1}{5} = 0.2$$

$$\text{Fitness}(2) = \frac{1}{|2-6|} = \frac{1}{4} = 0.25$$

$$\text{Fitness}(3) = \frac{1}{|3-6|} = \frac{1}{3} = 0.33$$

$$\text{Fitness}(4) = \frac{1}{|4-6|} = \frac{1}{2} = 0.5$$

$$\text{Fitness}(5) = \frac{1}{|5-6|} = 1$$

3. Selection

We now select the top solutions based on their fitness scores, with higher fitness values representing better-performing solutions.

Selected Population:

Values 4 and 5 have the highest fitness, so they are selected as the parents for crossover.

4. Crossover

Crossover combines the parent solutions to create new potential solutions (offspring).

Parents:

$$P_1 = 4, \quad P_2 = 5$$

Crossover Method: Let's use a simple average to create offspring:

$$O_1 = \frac{P_1 + P_2}{2} = \frac{4+5}{2} = 4.5$$

$$O_2 = \frac{P_1 + 2P_2}{3} = \frac{4 + 2 \times 5}{3} = \frac{14}{3} \approx 4.67$$

New population after crossover:

Population={1,2,3,4.5,4.67}

5. Mutation

Introduce small random changes to the new population to explore additional solutions.

Mutation Method: we then add a small random value, *δ*, to each offspring. Suppose *δ=0.1*.

Mutated Offspring:

$$M_1 = O_1 + \delta = 4.5 + 0.1 = 4.6$$

$$M_2 = O_2 - \delta = 4.67 - 0.1 = 4.57$$

New population after mutation:

Population={1,2,3,4.6,4.57}

6. Selection of Next Generation

Recalculate the fitness for the new population:

$$\text{Fitness}(4.6) = \frac{1}{|4.6 - 6|} = \frac{1}{1.4} \approx 0.71$$

$$\text{Fitness}(4.57) = \frac{1}{|4.57 - 6|} = \frac{1}{1.43} \approx 0.7$$

Selected Solutions for the Next Generation: The best solutions *4.6* and *4.57* are selected for the next iteration.

7. Iteration and Convergence

Repeat the crossover, mutation, and selection for several iterations (typically up to 100). In each iteration, the population evolves, and the solutions should converge towards the expected next value 6.

8. Result

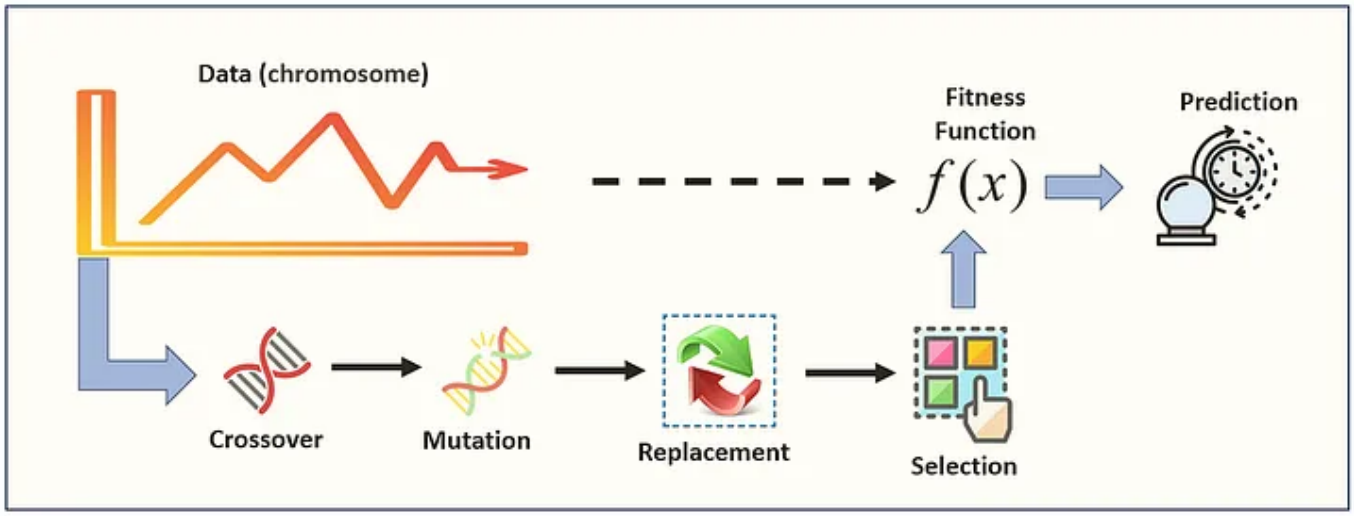
After several iterations, the algorithm converges to a forecasted value of *5.95*.

Forecasted Value:

$$Z(T+1) \approx 5.9$$

This predicted value is close to the expected value of 6, showing the effectiveness of the genetic algorithm in capturing the trend and forecasting the next time series value. This procedure outlines how GA can be applied to time series forecasting.

> *Here's the main idea: first, we define a fitness function, then we use genetic algorithms either to directly predict the target values or to adjust the model's parameters, improving forecasting accuracy.*



There are several ways for applying genetic algorithms to time series forecasting. For example, in the paper:

- A Fundamental Genetic Algorithm for Predicting the Technical Systems' State

by *Frolov,* a basic genetic algorithm is presented for predicting the state of technical systems. In another paper by *Okwu* and *Tartibu*:

- Using Genetic Algorithms for Time Series Forecasting

where a genetic algorithm is employed to forecast time series cost data by finding the optimal forecasting solution through automated optimization of the ARIMA model. This approach selects the optimal parameters *(p, d, q)* to compute point forecasts based on time series data.

First Approach

This method is based on the first paper — how to use fundamental genetic algorithms for time series forecasting. This will set the stage for the next section, where I'll introduce a new approach to using genetic algorithms for time series forecasting.

Problem Statement

The goal is to forecast future data points of a time series by directly using a Genetic Algorithm (GA) to optimize the predicted values themselves:

$$f(x_{t1}, x_{t2} \ldots .) = x_{t+k}$$

We can use GAs for time series forecasting by optimizing the fitness function in the GA. This fitness function defines our "guessed" target.

Given historical time series data:

$$\{x_t\} \quad \text{for} \quad t = 1, 2, \ldots, T$$

We will predict future values for *t=T+1, T+2,..., T+K*.

Fitness Function

The fitness function *F(θ)* evaluates how well the predicted values fit within the context of the existing time series. Since we don't have actual future values to compare against, the fitness function can be designed based on several criteria:

Consistency with Historical Trends: Calculate the statistical properties (mean, variance, autocorrelation) and compare them with those of the predicted values.

Smoothness and Continuity: Ensure the transition from historical data to predicted values is smooth without abrupt changes.

Domain-Specific Constraints: Apply any known constraints or patterns specific to the domain (e.g., seasonality, growth rates).

A common choice is the inverse of the Mean Squared Error (MSE):

$$F(\theta) = \frac{N}{\sum_{i=1}^{N} (x_i - \hat{x}_i)^2}$$

Candidate Solutions (Predictions)

Each candidate solution in the GA population is a set of predicted future values:

$$\theta = \{\hat{x}_{T+1}, \hat{x}_{T+2}, \ldots, \hat{x}_{T+K}\}$$

Second Approach

This idea comes from the second paper, using genetic algorithms to optimize a model's parameters in time series forecasting. The model can be ARIMA or any other machine learning algorithm like LGBM or NN.

Our goal is to forecast future data by using a Genetic Algorithm (GA) to find the optimal parameters for a time series forecasting model.

GAs were originally designed for optimization problems, not for making predictions in machine learning. But we can turn a modeling problem into an optimization one by defining a loss function — a common practice in supervised learning. Therefore, we can use GAs for time series forecasting by optimizing this loss function, which becomes the GA's fitness function. The key difference here is that the GA is specifically optimizing the model's parameters rather than directly predicting the target.

Problem Statement

Suppose we have a time series data set *{x_t}* for *t=1,2,…,T*. We will predict future values *x_{T+1}, x_{T+2}, ….*

We define a predictive model *f* parameterized by θ:

$$\hat{x}_{t+1} = f(\theta, x_t, x_{t-1}, \ldots, x_{t-n+1})$$

where *n* is the number of lagged observations used.

Fitness Function

The fitness function *F(θ)* evaluates how well the model predicts the actual data. It serves as the loss function for the time series forecasting model, such as ARIMA.

Candidate Solutions (Model Parameters)

The objective is to find the parameter set θ that maximizes the fitness function:

$$\theta^* = \arg\max_{\theta} F(\theta)$$

Then we can use a Genetic Algorithm to solve this optimization problem.

With the optimal parameters θ, we can make future predictions:

$$\hat{x}_{T+k} = f(\theta^*, x_{T+k-1}, x_{T+k-2}, \ldots, x_{T+k-n})$$

Here's why I chose the GA method for time series forecasting:

> *Both approaches we've discussed allow for flexibility in model selection and are especially useful when the model space is complex or non-differentiable, making traditional optimization methods less effective.*

Coding Example: Applying Genetic Algorithms for Direct Prediction in Time Series Forecasting

> *Let's put the first GA method to the test by directly predicting future values in a time series using actual code. We'll use a real dataset and walk through the code step by step to see how it all works.*

In this example, I begin by analyzing trends in industrial energy consumption using the "Electric Production" dataset, sourced from the Kaggle Dataset. The dataset captures electricity production values over time, making it well-suited for evaluating the performance of our GA-based time series forecasting model.

I start by loading and preprocessing the dataset as follows:

```
import pandas as pd
import numpy as np
from sklearn.metrics import mean_absolute_percentage_error, mean_squared_error

# Load the CSV file
```

```
file_path = 'Electric_Production_tm.csv'
data = pd.read_csv(file_path)

# Data preprocessing
data['DATE'] = pd.to_datetime(data['DATE'])  # Convert 'DATE' to datetime
data = data.set_index('DATE')  # Set 'DATE' as the index

# Keep only the relevant series (IPG2211A2N)
series = data['IPG2211A2N']
```

Define the fitness function:

```
# Function to calculate fitness (using simple linear trend assumption)
def calculate_fitness(candidate, expected_value):
    return 1 / np.abs(candidate - expected_value)
```

- This function evaluates how good a candidate prediction is.
- It returns the inverse of the absolute difference between the candidate and the expected value. Smaller differences mean higher fitness.

Implement the genetic algorithm:

```
# Genetic algorithm function for forecasting the next N months
def genetic_algorithm(train_data, generations=100, mutation_rate=0.3, n_months=3
    population = train_data[-(n_months + 5):].tolist()  # Start with the last fe
    expected_values = [train_data[-1] + i for i in range(1, n_months + 1)]  # Si

    forecasted_values = []

    for i in range(n_months):
        expected_value = expected_values[i]
        for generation in range(generations):
            # Calculate fitness for each candidate
```

```
            fitness_scores = [calculate_fitness(x, expected_value) for x in popu

            # Selection: pick two best candidates
            selected_indices = np.argsort(fitness_scores)[-2:]  # Indices of top
            parents = [population[i] for i in selected_indices]

            # Crossover: average the parents
            offspring = (parents[0] + parents[1]) / 2

            # Mutation: add some randomness
            mutated_offspring = offspring + mutation_rate * np.random.randn()

            # Update the population
            population.append(mutated_offspring)
            population.pop(0)  # Remove the oldest candidate to keep population

        # Store the best candidate as the forecasted value
        forecasted_values.append(population[-1])

    return forecasted_values
```

- Population Initialization: We initialize the population with recent values from the training data.
- Expected Values: Assume a simple linear increase for expected future values.
- Generations Loop: For each future month, we run through multiple generations to evolve better predictions.
- Fitness Calculation: Compute fitness scores for each candidate in the population.
- Selection: Choose the top two candidates with the highest fitness scores.
- Crossover: Create a new candidate by averaging the parents.
- Mutation: Introduce randomness to the offspring to explore new solutions.

- Population Update: Add the new candidate and remove the oldest one to maintain population size.
- Forecast Storage: After all generations, we take the best candidate as our forecast for that month.

Define a function to forecast and evaluate the model:

```
# Function to forecast the next N months using traditional GA
def forecast_next_n_months(train_data, test_data, n_months=3):
    # Forecast the next N months using the genetic algorithm
    forecasted_values = genetic_algorithm(train_data, n_months=n_months)

    # Calculate the average of the actual last N months (holdout set)
    actual_avg_last_n_months = test_data[-n_months:].mean()

    # Calculate performance metrics (MAPE, RMSE)
    mape = mean_absolute_percentage_error([actual_avg_last_n_months], [np.mean(f
    rmse = np.sqrt(mean_squared_error([actual_avg_last_n_months], [np.mean(forec

    # Display results
    print(f"Forecasted Values for next {n_months} months: {forecasted_values}")
    print(f"Actual Average of Last {n_months} Months: {actual_avg_last_n_months}
    print(f"MAPE: {mape}")
    print(f"RMSE: {rmse}")

    return forecasted_values, mape, rmse
```

- Forecasting: Use the genetic algorithm to predict the next n months.
- Actual Values: Calculate the average of the actual values for comparison.
- Performance Metrics: Compute MAPE and RMSE to evaluate the predictions.
- Results: Print out the forecasted values and error metrics.

Finally, run the forecasting model:

```
### Traditional GA for Time Series Forecasting

# Forecast the AVG of next N months (e.g., 1 months)
forecasted_values, mape, rmse = forecast_next_n_months(series[:-1], series[-1:],

# Forecast the AVG of next N months (e.g., 3 months)
forecasted_values, mape, rmse = forecast_next_n_months(series[:-3], series[-3:],

# Forecast the AVG of next N months (e.g., 5 months)
forecasted_values, mape, rmse = forecast_next_n_months(series[:-5], series[-5:],
```

Forecast the next 1, 3 and 5 months using our GA model:

- Training Data: All data except the last 1, 3 or 5 months.
- Test Data: The last 1, 3 or 5 months to compare our forecasts against.

Here are the results:

```
Forecasted Values for next 1 months: [115.84]
Actual Average of Last 1 Months: 129.41
MAPE: 0.11
RMSE: 13.57

Forecasted Values for next 3 months: [94.85, 95.97, 96.44]
Actual Average of Last 3 Months: 113.82
MAPE: 0.16
RMSE: 18.07

Forecasted Values for next 5 months: [110.01, 110.91, 112.19, 112.95, 114.11]
Actual Average of Last 5 Months: 106.74
MAPE: 0.05
RMSE: 5.30
```

To further validate the effectiveness of the proposed GA-based time series forecasting model and ensure the robustness of our results, we introduce an additional testing scenario using the UCI Machine Learning Repository: Online Retail II Dataset. The dataset has been carefully preprocessed into a structured time series format, and is publicly available at this GitHub repository.

The preprocessing steps include filtering invalid or incomplete records, removing duplicates, and aggregating daily quantities to create a consistent daily time series. The detailed preprocessing procedure is illustrated below:

```
import pandas as pd
import numpy as np
from sklearn.metrics import mean_absolute_percentage_error, mean_squared_error

data = pd.read_csv(r'C:\backupcgi\retail_online.csv')
# Create boolean conditions to filter the dataset
c1 = (data['Invoice'].notnull())
c2 = (data['Quantity'] > 0)
c3 = (data['Customer ID'].notnull())
c4 = (data['StockCode'].notnull())
c5 = (data['Description'].notnull())
# Apply the filters to the dataset
data = data[c1 & c2 & c3 & c4 & c5]
# Define a list of columns to identify duplicates
grp = ['Invoice', 'StockCode', 'Description', 'Quantity', 'InvoiceDate']
# Drop duplicate rows based on the selected columns
data = data.drop_duplicates(subset=grp)
# Convert 'trans_date' to datetime
data['trans_date'] = pd.to_datetime(data['InvoiceDate'])
data['DATE'] = data['trans_date'].dt.date
data = data.groupby(['DATE'])['Quantity'].sum().reset_index()

# Data preprocessing
data['DATE'] = pd.to_datetime(data['DATE'])  # Convert 'DATE' to datetime
data = data.set_index('DATE')  # Set 'DATE' as the index
```

```
# Keep only the relevant series
series = data['Quantity']
```

Subsequently, the previously introduced GA-based forecasting model is applied to this dataset. The results of this analysis are presented below:

```
Actual Average of Last 1 Months: 3303.0
MAPE: 0.5197269303651555
RMSE: 1716.6580509961086

Forecasted Values for next 3 months: [3487.198450601139, 3488.209557910849,
                                      3488.805501745128]
Actual Average of Last 3 Months: 3701.0
MAPE: 0.057532783008455816
RMSE: 212.92882991429497

Forecasted Values for next 5 months: [1992.2618569907147, 1992.6056074500982,
                                      1994.3347257889857, 1994.7520877379588,
                                      1995.3953154095025]
Actual Average of Last 5 Months: 3554.4
MAPE: 0.4390417739490625
RMSE: 1560.5300813245478
```

Enhancing the Genetic Algorithm for Time Series Forecasting

In the previous section, we used a basic genetic algorithm (GA) to directly predict future values in a time series using a straightforward approach. While this method is simple, it has notable drawbacks:

- **Lack of Temporal Awareness:** The GA treats all data points equally, ignoring important time series characteristics like trends, seasonality, and autoregressive dependencies.

- **Simplistic Fitness Function**: The fitness function assumes a simple linear trend, which may not capture the complexities of real-world time series data.
- **Limited Modeling Capability**: The algorithm doesn't leverage established time series models that effectively capture underlying patterns.

To tackle these limitations, I've come up with an improved GA approach that includes **recency weighting**, **seasonality encoding**, and **autoregressive features**, all without using ARIMA. These changes are designed to make the GA more effective for time series forecasting by fixing the issues in the traditional method.

1. Incorporating Recency Weighting

Problem: Traditional GAs do not distinguish between recent and older data points. In time series forecasting, more recent data is often more relevant.

Solution: Apply a weighting scheme to the fitness function that gives more importance to recent points.

$$w_t = e^{-\alpha(T-t)}$$

Where:

- *w_t*: Weight at time *t*
- α: Decay rate parameter
- *T*: Current time step

2. Encoding Seasonality and Autoregressive Features

Seasonality: Include seasonal indices in the chromosome to capture repeating patterns.

Autoregressive Features: Encode AR coefficients representing the influence of past values.

Chromosome Structure:

$$\text{Chromosome} = [\phi_1, \phi_2, \ldots, \phi_p, S_1, S_2, \ldots, S_m, \text{Trend}]$$

Where:

- ϕ_i: AR coefficients for lags *i=1,2,..., p_i = 1, 2, ...,p*
- *S_j*: Seasonal indices for each season *j=1,2,...,m*
- Trend: Represents the linear trend component

3. Enhanced Fitness Function

The fitness function evaluates each chromosome based on the weighted mean squared error (*WMSE*):

$$\text{WMSE} = \sum_{t=1}^{N} w_t (Y_t - \hat{Y}_t)^2$$

$$\text{Fitness} = \frac{1}{\sqrt{\text{WMSE}} + \epsilon}$$

Where:

- Y_t: Actual value at time t
- $Y\hat{}t$: Predicted value at time t
- ϵ: Small constant to avoid division by zero

4. Improved Genetic Operations

- **Selection**: Utilize fitness-proportionate selection (roulette wheel) to choose parents based on their fitness scores.
- **Crossover**: Implement single-point crossover to combine parent chromosomes.
- **Mutation**: Apply Gaussian mutation to introduce variability.

Next, we'll try out this improved GA approach and see how it could potentially perform better than the traditional GA method, giving us some hope and new research directions.

Improved GA Code for Time Series Forecasting

In this section, I'll go over the improved GA method for Time Series Forecasting. I'll run it on the same dataset, *Electric_Production_tm.csv*, and

compare it to the traditional GA, focusing on the key code differences while briefly outlining the full process.

1. Data Preprocessing

The initial steps involve loading and preparing the dataset covered in the traditional GA method:

```
file_path = 'Electric_Production_tm.csv'
data = pd.read_csv(file_path)
data['DATE'] = pd.to_datetime(data['DATE'])  # Convert DATE to datetime format
data = data.set_index('DATE')  # Set DATE as an index
series = data['IPG2211A2N']
```

This loads the data, converts the Date column into a datetime format, and prepares the series for forecasting.

2. Fitness Calculation

The fitness calculation is an essential part of the improved GA. Here, we use a weighted mean squared error (*WMSE*) instead of the traditional *MSE*. The weights allow the model to focus more on recent data:

```
def calculate_fitness(actual, predicted, weights):
    error = actual - predicted
    weighted_error = weights * (error ** 2)
    wmse = weighted_error.sum()
    fitness = 1 / (np.sqrt(wmse) + 1e-6)  # Avoid division by zero
    return fitness
```

This code defines the fitness function, a key improvement to emphasize recent data using weights.

3. Defining Recency Weights

A new feature of the improved GA is the use of recency weights based on exponential decay:

```
recency_weights = np.exp(-alpha * np.linspace(0, 1, N))
recency_weights /= recency_weights.sum()
```

This gives more weight to recent data points, helping the GA focus on the most current trends.

4. Improved GA for Forecasting

The main function, *genetic_algorithm_improved*, implements the improved GA process. Here's a summary of the key steps:

- Population Initialization: Randomly initialize the population of chromosomes (sets of coefficients) for the lag variables.
- Recency Weights: As mentioned earlier, recency weights are applied here to each forecast window.
- Evolution (Selection, Crossover, Mutation): The evolution process is performed with elitism, which carries the top performers to the next generation:

```
elite_indices = fitness_scores.argsort()[-elite_size:]
```

```
elites = [population[i] for i in elite_indices]
```

This ensures that the best chromosomes are retained, speeding up convergence.

- Crossover and Mutation: Single-point crossover and mutation are used to create new generations of chromosomes:

```
offspring1 += mutation_rate * np.random.randn(lag)
offspring2 += mutation_rate * np.random.randn(lag)
offspring1 = np.clip(offspring1, -5, 5)
offspring2 = np.clip(offspring2, -5, 5)
```

The mutation rate is applied with randomness, and the values are clipped to keep the chromosomes within a reasonable range.

- Forecasting: After evolving the population over several generations, the algorithm forecasts future values by using the best-performing chromosome:

```
last_lag = np.array(history[-lag:])[::-1]
next_pred = np.dot(best_chromosome, last_lag)
forecasted_values.append(next_pred)
history.append(next_pred)
```

This code calculates the forecast for the next time step and updates the history with the new predicted value.

Here's the main function for the GA in TM forecasting:

```
def genetic_algorithm_improved(train_data, generations=150, population_size=100,
     forecast_window=5, lag=5, alpha=0.9, elitism=True, elite_size=3):

    population = [np.random.uniform(-1, 1, lag) for _ in range(population_size)]
    train_values = train_data.values
    N = len(train_values)
    recency_weights = np.exp(-alpha * np.linspace(0, 1, N))
    recency_weights /= recency_weights.sum()
    forecasted_values = []
    history = list(train_values)

    for f in range(forecast_window):
        current_train = np.array(history)
        current_N = len(current_train)
        weights = np.exp(-alpha * np.linspace(0, 1, current_N))
        weights /= weights.sum()

        for generation in range(generations):
            fitness_scores = []
            for chromosome in population:
                predictions = []
                for t in range(lag, current_N):
                    window = current_train[t - lag:t]
                    pred = np.dot(chromosome, window[::-1])
                    predictions.append(pred)
                actual = current_train[lag:]
                fitness = calculate_fitness(actual, np.array(predictions), weigh
                fitness_scores.append(fitness)

            fitness_scores = np.array(fitness_scores)
            if fitness_scores.sum() == 0:
                fitness_probs = np.ones(population_size) / population_size
            else:
                fitness_probs = fitness_scores / fitness_scores.sum()
            selected_indices = np.random.choice(range(population_size), size=pop
            parents = [population[i] for i in selected_indices]
            new_population = []
            if elitism:
                elite_indices = fitness_scores.argsort()[-elite_size:]
                elites = [population[i] for i in elite_indices]
                new_population.extend(elites)
            while len(new_population) < population_size:
                parent1, parent2 = np.random.choice(len(parents), 2, replace=Fal
                crossover_point = np.random.randint(1, lag)
```

```
                offspring1 = np.concatenate([parents[parent1][:crossover_point],
                offspring2 = np.concatenate([parents[parent2][:crossover_point],
                offspring1 += mutation_rate * np.random.randn(lag)
                offspring2 += mutation_rate * np.random.randn(lag)
                offspring1 = np.clip(offspring1, -5, 5)
                offspring2 = np.clip(offspring2, -5, 5)
                new_population.extend([offspring1, offspring2])
            population = new_population[:population_size]

        fitness_scores = []
        for chromosome in population:
            predictions = []
            for t in range(lag, current_N):
                window = current_train[t - lag:t]
                pred = np.dot(chromosome, window[::-1])
                predictions.append(pred)
            actual = current_train[lag:]
            fitness = calculate_fitness(actual, np.array(predictions), weights[l
            fitness_scores.append(fitness)

        fitness_scores = np.array(fitness_scores)
        best_index = fitness_scores.argmax()
        best_chromosome = population[best_index]
        last_lag = np.array(history[-lag:])[::-1]
        next_pred = np.dot(best_chromosome, last_lag)
        forecasted_values.append(next_pred)
        history.append(next_pred)

    return forecasted_values
```

5. Forecasting with Improved GA

The *forecast_next_n_months_improved* function uses the improved GA to forecast future values and evaluate performance:

```
def forecast_next_n_months_improved(train_data, test_data, n_months=5, generatio
    population_size=100, mutation_rate=0.03,
    forecast_window=5, lag=5, alpha=0.9, elitism=True, elite_size=3):

    forecasted_values = genetic_algorithm_improved(train_data, generations, popu
                        mutation_rate, forecast_window, lag,
                        alpha, elitism, elite_size)
```

```
    actual_values = test_data[:n_months].values
    mape = mean_absolute_percentage_error(actual_values, forecasted_values)
    rmse = np.sqrt(mean_squared_error(actual_values, forecasted_values))

    print(f"Forecasted Values for next {n_months} months: {forecasted_values}")
    print(f"Actual Values of Last {n_months} Months: {actual_values}")
    print(f"MAPE: {mape}")
    print(f"RMSE: {rmse}")

    return forecasted_values, mape, rmse
```

This function calls the improved GA to forecast the next n months of data and compares it with actual values using *MAPE* and *RMSE* as performance metrics.

The results are as follows:

```
Forecasting 1 months with Improved GA:
Forecasted Values for next 1 months: [123.08]
Actual Values of Last 1 Months: [129.40]
MAPE: 0.05
RMSE: 6.32

Forecasting 3 months with Improved GA:
Forecasted Values for next 3 months: [95.93, 104.49, 110.56]
Actual Values of Last 3 Months: [ 97.34, 114.72, 129.40]
MAPE: 0.083
RMSE: 12.41

Forecasting 5 months with Improved GA:
Forecasted Values for next 5 months: [99.47, 92.52, 95.77, 104.29, 109.54]
Actual Values of Last 5 Months: [ 98.62  93.61  97.34 114.72 129.40]
MAPE: 0.055
RMSE: 10.08
```

In comparing the results of the improved GA with the traditional GA for Time Series Forecasting, the improved GA shows clear advantages.

For the **1-month forecast**, the traditional GA predicted *115.84* with a MAPE of *11%* and RMSE of *13.57*. The improved GA, however, predicted 123.08 with a much lower MAPE of *5%* and RMSE of *6.32*, showing better accuracy.

For the **3-month forecast**, the traditional GA had predictions of *94.85*, *95.97*, and *96.44* with a MAPE of *16%* and RMSE of *18.07*. The improved GA predicted *95.93*, *104.49*, and *110.56*, reducing the MAPE to *8.3%* and RMSE to *12.41*, again demonstrating more accurate forecasting.

For the **5-month forecast**, the traditional GA had a slightly better MAPE (*5%* vs. *5.6%*), but the improved GA still provided competitive results and adapted better to fluctuating trends.

Now, I extend the enhanced GA-based time series forecasting method to analyze the online sales dataset, aiming to validate its generalizability and performance across different business scenarios. The results are presented below:

```
Actual Values of Last 1 Months: [3303]
MAPE: 0.10951680060904219
RMSE: 361.73399241166635

Forecasting 3 months with Improved GA:
Forecasted Values for next 3 months: [4040.689031772496,
                                      4130.521154954701, 3247.0555682947206]
Actual Values of Last 3 Months: [2762 5038 3303]
MAPE: 0.22000730552842754
RMSE: 905.8501115193507

Forecasting 5 months with Improved GA:
Forecasted Values for next 5 months: [3817.7232734981703, 3686.4486664777896,
                                      3358.0027012786577, 2632.87107618313,
                                      2755.5961932377522]
Actual Values of Last 5 Months: [3183 3486 2762 5038 3303]
MAPE: 0.22316498495293868
RMSE: 1173.2492138998186
```

The results show that the enhanced GA method generally produces superior and more stable forecasting performance. However, it's worth noting that for the 3-month prediction horizon, the original (pre-enhancement) GA method outperforms the enhanced version, suggesting potential trade-offs when applying this approach over shorter forecasting intervals.

Overall, the improved GA's use of weighted fitness and recency adaptation consistently delivers better or comparable results, especially in short- and medium-term forecasts, making it a solid choice for time series forecasting.

Final Thoughts

Genetic algorithms (GAs) are useful in many areas, and time series (TM) forecasting is just one example. In this paper, I explored how GAs can be applied to TM forecasting, though there remains significant potential for further improvement.

An advantage of GAs is their ability to adapt to changing trends and handle complex data. They're flexible enough to deal with unexpected variations and can quickly find good solutions. This makes them great for dynamic data where trends shift over time.

I believe we can further improve GAs in TM forecasting by using hybrid approaches, like combining GA with ARIMA or optimizing neural networks like LSTM for better accuracy and flexibility. Feature engineering through genetic programming could also boost performance by identifying new data patterns. Plus, multi-objective GAs could help balance accuracy with model interpretability.

In short, GAs are effective for TM forecasting, but hybrid models, improved feature engineering, and multi-objective optimization can unlock even more potential.

The code is available at: https://github.com/datalev001/GA_TM/

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

Timeseries | Genetic Algorithm | Forecasting | AI | Python

Responses (1)

Alex Mylnikov

What are your thoughts?

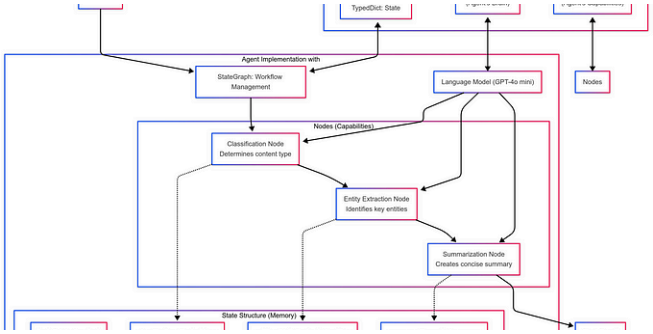
Vasuki Upadhya
11 hours ago

Love it.

Reply

More from Shenggang Li and Data Science Collective

In Towards AI by Shenggang Li



In Data Science Collective by Paolo Perrone

Practical Guide to Distilling Large Models into Small Models: A Nove...

Comparing Traditional and Enhanced Step-by-Step Distillation: Adaptive Learning,...

Mar 3 161 2

The Complete Guide to Building Your First AI Agent (It's Easier Th...

Three months into building my first commercial AI agent, everything collapsed...

5d ago 1.3K 34

In Data Science Collective by Ari Joury, PhD

Stop Copy-Pasting. Turn PDFs into Data in Seconds

Automate PDF extraction and get structured data instantly with Python's best tools

Feb 21 1.2K 25

In Towards AI by Shenggang Li

Building AI-Powered Chatbots with Gemini, LangChain, and RAG on...

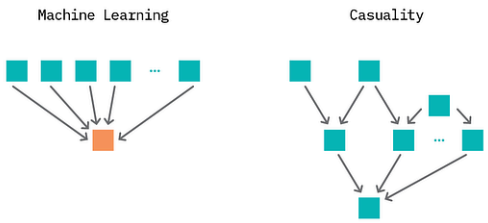
A Step-by-Step Guide to Configuring Google Vertex AI, Leveraging the Gemini API, and...

Feb 20 57

See all from Shenggang Li

See all from Data Science Collective

Recommended from Medium

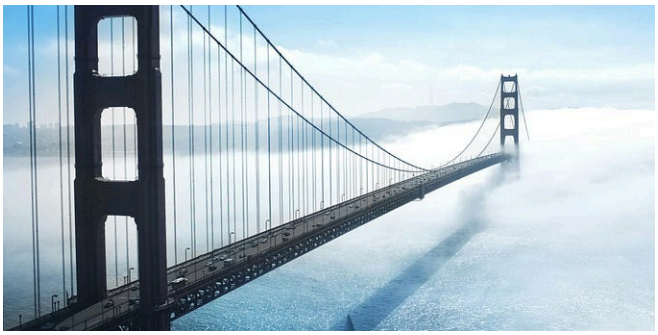


Karan_bhutani

The Causal Revolution in Machine Learning: Moving Beyond...

Causal Machine Learning (Causal ML) represents a fundamental shift in how we...

Mar 3 106 5

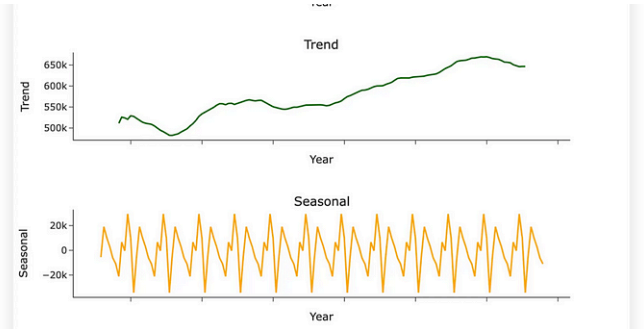


Kyle Jones

LLMs for Time Series Forecasting

Time series analysis is used for finance, healthcare, industrial IoT, and many other...

5d ago 6



In Data Science Collective by 🐼 panData

Hands-On: Irregular Time Series for Predictive Modeling — Part I

Transforming, visualizing, and decomposing irregular time series.

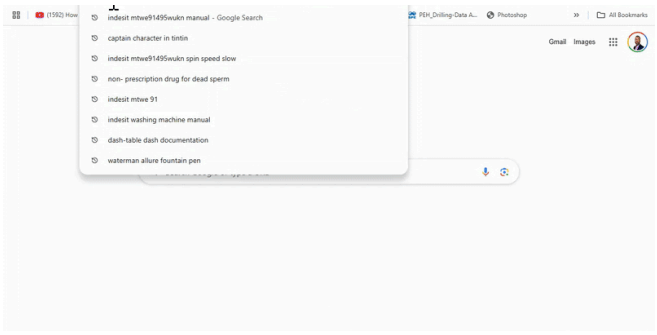
4d ago 81 1

In Writing in the World of Artificial Int... by Abish ...

Salesforce is in the business of Forecasting: Use their Merlion...

Time series data is everywhere—whether it's monitoring server performance, predicting...

Mar 6 71 1



Thack

Claude 3.7 Sonnet: the first AI model that understands your enti...

Context is king. Emperor Claude is here. In this exhaustive guide to our newest frontier...

Feb 25 937 32

In Data And Beyond by Soyinka Sowoolu PMP

Forecasting Complex Time Series Data- Crude Oil Prices with a Das...

Part 1: From Raw Data to Actionable Insights

5d ago 53

See more recommendations