Bloom filter

Bloom filter는 다시 정리해서 얘기해보자면 어떤 값이 집합에 속해 있는가?를 검사하는 필터 혹은 이를 구성하는 자료형이라고 합니다.

Bloom filter의 false positive값이 의도한대로 잘 나오는지 확인해보기 위해서 동물 데이터를 bloom filter로 만들고 여러가지 단어를 filter에 넣고 값이 의도한대로 나오는지 확인해 보았습니다.

```
C:₩Users₩hyeondin₩anaconda3₩python.exe C:/Users/hyeondin/Desktop/빅최기/bloom.py
false positive =  0.1
오판한 경우 : 9  총 데이터의 수 : 48  데이터를 바탕으로 계산한 false positive : 0.1875


C:₩Users₩hyeondin₩anaconda3₩python.exe C:/Users/hyeondin/Desktop/빅최기/bloom.py
false positive =  0.1
오판한 경우 : 12  총 데이터의 수 : 48  데이터를 바탕으로 계산한 false positive : 0.25


C:₩Users₩hyeondin₩anaconda3₩python.exe C:/Users/hyeondin/Desktop/빅최기/bloom.py
false positive =  0.5
오판한 경우 : 23  총 데이터의 수 : 48  데이터를 바탕으로 계산한 false positive : 0.4791666666666667

Process finished with exit code 0

C:₩Users₩hyeondin₩anaconda3₩python.exe C:/Users/hyeondin/Desktop/빅최기/bloom.py
false positive =  0.5
오판한 경우 : 31  총 데이터의 수 : 48  데이터를 바탕으로 계산한 false positive : 0.6458333333333334

Process finished with exit code 0
```

나온 결과값을 보면 어느정도 의도한 false positive값과 비슷하게 나온다는 것을 알 수 있습니다.

문제.

1억명의 사용자 계정이 시스템에 저장되어있고, 사용자가 회원가입 중에 동일한 계정명이 서버에 존재하는지 즉각 확인해주는 시스템을 개발하려고 할 때, bloom filter를 어떻게 사용하면 좋을까?

사용방법.

우선 정상적으로 만들어진 계정명을 보관한 집합을 만들고, 새롭게 만들 계정명을 bloom filter에 넣게 되면 false positive 성격 때문에 존재하지 않는 계정명이 존재한다고 할 수 있겠지만 존재하는 계정명에 대해서는 100%의 확률로 알아낼 수 있습니다.

여기서 생각해봐야 할 점은, 존재하지 않는 계정명을 존재한다고 오판하는 경우를 어떻게 줄일수 있고 가장 효율적인 비트배열의 크기와 false positive의 값을 구하는것 인 것 같습니다.

n=집합의 크기 m = bit사이즈 k=해쉬함수의 개수라 라고 할 때,

$$False\ Positive\ rate \approx \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^{k} \approx \left(1 - e^{-\frac{kn}{m}}\right)^{k}$$

$$Optimal\ \#\ of\ hash\ functions\ (k) \approx \ln 2\ \frac{m}{n} \approx 0.7\frac{\mathrm{m}}{\mathrm{n}}$$

라고 할 수 있습니다. 이를 바탕으로 생각해보면 비트배열의 크기가 커지면 커질수록 false positive의 값은 낮아지게 되어 오 탐지할 확률이 줄어들게 됩니다.

따라서 1억명의 사용자 계정이 저장된 시스템에서 만약 메모리의 여유가 있어서 오 탐지율을 낮추고 싶으면 bit배열의 크기를 늘려서 해쉬 함수의 개수를 늘리고 이를 바탕으로 false positive값을 낮추고 만약 메모리의 여유는 없고 그냥 서버에 존재하는지 확인만 하는 역할을 위한 시스템을 만들고 싶다면 bit배열의 크기를 낮추도록 합니다.

Bloom Filter를 사용하면 좋을 상황

비밀번호 검사, 의심스러운 URL, 스팸번호 검사 등에 사용하면 좋을 것 같다고 생각합니다. 그렇게 생각한 이유는 이런곳에서는 데이터의 양이 많아서 정확히 알아내기에는 메모리 액세스가 많아져 처리하는 시간이 길어지게 됩니다. 하지만 Bloom filter를 사용하게 되면, 의심스러운 항목들을 false positive 특성을 통해서 모두 모을수 있고 그 후에 모은 항목들을 다시 정확하게 검사함으로써 시간과 메모리 사용을 절약 할 수 있게됩니다.

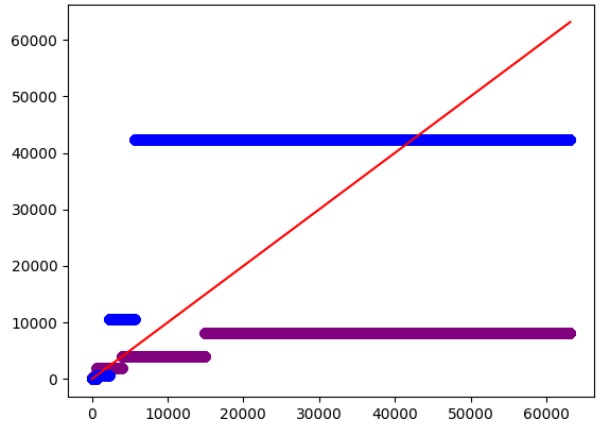
Flajolet-Martin

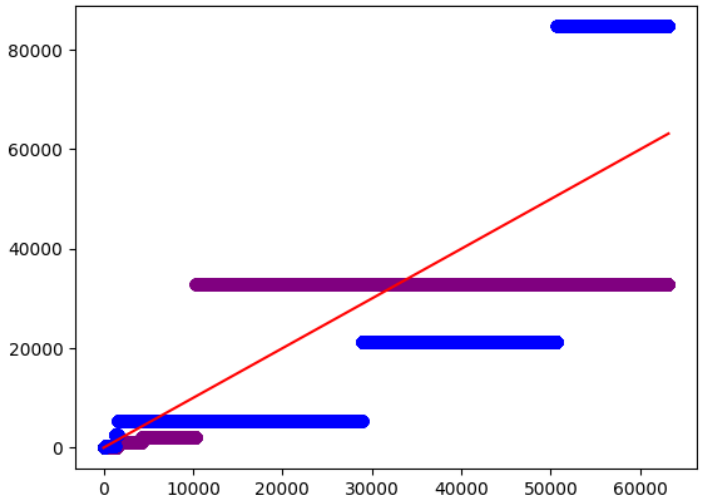
Flajolet-Martin version1의 방식은 지금까지 등장한 모든 a에 대해서 r(a)의 최대값 R을 구하는 것입니다. 그렇게 되면 2**R의 서로 다른 아이템의 수가 등장하게 됩니다. 이렇게 되면 생기는 문제점은 r(a)의 값이 우연찮게 너무 커지면 R값이 커져서 서로다른 아이템의 수가 갑자기 급상승하게 됩니다.

이를 해결하기 위해서 version2의 방식을 사용하였습니다. Version2의 방식은 아이템이 들어올 때마다 비트 배열 B를 업데이트 하는것입니다. 들어온 아이템 a에 대해서, B[r(a)]를 1로 설정 합니다. 그렇게 되면 서로 다른 아이템의 수 2**R/ϕ이 되기 때문에 version1에 비해서 오류를 줄일 수 있게 됩니다.

그래프를 이용해서 비교해보면 다음과 같습니다.

빨간색 : 그룹의 값　　보라색 : version1 예측　　파란색 : version2 예측





그래프를 보면 알 수 있듯이, version1에 비해서 version2가 더 잘 예측한다는 것을 알 수 있습니다.

version2의 방식이 version1의 방식보다는 정확도가 더 높지만 그래도 아직 오차가 심하다는 것을 알 수 있습니다. 이를 해결하기 위해서 hash함수를 여러개 사용하여 예측 값의 정확도를 높이려고 합니다. 이때 hash함수가 여러개 나오기 때문에 예측값이 여러 개 나오게 되는데, 예측값은 중앙값 평균내기를 이용하여 구하려고 합니다.

중앙값 평균내기란, 여러 예측값을 그룹으로 나누고 각 그룹별로 중앙값을 구하고 중앙 값들을 평균내는 방법입니다.

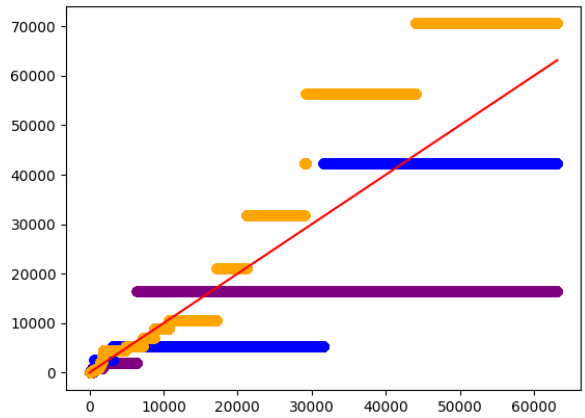
>그룹의 수는 고정하고 해쉬함수의 개수를 바꿔가며 실행

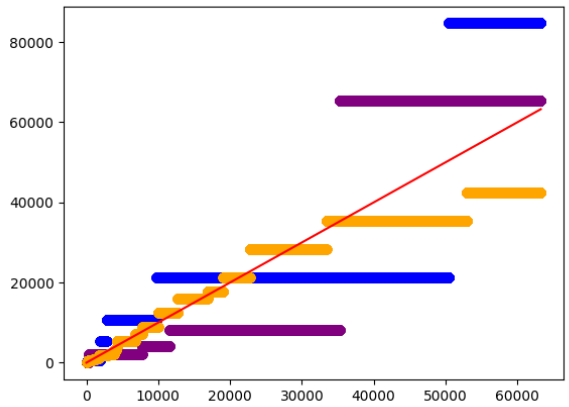
해쉬함수의 수 = 12개 - 그룹의 수 = 3개의 경우.

```
for j in range(0,9):
    fm3.put3(item,j)
    if j%3 == 0:
        m1.append(fm3.size3(j))
    elif j%3==1:
        m2.append(fm3.size3(j))
    else:
        m3.append(fm3.size3(j))
avr = (statistics.median(m1)+statistics.median(m2)+statistics.median(m3))/3
```

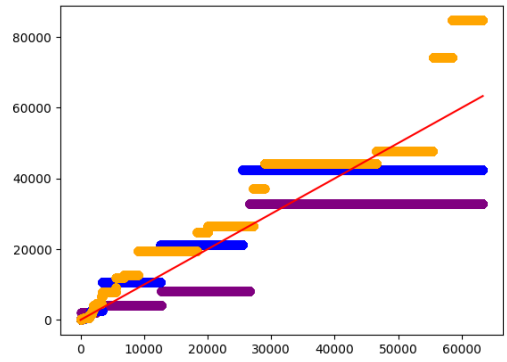
위의 코드처럼 그룹은 나머지 값이 같은 것끼리 묶었고, statistics 라는 라이브러리 함수를 이용하여 중앙값을 구한후, 값들의 평균을 구하였습니다.

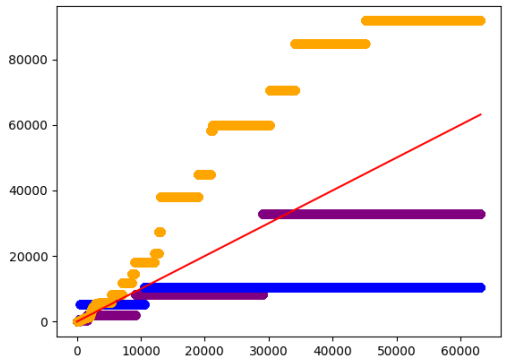
빨간색:그룹의 값 보라색:version1 예측 파란색:version2 예측 주황색:version2-hash함수 늘리기



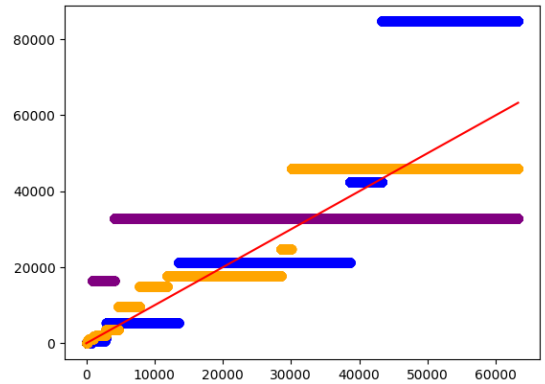


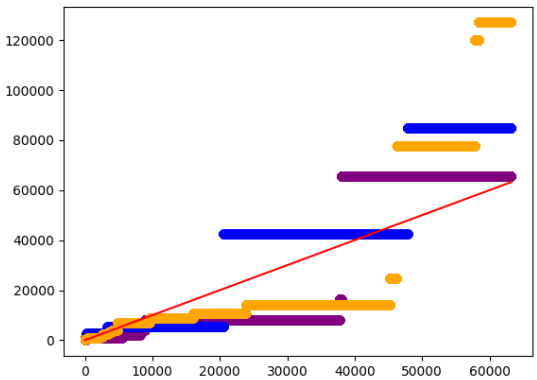
해쉬함수의 수 6개-그룹의 수 3개의 경우를 보겠습니다.





해쉬함수의 수 3개-그룹의 수 3개의 경우를 보겠습니다.

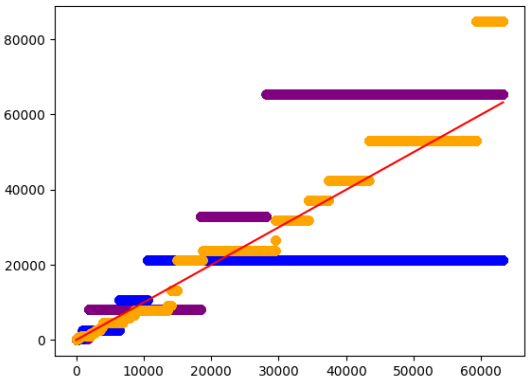


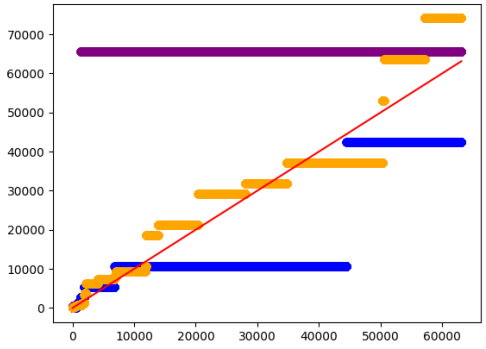


이렇게 그룹의 수는 3개로 고정시키고 해쉬함수의 개수를 바꿔가면서 그래프를 그려본 결과 해쉬함수의 개수가 많으면 많을수록 예측을 더욱 정확하게 한다는 것을 알 수 있었습니다. 대신에 메모리접근이 더욱 늘어나기 때문에 실행시간이 좀 더 걸린다는 단점을 또한 발견 할 수 있었습니다.

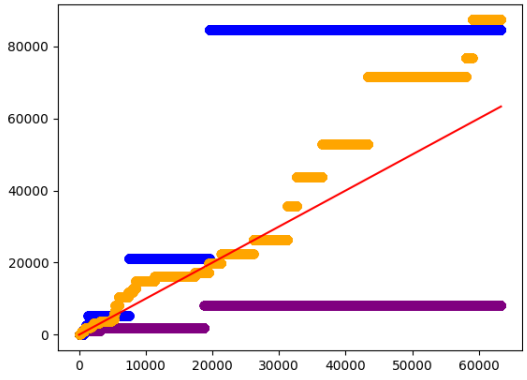
>해쉬함수의 개수는 고정하고 그룹의 수를 바꿔가며 실행

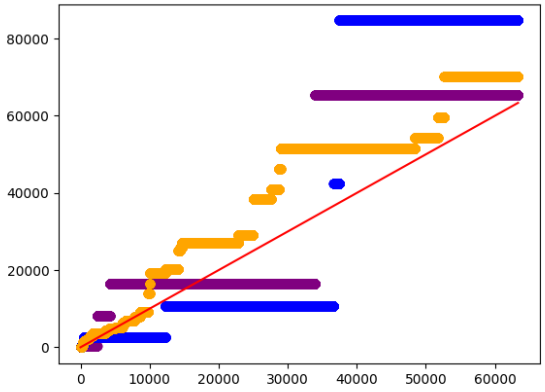
해쉬함수의 개수 8개 - 그룹의 수 2개의 경우입니다.



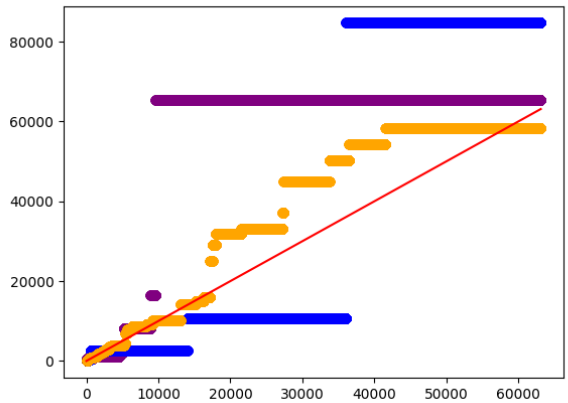


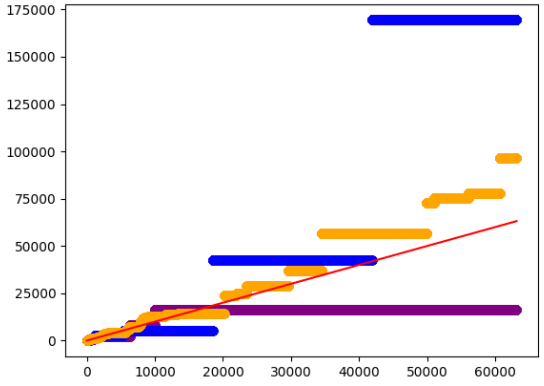
해쉬함수의 개수 8개 – 그룹의 수 4개의 경우입니다.





해쉬함수의 개수 8개 – 그룹의 수 8개의 경우입니다.





이렇게 해쉬함수의 개수는 고정하고 그룹의 수를 바꿔가면서 실행해본 결과 그룹을 너무 많이 나눈 것보다 적당히 나누는게 더 정확도가 높게 나타난다는걸 알 수 있었습니다. 왜 이렇게 나오는지 생각을 해봤는데, 그룹을 많이 나눈게 되면 결국에는 중앙값보다 평균값에 더 편향되기 때문에 outlier의 취약해서 그런게 아닌가하고 생각해봤습니다.

알게된 점.

결국에는 정확도와 실행시간 사이의 trade-off가 발생하기 때문에, 어떤 목적으로 만드냐에 따라서 해쉬함수의 개수와 그룹의 수가 정해진다는 것을 과제를 하면서 깨닫게 되었고, 정확도를 높이기 위해서는 해쉬함수를 많이 사용하면 되지만 그만큼 실행시간이 오래걸리게 된다는 점도 알게되었습니다.