## 5-5 통계 기초의 기초

> ‘빅데이터’ 라고 하는 말이 화제가 되고 있지만, 수학은 이전부터 큰 데이터를 다루기 위해 노력을 해 왔고, 이것을 ‘통계’ 라고 부른다. 이번에는 통계 기초의 기초 부분을 Streem에서 어떻게 실현하는지에 대해 해설한다.

큰 소리로 떠들 것은 못되지만, 학생시절에 나는 수학을 잘 하지는 못하였다. 일반적으로 프로그래밍은 이과계의 활동이고, 따라서 이과 사람들은 수학을 잘 하게 되어 있다. 그런데, 내가 수학을 못한다고 하면 의외라고 생각되어지는 것 같다. 그리고 잘 못한다고 해도, 바닥은 아니라고 생각되어지는 것 같다.

하지만 실제로 수학은 정말 잼병이어서, 고등학교 시절에는 수학 III의 성적이 ‘1’ (10단계 중)을 받거나, 고3 1학기에 정기시험의 평균점이 16점을 받는 등 비참한 성적이었다. 대학입시에서도 입학후에도 수학은 상당히 괴로운 과목이었다.

생각해보면, 수학과 산수에서의 손으로 푸는 계산 대해서는(컴퓨터에 맡기는게 좋은 것들) 모티베이션이 생기지 않아 흥미를 잃은 것이 이런 현상까지 오게 되지 않았나 생각된다. 실수를 잘하는 사람에게 정확함을 필요로 하는 계산을 시키는 것 자체가 잘못되지 않았나 하고 지금도 생각하고있다.

물론 컴퓨터 과학은 수학을 기반으로 성립되고 있어, 수학과는 끊을래야 끊을 수 없는 관계이다. 하지만 프로그래밍의 전부가 수학이 필요한 것이 아니다. 프로그래밍활동의 대부분은 유저의 요구사항을 파악하는 것이기 때문에 수학과의 관계는 그렇게 강하지는 않다. 특히 내가 옛날부터 흥미가 있었던 프로그래밍 언어의 설계와 유저 인터페이스라고 하는 분야에서는, 수학을 쓸 일이 전혀 없다.

이전에는 “그래서 수학은 그렇게 필요는 없어” 라고 호언장담을 했었지만, Ruby를 개발중에는 수학이 중요하게 쓰여지는 국면을 몇 번 만났다. 커뮤니티의 멤버의 도움을 받으며(틀린 부분을 지적받으며), 조금씩 이는 진척을 시키고 있다. 그래도 나는 수학을 못한다는 생각은 떨칠 수가 없다.

자, 여담이 길어졌는데, 스트리밍 프로그래밍을 구현하는 Streem에 있어서, 아마 주요 적용분야가 될 것이라 예상되는 분야 중의 하나가 데이터 처리이다. 전형적인 예로서, 테스트의 성적 데이터를 CSV로 읽어 들여, 성적처리를 하는 작업이다. 이런 작업을 하게 하려면 수학을 못한다고 하면 안될 것 같다. 이번에는 통계적 데이터 처리의 기초의 기초에 대하여 같이 배워보도록 하자.

#### 합계와 평균

자, 우선 초등학교 레벨의 평균을 생각해 보자. 초등학교 딸에게 물어보니, 평균은 초등학교 5학년때 배우는 것 같다. 교과서에는 평균의 정의를

```

평균 = 총합 ÷ 개수

```

로 하고 있다. 평균은 각각 다른 값을 평균화 시키면, 어느정도 되는지를 보여주는 값이다. 중간시험이 14점이고 기말시험이 18점이면 평균은

```

(14 + 18) ÷ 2 = 16

```

16점이 된다. 5-4절까지의 시점에서 Streem에는 스트림의 데이터 개수를 구하는 count()와 스트림의 합계를 구하는 sum()을 제공하고 있어, 이들을 조합한다면 평균을 구하는 것은 간단하다. average()함수의 구현은 (그림 1)과 같이 된다.

```

struct avg\_data {

double sum; strm\_int num;

};

static int

iter\_avg(strm\_stream\* strm, strm\_value data) {

struct avg\_data\* d = strm->data;

d->sum += strm\_value\_flt(data);

d->num++;

return STRM\_OK;

}

static int

avg\_finish(strm\_stream\* strm, strm\_value data) {

struct avg\_data\* d = strm->data;

strm\_emit(strm, strm\_flt\_value(d->sum/d->num), NULL);

return STRM\_OK;

}

static int

exec\_avg(strm\_stream\* strm, int argc, strm\_value\* args, strm\_value\* ret, int avg) {

struct avg\_data\* d;

if (argc != 0) {

strm\_raise(strm, "wrong number of arguments");

return STRM\_NG;

}

d = malloc(sizeof(struct avg\_data));

if (!d) return STRM\_NG;

d->sum = 0;

d->num = 0;

\*ret = strm\_stream\_value(strm\_stream\_new(strm\_filter, iter\_avg, avg\_finish,

(void\*)d));

return strm\_ok;

}

```

<center>

(그림 1) average()의 구현

</center>

exec\_avg함수로 태스크를 만들고 요소별로 iter\_avg함수를 실행하여 합계를 구하여, 마지막에 avg\_finish함수로 합계를 개수로 나누어 평균치를 구한다. 처리가 스트림을 구성하는 태스크로 분할되어 있어 조금은 파악하기 어렵겠지만, 해 보면 심플한 평균치 계산이다.

#### 합계의 함정

평균의 계산은 간단하다. 하지만 초등학교 5학년 레벨이어서 당연하지 않을까. 실제 세계는 상당히 위험한 곳이어서 이런 간단하게 보이는 처리에도 함정이 숨어 있다.

앞서 설명한대로 평균을 구하는 방법은 합계를 구하고 그 개수로 나눈다. 하지만 이런 합계를 구하는 방법에 함정이 있다. 그것은 오차이다.

컴퓨터는 정확한 실수를 표현하는 것이 불가능하기 때문에, 근사치로서 부동소수점수를 이용해 계산하지만, 이것에는 ‘근사치’에서 오는 오차가 발생한다. 부동소수점수에는 오차 관련 2가지 ‘함정’이 있다.

하나는 사람이 볼 때 ‘버려도 되는’ 값이 있어도 컴퓨터에서는 그런 것이 없다는 점이다. 예를 들어 0.1은 상당히 심플하지만, 그 의미는 ‘1을 10으로 나눈 것’ 이어서, 부동소수점수가 채용되고 있는 2진수로는 나눌 수가 없다. 결국 어딘가를 잘라내지 않으면 안되고 여기서 오차가 발생한다.

또 하나의 함정은, 부동소수점수 끼리 계산을 반복하면 오차가 축적되기 쉬운 성질이 있다는 것이다. 수개의 값을 합계를 내는 것은 문제가 없지만, 이것이 수만개, 수천만개 요소의 합계라고 한다면 오차를 무시할 수는 없는 경우가 될 가능성이 있다. 예를 들어 (그림 2)의 프로그램은 같은 수의 1000만개의 평균을 구하는 것이다. repeat()는 첫번째 인자로 지정된 값을 두번째 인자로 지정한 횟수만큼 생성하는 스트림을 만드는 함수이다.

```

repeat(0.15,10000000) | average() | stdout # 실행결과

# 0.1499999999834609

```

<center>

(그림 2)오차가 발생한 평균

</center>

같은 수의 평균을 취하는 것이기 때문에 결과도 같은 값이 나와야 하는데, 실제로는 오차가 축적이 되어 미묘한 오류가 발생하고 있다.

합계같은 것은 단순한 덧셈이라고 생각할 지 모르지만, 오차를 생각한다면 상당히 귀찮은 작업이 될 수도 있다.

#### Kahan의 알고리즘

물론, 컴퓨터과학은 이런 사태를 회피할 방법을 생각해 놓고 있다.

오차를 좁혀 부동소수점수를 합계를 내는 알고리즘으로 Kahan의 알고리즘이 알려져 있다. Kahan의 알고리즘에서는, 덧셈에 의해 손실되는 하위 비트의 정보를 다음 회의 계산에 다시 넣음으로서 오차를 보상하는 원리이다. 위키피디아에 있는 슈도 코드는 (그림 3)[^1]과 같이 되어 있다.

```

function kahanSum(input)

var sum = 0.0

var c = 0.0 ← 처리중 손실된 하위 비트들의 보상용 변수

for i = 1 to input.length do

y = input[i] - c ← 문제가 없다면 c값은 0

t = sum + y ← sum이 크고 y가 작다면 y의 하위 비트에서

손실 발생

c = (t - sum) - y ← (t-sum)은 y의 상위 비트무리에 해당하기 때문에

y를 빼면 하위 비트 무리가 구해진다

sum = t ← 수학적으로는 c는 항상 0일것이다

적극적으로 최적화에 주의

next i ← 다음 반복시에는 y의 손실된 하위 비트무리가

고려된다.

return sum

```

<center>

(그림 3) Kahan의 알고리즘 (https://en.wikipedia.org/wiki/Kahan\_summation\_algorithm)

</center>

이 알고리즘을 사용해, (그림 1)의 구현을 바꾼 것이 (그림 4)와 같다. 변경이 필요한 것은 struct avg\_data와 iter\_avg함수뿐이다.

이에 따라 계산량이 조금 증가하지만, 오자를 좁히는 것은 가능하다. 이의 개선을 한 후, (그림 2)의 프로그램을 실행하면, 반복회수에 상관없이, 정확한 답이 얻어지는 것을 볼 수 있다.

이 건도 그렇지만, 부동소수점을 포함한 계산에는 오차가 항상 발생한다. 알고리즘을 선택하는 경우도 가능하면 오차를 고려하여 선택해야 할 것이다.

```

struct avg\_data {

double sum;

double c;

strm\_int num;

};

static int

iter\_avg(strm\_stream\* strm, strm\_value data) {

struct avg\_data\* d = strm->data;

double y = strm\_value\_flt(data) - d->c;

double t = d->sum + y;

d->c = (t - d->sum) - y;

d->sum = t;

d->num++;

return STRM\_OK;

}

```

<center>

(그림 4) average()의 개선. Kahan알고리즘을 반영하였다.

</center>

#### 평균과 분산

평균에 따라 복수의 값에 대한 전체의 경향 파악이 가능하지만, 어디까지나 전체의 ‘평균’ 한 값이기 때문에, 어떻게든 정보의 손실은 발생한다. 예를 들어 1반이 20명이고 두개의 A, B반이 100점 만점의 시험을 봤을 때, A반에서는 전원이 50점, B반에서는 10명이 100점 나머지 10명이 0점이면, 둘 다 반평균은 50점이 된다.

A반과 B반에서는 성적의 경향은 완전히 다르지만, 평균에서 보면 구별이 되지 않는다. 이와 같은 차이를 검출하는데는 평균과는 별도로, 값의 분산현황을 파악할 필요가 있다. 값의 분산 현황은 ‘표준편차’를 통해 표현한다. X1, X2, ... Xi의 평균을 μ라고 할 때, 표준편차는 (그림 5)의 식에 의해 정의되는 ‘분산’의 평방근 σ 이다.

![](pic5-5-5.png)

<center>

(그림 5) 분산(표준편차의 제곱)의 정의

</center>

수학에 익숙하지 않은(나같은) 사람들에게는 어려워 보이는 식이지만, 요약하면 개개의 값과 평균의 차를 제곱하여 합한 값을 개수로 나눈 것이다.

이를 앞의 2개반 성적에 적용을 해 보면 A반은 전원이 같은 점수로서 편차가 없기 때문에 표준편차가 0, B반은 절반은 만점, 또 절반은 0점이므로 표준편차는 51.3이 된다. 같은 평균 50점이라도 성질이 다른 것을 알 수 있다.

#### 스트리밍 알고리즘

(그림 5)에서의 정의를 보면, 표준편차를 구하기 위해서는 우선 평균을 구하고, 각각의 값과의 차를 계산할 필요가 있다. 결국, 아무 생각없이 구현하면, 평균의 계산을 위해 전부 값을 읽어들이고, 그 후 표준편차를 위해 또 한번 같은 값을 처음부터 다시 읽어들일 필요가 있다. 예를 들어 (그림 6)의 프로그램은 루프를 2회 사용하여 표준편차를 계산한다(오쿠무라 하루히코(奥村晴彦)의 『C언어로된 최신 알고리즘 사전』p.254로부터 인용)

```

int i, n;

float x, s1, s2;

static float a[NMAX];

s1 = s2 = n = 0;

while (scanf("%f" &x) == 1) {

/\* 첫번째 루프 \*/

if (n >= NMAX) return EXIT\_FAILURE;

a[n++] = x; s1 += x;

}

s1 /= n; /\* 평균 \*/

for (i=0; i<n; i++) { /\* 두번째 루프 \*/

x = a[i] - s1;

s2 += x \* x;

}

s2 = sqrt(s2/(n-1)); /\* 표준편차 \*/

printf("개수:%d 평균:%d 표준편차: %g", n, s1, s2);

```

<center>

(그림 6) 표준편차의 계산

</center>

하지만, 같은 데이터를 여러 번 읽어들이는 것은 낭비이며, 특히 스트리밍 처리의 경우에는 처리 중의 (초 거대 데이터일지도 모르는) 데이터를 어딘가에 보존해야 할 것 같다(메모리를 써야 한다). 이런 낭비를 피하기 위해, 데이터를 하나씩 읽어 들이는 것만으로도 처리가능한 알고리즘을 ‘스트리밍 알고리즘’이라고 부른다. 조사해 보면, 표준편차의 계산에도 스트리밍 알고리즘이 존재하고 있었다.

앞에서 인용한 『C언어로된 최신 알고리즘 사전』에 의하면, (그림 7)의 프로그램과 같은 계산을 하면, 데이터를 한번 읽어들이는 것만으로, 이미 오차를 비교적 좁히면서 표준편차를 계산할 수 있다. 이 알고리즘을 사용하여, 분산과 표준편차를 계산하는 함수(stdev()와 variance())를 Streem에 추가했다. 지면상, 여기에 코드를 싣지는 않겠지만, 구조로는 (그림 1)의 average()함수와 동일하다. 소스코드에서는 stat.c의 exec\_stdev()함수쪽을 찾아보면 될 것이다.

#### Streem에서의 표준편차 계산

그러면, 이 새롭게 정의한 stdev()함수를 사용해 앞의 예제의 표준편차를 계산해 보자. 우선은 계산의 기초가 되는 전원이 50점을 맞은 A반의 성적을 생성하는 방법을 생각해 보자.

20명 전원이 50점을 맞았기 때문에, 50이 20개 연속하는 값을 생성하면 되는 것이다. 이를 위헤 위에서도 설명한 repeat()를 사용한다.

```

repeat(50,20)|stdout

```

이렇게 하면 20개 중 50이 표준출력으로 출력된다. 평균을 구하기 위해서는

```

repeat(50,20)|average()|stdout

```

로 한다. 간단하지 않은가.

B반쪽이, 100점이 10명, 0점이 10명이기때문에, 두개의 스트림을 결합하도록 하자. 100을 10개, 이어서 0을 10개를 늘여 세운 스트림을 얻기 위해서는 repeat()와 concat()을 조합하여, (그림 8)과 같이 기술이 가능하다. 그래도 예제의 반과 같은 전원이 같은 점수라든가, 절반이 만점, 절반이 0점과 같은 일은 현실에서는 일어나기 매우 힘들다. 테스트의 득점과 같은 작위적이지 않은 값은, 평균 주변이 가장 많고, 평균으로부터 멀어짐에 따라, 값이 작아지는 경향이 있어, 우리들은 경험적으로 이를 알고 있다.

```

concat(repeat(100,10),repeat(0,10))|stdev()|stdout

# 51.29891760425771

```

<center>

(그림 8) B반의 성적 생성과 표준편차

</center>

#### 편차값

평균과 표준편차는 스트리밍 알고리즘으로 계산이 가능했지만, 아무리 해도 스트리밍 알고리즘으로 계산이 불가능한 지표도 있다.

예를 들어, 성적처리 본래의 순위와 편차값은 둘다 스트리밍으로는 계산이 불가능하다. 순위의 계산을 위해서는 성적순의 정렬이 필요하고, 편차값의 계산을 위해서는 미리 평균과 표준편차를 계산해 놓을 필요가 있다.

여기서 예제 로서 편차값을 계산해 보자. 편차값의 정의는

```

편차값 = (득점-평균)×10/표준편차 + 50

```

이다. 이것으로 편차값을 계산하는 프로그램을 (그림 9)에 나타내었다.

```

input = fread("result.csv")|map{x->number(x)} avs = input | average() # 평균

sts = input | stdev() # 표준편차

zip(avs, sts) | each{ x ->

avg = x(0); std = x(1)

fread("result.csv") | map{x->number(x)}| each { score ->

ss = (score-avg)\*10 / std + 50

print("득점: ", score, "편차값:", ss)

}

}

```

<center>

(그림 9) 편차값의 계산

</center>

zip을 사용하는 부분이 좀 알기 어려운 부분인 것 같아 설명을 하면, 같은 입력 스트림으로부터 평균과 표준편차를 구했다. average()도 stdev()도 하나의 인수가 주어지는 스트림을 돌려준다. 이 두개의 스트림에 zip()함수를 적용하면, ‘요소를 추출하여 배열로 정리’ 하는 일을 한다.

그래서 계속 each()는 루프가 아닌, 추출한 값에 대해 처리만 하는 작용을 한다.

한번 데이터를 읽어 들여 평균과 표준편차를 구한 후, 또 한 번 데이터를 다시 가져와야 한다는 것이 아무래도 모양이 좋지 않다. 실제로 읽어들이는 것은 피할 수는 없지만, 좀 더 개선된 지정방법이 없을 까 조금 생각해 보자. future이라든가, promise를 사용하면, 어떻게든 개선이 가능할 것 같다. 이는 숙제로 남겨 놓기로 하자.

#### 정렬(sort)

데이터를 그 값의 크고 작음에 따라 재배열하는 정렬은 컴퓨터 과학에서는 중요한 토픽으로서, 속도를 높이기 위해 여러가지 알고리즘이 고안되고 있다. 현 시점에서 ‘가장 빠른’ 것으로 알려진 것이 그 이름도 ‘quick sort’라고 하는 알고리즘이다.

C에서는 표준 라이브러리에 qsort(3)이라는 함수가 제공되고 있어, 메모리상의 데이터를 간단히 정렬을 할 수 있다. 이는 나쁘지는 않지만, 한가지 중요한 문제가 있다. 바로 메모리에 넣을 정도 크기의 데이터 밖에 정렬이 안된다는 점이다.

이 문제는 ‘외부 병합 정렬’이라고 하는 기법에 의해 일단 해결 가능하다. ‘외부 병합 정렬’은 메모리에 넣을 수 있을 정도의 데이터를 분할하여 읽어 들인 후, 각각을 정렬하고 파일에 출력한다. 그리고 개별 정렬 파일을 앞에서부터 읽어 들여, 이를 이어서 전체의 정렬을 완성시킨다. 처리 순서는 아래와 같다.

1. 원 데이터 세트로부터 메인 메모리에 넣을 수 있는 사이즈의 데이터를 읽어들인다.

2. 읽어 들인 데이터를 정렬하고 파일로 출력한다.

3. 모든 데이터를 처리할 때 까지 1과 2를 반복한다.

4. 작성된 여러개의 파일로부터 각각의 1번째 데이터를 읽어들인다.

5. 가장 작은 데이터를 결과 파일에 작성한다.

6. 5를 실행한 파일의 경우는 그 다음 요소를 읽어들여, 5를 반복하여 실행한다. 이 작업을 모든 데이터가 처리될 때 까지 반복한다.

UNIX의 sort 명령어도 이 알고리즘을 써서 정렬을 한다.

#### 대규모 정렬은 어렵다

그런데 이 ‘외부 병합 정렬’에도 몇가지 문제가 있다. 첫번째 문제는 작업 파일을 만들기 때문에 디스크 공간을 소비한다는 점이다. 데이터의 규모가 커질 경우는 디스크용량을 생각하지 않을 수 없다. 외부 병합 정령이 필요한 상황은, 필연적으로 큰 데이터를 취급하는 경우여서, 이런 우려는 더 커진다. 이는 적절한 디스크 배치를 염두하면서 작업하는 수 밖에 다른 대책이 없을 것 같다. [^2]

또 하나는, Streem에서는 임의의 데이터를 스트림에 실어 보낼 수 있어, 정렬의 대상이 될 수 있다는 점이다. 결국, 단순한 수치와 문자열이라면 작업 파일로의 출력은 간단하지만, 구조가 있는 데이터는 정보를 잃어버리지 않고 파일에 출력하는 것은 좀 어렵다. 이 점에 대해서는 JSON과 MessagePack과 같은 구조 데이터를 표현가능한 방법으로 출력하는 정도로 대응이 가능할 것 같다.

이와 같이 데이터 규모가 커지면, 여러가지 고려해야 할 것들도 늘어난다는 것을 알 수 있다.

여기까지 대단한 것들을 설명했지만, 우선 빨리 작업을 들어가려고 하기 때문에, 이번에는 메모리에서 정렬을 하는 것으로 하겠다. 외부 병합 정렬에 의한 대규모 데이터로의 대응은 이후의 과제로 남기도록 하자.

```

input | sort() | output

```

이라는 파이프라인으로 데이터 정렬을 한다. 현재 구현에서는 input으로부터 데이터를 한번에 전부 메모리로 올려놓고 정렬을 수행하고, 그 결과를 한번에 output으로 출력한다.

정렬하는 데이터가 모두 수치라면 자연스럽게 정렬은 가능하지만, 예를 들어 각 데이터가 배열로 되어 있고, 그 n번째 요소를 기준으로 정렬하려고 하는 경우도 있을 것이다. 이런 경우에는 함수를 지정한다.

결국, 배열을 첫번째 요소(인덱스는 0으로부터 시작하기 때문에 실제는 2번째)로 정렬하기 위해서는

```

sort{x,y->cmp(x(1),y(1))}

```

와 같이 비교 함수를 지정한다.

#### 정렬의 응용

데이터의 정렬이 가능하면, 통계적인 의미가 있는 값을 취할 수가 있다. 가장 알기 쉬운 것이 순위이다. 순위를 얻는다는 것은 성적순으로 정렬하는 것과 같은 의미이다.

또한, 정렬을 하면 ‘중앙값’을 취하는 것도 가능하다. 이것은 정렬한 데이터에서 가운데 요소의 값이다. 데이터 총 수가 짝수인 경우는 중앙의 값이 없기 때문에 중앙에 가까운 2개의 값의 평균을 중앙값으로 취한다. 중앙값은 median으로 구한다.

중앙값은 평균치와 비슷하지만, ‘벗어난 값’에 의한 영향이 작다는 점이 특징이다. 평균은 모든 값의 영향을 받기 때문에, 측정의 실수 등으로 벗어난 값(다른 값보다 현저히 다른 값)이 있는 경우, 오차가 커져 버릴 위험성이 있다. 하지만 중앙값은 벗어난 값의 영향을 거의 받지 않는다.

#### 샘플링

‘빅 데이터’라는 말이 유행하고 있지만, 실제로 그렇게 규모가 큰 데이터를 취급하는 것은 쉽지 않다. 데이터 처리의 비용도 비용이지만, 처음부터 데이터를 모으는 것도 그렇게 만만한 작업이 아니다.

원래 ‘통계’ 라는 학문은, 실 데이터를 모으기 어려운 ‘빅 데이터’를 추정하기 위한 수단으로서 탄생하였다. 모집단이 어느 정도 크다면 의외일 정도로 적은 샘플로 전체의 경향을 파악할 수 있다. ±5% 오차 허용 범위에서, 모집단이 10만명인 경우, 경향을 파악하기 위해 필요한 표본 수는, 약 383명이다.

Streem에서도 데이터를 쉽게 취급하기 위한 샘플링 함수를 구현해 보자. 샘플링을 위한 스트리밍 알고리즘으로서 ‘레저보어 샘플링(Reservoir Sampling)’이 있다.

레저보어 샘플링은 아래와 같은 수서로 샘플링을 한다. N개의 샘플을 취하기 위해서는

1. 처음 N개의 샘플을 배열로 등록한다.

2. 그 이후의 i번째 요소에 대해 0부터 i-1까지의 난수 r을 생성한다.

3. 난수r이 N보다도 작을 경우, 테이블의 r번째를 i번째 요소로 치환한다.

‘레저보어(reservoir)’란 저수지의 의미이다. 처음 N개의 요소로 저수지를 가득 채우고, 상류로부터 요소가 흘러 들어 올 때, counter/size의 확률로 랜덤 하게 일부를 치환해 간다. 저수지의 요소가 새로운 요소로 치환되어가면서, 마지막으로는 모집단 전체로부터 샘플을 취할 수 있게 된다.

말로 설명하기보다는 코드를 보면서 이해하는게 더 빠른 사람들을 위해 (그림 10)에 레저보어 샘플링을 Ruby로 기술한것을 나타내었다. 이 알고리즘을 이용해서 샘플링을 하는 sample() 함수를 구현하였다.

```

def reservoir\_sampling(seq, k)

e = seq.to\_enum

reservoir = e.take(k)

n=k

e.each do |item|

r = rand(n)

n += 1

if r < k

reservoir[r] = item

end

end

return reservoir

end

# 호출

print reservoir\_sampling(0..1000000, 10)

```

<center>

(그림 10) Ruby에서의 레저보어 샘플링

</center>

예를들어, 파이프라인에 ‘sample(100)’을 자르면, 스트림 전체로부터 100개의 요소를 랜덤으로 선택하여 하위로 보낸다. 모집단이 충분히 클 경우, 전체 데이터의 경향을 유지하면서 데이터를 추출하는 것이 가능하다.

앞에서도 기술한 바와 같이, 5%정도의 오차는 허용한다고 하면, 큰 모집단에서도 놀라울 정도로 작은 수의 샘플로부터 경향을 파악하는 게 가능하다. 그래도, 모집단의 크기를 알 수 없는 단계에서 억지로 데이터를 추출하면 올바르지 못한 결과를 얻을 수 있기 때문에 주의가 필요하다.

#### 마치며

‘빅데이터’가 주목받고 있는 요즘, 통계는 점점 중요성이 커지고 있다. Streem이 언젠가 Excel정도로 간단히 쓸 수 있도록 통계분석 툴이 될 때 까지 성장하면 좋겠다는 생각이 든다.

<hr>

### 타임머신 칼럼

\*\*\*내가 못하는 분야는 다른 사람에게 맡기고 싶지만..\*\*\*

> 2016년도 7월호 게재분이다. 몇가지의 통계함수를 도입하였다.

>

> 본문중에서도 고백했지만, 나는 수학에 대해서 상당히 잼병이라는 의식을 가지고 있다. 따라서, 이번의 원고의 집필시에도 상당히 힘들었다. 초등학생인 딸의 교과서를 빌려 복습을 하고, 눈물을 머금으며 집필을 하였다.

>

> 컨커런트 프로그래밍에서도, 수학처리에서도 나 자신은 어려움을 겪고 싶지 않아, 이를 위한 도구를 원했었다. 하지만, 내가 원하는 도구는 존재하지 않았기 때문에 직접 만들 수 밖에 없었고, 직접 만들기 위해서는 마치 잘 하지 못하는 문제에 정면으로 대치하지 않으면 안되는 모순이 있었다. 이런 문제에 익숙한 사람과 같은 팀을 만들어 개발을 한다면 좋겠지만, 이런 서로의 결점을 보완해 줄 수 있는 동료를 만난다는게 그리 간단한 일은 아니다. 내 커뮤니케이션 장애가 문제였을까?

[^1]: 원문의 출전은 「https://ja.wikipedia.org/wiki/カハンの加算アルゴリズム」으로 되어 있으나, 영문 위키피디아로 변경하였다(옮긴이)

[^2]: 하지만, 영어판 위키피디아는 외부 병합 정렬을 in-place실행에 의해, 필요한 디스크 용량을 원 데이터와 같은 정도로 제한할 수 있다고 적혀 있다. 하지만, 이 부분에서는 [출전필요] 마크가 붙어 있다.