时序数据处理算子开发个人报告

探微书院 洪宇睿 2020013275

实验环境和项目介绍

本次作业由小组成员洪宇睿、刘喆骐、张梦遥完成，分工为：

- 洪宇睿：sql查询以及Integral,IntegralAvg,Mad, Median,QLB,Resample,Sample函数的编写和测试
- 刘喆骐：ACF,Histogram,Distinct,Segment,Spline,Spread,Zscore,Skew函数的编写和测试
- 张梦遥：MinMax,Mode,MvAvg,PACF,Percentile,Period函数的编写和测试

全部代码内容托管在GitHub上，链接为https://github.com/HongYurui/THSS_Data_Portraiture_Wheels，所有的实验代码均使用Python语言写成，支持Windows、GNU/Linux上至少Python3.6以上的环境。各个算子函数均存放在一个单独的文件中，该文件只含有实现相应函数的类，且与类同名；测试文件则命名为算子类名+UT的形式，分别负责对各自的算子类进行测试。除此之外，本次作业中还包含了一个批量运行所有测试文件的UTCollection.py及实现使用SQL语法进行查询这一功能的sql.py，以及测试和查询所需要用到的输入文件。
在函数的编写过程中，主要使用的函数库有用于栈帧操作和时间处理的Pandas和用于单元测试的unittest，部分函数的编写还涉及到了math, scipy, numpy等数学计算库和os等系统交互库。实验中，基本的输入输出格式均为封装了Dataframe的FlokDataframe，而所有的算子都实现为FlokAlgorithmLocal类的子类，分别重写了其run方法以实现不用的功能。在使用某个算子时，在正确提供了输入输出路径、参数的情况下，首先调用FlokAlgorithmLocal类的read方法从相应的文件中读取数据(若读取的内容中仅有部分为所需数据，还要先行调用SelectTimeSeries方法进行数据筛选，方法与一般的算子相同)，转化为FlokDataframe对象，然后创建相应的算子对象，调用其run方法计算得到FlockDataFrame类型的输出，然后视情况打印或调用继承的write方法将输出保存为文件。

主要功能和算法分析

栈帧输出格式化和SQL查询的实现

一种常见的SQL查询格式为：

```
select $columns from $table [where $condition];
```

即给定一个待查询的表名、一个或多个待查询的列名、可选的一个或多个条件，输出在表格中相应的列下满足条件的行中的数据。这在实际的数据库中只是一种基础的数据查询，在Python中，如果已有相应的表格，亦可通过pandasql库中提供的sqldf方法进行查询。然而，我们希望的是在查询中加入算子和相应的参数，即进行如下形式的查询：

```
select $algorithm($columns, $params) from $input [where $condition];
```

按照SQL语法，$algorithm($columns, $params)应为列名，$input应为表名，故基本的实现思路就是对每一次查询，调用相应的算子函数生成一个名为$input(即与输入文件的名称相同)、数据列名为$algorithm($columns, $params)这个字符串的DataFrame。为了方便在查看数据数值的同时关注其时间戳，还可加入Time列。这就要求在各算子能正确计算结果、格式化输出的基础上，在sql.py中根据输入文件名称(即一个任意字符串)创建并维护相应的对象。

对于不接受参数的算子，列名的格式化较为简单，只需要正确显示函数名和列名即可，如下：

```
```

```
# from Acf.py, as an example
value_column_name = 'acf({name})'.format(name=column)
```

或亦可直接使用简单的字符串拼接。

而对于接受参数的算子，列名的格式化较为复杂，若此参数列表在初始情况下为字典形式，则需要将其还原为形如Python函数调用的字符传形式。由于要对其中的等号，引号等进行格式控制，这里实现了一个简单可复用的模板：

```
# from Integral.py, as an example

# header format
value_header = 'integral(' + input_data.columns[1]
param_list = ['unit']
for param in param_list:
    if param in params:
        value_header += ', \'' + param + '\'=\'' + str(params[param]) + '\''
value_header += ')'
```

即对于每个函数，指定其名称和合理的参数键列表，然后遍历该列表，依次检查输入参数字典中的每个键是否在参数列表中，若存在则将其值按照格式规则拼接到`value_header`中。这样，若出现原始输入字典键值对集合相同而次序不一致的情况，仍可被识别为接受了相同参数。

栈帧对象的动态创建则利用了Python中的`globals()`或`locals()`方法，对于输入的文件名字符串，利用`globals()[$filename]=$value`就实现了名为`$filename`的对象的动态创建。为了减少不必要的磁盘读写，这里对每个输入文件的内容也同样地动态创建对象来存储在内存中，为了不至于混淆，统一表示为`globals()['orig' + $filename]`。

在同一张表格中，多组数据可能共享相同时间戳列，而事实上的长度并不相同，尾部可能存在空白数据。这对任何算子都非有效输入，甚至可能干扰算子的正常运行。因此，需要在查询之前对数据进行合理裁剪，方可进行相应的算子调用：

```
# from sql.py
i = len(dataframe) - 1
while isna(dataframe.iloc[i, 1]):
    i -= 1
dataframe = dataframe.iloc[:i + 1, :]
```

`sql.py`通过以`while`控制的字符串输入输出和正则表达式实现了一个类似于`PyPI`的`CLI`供进行查询。一方面，该交互模块可以根据一些简单的用户输入打印提示信息或利用`os`模块调用`ls, cat, clear`等命令实现一些简单的功能，另一方面，该模块在用户输入`SQL`查询的字符串时会存储下其内容，首先利用正则表达式进行解析，提取其中的输入文件名、列名、函数名和参数列表的信息，调用算子计算、格式化得到结果，最后使用原字符串进行`pandasql.sqldf(df)`的查询，并输出结果。

Integral积分算法的实现

```
# from Integral.py
i = 0
while pd.isna(value_data[i]):
    i += 1
while i < len(value_data) - 1:
```

```
    j = i + 1
    while pd.isna(value_data[j]):
        j += 1
    output_data.iloc[0, 1] += (value_data[i] + value_data[j])
                            * (time_data[j] - time_data[i]).seconds / 2
    i = j
```

此处积分算法的目标是对于给定的，可能含有空值的序列利用梯形法进行积分。一种想法是先对序列进行线性插值而保证计算得到的梯形面积不变，然后对插值后所有数据点进行求和，这样需要对序列至少遍历两次，而若动态地改变每个子梯形的高，即在遇到空值时先行跳过，遇到下一个非空值时再以之前走过的长度为高，利用梯形面积的公式进行计算，这样就只需要遍历整个序列一次。
在实现了积分的基础上，`IntegralAvg`这样一个积分均值算法就是对已有算子的简单应用，只需要对`Integral`的结果再除以时间序列长度即可。

Median中位数算法及Mad算法的实现

对于求中位数的任务，一个简单的想法是对排序后直接按位次求解。由于此时的输入范围没有限制，只能使用"基于比较的排序算法"，其中最好的快速排序算法的时间复杂度为`O(nlogn)`，这对求中位数这样的简单任务而言复杂度过高，在原始数据的规模较大时性能不够理想，而且对位次偏差比例的容忍度`error`也无从利用。

受到快速排序分治思想的启发，我们可以递归地用这样的方法求解中位数：首先，将"中位数"这个目标泛化为"从给定的序列中求某个指定位次的数"，即求第k小的数，然后，每次从当前序列中随机化地选择一个`pivot`，遍历一次序列将其分为大于等于`pivot`和小于`r`的两部分，并获取`pivot`的位次。这样，我们希望得到的位次必然位于两者之一，那就可以只求解一个规模期望为原来的$\frac{1}{2}$的子问题。由算法中的主定理，此即为一个线性复杂度的算法。

进一步，当给定了对位次偏差比例的容忍度`error`时，我们可以依此计算出可接受的最大位次偏差，递归地应用于各级子问题。当某次求得的`pivot`位次与目标位次的差小于这个值时，就可以直接返回`pivot`作为解，这样就利用这一容忍度节省了运行时间。

除此之外，对于原有偶数个值的序列，本应求出中间两个位次所对应的数值，但由于它们的位次相邻，故不需要真正进行两次不同的位次查询，而只要稍作讨论即可。线性复杂度，支持`error`、针对偶数长度序列进行了一定优化后的算法如下(实现为`Median`对象的方法)：

```
# from Median.py
# partition
def randomized_partition(self, data):
    r = len(data) - 1
    p = random.randint(0, r)
    data[p], data[r] = data[r], data[p]
    p = 0
    for j in range(r):
        if data[j] <= data[r]:
            data[p], data[j] = data[j], data[p]
            p += 1
    data[p], data[r] = data[r], data[p]
    return p, data

# find a single number of certain rank
def odd_find_rank(self, data, rank, error=0):
    p, data = self.randomized_partition(data)
    if p > rank + error:
```

```
            return self.odd_find_rank(data[:p], rank, error)
        if p < rank - error:
            return self.odd_find_rank(data[p + 1:], rank - p - 1, error)
        return data[p]

    # find two adjacent numbers of certain rank
    def even_find_rank(self, data, left_rank, error=0):
        p, data = self.randomized_partition(data)
        if left_rank > p + error:
            return self.even_find_rank(data[p + 1:], left_rank - p - 1, error)
        if left_rank + 1 < p - error:
            return self.even_find_rank(data[:p], left_rank, error)
        if error == 0:
            if left_rank == p:
                return (data[p] + self.odd_find_rank(data[p + 1:], 0, error)) / 2
            if left_rank + 1 == p:
                return (self.odd_find_rank(data[:p], left_rank, error)
                        + data[p]) / 2
        else:
            return (data[p - 1] + data[p]) / 2

    # find median through even and odd rank-finding methods
    def median(self, data, error=0):
        assert 0 <= error <= 1
        length = len(data)
        if length % 2 == 0:
            return self.even_find_rank(data, (length - 1) // 2,
                   int(error * length))
        else:
            return self.odd_find_rank(data, (length - 1) // 2,
                   int(error * length))
```

经过测试，该算法相比于基于排序的算法有运行效率上的显著优势，在error=0的默认情况下与pandas内置的median方法接近，而在error有一定大小时能显著提高运行效率。在999990个随机数的一次测试中，pandas内置方法、error=0，0.01的本方法的运行实时间分别为0.30526185035705566，0.23489761352539062，0.13114237785339355秒。

后续的Mad方法事实上也主要由两次中位数查询实现，即先调用一次Median得到中位数，遍历整个集合求得各个差的绝对值，然后再求一次中位数。这只是已实现的函数的简单调用，这里不再赘述。

QLB算法

QLB的基本思路是先求出序列的自相关系数，由QLB的计算公式：

$$Q(m) = T(T+2)\sum_{i=1}^{m}\frac{\hat{\rho_i}}{T-i}$$

先行计算出求和号中事实上为定值的各项，然后在输入参数要求的范围内每次更新前m项的总和，代入卡方分布的CDF中即可计算出相应的Q_{LB}。具体的代码如下：

```
# calculate square acf
acf_data = acf(value_data)[1:]
weighted_square_acf = [x ** 2 / (n - i - 1) for i, x in enumerate(acf_data)]
sum_weighted_square_acf = 0
```

```
for i in range(lag):
    timestamp += timedelta
    sum_weighted_square_acf += weighted_square_acf[i]
    output_data.iloc[i, 0] = timestamp.strftime("%Y-%m-%d %H:%M:%S.%f")[:-3]
    # LB Q-test
    output_data.iloc[i, 1] = 1 - chi2.cdf(n * (n + 2)
                            * sum_weighted_square_acf, i + 1)
```

Resample重采样算法和Sample采样算法

上述算法外，我在项目中实现的还有Resample和Sample。其中，Resample的编写较为繁琐，但数学思想和编程逻辑较为简单。在这一过程中，我认为在运行效率上值得一提的代码改进是针对输入参数进行一次判断，然后定义相应的resample_func函数进行具体的重采样，而非在resample_func函数的内部进行判断，从而节省了对每一次采样都重新判断采样方法的时间。大致的实现如下：

```
if period <= orig_period:
    if interp == "nan":
        def resample_func(timestamp, orig_idx):
            if time_data[orig_idx] < timestamp - time_tol:
                orig_idx += 1
            return orig_idx, value_data[orig_idx]
                if time_data[orig_idx] < timestamp + time_tol else pd.NA
    elif interp == "ffill":
        def resample_func(timestamp, orig_idx):
            if time_data[orig_idx] < timestamp + time_tol:
                orig_idx += 1
            return orig_idx, value_data[orig_idx - 1]
    ...
```

Sample函数更为简单，相比Resample连较为繁琐的过程也没有了，故不再赘述。

单元测试及其集合

利用unittest模块，我们能及其方便地对算法进行单元测试。对于每个单元测试文件，其组织架构为setUp->test_algorithm1->test_algorithm_2->...->test_algorithm_n->tearDown，其中setUp对测试进行初始化，一系列test_algorithm_n是各个测试函数的特有部分，最终由tearDown执行各个测试共有的算法主体部分。为了尽可能使用这种特性实现代码复用，我们仅将和输入数据、参数列表等的初始化放在test_algorithm_n中，而其他的算法主体部分则放在tearDown中。
在UTCollection中，为遍历以上所有的单元测试，只需要导入所有的算子类，创建一个以算子类名为元素的表格，由于各个单元测试文件中具体的测试函数严格用test_$algorithm_n的形式命名，故可以直接搜索这些文件中的全局变量名，进行批量导入：

```
for func in ["Acf", "Distinct", "Histogram", "Integral", "Integralavg",
             "Mad", "Median", "Minmax", "Mode", "Mvavg", "Pacf",
             "Percentile", "Period", "Qlb", "Resample", "Sample",
             "Segment", "SelectTimeseries", "Skew", "Spline", "Spread",
             "Stddev", "Zscore"]:
    func_ut = globals()[func + "UT"]
    i = 1
    while True:
        specific_test = "test_" + func.lower() + "_" + str(i)
        if hasattr(func_ut, specific_test):
            suite.addTest(func_ut(specific_test))
```

```
            i += 1
        else:
            break
```

然后运行r = unittest.TextTestRunner(),r.run(suite)即可，不需要额外的输入，结果中会显示所有单元测试的输出，以及测试的总组数和总时长。

代码测试及其结果

首先，对于题中给出的一般情景下的样例，我撰写的这些函数均给出了正确的结果，输入和输出如下：

输入样例：

Time	Integral	Mad	Median	Qlb	Resample	Sample	IntegralAvg
2022-01-01 00:00:00	1	0.5319929	0.5319929	1.22	3.09	1	1
2022-01-01 00:00:01	2	0.9304316	0.9304316	-2.78	3.53	2	2
2022-01-01 00:00:02	5	-1.4800133	-1.4800133	1.53	3.5	3	5
2022-01-01 00:00:03	6	0.6114087	0.6114087	0.7	3.51	4	6
2022-01-01 00:00:04	7	2.5163336	2.5163336	0.75	3.41	5	7
2022-01-01 00:00:05	8	-1.0845392	-1.0845392	-0.72		6	8
2022-01-01 00:00:06	NaN	1.0562582	1.0562582	-0.22		7	NaN
2022-01-01 00:00:07	10	1.3867859	1.3867859	0.28		8	10
2022-01-01 00:00:08	-0.45429882	-0.45429882	0.57		9		
2022-01-01 00:00:09	1.0353678	1.0353678	-0.22		10		
2022-01-01 00:00:10	0.7307929	0.7307929	-0.72				
2022-01-01 00:00:11	2.3167255	2.3167255	1.34				
2022-01-01 00:00:12	2.342443	2.342443	-0.25				
2022-01-01 00:00:13	1.5809103	1.5809103	0.17				
2022-01-01 00:00:14	1.4829416	1.4829416	2.51				
2022-01-01 00:00:15	1.5800357	1.5800357	1.42				
2022-01-01 00:00:16	0.7124368	0.7124368	-1.34				
2022-01-01 00:00:17	-0.78597564	-0.78597564	-0.01				

Time	Integral	Mad	Median	Qlb	Resample	Sample	IntegralAvg
2022-01-01 00:00:18		1.2058644	1.2058644	-0.49			
2022-01-01 00:00:19		1.4215064	1.4215064	1.63			
2022-01-01 00:00:20		1.2808295	1.2808295				
2022-01-01 00:00:21		-0.6173715	-0.6173715				
2022-01-01 00:00:22		0.06644377	0.06644377				
2022-01-01 00:00:23		2.349338	2.349338				
2022-01-01 00:00:24		1.7335888	1.7335888				
2022-01-01 00:00:25		1.5872132	1.5872132				

输出：

- Integral:

```
>>> select Time, "integral(s4)"
    from root_test_d1
    where time <= '2020-01-01 00:00:11';
                     Time  integral(s4)
0  1970-01-01 08:00:00.000          57.5
>>> select Time, "integral(s4, 'unit'='1m')"
    from root_test_d1
    where time <= '2020-01-01 00:00:10'

                     Time  integral(s4, 'unit'='1m')
0  1970-01-01 08:00:00.000                   0.958333
```

- Mad:
  (表中数据不全，故结果数值无参考意义)

```
>>> select Time, "mad(s5)" from root_test_d1;
                   Time   mad(s5)
0  2022-01-01 00:00:00  0.487902
```

- Median:
  (表中数据不全，故结果数值无参考意义)

```
>>> select Time, "median(s6)" from root_test_d1;
                     Time  median(s6)
0  1970-01-01 08:00:00.000    1.131061
```

- QLB:

```
>>> select Time, "qlb(s13)" from root_test_d1;
                       Time  qlb(s13)
0   1970-01-01 08:00:00.001  0.216870
```

```
1   1970-01-01 08:00:00.002  0.306895
2   1970-01-01 08:00:00.003  0.421786
3   1970-01-01 08:00:00.004  0.511454
4   1970-01-01 08:00:00.005  0.656062
5   1970-01-01 08:00:00.006  0.772240
6   1970-01-01 08:00:00.007  0.853249
7   1970-01-01 08:00:00.008  0.902858
8   1970-01-01 08:00:00.009  0.943499
9   1970-01-01 08:00:00.010  0.895028
10  1970-01-01 08:00:00.011  0.770105
11  1970-01-01 08:00:00.012  0.784554
12  1970-01-01 08:00:00.013  0.594303
13  1970-01-01 08:00:00.014  0.461841
14  1970-01-01 08:00:00.015  0.264595
15  1970-01-01 08:00:00.016  0.316753
16  1970-01-01 08:00:00.017  0.233001
17  1970-01-01 08:00:00.018  0.066661
```

- Resample:

```
>>> select Time, "resample(s14, 'every'='5m', 'interp'='linear')"
    from root_test_d1;
                      Time resample(s14, 'every'='5m', 'interp'='linear')
0   2021-03-06 16:00:00.000                                          3.09
1   2021-03-06 16:05:00.000                                      3.236667
2   2021-03-06 16:10:00.000                                      3.383333
3   2021-03-06 16:15:00.000                                          3.53
4   2021-03-06 16:20:00.000                                          3.52
5   2021-03-06 16:25:00.000                                          3.51
6   2021-03-06 16:30:00.000                                           3.5
7   2021-03-06 16:35:00.000                                      3.503333
8   2021-03-06 16:40:00.000                                      3.506667
9   2021-03-06 16:45:00.000                                          3.51
10  2021-03-06 16:50:00.000                                      3.476667
11  2021-03-06 16:55:00.000                                      3.443333
12  2021-03-06 17:00:00.000                                          3.41
>>> select Time, "resample(s14, 'every'='30m', 'aggr'='first')"
    from root_test_d1;
                     Time resample(s14, 'every'='30m', 'aggr'='first')
0  2021-03-06 16:00:00.000                                        3.09
1  2021-03-06 16:30:00.000                                         3.5
2  2021-03-06 17:00:00.000                                        3.41
>>> select Time, "resample(s14, 'every'='30m', 'start'='2021-03-06 15:00:00')"
    from root_test_d1;
                     Time resample(s14, 'every'='30m', 'start'='2021-03-06 15
0  2021-03-06 15:00:00.000                                             NaN
1  2021-03-06 15:30:00.000                                             NaN
2  2021-03-06 16:00:00.000                                            3.31
3  2021-03-06 16:30:00.000                                           3.505
4  2021-03-06 17:00:00.000                                            3.41
```

- Sample:

```
>>> select Time, "sample(s15, 'method'='reservior', 'k'='5')"
    from root_test_d1;
                  Time sample(s15, 'method'='reservoir', 'k'='5')
0  2022-01-01 00:00:00                                        1.0
1  2022-01-01 00:00:01                                        2.0
4  2022-01-01 00:00:04                                        5.0
2  2022-01-01 00:00:06                                        7.0
3  2022-01-01 00:00:08                                        9.0
>>> select Time, "sample(s15, 'method'='isometric', 'k'='5')"
    from root_test_d1;
                  Time sample(s15, 'method'='isometric', 'k'='5')
0  2022-01-01 00:00:00                                        1.0
1  2022-01-01 00:00:02                                        3.0
2  2022-01-01 00:00:04                                        5.0
3  2022-01-01 00:00:06                                        7.0
4  2022-01-01 00:00:08                                        9.0
```

- Integral:

```
>>> select Time, "integralavg(s21)"
    from root_test_d1;
                     Time  integralavg(s21)
0  1970-01-01 08:00:00.000              5.75
```

作业总结

通过这次作业，我进行了一系列时序数据算子的重新开发(造轮子)，掌握了Python单元测试和仿CLI的写法，对git这一代码管理工具的使用也有了一定的了解。