OneShotSTL：用于在线时间序列异常检测与预测的一次性季节-趋势分解

Xiao He Ye Li Jian Tan 阿里巴巴集团 阿里巴巴集团 阿里巴巴集团
xiao.hx@alibaba-inc.com liye.li@alibaba-inc.com j.tan@alibaba-inc.com

Bin Wu 阿里巴巴集团
binwu.wb@alibaba-inc.com

Feifei Li
阿里巴巴集团
lifeifei@alibaba-inc.com

arXiv:2304.01506v1 [cs.LG] 4 Apr 2023

摘要

季节-趋势分解是时间序列分析中最基础的概念之一，支持多种下游任务，包括时间序列异常检测和预测。然而，现有分解方法依赖批处理，时间复杂度为 $O(W)$，其中 W 是时间窗口内的数据点数量。因此，这些方法无法始终高效地支持对处理延迟要求较低的实时分析。为应对这一挑战，我们提出了 OneShotSTL，这是一种高效且准确的算法，可在线分解时间序列，更新时间复杂度为 $O(1)$。OneShotSTL 的速度比批处理方法快 1，000 多倍，准确率与最佳同类方法相当。在真实世界基准数据集上针对下游时间序列异常检测和预测任务开展的大量实验表明，OneShotSTL 的速度比最先进方法快 10 倍至 1，000 多倍，同时仍能提供相当甚至更高的准确率。

PVLDB 引用格式：
Xiao He、Ye Li、Jian Tan、Bin Wu、Feifei Li。OneShotSTL：用于在线时间序列异常检测与预测的一次性季节项-趋势分解。PVLDB，16(6)：1399–1412，2023。doi:10.14778/3583140.3583155

1 引言

随着数据收集和存储技术的快速发展，时间序列的应用数量正在大幅增加，尤其是在物联网（IoT [21]）和 IT 运维（智能运维 [4]）等领域。例如，云数据库系统会收集大量指标，包括 CPU 使用率和请求率，以促进对复杂系统的监控。在这一趋势的推动下，云厂商（例如，AWS CloudWatch [2] 和阿里云监控 [1]）提供了在线服务，以改善其用户对服务器和应用程序的可观测性。

从算法角度来看，时间序列的实时分析任务可分为异常检测（TSAD）或预测（TSF）。对于时间序列 $\{y_1, y_2, \ldots\ldots, y_t, \ldots\ldots\}$，在每个时间点 t，一个在线时间序列异常检测方法会输出当前数据 y_t 的异常分数，而在线时间序列预测方法会预测数据 $\hat{y}_{t+i}$ 在时间 $t+i$ 对应的未来值 $i > 0$。前者用于根据历史观测识别系统的性能问题（例如，通过告警或诊断），后者则用于预测未来，以便通过提前采取某些措施来防止此类问题（例如，自动扩展计算资源）。大规模系统产生的海量数据，加上对低服务延迟的要求，因处理过程具有在线特性，给时间序列异常检测和时间序列预测带来了巨大挑战。

文献中已提出大量用于时间序列异常检测 [12, 14, 27, 38, 44] 和时间序列预测 [11, 29, 34, 35, 45, 47] 的方法。本文聚焦于一类基于季节-趋势分解的算法。季节-趋势分解是时间序列分析中最基础的操作之一，在经济学和金融领域得到了广泛研究和应用 [16, 19]。许多研究表明，季节-趋势分解方法对于时间序列异常检测 [19,40, 42] 和时间序列预测 [18, 29], 十分有效，因为实际时间序列数据通常呈现趋势和季节性模式 [40]。然而，大多数现有的季节-趋势分解方法，例如 STL [16] 和 RobustSTL [40], ，都基于批处理，因此本质上是离线方法，不支持增量更新。唯一的例外是近期提出的 OnlineSTL [28], ，它通过有效利用显式数学闭式表达式带来的快速计算能力，以在线方式分解时间序列。值得注意的是，OnlineSTL 的处理速度比传统季节-趋势分解方法快 100 多倍。然而，它仅支持简单的趋势和季节性滤波方法（例如三次立方核和指数平滑）。因此，OnlineSTL 的分解质量不如功能完备的批处理对应方法，例如引入 l_1-范数正则项的 RobustSTL [40], 。此外，尽管 OnlineSTL 速度很快，其单次更新的复杂度仍为 $O(T)$，其中 T 为季节周期的长度。因此，随着 T 增大，其性能会下降。

为应对上述挑战，本文提出了一种新的在线时序分解算法，称为 OneShotSTL，其更新复杂度为 $O(1)$，可显著加快分解速度。利用新到达的数据点进行观测以更新其影响时，过去观测值中的必要信息由 OneShotSTL 的固定数量状态参数表示；随着在线处理持续进行，这些参数的值会向前传递。因此，OneShotSTL 对每个数据点只处理一次。为提高准确率，我们采用 a l_1-范数正则化项，并在在线计算中自适应地调整季节分量。值得注意的是，OneShotSTL 能够获得准确的分解结果

Xiao He 是通讯作者。本作品采用知识共享署名-非商业性使用-禁止演绎 4.0 国际许可协议授权。访问 https://creativecommons.org/licenses/by-nc-nd/4.0/ 查看本许可协议副本。对于超出本许可协议涵盖范围的任何使用，请发送电子邮件至 info@vldb.org 获取许可。版权归所有者/作者所有。出版权授权给 VLDB 基金会。
《VLDB 基金会会刊》，第 16 卷，第 6 期，ISSN 2150-8097。
doi:10.14778/3583140.3583155

表 1：不同 STD 算法的比较

算法	批处理或在线	趋势变化	季节性偏移	在线复杂度
STL [16]	批处理	No	No	-
TATBS [18]	批处理	No	No	-
STR [19]	批处理	No	No	-
RobustSTL [40]	批处理	Yes	Yes	-
FastRobustSTL [42]	批处理	Yes	Yes	-
OnlineRobustSTL [7]	在线	Yes	Yes	O(T)
OnlineSTL [28]	在线	No	No	O(T)
OneShotSTL	**在线**	**Yes**	**Yes**	**O(1)**

其结果可与 RobustSTL 相媲美（据我们所知，这是文献中最优秀的批处理方法），同时还能大幅缩短处理时间。在时间序列异常检测和时间序列预测任务上开展的大量实验表明，OneShotSTL 性能优异。

主要贡献。本文的贡献如下：

- 据我们所知，所提出的 OneShotSTL 是首个更新复杂度为 $O(1)$ 的在线 STD 方法，这极大提升了现有方法（$O(T)$）。
- OneShotSTL 在实际应用中速度极快。在典型商用服务器上使用单个 CPU核心处理每个数据点大约需要 $20\mu s$，比批量季节-趋势分解方法快 1,000 多倍，并且在季节周期为 12,800 的数据集上比 OnlineSTL 快 20 倍。
- OneShotSTL 同样十分准确。在用于 TSAD 和 TSF 的基准数据集上进行的大量实验表明，OneShotSTL 的速度比最先进方法快 10 倍至超过 1,000 倍，同时准确率相当甚至更高。

2 问题设定与相关工作

由于我们的在线时序分解框架源自批量季节-趋势分解，我们正式介绍批量季节-趋势分解和在线时序分解问题，并回顾该领域现有的相关研究。全文中，我们分别使用小写/大写粗体字母表示向量/矩阵。

2.1 批量季节-趋势分解

对于批处理，考虑以下具有趋势和季节性的模型，该模型针对有限批处理大小为 N 的时间序列 $\boldsymbol{y} \in \mathrm{R}^N$：

$$y_i = \tau_i + s_i + r_i, i = 1, 2, ...N, \tag{1}$$

其中，y_i、τ_i、s_i和 r_i分别表示时间点 i处的观测值、分解趋势、季节项和残差分量。我们仅考虑具有单个长度为 $T < N$ 的季节周期的数据分解，该周期可通过周期检测方法 [41]轻松估计。当存在多个季节项时，也可以如先前研究 [40, 42]所示轻松扩展到这种情况。在本文中，我们将具有多个季节项的数据视为单个季节序列，其中 T 为最长的季节周期。

2.2 在线 STD

对于在线处理，数据点以流式方式到达，表示为{y_1，y_2，……，y_t，……}，其中序列甚至可以是无界的。在时间 t，在线方法根据所有过去的观测值 {y_1，……，y_{t-1}} 和最新的观测值 y_t，仅将 y_t分解为趋势 τ_t、季节项 s_t和残差 r_t，其中 $y_t = \tau_t + s_t + r_t$。与 OnlineSTL [28],类似，我们将在线处理分为两个阶段：单次离线初始化阶段和持续的在线阶段。形式上，在时间 t，我们观测到 {y_1，y_2，……，y_{t_0}，y_{t_0+1}，……，y_t}。前 t_0 个数据点 {y_1，……，y_{t_0}} 用于初始化。在线处理从时间 $t_0 + 1$ 开始，并且仅使用在 t 之前观测到的数据点，持续为 $t > t_0$ 计算 τ_t、s_t 和 r_t。作为固定参数，季节周期 T 可以利用周期检测方法（例如 [39, 41]）根据 {y_1，……，y_{t_0}} 估计。在在线计算阶段，随着数据点不断到达，我们会观测到新的季节周期，每个周期都有其自身的周期长度 T_c。这些周期长度不一定相等。我们假设，流式数据序列中底层周期长度的每个 T_c 虽然未知，但仅会相对于 T 略有偏差。具体而言，T_c 可以是由超参数 H 决定的小区间内的任意整数值，即 $T_c \in [T - H, T + H]$。这一假设可避免周期长度发生剧烈变化，例如 T_c 在多个周期内逐渐从 1000 变为 2000。鉴于现有文献中关于这一主题的大量研究，更一般地处理周期长度变化值得专门研究 [33, 36, 39, 41]。

2.3 现有 STD 方法

批处理 STD 在经济学和金融领域已经被广泛研究了数十年，文献中提出了各种算法 [16, 18, 19, 28, 40]。其中，最流行的是 STL [16]，它采用交替算法，使用局部加权回归（LOcal regrESSion）平滑来估计趋势和季节性分量。随后，TATBS [18] 和 STR [19] 被提出，并为分解结果提供置信区间。然而，它们需要大量计算，尤其是对于季节性较长的数据。此外，这些方法在处理复杂模式时可能会遇到问题，例如趋势的突变（见图4）和季节性偏移（见图 3），这些现象在智能运维中用于监控的指标中很常见。为改善这些问题，RobustSTL [40] 及其加速版本 FastRobustSTL [42] 被提出，采用 l_1-范数趋势滤波和非局部季节性滤波。

所有这些批处理方法都可以用于在线场景，在长度为 $W > T$的滑动窗口上处理，该窗口包含最近的数据点。常见方法是将 W设置为若干个季节周期的长度，例如 $W = 4T$。因此，这些批处理方法的时间复杂度至少为 $O(T)$。然而，正如近期论文 OnlineSTL [28], 所指出的，由于处理延迟较长，大多数方法无法始终直接且高效地用于实时应用。OnlineSTL 是首个在线分解算法，它交替应用三次立方核（趋势）和指数平滑（季节项）滤波，并利用显式数学闭式表达式实现快速计算。OnlineSTL 注重速度，在典型的商用服务器上使用单个中央处理器核心，每秒可处理 10,000 个时间序列数据点。然而，它通过使用简单的趋势和季节项滤波牺牲了质量/准确率，无法处理复杂模式。更重要的是，OnlineSTL 及其他现有在线算法（例如在线 RobustSTL [7, 42]（缩写为 OnlineRobustSTL））的在线更新复杂度仍为 $O(T)$。因此，随着季节性周期长度的增加，性能会下降。

我们在表 1 中总结了批处理和在线时序分解方法的比较，其中包括我们提出的 OneShotSTL。请注意，OneShotSTL 是唯一具有 $O(1)$ 更新复杂度的工作。

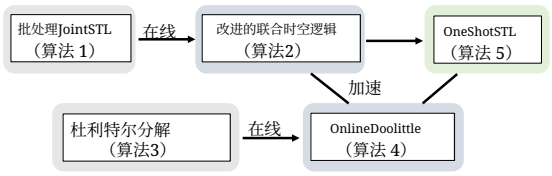


图1：OneShotSTL 算法的开发路线图。

3 模型

本节首先介绍一种新的批处理模型，即 JointSTL（算法 1），该模型在第 3.1 节中同时推导季节项和趋势信号。这个批处理框架是我们开发高效在线算法的起点。然而，要求在线算法精确求解批处理JointSTL问题并不容易。为此，我们在第 3.2 节中引入若干近似方法，通过修改 JointSTL（算法 2）为开发在线算法铺平道路。随后，第 3.3 节介绍 OneShotSTL（算法 5），它使用复杂度为 $O(1)$ 的在线 Doolittle 分解算法（算法 4），在得到与算法 2 完全相同结果的同时，大幅加快了计算速度。最后，我们在第 3.4 节介绍处理季节性偏移的方法。图 1 展示了 OneShotSTL 算法的开发路线。为简化说明，我们先在第 3.1、3.2 和 3.3 节中考虑 $H = 0$（即 $T_c = T$）的情况。在这种情况下，我们可能会滥用符号，将 T 和 T_c 互换使用。T_c 可以是 $[T-H, T+H], H \geq 1$ 中任意整数的一般情况将在第 3.4 节中讨论。

3.1 用于批处理设置的 JointSTL

给定一个时间序列 $\boldsymbol{y} \in \mathrm{R}^N$ 以及输入的季节周期参数T，我们将 JointSTL 的批处理 STD 问题建模如下：

$$\min_{\boldsymbol{\tau},\boldsymbol{s}} \left(\sum_{t=1}^{N} (\tau_t + s_t - y_t)^2 + \sum_{t=T}^{N} (s_t - s_{t-T})^2 + \lambda_1 \sum_{t=2}^{N} |\tau_t - \tau_{t-1}| + \lambda_2 \sum_{t=3}^{N} |\tau_t - 2\tau_{t-1} + \tau_{t-2}| \right) \tag{2}$$

为提高鲁棒性，对趋势分量的一阶和二阶差分使用 l_1-范数，并以 λ_1 和 λ_2 作为超参数来控制平滑度。

JointSTL 可以看作著名的 l_1 趋势滤波 [25]的一种简单扩展。换言之，当设置 $\boldsymbol{s} = \boldsymbol{0}$时，它就退化为 l_1 趋势滤波。为求解优化问题（2），我们采用迭代重加权最小二乘法（IRLS） [22] 方法，这对于进一步扩展到在线场景至关重要。遵循 IRLS 的思路 [22], ，我们引入辅助变量$\boldsymbol{p} \in \mathrm{R}^{N-1}$、 $\boldsymbol{q} \in \mathrm{R}^{N-2}$，并将问题（2）改写为：

$$\min_{\boldsymbol{\tau},\boldsymbol{s},\boldsymbol{p},\boldsymbol{q}} \left(\sum_{t=1}^{N} (\tau_t + s_t - y_t)^2 + \sum_{t=T}^{N} (s_t - s_{t-T})^2 + \lambda_1 \sum_{t=2}^{N} \left(p_t \left(\tau_t - \tau_{t-1}\right)^2 + \frac{1}{4} p_t^{-1} \right) + \lambda_2 \sum_{t=3}^{N} \left(q_t \left(\tau_t - 2\tau_{t-1} + \tau_{t-2}\right)^2 + \frac{1}{4} q_t^{-1} \right) \right) \tag{3}$$

其中， p_t和 q_t 表示 $\boldsymbol{p}$ 和 $\boldsymbol{q}$ 在时间 t的值。

算法 1： 批处理JointSTL

输入： $\boldsymbol{y} \in \mathrm{R}^N, T, \lambda$，最大迭代次数 I
输出： $\boldsymbol{\tau}, \boldsymbol{s}, \boldsymbol{r}$

```
p = q = 1;
for (i = 0; i < I; i ++)
    根据公式（6）构造 A 和 b；
    通过求解线性方程组 Ax = b得到 x；
    τ = x_{1:N}； s = x_{N:2N}； r = y − τ − s；
    根据公式（4）和（5）更新 p、 q；
```

当 p_t和 q_t 按如下方式计算时，问题（3）关于 $\boldsymbol{\tau}$和 $\boldsymbol{s}$等价于问题（2）：

$$p_t = \frac{1}{2\,|\tau_t - \tau_{t-1}|} \tag{4}$$

$$q_t = \frac{1}{2\,|\tau_t - 2\tau_{t-1} + \tau_{t-2}|} \tag{5}$$

只需将公式（4）和（5）代入问题（3），即可得到问题（2），从而证明这一点。

问题（3）可以通过交替优化求解，具体过程如算法 1 所示。根据关于迭代重加权最小二乘的 [13, 23]，算法 1 收敛到问题（2）的全局最优解。当 $\boldsymbol{\tau}$和 $\boldsymbol{s}$ 固定时，更新规则如公式（4）和（5）所示。当 $\boldsymbol{p}$ 和 $\boldsymbol{q}$固定时，将 $\boldsymbol{\tau}$和 $\boldsymbol{s}$ 拼接得到 $\boldsymbol{x} = [\boldsymbol{\tau};\ \boldsymbol{s}]$ 并将其代入问题（3），得到一个最小二乘问题，其最优解由线性方程组 $\boldsymbol{Ax} = \boldsymbol{b}$ 确定，其中

$$\boldsymbol{A} = \boldsymbol{B_1}^T \boldsymbol{B_1} + \boldsymbol{B_2}^T \boldsymbol{B_2} + \lambda_1 (\boldsymbol{p} \cdot \boldsymbol{B_3})^T \boldsymbol{B_3} + \lambda_2 (\boldsymbol{q} \cdot \boldsymbol{B_4})^T \boldsymbol{B_4}$$

$$\boldsymbol{b} = \boldsymbol{B_1}^T \boldsymbol{y}$$

$$\boldsymbol{B_1} = \begin{bmatrix} 1 & & & 1 & & \\ & \ddots & & & \ddots & \\ & & 1 & & & 1 \end{bmatrix} \in \mathbb{R}^{N \times 2N}$$

$$\boldsymbol{B_2} = \begin{bmatrix} \boldsymbol{0} & -1 & \cdots & 1 & & \\ & & \ddots & & \ddots & \\ & & & -1 & \cdots & 1 \end{bmatrix} \in \mathbb{R}^{(N-T) \times 2N} \tag{6}$$

$$\boldsymbol{B_3} = \begin{bmatrix} 1 & -1 & & & & \\ & \ddots & \ddots & & \boldsymbol{0} & \\ & & 1 & -1 & & \end{bmatrix} \in \mathbb{R}^{(N-1) \times 2N}$$

$$\boldsymbol{B_4} = \begin{bmatrix} 1 & -2 & 1 & & & \\ & \ddots & \ddots & \ddots & \boldsymbol{0} & \\ & & 1 & -2 & 1 & \end{bmatrix} \in \mathbb{R}^{(N-2) \times 2N}$$

在问题（2）中，我们对残差和季节性分量的前两项使用平方 l_2范数。原则上，可以像 RobustSTL 那样将它们替换为 l_1范数。不过，第一项（残差）使用平方 l_2范数、趋势项使用 l_1范数，符合经典的 l_1 趋势滤波 [25]；此外，我们的实际评估也表明，采用（2）能获得更稳健的结果。

3.2 面向在线场景的 ModifiedJointSTL

问题（2）和算法 1 是为批处理场景设计的。在本节中，我们对它们进行修改，使其适用于在线场景。

对于在线场景，整个数据处理流程由初始化阶段和在线更新阶段组成。具体而言，对于数据序列 $\{y_1, y_2, \ldots\ldots y_{t_0}, y_{t_0+1}, \ldots\ldots y_t\}$，前 t_0 个数据点 $\{y_1, y_2, \ldots\ldots y_{t_0}\}$ 用于初始化，$t_0 \geq W$，例如，$W = 4T$其中 T 为最长季节周期，如 OnlineSTL 中所示。我们可以采用批处理模型（例如 STL）对 $\{y_1, y_2, \ldots\ldots y_{t_0}\}$ 进行处理，从而得到 $\{\tau_1, \tau_2, \ldots\ldots \tau_{t_0}\}$ 和 $\{s_1, s_2, \ldots\ldots s_{t_0}\}$。因此，在在线阶段分解第一个数据点 t_0+1 时，我们已经获得了先前数据点的季节分量。此外，每次在线分解后，我们都可以在滑动窗口中保留最新周期的季节分量。我们将季节缓冲区命名为 $\boldsymbol{v} \in \mathrm{R}^T$，并将其初始化为 $\boldsymbol{v} = [s_{t_0-T}, \ldots, s_{t_0}]^T$。然后，我们修改问题（2）中关于季节分量的第二项，并将问题改写为：

$$\begin{aligned}\min_{\boldsymbol{\tau},\boldsymbol{s}}\Bigg(&\sum_{j=t_0+1}^{t_0+M}\left(\tau_j + s_j - y_j\right)^2 + \sum_{j=t_0+1}^{t_0+M}\left(s_j - v_{j\%T}\right)^2 \\ &+\lambda_1\sum_{j=t_0+1}^{t_0+M}\left|\tau_j - \tau_{j-1}\right| + \lambda_2\sum_{j=t_0+1}^{t_0+M}\left|\tau_j - 2\tau_{j-1} + \tau_{j-2}\right|\Bigg)\end{aligned} \tag{7}$$

其中，对于每个在时间戳 $t = t_0 + M$ 到达的新数据点 y_t，我们将所有 y_j 分解为 τ_j 和 s_j，以得到 $j \in [t_0+1, t_0+M]$，但只输出最后一次分解结果 τ_{t_0+M} 和 s_{t_0+M}。

类似地，我们可以引入辅助变量 $\boldsymbol{p} \in \mathrm{R}^M$ 和 $\boldsymbol{q} \in \mathrm{R}^M$，并使用迭代重加权最小二乘 [22] 来求解问题（7）。p_t 和 q_t 的更新规则与式（4）和（5）相同。但当 $\boldsymbol{p}$ 和 $\boldsymbol{q}$ 固定时，我们求解一个不同的线性方程组，以得到 $\boldsymbol{\tau}$ 和 $\boldsymbol{s}$。具体而言，我们将 $\boldsymbol{\tau}$ 和 $\boldsymbol{s}$ 拼接起来，构成 $\boldsymbol{x} = \{\tau_{t_0+1}, s_{t_0+1}, \ldots\ldots, \tau_{t_0+M}, s_{t_0+M}\}$。最优的 $\boldsymbol{x}$ 是线性方程组 $\boldsymbol{Ax} = \boldsymbol{b}$ 的解，满足

$$\begin{aligned}\boldsymbol{A} &= \boldsymbol{C_1}^T\boldsymbol{C_1} + \boldsymbol{C_2}^T\boldsymbol{C_2} + \lambda_1(\boldsymbol{p}\cdot\boldsymbol{C_3})^T\boldsymbol{C_3} + \lambda_2(\boldsymbol{q}\cdot\boldsymbol{C_4})^T\boldsymbol{C_4} \\ \boldsymbol{b} &= \boldsymbol{C_1}^T\boldsymbol{y} + \boldsymbol{C_2}^T\boldsymbol{u}, \text{where } \boldsymbol{u} = [\boldsymbol{v}^T, \ldots, \boldsymbol{v}^T]^T \in \mathbb{R}^M \\ \boldsymbol{C_1} &= \begin{bmatrix} 1 & 1 & & & \\ & \ddots & & & \\ & & & 1 & 1 \end{bmatrix} \in \mathbb{R}^{M\times 2M} \\ \boldsymbol{C_2} &= \begin{bmatrix} 0 & 1 & & & \\ & \ddots & & & \\ & & & 0 & 1 \end{bmatrix} \in \mathbb{R}^{M\times 2M} \\ \boldsymbol{C_3} &= \begin{bmatrix} 1 & 0 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & 0 & -1 \end{bmatrix} \in \mathbb{R}^{(M-1)\times 2M} \\ \boldsymbol{C_4} &= \begin{bmatrix} 1 & 0 & -2 & 0 & 1 & & \\ & \ddots & \ddots & \ddots & \ddots & \ddots & \\ & & 1 & 0 & -2 & 0 & 1 \end{bmatrix} \in \mathbb{R}^{(M-2)\times 2M}\end{aligned} \tag{8}$$

然而，上述方法仍然求解的是一个批处理问题。此外，基于该方法很难推导出高效的在线算法。请注意，在在线场景中，我们的目标是对新到达的数据点进行分解。因此，我们针对在线场景修改了 JointSTL，使其能够很好地近似最后一次分解，如算法2所示。

在算法2的初始化阶段，我们对 $\{y_1, \ldots, y_{t_0}\}$ 使用 STL 或 JointSTL，并按照前文所述将 $\boldsymbol{v}$ 设为。然后我们

算法2：在线场景下改进的批处理JointSTL

```
//初始化阶段
输入： y_1, ..., y_{t_0}、T、λ，最大迭代次数 I
通过 STL 或 JointSTL 获得 τ, s, r；设置 v = [s_{t_0-T}, ..., s_{t_0}];
p^0 = q^0 = [1]; p^i = q^i = [], 对于 i ∈ [1, I];

//在线 阶段
输入： y = {y_1, y_2, ..., y_t}, T, λ, I
Output: τ_t, s_t, r_t
for (i = 0; i < I; i ++)
    使用 p^i、q^i，根据公式（8）构造 A 和 b；
    通过求解线性方程组 Ax = b 获得 x；
    τ_t = x_{-2}; s_t = x_{-1}; r_t = y_t − τ_t − s_t;
    根据公式（4）和（5）更新 p_t 和 q_t；
    p^{i+1} = [p^{i+1}, p_t]; q^{i+1} = [q^{i+1}, q_t];
p^0 = [p^0, 1]; q^0 = [q^0, 1];
v_t%T = s_t;
```

为第 i 次迭代设置 $\boldsymbol{p} = \boldsymbol{p}^i$ 和 $\boldsymbol{q} = \boldsymbol{q}^i$，用于在公式（8）中构造 $\boldsymbol{A}$ 和 $\boldsymbol{b}$。因为在在线阶段，对于每次迭代 i，我们都有使用公式（4）和（5）更新的不同 $\boldsymbol{p}^i$ 和 $\boldsymbol{q}^i$ 向量。与 JointSTL 在分解 y_t 时更新整个 $\boldsymbol{p}$ 和 $\boldsymbol{q}$ 向量不同，我们只将根据第 i 次迭代中对 τ_t 和 s_t 的分解计算得到的最新 p_t 和 q_t 追加到 $\boldsymbol{p}^i$ 和 $\boldsymbol{q}^i$。最后，我们输出最后一次迭代的分解结果。在实践中（参见第5节的实验），我们发现这种近似能提供足够好的分解。

然而，算法2仍不切实际，因为对于每个时间点 $t = t_0 + M$，都需要在算法2的每次迭代中（第4–8行）求解一个线性方程组 $\boldsymbol{Ax} = \boldsymbol{b}$，其中 $\boldsymbol{A} \in \mathrm{R}^{2M\times 2M}$，且随着在线处理更多数据点，其规模不断增大。回想一下，在在线场景中，在时间点 t，只需部分求解 $\boldsymbol{Ax} = \boldsymbol{b}$，并输出最新的分解 τ_t 和 s_t。此外，由公式（8）可知，无论 M 如何增长，也无论 $\boldsymbol{p}$ 和 $\boldsymbol{q}$ 取何值，$\boldsymbol{A}$ 都是带宽为9的对称带状矩阵，因为对于任意 $M \geq 5$，$\boldsymbol{C_1}^T\boldsymbol{C_1}$、$\boldsymbol{C_2}^T\boldsymbol{C_2}$、$\boldsymbol{C_3}^T\boldsymbol{C_3}$ 和 $\boldsymbol{C_4}^T\boldsymbol{C_4}$ 都是带宽分别为3、3、5和9的带状矩阵。因此，可以使用乔列斯基/杜利特尔因式分解和高斯消元高效求解线性方程组 $\boldsymbol{Ax} = \boldsymbol{b}$。关键在于，乔列斯基/杜利特尔因式分解以及高斯消元中的前向代入都可以在线进行。具体而言，在每个时间点和算法2的每次迭代中，我们只需执行回代的前几步，并输出 $\boldsymbol{x}$ 的最后两个元素，即 τ_t 和 s_t。基于这一思路，我们提出了 OneShotSTL，详情见下一节。

3.3 OneShotSTL 算法

本节介绍在线对称杜利特尔分解算法，其更新复杂度为 $O(1)$，可加速算法2，从而得出完整的 OneShotSTL 算法。

3.3.1 在线对称杜利特尔分解。 对于每个时间点 t，算法2需要依次求解 $\boldsymbol{Ax} = \boldsymbol{b}$ 的 I 个不同线性方程组，其中 I 是一个用于配置数量的参数。

迭代次数。请注意，数字 I是预先设定的，且与输入数据规模无关，例如， $I=8$。这是因为，基于迭代重加权最小二乘使用算法2对问题（7）进行优化时，通常收敛得非常快 [22]。因此，我们只需进行少量迭代即可获得令人满意的结果（参见第5节中的实验）。该算法分为两个步骤。第一步初始化首次迭代，第二步则基于上一次迭代的结果重复剩余的 $I-1$ 次迭代。

算法3： 对称杜利特尔分解

输入: $\boldsymbol{A} \in \mathbb{R}^{M\times M}$
输出： $\boldsymbol{L} \in \mathbb{R}^{M\times M}, \boldsymbol{D} \in \mathbb{R}^{M\times M}$

对于 $(k = 1; k < M; k ++)$
  $L_{kk} = 1;$
  $D_{kk} = A_{kk} - \sum_{i=1}^{k-1} D_{ii} \cdot L_{ki}^2;$
  对于 $(j = k+1; j < M; j ++)$
    $L_{kj} = 0;$
    $L_{jk} = (A_{jk} - \sum_{i=1}^{k-1} L_{ji} \cdot D_{ii} \cdot L_{ki}) / D_{kk};$

在时间点 $t = t_0 + M$，我们可以在算法2的第一次迭代中使用公式（8）和 $\boldsymbol{p} = \boldsymbol{q} = \mathbf{1}$ 构造 $\boldsymbol{A}^{\boldsymbol{t}} \in \mathrm{R}^{2M\times 2M}$。类似地，对于下一个时间点 $t+1$，我们可以构造 $\boldsymbol{A}^{\boldsymbol{t+1}}$。有趣的是， $\boldsymbol{A}^{\boldsymbol{t}}$和 $\boldsymbol{A}^{\boldsymbol{t+1}}$ 的某些子矩阵是相同的。图 2展示了 $\boldsymbol{A}^{\boldsymbol{t}}$和 $\boldsymbol{A}^{\boldsymbol{t+1}}$ 的两个示例，分别对应 $M=5$和 $M=6$。如图所示，我们可以根据虚线将每个矩阵分成4个部分。 $\boldsymbol{A}^{\boldsymbol{t}}$和 $\boldsymbol{A}^{\boldsymbol{t+1}}$的左上角子矩阵是相同的，并且在追加全零行和列后， $\boldsymbol{A}^{\boldsymbol{t}}$和$\boldsymbol{A}^{\boldsymbol{t+1}}$ 的右上角/左下角子矩阵也相同。唯一的差异在于红框中的右下角子矩阵，其中 $\boldsymbol{A}^{\boldsymbol{o}} \in \mathrm{R}^{4\times 4}$ （ $\boldsymbol{A}^{\boldsymbol{t}}$中的）被 $\boldsymbol{A}^{*} \in \mathrm{R}^{6\times 6}$ （ $\boldsymbol{A}^{\boldsymbol{t+1}}$中的）替换。很容易验证，对于任何 $M \geq 5$，也会出现相同的情况。

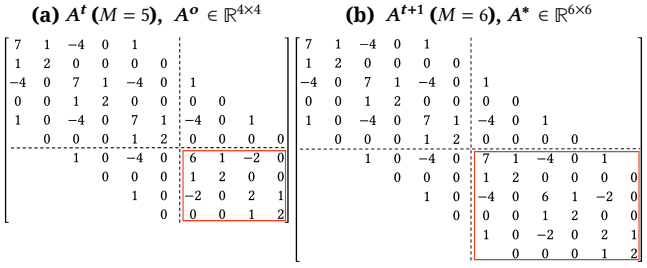


图 2： $\boldsymbol{A}$与 $\boldsymbol{p}$=$\boldsymbol{q}$=1 和 $\lambda_1=\lambda_2=1$的示例。

接下来，我们将说明如何利用相邻时间戳中 $\boldsymbol{A}^{\boldsymbol{t}}$ 和 $\boldsymbol{A}^{\boldsymbol{t+1}}$ 的共同部分进行在线处理。首先，我们可以采用算法3中所示的对称杜利特尔分解（SDF）对 $\boldsymbol{A} = \boldsymbol{L}\boldsymbol{D}\boldsymbol{L}^T$进行因式分解，其中 $\boldsymbol{D}$ 是对角矩阵， $\boldsymbol{L}$ 是下三角矩阵。然后，在使用 SDF 对 $\boldsymbol{A}^{\boldsymbol{t}} = \boldsymbol{L}^{\boldsymbol{t}}\boldsymbol{D}^{\boldsymbol{t}}\boldsymbol{L}^{\boldsymbol{t}T}$进行因式分解时，我们可以在处理 $\boldsymbol{A}^{\boldsymbol{o}}$之前缓存最新的 $\boldsymbol{L}^{\boldsymbol{t}}$和 $\boldsymbol{D}^{\boldsymbol{t}}$（图 2(a)）。此外，由于 $\boldsymbol{L}^{\boldsymbol{t+1}}$ 和 $\boldsymbol{D}^{\boldsymbol{t+1}}$ 的左上部分与 $\boldsymbol{L}^{\boldsymbol{t}}$ 和 $\boldsymbol{D}^{\boldsymbol{t}}$相同，在对 $\boldsymbol{A}^{\boldsymbol{t+1}}$ 进行因式分解的同时，我们可以继续处理时间点 $t+1$ 的$\boldsymbol{A}^{*}$ （图 2(b)）。最后，由于 $\boldsymbol{A}$是一个长度为 9 的带状矩阵，我们只需缓存大小为 $\mathrm{R}^{10\times 10}$ 的 $\boldsymbol{L}$ 和 $\boldsymbol{D}$ 子矩阵，因为其他部分要么为 0，要么不会用于计算。

完整的 OnlineDoolittle 算法，用于部分解决$\boldsymbol{Ax} = \boldsymbol{b}$ 在线问题，见算法4，包括前向

算法4： OnlineDoolittle

输入： $\boldsymbol{A}^{*} \in \mathbb{R}^{6\times 6}$， $\boldsymbol{b}^{*} \in \mathbb{R}^{6}$, ($\boldsymbol{L}^{\boldsymbol{o}} \in \mathbb{R}^{8\times 4}$， $\boldsymbol{D}^{\boldsymbol{o}} \in \mathbb{R}^{4\times 4}$ 以及 $\boldsymbol{b}^{\boldsymbol{o}} \in \mathbb{R}^{4}$ 首次调用需要的内容)
Output: τ_t， s_t， r_t

$\boldsymbol{L}^{\boldsymbol{o}}$、 $\boldsymbol{D}^{\boldsymbol{o}}$ 和 $\boldsymbol{b}^{\boldsymbol{o}}$ 在首次调用时初始化，并在线更新;
$\bar{\boldsymbol{L}} = \text{zeros}(10,10)$; $\bar{\boldsymbol{L}}_{1:8,1:4} = \boldsymbol{L}^{\boldsymbol{o}}$;
$\bar{\boldsymbol{D}} = \text{zeros}(10,10)$; $\bar{\boldsymbol{D}}_{1:4,1:4} = \boldsymbol{D}^{\boldsymbol{o}}$;
$\bar{\boldsymbol{b}} = [\boldsymbol{b}^{\boldsymbol{o}}; \boldsymbol{b}^{*}]$;
for $(k = 5; k \leq 10; k ++)$
  $\bar{L}_{kk} = 1;$
  $\bar{D}_{kk} = A^{*}_{k-4,k-4} - \sum_{i=k-4}^{k} \bar{D}_{ii} \cdot \bar{L}_{ki}^2; \bar{b}_k = -\sum_{i=k-4}^{k} \bar{L}_{ki} \cdot \bar{b}_i$; //前向代入部分1
  for $(j = k+1; j \leq 10; j ++)$
    $\bar{L}_{jk} = (A^{*}_{j-4,k-4} - \sum_{i=k-4}^{4} \bar{L}_{ji} \cdot \bar{D}_{ii} \cdot \bar{L}_{ki}) / \bar{D}_{kk};$
for $(j = 5; j \leq 10; j ++)$
  for $(i = j+1; i < j+5; i ++)$
    $\bar{b}_i -= \bar{L}_{ij} \cdot \bar{b}_j$; //前向代入部分2
for$(j = 10; j \leq 5; j --)$
  对于 $(i = j+1; i \leq 10; i ++)$
    $\bar{b}_i -= \bar{D}_{ii} \cdot \bar{L}_{ji} \cdot \bar{b}_j$; //回代
$\boldsymbol{L}^{\boldsymbol{o}} = \bar{\boldsymbol{L}}_{3:10,3:6}$; $\boldsymbol{D}^{\boldsymbol{o}} = \bar{\boldsymbol{D}}_{3:6,3:6}$; $\boldsymbol{b}^{\boldsymbol{o}} = \bar{\boldsymbol{b}}_{3:6}$;
$\tau_t = \bar{b}_9$; $s_t = \bar{b}_{10}$; $r_t = y_t - \tau_t - s_t$;

以及高斯消元的回代，这些步骤可以采用相同的原理进行修改。需要注意的是，OnlineDoolittle与原始的SDF（算法3）几乎相同，在这种方法中，我们通过在线进行因式分解和前向/回代，持续更新最新的 $\boldsymbol{L}^{\boldsymbol{o}}$、 $\boldsymbol{D}^{\boldsymbol{o}}$ 和 $\boldsymbol{b}^{\boldsymbol{o}}$。最后，它会针对每个时间戳$t$输出 τ_t 和 s_t 的分解结果。算法4所需的计算次数始终为常数，因此其运行时间复杂度为 $O(1)$。

接下来，我们说明 OnlineDoolittle 算法也可用于部分求解 Algorithm 2 的第 i 次迭代中的 $\boldsymbol{Ax} = \boldsymbol{b}$。为了在第 i 次迭代中使用公式（8）构造 $\boldsymbol{A}$ 和 $\boldsymbol{b}$，我们需要使用上一轮迭代更新后的 $\boldsymbol{p}^{i-1}$ 和 $\boldsymbol{q}^{i-1}$。请注意，在每个时间点 t，我们只会将第 $i-1$ 次迭代中计算出的 p_t 和 q_t 追加到 $\boldsymbol{p}^{i-1}$ 和 $\boldsymbol{q}^{i-1}$。因此，在第 i 次迭代处理时间点 t 时，所有先前时间戳对应的 $\boldsymbol{p}_{1:t-1}$ 和 $\boldsymbol{q}_{1:t-1}$ 的值都是固定的。因此，如图 2 所示，第 i 次迭代中相邻时间戳的 $\boldsymbol{A}^{\boldsymbol{t}}$ 和 $\boldsymbol{A}^{\boldsymbol{t+1}}$ 也具有相同的性质。随后，我们可以为每次迭代建立一个规模不断增长的线性方程组，其构造依赖于上一轮迭代的结果。最终，可以使用 OnlineDoolittle 算法在线部分求解这些方程组，从而得到最终的分解结果。

3.3.2 ***OneShotSTL*** **算法详细信息。** 结合之前介绍的所有组成部分，我们得到了完整的 OneShotSTL 算法，如算法 5 所示。它以在线方式解决与算法2相同的问题。在初始化阶段，我们对现有数据执行 STL 或批处理JointSTL，以获得一个周期的季节分量 $\boldsymbol{v}$。在在线阶段，在时间点 t，OneShotSTL 处理新的观测值 y_t，并输出分解结果 τ_t 和 s_t。具体而言，对于每次迭代 i，它使用之前迭代中的 τ_t 和 s_t，依次更新 p_t 和 q_t，并使用公式（8）按照图 2 构建 $\boldsymbol{A}^{*}$ 和 $\boldsymbol{b}^{*}$。然后，每次迭代都会增量应用对称 Doolittle

算法 5： OneShotSTL

//初始化阶段

输入： $y_1, ..., y_{t_0}$, T, λ, 最大迭代次数 I

$M = 0$；通过 STL 或 JointSTL 获取 $\boldsymbol{\tau}$，$\boldsymbol{s}$，$\boldsymbol{r}$；设置 $\boldsymbol{v} = [s_{t_0-T}, ..., s_{t_0}]$；

$\boldsymbol{p}^0 = \boldsymbol{q}^0 = [1]$；$\boldsymbol{p}^i = \boldsymbol{q}^i = []$，适用于 $i \in [1, I]$；

//在线阶段

输入： y_t, t, n, λ, I

输出： τ_t，s_t，r_t

对于 ($i = 0; i < I; i$++)

  使用公式（8）构造 $\boldsymbol{A}^*$ 和 $\boldsymbol{b}^*$，如图 2 所示；

  τ_t, s_t, r_t = OnlineDoolittle$_i$($\boldsymbol{A}^*, \boldsymbol{b}^*$);

  使用公式（4）和（5）更新 p_t 和 q_t；

  $\boldsymbol{p}^{i+1} = [\boldsymbol{p}^{i+1}, p_t]$; $\boldsymbol{q}^{i+1} = [\boldsymbol{q}^{i+1}, q_t]$;

$\boldsymbol{p}^0 = [\boldsymbol{p}^0, 1]$; $\boldsymbol{q}^0 = [\boldsymbol{q}^0, 1]$;

$v_{t\%T} = s_t$;

在算法4中，使用 OnlineDolittle 通过 $\boldsymbol{A}^*$ 和 $\boldsymbol{b}^*$ 进行因式分解，记为第 i 次迭代的 OnlineDolittle$_i$。最后，我们输出最后一次迭代的分解结果 τ_t 和 s_t，并使用分解得到的季节项 s_t 更新 $\boldsymbol{v}$ 中的相应值。

OnlineDolittle（算法4）的时间复杂度为 $O(1)$。因此，OneShotSTL 的最坏情况在线更新复杂度为 $O(I)$，其中 I 是最大迭代次数。请注意，I 是一个固定常数，与问题规模无关，这意味着 OneShotSTL 的在线更新复杂度为 $O(1)$。在实践中，它是影响 OneShotSTL 运行速度的主要因素。不过，较小的取值（例如 $I = 8$）就足以获得令人满意的结果；参见第5节中的实验)。

3.4 处理季节性偏移

本节处理季节性偏移的情况，此时底层的 T_c 可以是 $[T - H, T + H], H \geq 1$ 中的任意整数。

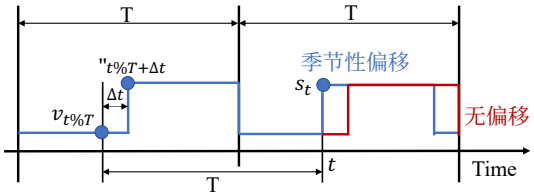


图 3： Δt **个时间点的季节性偏移。**

图 3 展示了一个发生季节性偏移的示例，偏移了 Δt 个时间点；这种情况在智能运维的指标监控应用中十分常见。蓝线表示时间序列数据中季节分量两个周期的真实值。从时间点 t 开始，发生了 Δt 个时间点的偏移。红色虚线表示不存在偏移时的假想数据。为简单起见，我们假设直到时间点 t，$\boldsymbol{v}$ 都等于图 3 中的第一个季节周期。显然，算法 5 中的 OneShotSTL 在时间点 t之后无法很好地分解该序列，因为 $v_{t\%T}$与真实值 s_t有显著差异。如果我们知道 Δt，就可以将问题（7）中的 $(s_t - v_{t\%T})$ 替换为 $(s_t - v_{t\%T+\Delta t})$，以解决季节性偏移问题。然而，在实践中，我们不知道每个时间点 t对应的正确 Δt。

为解决这个问题，我们首先使用超参数H，用 H内所有可能的 Δt构成邻域 E，例如 E $= \{0\}$ 表示】【。

算法 6： NSigma

输入： r_t，n

输出： isAnomaly，score

是异常 =False;

如果 未初始化 **则**

  sum = sumSquared = count = 0;

如果 计数 > 0**则**

  均值 = 总和 / 计数；STD = 平方根(sumSquared / 计数 - 均值^2);

  分数 =绝对值(r_t - 均值) / STD;

  如果 分数 > n **则**

    是异常 = 真;

计数 += 1；总和 += r_t；平方和 += r_t^2;

$H = 0$ 和 E $= \{-1, 0, 1\}$ 对于 $H = 1$。此外，我们观察到，当发生季节性偏移时，分解残差 r_t 中会出现尖峰异常点。因此，我们监控残差r_t，并使用算法 6 中所示的流式 NSigma 算法来检测对于每个时间点 t，r_t 中是否存在异常。在算法 6 中，n 是一个输入参数，表示异常程度，例如，$n = 5$。一旦发现异常，我们就对用于构建 $\boldsymbol{A}^*$ 和 $\boldsymbol{b}^*$（第 4 行）的每个 $\Delta t \in$ E，重复算法 5 的第3 – 8 行。然后，我们选取 Δt，其r_t的绝对值最小，并输出相应的 τ_t和 s_t。在我们的模拟中，这种方法得出了最准确的分解结果。

采用上述方法处理季节性偏移后，OneShotSTL 的计算复杂度为 $O(I \cdot H)$，其中 H 是预先设定的参数，表示允许的周期长度偏移最大变化量，例如 $H = 20$。由于 I 和 H 都是固定常数，与每个周期长度 T_c 无关，这意味着 OneShotSTL 的在线更新复杂度仍为 $O(1)$。此外，在实践中，这也不会对 OneShotSTL 的速度产生太大影响，因为残差信号中的异常（例如，具有 $n > 5$）很少见（通常少于 1%）。

4 下游任务扩展

OneShotSTL 和其他在线 STD 方法可以轻松扩展到在线时间序列异常检测和时间序列预测等下游任务，如下所示：

(1) **单变量时间序列异常检测**：对于无界时间序列，在每个时间点 t，在线时间序列异常检测方法都会观察所有历史数据 $\{y_1, y_2, ..., y_t\}$，并输出 y_t 的异常分数。通过结合算法 6 中所示的流式 NSigma，OnlineSTL 和 OneShotSTL 可以扩展为在线时间序列异常检测方法。除了输出分解结果外，它们还可以使用 NSigma 基于分解后的残差信号输出异常分数。

（2）**单变量时间序列预测**：类似地，在线时间序列预测方法在时间点 t观测 $\{y_1, ..., y_t\}$，并对时间点 $t + i$的数据未来值 $\hat{y}_{t+i}$进行 i步预测。季节信号具有周期性，因此我们可以利用它们进行长序列预测。在在线分解期间，我们采用一种简单的方法，缓存最近的分解趋势（τ_{t-1}）和一个周期的季节信号（$\boldsymbol{v}$）。然后，为了在时间戳 t进行 i步预测，我们只需使用 $\hat{y}_{t+i} = \tau_{t-1} + v_{t\%T+i}$，其中 T为周期长度。相同的方法也适用于 OnlineSTL。

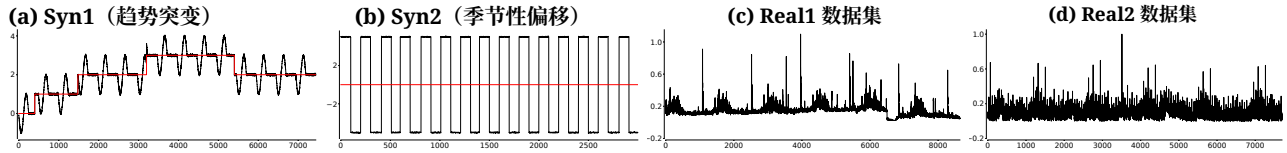


图4：用于评估 STD 质量的合成与真实世界数据集。

5 实验

我们通过全面的实验，将 OneShotSTL 与其他方法进行比较评估。在介绍实验设置后，我们评估分解准确率和可扩展性，随后比较其在下游时间序列异常检测和时间序列预测任务上的表现。

5.1 实验设置

5.1.1 数据集. 评估使用了三组数据集。

1. 合成和真实世界 STD 数据集。 我们生成了两个合成数据集（**Syn1** 和**Syn2**），并使用两个真实世界数据集（**Real1** 和**Real2**），如图4所示。这两个真实世界数据集记录了阿里云数据库系统中某些内部 API 的请求速率。图4 (a) 和 (b) 中的红线表示合成数据集趋势信号的真实值。**Syn1**和**Syn2** 分别针对趋势突变和季节性偏移的情况设计。在**Syn2** 中，有4个偏移了10个数据点的周期，这些偏移在视觉上无法区分。由于篇幅有限，合成数据集的生成细节见补充材料 [10]。对于这两个真实世界数据集，**Real1** 呈现出趋势突变，而**Real2** 则包含可观察到的噪声和弱季节性。

2. 单变量时间序列异常检测数据集。 我们使用 TSB-UAD 基准 [31]评估单变量时间序列异常检测任务。TSB-UAD 基准包含 2000 多条由领域专家标注异常的真实世界时间序列。其中最著名的是 KDD CUP 2021（KDD21）时间序列异常检测竞赛数据集 [17]，包含 250 条单变量时间序列。有关这些数据集的详细信息，请参阅 [31]。对于 KDD21、IOPS、NASA-MSL 和 NASA-SMAP 数据集，我们使用数据中的训练部分进行初始化。对于其余数据集，我们使用前 3000 个数据点进行初始化。

3. 单变量时间序列预测数据集。我们在来自 Informer [45] 的六个真实世界数据集（ETT、Electricity、Exchange、Traffic、Weather 和 Illness ）上评估时间序列预测任务。这些数据集已被近期研究广泛用于长序列时间序列预测基准测试，例如 FEDformer [47] 和 FiLM [46]。我们遵循这些方法的相同实验设置，Illness 的预测长度为 {24，36，48，*60}*，其余数据集的预测长度为 *{96*，192，336，720}。训练集、验证集和测试集的划分详情见 [45]，相关数据已公开提供 [5]。

5.1.2 对比方法。我们将 OneShotSTL 与最先进的 STD、时间序列异常检测和时间序列预测基线方法进行比较。

1. 时间序列分解基线方法。 我们将 OneShotSTL 与两种批量季节-趋势分解方法（STL [16]和 RobustSTL [40, 42]）以及两种在线方法（OnlineSTL [28] 和 OnlineRobustSTL [7, 42]）进行比较。在在线场景下，也可以在最近的数据点构成的滑动窗口上使用 STL 和 RobustSTL，分别称为 Window-STL 和 Window-RobustSTL。我们使用公开的 Java 版 STL 实现 [6]，并用 Java 实现 OnlineSTL 和 OneShotSTL。对于 RobustSTL，我们使用 SREWorks [7]软件包中公开的 Python 实现。

2. 时间序列异常检测基线方法。我们将其与基于矩阵轮廓的方法 NormA [14],STOMPI [44],SAND [15], 以及 DAMP [27], 进行比较，这些方法在 TSB-UAD基准上展现了单变量时间序列异常检测的最先进性能。NormA 是一种批处理时间序列异常检测方法，而 STOMPI、SAND 和 DAMP 是在线时间序列异常检测方法。批处理方法读取时间序列的全部数据，并为每个数据点输出一个异常分数，而在线方法使用训练数据进行初始化，并在在线检测期间更新模型。进一步地，我们还将其与基于深度学习的时间序列异常检测方法进行比较，包括 LSTM [32], USAD [12] 以及 TranAD [38]。USAD 和 TranAD 是近年来设计用于快速训练的方法。我们使用 TSB-UAD基准提供的 LSTM、NormA 和 SAND 的 Python 实现 [31]。对于 USAD 和 TranAD，我们分别使用作者从 [20] 和 [37] 提供的公开可用 Python 代码。对于 STOMPI，我们使用 *STUMPY* [26] Python 软件包中的代码。DAMP 使用 Matlab 实现，并已公开提供 [43]。

3. 时间序列预测基线方法。我们选取了六种时间序列预测方法进行比较，包括自回归积分滑动平均模型 [11], DeepAR [34], NBEATS [29], Informer [45],FEDformer [47],和 FiLM [46]。自回归积分滑动平均模型是一种传统统计方法，其余五种是基于深度学习的方法，其中包括最新的方法（例如 FEDformer 和 FiLM），它们在所评估的数据集上具有最先进的预测准确率。对于自回归积分滑动平均模型，我们使用 *statsforecast* [9] Python 包中的实现。对于 DeepAR 和 NBEATS，我们使用 *pytorch-forecasting* [8]中的代码。对于 Informer [45], FEDformer [47] 和 FiLM [46],，我们使用作者提供的 Python 代码，这些代码均已公开。

5.1.3 评估指标。我们还针对不同的时间序列分析任务采用了多种评估指标。

1. 分解指标。我们使用合成数据集上分解结果与真实信号之间的平均绝对误差（MAE）来评估分解质量。此外，我们展示了合成数据集和真实世界 STD 数据集上的分解信号可视化结果。

2. 单变量时间序列异常检测指标。近期有研究指出，TSAD 方法的传统评估指标（如精确率、召回率和曲线下面积）对噪声标签过于敏感 [24, 30]。因此，我们采用近期提出的 VUS-ROC 评估方法 [30] （该方法也由 TSB-UAD 的作者提出）进行评估。研究表明，与其他 TSAD 评估指标相比，VUS-ROC 对噪声标签更加稳健 [30]。此外，在 KDD21数据集 [17] 上，我们采用 KDD CUP 2021 TSAD 竞赛中的评分函数进行评估。由于 KDD21 数据集中每条时间序列只有一个异常事件，该函数只检查排名最高的异常点是否位于标记的异常事件的邻域内。

3. 单变量时间序列预测指标。我们采用测试集中真实值与预测值之间的平均绝对误差（MAE）来评估预测任务。

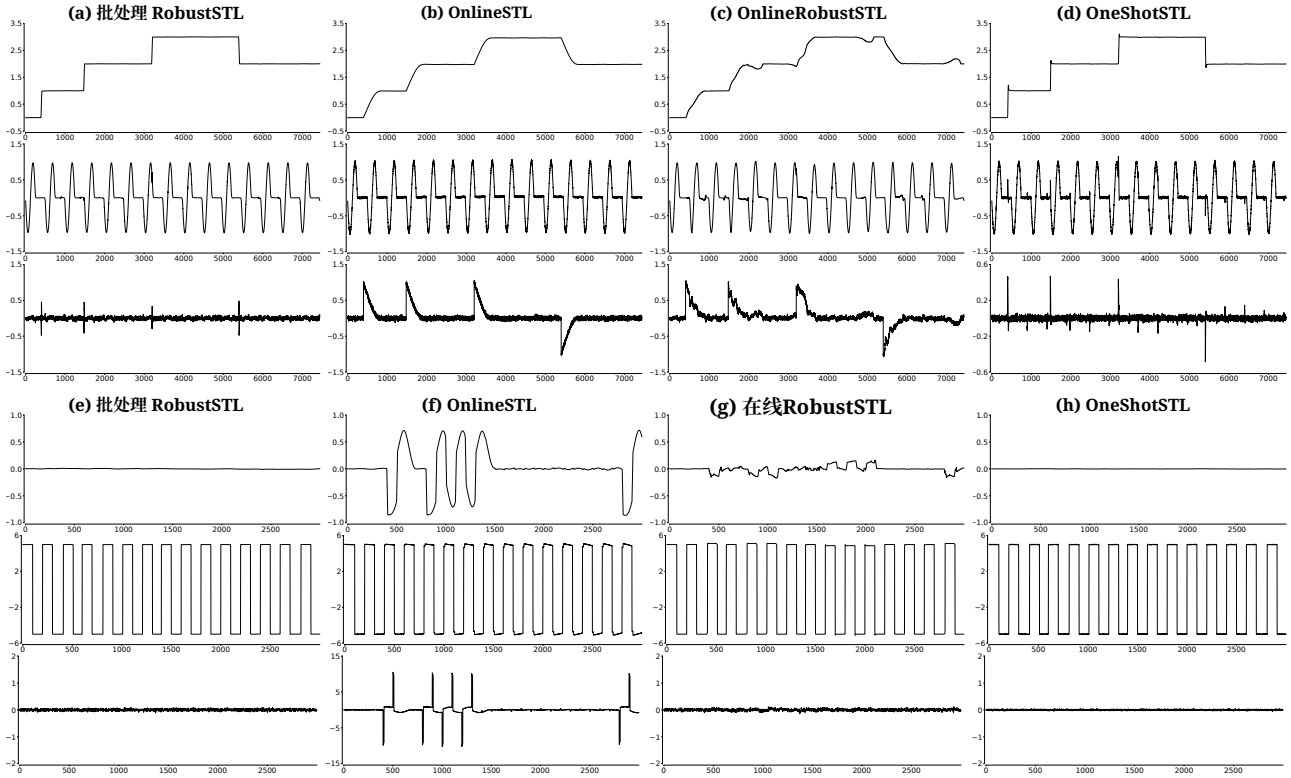


图 5：Syn1（(a)-(d)）和 Syn2（(e)-(h)）上的分解结果比较。

5.1.4 超参数设置。 所有 STD 和时间序列异常检测方法（深度学习方法除外）都需要将周期长度 T作为输入。对于合成数据集，我们使用真实值 $T = 500$ 作为**Syn1**，以及 $T = 250$ 作为**Syn2**。对于真实世界数据集，我们使用 TSB-UAD 软件包中的查找_长度函数 [3] 来检测 T，该函数基于自相关。

1. STD 方法设置。STL 不需要任何其他超参数。RobustSTL 需要十几个超参数，我们采用作者推荐的默认值 [7]。按照作者的建议，我们将 On- lineSTL 的 $\alpha = 0.7$ 设为 [28]。OneShotSTL 的关键超参数是 λ_1 和 λ_2，它们控制分解趋势的平滑度。我们始终将它们设为与 $\lambda = \lambda_1 = \lambda_2$ 相同，并按以下方式进行调整。在训练数据上，我们使用 $\lambda \in \{10_0, 10_1, \ldots\ldots, 10_4\}$运行 STL 和 OneShotSTL。然后，我们选取与 STL 结果最接近（平均绝对误差最小）的结果。如果未另行说明，我们将 NSigma 的季节性偏移窗口 $H = 20$、最大迭代次数 $I = 8$ 和 $n = 5$ 设为。

2. TSAD 方法设置。 我们使用作者建议的超参数，这些参数通常是默认值。对于在 TSB-UAD 中实现的方法，其默认超参数已用于 TSB-UAD 基准论文 [31] 中的评估。

3. TSF 方法设置。对于 Informer、FEDformer 和 FiLM，我们使用作者已在所评估的数据集上经过广泛调优的超参数。对于自回归积分滑动平均模型，我们使用来自 *statsforecast* [9] 包的无参数 AutoArima。对于 DeepAR 和 NBEATS，初始学习率和隐藏层大小的超参数按照验证集上的以下方式进行调优说明 [8]。对于 OnlineSTL 和 OneShotSTL，我们分别使用验证集调优 α和 λ_1 以及 λ_2。

5.1.5 硬件与设置. NSigma、DAMP、OnlineSTL 和 OneShotSTL 的所有实验均在配备 4 个 CPU 核心（Intel Core i5）和 16 GiB 内存的 Macbook Pro 2018 上进行。其他方法的实验在阿里云上进行，使用一台 ECS（ecs.gn6e、48 个 CPU 核心、368 GiB 内存、NVIDIA V 100×4 GPU 卡）。对于所有基于深度学习的方法，均使用 NVIDIA V100 GPU 卡以高效地进行训练和测试。

5.2 OneShotSTL 的准确度如何?

我们首先将 OneShotSTL 的分解质量与此前的 STD 方法进行比较。图 5 展示了 Syn1（从 (a) 到 (d)）和 Syn2（从 (e) 到 (h)）上的分解结果。如子图 (a) 至 (d) 所示，OnlineSTL 和 OnlineRobustSTL 在 Syn1 上输出的趋势过于平滑，无法恢复趋势的真实信号，而 OneShotSTL 则成功处理了这类情况。此外，OnlineSTL 无法处理 Syn2 上的季节性偏移，输出的趋势信号和残差信号与真实信号有显著差异。相比之下，OneShotSTL 能像批处理方法 RobustSTL 一样完美地处理这类情况。此外，表 2 展示了批处理与在线方法分解得到的信号与真实信号之间的平均绝对误差。可以看出，在两个合成数据集上，RobustSTL 和 OneShotSTL（以粗体显示）分别是表现最佳的批处理方法和在线方法。两者的平均绝对误差非常接近，且显著优于其他所有对比方法。

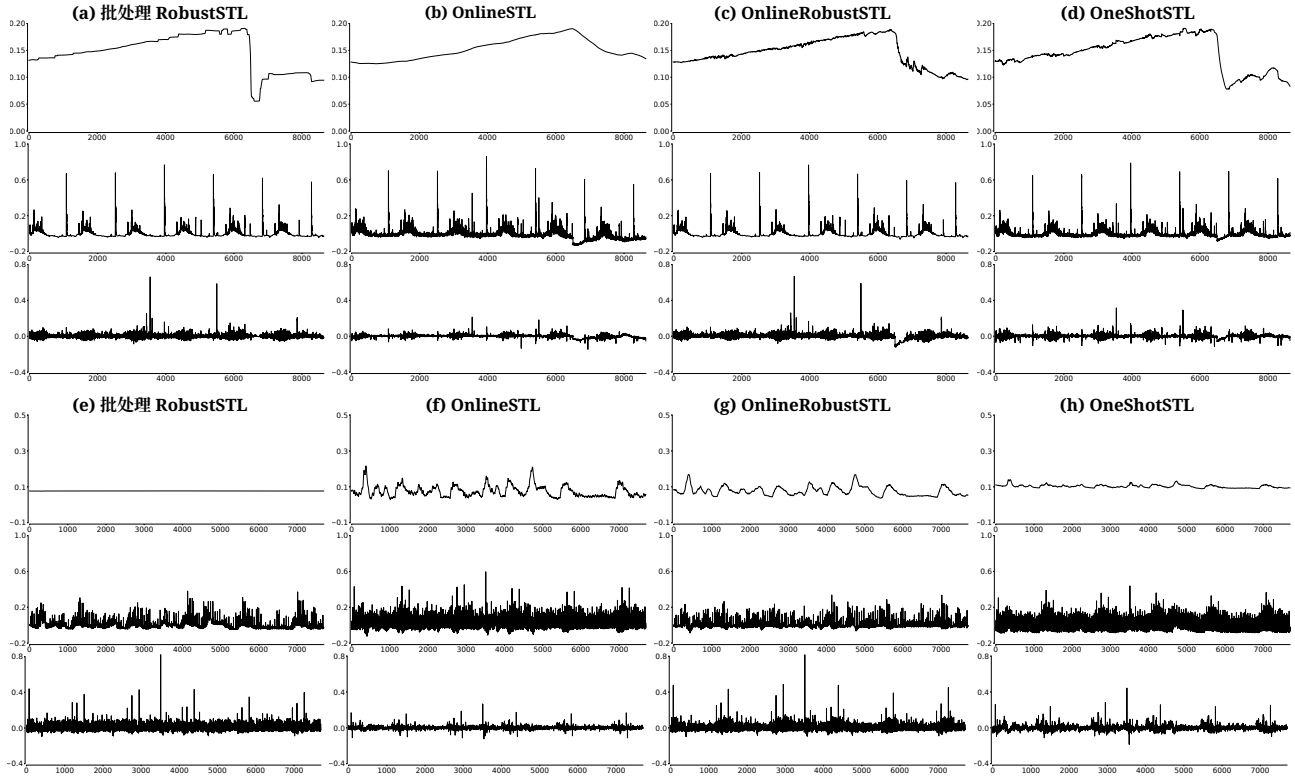


图6：Real1（(a)-(d)）和 Real2（(e)-(h)）数据集上的分解结果可视化。

表 2：合成数据集上的分解对比

Data	Type	算法	趋势 (平均绝对误差)	季节项 (平均绝对误差)	残差 (平均绝对误差)
Syn1	批处理	STL	0.134	0.015	0.144
		RobustSTL	**0.004**	**0.013**	**0.016**
	在线	Window-STL	0.134	0.092	0.174
		OnlineSTL	0.104	0.023	0.093
		Window-RobustSTL	0.045	0.018	0.046
		OnlineRobustSTL	0.131	0.033	0.123
		OneShotSTL	**0.007**	**0.014**	**0.019**
Syn2	批处理	STL	0.084	0.433	0.505
		RobustSTL	**0.004**	**0.004**	**0.004**
	在线	Window-STL	0.084	0.313	0.313
		OnlineSTL	0.225	0.374	0.571
		Window-RobustSTL	0.032	0.031	**0.006**
		OnlineRobustSTL	0.037	0.031	0.013
		OneShotSTL	**0.004**	**0.013**	0.013

接下来，我们在图 6 的 (a) 至 (d) 和 (e) 至 (h) 中分别可视化真实世界数据集 Real1 和 Real2 的分解信号。如图 6 的子图 (a) 至 (d) 所示，OneShotSTL 对 Real1 分解得到的趋势信号比 OnlineSTL 和 OnlineRobustSTL 更接近 RobustSTL 的结果，因此能更好地发现趋势的突变。此外，从图 6 子图 (e) 至 (h) 中 Real2 的结果可以看出，OnlineSTL 和 OnlineRobustSTL 分解得到的趋势信号波动较大。这可能会增加基于分解趋势信号进行异常检测时发生误报的可能性。相比之下，OneShotSTL 能更好地处理这种情况。

5.3 OneShotSTL 有多快?

接下来，我们比较不同 STD 方法的效率。请记住，OneShotSTL 是唯一更新复杂度为 $O(1)$ 的在线方法。所有其他方法的更新复杂度均为 $O(T)$。具体而言，我们通过重复**Syn1**，构建了一个包含 200,000 个数据点的新合成数据集。请注意，所有 STD 方法的运行时间仅取决于输入时间序列的大小和周期长度 T。然后，我们使用不同的输入 $T \in \{100, 200, 400, 800, 1600, 3200, 6400, 12800\}$评估这些分解方法，并报告处理一个数据点的平均延迟。我们评估这一范围的 T，因为在智能运维的实时指标监控场景中，通常存在每日或每周的季节性模式。因此，对于分钟级数据，每日和每周季节性的周期长度分别为 1,440 和 10,080。

结果如图 7 所示。除 OneShot- STL 外，所有方法的延迟均随 T 的增加而呈线性增长。OneShotSTL 是唯一一种处理单个数据点时延迟保持恒定（约为 $20\mu s$）的方法，这与第 3 节中的理论分析一致。粗略而言，使用单个 CPU核心，每秒最多可处理 50,000 个时间序列。当 $T < 800$时，OnlineSTL 的表现更好；而当 $T > 800$时，OneShotSTL 的速度更快。当 $T = 12,800$ 时，OnlineSTL 分解单个数据点需要约 $450\mu s$，其速度比 OneShotSTL 慢 20 倍以上。其他三种方法 Window-STL、Window-RobustSTL 和 OnlineRobustSTL 的速度至少比 OnlineSTL 和 OneShotSTL 慢两个数量级。因此，在后续对时间序列异常检测和时间序列预测任务的评估中，我们仅纳入 OnlineSTL 和 OneShotSTL。

表 3：TSB-UAD基准中 16 个数据集上的单变量时间序列异常检测结果（VUS-ROC）。VUS-ROC 越高，表示效果越好。方法的排名显示在括号中，最佳结果以粗体显示。

方法	LSTM	USAD	TranAD	NormA	SAND	STOMPI	DAMP	NSigma	OnlineSTL	OneShotSTL
Daphnet	0.476 (7)	0.653 (3)	0.548 (6)	0.462 (10)	0.594 (5)	0.465 (7)	0.475 (8)	0.605 (4)	0.696 (2)	**0.735 (1)**
道奇队	0.743 (9)	**0.872 (1)**	0.768 (7)	0.815 (4)	0.788 (6)	0.794 (5)	0.630 (10)	0.825 (2)	0.756 (8)	0.819 (3)
ECG	0.745 (9)	**0.936（1）**	0.759 (7)	0.935 (2)	0.884（4）	0.891 (3)	0.864 (5)	0.715 (10)	0.754 (8)	0.767 (6)
创世记	0.507 (10)	**0.827 (1)**	0.742 (4)	0.669 (5)	0.507 (9)	0.530 (8)	0.644 (7)	0.823 (2)	0.666 (6)	0.762 (3)
GHL	0.627 (8)	**0.913 (1)**	0.831 (4)	0.759 (5)	0.727（7）	0.603 (9)	0.553（10）	0.732 (6)	0.866 (3)	0.872 (2)
IOPS	0.743 (3)	0.740 (5)	0.690 (6)	0.503 (9)	0.487 (10)	0.550 (8)	0.616 (7)	**0.774 (1)**	0.741 (4)	0.749 (2)
MGAB	0.683 (2)	0.628 (10)	0.647 (4)	0.628 (9)	0.637 (5)	0.628 (8)	**0.731 (1)**	0.665 (3)	0.634 (6)	0.628 (7)
MITDB	0.613 (10)	0.745 (5)	0.667（8）	**0.824（1）**	0.792（2）	0.746（4）	0.773（3）	0.671（7）	0.664 (9)	0.685 (6)
NAB	0.510 (10)	**0.631 (1)**	0.551 (6)	0.580 (4)	0.519 (9)	0.520 (8)	0.540 (7)	0.570 (5)	0.584 (3)	0.590（2）
NASA-MSL	0.613 (8)	**0.699 (1)**	0.650 (6)	0.645 (7)	0.695 (3)	0.606 (10)	0.609 (9)	0.694 (5)	0.696 (2)	0.688 (5)
NASA-SMAP	0.642 (8)	0.711 (4)	0.551 (10)	**0.818 (1)**	0.800 (2)	0.711 (3)	0.665 (6)	0.642 (7)	0.601 (9)	0.666 (5)
占用率	0.532 (6)	0.421 (7)	0.845 (2)	0.626 (5)	0.254 (10)	0.399 (9)	0.408（8）	0.770 (4)	**0.855 (1)**	0.840 (3)
Opportunity	0.564 (6)	0.481 (7)	0.454 (8)	**0.808 (1)**	0.777（2）	0.767（3）	0.707 (4)	0.586（5）	0.418 (9)	0.404 (10)
SensorScope	0.546 (10)	0.555 (8)	0.573 (7)	0.632（4）	0.551（9）	0.587（6）	0.609（5）	0.668（2）	0.646（3）	**0.672 (1)**
SMD	0.606（6）	0.697（4）	0.699（3）	0.612 (9)	0.580 (10)	0.622 (8)	0.635 (7)	0.680（5）	0.732 (2)	**0.749 (1)**
SVDB	0.650 (7)	0.731（5）	0.647 (10)	**0.924 (1)**	0.896 (2)	0.830 (3)	0.815 (4)	0.649 (8)	0.647 (9)	0.690 (6)
YAHOO	0.768 (6)	0.626 (9)	0.677 (8)	**0.915 (1)**	0.895 (2)	0.544 (10)	0.810 (4)	0.761 (7)	0.833 (3)	0.796 (5)
VUS-ROC平均值	0.624 (10)	0.698 (3)	0.664 (7)	**0.713 (1)**	0.669 (6)	0.634 (9)	0.652（8）	0.695 (4)	0.693（5）	**0.713 (1)**
平均排名	7.35 (10)	4.29 (2)	6.23 (8)	4.58 (3)	5.70 (6)	6.70 (9)	6.17 (7)	4.82 (4)	5.11 (5)	**4.00 (1)**
平均时间（秒）	13995 秒 (9)	7465 秒 (8)	545 秒 (4)	1164 秒 (5)	23601 秒 (10)	3689 秒（6）	5298 秒（7）	**2 秒（1）**	61 秒（2）	160 秒（3）

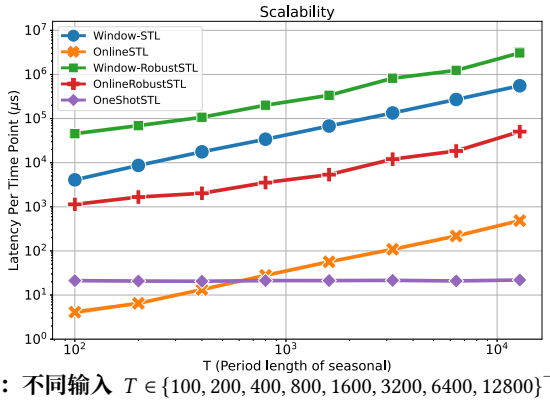


图 7：不同输入 $T \in \{100, 200, 400, 800, 1600, 3200, 6400, 12800\}$下每个数据点的平均延迟对比。

表 4：KDD21 数据集上的单变量时间序列异常检测结果

Data	Type	方法	得分	Time
KDD21	Deep	LSTM	0.460	1842 分钟
		USAD	0.168	54 分钟
		TranAD	0.196	54 分钟
	MP	NormA	**0.500**	24 分钟
		STOMPI	0.360	68 分钟
		SAND	0.388	13534 分钟
		DAMP	**0.512**	384 分钟
	STD	NSigma	0.132	< 0.1 分钟
		OnlineSTL	0.268	4.3 分钟
		OneShotSTL	0.288	7.5 分钟
	STD+MP	NSigma+DAMP	0.324	4.7 分钟
		OnlineSTL+DAMP	0.408	7.2 分钟
		OneShotSTL+DAMP	**0.508**	9.5 分钟

5.4 时间序列异常检测中的方法如何比较?

在本小节中，我们将 STD 方法应用于时间序列异常检测任务，并将其与相应的最先进方法进行比较。

首先，我们评估了 TSB-UAD 基准中的十七个数据集，结果如表 3 所示。表格的每一行显示了一个数据集的平均 VUS-ROC 分数，以及该方法的排名，排名标注在括号中。所有十七个数据集的平均 VUS-ROC 分数、排名和执行时间显示在表格最后三行中。表格显示，OneShot- STL 以 0.713 的平均 VUS-ROC 分数和 4.00 的平均排名，超越了大多数对比方法。表格还表明，没有任何特定方法能够始终优于其他方法。具体而言，USAD、NormA 和 OneShotSTL 分别在 6 个、5 个和 3 个数据集上表现最佳。值得注意的是，我们还观察到，这三种方法在其他数据集上的排名较低。OneShotSTL 的平均排名和 VUS-ROC 分数均高于其他方法，这是因为它在不同数据集上的表现始终稳定。令人惊讶的是，简单的 NSigma 取得了具有竞争力的结果，平均 VUS-ROC 分数为 0.695，优于许多复杂方法，例如 SAND (0.669) 和 DAMP (0.652)。此外，NSigma 也是最快的方法，平均运行时间仅为 2 秒，其次是 OnlineSTL（61 秒）和 OneShotSTL（160 秒）。其他方法则需要数千甚至数万秒才能完成。

接下来，我们评估 KDD21 数据集。表 4 展示了 250 个时间序列的平均准确率，以及训练/测试的总时间。长短期记忆网络的准确率表现良好，达到 0.460，但其运行时间（1842 分钟）远长于其他方法，只有 SAND（13534 分钟）例外。另两种基于深度学习的方法速度更快，但在 KDD21 数据集上的准确率表现不佳。在线时间序列异常检测方法 DAMP 的表现最佳，准确率为 0.512。不过，DAMP 需要 384 分钟（约 6 小时）才能完成对所有数据集的检测。此外，OnlineSTL 和 OneShotSTL 可将 NSigma 的结果（0.132）显著提升至 0.268 和 0.288。然而，它们的准确率仍明显低于 DAMP。这是因为 STD 方法只能

表 5：六个数据集上单变量时间序列预测的平均绝对误差结果，预测长度为 {96,192, 336, 720}（Illness 数据集为 {24, 36,48, 60}）。方法的排名列于括号中，最佳结果以粗体突出显示。

方法		FiLM	FEDformer	Informer	NBEATS	DeepAR	AutoArima	OnlineSTL	OneShotSTL
ETTm2	96	**0.189 (1)**	**0.189 (1)**	0.225 (5)	0.201 (3)	0.942 (8)	0.379 (7)	0.375 (6)	0.211 (4)
	192	**0.233 (1)**	0.245 (3)	0.283 (5)	0.248 (4)	0.993 (8)	0.419 (7)	0.391 (6)	0.244 (2)
	336	0.274 (2)	0.279 (4)	0.336 (5)	0.274 (2)	0.442（8）	0.437（7）	0.419（6）	**0.273（1）**
	720	0.323（2）	0.325（3）	0.435 (4)	0.468 (6)	1.028 (8)	0.475 (7)	0.452 (5)	**0.321 (1)**
电力	96	**0.247 (1)**	0.370 (3)	0.538 (5)	0.387 (4)	0.641 (6)	0.898 (7)	1.031 (8)	0.331 (2)
	192	**0.258 (1)**	0.386 (3)	0.558 (5)	0.458（4）	0.698（6）	0.925（7）	1.013（8）	0.355（2）
	336	**0.283（1）**	0.431（3）	0.613（5）	0.494（4）	0.747（6）	1.013（7）	1.037（8）	0.389 (2)
	720	**0.341 (1)**	0.484 (4)	0.682 (5)	0.483 (3)	0.796 (6)	1.096 (8)	1.070 (7)	0.444 (2)
Exchange	96	0.259 (4)	0.284 (5)	0.615 (8)	0.358 (6)	0.597 (7)	**0.241 (1)**	**0.241 (1)**	0.252 (3)
	192	0.352（4）	0.420（7）	0.912（8）	0.387（5）	0.419（6）	**0.321（1）**	0.327 (2)	0.334 (3)
	336	0.461 (4)	0.511 (5)	0.984 (8)	0.652 (6)	0.920 (7)	**0.432 (1)**	0.433 (2)	0.436 (3)
	720	0.708 (4)	0.832 (6)	1.072 (7)	1.114 (8)	0.710 (5)	0.681 (3)	0.626 (2)	**0.624 (1)**
Traffic	96	0.215 (3)	0.263 (4)	0.353 (5)	0.206 (2)	0.614 (6)	1.036 (7)	1.465 (8)	**0.181 (1)**
	192	0.199 (2)	0.265 (4)	0.376 (5)	0.211（3）	0.992 (6)	1.101 (7)	1.461（8）	**0.181 (1)**
	336	0.212 (3)	0.266 (4)	0.387 (5)	0.209 (2)	1.105 (6)	1.138 (7)	1.470 (8)	**0.182（1）**
	720	0.252（3）	0.286（4）	0.436（5）	0.243（2）	1.166（6）	1.213 (7)	1.461 (8)	**0.199 (1)**
天气	96	0.026 (4)	0.046 (7)	0.044 (6)	**0.022 (1)**	0.059 (8)	0.025 (3)	0.031 (5)	0.024 (2)
	192	0.029 (4)	0.059 (8)	0.040 (6)	0.026 (2)	0.047 (7)	0.028 (3)	0.032 (5)	**0.025 (1)**
	336	0.030 (2)	0.050 (8)	0.049 (7)	0.031 (3)	0.035 (6)	0.031 (3)	0.033 (5)	**0.026 (1)**
	720	0.037 (5)	0.091 (8)	0.042 (7)	0.033 (2)	0.033 (2)	0.035 (4)	0.037 (6)	**0.030 (1)**
Illness	24	0.538 (2)	0.629 (3)	2.050 (8)	**0.467 (1)**	0.726 (5)	0.821 (6)	0.917 (7)	0.682 (4)
	36	**0.481 (1)**	0.604 (3)	1.916 (8)	0.519 (2)	0.810 (5)	0.928 (7)	0.840 (6)	0.757 (4)
	48	**0.584 (1)**	0.699 (3)	1.846 (8)	0.678 (2)	0.861 (6)	0.941 (7)	0.843 (5)	0.773 (4)
	60	**0.644 (1)**	0.828 (4)	2.057 (8)	0.773 (2)	0.873 (5)	0.913 (6)	0.963 (7)	0.826 (3)
平均绝对误差		**0.308 (1)**	0.368 (3)	0.702 (7)	0.373 (4)	0.677 (6)	0.647 (5)	0.707 (8)	0.337 (2)
平均排名		2.37 (2)	4.45 (4)	6.16 (7)	3.29 (3)	6.20 (8)	5.41 (5)	5.79 (6)	**2.08 (1)**
平均时间（秒）		7860 s (7)	2110 s (5)	2003 s (4)	790 秒 (3)	4709 秒（6）	8779秒（8）	**0.3 秒 (1)**	**0.3 秒（1）**

处理季节性模式，而 KDD21 数据集包含非季节性模式。不过，OnlineSTL 和 OneShotSTL 的速度远快于 DAMP。因此，我们进一步将它们与 DAMP 结合，将其用作预过滤步骤。对于每个时间序列，我们通过 STD 方法筛选出排名靠前的 1% 测试点，并将其交由 DAMP 检测。通过这种方式，OneShotSTL 可将 DAMP 的计算时间从 384 分钟缩短至 9.5 分钟（速度提升超过 40 倍），而准确率几乎没有损失（从 0.512 降至 0.508）。同时可以观察到，在作为 DAMP 的筛选方法时，OneShotSTL 的表现优于 OnlineSTL（0.408）。

基于上述结果，我们得出结论：在时间序列异常检测任务中，OneShotSTL 在大多数情况下都能提供更高的检测准确率，表现优于 OnlineSTL。此外，OneShotSTL 和基于矩阵剖面的方法在 TSB-UAD基准的不同数据集上均表现良好，但 OneShotSTL 的速度快 10 到 100 倍。

5.5 这些方法在时间序列预测中的表现如何?

在本小节中，我们将 STD 方法应用于时间序列预测任务，并将其与相应的最先进方法进行比较。

表 5 展示了 8 种对比方法在 6 个基准数据集上的表现，预测长度为 {96,192, 336, 720}（Illness 为 {24, 36,48, 60}）。表格的每一行展示了预测误差（以平均绝对误差衡量），各方法在每种设置下的排名列于括号中，最佳结果以粗体突出显示。表格最后 3 行分别列出所有设置下的平均平均绝对误差、排名和执行时间（包括训练和测试时间）。首先，从表中可以看出在预测误差方面，FiLM 和 OneShotSTL 表现最佳，平均平均绝对误差分别为 0.308 和 0.337，其次是 FEDformer（0.368）和 NBEATS（0.373）。从平均排名来看，OneShotSTL 是表现最好的方法，平均排名为 2.08，其次是 FiLM（2.37）、NBEATS（3.29）和 FEDformer（4.45）。具体而言，OneShotSTL 在季节性较强的数据集（ETTm2、电力、Traffic 和天气）上表现更好，而在季节性较弱或没有季节性的数据集 Exchange 和 Illness 上预测效果相对较差。此外，STD 方法在速度上相较于对比 TSF 方法具有显著优势。在普通机器上使用单个 CPU核心时，OnlineSTL 和 OneShotSTL 的平均运行时间不到 1 秒；而基于深度学习的方法使用 NVIDIA V100 GPU 的处理时间为数百至数千秒（慢 1000 倍以上）。

总而言之，OneShotSTL 的预测准确率在所评估的数据集上与最先进的深度学习方法 FiLM 相当。同时，其运行时间比非STD方法快1000多倍。

5.6 OneShotSTL 消融研究

在本小节中，我们针对 OneShotSTL 的三个输入参数进行消融研究：周期长度 T、最大季节性偏移长度 H 和迭代次数 I。

首先，我们研究估计 T时产生的误差如何影响最终性能。请记住，T是在 OneShotSTL 的初始化阶段自动发现的。因此，我们使用 $\Delta T =$的不同取值$\{0, 5, 10, 15, 20\}$，并应用 $T \leftarrow T + \Delta T$作为

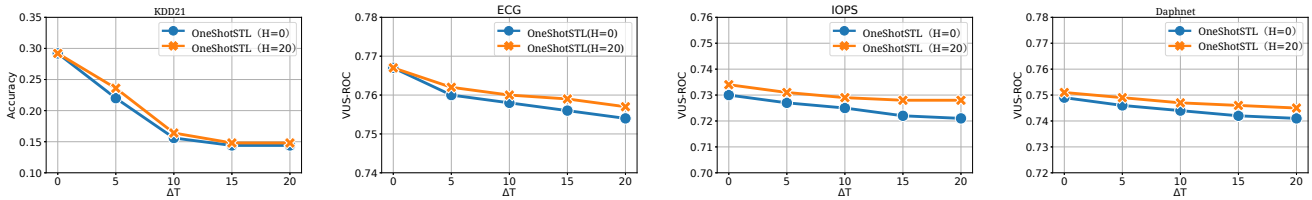


图8：OneShotSTL 在时间序列异常检测任务中超参数 T和 H的消融研究。我们将噪声以 $\Delta T \in \{0, 5, 10, 15, 20\}$的幅度添加到 T中。Y 轴表示准确率和 VUS-ROC 分数（越高越好）。

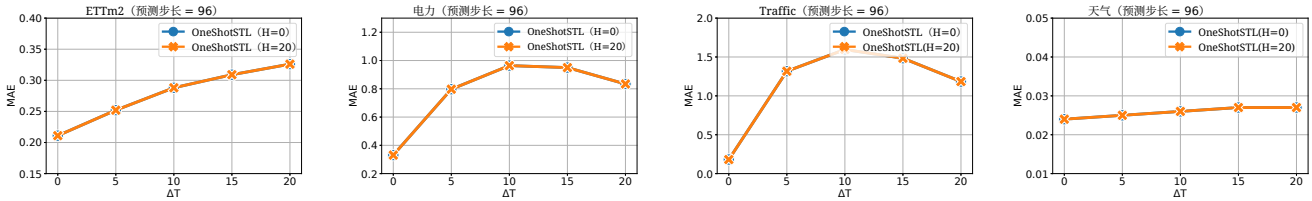


图9：OneShotSTL 在时间序列预测任务中超参数 T和 H的消融研究。我们将噪声以 $\Delta T \in \{0, 5, 10, 15, 20\}$的幅度添加到 T中。Y 轴表示平均绝对误差（越低越好）。

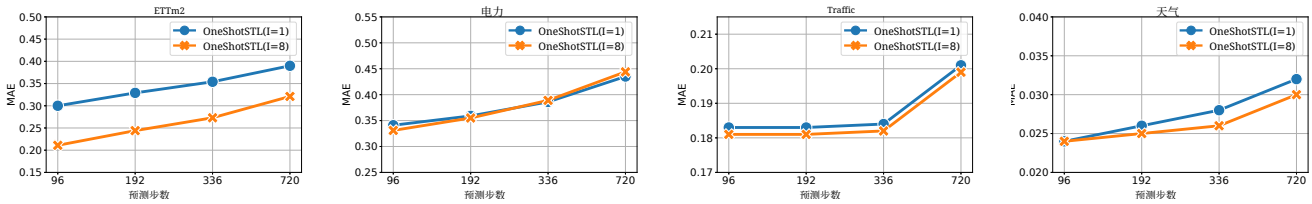


图10：消融研究 o f OneShotSTL 在时间序列预测任务中的超参数 I。Y 轴表示平均绝对误差（越低越好）。

周期长度参数。同时，对于每个数据集，我们分别将 OneShotSTL 与季节性偏移窗口 $H = 0$ 和$H = 20$ 进行比较。图8展示了 OneShotSTL 在四个时间序列异常检测数据集上采用不同 ΔT 时的性能。如图8所示，在所有情况下，OneShotSTL 在 $H = 20$ 时的表现都更好。此外，对于所有四个数据集，随着 ΔT 增加，OneShotSTL 的性能都会下降。然而，在 KDD21 上，$\Delta T = 5$之后 OneShotSTL 的得分立即快速下降，而在其他三个数据集上，其性能则缓慢下降。此外，我们以相同的方式评估了 T 对时间序列预测任务的影响，结果如图9所示。从图中可以看出，在 $H = 0$ 和 $H = 20$两种情况下，OneShotSTL 在所有数据集上的预测误差都显著增加。这可能是因为我们只能通过在 $[T - H, T + H]$内搜索最优值来校正 T_c。这种方法适用于时间序列异常检测任务，因为我们只关注新到达的数据点。然而，对于时间序列预测任务，由于我们无法校正未来的 T_c，因此只能直接使用 T来预测未来数据（第4节中的$\hat{y}_{t+i} = \tau_{t-1} + v_{t\%T+i}$）。

接下来，我们评估输入参数 I 对 OneShot- STL 的影响。图10展示了在相同 $H = 20$ 下，使用不同预测步数时，OneShotSTL 在 $I = 8$ 和 $I = 1$ 条件下的平均绝对误差。由于篇幅有限，我们省略了 I 在时间序列异常检测任务上的消融研究结果；我们在其中得到了类似的观测结果。从图10可以看出，使用 $I = 8$ 时，OneShotSTL 更常产生比使用 $I = 1$时更低的预测误差。具体而言，在 ETTm2 数据集上，差距较大；而在其他三个数据集上，使用 $I = 8$时仅观察到小幅提升。总体而言，设置更大的 I 会提高分解质量，但会牺牲速度。

6 结论

在本文中，我们提出了一种准确且高效的在线时序分解算法 OneShotSTL，该算法能够以 $O(1)$ 的更新复杂度对时间序列进行在线分解。它可以显著降低现有方法的处理时间复杂度；这些方法通常需要 $O(T)$，例如 OnlineSTL [28]。因此，使用 OneShotSTL 对单个时间点进行在线更新的时间比批量季节-趋势分解方法快 1,000 多倍，同时准确率可与最佳同类方法相媲美，例如 RobustSTL [40]。在真实世界基准数据集上针对下游任务（例如时间序列异常检测和时间序列预测问题）开展的大量实验表明，OneShotSTL 的速度比当前最先进的时间序列异常检测和时间序列预测方法快 10 倍至 1000 多倍，同时仍能提供相当甚至更好的结果。

该方法也存在局限性。所提出的 OneShotSTL 假设底层的 $T_c \in [T - H, T + H]$ 由两个常数 T 和 H 决定。当 T_c 在多个周期内发生剧烈变化时，这一假设可能不成立。在线自适应检测季节周期 T_c 对在线季节-趋势分解问题至关重要。此外，在实际应用中，处理缺失数据点或不规则采样时间序列也至关重要，而目前大多数 STD 方法都无法处理这些情况。未来，我们计划探索 OneShotSTL 在时间序列预测任务中的潜力，因为将分解趋势和季节信号与其他技术相结合，或可助力长序列预测。此外，将 OneShotSTL 与基于矩阵轮廓的方法结合，用于在线时间序列异常检测任务，也是一个有趣的研究方向。所有这些都指向了在线季节-趋势分解问题未来的研究方向。

参考文献

[1] [无日期]。阿里云监控。https://www.alibabacloud.com/product/cloud-monitor。最后访问时间：2022-09-21。[2] [无日期]。AWS CloudWatch。https://aws.amazon.com/cloudwatch/。最后访问时间：2022-09-21。[3] [无日期]。查找 TSB-UAD 的长度函数。https://github.com/TheDatumOrg/TSB-UAD/blob/main/TSB_UAD/utils/slidingWindows.py。最后访问时间：2022-12-05。[4] [无日期]。如何开始使用智能运维。https://www.gartner.com/smarterwithgartner/how-to-get-started-with-aiops。最后访问时间：2022-09-21。[5] [无日期]。Informer 数据集。https://github.com/zhouhaoyi/Informer2020。最后访问时间：2022-12-05。[6] [无日期]。stl-decomp-4j 软件包中的 STL Java 实现。https://github.com/ServiceNow/stl-decomp-4j。最后访问时间：2022-09-21。[7] [无日期]。SREWorks 中 FastRobustSTL 的 Python 实现。https://github. com/alibaba/SREWorks/。最后访问时间：2022-09-21。[8] [无日期]。Pytorch-forecasting 软件包。https://pytorch-forecasting.readthedocs.io/en/stable/index.html。最后访问时间：2022-09-21。[9] [无日期]。Statsforecast 软件包。https://nixtla.github.io/statsforecast/。最后访问时间：2022-09-21。[10] [无日期]。补充材料。https://github.com/xiao-he/OneShotSTL/。最后访问时间：2023-02-17。[11] Adebiyi A. Ariyo、Adewumi O. Adewumi 和 Charles K. Ayo。2014。使用 ARIMA 模型进行股票价格预测。载于《2014 年第16届UKSim-AMSS计算机建模与仿真国际会议论文集》（UKSIM '14）。IEEE计算机学会，美国，106 112 10 1109 2014 67 –。[12] https://doi.org/。/UKSim.。Julien Audibert、Pietro Michiardi、Frédéric Guyard、Sébastien Marti 和 Maria A. Zuluaga。2020。USAD：多变量时间序列上的无监督异常检测。载于 KDD '20：第26届 ACM SIGKDD 知识发现与数据挖掘会议，线上会议，美国加利福尼亚州，2020年8月23日至27日，Rajesh Gupta、Yan Liu、Jiliang Tang 和 B. Aditya Prakash（编）。ACM，3395–3404。https: //doi.org/10 1145 3394486 3403392 。/。Amir Beck。。关于凸规划中交替最小化的收敛性及其在迭代重加权最小二乘法和分解方案中的应用。《SIAM 优化杂志》25卷第1期（2015年），185–209页。[13] 2015[14] Paul Boniol、Michele Linardi、Federico Roncallo、Themis Palpanas、Mohammed Meftah 和 Emmanuel Remy。2021。大规模数据序列中的无监督可扩展子序列异常检测。VLDB期刊 30卷第6期（2021年），909–931页。https: //doi.org/10 1007 00778 021 00655 8 。/s - - - Paul Boniol、John Paparrizos、Themis Palpanas 和 Michael J. Franklin。2021。SAND：流式子序列异常检测。《VLDB论文集》14卷第10期（2021 1717 1729 10 14778 3467861 3467863 ），–。https://doi.org/。/。[15] [16] Robert B. Cleveland、William S. Cleveland、Jean E. McRae 和 Irma Terpenning。1990。STL：基于局部加权回归的季节-趋势分解方法（附讨论）。《官方统计学杂志》6卷（1990年），3–73页。[17] Dau、Hoang Anh、Keogh、Eamonn、Kamgar、Kaveh、Yeh、Chin-Chia Michael、Zhu、Yan、Gharghabi、Shaghayegh、Ratanamahatana、Chotirat Ann、Yanping、Hu、Bing、Begum、Nurjahan、Bagnall、Anthony、Mueen、Abdullah、Batista、Gustavo 和 Hexagon-ML。2018。UCR时间序列档案库。https://www.cs.ucr.edu/-eamonn/time_series_data_2018/。[18] Alysha M De Livera、Rob J Hyndman 和 Ralph D Snyder。2011。使用指数平滑法预测具有复杂季节性模式的时间序列。《美国统计学会杂志》106卷第496期（2011年），1513–1527页。[19] Alexander Dokumentov、Rob J Hyndman 等。2015。STR：一种基于回归的季节-趋势分解方法。莫纳什大学：澳大利亚墨尔本（2015年），23页。[20] Francesco Galati、Julien Audibert 和 Maria A. Zuluaga。[无日期]。USAD 的 Python 实现。https://github.com/manigalati/usad。最后访问时间：2022-09-21。[21] Sam George。2019。IoT Signals报告：通过解决技能短缺、复杂度和安全问题，释放物联网的潜力。https://blogs.microsoft.com/blog/2019/07/30/。最后访问时间：2022-09-21。[22] P. J. Green。1984。用于最大似然估计的迭代重加权最小二乘法，以及一些稳健和抗差替代方法。《英国皇家统计学会杂志》B辑（方法论）46卷第2期（1984年），149–192页。2345305 http://www. jstor.org/stable/[23] Xiao He、Francesco Alesiani 和 Ammar Shaker。2019。海量任务上的高效且可扩展的多任务回归。载于第三十三届AAAI人工智能会议（AAAI 2019），美国夏威夷州檀香山，2019年1月27日至2月1日。AAAI出版社，3763–3770。https://doi.org/10.1609/aaai.v33i01.33013763[24] Siwon Kim、Kukjin Choi、Hyun-Soo Choi、Byunghan Lee 和 Sungroh Yoon。2022。迈向时间序列异常检测的严格评估。载于第三十六届AAAI人工智能会议、第三十四届人工智能创新应用大会（IAAI 2022）、第十二届人工智能教育进展研讨会（EAAI 2022），线上会议，2022年2月22日至3月1日。AAAI出版社，7194–7201。https: //ojs.aaai.org/index.php/AAAI/article/view/20650 [25] Seung-Jean Kim、Kwangmoo Koh、Stephen Boyd 和 Dimitry Gorinevsky。2009。\ell_1 趋势滤波。SIAM 评论 51，2（2009），339–360。[26] Sean M. Law。2019。STUMPY：用于时间序列数据挖掘的强大且可扩展的 Python 库。《开放源代码软件期刊》4，39（2019），1504。[27] Yue Lu、Renjie Wu、Abdullah Mueen、Maria A. Zuluaga 和 Eamonn J. Keogh。2022。矩阵剖面 XXIV：将时间序列异常检测扩展至数万亿数据点和超快速到达的数据流。载于 KDD '22：第 28 届 ACM SIGKDD 知识发现与数据挖掘会议，华盛顿特区，美国，2022 年 8 月 14 日至 18 日，Aidong Zhang 和 Huzefa Rangwala（编）。ACM，1173 1182 10 1145 3534678 3539271 –。https://doi.org/。/。Abhinav Mishra、Ram Sriharsha 和 Sichen Zhong。2022。OnlineSTL：将时间序列分解扩展 100 倍。Proc. VLDB Endow. 15，7（2022），1417–1425。https://www.vldb.org/pvldb/vol15/p1417-mishra.pdf [28] [29] Boris N. Oreshkin、Dmitri Carpov、Nicolas Chapados 和 Yoshua Bengio。2020。N-BEATS：用于可解释时间序列预测的神经基展开分析。载于第 8 届国际学习表示会议，ICLR 2020，埃塞俄比亚亚的斯亚贝巴，2020 年 4 月 26 日至 30 日。OpenReview.net。https://openreview.net/ forum?id=r1ecqn4YwB[30] John Paparrizos、Paul Boniol、Themis Palpanas、Ruey S Tsay、Aaron Elmore 和 Michael J Franklin。2022。曲面下体积：一种用于时间序列异常检测的新型准确性评估指标。《VLDB 基金会会议录》15，11（2022），2774–2787。[31] John Paparrizos、Yuhao Kang、Paul Boniol、Ruey S. Tsay、Themis Palpanas 和 Michael J. Franklin。2022。TSB-UAD：单变量时间序列异常检测的端到端基准测试套件。Proc. VLDB Endow. 15，8（2022），1697–1711。https://www.vldb.org/pvldb/vol15/p1697-paparrizos.pdf [32] Daehyung Park、Yuuna Hoshi 和 Charles C. Kemp。2018。使用基于长短期记忆网络的变分自编码器进行机器人辅助喂食的多模态异常检测器。IEEE Robotics Autom. Lett. 3，2（2018），1544–1551。https: //doi.org/10 1109 2018 2801475 . /LRA.。Tom Puech、Matthieu Boussard、Anthony D'Amato 和 Gaëtan Millerand。2019。时间序列的全自动周期性检测。载于时序数据高级分析与学习——第 4 届 ECML PKDD 研讨会，AALTD 2019，德国维尔茨堡，2019 年 9 月 20 日，修订精选论文（计算机科学讲义），Vincent Lemaire、Simon Malinowski、Anthony J. Bagnall、Alexis Bondu、Thomas Guyet 和 Romain Tavenard（编），第 11986 卷。Springer，43–54。https: //doi.org/10 1007 978 3 030 39098 3_4 . / - - - - David Salinas、Valentin Flunkert、Jan Gasthaus 和 Tim Januschowski。2020。DeepAR：使用自回归循环网络进行概率预测。《国际预测学期刊》36，3（2020），1181–1191。[33] [34] Sean J. Taylor 和 Benjamin Letham。2017。大规模模预测。PeerJ Prepr. 5（2017 3190 10 7287 3190 1），e。https://doi.org/。/peerj.preprints. v Maximilian Toller、Tiago Santos 和 Roman Kern。SAZED：时间序列数据中无参数、领域无关的季节长度估计。《数据挖掘与知识发现》33，6（2019），1775–1798。https://doi.org/10.1007/s10618-019-00645-z [35] [36] Shreshth Tuli。[n.d.]。TranAD 的 Python 实现。https://github.com/ imperial-qore/TranAD。最后访问于 2022-09-21。[37] [38] Shreshth Tuli、Giuliano Casale 和 Nicholas R. Jennings。2022。TranAD：用于多变量时间序列数据异常检测的深度 Transformer 网络。Proc. VLDB Endow. 15，6（2022），1201–1214。https://www.vldb.org/pvldb/vol15/ p1201-tuli.pdf [39] Michail Vlachos、Philip S. Yu 和 Vittorio Castelli。2005。关于周期性检测与结构周期相似性。载于《2005 年 SIAM 国际数据挖掘会议论文集》，SDM 2005，美国加利福尼亚州纽波特海滩，2005 年 4 月 21 日至 23 日，Hillol Kargupta、Jaideep Srivastava、Chandrika Kamath 和 Arnold Goodman（编）。SIAM，449–460。https://doi.org/10.1137/1.9781611972757.40[40] 文青松、Jingkun Gao、Xiaomin Song、孙亮、Huan Xu 和 Shenghuo Zhu。2018。RobustSTL：一种适用于长时间序列的鲁棒季节-趋势分解算法。CoRR abs/1812.01767（2018）。arXiv:1812.01767 http: //arxiv.org/abs/1812 01767 。文青松、Kai He、孙亮、Yingying Zhang、Min Ke 和 Huan Xu。2021。RobustPeriod：用于多周期性检测的鲁棒时频挖掘。载于 SIGMOD '21：数据管理国际会议，线上会议，中国，2021 年 6 月 20 日至 25 日，Guoliang Li、Zhanhuai Li、Stratos Idreos 和 Divesh Srivastava（编）。ACM，2328 2337 10 1145 3448016 3452779 –。[41] [42] https://doi.org/。/。文青松、Zhe Zhang、Yan Li 和孙亮。Fast RobustSTL：针对复杂模式时间序列的高效鲁棒季节-趋势分解。载于 KDD '20：第 26 届 ACM SIGKDD 知识发现与数据挖掘会议，线上会议，美国加利福尼亚州，2020 年 8 月 23 日至 27 日，Rajesh Gupta、Yan Liu、Jiliang Tang 和 B. Aditya Prakash（编）。ACM，2203–2213。https://doi.org/10 1145 3394486 3403271 . /。Renjie Wu、Abdullah Mueen、Maria A. Zuluaga 和 Eamonn J. Keogh。[n.d.]。DAMP 的 Matlab 实现。https://sites.google.com/view/discord-aware- matrix-profile。最后访问于 2022-09-21。[43] [44] Chin-Chia Michael Yeh、Yan Zhu、Liudmila Ulanova、Nurjahan Begum、Yifei Ding、Hoang Anh Dau、Zachary Zimmerman、Diego Furtado Silva、Abdullah Mueen 和 Eamonn J. Keogh。2018。时间序列连接、模式、异常片段和形状片段：

一种利用矩阵剖面实现统一视角的方法。数据挖掘与知识发现32, 1 (2018), 83–123. https://doi.org/10.1007/s10618-017-0519-9[45] 周浩毅、张尚航、彭杰琪、张帅、李建新、熊辉和张万才。2021。Informer：超越高效Transformer，实现长序列时间序列预测。载于*AAAI* 人工智能会议论文集，第35卷。11106–11115。[46] 周天、马梓清、王雪、文青松、孙亮、姚涛、尹卧涛和金荣。2022。FiLM：频率增强的勒让德记忆模型。长期时间序列预测。*CoRR* abs/2205.08897（2022）。https: //doi.org/10 48550 2205 08897 2205 08897 。/arXiv。arXiv:。周天、马梓清、文[47]青松、王雪、孙亮和金荣。2022。FEDformer：用于长期时间序列预测的频率增强分解式Transformer。载于国际机器学习会议，*ICML 2022*，*2022*年*7*月*17*日至*23*日，巴尔的摩，马里兰州，美国（机器学习研究论文集），卡玛莉卡·乔杜里、斯特凡妮·耶格尔卡、宋乐、查包·塞佩什瓦里、牛刚和西万·萨巴托（编），第162卷。机器学习研究论文集，27268–27286。