Оглавление

Теоретическое введение

KZ-фильтр

KZ-фильтр является фильтром низких частот. Он основан на дискретном преобразовании Фурье. Для алгоритма фильтра производные высших порядков для дискретных функций были определены через конечные разности высших порядков. Была взята конечно-дифференцируемая оконная функция с конечным носителем, для которой был построен дискретный аналог.

Данный фильтр предназначен для выявления особенностей во временных рядах, таких как тренды, краткосрочные/долгосрочные колебания, сезонность.

Формальное определение

Пусть $X(t), t = \pm 1, \pm 2, \cdots -$ вещественный временной ряд, KZ-фильтр с параметрами m и k определяется следующим образом:

$$KZ_{m,k}[X(t)] = \sum_{s=-k(m-1)/2}^{k(m-1)/2} X(t+s) \times a_s^{m,k}$$

где коэффициенты

$$a_s^{m,k} = \frac{c_s^{k,m}}{m^k}, s = \frac{-k(m-1)}{2}, \ldots, \frac{k(m-1)}{2}$$

определяются коэффициентами многочлена, полученного из уравнения

$$\sum_{r=0}^{k(m-1)} z^r c_{r-k(m-1)/2}^{k,m} = (1 + z + \cdots + z^{m-1})^k$$

Это формула явного вида коэффициентов, вычисляемых на итерациях скользящего среднего. Необходимо дополнительное исследование точности получаемых по данной формуле значений.

Через итерации скользящего среднего:

$$KZ_{m,k=1}[X(t)] = \sum_{s=-(m-1)/2}^{(m-1)/2} X(t+s) \times \frac{1}{m}$$

$$KZ_{m,k=2}[X(t)] = \sum_{s=-(m-1)/2}^{(m-1)/2} KZ_{m,k=1}[X(t+s)] \times \frac{1}{m}$$

$$= \sum_{s=-2(m-1)/2}^{2(m-1)/2} X(t+s) \times a_s^{m,k=2}$$

В общем случае k-я итерация представляет собой применение фильтра скользящего среднего к $(k-1)$-й итерации.

Полученный в результате фильтрации временной ряд является низкочастотной составляющей изначального временного ряда.

Применение и преимущества

KZ-фильтр применим для сглаживания периодограмм. Для класса стохастических процессов, при наихудшем сценарии, когда единственной доступной информацией о процессе является его спектральная плотность и гладкость, определяемая показателем Липшица, оптимальная ширина спектрального окна зависит только от базовой гладкости спектральной плотности. При сравнении с другими оконными функциями, Журбенко в работе *The Spectral Analysis of Time Series (North-holland Series in Statistics and Probability)* был сделан вывод о практической оптимальности окна, используемого в фильтре Колмогорова-Журбенко.

Результат фильтрации получается путем многократных итераций простого скользящего среднего. Потому он хорошо работает в условиях недостающих данных, особенно в многомерных временных рядах, где эта проблема возникает из-за пространственной разреженности. Фильтр является устойчивым к выбросам.

Однако KZ-фильтр имеет тенденцию сглаживать резкие разрывы (включая всплески и впадины), что затрудняет оценку свойств временного ряда.

KZA-фильтр

Адаптивная версия KZ-фильтра, названная KZA-фильтром, была разработана для поиска разрывов в непараметрических, сильно зашумленных сигналах. KZA-фильтр сначала определяет потенциальные временные интервалы, в которых происходит разрыв. Затем проводится более тщательное исследование этих временных интервалов и уменьшение размера окна таким образом, что качество сглаженного результата улучшается.

То есть, KZA является расширением KZ-фильтра и позволяет выявлять любые резкие разрывы во временных рядах, не искажая другие, свойственные ряду,

закономерности. По сравнению с KZ-фильтром, преимущество KZA заключается в применении динамического окна сглаживания, во время выполнения процесса скользящего среднего.

Формальное определение

Обозначим

$$Z(t) = KZ_{m,k}[X(t)]$$

Абсолютные значения $D(t)$ дифференцированных значений $Z(t)$ определяются следующим образом

$$D(t) = |Z(t+w) - Z(t-w)|, w = \frac{k(m-1)}{2}$$

Скорость изменения $D(t)$ определяется

$$D(t)' = D(t+1) - D(t)$$

Когда точка данных находится в области возрастания $D(t)$, полудлина окна сглаживания перед этой точкой $(left, w_T)$ равна w, а длина перед точкой $(right, w_H)$ уменьшается пропорционально $D(t)$. В области убывания $D(t)$ будет уменьшаться только длина за точкой данных $(left)$.

Адаптивный фильтр можно определить следующим образом:

$$KZA[X(t)] = \frac{1}{w_T(t) + w_H(t)} \sum_{s=-q_T(t)}^{q_H(t)} X(t+s)$$

где

$$q_H(t) = \begin{cases} w, D'(t) < 0 \\ f(D(t))w, D'(t) \geq 0 \end{cases}.$$

$$q_T(t) = \begin{cases} w, D'(t) > 0 \\ f(D(t))w, D'(t) \leq 0 \end{cases}.$$

w - полудлина окна в исходном KZ-фильтре. $f(D(t))$ определяется следующим образом:

$$f(D(t)) = 1 - \frac{D(t)}{max[D(t)]}$$

Затем исходные данные подвергаются итерационной фильтрации, как в случае KZ-

фильтра, но с использованием новых длин w_H и w_T.

Применение и преимущества

Для одномерных данных он может использоваться для выявления проблем с качеством данных.

Для двумерных данных (таких как данные двумерных компьютерных томографов, двумерные метеорологические данные) KZA может быть применен для обнаружения разрывов (изменение цвета и/или плотности).

Для трехмерных данных (например трехмерные пространственные данные или трехмерные медицинские изображения) KZA также может быть применен для обнаружения разрывов (границ между слоями)

В случае данных с более высокой размерностью, KZA с некоторыми модификациями также может быть использован для обнаружения разрывов сигнала.

Фильтр демонстрирует очень высокую чувствительность для обнаружения разрывов даже при очень низком соотношении сигнал/шум.

Префиксные суммы

Определние. Префиксными суммами массива $[a_0, a_1, .., a_{n-1}]$ называется масив $[s_0, s_1, .., s_n]$, определенный следующим образом:

- $s_0 = 0$
- $s_1 = a_0$
- $s_2 = a_0 + a_1$
- $s_3 = a_0 + a_1 + a_2$
- ...
- $s_n = \sum_{i=0}^{n-1} a_i$

Формулу для s_k можно записать рекуррентно как $s_k = s_{k-1} + a_{k-1}$, что сразу дает возможность подсчитывать префиксные суммы за линейное время.

Рассмотрим задачу, в которой использование префиксых сумм позволяет существенно ускорить наивное решение. Пусть дан массив чисел и дано q запросов вида "найти сумму на полуинтервале с позиции l_i до позиции r_i, $i = 1, 2, ..., q$.

Наивное решение заключается в итеративном суммировании элементов, находящихся внутри полуинтервала $[l_i, r_i)$ для каждого запроса. Поэтому асимптотика наивного решения равна $O(q * n)$.

Далее предподсчитаем перед ответами на запросы массив префиксных сумм для исходного массива. Тогда если бы во всех запросах l было равно нулю, то ответом на запрос просто могла быть префиксная сумма s_{r_i}. Но как быть, если $l \neq 0$?

Заметим, что в префиксной сумме s_{r_i} содержатся все нужные нам элементы, однако есть еще лишние - $a_0, a_1, .., a_{l_i-1}$. Тогда заметим, что такая сумма в свою очередь равна уже посчитанной префиксной сумме s_{l_i}. Таким образом, выполнено тождество:

$$a_{l_i} + a_{l_i+1} + ... + a_{r_i-1} = s_{r_i} - s_{l_i},$$

то есть для ответа на запрос поиска суммы на произвольном полуинтервале нужно просто вычесть друг из друга две предподсчитанные префиксные суммы. Таким образом, итоговая асимптотика полученного решения равно $O(n + q)$, где первое слагаемое равно времени, затраченному на предподсчет префиксных сумм, а второе время ответа на все q запросов, поскольку на каждый в отдельности мы умеем отвечать за $O(1)$.

Оптимизации и распараллеливание

Оптимизация с помощью префиксных сумм

Рассмотрим применение префиксных сумм для подсчета значения фильтра в фиксированном окне.

Пусть значения расположены в массиве data. Построим массив префиксных сумм $pref\_sum$, как было объяснено выше, с той лишь разницей, что будем игнорировать элементы, которые бесконечны или NaN, по правилам

- $pref\_sum[0] = 0$
- $pref\_sum[i] = pref\_sum[i-1] + data[i-1]$

Аналогично построим массив префиксного количества элементов $pref\_finite\_cnt$ с разницей в том, что

$$pref\_finite\_cnt[i] = pref\_finite\_cnt[i-1] + 1$$

для $i = 1, 2, ...$

В приведеннном ниже коде представлен процесс построения описанных выше массивов

```
static void calc_prefix_sum(const double *data, int size, double
    *pref_sum, int *pref_finite_cnt)
{
    int i;

    pref_sum[0] = 0;
    pref_finite_cnt[0] = 0;
    for (i = 1; i <= size; i++) {
        pref_sum[i] = pref_sum[i-1];
        pref_finite_cnt[i] = pref_finite_cnt[i-1];
        if (isfinite(data[i-1])) {
            pref_sum[i] += data[i-1];
            ++pref_finite_cnt[i];
        }
    }
}
```

Листинг 1. Построение префиксных сумм и количеств

Предположим, мы зафиксировали длину окна w слева и справа от центра $window\_center$ и z - количество элементов в окне. Тогда мы хотим посчитать

$$(data[window\_center - w + 1] + data[window\_cneter - w + 2] + ...$$
$$+data[window\_center + w - 1] + data[window\_center + w])/z$$

Заметим, что числитель выражения может быть представлен через предподсчитанные нами префиксные суммы как

$$\frac{pref\_sum[window\_center + w + 1] - pref\_sum[window\_center - w]}{pref\_cnt[window\_center + w + 1] - pref\_cnt[window\_center - w]}$$

Таким образом, для того, чтобы подсчитать значение фильтра в фиксированном окне, нам достаточно предподсчитать оба массива, а затем за $O(1)$ мы можем отвечать на данный запрос. В приведенном ниже коде представлена функция, возвращающая значение фильтра в окне:

```
static double mavg1d(const double *pref_sum, const int *pref_finite_cnt
    , int length, int col, int w)
{
    double s;
    int z;
    int start_idx, end_idx;

    /* window length is 2*w+1 */
    start_idx = (window_center+1) - w; /* the first window value index
    */
    if (start_idx < 0)
        start_idx = 0;

    end_idx = (window_center+1) + w; /* the last window value index */
    if (end_idx > data_size)
        end_idx = data_size;

    /* (window sum) = (sum containig window) - (sum before window) */
    s = (pref_sum[end_idx] - pref_sum[start_idx-1]);
    z = (pref_finite_cnt[end_idx] - pref_finite_cnt[start_idx-1]);
    /*
    if (z == 0)
        return nan("");
```

```
22      */
23      return s/z;
24  }
```

Листинг 2. Расчет значения фильтра в окне с помощью префиксных сумм и количеств

Стратегии распараллеливания программы

В программе реализованы две стратегии распараллеливания: клиент-серверная (в коде: `KZ_THREADS_CLIENT_SERVER` и `KZA_THREADS_CLIENT_SERVER`) и распараллеливание цикла (`KZ_THREADS_LOOP`).

Распараллеливание способом клиент-сервер

Одной из стратегий распараллеливания задач является клиент-серверный способ. В этом сценарии независимые задачи распределяются между несколькими рабочими потоками-клиентами, которые взаимодействуют с потоком-сервером. Сервер распределяет задачи между клиентами, и когда происходит какое-либо событие, рабочий поток сообщает об этом управляющему, в он, в свою очередь, обрабатывает данные в контексте программы.

Рассмотрим применение данного подхода в рамках нашей задачи. В функции сервера создается и инициализируется семафор, создаются потоки. После чего на каждой итерации сервер ждет, пока освободится хотя бы один работник. Если количество свободных клиентов равно числу тредов, мы запускаем итерацию пересчета и сообщаем семафору, что потоки свободны. После запуска тредов на последней итерации сервер ожидает их завершения (функция `wait_treads`), для того чтобы освободить ресурсы, используемые семафором. Ниже приведен листинг функции сервера.

```
1  static void threads_server_loop(struct thread_data *th, int threads_cnt
       , struct task_data *task)
2  {
3      int iter, idle_workers_count;
4      sem_t finished_workers_sem;
5
6      sem_init(&finished_workers_sem, 0, 0);
7
8      start_threads(th, threads_cnt, task, &finished_workers_sem);
9
```

```
    idle_workers_count = 0;
    iter = 0;
    while (iter < task->iterations) {
        sem_wait(&finished_workers_sem);
        idle_workers_count++;
        if (idle_workers_count == threads_cnt) {
            int i;
            update_iteration_data(task->data, task->ans, task->
   data_size);
            idle_workers_count = 0;
            for (i = 0; i < threads_cnt; i++)
                sem_post(&th[i].can_work_sem);
            iter++;
        }
    }

    wait_threads(th, threads_cnt);
    sem_destroy(&finished_workers_sem);
}
```

Листинг 3. Функция сервера в стратегии клиент-сервер

В приведенной ниже функции для каждого клиента создается семафор `can_work_sem`, отвечающий за ожидание разрешения на выполнение итерации. После завершения этой итерации поток-клиент увеличивает семафор, сообщающий серверу о готовности потока к следующей итерации.

```
static void *worker(void *data)
{
    int k;

    struct thread_data *thread_data = data;
    struct task_data *task = thread_data->task;

    for (k = 0; k < task->iterations; k++) {
        sem_wait(&thread_data->can_work_sem);

        perform_task_iteration(task, thread_data->start_idx,
                               thread_data->end_idx);

        sem_post(thread_data->finished_workers_sem);
    }
```

```
16
17      return NULL;
18  }
```

Листинг 4. Функция клиента (рабочего) в стратегии клиент-сервер

Аналогичная стратегия распараллеливания используется в коде программы KZ-фильтра.

Распараллеливание цикла

При таком подходе цикл разбивается на задачи, выполняемые параллельно.

При данной стратегии распараллеливания, в функции `kz1d` мы инициализируем задачи, после чего на каждой итерации алгоритма мы запускаем нити, каждая из которых независимо от других считает значение фильтра в своем окне. В конце итерации нити уничтожаются и пересчитываются массивы.

```
1  task_size = length / tasks_cnt;
2
3  init_tasks(tasks, tasks_cnt, task_size, window, length,
4             ans, pref_sum, pref_finite_cnt);
5
6  for (k = 0; k < iterations; k++) {
7      start_threads(th, tasks_cnt, tasks);
8
9      for (i = (tasks_cnt)*task_size; i < length; i++)
10         ans[i] = mavg1d(pref_sum, pref_finite_cnt, length, i, window);
11
12     wait_threads(th, tasks_cnt);
13
14     calc_prefix_sum(ans, length, pref_sum, pref_finite_cnt);
15 }
```

Листинг 5. Стратегия распараллеливания цикла

Сравнение стратегий

- В клиент-серверной стратегии потоки создаются один раз и уничтожаются после завершения алгоритма.
- При распараллеливании цикла создание потока происходит в начале каждой итерации, а уничтожение - по ее завершении.
- Стратегию клиент-сервер выгоднее использовать при небольшом размере данных.

- При большом размере данных выгоднее использовать распараллеливание цикла, поскольку время создания тредов мало в сравнении с временем обработки данных.
- В клиент-серверной стратегии используются объекты синхронизации, что приводит к дополнительной трате ресурсов процессора.

Список литературы

1. Zurbenko, I. (1986) The Spectral Analysis of Time Series, North-Holland Series in Statistics and Probability. Elsevier, Amsterdam.

2. Yang, W., Zurbenko, I. (2010). Nonstationarity. In WIREs Computational Statistics (Vol. 2, Issue 1, pp. 107–115). Wiley. https://doi.org/10.1002/wics.64

3. Zurbenko, I., Porter, P. S., Rao, S. T., Ku, J. Y., Gui, R., Eskridge, R. E. (1996). Detecting Discontinuities in Time Series of Upper-Air Data: Development and Demonstration of an Adaptive Filter Technique. Journal of Climate, 9(12), 3548–3560. http://www.jstor.org/stable/26201469

4. G Zurbenko, I., Sun, M. (2017). Applying Kolmogorov-Zurbenko Adaptive R-Software. In International Journal of Statistics and Probability (Vol. 6, Issue 5, p. 110). Canadian Center of Science and Education. https://doi.org/10.5539/ijsp.v6n5p110

5. Yang, W., Zurbenko, I. (2010). Kolmogorov–Zurbenko filters. In WIREs Computational Statistics (Vol. 2, Issue 3, pp. 340–351). Wiley. https://doi.org/10.1002/wics.71

6. Zurbenko, I. G., Smith, D. (2017). Kolmogorov–Zurbenko filters in spatiotemporal analysis. In WIREs Computational Statistics (Vol. 10, Issue 1). Wiley. https://doi.org/10.1002/wics.1419

Исходники

Ссылка на исходники: https://github.com/jakosv/kza