

Basic Approaches for Missing Values



Basic Approaches for Missing Values

We will now discuss a few simple approaches to deal with missing values

We will use **partially synthetic data**

- We will focus on specific (and mostly intact) sections of our series
- Then we will remove values artificially
- ...And measure the accuracy of our filling approaches via the Root MSE

$$RMSE = \sqrt{\frac{1}{n} \sum_{i=0}^n (x_i - \hat{x}_i)^2}$$

Where \hat{x}_i is a value from the filled series and x_i the ground truth

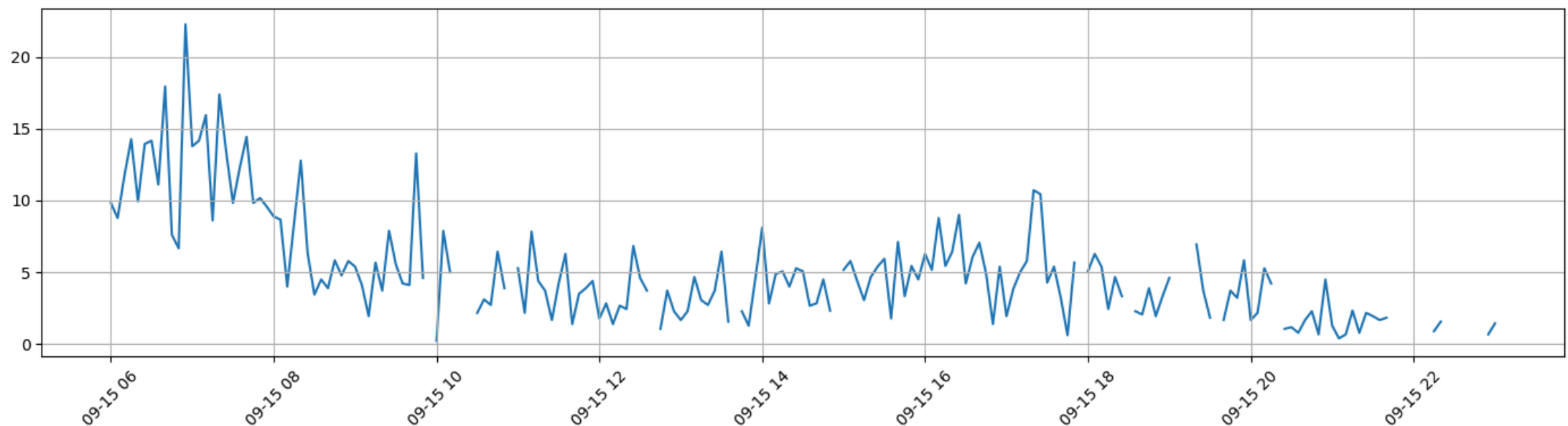
- $x_i = \hat{x}_i$ if no value is missing
- Hence, any MSE difference is entirely due to missing values



The Benchmark Dataset

Our benchmark dataset will consist of this particular stretch from our traffic series

```
In [3]: util.plot_series(ddata, figsize=figsize)
```



- There are (comparatively) few missing values
- Some of them are isolated, some for contiguous "holes"

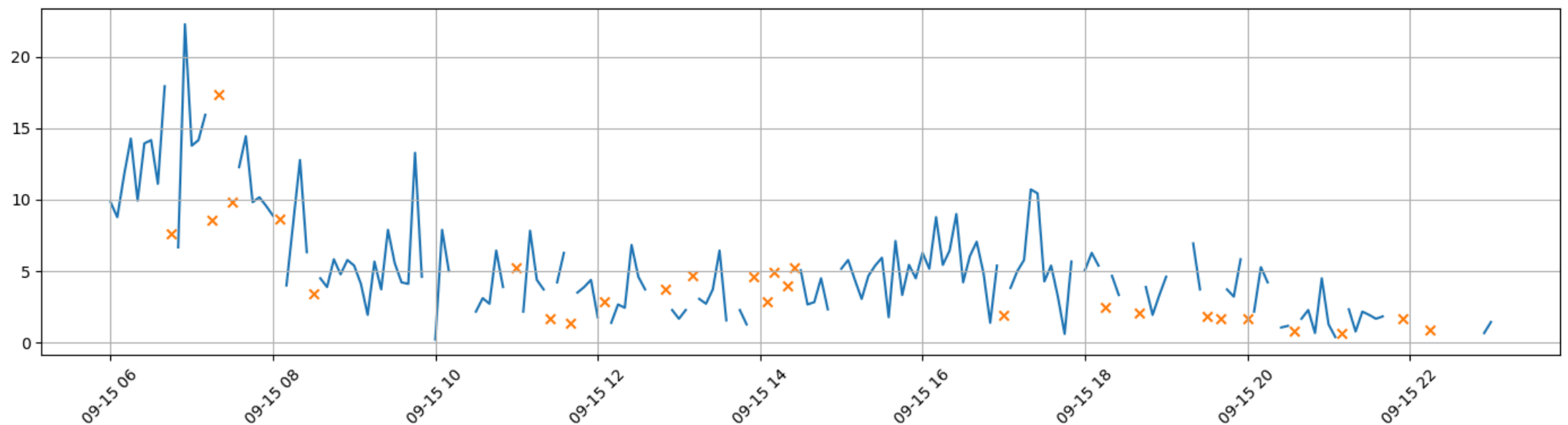


The Benchmark Dataset

We now introduce some missing values

...By drawing them at random:

```
In [5]: np.random.seed(42) # seed (to get reproducible results)
mv_idx = np.random.choice(range(len(ddata.index)), size=30, replace=False)
ddata_mv = ddata.copy()
ddata_mv.iloc[mv_idx] = np.nan
util.plot_series(ddata_mv, figsize=figsize)
plt.scatter(ddata.index[mv_idx], ddata.iloc[mv_idx], color='tab:orange', marker='x');
```



■ The orange markers represent the value that have been removed

Forward/Backward Filling

The easiest approach for missing values consists in **replicating nearby observations**

- **Forward filling**: propagate forward the last valid observation
- **Backward filling**: propagate backward the next valid observation

An important observation:

- When filling missing values, **we have access to the whole series**
- ...So we can reason **both forward and backward**

Forward/backward filling are simple methods, but they can work well

- Rationale: most time series have a certain "inertia"
- ...i.e.: a strong level of local correlation
- For this reason (e.g.) **the last observation is often a good predictor** for the next

one



Forward/Backward Filling on the Benchmark

Forward and backward filling are pre-implemented in pandas

They are available through the `ffill` and `bfill` method:

```
In [9]: nan_mask = ddata['value'].isnull()
ffseries = ddata_mv.ffill()
ffseries[nan_mask] = np.nan # We empty the values that were originally empty
bfseries = ddata_mv.bfill()
bfseries[nan_mask] = np.nan # We empty the values that were originally empty
```

We can check the corresponding RMSE:

```
In [11]: rmse_ff = np.sqrt(mean_squared_error(ddata[~nan_mask], ffseries[~nan_mask]))
rmse_bf = np.sqrt(mean_squared_error(ddata[~nan_mask], bfseries[~nan_mask]))
print(f'RMSE for forward filling: {rmse_ff:.2f}, for backward filling {rmse_bf:.2f}')
```

RMSE for forward filling: 1.33, for backward filling 0.87

- In this case backward filling seems to work better

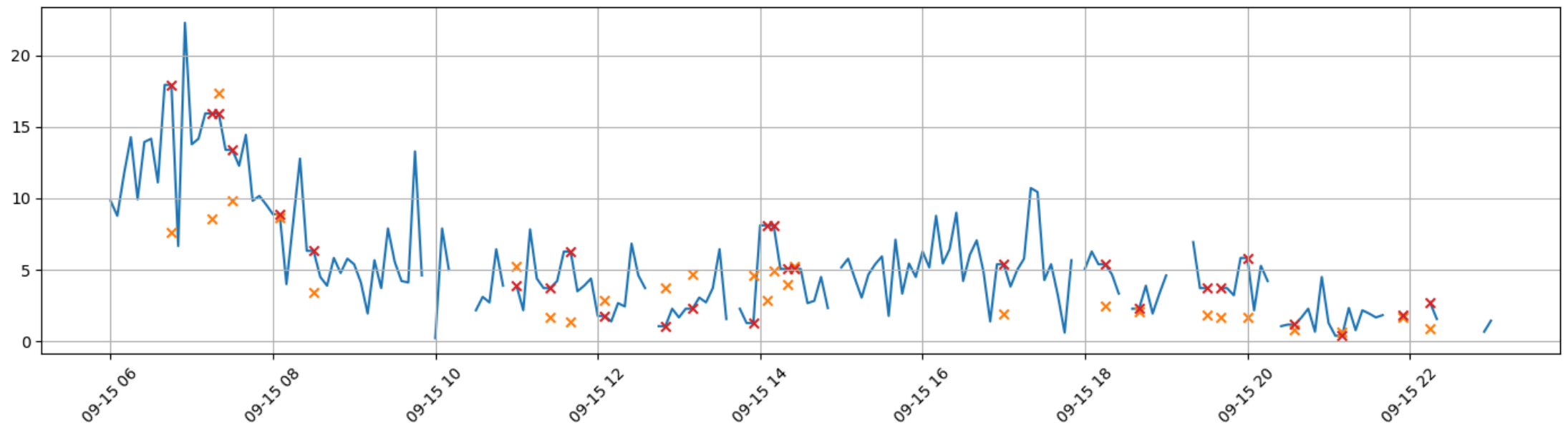


The results are of course application-dependent

Forward/Backward Filling on the Benchmark

Let's have a close look at the results for **forward filling**

```
In [12]: util.plot_series(ffseries, figsize=figsize)
plt.scatter(ddata.index[mv_idx], ddata.iloc[mv_idx], color='tab:orange', marker='x');
plt.scatter(ddata.index[mv_idx], ffseries.iloc[mv_idx], color='tab:red', marker='x');
```



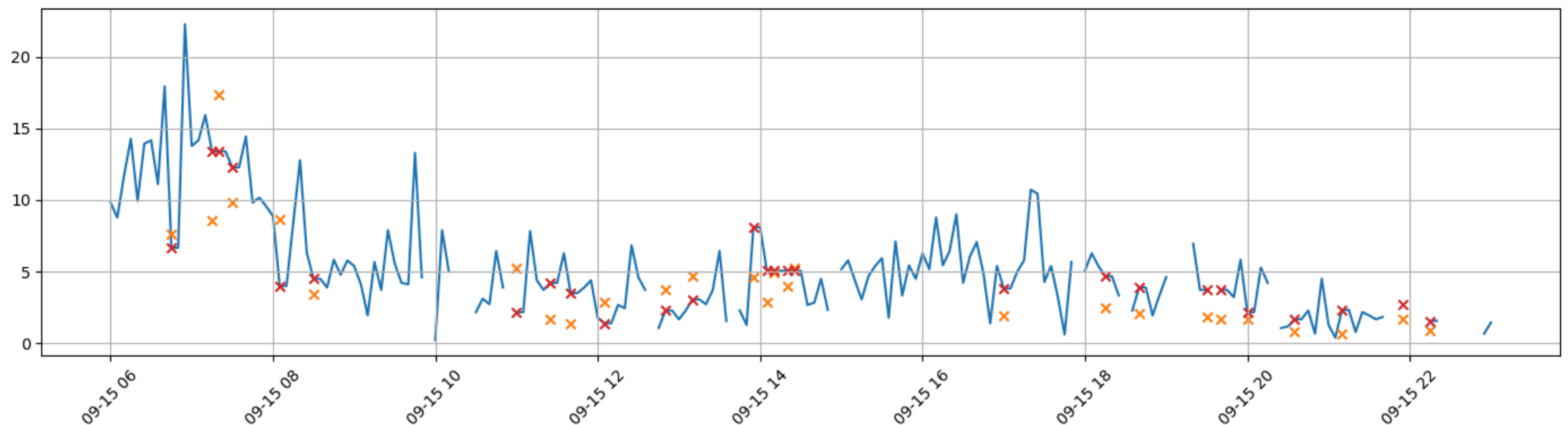
- The red marks represent the values that have been filled



Forward/Backward Filling on the Benchmark

Let's have a close look at the results for **backward filling**

```
In [13]: util.plot_series(bfseries, figsize=figsize)
plt.scatter(ddata.index[mv_idx], ddata.iloc[mv_idx], color='tab:orange', marker='x');
plt.scatter(ddata.index[mv_idx], bfseries.iloc[mv_idx], color='tab:red', marker='x');
```



- Forward/backward filling tend to work well for **low variance** sections
- ...And conversely work worse for **high variance** sections



(Geometric) Interpolation

A few more options are available via the interpolate method

```
DataFrame/Series.interpolate(method='linear', ...)
```

The **method** parameter determines how NaNs are filled:

- "linear" uses a linear interpolation, assuming uniformly spaced samples
- "time" uses a linear interpolation, but supports non-uniformly spaced samples
- "nearest" uses the closest value
- "polynomial" uses a polynomial interpolation
- Even "ffill" and "bfill" are available

Both "polynomial" and "spline" require to specify the additional parameter **order**

- E.g. `df.interpolate(method='polynomial', order='3')`



(Geometric) Interpolation

Let us check the performance of some approaches

```
In [22]: args = [{'method': 'linear'}, {'method': 'time'}, {'method': 'nearest'},  
                {'method': 'polynomial', 'order': 2}, {'method': 'spline', 'order': 4}]  
filling_res = {}  
for a in args:  
    filling_res[a['method']] = ddata_mv.interpolate(**a)  
    filling_res[a['method']][nan_mask] = np.nan  
    rmse = np.sqrt(mean_squared_error(ddata[~nan_mask], filling_res[a['method']][~nan_mask])  
    print(f'RMSE for {a["method"]}: {rmse:.2f}')
```

```
RMSE for linear: 0.98  
RMSE for time: 0.98  
RMSE for nearest: 1.35  
RMSE for polynomial: 1.10  
RMSE for spline: 1.03
```

- "linear" and "time" are equivalent (we have uniformly-spaced samples)
- "polynomial" is the most complex, and in this case also the worst

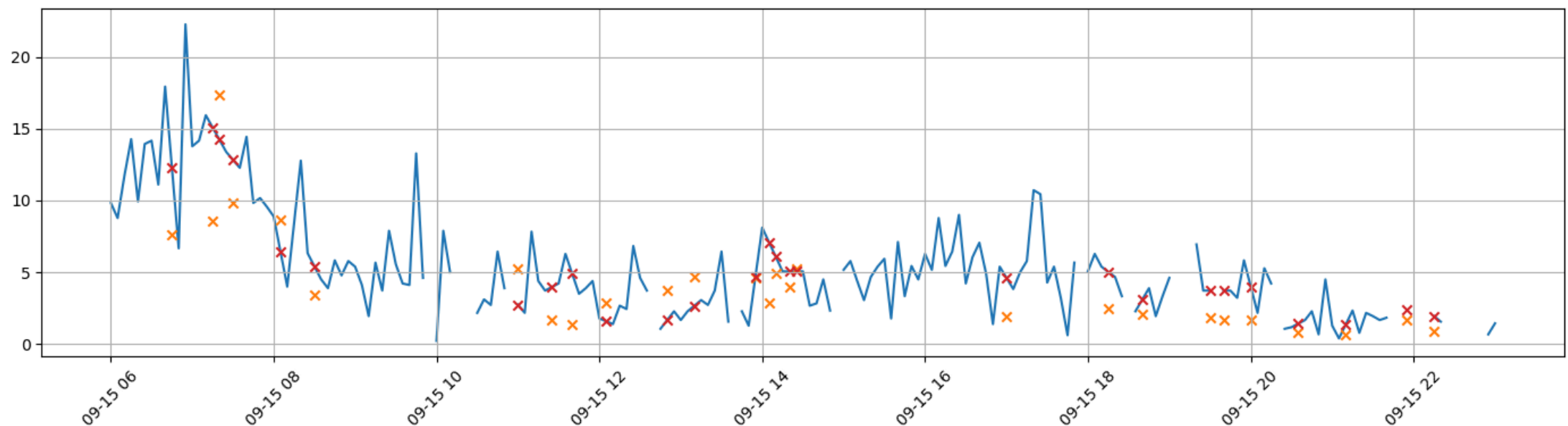
All perform **worse** than backward filling (at least in this case)!



Linear Filling on the Benchmark

Let's have a close look at the results for **linear filling**

```
In [23]: util.plot_series(filling_res['linear'], figsize=figsize)
plt.scatter(ddata.index[mv_idx], ddata.iloc[mv_idx], color='tab:orange', marker='x');
plt.scatter(ddata.index[mv_idx], filling_res['linear'].iloc[mv_idx], color='tab:red', marker='x');
```



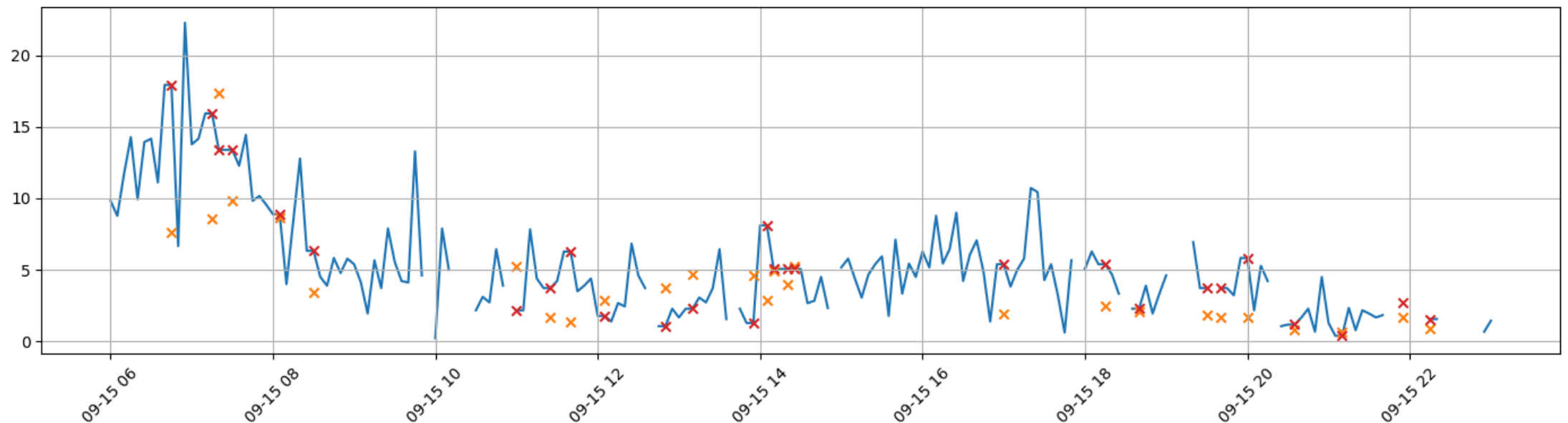
- Linear filling works well for series with **slower dynamics**
- ...But does not work well for series that **faster dynamics**



Nearest Filling on the Benchmark

Let's have a close look at the results for **nearest filling**

```
In [24]: util.plot_series(filling_res['nearest'], figsize=figsize)
plt.scatter(ddata.index[mv_idx], ddata.iloc[mv_idx], color='tab:orange', marker='x');
plt.scatter(ddata.index[mv_idx], filling_res['nearest'].iloc[mv_idx], color='tab:red', marker='x');
```



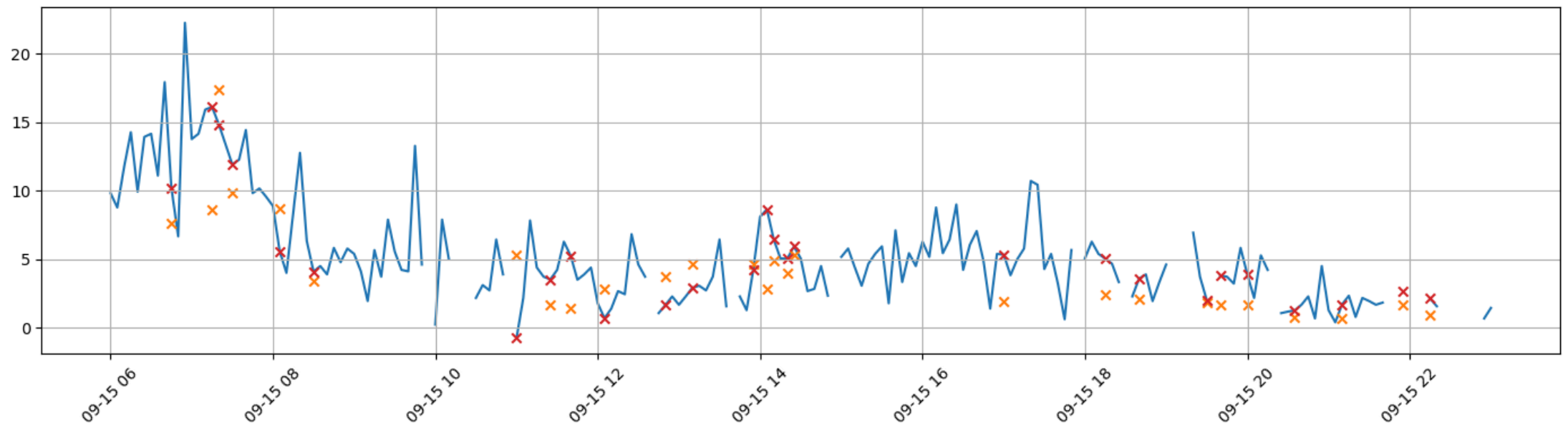
- Nearest filling is a **compromise between forward and backward filling**
- ...And in our case its performance is the average of the two



Polynomial Filling on the Benchmark

Let's have a close look at the results for **polynomial filling**

```
In [26]: util.plot_series(filling_res['polynomial'], figsize=figsize)
plt.scatter(ddata.index[mv_idx], ddata.iloc[mv_idx], color='tab:orange', marker='x');
plt.scatter(ddata.index[mv_idx], filling_res['polynomial'].iloc[mv_idx], color='tab:red', ma
```



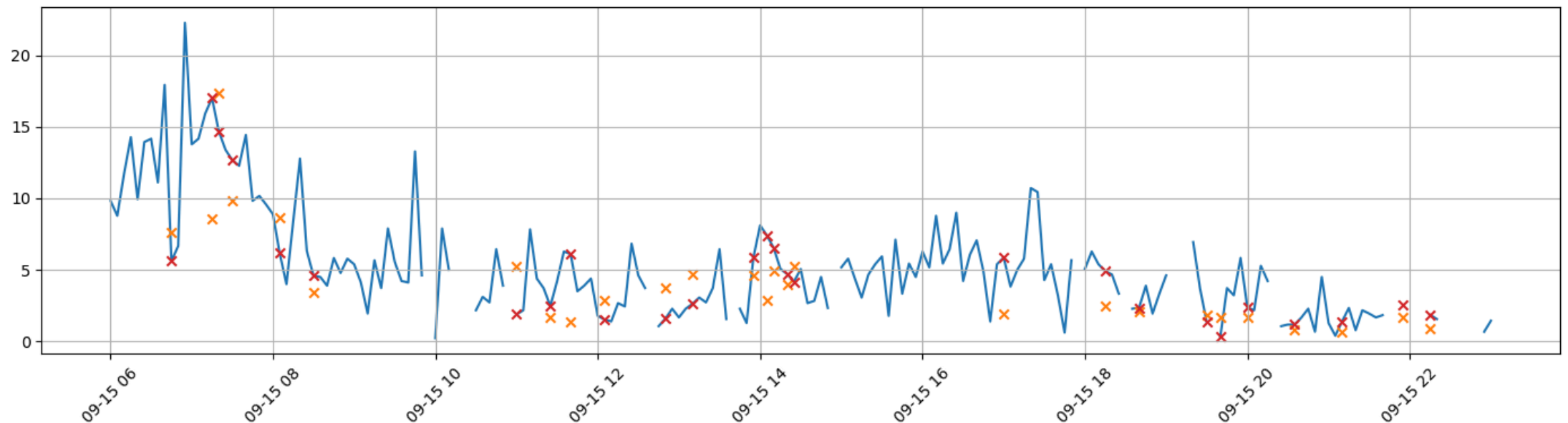
- Polynomial interpolation relies on nearby values to fit a polynomial
- High-order polynomial often vary too much and work less well



Spline Filling on the Benchmark

Let's have a close look at the results for **spline filling**

```
In [27]: util.plot_series(filling_res['spline'], figsize=figsize)
plt.scatter(ddata.index[mv_idx], ddata.iloc[mv_idx], color='tab:orange', marker='x');
plt.scatter(ddata.index[mv_idx], filling_res['spline'].iloc[mv_idx], color='tab:red', marker='x');
```



- Spline interpolation relies on piecewise polynomial curves
- ...And it's often **more robust than polynomial interpolation**



**How do we choose which method is best? Is the
(R)MSE enough?**

