BACHELOR

Frequenties bijhouden met beperkte ruimte

Groot Bruinderink, L.

Award date:
2013

Frequenties bijhouden met beperkte ruimte

Leon Groot Bruinderink
StudentID: 0682427
Bachelorproject Technische Wiskunde
TU Eindhoven
Begeleider: Berry Schoenmakers
Tweede begeleider: Benne de Weger
Opdrachtgever: Coosto/WiseGuys b.v. te Eindhoven

13 September 2013

Summary

This project is done at Coosto, a company specialized in monitoring social media, like Twitter and Facebook. Many sources are visited on a daily basis and news, comments or blogs are registered in multiple databases. Customers can search these databases for information they find important, for example they can search for opinions about a new product they're selling. An important aspect in social media are trending topics: topics people are discussing more than usual in some period. To determine these trending topics, one needs to know the frequencies of the words being searched for. Since Coosto uses multiple databases, and topics are registered in multiple databases, the frequencies of all relevant words in all databases needs to be combined. This would take too much space, so instead of doing it precise one wants to approximate the frequency of words, using a small database. One way to do it is using a method called "sketch". A sketch is a small databases, where all frequencies of all words are registered. When all words are registered, one can search for the frequency in this sketch, but errors will arise.

Several methods for determining the frequencies are compared by analysis and simulation, all with different properties and errors. The method most used in practice, the Count-min sketch, is always overestimating the frequency. An interesting improvement uses an unbiased estimator for the frequency, but has serious worst-case scenarios. A new method is also introduced in this paper and determines an overestimation of the error and uses a different manner of constructing the sketch: the count sketch with collision reduction.

An important conclusion is that different datasets require different methods to work best. For Coosto there is one important assumption of their dataset: frequencies of words are often Zipfian distributed. For this dataset, the new method seems to work best. To really test if this method works best, one would have to test it with real data.

Inhoudsopgave

1 Inleiding

Coosto monitort social media waaronder Twitter, Hyves, Facebook, maar ook vele andere blogs en fora. Deze bronnen wordt dagelijks bezocht en geïndexeerd, wat zorgt voor gigantische databases. Klanten kunnen hieruit informatie opvragen en filters toepassen, welke continu ge-update kunnen worden. Een belangrijke functie is de trending topics: hoe vaak wordt een woord of groep woorden gezegd in een bepaalde periode. Coosto gebruikt voor het monitoren meerdere databases en als al deze frequenties van woorden van alle databases gecombineerd moeten worden, zal dat erg veel ruimte gaan vragen. Ook is de kans groot dat veel woorden of woordgroepen maar één keer voor komen. Een manier om dit te doen, is de grote hoeveelheid aan data verkleinen naar een zogenaamde "sketch", een database met gelimiteerde ruimte. Uiteraard kan je niet alle frequenties per woord tellen, maar de werkelijke waarden van de oorspronkelijke data moeten weer zo goed mogelijk gereconstrueerd kunnen worden met deze sketch. Gevraagd wordt om een algoritme, die zo goed mogelijk omgaat met de beperkte ruimte en een zo klein mogelijke fout produceert bij het construeren van de originele data.

2 Analyse van het probleem

In de uitwerking van dit probleem gaan we ons beperken tot key-value paren, waarbij de key een woord is en de value een positief, geheel getal. De manier van het construeren van de sketch, zal hetzelfde zijn als wat in de praktijk het meest gebruikt wordt. We zullen hashfuncties gebruiken om een key naar een bepaalde waarde in de sketch te laten wijzen. Omdat de sketch minder ruimte heeft dan de keyruimte, zullen er meerdere hash-collisions voorkomen. Het eerste probleem is dan ook om een algoritme te bedenken die de sketch zelf maakt: hoe groot moet ik mijn sketch maken om een fout niet groot te krijgen. Daarna moeten de originele waarden zo goed mogelijk gereconstrueerd kunnen worden uit deze sketch: welke values te gebruiken en hoe, zodat de fout die gemaakt wordt zo klein mogelijk is.

2.1 Aannames en Wiskundig Model

In de uitwerking en analyse zullen we gebruikmaken van de Count-sketch, de meest gebruikte sketch in praktijk en literatuur. Deze sketch is in feite een rij van hash functies, waarbij een element voor iedere rij wordt gehashed naar een bepaalde kolom in de sketch, afhankelijk van zijn key. Ook is deze sketch lineair, in de zin dat je meerdere sketches simpel bij elkaar kunt optellen. In praktijk is dit erg handig, omdat je vaak met meerdere databases te maken hebt. We introduceren nu de volgende variabelen:

Vector $U = (u_1, u_2, ..., u_M)$ beschrijft de lijst van key-value paren met $u_i = (k_i, v_i)$, waarbij de key $k_i \in K \subset \mathbb{Z}$ de numerieke waarde van key k is (een hashwaarde), met K de keyruimte van de keys die gebruikt wordt voor de sketch. $v_i \in \mathbb{N}$ is de frequentie-toename van de key.

Totale frequentie De totale frequentie $T(k) = \sum_{i=0}^{M} v_i\{k == k_i\}$ van een woord in een bepaalde periode moet zo goed mogelijk gereconstrueerd kunnen worden met behulp van de sketch. We nemen dus aan dat deze $T(k)$ Zipf-verdeeld zijn.

Sketch $S \in \mathbb{N}^{d \times w}$ is een $d \times w$ matrix, waarbij $d \cdot w \ll |K|$, en beschrijft de beschikbare ruimte van de sketch. Na elk nieuw paar u_i word S geupdate, afhankelijk van k_i en v_i.

Hashfunctie H_i beschrijft de familie van onafhankelijke hashfuncties die elementen in S update, afhankelijk van het key-value paar u_j. Voor elke $i = 1, ..., d$ update we: $S_{i,H_i(k_j)}$ met v_j.

Requestfunctie $R : K \to \mathbb{N}$ beschrijft de functie die, gegeven een key k, de waarde van $T(k)$ zo goed mogelijk moet benaderen met behulp van de sketch S.

Totale fout $\Delta_{tot} = \sum_{k \in K} |T(k) - R(k)|$ is de som van alle gemaakte fouten.

Maximale fout $\Delta_{max} = \max_{k \in K} |T(k) - R(k)|$ is de maximale fout die gemaakt wordt. Deze willen we graag afschatten.

Er is veel onderzoek gedaan naar de verdeling van woorden die voor komen in een stuk tekst. Het is gebleken dat de frequenties van woorden in een stuk tekst vaak een zogenaamde Zipf-verdeling volgen**(Zie referenties: Rf[1])**. Deze verdeling neemt aan dat er één woord is met de hoogste frequentie, dat de een-na-hoogste frequentie twee keer zo klein is als de hoogste, de twee-na-hoogste drie keer zo klein, etc. Dus de frequenties van de Zipf-verdeling volgen de lijn: $f(i) = \lceil \frac{\max_k T(k)}{i} \rceil$ voor het i-de woord in aflopende grootte. We nemen aan dat de frequenties $T(k)$ dus zo verdeeld zijn.
Er zijn nu een aantal belangrijke doelen, die we gaan gebruiken om de sterkte van sketches te vergelijken:

1. We willen graag de totale fout Δ_{tot} zo klein mogelijk hebben, want dat betekent dat de originele waarden zo nauwkeurig mogelijk zijn gereconstrueerd met de sketch.

2. De afschatting van de maximale fout Δ_{max} willen we zo nauwkeurig mogelijk bepalen. Deze fout geeft ons namelijk een betere indruk van de nauwkeurigheid van het reconstrueren dan de totale fout Δ_{tot}.

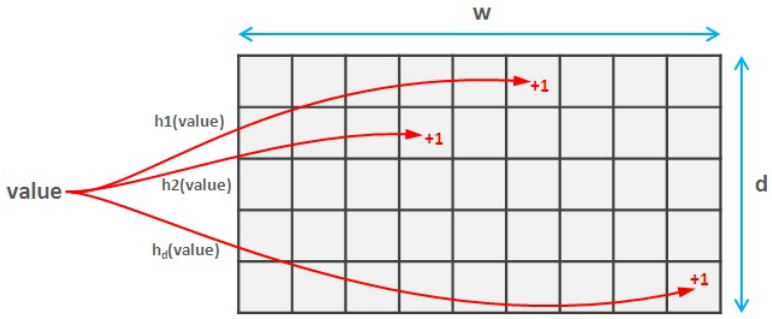
3. De toepassing ligt in trending topics, dus we willen dat woorden met een relatief hoge frequentie een relatief hoge waarde terugkrijgen bij het reconstrueren door de sketch. Daarnaast moet de sketch slim omgaan met de grote groep woorden met frequentie één en deze moeten dus een relatief lage waarde terugkrijgen bij het reconstrueren, want een hoge waarde zou misleidend zijn. We willen dus kijken hoe Δ_{max} zich gedraagt bij verschillende frequenties T_i, met extra aandacht voor die fout bij $T_i = 1$ en $T_i = \max_i T(i)$

3 Uitwerking

We zullen voor de uitwerking verschillende varianten van de Count-sketch gaan analyseren. Daarbij gaan we dus de doelen, gegeven in het Wiskundig model, analyseren voor elke sketch en zo bekijken welke sketch het beste de totale frequenties $T(k)$ kan schatten voor iedere k. De Count sketch wordt dus altijd gevormd door de volgende manier:

$$\text{Voor elke } i = 1, .., d \text{ updaten we na element } u_j = (k_j, v_j):$$
$$S_{i,H_i(k_j)} = S_{i,H_i(k_j)} + v_j.$$

De manier van updaten ziet er dus zo uit: Uit



Figuur 1: Werking van de Count sketch

3.1 Count-Min sketch

Dit is de meest gebruikte methode**(Rf[2])** en is tamelijk eenvoudig qua constructie en analyse.

$R(k) = \min_i S_{i,H_i(k)}, i = 1, .., d.$

Deze methode werkt altijd, omdat vrij makkelijk is in te zien dat: $T(k) \leq R(k)$ voor iedere k. Als er hash-collisons voorkomen, wat altijd het geval is, dan zal $R(k)$ namelijk alleen maar toenemen. De waarde die het dichtst bij $T(k)$ ligt, is het minimum van alle $S_{i,H_i(k)}$ voor elke key k.

Met deze methode zul je dus altijd de waarde $T(k)$ overschatten. Vooral bij het tellen van frequenties van woorden, waarbij veel woorden maar één keer voorkomen, kan dit problematisch worden. Als er bijvoorbeeld gemiddeld n hash-collisions voorkomen, met $1 \ll n$, dan zullen woorden die slechts één keer voorkwamen, toch een gemiddelde frequentie van n terugkrijgen van de Count-Min sketch.

Een variant**(Rf[3])** van de Count-Min sketch, speciaal voor het tellen van frequenties van woorden, is eenzelfde methode van $R(k), F(k)$ die tussendoor, na γ waarnemingen, de sketchwaarden naar beneden update. Deze variant heet Conservative update. Een update is bijvoorbeeld: $S_{i,j} = S_{i,j} - 1, \forall i, j$ na elke γ waarnemingen. Op deze manier probeer je de overschatting klein te houden. De keuze van γ is hier erg belangrijk. Kies je γ klein, dan is de overschatting door hash-collisions relatief klein, maar kan je bij woorden met hoge frequenties ook een onderschatting krijgen. Kies je γ groot, dan blijft de overschatting ook relatief groot.

3.2 Count sketch met een zuivere schatter

De fouten in de Count-min sketch worden veroorzaakt door eventuele hash-collisions en zorgt altijd voor een overschatting. Het blijkt dat deze overschatting zuiver geschat kan worden, met behulp van het gemiddelde van alle andere waarden in dezelfde rij**(Rf[4])**. Nemen we $\Sigma_{|K|} = \sum_{j=1}^{w} S_{1,j} = \sum_{k \in K} T(k)$, dan gaat het construeren van de sketch hetzelfde als in de Count-min sketch, maar de requestfunctie wordt nu gegeven door:
$R(k) = \max(mediaan_i[S_{i,H_i(k)} - \frac{\Sigma_{|K|} - S_{i,H_i(k)}}{w-1}], 1)$.

Nu blijkt namelijk dat de eerste schatter zuiver is voor $T(k)$. Voor een willekeurige rij i in de sketch, geld voor key k namelijk het volgende.
Nemen we $X_x = \begin{cases} 1, & \text{als } H_i(k) = H_i(x); \\ 0, & \text{anders.} \end{cases}$.
Dan geldt dat voor iedere sketchwaarde: $S_{i,H_i(k)} = T(k) + \sum_{x \neq k} X_x * T(x)$. Hier is dus alleen $\sum_{x \neq k} X_x * T(x)$ onbekend. Als we aannemen dat de hash-functies alle keys willekeurig, uniform verdelen over alle w mogelijkheden, dan geldt voor een willekeurige key x, dat $\mathbb{P}[X_x = 1] = \frac{1}{w}$ en $\mathbb{P}[X_x = 0] = 1 - \frac{1}{w}$, dus $E[X_x] = \frac{1}{w}$.

Nemen we schatter: $\widetilde{t_k} = S_{i,H_i(k)} - \frac{1}{w-1}(\Sigma_{|K|} - S_{i,H_i(k)})$, dan geldt hiervoor:

$\mathbb{E}[\widetilde{t_k}] = \mathbb{E}[S_{i,H_i(k)}] - \frac{1}{w-1}(\Sigma_{|K|} - \mathbb{E}[S_{i,H_i(k)}])$

$= T(k) + \mathbb{E}[\sum_{x \neq k} X_x \cdot T(x)] - \frac{1}{w-1}(\Sigma_{|K|} - (T(k) + \mathbb{E}[\sum_{x \neq k} X_x \cdot T(x)]))$

$= T(k) + \sum_{x \neq k} \mathbb{E}[X_x] \cdot T(x) - \frac{1}{w-1}(\Sigma_{|K|} - (T(k) + \sum_{x \neq k} \mathbb{E}[X_x] \cdot T(x)))$

$= T(k) + \frac{w}{w-1} \sum_{x \neq k} \frac{1}{w} \cdot T(x) - \frac{1}{w-1}(\Sigma_{|K|} - T(k))$

$= T(k) + \frac{1}{w-1} \sum_{x \neq k} T(x) - \frac{1}{w-1}(\sum_{x \neq k} T(x))$

$= T(k)$.
Dus $\widetilde{t_k}$ is zuiver voor $T(k)$. Nemen we de mediaan van alle gevonden waarden, dan worden hoge en lage uitschieters zo goed mogelijk genegeerd, dus is de kans dat dat de juiste waarde is groter. Het komt soms voor dat deze mediaan negatief is, dus in dat geval neem je de waarde 1, want een negatieve frequentie zegt natuurlijk niks.

3.3 Count sketch met kleinste kwadraten methode

Deze methode is alleen bruikbaar als je een set van keys $k_1, ..., k_m$ hebt, waarvan je weet dat iedere key in deze set veel vaker voorkomt dan andere keys. Bij de requestfunctie van de Count-Min sketch, gooi je eigenlijk veel informatie weg die tot je beschikking is. Want elke $S_{i,H_i(k)}$ bevat de waarde $T(k)$, plus een bepaalde ruis Y, die afkomstig is van andere elementen. Een methode**(Rf[5])** die daarom ontwikkeld is, maakt gebruik van de kleinste kwadraten methode. De vector $K_H = (k_1, ..., k_m)$ is de vector van alle keys, waarvan we weten dat deze relatief hoge waarden heeft. Voor $R(k)$ gebeurt nu het volgende:

Construeer de matrix $A \in \{0,1\}^{d*w\times(m+1)}$ met voor $k_l \in k_1, .., k_m$:

$$\begin{cases} A_{w*i+H_i(k_l)+1,l} = 1 & \text{als } k_l \text{ in } S_{i,j} \text{ gehashed wordt.;} \\ A_{i,(m+1)} = 1, & \text{voor alle rijen } i; \\ A_(i,j) = 0, & \text{elders.} \end{cases}$$

Zij dan $\underline{b} = (S_{K*i+j+1})^T, i = 1, .., d; j = 1, .., w$ de $(d * w + 1)$ vector van alle waarden van de sketch. Er geldt voor iedere key k dat $S_{i,H_i(k)} = T(k) + Y$. Als we dus de vector $\underline{x} = (T(k_1), T(k_2), .., Y)$ nemen, dan willen we dus eigenlijk minimaliseren: $\min_{\underline{x}} ||A\underline{x} - b||$. Dit kan met behulp van de kleinste kwadraten methode en de Moore-Penrose pseudo-inverse A^+.
Het minimum wordt dan namelijk gegeven door: $\underline{x} = A^+\underline{b}$. De waarde $R(k_i) = \underline{x}_i$ van de vector is een betere schatter voor $T(k_i)$. $\underline{x}_i$ wordt namelijk geschat met behulp van alle sketchwaarden en niet alleen degene waarop k gehashed wordt.

Omdat deze schatter dus alleen bruikbaar is voor woorden met hoge frequenties, is deze in zijn algemeen niet bruikbaar en gaan we deze ook niet analyseren. Deze schatter is vooral genoemd om te illustreren wat er allemaal mogelijk is met een Count sketch.

3.4 Aanmerkingen bij de sketches

3.4.1 Andere methoden

De methodes die onderzocht zijn, berusten allemaal op de werking van de Count sketch. Er zijn natuurlijk andere methoden, waarbij frequenties achterhaald kunnen worden. Men zou bijvoorbeeld ook kunnen kiezen voor een Counting filter(**Rf[6]**). Dit is één grote array van n-bit-counters, waarbij elk woord een of meerdere keren in deze array gehashed wordt en frequenties worden bijgehouden in de n-bits. Dit lijkt een beetje op Count sketch met $d = 1$, alleen worden frequenties daar op bit-niveau geupdate. Het grote probleem hierbij, is dat deze array snel vol loopt. Om de fout klein te houden voor zelfs een kleine hoeveelheid verschillende keys, moeten de keys meerdere malen gehashed worden in de filter. Op een gegeven moment zijn er geen vrije plekken meer en zeker in het geval dat veel woorden één keer voorkomen, is dat een probleem.

Een andere, veelgebruikte methode(**Rf[7]**) is om niet alle updates te onthouden, maar bijvoorbeeld maar C woorden en hun frequenties bij te houden. Als er nog plek is in C, dan kan een woord worden toegevoegd. Is er geen plek meer én een woord komt niet in C voor, dan worden alle frequenties verlaagd met 1. Als bij een woord de frequentie op 0 staat, wordt het verwijderd uit C. Het probleem hierbij is dat de kans groot is dat je op het einde woorden overhoud, die niet belangrijk zijn.

3.4.2 Hashen van woorden

Woorden maken deel uit van Σ^*, wat een oneindige set is. Om dit aan te kunnen en om te kunnen werken met getallen, worden de woorden dus als eerste gehashed naar een 32-bits getal. De werking van de sketches vereist nu ook dat dit getal strikt positief is, dus er wordt één bit weggegooid. Daarna wordt het woord gehashed naar zijn positie in de sketch. Het lijkt omslachtig om twee keer te hashen, maar het zorgt er wel voor dat de sketch een grote set als Σ^* aan kan. Alle woorden uniek tellen is simpelweg ook de ruimte niet voor. Het gaat nu echter wel helemaal fout als twee woorden absoluut naar hetzelfde 32-bits getal gehashed worden, want dan worden ze op dezelfde plekken gehashed in de sketch. Maar deze kans negeren we.

3.4.3 Motivatie voor een nieuwe methode

Bij het analyseren van de meest gebruikte Count sketches, zijn er een aantal punten die ervoor zorgen dat de fout erg groot wordt: Woorden die één keer voorkomen kunnen uiteindelijk toch voor hoge frequenties zorgen, wat misleidend is. Het hoge aantal hash-collisions zorgt hiervoor. Dit aantal kun je verlagen, door een grotere sketch te nemen. Maar vaak is de sketch van beperkte, vaste grootte, waardoor dit dus niet mogelijk is.
We kunnen echter het aantal ongewilde hash-collisions ook gaan schatten. Als de familie van onafhankelijke hash-functies H_i altijd de waarden ongeveer gelijk verdeeld over alle beschikbare plaatsen, wat in de sketch-methode dus w is, dan geldt het volgende:
$HC \approx \frac{|K|}{w}$, met HC het aantal hash-collisions, $|K|$ de kardinaliteit van de ruimte van keys, die gebruikt zijn in deze sketch. Het komt er dus op neer dat we HC willen gaan schatten.

Bij de zuivere schatter, haal je elke keer het gemiddelde van de overige sketchwaarden van de rij eraf. Dit zal echter nadelig werken, bij woorden met een frequentie om en rond het gemiddelde van alle frequenties. Daar zal het er namelijk om spannen of er een negatief waarde uitkomt bij de schatter of een positieve. Omdat we aannemen dat $T(k)$ Zipf-verdeeld is, en er dus zeer veel woorden met frequentie 1 voorkomen maar ook hoge uitschieters zijn, zullen waarden rond het gemiddelde toch veel groter zijn dan 1. Als er dan een negatieve schatter wordt teruggegeven, dus uiteindelijk 1 als resultaat, zal dat een grote fout zijn. De verwachting bij de zuivere schatter is dan ook dat de fouten het grootst zijn rond het gemiddelde van alle frequenties. Dit kunnen we verhelpen door een "veiligere"keuze van de overschatting te nemen, namelijk het aantal hash-collisions HC.

3.5 Count sketch met collisionreductie

Met de extra aanname, dat $T(k)$ Zipf-verdeeld zijn, dus veel woorden één keer voorkomen, gaan we de volgende schatter voor HC gebruiken:
$\widetilde{hc_i} = \min_j S_{i,j}$ voor alle $i = 1, .., d$.
Nemen we weer X_x als in sectie [3.3], dan geldt:
$\mathbb{E}[\widetilde{hc_i}] = \mathbb{E}[\min_j S_{i,j}] = \mathbb{E}[\min \sum_{x \in K} X_x \cdot T(x)] \geq \mathbb{E}[[\sum_{x \in K} X_x]] \cdot \min_x [T(x)] \geq$
$\mathbb{E}[\sum_{x \in K} X_x] \cdot 1 = \frac{|K|}{w} = HC$.
Dus de verwachting is dat deze schatter het aantal hashcollisions overschat.
Voor $R(k)$ gebruiken we nu het volgende:

$$R(k) = \max(\min_i [S_{i,H_i(k)} - \min_j S_{i,j}], 1).$$

De enige keer dat we nu een 1 terug verwachten, is bij de k waarvoor geldt: $S_{i,H_i(k)} = \min_j S_{i,j}$. Maar hiervan nemen we toch aan dat dat alleen woorden zijn met frequentie 1. Omdat $\widetilde{hc}$ het aantal hash-collisions overschat, zal $R(k) \leq T(k)$ vaak gelden. Maar de verwachting is dat de fout wel een stuk kleiner is.

Bij de werking van de Count sketch, is het heel logisch om een zo hoog mogelijke w te kiezen. Het is gebleken **(Rf[2], Rf[3])** voor de Count-min sketch, dat $d = 5$ al goed genoeg is en daarna w zo hoog mogelijk gekozen moet worden. Je verkleint hiermee namelijk het aantal hash-collisions zo goed mogelijk en behoud genoeg verschillende waarden om je eenmaal minimum te beperken. Met de collisionreductie en de aanname van de Zipf-verdeling is het echter zo, dat je het aantal hash-collisions probeert te schatten en daarmee w niet zo hoog mogelijk hoeft te kiezen. Het voordeel van een grotere d, is dat de kans steeds groter wordt dat $R(k)$ steeds dichter naar $T(k)$ komt. We verwachten dus dat als we d verhogen met deze manier, de fout steeds kleiner wordt. We gaan bij deze methode dus eerst w bepalen, en daarna d zo hoog mogelijk kiezen. We kiezen w in ieder geval groter dan het aantal woorden met hoge frequenties. Met een Zipf-verdeling verwachten we bijvoorbeeld dat we w erg klein kunnen kiezen.

3.6 Combineren van sketches

Het mooie aan de werking van Count sketches, is dat sketches makkelijk gecombineerd kunnen worden. Als we namelijk verschillende sketches bijhouden, met allemaal dezelfde hashfuncties $H_i, i = 1, .., d$ en zelfde afmetingen $d \times w$, dan kunnen we de sketches in ieder geval elementsgewijs bij elkaar optellen. Want woorden die hetzelfde zijn, zullen in beide sketches op dezelfde plaatsen gehashed worden.
Dus $S^{(1)}_{i,j} + S^{(2)}_{i,j} = S^{*}_{i,j}, i = 1, .., d; j = 1, .., w$.
Hierbij moeten we wel aannemen dat een update u_i niet door beide sketches geregistreerd wordt. Deze methode kan ook gebruikt worden bij de andere methoden, omdat op dezelfde manier geteld wordt.

4 Resultaten

4.1 Analytische resultaten

Voor een sketch $S \in \mathbb{N}^{d \times w}$, gaan we de fouten, gegeven in de wiskundige uitwerking, afschatten op basis van $\Sigma_{|K|} = \sum_{k \in K} T(k)$, d en w. We nemen aan dat de d hashfuncties alle waarden willekeurig, onafhankelijk verdelen en er dus een verwachtte kans op een hashcollision is van $p = \frac{1}{w^d}$.

4.1.1 Count-Min sketch

Zoals gezegd is de manier van updaten bij de Count-sketch, zodanig dat altijd geldt, voor willekeurige key k: $T(k) \leq R(k)$, met $R(k) = \min_i S_{i,H_i(k)}$. Verder geldt dat de verwachtte kans op een hashcollision $\frac{1}{w}$ is, dus als bovenschatting nemen we dan: $R(k) \leq T(k) + \frac{1}{w}(\Sigma_{|K|})$. Nu geldt, gebruikmakend van de Markov-Ongelijkheid, dat de kans dat $R(k)$ boven deze waarde uitkomt, $1 - \frac{1}{d}$ is.
Als we kijken naar de maximale fout, zal een bovenschatting hiervoor dus worden:
$\Delta_{max} \leq \frac{1}{w}\Sigma_{|K|}$. Naarmate we dus $|K|$ vergroten, zal deze maximale fout naar verwachting dus lineair toenemen. Deze overschatting is echter ruim genomen, dus in werkelijkheid zal het wat kleiner zijn.

4.1.2 Count sketch met zuivere schatter

Te verwachten is dat deze sketch pas goed gaat werken, als we $|K|$ groot genoeg nemen. Bij lage $|K|$, is het onnodig om het gemiddelde van de rest van de rij eraf te halen, omdat de overschatting waarschijnlijk veel lager ligt. Zodra de overschatting groot genoeg wordt, zal de schatter dus beter gaan werken.
De maximale fout die gemaakt wordt, zal liggen bij een woord dat een lage frequentie heeft, maar van de schatter een hoge waarde terugkrijgt. De maximale fout is dan:
$\Delta_{max} \leq \max_{i,j}[S_{i,j} - \frac{1}{w-1}(\Sigma_{|K|} - Si, j) - 1]$
$= [(1 - \frac{1}{w-1}) \max_{i,j} S_{i,j} - \frac{\Sigma_{|K|}}{w-1} - 1]$
$\leq (1 - \frac{1}{w-1})(2 * \max_k T(k) + \frac{\Sigma_{|K|}}{w-1}) - \frac{\Sigma_{|K|}}{w-1}$
$= 2(1 - \frac{1}{w-1}) \max_k T(k) + \frac{\Sigma_{|K|}}{(w-1)^2}$.
Dan moet een woord k met $T(k) = 1$ namelijk $\lceil \frac{d}{2} \rceil$ keer een hash-collision hebben met een woord met maximale frequentie. Dit gebeurt met kans $p \leq (\frac{1}{w})^{\lceil \frac{d}{2} \rceil}$, want dat is de kans dat k met een woord met hoogste frequentie gehashed wordt in zijn schatter. Toch kan dit ervoor zorgen dat de maximale fout zeer groot wordt en erg misleidend is. De gemiddelde fout bij deze schatter zal echter veel lager zijn dan de Count-Min sketch.

4.1.3 Count sketch met collisionreductie

De maximale fout zal alleen worden veroorzaakt door hash-collisions met woorden k met $T(k) > 1$. Daarom zal deze fout net als bij de Count-Min sketch lineair lopen, maar nu met de volgende fout:
$\Delta_{max} \leq \frac{1}{w}(\Sigma_{|K|} - |K|)$.
Dit komt omdat alle hash-collisions eraf zijn gehaald. De maximale fout blijft hier weer onder met met dezelfde kans van $1 - \frac{1}{d}$ als bij de Count-Min sketch, aan de werking is namelijk niks veranderd.

Zijn de woorden echter Zipf-verdeeld, dan geldt een nog betere afschatting:
$\Delta_{max} \leq \frac{\max_k T(k)}{d}$ als $w > |HH|$ met $|HH|$ het aantal woorden met frequentie boven het gemiddelde.

4.1.4 Grootte van S

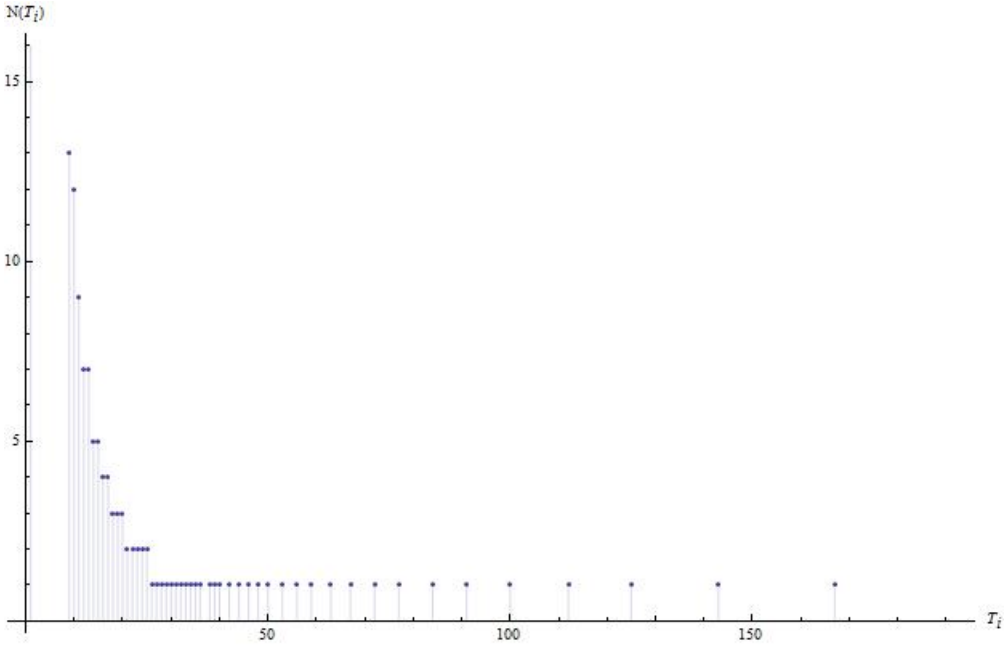
De grootste fout die gemaakt wordt met de Count-methode, is als een woord d keer een hash-collision heeft met een woord met maximale frequentie. Dit gebeurt met kans $p_{maxhash} = |K|\frac{1}{w}^d$. Deze kans is 1 als $K = \frac{1}{\frac{1}{w}^d}$. Daarom moeten w en d in ieder geval altijd zo groot hebben, dat deze kans zeer klein is. Zoals gezegd bij de analyse van de zuivere schatter, is de kans dat de maximale fout voorkomt wat groter, want hiervoor hoeft maar een woord maar $\lceil \frac{d}{2} \rceil$ keer een hash-collision krijgen met een woord met maximale frequentie.

In de analyse van de sketches hebben we de maximale fout afgeschat met Δ_{max}. Als we een maximale tolerantie hebben voor deze fout en gegeven schatters van $\max_k T(k)$ en $\Sigma_{|K|}$, dan is het interessant om te weten hoe groot we S minimaal moeten kiezen. Bij de verschillende schatters hebben we dan de volgende eis aan w en d:

- Count-Min sketch: $w \geq \frac{1}{\Delta_{max}} \cdot \Sigma_{|K|}$ en wordt voldaan aan de tolerantie met kans $p = 1 - \frac{1}{d}$.
- Count sketch met zuivere schatter: de uitdrukking in de analyse van de zuivere schatter is zo complex, dat we w niet expliciet gaan bepalen. w moet gewoon gekozen worden, zodat:
  $\Delta_{max} \leq 2(1-\frac{1}{w-1}) \max_k T(k) + \frac{\Sigma_{|K|}}{(w-1)^2}$ en wordt voldaan aan de tolerantie met kans $p \geq 1 - \frac{1}{w}^{\lceil \frac{d}{2} \rceil}$.
- Count sketch met collision reductie: In het algemene geval moet: $w \geq \frac{1}{\Delta_{max}} \cdot (\Sigma_{|K|} - |K|)$ en wordt aan de tolerantie voldaan met kans $p = 1 - \frac{1}{d}$.
  Als de frequenties Zipf-verdeeld zijn, geldt echter een andere eis:
  De maximale fout is ongeveer $\frac{\max_k T(k)}{d}$ als $w > |HH|$ met $|HH|$ het aantal woorden met relatief hoge frequentie (Heavy Hitters). Dus dan moet $d \geq \frac{\max_k T(k)}{\Delta_{max}}$.

4.2 Resultaten simulaties

Voor de resultaten in de simulatie, gebruiken we een gesimuleerde dataset, waaruit willekeurige variabelen getrokken worden. Bij elke simulatie is nu een dataset van 10000 woorden met frequenties, waarbij de frequenties Zipf verdeeld zijn, in een sketch met $d = 5, w = 100$ samengevat. De frequenties voor woorden in een Zipf-verdeling volgt ongeveer de lijn $\lceil \frac{\max_k T(k)}{i} \rceil$, met i het i-de grootste woord. Nemen we $\max_k T(k) = 1000$, dan ziet het toaal aantal woorden $N(T_i)$ met een frequentie T_i tussen 1 en 1000 er zo uit:



Figuur 2: Aantal woorden per frequentie

De lange staart van woorden met frequentie 1 is dus duidelijk zichtbaar. De gemiddelde waarde is ongeveer 170, wat dus goed aangeeft dat er maar heel weinig woorden boven dit gemiddelde zitten.

Om te kijken hoe goed een sketch werkt, gaan we de volgende fouten bekijken:

- $\Delta_{max}(T_i) = \max\limits_{\substack{k \in K \\ T(k)==T_i}} |R(k) - T_i|$, oftewel de maximale fout per frequentie T_i, die gemaakt wordt bij het reconstrueren met een bepaalde sketchmethode.
- $\Delta_{gem}(T_i) = \frac{1}{N(T_i)} \sum\limits_{\substack{k \in K \\ T(k)==T_i}} |R(k) - T_i|$ de gemiddelde fout bij woorden met frequentie T_i, met $N(T_i)$ het totale aantal woorden met frequentie T_i.

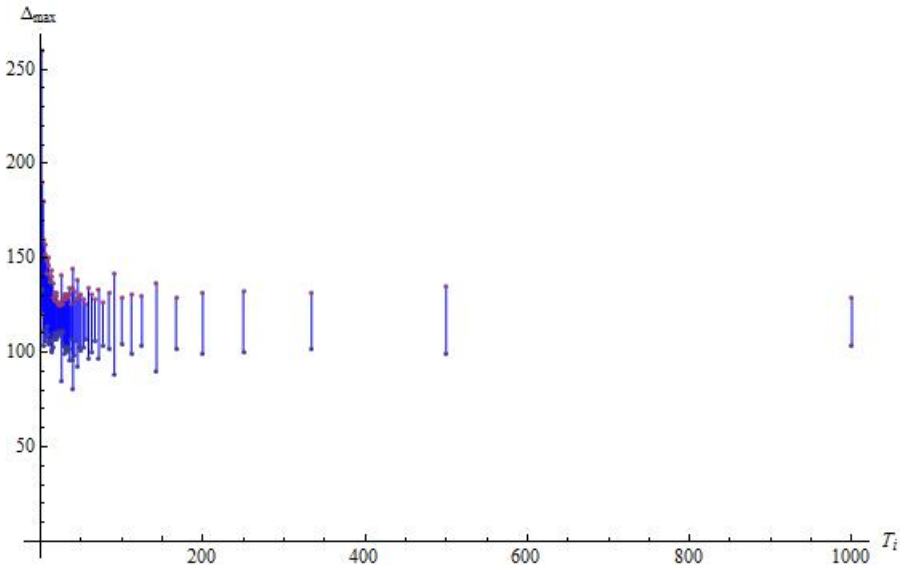
Voor deze verschillende soorten fouten, zijn er 95% betrouwbaarheidsintervallen gegeven voor elke T_i. Daarnaast gaan we kijken hoe de maximale fout Δ_{max} zich gedragen als we $|K|$ vergroten en $\max_k T(k)$. Daarbij gaan we voor elke grootte van $|K|$ en $\max_k T(k)$ 100 keer simuleren en het gemiddelde van de maximale fout wordt geplot. Hierbij vergroten we K steeds met $d \cdot w = 500$, oftewel een volledige sketchsize, en verhogen we $\max_k T(k)$ met $\frac{|K|}{10} = 50$. Deze gaan we vergelijken met het aantal hash-collisions HC, want de maximale fout van de Count-Min ligt in die buurt. Als de fout hoger is dan veroorzaakt door dit aantal hash-collisions, wordt het verschil rood gekleurd. Ligt de fout lager, dan wordt het groen gekleurd, want dat betekent dat het beter zal werken dan Count-Min.

Daarna gaan we kijken hoe de maximale fout zich gedraagt als we d en w variëren. Hierbij gaan we nog steeds een totale ruimte nemen van $|S| = 500$, maar hierbij gaan we d ophogen van 1 tot 20 en nemen we $w = \frac{500}{d}$.

Tot slot gaan we onze dataset helemaal random maken, waar voor elk woord wordt een frequentie tussen 1 en $\frac{|K|}{10}$ uniform random wordt genomen en K weer met een sketchsize van $S = 500$ verhoogd wordt. Dit ter illustratie voor de algemene werking van de sketch.

4.2.1 Resultaten Count-Min sketch

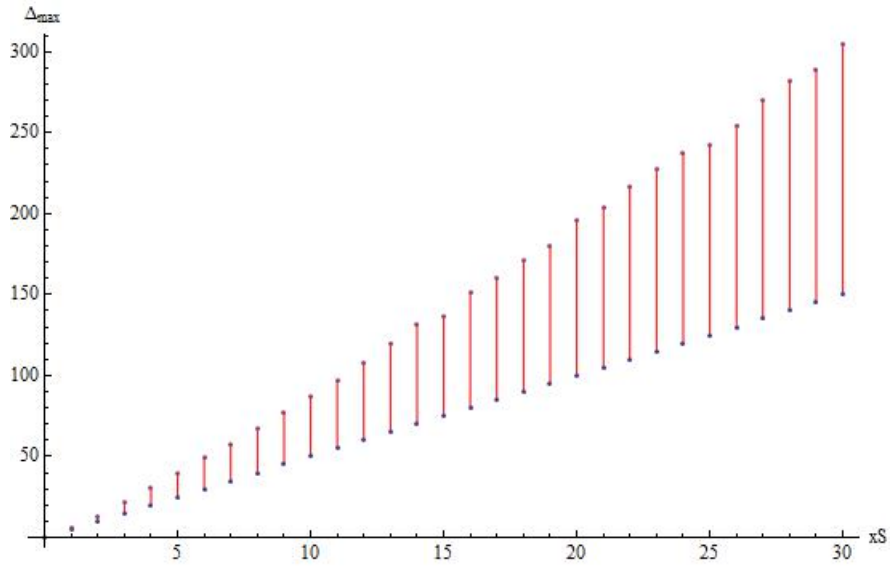
De Count-Min sketch geeft zoals gezegd altijd een overschatting. Zeker in het geval dat veel woorden maar één keer voorkomen, wordt dat problematisch.
Als we kijken naar de totale fout Δ_{tot}, dan wordt een 95% betrouwbaarheidsinterval gegeven door:
$\Delta_{tot} \in [1071586, 1275910]$.
Onderstaande grafieken geven aan hoe groot de maximale overschatting is bij de geanalyseerde fouten:



Figuur 3: 95% betrouwbaarheidsintervallen van maximale fout

De fouten liggen ongeveer allemaal op dezelfde hoogte, rond de 100. Dit is dus ongeveer het aantal hash-collisions, want bij een dataset van 10000 is dat $\frac{10000}{w} = 100$.

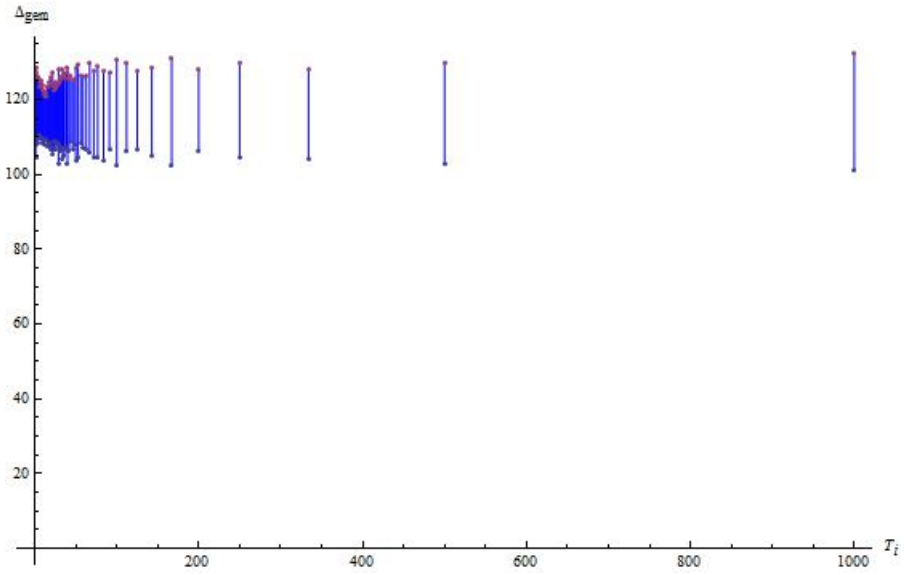
Nu willen we kijken naar de toename van de maximale fout Δ_{max}, ten opzichte van de grootte van $|K|$ en $\max_k T(k)$. Nemen we dan $xS = x \cdot (d \cdot w)$, dan komt daar de volgende grafiek uit:



Figuur 4: Maximale fout bij toenemende $|K|$ en $\max_k T(k)$

Hier is goed te zien dat de maximale fout lineair oploopt. Dit zal alleen komen door het vergroten van $|K|$. De grafiek volgt, zoals verwacht, dus ongeveer het aantal hash-collisions met een extra fout, veroorzaakt door woorden k met $T(k) > 1$.

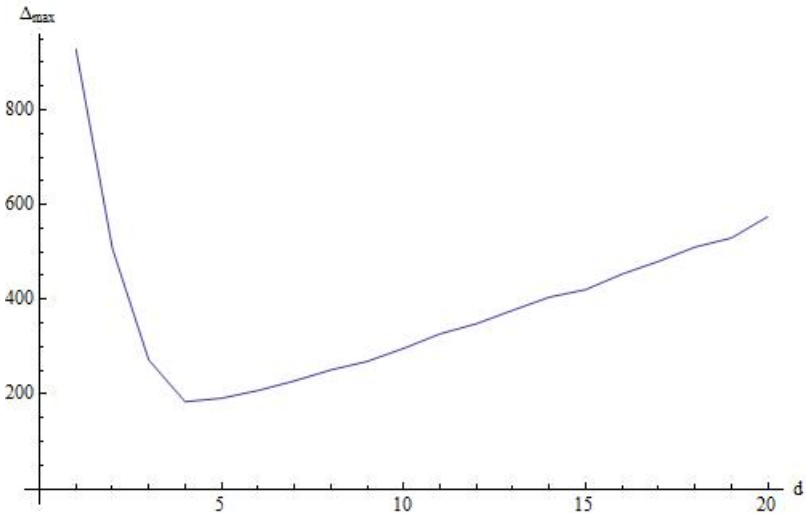
De gemiddelde fout van de Count-Min sketch zal ongeveer op het gemiddelde aantal hash-collisions liggen. Dat is dus $\frac{|K|}{w}$ en in deze simulatie dus 100:



Figuur 5: 95% betrouwbaarheidsintervallen van de gemiddelde fout

Het blijkt hier dus al uit dat als je het aantal hash-collisions kan schatten en van de minimale waarde aftrekt, je maximale én gemiddelde fout dus drastisch verbeterd worden. De overschatting boven het aantal hash-collisions, komt van het aantal hash-collisions met woorden k met $T(k) > 1$.

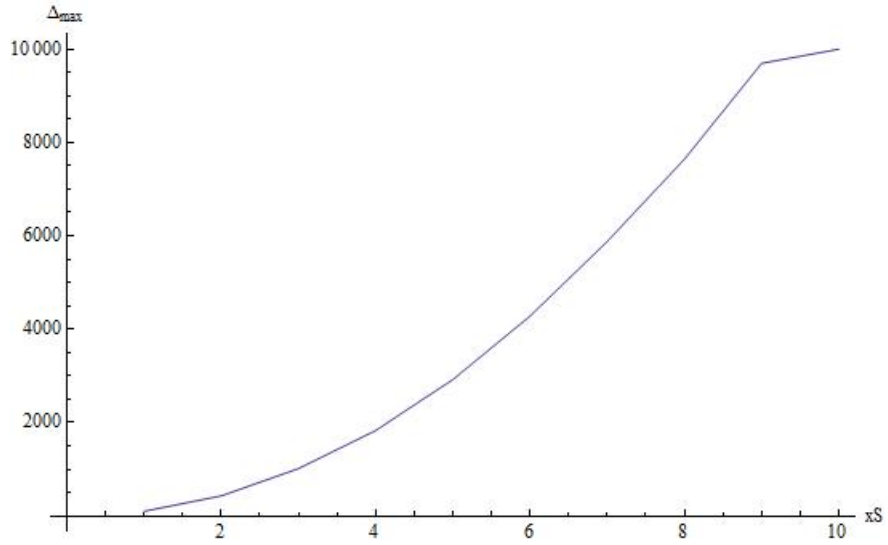
Als we kijken naar het gedrag van Δ_{max} bij het verhogen van d, verwachten we eerst een daling en daarna weer een stijging van de maximale fout. De daling komt doordat de kans op een hash-collision met een woord met zeer hoge frequentie steeds kleiner wordt. De stijging zal komen door de stijging van het aantal hash-collisions, aangezien w ook steeds kleiner wordt. De grafiek ziet er zo uit:



Figuur 6: Maximale fout bij toenemende d

De maximale fout door een hash-collision met een woord met maximale frequentie is ongeveer $\frac{\max_k T(k)}{d} = \frac{1000}{d}$. Het gemiddelde aantal hash-collisions, en in een Zipf-verdeling dus de gemiddelde overschatting, is $\frac{|K|}{w} = \frac{10000}{\frac{500}{d}} = d\frac{10000}{500} = 20d$. Dus deze grafiek volgt ongeveer de lijn $f(x) = \frac{1000}{x} + 20x$. Het optimum van f ligt dan in het punt waar $f'(x) = 0 = \frac{-1000}{x^2} + 20$. Dus als $x = \sqrt{\frac{1000}{20}} \approx 7$. In werkelijkheid ligt het dus iets lager, bij d = 4. Na het optimum neemt de stijging van het aantal hash-collisions weer de overhand en stijgt de grafiek lineair, zoals te verwachten. De optimale d zal echter ook verschillen met je dataset.

Als de data uniform random wordt, is de kans veel groter dat één hash-collision al een grote fout oplevert door een relatief hoge frequentie. Als al deze grote fouten van alle hash-collisions bij elkaar op wordt geteld, zal de fout dus extreem groot worden. Dat is ook te zien aan de grafiek:

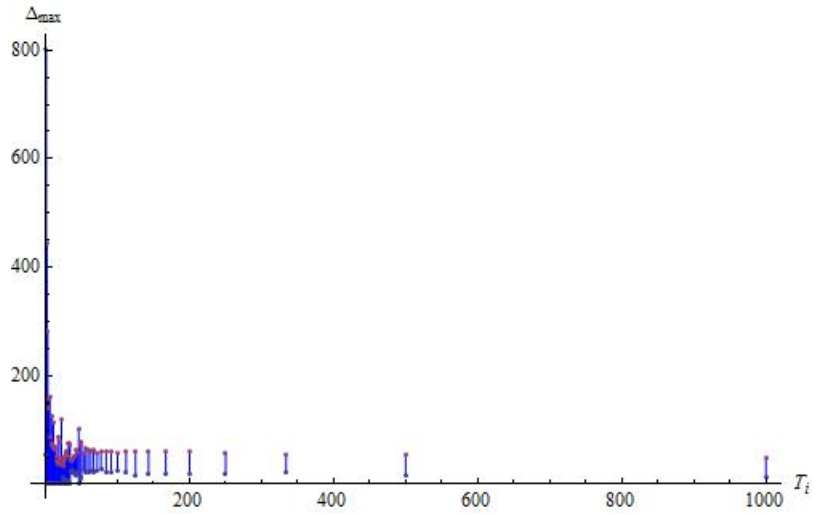


Figuur 7: Maximale fout bij toenemende $|K|$ en $\max_k T(k)$ met uniforme, random data

De maximale fout is naar verwachting kleiner of gelijk aan $\frac{\Sigma_{|K|}}{w}$. $|K|$ verhogen we elke stap met 500 en de gemiddelde waarde met $\frac{50}{2}$, dus de maximale fout is ongeveer een kwadratische grafiek van $f(x) = \frac{500x \cdot \frac{50}{2} x}{100} = 125x^2$. De maximale fout ligt hier dus onder, maar volgt ongeveer dezelfde trend.

4.2.2 Resultaten Count sketch met een zuivere schatter

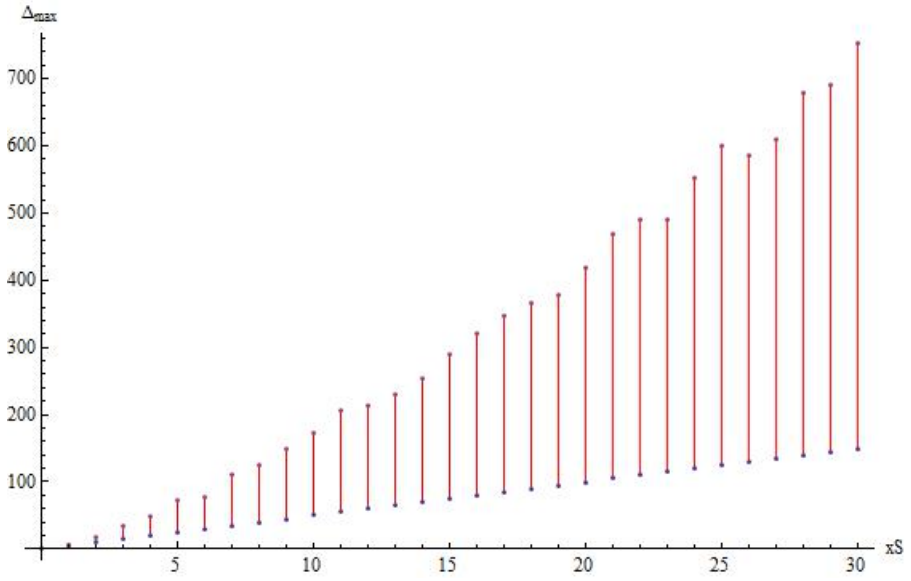
De zuivere schatter kan een onderschatting opleveren, zeker als de schatter negatief is en waarde 1 wordt teruggegeven. Als we kijken naar de totale fout, is dat echter veel lager dan bij de Count-Min sketch, namelijk:
$\Delta_{tot} \in [13996, 33656]$
Als we kijken naar de maximale fout bij de verschillende frequenties en daar 95% betrouwbaarheidsintervallen van plotten, komt daar de volgende grafiek uit:



Figuur 8: 95% betrouwbaarheidsintervallen van maximale fout per frequentie

Duidelijk is te zien dat de maximale fout bij $T_i = 1$ erg hoog is, zoals van te voren ook voorspelt. De maximale fouten vertonen daarnaast nog veel pieken bij de gemiddelde T_i, want daar zal het er dus om spannen of de schatter positief of negatief is.

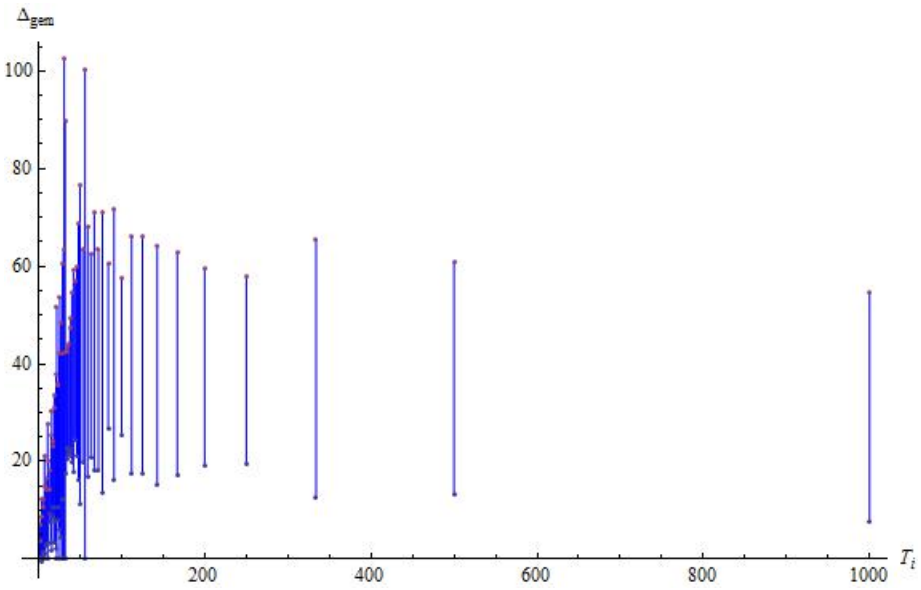
Als we kijken naar maximale fout bij een oplopende $|K|$ en oplopende $\max_k T(k)$ en deze vergelijken met het gemiddelde aantal hash-collisions, dan zal deze fout ver daar boven gaan liggen. Bij het vergelijken met het aantal hash-collisions, komt daar ook de volgende grafiek uit:



Figuur 9: Maximale fout bij toenemende $|K|$ en $\max_k T(k)$

Hier is goed te zien dat de maximale fout bij woorden met frequentie 1 blijft stijgen, omdat $\max_k T(k)$ ook steeds groter wordt. De gemiddelde fout zal veel lager liggen, maar deze maximale fout is dus vrij ernstig. Dit komt dus alleen omdat $\max_k T(k)$ toeneemt.

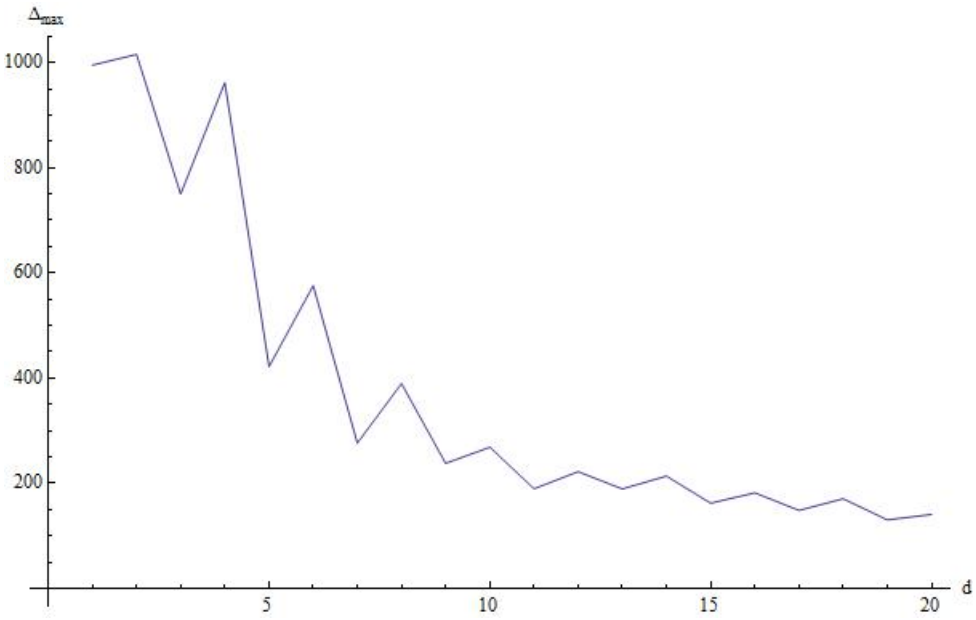
Bij de maximale fout bij woorden met frequentie 1, bleek ook dat de variantie daar erg groot is, omdat het 95% betrouwbaarheidsinterval zo groot was. Dit betekend dat het gemiddeld waarschijnlijk veel beter zal zijn.
De gemiddelde fout bij de frequenties ziet er namelijk zo uit:



Figuur 10: 95% betrouwbaarheidsintervallen van de gemiddelde fout per frequentie

Hier is dus goed te zien dat de gemiddelde fout bij $T_i = 1$ heel klein is. Dit komt dus omdat de verwachting van de fout 0 is, dus zal het gemiddelde deze ook naderen. Een woord met frequentie één heeft echter nog een voordeel in de berekende schatter, namelijk dat als de schatter gemiddeld negatief is, wat dus het geval is als de sketchwaarde een relatief lagere waarde heeft dan de rest in dezelfde rij, een frequentie van 1 wordt teruggegeven. Hiermee zijn ook gelijk de uitschieters rond het gemiddelde te verklaren, want daar zal het er om spannen of de schatter positieve of negatieve waarde teruggeeft. Er zal dus gemiddeld vaker een 1 worden teruggegeven, terwijl de waarde veel hoger ligt maar net nog onder het gemiddelde.

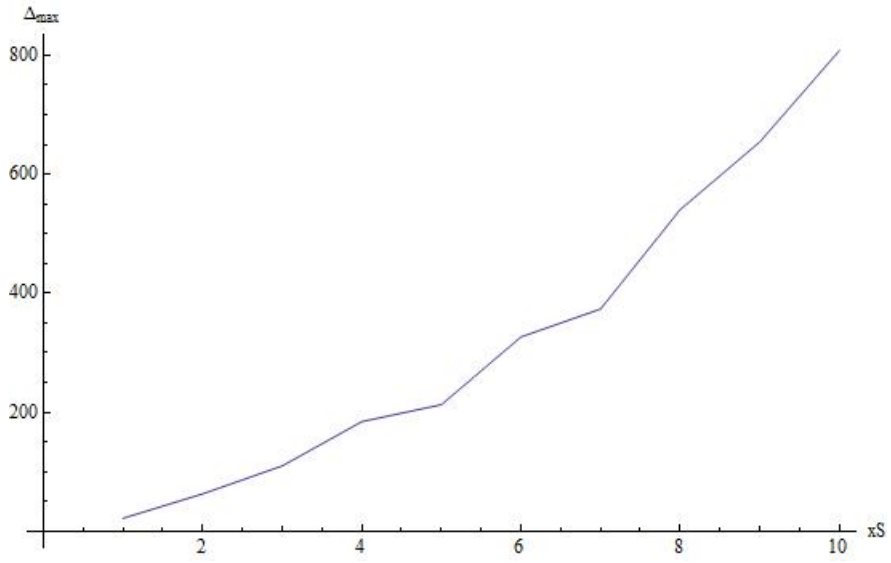
Als we kijken naar hoe goed de sketch werkt als we d ophogen, zien we een dalend,alternerende lijn:



Figuur 11: Maximale fout bij toenemende d

Het alterneren komt waarschijnlijk omdat er dan te weinig simulaties (100 is niet veel) zijn uitgevoerd en de variantie tussen de waarden erg groot is. De maximale fout kan namelijk extreem groot worden, maar ook laag zijn. Deze onvoorspelbaarheid zie je dus veel terug in de grafiek. Wel is ook duidelijk te zien dat de maximale fout daalt, omdat de kans dat een schatter erg fout zit steeds kleiner wordt.

Deze methode werkt gemiddeld dus vrij goed en dat zal ongeveer hetzelfde blijven bij random data:

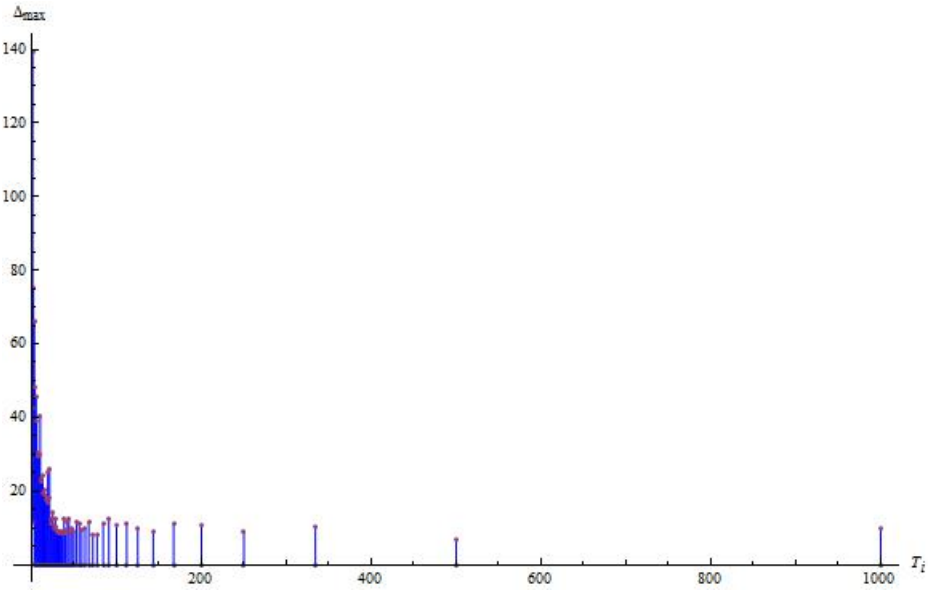


Figuur 12: Maximale fout bij toenemende $|K|$ en $\max_k T(k)$ met uniforme, random data

De maximale fout neemt dus nog steeds toe, maar wel veel minder hard dan bij de Count-Min sketch. De verwachting van de schatter is namelijk nog steeds de daadwerkelijke frequentie, maar door het vergroten van je dataset $|K|$ en de maximale waarde, zal de fout wel groeien.

4.2.3 Resultaten Count sketch met collisionreductie

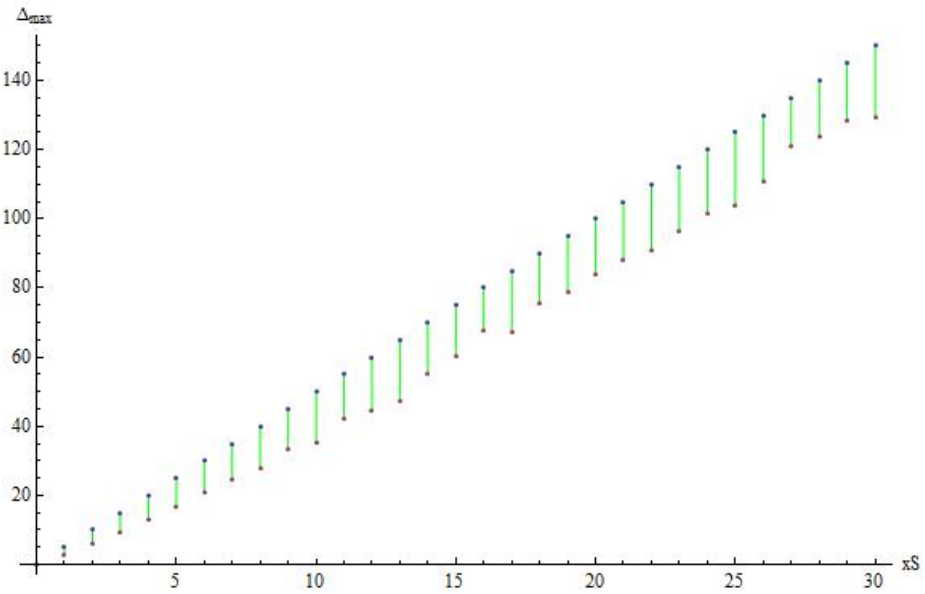
De verwachting is dus dat deze count-sketch bijna geen onderschatting én overschatting meer geeft. Zeker voor waarden van $T(k)$ die rond het gemiddelde liggen, zal dit een verbetering zijn ten opzichte van de zuivere schatter. Wel zullen er gemiddeld meer fouten worden gemaakt bij de schatter, dus de totale fout zal wat hoger liggen dan bij de zuivere schatter. Dat blijkt ook uit het 95% voor de totale fout:
$\Delta_{tot} \in [39283, 90425]$.
Als we dan ook kijken naar de maximale fout per T_i, dan komt daar het volgende uit:



Figuur 13: 95% betrouwbaarheidsintervallen voor de maximale fout per T_i

Hier is goed te zien dat de grootste fout nog steeds ligt bij 1, maar dat de fout daarna snel daalt. Ook is deze veel lager dan bij de zuivere schatter, want hij zal nu het minimum pakken. Dit minimum zal nooit liggen bij $\max_{i,j} S_{i,j}$, dus dat is zeker een verbetering ten opzichte van de zuivere schatter. Ook is hier goed te zien dat bij de gemiddelde waarden van $T(k)$, er geen hoge pieken meer voorkomen. Dit komt dus omdat de overschatting nu "goed"gaat en er geen 1 wordt teruggegeven. De overschatting wordt nu alleen nog gegeven door hash-collisions van woorden met een frequentie hoger dan 1. Deze overschatting is echter vrij laag, zoals te zien in de grafiek.

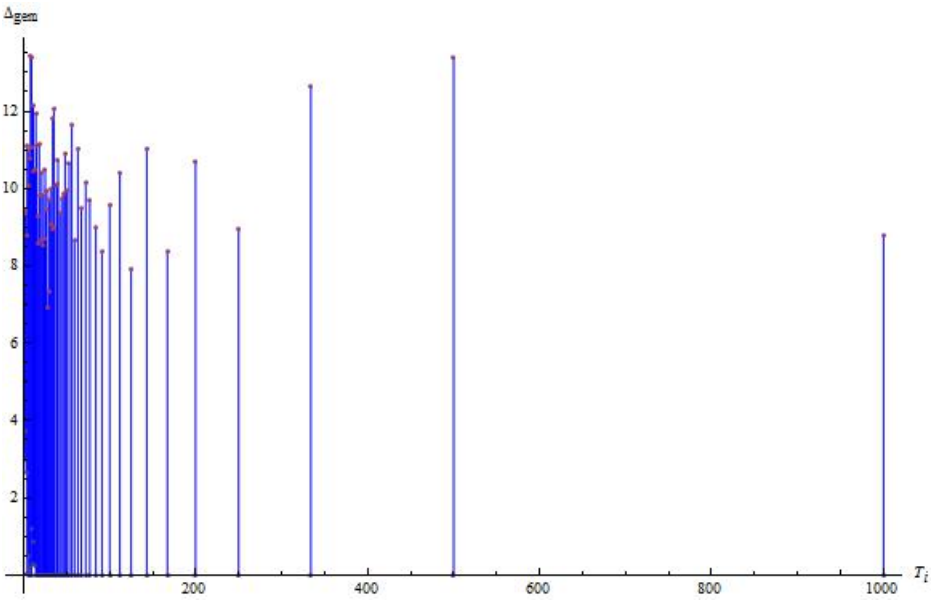
We verwachten dat als we $|K|$ en $\max_k T(k)$ verhogen, de maximale fout bij deze sketch dus lineair zal lopen, maar wel onder het aantal hash-collisions. Deze heb je er namelijk al afgehaald, dus de fout wordt nu, zoals eerder gezegd, alleen nog veroorzaakt door woorden k met $T(k) > 1$. Onderstaande grafiek geeft aan hoe de maximale fout verloopt:



Figuur 14: Maximale fout bij toenemende $|K|$ en $\max_k T(k)$

Duidelijk is dus te zien dat de fout onder het aantal hash-collisions blijft, wat dus fijn is. De fout stijgt lineair mee met de grootte van K, wat dus ook te verwachten was.

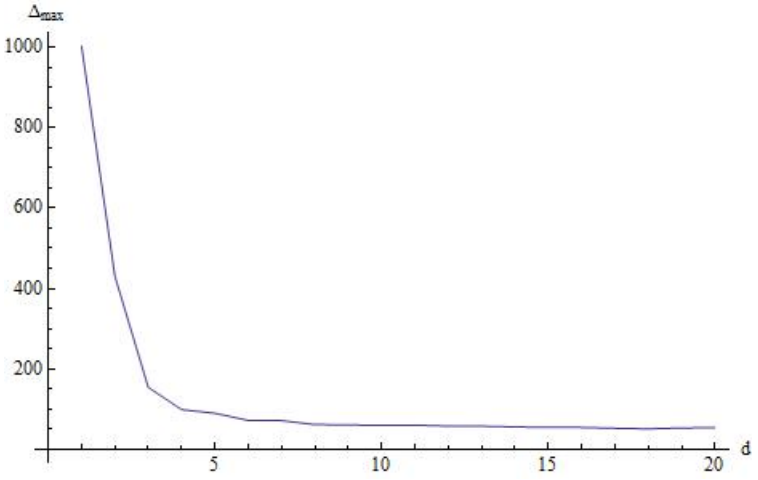
Ook bij de gemiddelde fout verwachten we lage waarden terug te krijgen. We halen naar verwachting namelijk precies het deel eraf dat door hash-collisions is opgeteld. De gemiddelde fouten per T_i:



Figuur 15: 95% betrouwbaarheidsintervallen voor de maximale fout per T_i

Hieruit is duidelijk te merken dat de gemiddelde fout relatief erg laag is, voor alle waarden. Ze liggen allemaal vrij dicht bij elkaar, wat ook te verwachten is, want de enige fout komt nu door overschatting met woorden met $T(k) > 1$. Dat zal voor alle frequenties hetzelfde zijn. De zuivere schatter is weliswaar gemiddeld beter bij $T_i = 1$, maar zijn hoge uitschieters is juist iets wat je niet wil hebben.

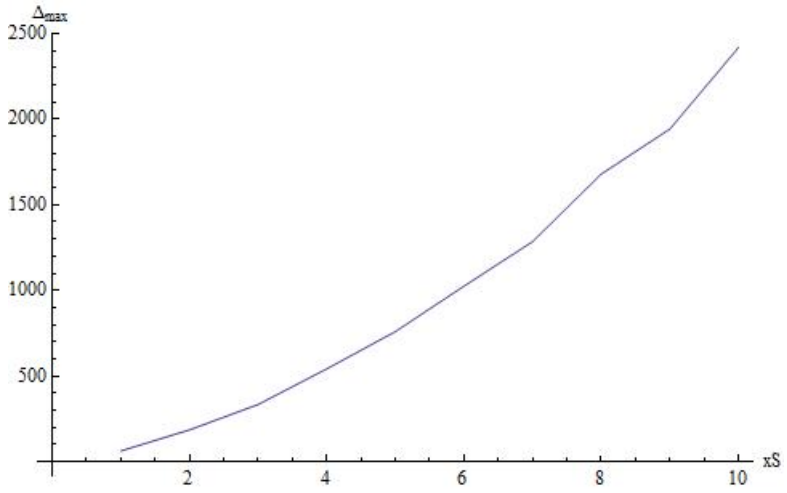
Als we d ophogen, zal het aantal hash-collisions toenemen, maar wordt de kans steeds kleiner dat een woord elke keer gehashed wordt met een woord met hoge frequentie. Omdat je het aantal hash-collisions er toch afhaalt, is de verwachting dat de maximale fout daalt als we d ophogen, totdat w kleiner wordt dan het aantal woorden met hoge frequentie. Dit aantal ligt bij Zipf-verdeling erg laag en als je deze kan schatten hoef je w ook maar een klein gedeelte hoger te kiezen. De grafiek van d ten opzichte van Δ_{max} ziet er zo uit:



Figuur 16: Maximale fout bij toenemende d

Het verschil met de Count-Min sketch is hier dus heel duidelijk te zien, want in tegenstelling tot de Count-Min sketch blijft deze grafiek dalen en volgt het de lijn $f(x) = \frac{1000}{x}$, want de stijging van het aantal hash-collisions komt dus niet voor. De grafiek zal dus pas weer gaan stijgen als w kleiner wordt dan het aantal woorden met hoge frequentie. Waar dat ligt verschilt en deze grafiek is dus per dataset verschillend. Als je het aantal woorden met hoge frequentie echter kunt schatten, kan je je sketch dus zeer nauwkeurig maken.

Deze sketch haalt dus het aantal hash-collisions er ongeveer af en dat werkt erg goed als dat gelijk is aan je overschatting, dus als veel woorden één keer voorkomen. Hebben we echter een random dataset, dan werkt dat niet zo goed meer:



Figuur 17: Maximale fout bij toenemende $|K|$ en $\max_k T(k)$ met uniforme, random data

De verwachting van de fout is wederom het aantal hash-collisions keer de gemiddelde frequentie. Bij de Count-Min sketch volgde dat dus de lijn $f(x) = 125x^2$. Omdat we nu het aantal hash-collisions reduceren met een factor $\frac{1}{4}$, volgt deze grafiek ongeveer de lijn $f(x) = \frac{1}{4} * 125x^2 = 37.5x^2$. Daar blijft de lijn dus onder.

5 Conclusie

Er zijn verschillende manieren met elkaar vergeleken in de uitwerking van dit probleem. Door middel van analyse en simulatie is per methode gekeken hoe goed het werkt. Misschien wel de meest belangrijke conclusie is dat de dataset waarover je de sketch wilt maken, allesbepalend is voor hoe goed een methode werkt. De dataset waar de meeste resultaten op gebaseerd zijn, is Zipf-verdeeld. Deze verdeling komt veel voor als men kijkt naar de frequentie van hoe vaak een woord voorkomt in een stuk tekst en is dus aannemelijk voor het probleem van Coosto. De aanpassingen die gedaan zijn op de bestaande methoden, het minimum van alle waarden in een rij erafhalen én het verhogen van d, werken ook alleen goed bij deze verdeling. Voor een willekeurige dataset werkt het veel minder goed, dat is duidelijk terug te zien in figuur 17. Voor het probleem van Coosto kunnen we wel concluderen dat de Count-sketch met collision reductie het beste werkt.

De krachtige eigenschap van een zuivere schatter is goed zichtbaar in de resultaten. De gemiddelde fouten liggen laag en in het geval van een algemene dataset werkt het ook goed. Het is echter belangrijk om te beseffen dat deze methode extreme waarden kan genereren. Als je een maximale tolerantie hebt voor een fout, heb je weinig aan het "gemiddelde geval". Als er helemaal geen kennis is van de data én je tolerantie is redelijk hoog, dan is dit een prima methode.

De meestgebruikte methode, de Count-min sketch, kan dus zowel in het algemene geval als in het geval van de Zipf-verdeelde dataset, goed verbeterd worden. De analyse van deze methode is echter wel het simpelst en je kan eenvoudig bepalen hoe groot je sketch moet zijn. Zeker ten opzichte van de zuivere schatter is dat een pluspunt.

6 Discussie

De resultaten van de methode bij de simulaties, is slechts een greep van wat je allemaal kan vergelijken. Het geeft zeker een goede indicatie en er zijn belangrijke eigenschappen en conclusies uit te trekken, maar hier is nog veel meer in te bekijken. Een vervolgstudie kan bijvoorbeeld zijn om in de simulatie datasets nog meer te gaan variëren of nog beter: om real-life data erin te gaan betrekken. De dataset is namelijk allesbepalend en op die manier kan je nog beter je conclusies trekken over wat de beste methode is voor een probleem. Maar ook in de grootte van de dataset en de sketch kan nog meer gevariëerd worden. De grootte van de sketch bepalen is een erg belangrijk aspect en ook dat kan per dataset verschillend zijn. Nu is de grootte alleen bepaald voor het algemene geval, maar het is natuurlijk veel interessanter als je de grootte kan bepalen voor de dataset waar het in gebruikt wordt. Ook daar zou je met real-life data moeten werken.

7 Referenties

[1]Zipfs law and the Internet
Lada A. Adamic en Bernardo A. Huberman; 2002

[2]An Improved Data Stream Summary: The Count-Min Sketch and its Applications
Graham Cormode en S. Muthukrishnan; 2003

[3]Lossy Conservative Update (LCU) sketch: Succinct approximate count storage
Amit Goyal en Hal Daumé III

[4]New Estimation Algorithms for Streaming Data: Count-min Can Do More
Fan Deng en Davood Rabei

[5]Improving Sketch Reconstruction Accuracy Using Linear Least Squares Method
Gene Moo Lee, Huiya Liu, Young Yoon en Yin Zhang; 2005

[6]http://en.wikipedia.org/wiki/Bloom_filter

[7]Finding Repeated Elements
J.Misra en David Gries; 1982

8 Bijlage

8.1 Pythoncode: Count-min sketch

```
BIG_PRIME = 9223372036854775783

def random_parameter():
    return random.randrange(0, BIG_PRIME - 1)

class Sketch_CM:
    def __init__(self, w, d):
        self.w = w
        self.d = d
        self.hash_functions = [self.__generate_hash_function() for i in range(self.d)]
        self.count = [[0 for x in range(self.w)] for x in range(self.d)]

    def update(self, key, increment):
        for row, hash_function in enumerate(self.hash_functions):
            column = hash_function(abs(hash(key)))
            self.count[row][column] += increment

    def get(self, key):
        value = sys.maxsize
        for row, hash_function in enumerate(self.hash_functions):
            column = hash_function(abs(hash(key)))
            value = min(value, self.count[row][column])
        return value

    def __generate_hash_function(self):
        a, b = random_parameter(), random_parameter()
        return lambda x: (a * x + b) % BIG_PRIME % self.w
```

8.2 Pythoncode: Count sketch met zuivere schatter

```
class Sketch_EST:
    def __init__(self, w, d):
        self.w = w
        self.d = d
        self.hash_functions = [self.__generate_hash_function() for i in range(self.d)]
        self.count = [[0 for x in range(self.w)] for x in range(self.d)]
        self.counting = True
        self.N = 0

    def update(self, key, increment):
        if not self.counting:
            self.counting = True
        for row, hash_function in enumerate(self.hash_functions):
            column = hash_function(abs(hash(key)))
            self.count[row][column] += increment

    def get(self, key):
        value = 10000
        self.arr = []
        if self.counting:
            self.counting = False
            self.N = 0
            for j in range(self.w):
                    self.N += self.count[0][j]
        for row, hash_function in enumerate(self.hash_functions):
            column = hash_function(abs(hash(key)))
            est = self.count[row][column]-(self.N-self.count[row][column])/(self.w-1)
            self.arr.append(est)
        return max(math.floor(self.median()),1)

    def __generate_hash_function(self):
        a, b = random_parameter(), random_parameter()
        return lambda x: (a * x + b) % BIG_PRIME % self.w

    def median(self):
        for i in range(self.d):
            current = self.arr[i]
            j = i - 1
            while j >= 0 and current < self.arr[j]:
                self.arr[j+1] = self.arr[j]
                j-=1
            self.arr[j+1] = current
        return self.arr[math.floor(self.d/2)]
```

8.3 Pythoncode: Count sketch met collisionreductie

```
BIG_PRIME = 9223372036854775783

def random_parameter():
    return random.randrange(0, BIG_PRIME - 1)

class Sketch_CM:
    def __init__(self, w, d):
        self.w = w
        self.d = d
        self.hash_functions = [self.__generate_hash_function() for i in range(self.d)]
        self.count = [[0 for x in range(self.w)] for x in range(self.d)]
        self.counting = True
        self.arr = []
        self.min = 0

    def update(self, key, increment):
        if not self.counting:
            self.counting = True
        for row, hash_function in enumerate(self.hash_functions):
            column = hash_function(abs(hash(key)))
            self.count[row][column] += increment

    def get(self, key):
        value = sys.maxsize
        if self.counting:
            self.arr = []
            self.counting = False
            for i in range(self.d):
                self.min = sys.maxsize
                for j in range(self.w):
                    self.min = min(self.min,self.count[i][j])
                self.arr.append(self.min)
        for row, hash_function in enumerate(self.hash_functions):
            column = hash_function(abs(hash(key)))
            value = min(value, self.count[row][column]-self.arr[row])
        return max(value,1)

    def __generate_hash_function(self):
        a, b = random_parameter(), random_parameter()
        return lambda x: (a * x + b) % BIG_PRIME % self.w
```