Векторен модел за анализ на текст

Векторният модел за анализ на текст (vector space model) [1] представлява неявен метод за описание, при който смисълът на документа се описва (или извлича) от самото му съдържание.

Както и името му говори, според векторния модел, документите, както и заявката, се представят във вид на вектори. Но за разлика от моделите за представяне, базирани на теория на множествата, тук векторите не са бинарни, а съдържат реални числа, представляващи теглата на думите (*term weights*).

Например:

d_1 = {0.15, 0.83, 2.12, 0.87, 0, 0, 0, 0.43, 0.25, 0, 0, … }

Важно е обаче да се отбележи, че *размерът на вектора не е равен на броя на думите в документа*, *а на броя на уникалните думи в цялата колекция от документи*. Ако броят на уникалните думи в колекцията е *n*, тогава всички вектори, представляващи документите ще бъдат представени в *n*-мерното пространство. Т.е. ще имат по n на брой елементи. Това на практика означава, че много по-голямата част (над 90%) от елементите на всеки вектор ще бъдат нули, защото повечето думи изобщо няма да се срещат в съответния документ, т.е. теглото им по отношение на него ще бъде 0.

Една сравнително малка колекция от около 5000 документи, всеки от по 150-200 думи, обикновено води до над 20 000 уникални думи. Работата с множество вектори, състоящи се от десетки, дори стотици, хиляди елементи *изисква много памет и изчислителни ресурси*. Затова на практика моделът се реализира по малко по-различен начин с използването на *инвертиран/обърнат индекс*. Как точно става това е подробно описано в края на документа.

Точността на изчисления коефициент на подобие между два документа (или между заявката и даден документ) до голяма степен зависи от начина на изчисляване на теглата на думите. В научната литература има предложени множество модели за изчисляването им. На английски се намират *term-weighting models*. Всички те се базират на две основни характеристики:

- Честота на поява на дадена дума t_i в рамките на *j*-тия документ d_j. Нарича се *term-frequency* – $\boldsymbol{tf_{i,j}}$. Това е *локална характеристика* на думата, в рамките само на текущия документ. Презумпцията е, че колкото по-често се среща дадена дума в рамките на документа, толкова по-важна е тя за неговото описание и смисъл.
- Брой на документите, съдържащи дадената дума t_i. Нарича се *document frequency* – $\boldsymbol{df_i}$. Представлява *глобална характеристика* на думата по отношение на цялата колекция от документи. За разлика от *tf* обаче, това е обратно пропорционална мярка за информативността на думата. Т.е. в колкото повече документи се среща тази дума, толкова по-малко конкретна информация носи тя. Защо? Защото думите, които се срещат във всички документи, без значение от тяхната тематична насоченост, обикновено са съюзи, предлози, определителни членове (на

английски) и др. Това са т.нар. *семантично незначими* думи. Затова при изчисляването на теглата на думите се използва не тази характеристика, а нейната „инверсия“ – *inverse document frequency* – ***idf***$_i$. Презумпцията тук е, че в колкото по-малко документи участва думата, толкова по-голяма конкретика и специфичен смисъл носи. *idf* не е 1 върху *df*, а се изчислява чрез уравнение 1.

В общия случай теглото на думите се изчислява чрез произведението на *tf* и *idf*. Само, че те обикновено не се използват директно, със „суровите“ (raw) си стойности. Защо? Представете си, че търсите „голяма пица“. Ако в документа d_1 „пица“ се среща 1 път, а в документа d_2 – 10 пъти, тогава ясно, че d_2 трябва да има по-голяма степен на подобие със заявката, но не чак 10 пъти по-голяма. Освен това една дума, която присъства в документа много пъти, може значително да намали тежестта на останалите думи, което разбира се не е особено желателно. Затова, при изчисляването на теглата, *tf* обикновено се логаритмува. Подобно и при *idf*. Там логаритмуването е задължително (уравнение 1), защото в противен случай, ако колекцията от документи е много голяма, *idf* може да има несъразмерно по-голяма стойност от *tf*.

$$idf_i = \log(\frac{d}{df_i}) \quad (1)$$

където:

idf_i – инверсна честота на поява на думата t_i в цялата колекция от документи.

d – брой на документите в колекцията.

df_i – брой на документите, които съдържат t_i.

Най-простата tf-idf схема за изчисляване на теглата на думите е:

$$w_{i,j} = tf_{i,j} * idf_i = \left(tf_{i,j}\right) * \log\left(\frac{d}{df_i}\right) \quad (2)$$

където:

$w_{i,j}$ – теглото на *i*-тата дума в *j*-тия документ.

$tf_{i,j}$ – честотата на поява на думата t_i в *j*-тия документ.

Останалите означения са вече изяснени от предходното уравнение

Както стана дума обаче, *tf* обикновено също се логаритмува, а не се използва директно. Така най-често използваната tf-idf схема, всъщност се явява уравнение (3). Тя съвсем не е перфектна, но се приема за *базова*, тъй като е лесна за реализация и всъщност дава добри резултати. Съществуват и други, някои от които са разгледани в следващата точка.

$$w_{i,j} = \left(1 + \log(tf_{i,j})\right) * \log(\frac{d}{df_i}) \quad (3)$$

Разбира се теглата на едни и същи думи в различните документи ще бъдат различни, защото в тях участва и локалната честота на поява на съответните думи в рамките на дадения документ (*tf*).

След като се изчислят теглата на думите и се формират векторите на документите, степента на подобие между тях може да се намери чрез различни мерки за сходство, включително алгебричните версии на Dice (уравнение 4) и Jaccard (уравнение 5), но обикновено се използва т.нар. косинусово подобие (*cosine similarity*) – уравнение 6.

$$Sim_{Dice}(q, d_j) = \frac{2\sum_{i=1}^{n} w_{qi} w_{d_j i}}{\sum_{i=1}^{n}(w_{qi})^2 + \sum_{i=1}^{n}(w_{d_j i})^2} \quad (4)$$

където:

$Sim(q, d_j)$ – степента на подобие между заявката *q* и документа d_j.

w_{qi} – теглото на *i*-тата дума от вектора на заявката.

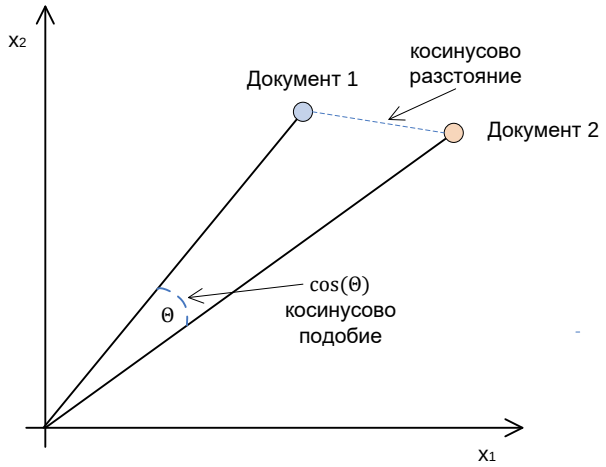
$w_{d_j i}$ – теглото на *i*-тата дума от вектора на *j*-тия документ.

n – броя на уникалните думи в цялата колекция от документи.

$$Sim_{Jaccard}(q, d_j) = \frac{\sum_{i=1}^{n} w_{qi} w_{d_j i}}{\sum_{i=1}^{n}(w_{qi})^2 + \sum_{i=1}^{n}(w_{d_j i})^2 - \sum_{i=1}^{n} w_{qi} w_{d_j i}} \quad (5)$$

Косинусовото подобие изчислява косинуса на ъгъла между двата вектора. Ако ъгълът е 0 градуса, т.е. векторите съвпадат, косинусът (подобието) е 1. Ако ъгълът между векторите е 90 градуса, т.е. те нямат нищо общо помежду си, косинусът е 0. Идеята е илюстрирана на фигура 1, за случая с вектори, представени в двумерното пространство. Разбира се, ако измеренията са над 3, няма как да се изчертае подобна графика, но не е и нужно.

$$Sim_{cosine}(q, d_j) = \frac{\sum_{i=1}^{n} w_{qi} w_{d_j i}}{\sqrt{\sum_{i=1}^{n}(w_{qi})^2}\sqrt{\sum_{i=1}^{n}(w_{d_j i})^2}} \quad (6)$$



Фигура 1. Илюстрация на идеята за намиране на подобие (cosine similarity) между два документа чрез косинуса на ъгъла между техните вектори.

Числителят на уравнение 6 представлява *скаларното произведение* на двата вектора - сума от позиционното произведение на всеки елемент от единия вектор с елемента, стоящ на същата позиция, в другия вектор.

Знаменателят представлява т.нар. косинусова нормализация - получената стойност от числителя се нормализира с произведението от дължините на двата вектора. Дължината на вектора е корен квадратен от сумата на всички елементи на квадрат. Защо е необходима нормализация? Защото колкото по-дълъг е даден документ, толкова по-голяма вероятност има в него да се намират думи от заявката, и то срещащи се многократно, което да доведе до по-голямото ѝ подобие с него, отколкото с друг по-къс документ, който примерно

съдържа същите думи, но повтарящи се по-малко. Тъй като скаларното произведение е сума от произведения, то може да нараства неограничено много (особено при думи с високи тегла), което затруднява съпоставянето и подредбата на отделните коефициенти на подобие между документите. Затова е необходимо всички изчислени подобия да се изменят в един и същ интервал, примерно от 0 до 1, за да могат в последствие обективно да се сравняват един с друг. Именно това прави нормализацията - „вкарва“ получената стойност в интервала от 0 до 1, без значение от дължината на заявката и на документите.

По принцип е възможно теглата на думите във векторите да се нормализират и преди да се изчисли подобието. Това се прави, когато е необходимо да се приложи (и) друг вид нормализация. В този случай знаменателят от уравнение 5.6 може и да отпадне, но не е задължително – зависи от това каква нормализация е приложена върху думите.

Пример: Нека са дадени следните два вектора

d_1 = {0.15, 0.83, 2.12, 0.87, 0, 0, 0, 0.43, 0.25, 0, 0}

d_2 = {0.67, 0, 0, 0.19, 0, 0.98, 1.27, 0.43, 0.53, 0, 0}

Тогава

Числителят = 0.15*0.67 + 0.83*0 + 2.12*0 + 0.87*0.19+ 0*0 + 0*0.98 + 0*1.27 + 0.43*0.43 + 0.25*0.53 + 0*0 + 0*0 = 0.1005 + 0 + 0 + 0.1653 + 0 + 0+ 0 + 0.1849 + 0.1325 + 0 + 0 = 0.5832

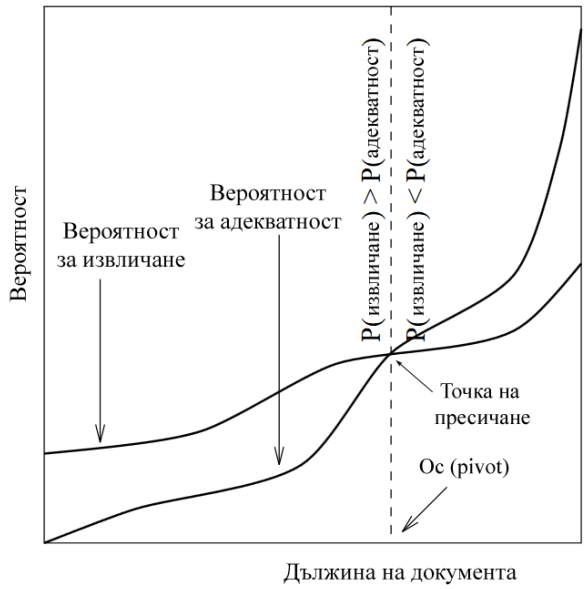
Знаменателят = 6.2101 * 3.5241 = 21.8850

или

$Sim(d_1 , d_2)$ = 0.5832/21.8850 = 0.026648

Модели за изчисляване на теглата на думите

Тъй като подобието между два вектора се намира чрез скаларното им произведение, то по-дългите документи, които имат по-голям брой ненулеви елементи във векторите си, логично ще дават и по-голяма степен на подобие със заявката. Затова именно е налична нормализацията в уравнение 6, за да редуцира този нежелан ефект и влиянието на дължината на векторите. Amit Singhal обаче забелязва, че в резултат на косинусовата нормализация късите документи дават по-големи коефициенти на подобие със заявката, и следователно имат по-голяма вероятност за извличане (намиране) [2]. В опити да реши този проблем, той сравнява вероятността даден документ да бъде извлечен с вероятността той да бъде адекватен, в зависимост от дължината му. Ако въпросните вероятности се изчертаят в двумерното пространство, в идеалния случай двете криви би трябвало да съвпадат. Те обаче не съвпадат, а се пресичат в точка, наречена *pivot* (на български – повратна точка, център, ос на въртене) – фигура 2.



Фигура 2. Криви, описващи вероятността за извличане и вероятността за адекватност на документа в зависимост от дължината му. Илюстрация на идеята за „завъртяна“ (pivoted) нормализация. Графиката е заимствана от публикацията на Singhal [2], в която предлага модела си.

От графиката се вижда, че преди точката на пресичане (т.е. при по-късите документи), вероятността документът да бъде извлечен е по-голяма от тази да бъде адекватен. А след точката на пресичане – обратно. Т.е. след косинусовата нормализация, своеобразен „бонус“ получават късите документи. Та идеята му е да се намери такава корекция на нормализацията, че кривата, описваща вероятността за извличане *да се завърти около точката на пресичане* и да съвпадне по-точно с вероятността за адекватност на документа. За целта Singhal предлага върху изчисленото вече тегло на думата да се приложи корекцията от (уравнение 7) [2]. В резултат подобието с късите документи намалява, а с дългите се увеличава. В публикацията си Singhal не посочва конкретен модел за изчисляване на теглата на думите преди корекцията, но споменава, че традиционната логаритмична нормализация на tf (т.е. $1 + \log(tf_{i,j})$) върши добра работа.

$$w_{i,j} = \frac{tf\_idf\ weight}{1 - s + s\frac{dl(d_j)}{avdl}} \qquad (7)$$

където:

s – наклон на завъртането. $s \in [0,1]$. По подразбиране s = 0.2.

dl(d$_j$) – дължина на текущия (*j*-тия) документ.

avdl – средно-аритметична дължина на документите (за цялата колекция). Точно тази средно-аритметична дължина определя проекцията на точката на завъртане върху абсцисата. Определя се като средно-аритметичен брой символи в документите или средно-аритметичен брой уникални думи в документите.

Стойността на наклона от 0.2 е предложена от самия Singhal и в последствие експериментално е потвърдена, като най-подходяща, от множество учени и експерти. При проведените собствени експериментални изследвания се оказа, че за конкретната предметна област (изчисляване на подобия между доклади и рецензенти), стойността на наклона няма съществено значение. Да, различните стойности на s, водят до различни коефициенти на подобие, но отношението между тях почти във всички случаи се запазва същото. А отношението между коефициентите е това, което определя подредбата на резултатите.

В по-късна публикация [3], Singhal предлага tf да се нормализира, чрез двукратно логаритмуване, което е особено ефективно за случаите, когато едни и същи думи се срещат много на брой пъти в едни и същи документи. Така моделът за изчисляване на теглата на думите, който предлага, придобива вида (8).

$$w_{i,j} = \frac{1 + \log\big(1 + \log(tf_{i,j})\big)}{1 - s + s\frac{dl(d_j)}{avdl}} \log(\frac{d+1}{df_i}) \tag{8}$$

Уравнение 5.8 представлява модел за изчисляване на теглата на думите с композитна (съставна) *tf* функция - първо се реализира двукратна логаритмична нормализация на *tf*, след което и предложената *„завъртяна“ нормализация* (pivoted normalization). Последната се нарича още нормализация по средната дължина на документите, тъй като оста се избира така, че да пресече абсцисата в мястото, което съответства на средно-аритметичната дължина на всички документи в колекцията. Дължината може да се определя от брой символи или брой уникални думи.

Stephen E. Robertson предлага алгебричен вариант на вероятностния си модел BM 25 [4,5] (уравнение 9), който също използва композитна *tf* функция, като първо се прилага нормализацията на Singhal, след което k-concavity нормализация.

$$w_{i,j} = \frac{(k_1 + 1)(tf_{i,j})}{k_1\left(1 - s + s\frac{dl(d_j)}{avdl}\right) + tf_{i,j}} \log(1 + \frac{d}{df_i}) \tag{9}$$

където:

k_1 – константа, установена на 1.2 по подразбиране.

Собствените ми експериментални изследвания показват, че всъщност това като че ли е най-добрият алгебричен модел за изчисляване на теглата на думите, използван при векторния модел за анализ на текст.

Rousseau и Vazirgiannis [6] предлагат друг модел за изчисление на теглата на думите (10), използващ съставна *tf* функция, който също разчита на нормализацията на Singhal по средната дължина на документите. При него първо се извършва именно тя, след това нормализация с поставяне на долна граница δ и накрая двойна логаритмична нормализация.

$$w_{i,j} = 1 + \log\left(1 + \log\left(\frac{tf_{i,j}}{1 - s + s\frac{dl(d_j)}{avdl}} + \delta\right)\right)\log(\frac{d+1}{df_i}) \quad (10)$$

където:

δ – долна граница за нормализиране. Всъщност е константа. Трябва да бъде 0.5, ако δ се прилага веднага след завъртяната нормализацията (както е в случая).

Първоначално въвеждане на долна граница δ предлагат Lv и Zhai [7]. С нея се опитват да решат следната ситуация, която би се получила при документи с много голяма дължина: Ако съществуват два документа и в единия се срещат повече думи от заявката, а в другия по-малко, но по-често, тогава подобието на двата документа със заявката би се получило горе долу съизмеримо, и не е ясно кой ще бъде подреден пред другия. С помощта на тази долна граница δ, Lv и Zhai искат да дадат приоритет на документа, който съдържа по-голям брой уникални думи от заявката, и той да получи по-висока степен на подобие с нея, отколкото другия документ, който съдържа по-малък брой думи от заявката, но пък които се срещат по-често в него.

Според експерименталните изследвания на Rousseau и Vazirgiannis, техният модел (10) дава малко по-добри резултати от BM 25, но моите експериментални изследвания не потвърждават напълно това твърдение. Да, в някои случаи техният модел превъзхожда BM25, особено в случаите без предварително стемиране на думите, но при стемиране в огромната част от случаите BM25 е по-добър.

Влияние на IDF върху точността и адекватността на извлечените резултати

Вероятно вече е направило впечатление, че в посочените по-горе модели инверсната честота на поява на дадена дума в документите (idf) се изчислява по различни начини.

$$idf_i = \log(\frac{d}{df_i}) \quad (11)$$

$$idf_i = \log(\frac{d+1}{df_i}) \quad (12)$$

$$idf_i = \log(\frac{d}{df_i} + 1) \quad (13)$$

$$idf_i = \log(\frac{d - df_i + 0.5}{df_i + 0.5}) \quad (14)$$

където

d – броя на документите в колекцията

df_i – броя на документите, които съдържат думата t_i.

Вариант (11) е така да се нарече „класическия“ или традиционния вариант за изчисляване на IDF. При него обаче има проблем – ако дадена дума се среща във всички

документи, тогава за idf ще се получи логаритъм от 1, което всъщност е 0. Т.е. в този случай, получената стойност за idf ще доведе до пълно *игнориране* думата – все едно, че тя изобщо не участва в заявката и/или документите. Това може да бъде проблем за някои силно специализирани колекции от документи, свързани с една или няколко много близки предметни области. И ако дадена заявка за търсене съдържа малко на брой думи, голяма част от които термини, които се срещат във всички документи, то заявката може да не върне нищо. Дори и да не се стигне точно до този сценарий, игнорирането на специализирани термини не е добра идея и в никакъв случай няма да допринесе за повишаване на точността на търсене. Затова, вариант (12) прави съвсем лека корекция, като добавяйки +1 в числителя, гарантира, че idf винаги ще има ненулева стойност и терминът, който се среща във всички документи ще получи много ниско тегло, но все пак няма напълно да се игнорира. Вариант (13) извършва по-сериозна (т.нар. изглаждаща) корекция, като силно намалява „обезличаването“ думата и дори тя да се среща във всички документи, вече няма да получи прекалено ниско тегло. По този начин всъщност се намалява влиянието на idf върху думите в заявката и/или документите.

Вариант (14) е първоначалният предложен от Robertson и Sparck Jones за BM25 модела. При него обаче също има проблем и Robertson го отбелязва в публикацията си [4] - ако дадената дума се среща в повече от половината документи, тогава idf ще стане отрицателна, т.е. думата не просто ще се игнорира, а присъствието ѝ ще води до по-голяма вероятност за **не**откриване (избягване) на документа. За да се преодолее този „странен“ ефект, Robertson предлага просто да отпадне df_i от числителя.

От представените варианти за изчисляване на idf, най-използваните са (12) и (13), тъй като те решават проблема с класическия вариант, без да създават нови.

Върху точността и адекватността на резултатите влияе не само начинът на изчисляване на idf, но също и това върху кои (типове) документи се прилага. От научната литература е известно (за справка в [8] или в [9]), че *най-добри резултатите се постигат когато idf се приложи <u>само</u> върху думите в заявката*, но не и върху думите в документите. И в това има логика. Проведените собствени експериментални изследвания също подкрепят това твърдение. Когато инверсната честота на поява (idf) се приложи върху думите в заявката, тя значително намалява теглото на тези, които се срещат във всички или в по-голямата част от документите. А често това са семантично незначими думи и части на речта като съюзи, предлози, наречия, местоимения и др., които не са били премахнати при предварителната обработка на текста. В този смисъл, прилагането на idf върху теглата на думите в заявката е резонно. Прилагането ѝ върху думите в документите обаче, може силно да редуцира теглото и на често срещани термини, които наистина описват предметната област на документите. Това е особено валидно в случаите на силно специализирани колекции от документи, за които стана дума по-рано. Всъщност при тях е възможно idf да се игнорира, дори и за думите в заявката. Или ако се прилага, то да се използват вариантите с изглаждаща корекция.

Системата SMART (System for the Mechanical Analysis and Retrieval of Text), разработена от университета Cornell в средата на 60-те години на миналия век, предоставя мнемоничен код, с който се описва как точно се изчисляват теглата на думите в документите и в заявката.

Въпреки, че нотацията (на английски SMART triple notation) е предложена преди повече от половин век, тя се използва и до днес. Състои се от две тройни компоненти във формат:

ddd.qqq

Първата компонента представя модела за изчисляване на теглата на думите в документите, а втората – теглата на думите в заявката. Всяка компонента се състои от три символа, които означават:

- Първият символ – дали и как се нормализира tf характеристиката (броя появи на думата t_i в документа d_j). Възможните стойности са: b (binary) – 1, ако думата присъства в документа, 0 – в противен случай; n – броя на появи на думата в документа; l – логаритмувана стойност на tf; d – двойно логаритмувана стойност на tf (като в (8)); a – „добавена“ нормализация (отношението на честотата на срещане на конкретната дума към броя появи на най-често срещаната дума в документа, с добавена константа).
- Вторият символ – дали се използва idf и ако да, как се изчислява. Възможните стойности са: n – не се прилага idf; f – idf във класическата си версия (11); t – idf във версия (12).
- Третият символ – дали се използва нормализация по отношение на дължината на документите и ако да, каква. Възможните стойности са: n – не се използва никаква нормализация (което е нереалистично); c – само косуносова нормализация; b – „завъртяната“ (осева) нормализация на Singhal (уравнение (7)); u – pivoted unique нормализация на Singhal. Последните две са почти идентични, като единствената разлика е, че при b средно-аритметичната дължина на документите се определя от броя символи в тях, а при u - от броя уникални думи. Т.е. u също отговаря на уравнение (7). Изпъкналата (k-concavity) нормализация на Robertson не присъства сред възможните стойности, защото е предложена доста по-късно.

Така например, ако теглата на думите, и в заявката, и в документите, се изчисляват по уравнение (3) и в последствие се използва косинусово подобие, то моделът се записва като **ltc.ltc**. Ако, както е препоръчително, idf се прилага само върху думите в заявката, но не и върху думите в документите, тогава моделът е **lnc.ltc**.

Друг пример – за моделът на Singhal (8), ако idf се прилага само върху думите в заявката, моделът е: **dnb.dtb** (ако средната дължина на документите се определя чрез броя символи в тях) или **dnu.dtu** (ако средната дължина се определя чрез броя на уникалните думи в документите).

Очевидно, тройната SMART нотация не би могла да се използва при модели, реализиращи композитна нормализация, съставена от три или повече нормализиращи функции (какъвто е (10) на Rousseau и Vazirgiannis), както и при модели, използващи нормализиращи функции, които не присъстват в нотацията (например BM25 на Robertson). В тези случаи нотацията, използвана от Rousseau и Vazirgiannis, която показва последователността на прилагане на нормализиращите функции и няма мнемонични кодове (т.е. ограничения) за тях, е много по-удобна.

Според нея, нормализираната TF характеристиката на уравнение (8), т.е. моделът на Singhal, се записва като $\mathbf{TF_{p \circ l}}$ – първо се извършва логаритмичната нормализация (*l*), след това „завъртяната“ нормализация на Singhal (*p*).

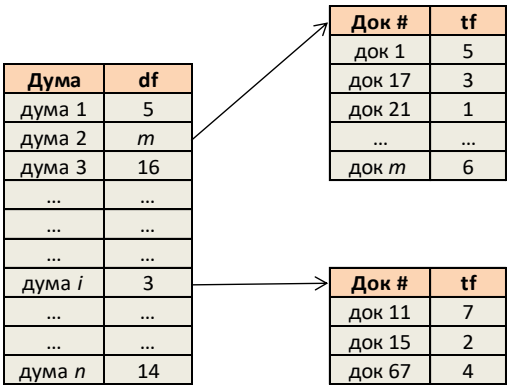
Нормализираната TF характеристика на BM25 се записва като $\mathbf{TF_{k \circ p}}$ – първо се извършва нормализацията на Singhal (*p*), след което k-concavity нормализацията на Robertson (*k*).

Тройно-нормализираната TF характеристика на модела на Rousseau и Vazirgiannis се записва като $\mathbf{TF_{l \circ \delta \circ p}}$ – първо се извършва нормализацията на Singhal (*p*), след това нормализацията с поставяне на долна граница (δ) и накрая логаритмичната нормализация (*l*).

Използване на инвертиран/обърнат индекс за по-ефективна реализация

При векторния модел за анализ на текст, всеки документ, включително и заявката, се представя във вид на вектор с толкова на брой елементи, колкото е броят на уникалните думи в цялата колекция от документи, т.е. обикновено десетки хиляди. Дори повече. При това около 99% от елементите в даден вектор вероятно ще бъдат нули, просто защото съответните думи не присъстват в документа изобщо, и техният tf=0. Но дори и нулата заема място и трябва да се обработва. Обработката всъщност е по-големият проблем. Ако броят на уникални думи в колекцията е 50 000 (традиционна ситуация), а колекцията съдържа 5000 документа, тогава за да се изчисли подобието между заявката и един документ трябва да се намери скаларното произведение между техните вектори и в последствие то да се нормализира. Но векторите са с размер 50 000 елемента. Т.е. трябва да се реализира един цикъл с 50 хиляди итерации, за да се намерят скаларното произведение и дължините на векторите. И това само, за да се изчисли коефициента на подобие на заявката с един единствен документ! После цялото това трябва да се повтори още 4999 пъти, за да се намери степента на подобие и с останалите документи. Очевидно това изчисляване ще се извърши много бавно. Но по-важното е обаче, че и до голяма степен е безсмислено, защото 99% от елементите във векторите ще бъдат 0, и съответно „локалното“ им позиционно произведение – също. Затова, за да се оптимизира практическата реализация на модела се използва т.нар. ***обърнат (инвертиран) индекс***.

Вместо да се съхраняват хиляди вектори с размерност от десетки хиляди елементи, които обаче са пълни почти изцяло с нули, се конструира *речник*. Все едно асоциативен масив, чийто индекси са самите уникални думи. Стойността на всеки елемент е друг масив, който показва в кой документ (d_j) се среща дадената дума ($term_i$) и колко пъти ($tf_{i,j}$). Идеята е представена на фигура 3.



Фигура 3. Обърнат (инвертиран) индекс.

Изграждането на инвертиран индекс е лесно – когато нов документ се добави в колекцията от документи, всяка негова дума се обработва поотделно. За всяка се проверява дали вече присъства в индекса. Ако да, обновява се нейната глобална честота на поява (df+1) и към масива, който съдържа информация в кои документи се среща тя и колко често, се добавя нов елемент с идентификатора на документа и локалната tf характеристика по отношение на този документ. Ако думата не присъства в индекса, тогава се добавя, глобалната ѝ df характеристика се установява на 1 и се създава (под)масив, чийто първи елемент съдържа идентификатора на документа, в който е открита, както и честотата ѝ на поява tf в него.

Фигурата показва наличието на множество масиви (броя на думите в речника + 1), от които левият е основният, в който се съхраняват думите, а в останалите се съхраняват идентификаторите на документите, в които се срещат тези думи и локалните им tf характеристики. На практика обаче този пълен речник може да се реализира и в една единствена динамична *многомерна* структура от данни от тип *асоциативен масив*, *обект* или *структура*.

Известно е, че асоциативните масиви обикновено се представят като хеш таблици. Елементите им не се съхраняват последователно в паметта, от където веднага отпада изискването те да имат еднакъв размер, а от там отпада и изискването те да бъдат от един и същ тип. Това позволява различните елементи в масива да бъдат от различен тип и да имат различен размер. Като не само стойностите могат да бъдат от различен тип, а и самите индекси (в асоциативните масиви, индексите се наричат *ключове*).

Инвертираният индекс може да се съхрани например в следния асоциативен масив:

```
index[<дума>][df] = df;
index[<дума>][documents][<docId_1>] = tf_1;
...
index[<дума>][documents][<docId_m>] = tf_m;
```

В случая масивът е тримерен. Индексът в първото измерение е самата дума (на съответния си език, стемирана или не). Във второто измерение, възможните индекси са два – df и documents. Елементът на df съдържа броя на документите, в които се среща тази дума. Елементът documents представлява друг едномерен масив, чийто индекси са идентификаторите на документите, в които се съдържа думата, а стойностите tf

характеристиките на конкретната дума, по отношение на конкретния документ. Например, ако *дума i* във фигурата е „ракета“ и тя, както е посочено на фигурата се среща в 3 документа (11, 15 и 67), това може да се запиша в тримерния асоциативен масив така:

```
index['ракета']['df'] = 3;
index['ракета']['documents']['док_11'] = 7;
index['ракета']['documents']['док_15'] = 2;
index['ракета']['documents']['док_67'] = 4;
```

Където „ракета“ е самата дума от речника, а „док_11“, „док_15“ и „док_67“ са реалните идентификатори на документите, в които се среща думата ракета.

Времевата сложност за създаване на индекса очевидно е линейна по отношение на общия брой думи (включително дублиранията) в колекцията от документи. Това на пръв поглед може да изглежда много, тъй като в една голяма колекция от дълги документи, броят на думите може да достигне стотици хиляди, дори милиони. Но първо зависимостта е линейна, и второ – така или иначе всяка една дума от всеки един документ трябва да бъда обработена индивидуално, независимо дали ще се търси чрез инвертиран индекс или ще се формират изключително дългите вектори на документите. Т.е. индивидуалната обработка на всяка една дума, от всеки един документ е неизбежна по принцип.

Изчисляването на степен на подобие между заявката и документите в колекцията чрез инвертиран индекс също е лесно за реализация, но не става чрез съвсем директно прилагане на формулите за подобие, защото при използване на инвертирания индекс де факто няма формирани вектори. Вместо това се изчислява подобие на всяка дума от заявката до всеки документ, който я съдържа, след което тези „частични“ подобия итеративно се наслагват, за да се получи пълното подобие между заявката и документите. Процесът е представен на фигура 4.

Основните структури от данни, използвани в псевдо кода на алгоритъма са:

`similarities[docId]` – масив, чийто индекси са идентификаторите на документите, а в стойностите на елементите се съхраняват с натрупване (кумулативно) скаларните произведения между заявката и съответните документи, в които се срещат думите от заявката. Тъй като думите в заявката се обработват итеративно, то до достигането на последната дума всяко скаларно произведение е „частично“, изчислено чрез сума от произведенията на теглата на вече обработените думи от заявката с теглата на същите думи в съответните документи. След обработване на всички думи и нормализация на пълните скаларни произведения, елементите в този масив съдържат крайните коефициенти на подобие между заявката query и всеки един от документите, определен чрез своя docId идентификатор. След края на алгоритъма, масивът **similarities** може да се сортира в низходящ ред преди извеждането на резултатите. По този начин най-подобните със заявката документи ще се подредят най-отгоре.

`documentLen[docId]` – масив, чийто индекси са идентификаторите на документите, а в стойността на всеки елемент се изчислява и съхранява дължината на съответния документ.

index - многомерен асоциативен масив, в който се съхранява инвертирания индекс. Структурата на този масив е детайлно описана по-горе.

calculateQueryWeight() – функция, която изчислява теглото на дадена дума от заявката, посредством нейните локална tf и глобална df характеристики. В тази функция може да се реализира който и да е модел за изчисляване на теглата на думите, описани по-рано.

calculateDocumentWeight() - функция, която изчислява теглото на дадена дума от j-тия документ, посредством нейните локална tf и глобална df характеристики. Тази функция целенасочено е дефинирана отделно от предходната, за да може теглата на думите в заявката и в документите да се изчисляват чрез различни tf-idf модели.

==========

```
// Заявката query се добавя в индекса
addDocument(query, index);

// Изчисляване на подобието между заявката
// и всеки един от документите.
// Резултатът се записва в similarities[docId] масива.

similarities = array(); // инициализиране като празен масив

documentLen = array();

queryLen = 0;

for всяка дума term_i от заявката query {

   if (term_i принадлежи на stopWords масива) {

      // тъй като думата е семантично незначима се прескача
      continue;

   }

   // tf_iq - tf (броя на появи) на i-тата дума в заявката
   tf_iq = index[term_i]['documents']['query'];

   // df_i - броя документи, които съдържат i-тата дума
   df_i = index[term_i]['df'];

   // Изчислява се теглото на i-тата дума w_iq от заявката
   w_iq = calculateQueryWeight(tf_iq, df_i);

   // изчисляване на дължината (без корен квадратен)
   // на вектора на заявката, т.е. първия множител
   // от знаменателя на ф-ла 6
   queryLen = queryLen + (w_iq * w_iq);

   for всеки документ с индекс docId_j, който съдържа
            думата, от index[term_i]['documents'] масива {

      if (несъществува similarities[docId_j]) {

         // в similarities[docId_j] се записва частичното
         // кумулативно подобие на обработените до момента
         // думи от заявката с документа docId_j

         similarities[docId_j] = 0;

         documentLen[docId_j] = 0;

      }

      tf_ij = index[term_i]['documents'][docId_j];

      // Изчислява се теглото на i-тата дума w_ij
      // от j-тия документ
```

```
      w_ij = calculateDocumentWeight(tf_ij, df_i);

      // Изчислява се частичното кумулативно
      // скаларно произведение на обработените
      // до момента думи от заявката
      // със същите от документа docId_j,
      // т.е. числителят на ф-ла 6

      similarities[docId_j] =
                    similarities[docId_j] + (w_iq * w_ij);

      // дължината на вектора на j-тия документ docId_j
      // т.е. втория множител от знаменателя на ф-ла 6
       documentLen[docId_j] =
                      documentLen[docId_j] + (w_ij * w_ij);

    } // endfor за всеки документ docId_j

} // endfor за всяка дума от заявката term_i

// Изчисляване на окончателната дължина на вектора на заявката
queryLen = sqrt(queryLen);

// До тук са изчислени числителят на ф-ла 6, както и двете
// суми в знаменателя. Следва да се извърши нормализацията,
// т.е. да се конструира цялата формула.

for всеки елемент с индекс docId от similarities масива {

   if (docId == 'query') continue;

   // намиране на окончателната дължина на документа docId
   documentLen[docId] = sqrt(documentLen[docId]);

   // изчисляване на косинусовата нормализация
   // т.е. знаменятеля на ф-ла 6
   cosNorm = queryLen * documentLen[docId];

   // нормализиране на скаларното произведение
   // с дължината на векторите (т.е. косунусова нормализация)
   similarities[docId] = similarities[docId] / cosNorm;

}
```

==========

Фигура 4. Алгоритъм за търсене по текстово съдържание чрез използване на инвертиран индекс

От къде идва подобрението/ускорението? От това, че скаларно произведение се изчислява не между вектори с размери броя на уникалните думи в *колекцията* (т.е. десетки хиляди), а между „вектори“ с размери броя на уникалните думи в *заявката*. Така, в най-лошия случай (ако всички документи съдържат думите от заявката), ще се изпълнят <*броя на документите*> * <*броя на уникалните думи в заявката*> итерации. За сравнение, ако подобието се изчислява по оригиналния вариант чрез вектори с дължина броя на думите в колекцията, то ще трябва да се изпълнят <*броя на документите*> * <*броя на уникалните думи в колекцията*> итерации. Броят на уникалните думи в колекцията обикновено е десетки хиляди, докато уникалните думи в заявката най-често варират от само няколко до максимум няколко стотин.

При подготовката на този файл са използвани материали от:

Калмуков, Й. Методи и алгоритми за търсене и извличане на документи. Издателство Primax Русе, 2022 г., ISBN 978-619-7242-93-5

Литература

1. Salton, G., A. Wong , C. S. Yang, A vector space model for automatic indexing, Communications of the ACM, v.18 n.11, p.613–620, Nov. 1975
2. Singhal, A., C. Buckley, and M. Mitra. Pivoted document length normalization. In Proceedings of SIGIR'96, pages 21–29, 1996.
3. Singhal, A., J. Choi, D. Hindle, D. Lewis, and F. Pereira. AT&T at TREC-7. In Proceedings of TREC-7, pages 239–252, 1999.
4. Robertson, Stephen. "Understanding inverse document frequency: on theoretical arguments for IDF." Journal of documentation, vol. 60 no. 5, pp 503–520, 2004.
5. Robertson, S. E., S. Walker, K. Spärck Jones, M. Hancock-Beaulieu, and M. Gatford. Okapi at TREC-3. In Proceedings of TREC-3, pages 109–126, 1994.
6. Rousseau, F., and M. Vazirgiannis. "Composition of TF normalizations: new insights on scoring functions for ad hoc IR." In Proceedings of the 36th international ACM SIGIR conference on Research and development in information retrieval, pp. 917-920. 2013.
7. Yuanhua Lv and ChengXiang Zhai. 2011. Lower-bounding term frequency normalization. In Proceedings of the 20th ACM international conference on Information and knowledge management (CIKM '11). Association for Computing Machinery, New York, NY, USA, 7–16. DOI:https://doi.org/10.1145/2063576.2063584
8. Manning, C., P. Raghavan, H. Schütze. An Introduction to Information Retrieval, Cambridge University Press, England, 2009.
9. Grossman, D., Frieder, O. Information Retrieval: Algorithms and Heuristics 2nd Ed. Springer, The Netherlands, 2004, ISBN: 1-4020-3004-5.