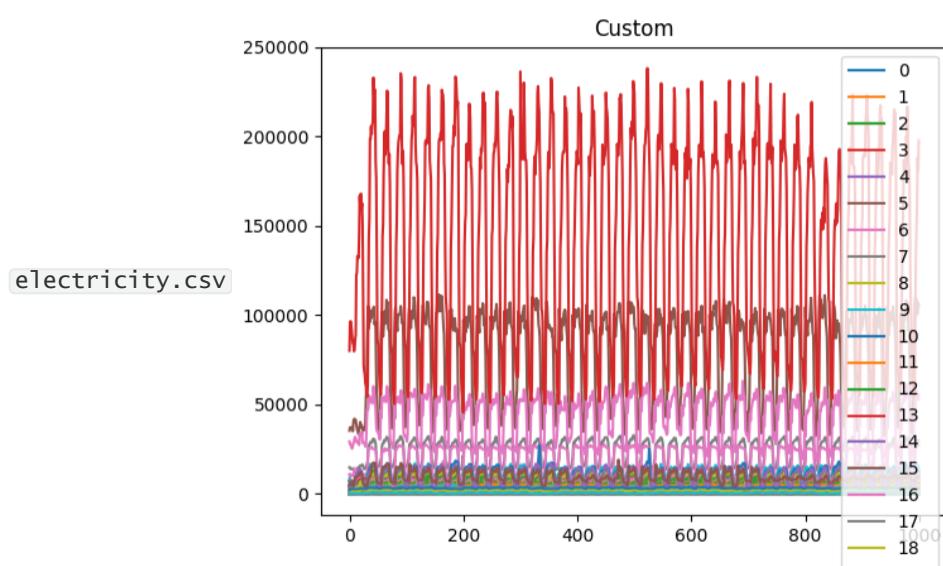
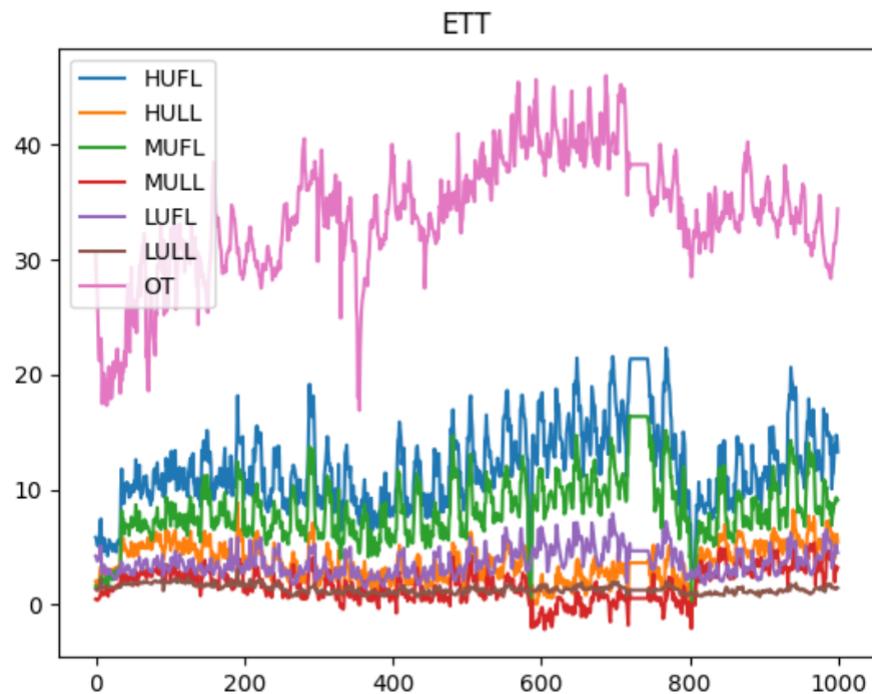


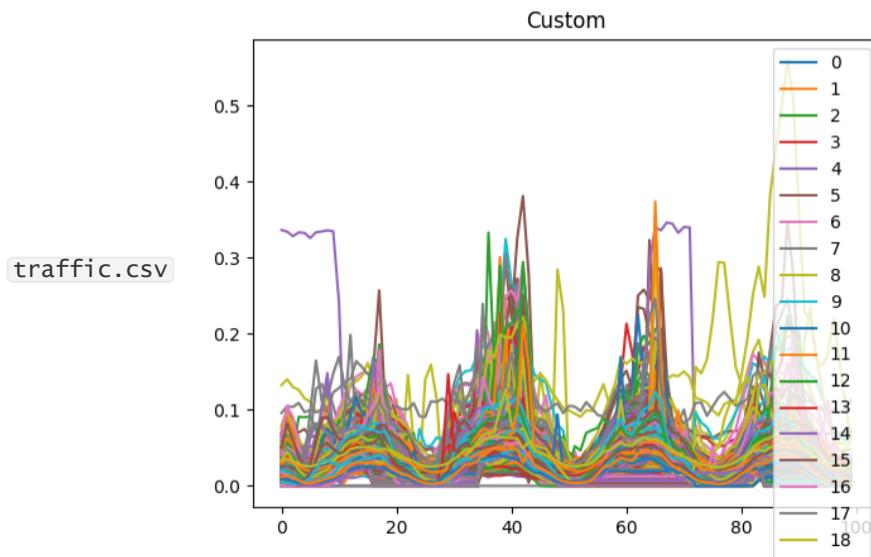
hw1实验报告

201300086史浩男

一、数据集

可视化1000个数据点：





二、转换

代码中，如果同一个transform被多次使用，注意是否更新存储的值，可以添加一个update参数用于记录

```
def transform(self, data, update=False):
    if update:
        self.mean = data.mean()
        self.std = data.std()
    # 先检测数据的标准差是否为0
    if self.std == 0:
        return data - self.mean
    # 将数据标准化到0, 1之间
    norm_data = (data - self.mean) / self.std
    return norm_data
```

仅在训练集上update，test_X则不update：

```
def train(self):
    train_X = self.dataset.train_data
    t_X = self.transform.transform(train_X, update=True)
    self.model.fit(t_X)

    test_X = self.transform.transform(test_X)
    fore = self.model.forecast(test_X, pred_len=pred_len)
    # test_Y = self.transform.transform(test_Y)
    fore = self.transform.inverse_transform(fore)
```

数学公式

1. 归一化 (Normalization)

- 变换 (Transform):

- 如果 `max-min` 不为0，变换公式为:
$$\text{norm_data} = \frac{\text{data} - \text{min}}{\text{max} - \text{min}}$$
- 如果 `max-min` 为0，则返回 `data - mean`。

- 逆变换 (Inverse Transform):

- 如果 `data` 的极差不为0，逆变换公式为:
$$\text{inverse_data} = \text{data} \times (\text{max} - \text{min}) + \text{min}$$
- 如果 `data` 的极差为0，则返回 `data * mean`。

2. 标准化 (Standardization)

- 变换 (Transform):

- 如果 `std` 不为0，变换公式为:
$$\text{norm_data} = \frac{\text{data} - \text{mean}}{\text{std}}$$
- 如果 `std` 为0，则返回 `data - mean`。

- **逆变换 (Inverse Transform):**

- 如果 `data` 的标准差不为0, 逆变换公式为: $\text{inverse_data} = \text{data} \times \text{std} + \text{mean}$
- 如果 `data` 的标准差为0, 则返回 `data + mean`。

3. 均值归一化 (Mean Normalization)

- **变换 (Transform):**

- 如果 `max-min` 不为0, 变换公式为: $\text{norm_data} = \frac{\text{data}-\text{mean}}{\text{max}-\text{min}}$
- 如果 `max-min` 为0, 则返回 `data - mean`。

- **逆变换 (Inverse Transform):**

- 如果 `data` 的极差不为0, 逆变换公式为: $\text{inverse_data} = \text{data} \times (\text{max} - \text{min}) + \text{mean}$
- 如果 `data` 的极差为0, 则返回 `data * mean`。

4. Box-Cox 变换 (BoxCox Transform)

- **变换 (Transform):**

- 数据首先转化为正数, 然后应用 Box-Cox 变换, 公式为: $\text{norm_data} = \text{boxcox}(\text{data}, \text{lam})$

- **逆变换 (Inverse Transform):**

- 应用 Box-Cox 的逆变换, 公式为: $\text{inverse_data} = \text{inv_boxcox}(\text{data}, \text{lam})$

其他Scaler

除了手动实现的这些，sklearn中还提供了可以直接调用的常用Scaler：

1. StandardScaler:

对于每个特征 x , StandardScaler 首先计算特征的均值 μ 和标准差 σ , 然后应用以下公式：

$$x_{\text{scaled}} = \frac{x - \mu}{\sigma}$$

2. MinMaxScaler:

MinMaxScaler 将每个特征的值缩放到指定的范围内（通常是 0 到 1）。对于每个特征 x , 应用以下公式：

$$x_{\text{scaled}} = \frac{x - x_{\min}}{x_{\max} - x_{\min}}$$

其中, x_{\min} 和 x_{\max} 分别是特征 x 的最小值和最大值。

3. RobustScaler:

RobustScaler 使用中位数和四分位数范围对数据进行缩放，以减少离群值的影响。对于每个特征 x , 应用以下公式：

$$x_{\text{scaled}} = \frac{x - Q_1(x)}{Q_3(x) - Q_1(x)}$$

其中, $Q_1(x)$ 和 $Q_3(x)$ 分别是特征 x 的第一四分位数和第三四分位数。

4. MaxAbsScaler:

MaxAbsScaler 通过除以每个特征的最大绝对值来缩放数据。对于每个特征 x , 应用以下公式：

$$x_{\text{scaled}} = \frac{x}{\max(|x|)}$$

5. Normalizer:

Normalizer 对单个样本的特征向量进行缩放，使其具有单位范数（长度）。这通常用于文本分类和聚类。对于样本 x , 应用以下公式：

$$x_{\text{scaled}} = \frac{x}{\|x\|_p}$$

其中, $\|x\|_p$ 是 p 范数, 常见的 p 包括 1 (曼哈顿距离)、2 (欧几里得距离) 等。

- 如果数据包含许多异常值, 使用 RobustScaler 可能更合适。
- 当处理稀疏数据时, 应谨慎选择 Scaler。例如, StandardScaler 和 MinMaxScaler 可能会改变数据的稀疏性, 而 MaxAbsScaler 则保持数据的稀疏结构。

三、指标

1. 均方误差 (MSE - Mean Squared Error):

$$\text{MSE}(\text{predict}, \text{target}) = \text{mean}((\text{target} - \text{predict})^2)$$

2. 平均绝对误差 (MAE - Mean Absolute Error):

$$\text{MAE}(\text{predict}, \text{target}) = \text{mean}(\text{abs}(\text{target} - \text{predict}))$$

3. 平均绝对百分比误差 (MAPE - Mean Absolute Percentage Error):

$$\text{MAPE}(\text{predict}, \text{target}) = \text{nanmean}\left(\text{abs}\left(\frac{\text{target}_{\text{nonzero}} - \text{predict}_{\text{nonzero}}}{\text{target}_{\text{nonzero}}}\right)\right)$$

为0的情况需要特殊处理

4. 对称平均绝对百分比误差 (sMAPE - Symmetric Mean Absolute Percentage Error):

$$\text{sMAPE}(\text{predict}, \text{target}) = \text{mean}\left(\frac{2 \times \text{abs}(\text{target} - \text{predict})}{\text{abs}(\text{target}) + \text{abs}(\text{predict})}\right)$$

为0的情况需要特殊处理

5. 平均绝对误差比例 (MASE - Mean Absolute Scaled Error):

$$\text{MASE} = \frac{1}{H} \sum_{i=1}^H \frac{|y_{T+i} - \hat{y}_{T+i}|}{\frac{1}{T+H-m} \sum_{j=m+1}^{T+H} |y_j - y_{j-m}|}$$

```
def naive_forecast(y:np.array, season:int=1):
    "naive forecast: season-ahead step forecast, shift by season step ahead"
    return y[:-season]
2 usages new *
def mase(predict, target, season=24):
    return mae(target, predict) / mae(target[season:], naive_forecast(target, season))
```

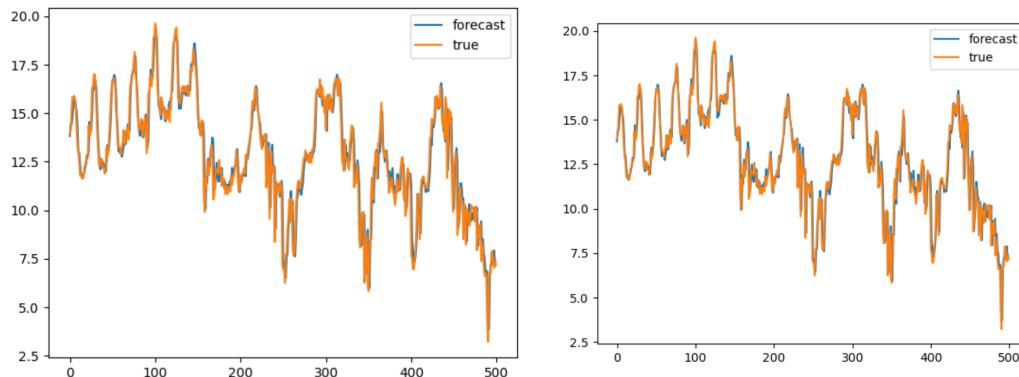
切比雪夫距离：全部维度上的距离中最大值

四、自回归&指数平滑

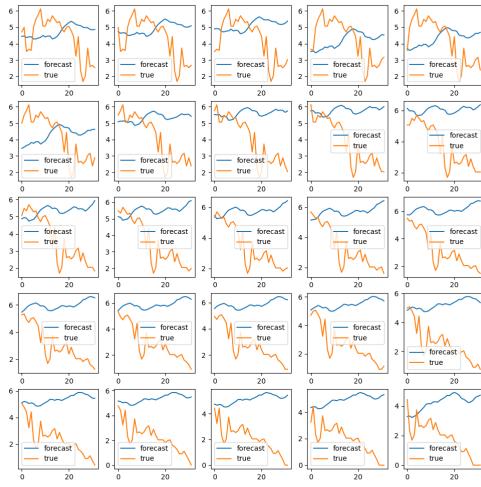
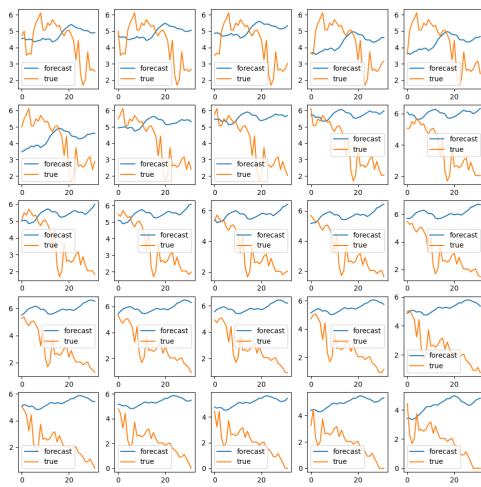
1、ETTh1.csv

两种方法预测效果图：

(取每个pred_len=32的结果向量的第一维的值，拼接后作为forecast值画图)



(连续取25个pred_len=32的结果与真实的长度为32的值对比)

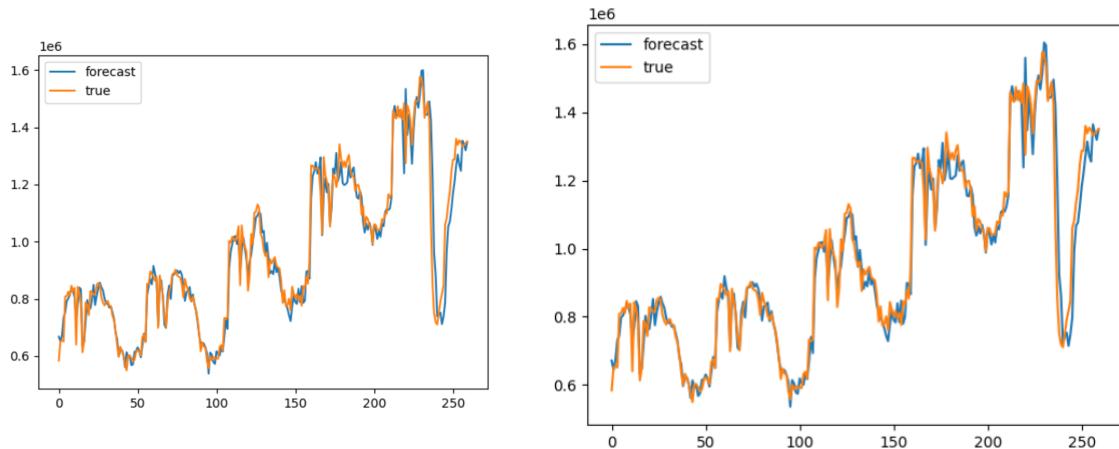


模型	转换	MSE	MAE	MAPE	SMAPE	MASE
AR	无&归一化&标准化	3.72368	1.42236	0.31643	0.27864	0.82886
EMA	无&归一化&标准化	3.72124	1.42131	0.31620	0.27859	0.82825
AR (MIMO)	无&归一化&标准化	3.69678	1.41219	0.31612	0.277	0.82293
AR (MIMO)	Boxcox	3.70217	1.41457	0.31571	0.2772	0.82432
EMA (MIMO)	无&归一化&标准化	3.69542	1.41189	0.31585	0.27708	0.82276
EMA (MIMO)	Boxcox	3.70251	1.4146	0.31552	0.27732	0.82433

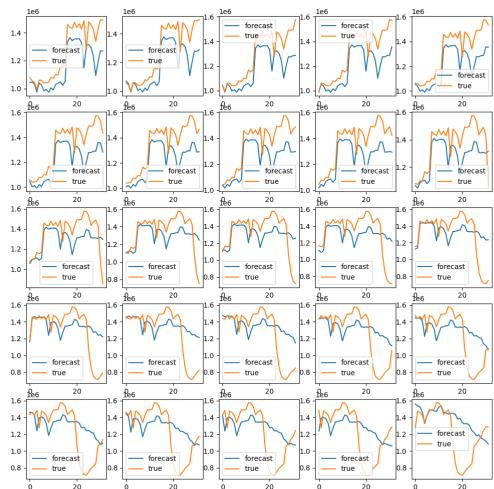
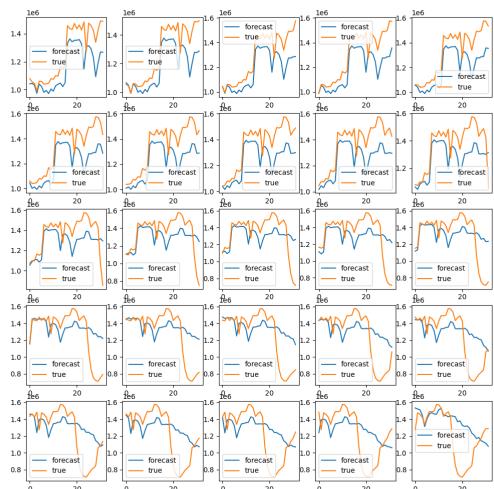
2、national_illness.csv

三种方法预测效果图：

(取每个pred_len=32的结果向量的第一维的值，拼接后作为forecast值画图)



(连续取25个pred_len=32的结果与真实的长度为32的值对比)



模型	参数	转换	MSE	MAE	MAPE	SMAPE	MASE
AR	无	无&归一化&标准化	27439193740	108850	0.103495	0.105101	0.60825
AR (MIMO)	无	无&归一化&标准化	28285657974	112030	0.10616	0.10909	0.62602
AR (MIMO)	无	Boxcox	29915947860	120508	0.11353	0.11721	0.63892
EMA	$\alpha=0.9$	无&归一化&标准化	27419252529	108690	0.103360	0.104980	0.60736
EMA	$\alpha=0.9$	Boxcox	27835536947	114222	0.10924	0.11085	0.63827
EMA (MIMO)	$\alpha=0.9$	无&归一化&标准化	28260900226	111819	0.10601	0.10894	0.62484
EMA (MIMO)	$\alpha=0.9$	Boxcox	29858184756	120251	0.11332	0.11699	0.67196
EMA	$\alpha=0.8$	归一化	27416029462	108562	0.10326	0.10491	0.60664
DES	$\alpha=0.9$ $\beta=0.2$	无	36208536448	137220	0.12926	0.13564	0.76678
DES (MIMO)	$\alpha=0.9$ $\beta=0.2$	无	36430132738	138729	0.1311	0.13815	0.77521
DES	$\alpha=0.9$ $\beta=0.2$	标准化	35451148797	136333	0.12828	0.134559	0.76183
DES	$\alpha=0.8$ $\beta=0.2$	标准化	35477352164	136305	0.128274	0.134558	0.76167
DES	$\alpha=0.9$ $\beta=0.1$	标准化	35346409865	136005	0.128026	0.134252	0.75999

两参数指数平滑 (Double Exponential Smoothing) , 也被称为霍尔特 (Holt's) 线性趋势模型, 用于具有趋势但无季节性的时间序列数据

结论:

- 不同的转换方法, 误差在10位小数范围内没有任何区别 (**这正常吗? ? ?**)
- 关于EMA的理解:
 - 波动剧烈时alpha要设置的大
 - alpha=1等价于不平滑
 - 平滑有助于不稳定带来的累计误差增大现象
- 我还尝试了DES, 即考虑了水平和趋势两个平滑参数的指数平滑模型, 其性能更差了, 但不同归一化方法的表现有了细微差别

- 减少DES的两个参数值，可以有细微的性能提升，但DES本身很烂

五、TsfKNN

1、实现自定义距离度量

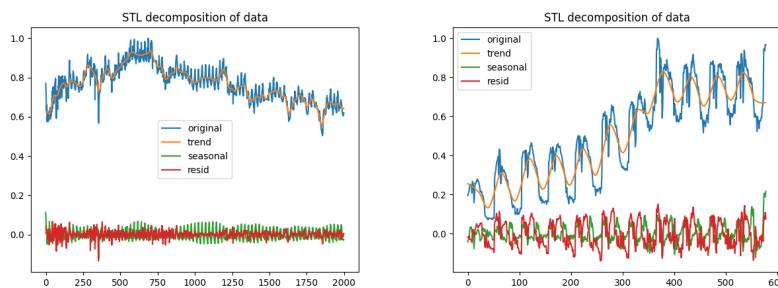
我的方法是增加一个参数 `self.decompose`，用于决定在距离度量和搜索的时候是否考虑季节性和趋势性。

(1) 具体实现方法：

1. 对整个序列进行STL分解，分解后有 `trend+seasonal+resid` 三种分量

```
def _fit(self, X: np.ndarray) -> None:
    self.X = X[0, :, -1]
    if self.decompose:
        self.X_stl = STL(self.X, period=self.period).fit() # 对整个序列进行STL分解
        plot_STL(self.X_stl, 2000)
```

分解结果如图：(左ETTh1, 右illness)



2. 将距离的计算方法改为在季节（和残差）分量上的距离，预测时使用季节或季节+残差作为预测

简写为 $s/s+r$ ，在性能表格中有体现

```
# distances=self.distance(x_stl_seasonal+x_stl_resid, X_s_seasonal[:, :seq_len]+X_s_resid[:, :seq_len])#使用季节性和残差计算距离
distances = self.distance(x_stl_seasonal, X_s_seasonal[:, :seq_len]) # 使用季节性计算距离
indices_of_smallest_k = np.argsort(distances)[:self.k]
neighbor_fore = X_s_resid[indices_of_smallest_k, seq_len:] + X_s_seasonal[indices_of_smallest_k, seq_len:] # 使用季节性+残差作为预测
# neighbor_fore = X_s_seasonal[indices_of_smallest_k, seq_len:] # 使用季节性作为预测
```

3. 单独创建一个search函数，需要传递数据分解后的结果

```

def STL_search(self,x_stl_origin, x_stl_trend,x_stl_seasonal , x_stl_resid, seq_len, pred_len):
    #在STL分解后的序列上进行搜索

    # X_s_origin = sliding_window_view(self.X_stl.observed, seq_len + pred_len)
    X_s_trend = sliding_window_view(self.X_stl.trend, seq_len + pred_len)
    X_s_seasonal = sliding_window_view(self.X_stl.seasonal, seq_len + pred_len)
    X_s_resid = sliding_window_view(self.X_stl.resid, seq_len + pred_len)
    ...

    优化后的代码，快了3倍，不再需要self._stl_modified_distance函数，但需要self.distance函数支持向量化
    ...

    if self.approximate_knn == False and self.msas == 'MIMO':
        if self.trend == 'STL':
            distances = self.distance(x_stl_trend+x_stl_seasonal[:, :seq_len]+X_s_seasonal[:, :seq_len])
            # distances = self.distance(x_stl_origin, X_s_origin[:, :seq_len])
            indices_of_smallest_k = np.argsort(distances)[:self.k]
            # neighbor_fore = X_s_origin[indices_of_smallest_k, seq_len:]#等价于不适用STL
            neighbor_fore = X_s_trend[indices_of_smallest_k, seq_len:] + X_s_seasonal[indices_of_smallest_k, seq_len:]
        elif self.trend == 't_s':
            distance_t = self.distance(x_stl_trend, X_s_trend[:, :seq_len])
            distance_s = self.distance(x_stl_seasonal, X_s_seasonal[:, :seq_len])
            indices_of_smallest_k_t = np.argsort(distance_t)[:self.k]
            indices_of_smallest_k_s = np.argsort(distance_s)[:self.k]
            neighbor_fore = X_s_trend[indices_of_smallest_k_t, seq_len:] + X_s_seasonal[indices_of_smallest_k_s, seq_len:]
        else:
            # distances=self.distance(x_stl_seasonal+ x_stl_resid, X_s_seasonal[:, :seq_len]+X_s_resid[:, :seq_len])#使用季节性和残差
            distances = self.distance(x_stl_seasonal, X_s_seasonal[:, :seq_len]) # 使用季节性计算距离
            indices_of_smallest_k = np.argsort(distances)[:self.k]
            neighbor_fore = X_s_resid[indices_of_smallest_k, seq_len:] + X_s_seasonal[indices_of_smallest_k, seq_len:] # 使用季节性作为预测
            # neighbor_fore = X_s_seasonal[indices_of_smallest_k, seq_len:] # 使用季节性作为预测

```

4. 对于趋势分量的处理一共有四种方案，简称为[plain, AR, STL, t_s]，具体含义如图中注释

经测试， plain方案的效果最好

```

# trend predict method distance used in decompose STL
parser.add_argument('--trend', type=str, default='plain', help='options: [plain, AR, STL, t_s]')#只用96个点训练线性模型，预测接下来的32个点
# parser.add_argument('--trend', type=str, default='AR', help='options: [plain, AR, STL, t_s]')#用全部trend训练AR模型，再用96个点预测接下来的32个点
# parser.add_argument('--trend', type=str, default='STL', help='options: [plain, AR, STL, t_s]')#在STL计算距离时考虑trend，实际效果基本相当于没做STL
# parser.add_argument('--trend', type=str, default='t_s', help='options: [plain, AR, STL, t_s]')#将趋势和季节分量用两个KNN匹配，再相加预测

```

5. 在forecast函数中，增加STL_search部分

```

def _forecast(self, X: np.ndarray, pred_len) -> np.ndarray:
    fore = []
    for x in tqdm(X):
        x = np.expand_dims(x, axis=0)
        x_stl = STL(x[0], period=self.period).fit()

        if self.decompose:
            # 传入STL分解后的序列进行搜索
            x_fore = self.STL_search(x_stl.observed,x_stl.trend,x_stl.seasonal,x_stl.resid, self.seq_len, pred_len)
            if self.trend == 'AR':
                x_stl_trend = self.trend_model.predict(x_stl.trend.reshape((1, -1))[:, -self.seq_len:])
                x_fore += x_stl_trend.ravel()
            elif self.trend == 'plain':
                self.trend_model.fit(np.arange(self.seq_len).reshape((-1, 1)),x_stl.trend.reshape((-1, 1))[:, -self.seq_len:])
                x_stl_trend = self.trend_model.predict(np.arange(self.seq_len, pred_len + self.seq_len).reshape((-1, 1)))
                x_fore += x_stl_trend.ravel()
            else:
                x_fore = self._search(x, self.X_slide, self.seq_len, pred_len)
        fore.append(x_fore)

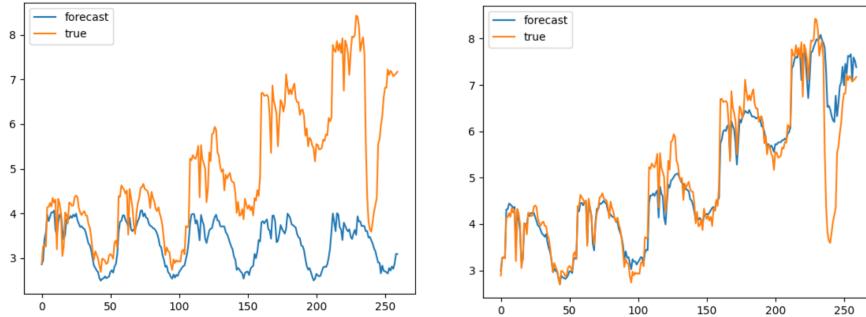
    fore = np.array(fore).reshape((-1, pred_len))
    return fore

```

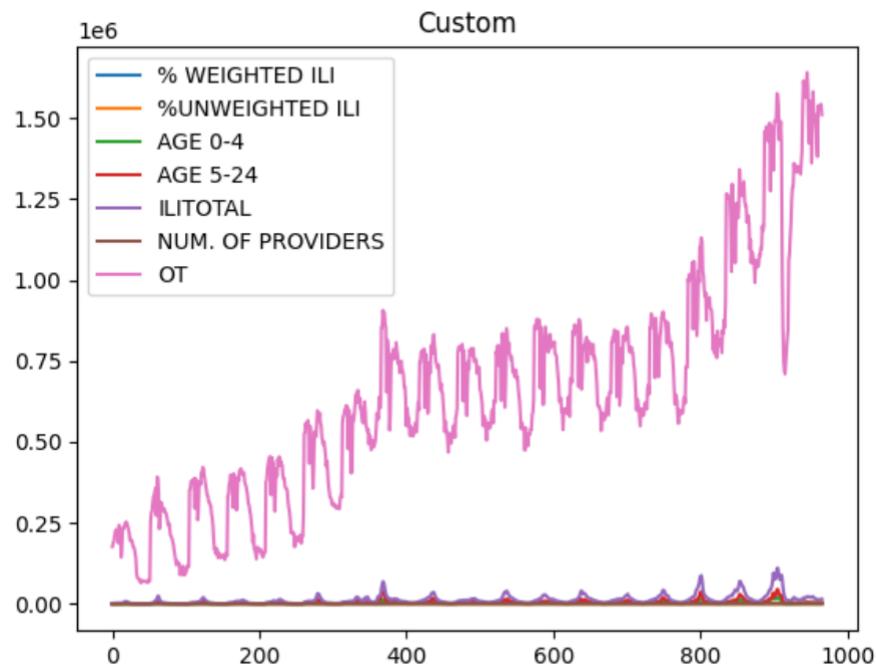
(2) 修改后的效果

TSFKNN在没见过的trend上，表现不好

使用STL分解前后的预测效果对比：



不使用STL会导致预测失效的深层原因：测试数据的分布与训练数据不同，模型没见过测试数据：



2、LSH

尝试一：MinHashLSH（失败）

```
MinHashLSH(threshold=0.01, num_perm=Num_perm)
```

用于找到Jaccard相似性下的近邻，在处理文本数据或离散数据时尤其有效

尝试了很多阈值threshold和num_perm，结果都是无法完成哈希分桶和搜索

尝试二：LSHash（成功）

```
self.lsh_model = LSHash(hash_size=self.hash_size, input_dim=self.seq_len)
```

更通用的LSH方法，可以应用于多种距离度量和数据类型。

1、建立LSH的index

```
if self.approximate_knn:  
    self.lsh_model = LSHHash(hash_size=self.hash_size, input_dim=self.seq_len)  
    for i, d in enumerate(self.X_slide):  
        point = d[:self.seq_len]  
        self.lsh_model.index(point, extra_data=i)
```

2、query

```
result = self.lsh_model.query((x_stl_trend + x_stl_seasonal+x_stl_resid).ravel(), num_results=self.k)  
  
if len(result) <= 1:  
    #搜不到时，不使用LSH  
    distances = self.distance(x_stl_seasonal, X_s_seasonal[:, :seq_len]) # 使用季节性计算距离  
    indices_of_smallest_k = np.argsort(distances)[:self.k]  
else:  
    indices_of_smallest_k = [res[0][1] for res in result]  
neighbor_fore = X_s_resid[indices_of_smallest_k, seq_len:] + X_s_seasonal[indices_of_smallest_k, seq_len:] # 使用季节性+残差作为预测
```

3、注意：

- query时不能只用季节性，因为太小了，lsh没见过这么小的数据，会匹配不到
- 总会出现搜不到的情况，注意处理，即使用非模糊knn

4、结论

- hash_size增大可以显著加快LSH速度，加到100时与原始速度相同。可以做到加速，但做不到性能提升
- recursive又菜又慢，要20min，stl分解和recur暂不兼容
- 探索STL分解后数据的使用方法改良，即季节性和残差到底要哪些的结论：寻找近邻时性能影响不大，预测时最好只用季节不用残差

3、TsfKNN性能表格

(1) illness.csv

- p=52指STL分解时周期为52
- s/s+r指在计算k近邻时用季节性计算分量，在合并预测值时用季节性分量+残差计算

decompose	LSH	mfas	k	distance	转换	MSE	MAE	MAPE	SMAPE	MASE
False	F	MIMO	5	euclidean	标准化&归一化	182565121780	336669	0.29224	0.36783	1.88129
False	T	MIMO	5	euclidean	标准化	184149265431	338432	0.2937	0.36952	1.89115
False	T	MIMO	9	chebyshev	归一化	197121862157	364129	0.32497	0.40658	2.03474
p=52, s/s+r	F	MIMO	5	euclidean	标准化	20410523866	81241	0.08449	0.07826	0.45398
p=52, s/s	F	MIMO	5	euclidean	标准化	20440592194	81130	0.08489	0.07854	0.45335
p=52, s+r/s+r	F	MIMO	5	euclidean	标准化	20344269303	81435	0.0852	0.07907	0.45506
p=52, s/s+r	F	MIMO	5	euclidean	标准化	20315292682	80639	0.08465	0.07828	0.45061
p=52, s/s+r	F	MIMO	5	euclidean	Boxcox	27316634402	93651	0.0967	0.08845	0.52332

decompose	LSH	msas	k	distance	转换	MSE	MAE	MAPE	SMAPE	MASE
p=52, s/s+r	F	MIMO	5	manhattan	归一化	20355997712	81102	0.08455	0.07828	0.4532
p=52, s+r/s	F	MIMO	5	manhattan	归一化	20420803406	81807	0.08576	0.07944	0.45714
p=52, s/s+r	F	recur	5	manhattan	归一化	20474119147	81353	0.08505	0.07867	0.4546
p=52, s+r/s+r	F	recur	5	manhattan	归一化	20943499198	83744	0.08743	0.08129	0.46796
p=52, s/s+r	F	MIMO	5	manhattan	Boxcox	27135771726	93079	0.09621	0.08808	0.52012
p=52, s/s+r	F	MIMO	5	chebyshev	归一化	20595357040	82216	0.08493	0.07859	0.45942
p=52, s/s+r	F	MIMO	9	chebyshev	归一化	20592625195	83264	0.08677	0.08047	0.46528
p=52, s/s+r	F	MIMO	3	chebyshev	归一化	20836098419	83974	0.087	0.0808	0.46924

(2) ETTh2.csv

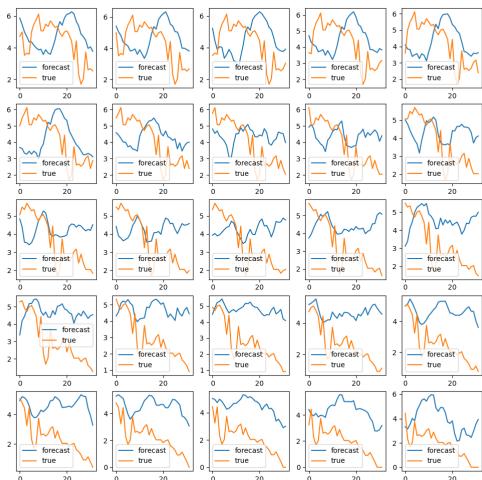
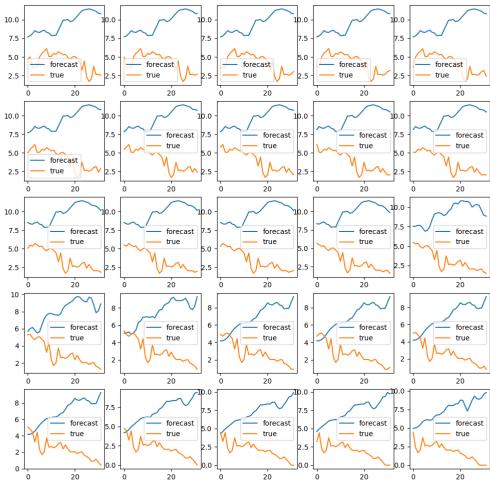
decompose	LSH	msas	k	distance	转换	MSE	MAE	MAPE	SMAPE	MASE
p=24	F	MIMO	5	euclidean	标准化&归一化	30.3512	4.09412	0.4325	0.25	1.2844
False			9	chebyshev	归一化	26.52059	3.98977	0.90756	0.22309	1.25167

(3) ETTh1.csv

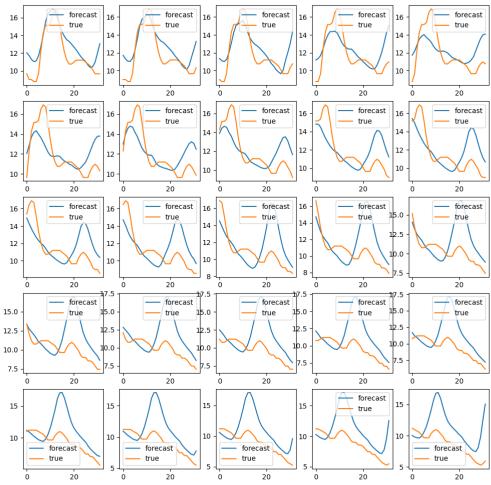
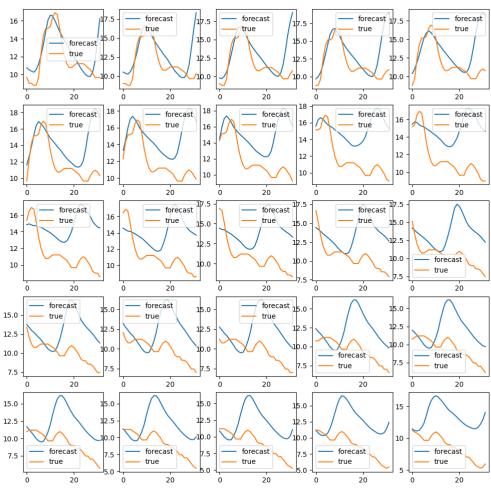
decompose	LSH	time	k	distance	转换	MSE	MAE	MAPE	SMAPE	MASE
p=24	F		5	euclidean	标准化	8.44004	2.1959	0.46301	0.42764	1.27963
p=24	F		5	manhattan	标准化	8.40811	2.19249	0.45526	0.42739	1.27764
p=24,s/s+r	F		9	manhattan	标准化	8.27351	2.17174	0.45126	0.4234	1.26555
p=24,s+r/s+r	F		9	manhattan	标准化	8.23744	2.17695	0.45028	0.42628	1.26858
p=24,s+r/s	F	1min	9	manhattan	标准化	8.13158	2.15309	0.4471	0.41932	1.25468
p=24,s+r/s	T,150	30s	9	manhattan	标准化	8.32322	2.19015	0.4531	0.42295	1.27628
p=24,s+r/s	T,200	30s	9	manhattan	标准化	8.38154	2.20315	0.45517	0.42645	1.28385
p=24,s+r/s	T,120	40s	9	manhattan	标准化	8.19432	2.16975	0.44936	0.4201	1.26439
p=24,s+r/s	T,100	1min	9	manhattan	标准化	8.14404	2.16076	0.44778	0.41825	1.25915
p=24,s+r/s	T,50	3min	9	manhattan	标准化	8.12842	2.15914	0.44713	0.41821	1.25821
p=24, s/s	F	1min	9	manhattan	标准化	8.14852	2.15086	0.44692	0.41838	1.25338
p=24,s/s	T,20	7min	9	manhattan	标准化	8.18717	2.1655	0.4497	0.42153	1.26191
p=24,s/s	T,120	40s	9	manhattan	标准化	8.19432	2.16975	0.44936	0.4201	1.26439
p=24	F		9	euclidean	标准化	8.29282	2.1735	0.45822	0.42323	1.26657
p=24	F		9	chebyshev	标准化	8.35133	2.1813	0.46039	0.42344	1.27112
False	F		5	euclidean	标准化	17.13865	3.31489	0.90555	0.48817	1.9317
False	F		5	euclidean	Boxcox	17.7098	3.37504	0.92267	0.49321	
False	F		9	euclidean	标准化	16.07376	3.19938	0.8922	0.47642	
False	F		9	manhattan	标准化	17.59547	3.32102	0.94073	0.48346	1.93527
False	F		9	chebyshev	标准化	15.01936	3.11276	0.88115	0.47039	

decompose	LSH	time	k	distance	转换	MSE	MAE	MAPE	SMAPE	MASE
False	F		9	chebyshev	Boxcox	15.17964	3.13046	0.88268	0.47206	
False	F		9	chebyshev	归一化	15.01813	3.1127	0.88106	0.47041	1.81388
False, recur	F		9	chebyshev	归一化	14.13371	2.98788	0.82238	0.45704	1.74114
False	T		9	chebyshev	归一化	15.52004	3.16088	0.88425	0.47185	1.84195

在ETTh1.csv上使用decompose前后的性能比较



在ETTh2.csv上使用decompose前后的性能比较



4、TsfKNN探索结论

- decompose: 使用STL分解后性能天差地别
- msas: MIMO速度和性能在大多数情况下都优于单步迭代预测recur
- 不同的数据集: s/s+r的最优性和最优聚类中心数都不同
- 归一化方法: Boxcox的性能最差, 其他几乎没有区别
- 模糊KNN: 性能无提升, 在hash_size很大时速度可以快一倍, hash_size很小时速度慢20倍
- distance函数: 在不分解数据的情况下, 不同的距离计算函数性能都差不多

六、代码方面的收获

本次作业的代码框架比较先进, 加上自己探索过程中的一些尝试, 因此也有一些代码方面收获

1. 内部方法子类重写raise NotImplementedError
2. 滑动窗口可以掉包实现

```
subseries = np.concatenate(([sliding_window_view(v, self.seq_len+1) for v in
x_target]))
```

3. MAPE在处理0问题时，小常数不行，容易爆掉。MASE可以用两个MAE实现
4. distance函数写成矢量化的形式更方便代码快速执行
5. transform直接调用各种Scaler，常用的方法有fit, transform, fit_transform, inverse_transform, partial_fit
6. 代码加速时，重点在for循环和每次都要调用的如distance类函数

如，消除距离函数的for循环，快了三倍

```
def _stl_modified_distance(self, x_component, y_components_series):  
    # x_component 是单个时间序列的 STL 分解结果（趋势或季节性组件）  
    # y_components_series 是原数据滑动窗口的 STL 分解结果（趋势或季节性组件）  
  
    # distances = []  
    # for y_component in y_components_series:  
    #     # 计算 x_component 与 y_component 之间的距离  
    #     dist = self.distance(x_component, y_component)  
    #     distances.append(dist)|  
    # return np.array(distances)  
  
    # 优化后的代码，快了3倍，但需要distance函数支持向量化  
distances = self.distance(x_component, y_components_series)  
return distances
```

7. 不要浪费过多时间探索本就性能差的方法
8. 预测效果出现了不理解的异常情况时一定要搞清楚原因，是哪里写错了还是没理解对