Implementing Gap-Filling Algorithms in Chaotic Time Series Analysis

Sirui Ray Li

May 9, 2024

1 Abstract

This paper introduces a software tool designed to fill gaps in chaotic time series data, a common problem in data analysis that can significantly affect results. Using techniques based on Takens' Theorem [1] for reconstructing dynamics from incomplete time series, our software estimates missing data points to maintain the continuity and integrity of the data. The software was tested with synthetic data generated using the Lorenz system, demonstrating its ability to effectively predict and fill missing data across various scenarios. This capability is crucial for improving the reliability of analyses conducted with chaotic time series, making the tool valuable for researchers and analysts dealing with incomplete datasets.

2 Introduction

2.1 Objective

This paper introduces a software tool specifically designed to address the challenge of filling gaps in chaotic time series data. Utilizing algorithms based on Takens' Theorem for dynamical reconstruction, our software estimates and interpolates missing data points to restore the continuity of chaotic systems. The objective is to enhance the reliability of analyses and predictions derived from incomplete datasets commonly encountered in scientific research.

2.2 Structure of the Paper

The structure of the paper is organized as follows:

- **Methodology:** We outline the core algorithms and the structure of the software, detailing the implementation of the various modules such as `Functional J Strategies`, `Vector Field F Strategies`, and `Min Distance Strategies` .

- **Software Demonstration and Results:** Demonstrations using synthetic data generated from the Lorenz system illustrate the tool's capabilities in practical scenarios, supported by code snippets and performance metrics.
- **Discussion:** The effectiveness of the software is discussed, along with challenges faced during development and potential improvements.
- **Conclusion:** We summarize the contributions of the tool to chaotic time series analysis and suggest future enhancements.

This introduction sets the stage for a detailed presentation of our software's capabilities and its application in filling data gaps within chaotic time series, thereby ensuring more robust data analysis.

3 Methodology

3.1 Overview of the Original Algorithm

This implementation is based on Takens' Theorem, which allows the reconstruction of a dynamical system from a time series. The software leverages this theorem to estimate missing data points effectively, ensuring that the dynamics of the chaotic system remain intact even in the presence of data gaps.

3.2 Software Architecture

The software is structured into several modules, each responsible for a different aspect of the gap-filling process:

- **FunctionalJStrategies:** Manages strategies related to the functional J, a mathematical tool used to measure the fitness of filled gaps against the expected dynamics.
- **VectorFieldFStrategies:** Handles the estimation of the vector field F, which describes the dynamics of the system.
- **MinDistanceStrategies:** Implements methods to calculate the minimum distances within the time series, aiding in identifying the most probable trajectories for gap filling.

3.3 Implementation Details

The software's core functionalities are divided among several Python modules, each encapsulating a distinct strategy. Here we describe the roles and functionalities:

3.3.1 Data Generator

The data_generator module uses the Lorenz system parameters to create synthetic datasets for testing. This module is crucial for generating realistic scenarios where data might be missing, allowing for thorough testing and optimization of the gap-filling algorithms.

```python
def demo_data_generator():
    x, t = iterate_solver(Runge_Kutta, Lorenz63, [-1., 1., 18.4],
    0, 0.01, 50.)
    x_gapped = copy.deepcopy(x)
    x_gapped[2000:3000] = np.nan
    ts = x_gapped[:, 0]
    return ts, t
```

3.3.2 Functional J Strategies

This module implements the discrete J1 strategy, which evaluates how well the proposed gap filling aligns with the known dynamics of the system. The strategy is crucial for ensuring that the filled gaps are not only statistically plausible but also dynamically consistent.

```python
class DiscreteJ1Strategy(FunctionalJStrategy):
    def J(self, gap_filling_vectors: np.ndarray,
    reconstructed_vectors: np.ndarray, t: np.ndarray,
          F: Callable[[int, np.ndarray, np.ndarray, np.ndarray], np
    .ndarray]) -> int:
        w = gap_filling_vectors
        l = len(w)
        sum_squares = 0
        for j in range(l):
            delta_w = (w[j] - w[j - 1]) / t[j + 1] - t[j]
            F_value = F(j=j, reconstructed_vectors=
    reconstructed_vectors, gap_filling_vectors=gap_filling_vectors,
     t=t)
            sum_squares += linalg.norm(delta_w - F_value) ** 2
        return sum_squares
```

3.3.3 Vector Field F Strategies

The VectorFieldFDiscreteMidPointStrategy is implemented to approximate the vector field F at midpoints between data points, providing a crucial estimation of the system's dynamics necessary for the reconstruction of missing data.

```python
class VectorFieldFDiscreteMidPointStrategy(VectorFieldFStrategy):
    def F(self, j: int, gap_filling_vectors: np.ndarray,
    reconstructed_vectors: np.ndarray, t: np.ndarray):
        x_j = gap_filling_vectors[j]

        # get closest vector
        dis_list = self.get_distance_list(vector=x_j,
    reconstructed_vectors=reconstructed_vectors)
        closest_x_j_index = np.argmin(dis_list)
        while closest_x_j_index - 1 < 0 or any(np.isnan(reconstructed_vectors[closest_x_j_index - 1])):
            dis_list[closest_x_j_index] = np.inf
            closest_x_j_index = np.argmin(dis_list)

        # similar code for second closest
        x_j_bar = reconstructed_vectors[closest_x_j_index]
        #...

        x_j_bar_bar = reconstructed_vectors[sec_closest_x_j_index]
        p_x_j_bar_bar = reconstructed_vectors[sec_closest_x_j_index - 1]
        delta_t = t[1] - t[0]

        return (x_j_bar - p_x_j_bar + x_j_bar_bar - p_x_j_bar_bar) / (2 * delta_t)

    def get_distance_list(self, vector, reconstructed_vectors):
      #...
```

3.3.4 Min Distance Strategies

The `MinDistanceBruteforceStoreAll` class calculates the closest vector indices within the time series, which are essential for identifying the paths likely to be followed by the dynamical system during the missing periods.

```python
class MinDistanceBruteforceStoreAll(MinDistanceStrategy):
    def get_dis_matrix(self, vectors: np.ndarray):
        dis_matrix = np.zeros((len(vectors), len(vectors)))
        for m, i in enumerate(vectors):
            for n, j in enumerate(vectors):
                if any(np.isnan(i)) or any(np.isnan(j)):
                    dis_matrix[m, n] = np.inf
                    continue
                if m == n:
                    dis_matrix[m, n] = np.inf
                    continue
                dis_matrix[m, n] = linalg.norm(i - j)
        self.dis_matrix = dis_matrix
```

3.3.5 Gap Filler Module

This module serves as the integration point for all strategies. It orchestrates the entire process of gap filling from data preparation, gap detection, branching strategies implementation, and finally, the optimization of the gap vectors using the scipy minimization tools to ensure minimal deviation from the expected dynamical behaviour.

Each of these components is designed to be interchangeable, allowing for future upgrades and modifications as new techniques or improvements in computational methods become available. This modular design not only enhances the maintainability of the software but also its adaptability to different types of dynamical systems and datasets.

```python
class GapFiller:
    time_data: np.ndarray  # time
    ts: np.ndarray
    m: int
    t: int
    l: int  # gap length
    n_f: int  # forward branching time
    n_b: int  # backward branching time
    r: int  # stride step
    minimize: Callable[[Any], Any]

    fjs: FunctionalJStrategy
    vffs: VectorFieldFStrategy

    #... other attributes

    def __init__
                 # assign values

    def get_branches_backward
    def get_branches_forward
    def get_one_branch
    def calc_last_valid_closest_neighbor_index
    def reconstruct_timeseries
    def get_breaking_points
    def get_closest_points_layer
    def get_closest_points_one_layer
    def get_gap_vector_and_index_list
    def minimize_gap_vectors
```

3.3.6 Detailed Steps of the Gap Filling Process

```python
mds = MinDistanceBruteforceStoreAll()
ts, time_data = demo_data_generator()
t = 10  # the gap between each sampling
m = 100  # sample vector dimension
n_f = 200  # same n_f in the paper, stride time
n_b = 5  # same n_d in the paper, stride time
r = 2  # stride step
fjs = DiscreteJ1Strategy()
vffs = VectorFieldFDiscreteMidPointStrategy()
gp = GapFiller(time_data=time_data, ts=ts,
      m=m, n_f=n_f, r=r, t=t, n_b=n_b,
                  mds=mds, fjs=fjs, vffs=vffs, minimize=minimize)
gp.get_branches_forward()
gp.get_branches_backward()
gp.get_closest_points_layer()
gp.get_gap_vector_and_index_list()
gp.minimize_gap_vectors()
```

The software executes the gap filling process through several detailed steps outlined below:

1. **Reconstruct the Vector Field:** The vector field is reconstructed based on the embedding delay and embedding dimension, utilizing the

`reconstruct_timeseries` function. The main logic involves generating indices for each dimension of the vector using `np.arange(i, len(self.ts), self.t)`, which are then appended into a list.

2. **Identify the Breaking Points:** The `get_breaking_points` function identifies the breaking points by iterating through the `reconstructed_vectors` and tagging vectors that contain `np.nan` values, determining the predecessor and successor of the gap.

3. **Generate Branches:** Branches are generated using the `get_branches_forward` and `get_branches_backward` functions, based on values `n_f`, `n_b`, and `r`. These functions implement the paper's algorithm, moving through the `reconstructed_vectors` `l` times, where `l` is the number of missing vectors. Every `r` steps, a new branch is initiated by finding the nearest valid vector using the `MinDistanceBruteforceStoreAll` strategy, which stores the distance from any vector to an invalid vector as infinity. This branching continues for `n_f` iterations or until another gap or the end of the vector series is reached. The results are stored in a `Dict[int, Set]`, mapping nodes to sets of child nodes, and maintaining a reverse mapping for backtracking.

4. **Form a Specific Route:** Following the algorithm outlined in the referenced paper, a specific route is formed from the forward and backward branches. The tree structure is converted from node-children form to layer form, and the minimum distances between corresponding layers are calculated to determine the closest points. This structured approach allows for tracing a route through the tree that best fills the identified gap.

5. **Minimize the J Function:** The initial route serves as a guess, which is then refined by minimizing the J function [2] using the `DiscreteJ1Strategy` for the J function and the `VectorFieldFDiscreteMidPointStrategy` for the F function [2]. The `minimize` function from `scipy.optimize` is employed to optimize the vector values, ensuring the reconstructed data closely aligns with the theoretical dynamics of the system.

These steps are executed in sequence to ensure that the filled gaps are not only statistically plausible but also adhere closely to the dynamical expectations of the chaotic system being analyzed.

3.4 Results Analysis

3.4.1 Outcomes and Demonstration

The implemented algorithm successfully fills gaps in chaotic time series, as evidenced by the results detailed in the interactive demo files `filling_gap_implement.ipynb` and `filling_gap_implement_demo.html`. The notebook includes explicit explanations for each function, describing each step, variable type, and shape. Interactive graphs created using Plotly effectively visualize the gap-filling process, while the corresponding static HTML file maintains all outputs and visualizations for a comprehensive demonstration.

3.4.2 Synthetic Data Example

To verify the accuracy of the algorithm, synthetic data generated with the `data_generator` module was used. This module simulates the Lorenz system with parameters `s = 10`, `r = 28`, and `b = 8/3`, which provided an ideal dataset for testing the software's gap-filling capabilities.

3.4.3 Effectiveness and Metrics

The results show that the implemented algorithm accurately reconstructed the missing data. The minimization of the J function significantly improved over 50% during the optimization process. This demonstrates that the gap-filling strategy effectively interpolates missing points and aligns the filled data with the system's dynamics. Metrics such as minimized error rates and improved continuity between data points underscore the reliability of the algorithm in chaotic time series analysis.

4 Discussion

4.1 Challenges and Limitations

During the implementation of the gap-filling software, several challenges were encountered, particularly in the area of computational efficiency and design architecture. One significant challenge is the method used for finding the closest neighbors among the vectors. Currently, the software utilizes a distance matrix approach, which for a time series of 5,000 vectors results in a matrix with 25 million elements, occupying over 100 MB of RAM. This method, while straightforward, is not scalable and can be inefficient for larger datasets.

Furthermore, the software's design could be enhanced by implementing more components as strategy interfaces. Currently, the minimizing function, branching processes, and gap-filling mechanics are hardcoded, which limits flexibility. By abstracting these components into interchangeable strategies, the software could be made more modular and adaptable to different types of time series data or dynamical systems.

4.2 Future Work

The current implementation offers a solid foundation for gap filling in chaotic time series, but there is substantial scope for enhancement. Future work could explore alternative embedding techniques that better capture the dynamics of high-dimensional systems in a lower-dimensional space. The relationship between the embedding delay (`t`), embedding dimension (`m`), and characteristics of the time series (such as standard deviation and mean) could be analyzed to optimize the reconstruction of vector sets.

Additionally, investigating more efficient algorithms for nearest neighbor searches, such as tree-based methods or hashing, could significantly reduce the computational overhead and enhance the software's performance on large datasets. Further development could also include the implementation of machine learning models to predict the optimal parameters for reconstruction based on the properties of the time series, thereby automating and improving the accuracy of the gap-filling process.

5 Conclusion

This paper has presented a software tool designed to address the significant challenge of filling gaps in chaotic time series data. The methodology leverages Takens' Theorem to reconstruct dynamics from incomplete data sets, utilizing a combination of strategies for embedding, branching, and minimizing error metrics.

- **Methodology:** We outlined the software's core algorithms and described its modular architecture, which includes various strategies for calculating dynamics (Functional J Strategies), estimating vector fields (Vector Field F Strategies), and determining minimum distances (Min Distance Strategies). Each component is designed to ensure that the interpolated data maintains the continuity and dynamical integrity of the original time series.

- **Results:** The effectiveness of the software was demonstrated using synthetic data generated from the Lorenz system. The tests confirmed that the software could successfully interpolate missing data with significant accuracy, as evidenced by a more than 50% improvement in the optimization of the J function.

- **Discussion:** Challenges such as the scalability of the nearest neighbor calculations and the flexibility of the software's design were identified. These issues provide a clear direction for future enhancements, including the potential for more sophisticated methods of data embedding and nearest neighbor search.

In conclusion, the developed software represents a valuable tool for researchers and analysts dealing with chaotic time series, enabling more reliable data analysis and helping to mitigate the impact of missing data. Future enhancements will focus on improving the efficiency and adaptability of the tool, ensuring it can meet a broader range of needs within the scientific community.

6 References

References

[1] Takens, F. (1981). Detecting strange attractors in turbulence. In: Rand, D., Young, L.S. (eds) *Dynamical Systems and Turbulence, Warwick 1980. Lecture Notes in Mathematics, vol 898.* Springer, Berlin, Heidelberg. https://doi.org/10.1007/BFb0091924

[2] Packard, N. H., Crutchfield, J. P., Farmer, J. D., & Shaw, R. S. (1980). *Geometry from a Time Series.* Physical Review Letters, 45(9), 712–716. https://arxiv.org/pdf/nlin/0502044