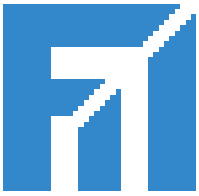
UNIVERSITATEA ALEXANDRU IOAN CUZA IAŞI
FACULTATEA DE INFORMATICĂ



LUCRARE DE LICENȚĂ

Rezumatul unui site

propusă de
Valentin-George Gainaț

Sesiunea: *iunie, 2015*

Coordonator științific
Conferențiar, Dr. Mihaela Breabăn

UNIVERSITATEA ALEXANDRU IOAN CUZA IAȘI
FACULTATEA DE INFORMATICĂ

Rezumatul unui site

Valentin George Gainaț

Sesiunea: *iunie, 2015*

Coordonator științific
Conferențiar, Dr. Mihaela Breabăn

Cuprins

1. Introducere

1.1 Context și motivație

În zilele noastre web-ul crește exponențial și majoritatea informațiilor sunt stocate on-line. Luând în considerare dependența de mediul online, lipsa de acces la aceste informații ar duce la un adevărat colaps. Perpetua sete de cunoaștere și nevoia de conservare a informațiilor a contribuit la această dezvoltare-rapidă mai ales în ultimele decenii-care a dus la apariția conceptului de big-data.

Ce este în principiu acest concept de big-data? Gândiți-vă la cei 50 Gb de fișiere prin 2002 care par acum de o mărime redusă față de o bază de date modernă-de marketing, spre exemplu-care foarte ușor în zilele noastre depășește 1Tb de date. Pentru ca aceste date sa fie disponibile se folosesc multe metode de obținere a lor, printre care si crawlerele web.

Un crawler web în primă fază ar avea o singură întrebuințare principală, putând fi redusă în termeni simpli la „a merge și a lua tot”. Dar având în vedere volumul foarte mare de date, un crawler simplu nu ar face față, sau dacă ar face față ar dura extrem de mult timp să ia și să proceseze aceste date; de aceea un crawler poate fi împărțit în mai multe crawlere care pot fi puse pe mai multe mașini virtuale și rulând în același timp, să returneze date la un crawler centralizat.

Revenind la partea de crawling, după cum spuneam, un crawler ar trebui să ofere date necesare utilizatorilor pentru prelucrare, în cel mai scurt timp. Dar acest cel mai scurt timp poate fi o problemă, având in vedere că unele servere mai mici nu pot susține cantitatea de request-uri venită de la crawler. De aceea în mod normal un crawler care nu este invadativ ar trebui să pună un accent mare pe această latură a crawling-ului-de a proteja și site-urile, prin atașarea unor timpi între cereri, prin verificarea fisierelelor de tip robots.txt.

Multe programe de crawling nu respectă aceste request-uri simple, în opinia mea. Din această cauză, au început să apară tot mai multe contramăsuri pe partea de web. Multe site-uri și-au dezvoltat metode proprii de protejare sau încetinire a crawlerelor. Unele din măsuri este de a bloca IP-ul de pe mașina de unde vin cererile, dar acest lucru poate fi combătut la rândul lui cu mașinile virtuale.

Amazon care pot face switch între ele cu IP-uri diferite prin intermediul unor proxy.

O altă masură întâlnită este de încetinire prin setarea unor cookie-uri care se verifică pe partea de server și se setează pe partea de browser la client. Această metodă permite servelor să încetinească răspunsurile trimise acelor cereri eliberând din banda alocată acelor cereri în acest timp, ex. Forbes.

În același timp majoritatea programelor de crawling pot seta un timp între requesturi pentru a faciliza funcționarea serverului cât timp se execută crawling-ul. Un astfel de timp se poate seta si din fisierele de tip robots.txt prin Crawl-delay ceea ce îi va spune crawler-ului să aștepte un anumit timp între cereri.

1.2 Scopul lucrării

Principalul obiectiv al acestui proiect este crearea unui crawler web care să ofere date în cel mai scurt timp, protejând în același timp și partea de server/site. Acesta va prezenta utilizatorului rezumatul unui site, oferit prin redarea unor cuvinte cheie care o să arate in timp real al crawling-ului stagiul în care evoluează algoritmii folosiți pentru crawling și sortare a cuvintelor.

Lucrarea redă aceste lucruri, este scrisă în framework-ul Rails și limbajul de programare Ruby, datele sunt trimise la server asincron fiind prezentate printr-un cloud de cuvinte realizat prin JavaScript.

Am ales acest limbaj deoarece momentan eu lucrez la anumite proiecte cu el, dar și din lipsa de dezvoltare a librariilor de crawling pe Ruby on Rails. De asemenea, o continuare a acestui proiect poate avea scopul de a crea o librărie cu utilizări multiple poate fi considerat un scop final al proiectului.

Inițial voi prezenta conceptele generale și metodele folosite pentru construirea acestui sistem, după care voi aprofunda fiecare metodă cu conceptul specific in parte, cât si redarea unor metode adiacente de realizare a acestor procedee.

1.3 Prezentare generală

Primul capitol redă o prezentare generală a conceputului de crawling însoțit de problemele generale legate de acest lucru. De asemenea, ne oferă și o idee generală despre continutul lucrării, plus limbajul folosit și concepte generale pe scurt.

În capitolul 2 sunt prezentate conceptele generale care sunt incluse în lucrare, precum Web Crawlers, Web mining, TF-IDF, dar și majoritatea modalităților de abordare a acestora. Din această prezentare se deprinde un schelet al aplicației, fiind urmată de prezentarea explicită a algoritmilor. Pe lânga Craling si web mining putem adăuga și big-data ca și concept general.

În capitolul 3 este prezentată arhitectura aplicației, fiind urmată de modul de construire a sistemului, modalități de implementare, prezentarea algoritmilor folositi și modalitățile de folosire urmate de câteva concluzii ale proiectului, dar și idei pentru viitoarele dezvoltări ale proiectului în capitolele 4 și 5.

2. Concepte și domenii generale

2.1 Web Crawler

Un crawler web este un program prin care în mod normal un utilizator descoperă și descarcă conținutul unui site prin intermediul unui protocol HTTP. Ca și concept general, un crawler realizează o pânză de păianjen construită dintr-un link sau un set de link-uri de start. O primă fază în construirea pânzei este reprezentată de descărcarea conținutului link-urilor de start, fiind urmată de accesarea fiecărui link din pagini. Primul pas este repetat recursiv pentru fiecare link accesat, în acest fel construindu-se o hartă a site-ului. Acest lucru se poate asemăna unei persoane care accesează link-urile unui site într-un browser.

Partea de content este destul de simplă având în vedere că acest lucru se realizează printr-un request HTTP, fiind urmată de un response HTTP și închiderea conexiunii. [1]

2.1.1 Tipuri generale

Un crawler web poate fi folosit în numeroase moduri. Un mod general de folosire a unui crawler web este de returna utilizatorilor link-uri specifice dintr-un site pe baza query-urilor. Acest mod de folosire a unui crawler este unul de bază, întâlnit la majoritatea programelor care fac crawling la site-uri.

O altă utilizare a unui crawler este de arhivare web. Această metodă este folosită pentru colectarea periodică a unor site-uri mari și adăugarea/eliminarea din baza de date a unor date specifice acelor site-uri.

Un crawler poate fi folosit și pentru data-mining, unde conținutul paginilor este analizat pentru crearea unor statistici, este folosit pentru a observa și colecta similaritățile dintre pagini. După acești pași analizele datelor sunt realizate pe rezultatele crawlerului. [1]

2.1.2 Tipuri specifice

În mod normal, un crawler web este folosit pentru indexarea unui site la un moment dat, acesta oferind date generale despre site, metadate. În general aceste crawlere pot varia foarte mult, având în vedere că se pot extrage foarte multe informații de pe un site care pot fi folosite in diferite cazuri. Cele mai populare crawlere pot fi considerate cele folosite de Google/Bing/Yahoo care fac crawling în datele publice de pe web oferind utilizatorilor site-

uri specifice pentru căutările lor.

Un alt tip de crawlere sunt acele crawlere care fac crawling pe anumite url-uri din site-uri în anumite porțiuni de timp cu scopul de a alarma utilizatorii de anumite schimbări în conținutul acelor site-uri. Aceste crawlere sunt întâlnite in special pe site-uri de shopping online unde sunt urmărite schimbările de preț sau sunt urmărite aparițiile unor produse.

Crawlere folosite de corporații sunt specializate în extragerea de informații din documente specifice. Aceste crawlere rulează de obicei ca și background jobs, care oferă rezultate particulare pentru companii. Cel mai important aspect al acestor crawlere este să înțeleagă pe deplin aspectele caracteristice fiecărui tip de document din care trebuie sa extragă informații, deoarece anumite tipuri de documente precum excel/zip și altele pot varia foarte mult; chiar și cea mai mică modificare survenită in structura documentului poate cauza o eroare uriașă la crawler și oferire de date eronate utilizatorului. [1]

2.1.3 Componente generale

Printre componentele generale ale unui crawler se pot include o coadă în care se țin URL-urile din site, un fetcher care rezolvă partea de descărcare a conținutului, un extractor care este responsabil cu căutările de noi pagini. Acest lucru se realizează prin parcurgerea paginilor și extragerea textului care este încadrat in tag-ul HTML <a>. Un ultim element necesar unui crawler este un content repository unde vom ține contentul paginilor. În acest fel serverul va avea acces la componentele de bază ale unui url de bază. Pe lângă aceste componente un crawler ar trebui să aibă o metodă de a verifica la orice site fisierul robots.txt, pentru a nu apărea probleme care țin de accesarea de conținut restrictiv de pe un site. [1]

2.1.4 Tehnici de crawling

Crawling de bază

Crawling normal (de bază) este bazat pe pe un algoritm ca și BFS. Ca și componență acest tip de crawling conține o coadă, Q, în care sunt adăugate URL-urile, două locații în unul fiind stocate documentele in celălalt fiind stocate URL-urile.

Dat un set inițial de URL-uri, la fiecare pas un URL este luat din coada Q, descărcat și parsat pentru căutarea de noi URL-uri in document care vor fi stocate in coadă doar dacă nu au fost adaugate pană acum.

Condiții de terminare: timpul alocat crawlerului a expirat, stocarea dedicată crawlerlui este plină, multiple URL-uri care conțin query-uri si documente de download.

Probleme: Timpul de crawlare nu poate fi estimat, din multiple cauze: căutarea DNS poate dura mult timp, congestia rețelei, întârzierile conectărilor, exploatarea benzii prin folosirea tread-urilor pentru crawlare multiplă. [2]

Crawling selectiv

Acest tip de crawling recunoaște importanța site-urilor, limitând descărcarea url-urilor doar la seturi importante de date. Crawl-ingul selectiv definește o funcție de relevanță a unui site:

$s_{\Theta}^{(\xi)}(u)$: where u is a URL,
ξ is the relevance criterion,
Θ is the set of
parameters.

Pe lângă relevanță, acest crawler oferă și următoarele funcții:

- măsurarea eficienței rt /t, t = #pages fetched, rt = #fetched pages with score > st (ideal rt =t)

- popularitatea unei pagini asignând importanță după popularitate:

$$s_{\tau}^{(backlinks)}(u) = \begin{cases} 1, \text{ if indegree}(u) > \tau \\ 0, \text{ otherwise} \end{cases}$$

- adancime: limitează documentele descărcate de pe un singur site prin setarea unui prag, prin setarea adâncimii în arborele de directoare sau prin limitarea lungimii url-ului: $s_{\delta}(depth)(u) =$

$$\begin{cases} 1, \text{ if } |\text{root}(u) \rightsquigarrow u| < \delta, \text{ root}(u) \text{ is root of site with} \\ 0, \text{ otherwise} \end{cases}$$

Crawlarea centralizată

Caută informații specifice pentru anumiți utilizatori. În acest caz relevanța poate fi dată de paginile returnate cu conținut specific căutării. Un crawler centralizat, poate fi considerat un crawler selectiv, bazat pe anumite query-uri.

Oferă relevanța prezicerilor prin definirea unui scor prin care un document este relevant pentru un text dat:

$$s_{\theta}^{(\text{topic})}(u) = P(c \vee d(u), \Theta)$$

c is topic of interest
Θ are adjustable params of classifier

Grafuri contextuale au ca avantaj cunoașterea topologiei internetului pentru a antrena sisteme de invățare automată în scopul de a găsi adâncimea de parcurgere a unui site pentru găsirea de informații relevante, pornind de la o pagina dată.

Învățarea automată reprezintă capabilitatea de a recompensa un crawler pentru descărcarea unui document relevant cu scopul de a maximiza recompensele pe crawling de termen lung. Crawler-ul învață din recompensările anterioare. [2]

Crawling distribuit

Reprezintă un sistem scalabil de a împarți crawling-ul prin metoda "divide-and-conquer". Când încercăm să realizăm un crawling pe site-uri extrem de mari într-un timp scurt apar multe probleme cum ar fi banda necesară pentru crawlerul central, eventualele întreruperi de conexiune dar și consumul de memorie. Contrucția unui sistem de crawlare prin procese multiple poate fi considerat o forma de "divide-and-conquer", rezultatul fiind un sistem scalabil. Interacțiunea dintre crawlere poate fi caracterizată prin: coordonare, limitare și partiționare. [2]

Dinamica Web-ului

Acest procedeu se referă la schimbările site-urilor în timp. Trei parametrii sunt folosiți pentru acest procedeu: timpul de viață a documentelor, prospețimea documentelor ceea ce ne spune dacă un document este up-to-date in indexare la un timp dat t și cât de recent este documentul. Indexul unul document reprezintă timpul de viata a documentului dacă ceea ce am stocat noi în baza de date este expirat și 0 dacă copia locală este nouă. [2]

2.1.5 Arhitectura

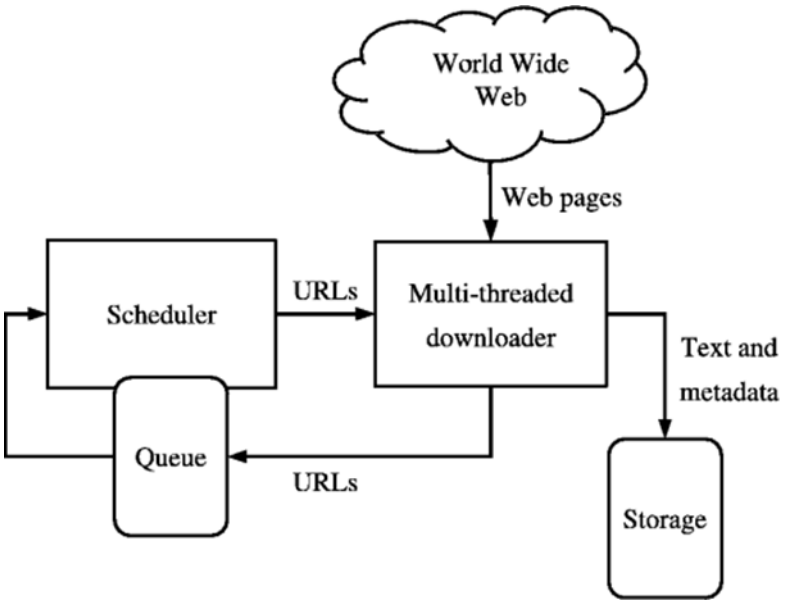


Figura 1: Arhitectura tipică a unui crawler web [2]

În mod general un crawler generează copii ale paginilor vizitate procesându-le cu un sistem de crawling care indexează paginile descărcate pentru a putea fi procesate rapid după care vor fi stocate în baza de date anumite rezultate după preferințe. Cel mai cunoscut crawler este GoogleBot.

În mod normal un crawler neinvadativ, ar trebui să respecte un protocol general numit Robots exclusion standard. Acest standard apare in site sub forma unui fisier de tipul robots.txt care îi comunică crawler-ului web care din părțile unui site nu ar trebui să fie procesate. [4]

Acest fisier va fi adăugat in site in forma “https://www.example.com/robots.txt” si

poate avea forme ca și:

- site-ul poate fi vizitat de orice robot, având in vedere ca user-agent include toate url

  User-agent:*

  Disallow:

- un site care conține un fișier de tipul ăsta nu poate să I se aplice crawling

  User-agent: *

  Disallow: /

- un alt exemplu spune roboților sa ignore anumite url-uri

  User-agent: *

  Disallow: /cgi-bin/

  Disallow: /tmp/

- un alt tip de fișiere pot preciza și tipuri explicite de roboți pe care vor să îi oprească din a realiza crawling:

  User-agent: BadBot

  Disallow: /

2.1.6 Algoritmi de crawling

BFS. Paginile sunt descărcate în ordinea în care sunt descoperite și acest tip de crawling oferă o garanție de acoperire a site-ului mare.

Prioritizare după index. Paginile cu cele mai multe accesări în descărcările precedente sunt adăugate mai rapid în lista de decărcări.

Priotizare după PageRank. Paginile sunt descărcate în ordine descrescătoare după rank-ul paginii. Acesta este bazat pe paginile și link-urile descărcate până atunci de către crawler. Acest rank al paginilor poate fi calculat după fiecare decărcare sau periodic. [2]

2.2 Web Mining

2.2.1 Web Mining General

Web mining-ul reprezintă folosirea tehnicilor de data-mining pentru a descoperi și a extrage automat informații de la servicii si documente Web. Web mining-ul reprezintă integrarea informațiilor prin metodologiile tradiționale de data-mining și tehnologiile de obținere a informațiilor de pe world wide web. [5]

Conținutul paginilor web se obține prin următorii pași:

-colectarea: aducerea conținutului de pe web

-parsarea: extragerea de date utile din documentele formatate(HTML, PDF, etc)

-analizarea: clasificare, sortare, grupare, marcarea, adăugarea de scoruri

-produsul: transformarea rezultatelor în ceva folositor(rapoarte, indexări, etc) [6]

Web Mining-ul poate fi divizat in trei categorii având în vedere tipurile de date care pot fi extrase:

Web Mining pe conținut

Este procesul de colectare a datelor folositoare din contextul paginilor web. Este o colecție de "fapte" a paginilor și poate conține text, imagini, fișiere audio, video și structuri de date cum ar fi liste sau tabele. Cercetările asupra web mining-ului au realizat tehnici ca și Information Retrieval (IR) și Natural Language Processing (NPL) . Având în vedere că există foarte multă muncă în extragerea informațiilor din imagini în procesare de imagini, aplicațiile acestor tehnici au fost limitate. [5]

Web Mining bazat pe structură

Structura generală a unui arbore web este formată din paginile web ca și noduri și link-uri ca și muchii conectând paginile precizate. Web mining-ul bazat pe structură reprezintă procesul descoperirii informațiilor structurate de pe web. Acest domeniu poate fi împărțit în două subdomenii: [5]

-Link-uri : este o unitate structurală care conectează o locație din pagina web cu o altă locație

-Structura documentului: conținutul unui document, care la rândul lui poate fi organizat ca și un arbore, având in vedere că poate fi HTML sau XML care conține tag-uri.

Web Usage Mining

Reprezintă tehnica de a descoperi utilizări ale modelor de web usage data pentru a înțelege și servi necesitățile aplicațiilor. Usage data capturează identitatea sau originile utilizatorilor web, alături de comportamentul lor din browser pe un site. Web usage mining poate fi clasificat in funcție de datele luate:

-Web Server Data: log-urile utilizatorilor sunt colectate de serverul web, incluzând adresa IP, referința la pagină și timpul accesării

-Application Server Data: Serverele aplicațiilor comerciale pot activa aplicațiile E-commerce, deoarece au abilitatea de a urmări multe evenimente ale afacerilor.

-Application Level Data:

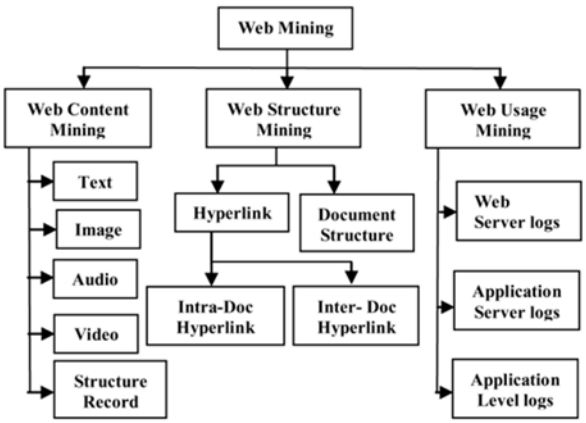


Figura 2: Taxonomia web mining [6]

Având în vedere că web-ul crește foarte repede, cresc și oportunitățile de analizare a lui și extragere de date in cât mai multe modalități. [5]

2.3 Big Data

2.3.1 Concepte Generale

Conceptul de big-data nu este străin, avand in vedere că de la începutul anilor '90 se creau locații speciale de a ține datele, doar că atunci 1Tb era considerat ca și big-data. În zilele noastre s-a ajuns la petabytes de date, spre exemplu eBay are flow de 1 Terabyte pe minut si susține 40 de petabyes de date.

Big data este un termen care este folosit pentru a descrie date într-un volum mare, velocitate ridicată și/sau varietate ridicată; are nevoie de tehnologiile si tehnicile noi pentru a fi capturată, prelucrată și analizată; și este folosită pentru a spori luarea de decizii, pentru a descoperi și pentru a suporta și optimiza procesele. [Mills, Lucas, Irakliotis, Rappa, Carlson, and Perlowitz, 2012; Sicular, 2013]

Este important de precizat că mărimea de acum la big-data în viitorul apropiat va fi considerată normal-data. [8]

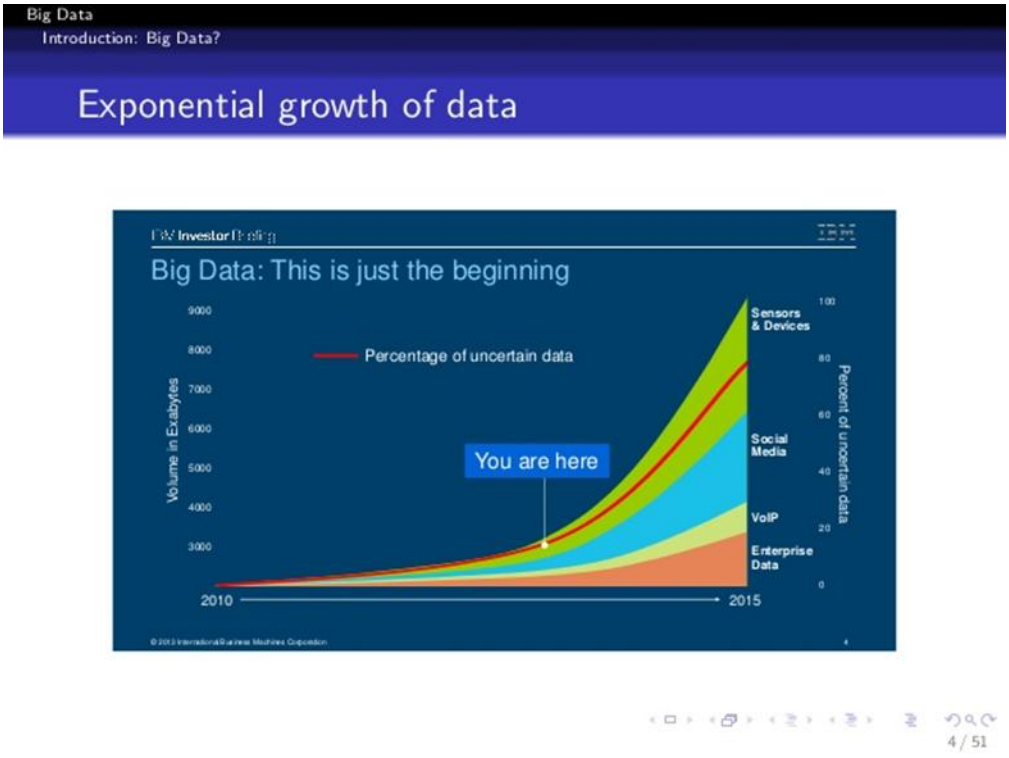


Figura 3: The Exponential Growth of Big Data (Source: Palfreyman, 2013)

2.3.2 Surse

Big-data poate fi colectată din foarte multe surse. De exemplu, orice click in browser al unui user poate fi capturat in log-urile serverelor și poate fi folosit spre exemplu în realizarea unor statistici în urma accesărilor anumitelor URL-uri ca după să îi fie recomandate anumite tipuri de pagini/site-uri. Facebook si Twitter produc zilnic foarte multe date prin mesaje/post-uri. Aceste date pot fi capturate și prelucrate pentru a crea și observa părerile utilizatorilor despre anumite topic-uri, despre anumite produse, dar se pot face si statistici.

Anumite programe generează metrici pe aceste statistici și în acest fel se creează o rețea pentru fiecare utilizator în parte, care conține toată activitatea lui și acesta reprezintă doar un nod dintr-o rețea uriașă. Tot in big-data se stochează și un număr mare de date geospațiale(GPS), cum ar fi datele create de telefoanele mobile. Aceste date pot fi folosite de multe aplicații care ajută utilizatorul sa se conecteze cu alte persoane din proximitatea sa care folosesc aceleași aplicații. Imaginile, videoclip-urile si datele audio pot fi folosite pentru algoritmi de face recognition and voice recognition, și în sisteme de securitate. [8]

2.3.3. Importanța crawling-ului în big-data

Companii ca și Google, Facebook, LinkedIn folosesc crawling-ul si big-data pentru a colecta o cantitate enormă de date. Spre exemplu Google a colectat un volum foarte mare de date prin crawling-ul a bilioane de site-uri din întreaga lume. Utilizând acele date și în combinație cu algoritmi și alte tipuri de date au reușit să creeze un sistem prin care au maximizat eficiența căutărilor și în același timp au crescut căutările specifice pentru fiecare user în parte. Prin acest sistem Google a obtinut un profit foarte mare pentru ei și pentru clienții lor Adwords. În același timp ei își mențin si utilizatorii fericiți. În același mod și Facebook-ul si LinkedIn si-au construit crawlerele lor specializate pentru a conecta utilizatorii în cel mai bun mod posibil și au folosit big-data pentru a stoca cât mai multe date despre utilizatori și în acest fel sunt capabili să ofere reclame și să recomande produse cât mai apropiate de cerințele utilizatorilor.

Pentru a construi un sistem la acest nivel sunt necesare foarte mult timp, efort si resurse financiare. Pentru aceste tipuri de sisteme ai nevoie de: inputul de la user(search-uri), un third party api cum ar fi facebook, twitter, google pentru colectarea informatiilor publice

despre user, server logs, log-uri de la servere ca si Apache, nginx pentru a vedea browsing-ul userului si web crawling sau scrapping. Crawling-ul pentru big-data poate fi extrem de costisitor și să dureze foarte mult timp. [7]

2.4 TF-IDF

TF-IDF prescutare de la term frequency-inverse document frequency, este o statistică numerică care dorește să reflecte cât de important este un cuvânt într-un document dintr-o colecție de documente. Este de obicei folosit ca de obicei în information retrieval și text mining. Valoarea tf-idf crește proporțional cu numărul apariției în document, dar este echilibrată de frecvența cuvântului în colecția de documente, care ajustează faptul că unele cuvinte apar mai mult in general.

Variațiile scorurilor tf-idf variază in mod normal în funcție motoarele care fac crawling pe site-uri și ce informații filtrează din link-urile descărcate, ce cuvinte de legătură elimină din context si ce sortări sunt aplicate pe text, având în vedere că fiecare programator poate adăuga propriile cuvinte sau poate oferi userului posibilitatea de a elimina cuvinte. [9]

2.4.1 Term frequency

Term frequency măsoară cât de frecvent apare un termen într-un document. Deoarece documentele variază foarte mult în dimensiuni, un termen poate apărea într-un document cu un scor foarte mare și în alt document cu un scor mult mai mic. Din cauza asta s-a introdus conceputul de normalizare care adaugă la frecvență și un divide cu numărul termenilor din document. [10]

-binar : frecvențele au doar două valori: 1 dacă termenul apare și 0 dacă termenul nu apare in document:

$$tf(t,d) = \{0,1\}$$

-frecvența normală:

$$tf(t,d) = f(t,d)$$

-frecvență scalată logaritmic:

$$tf(t,d) = 1 + \log f(t,d)$$

-frecvența augmentată: pentru a preveni o interferență în documentele foarte mari, e.g.

frecvențele normale divizate de frecvența maximă normală a oricărui termen din document:

$$\mathrm{tf}(t,d) = 0.5 + \frac{0.5 \times \mathrm{f}(t,d)}{\max\{\mathrm{f}(w,d) : w \in d\}}$$

-frecvența augmentată K, are aceeași formulă cu frecvența augmentată doar că nu este pentru o valoare specifică 0.5 ci este pentru o valoare K

2.4.2 Inverse document frequency

Frecvența inversă a unui document măsoară cât de multă informație cuvântul oferă, mai specific, ne spune dacă termenul este rar sau des intâlnit în toate documentele. Este fracția scalată logartmic a documentelor care conțin cuvântul, obținut prin divizarea numărului total de documente la numărul de documente care conțin termenul respectiv. [10]

$$\mathrm{idf}(t,D) = \log \frac{N}{|\{d \in D : t \in d\}|}$$

Variante ale IDF:

-unara: 1

-frecvență inversă

$$\log \frac{N}{n_t}$$

-frecvență inversă maxima

$$\log \left(1 + \frac{\max_t n_t}{n_t}\right)$$

-frecvență inversă probabilistică

$$\log \frac{N - n_t}{n_t}$$

2.4.3 Term frequency–Inverse document frequency

Tf-idf este calculată în modul următor:

$$\mathrm{tfidf}(t, d, D) = \mathrm{tf}(t, d) \times \mathrm{idf}(t, D)$$

Un tf-idf mare este rezultat printr-o frecvență mare în documentul dat și o frecvență joasă a termenului în toată colecția de documente. Având în vedere că rația din funcția logaritm la idf este întotdeauna mai mare decât 1, atunci valorile lui idf (si tf-idf) sunt mai mari decât 0. Dacă un termen apare în multe documente, rația din interiorul logaritmului se apropie de 1, ducând valoarile idf si tf-idf spre 0. [9]

Schema scorurilor	Scorul termenilor pe document	Scorul termenilor pe interogare
1	$f_{t,d} \times \log \frac{N}{n_t}$	$\left(0.5 + 0.5\frac{f_{t,q}}{\max_t f_{t,q}}\right) \times \log \frac{N}{n_t}$
2	$1 + \log f_{t,d}$	$\log(1 + \frac{N}{n_t})$
3	$(1 + \log f_{t,d}) \times \log \frac{N}{n_t}$	$(1 + \log f_{t,q}) \times \log \frac{N}{n_t}$

Scheme TF-IDF recomandate. [9]

2.5 Lucene

Lucene combină Boolean model (BM) of Information Retrieval cu Vector Space Model (VSM) of Information Retrieval. În VSM, documentele și cererile sunt reprezentate ca și Vectori ponderați într-un spațiu multi-dimensional, unde fiecare index reprezintă o dimensiune și coloanele sunt valorile tf-idf.

VSM nu are nevoie ca ponderile să fie valori tf-idf, dar valorile tf-idf sunt valori de calitate ridicată, așa că și Lucene folosește tf-idf.

Scorul VSM al unui document d pentru un query q este Similaritatea Cosine ai vectorilor ponderați V(d) si V(q): → cosine-similarity = V(q) * V(d) / |V(q)| |V(d)|

Unde V(q) * V(d) este produsul scalar dintre cei doi vectori, iar |V(q)| și |V(d)| sunt normele Euclediene.

Formula Lucene pentru aplicarea scorurilor este: score(q,d) = coord(q,d) * queryNorm(q) * ∑(tf(t in d) • idf(t)2 • t.getBoost() • norm(t,d))

-tf(t in d) → reprezintă fracvența termilor, definită ca și numărul în care termenul t apare cu un scor anume în documentul d

-idf(t) → reprezintă Inverse Document Frequency.

-coord(q,d) → este un scor bazat pe multitudinea termenilor queries care sunt găsiți în documentul respectiv.

-queryNorm(q) →reprezintă un factor de normalizare folosit pentru a crea scoruri între queriuri comparabile.

-t.getBoost → este un time boost a termenului t, în queriul q cum este specificat in textul queriului

-norm(t,d) → încapsulează câțiva indecsi pentru timpi si lungimi. [13]

3. Arhitectura aplicației

Sistemul contruit in proiect folosește algoritmul de crawling web pentru a lua toate paginile unui site. Acest algoritm presupune un un url/set de url-uri de start primit de la utilizator care va fi preluat de crawler si va executa urmatorii pasi:

- identificarea și indexarea paginilor de pe site-ul oferit prin link-uri;

-descărcarea contextului din pagini, a url-ului, a headerelor, la care se adaugă pagina de redirect în cazul în care există sau de eroare;

-pe lângă aceste funcționalități au fost adăugate și o functionalitate de a urmări automat redirectările către link-urile din același domeniu și o modalitate de a opri coada crawler-ului când aceasta depășește o limită de memorie;

-după descărcare vor fi stocate si codul de răspuns al paginii, dar și dacă a mai fost vizitată precum și timpul de răspuns;

După acești pași se selectează doar paginile care nu sunt link-uri de desărcat, aceasta poate fi un pas ulterior în proiect, care ne va da un scor mai precis pentru crawling.

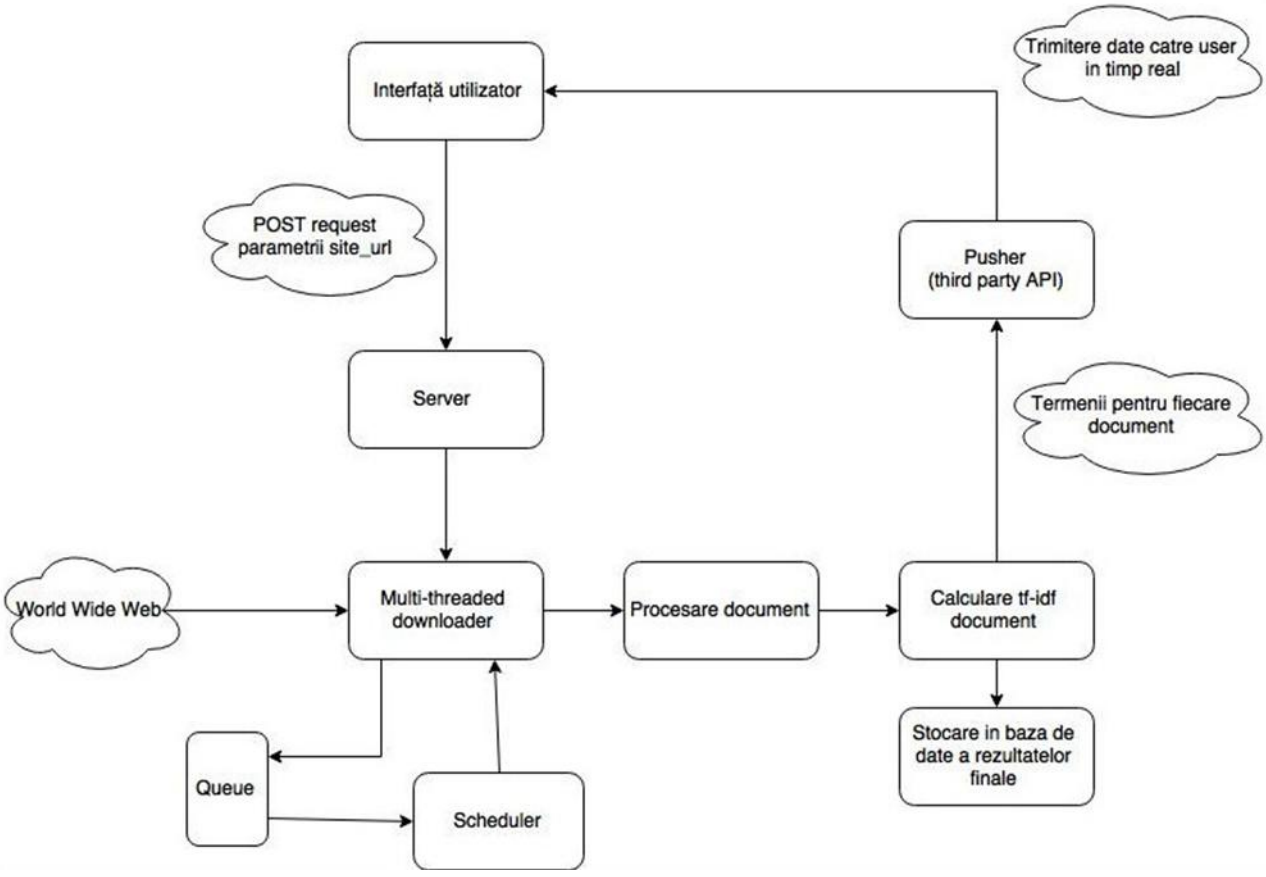
Procesarea documentelor, constă în eliminarea tag-urilor HTML și a scripturilor javascript din pagini, dar și eliminarea cuvintelor de legatură și eliminarea cuvântului din url, care este logic că va apărea de cele mai multe ori în pagini. După acești pași vom crea un document special tf-idf care va fi adăugat la o matrice formată din toate documentele, pentru a calcula scorurile fiecărui document în comparație cu toate documentele.

În baza de date momentan sunt stocate url-ul site-ului, termii site-ului ordonați după scurul obținut in ordine descrescătoare, data când a fost efectual crawling-ul și durata acestuia.

Datele sunt trimise către server în timp real prin intermediul unui third party API numit Pusher, care preia asincron datele de pe server si le trimite la utilizator in timp real.

Partea de client procesează datele primite de la server asincron. La fiecare pas

aplicația trimite obiecte spre client, aceste obiecte sunt transformate pe partea de client cu ajutorul jQuery și JavaScript. Ele sunt afișate pe partea de client printr-un cloud de cuvinte generat live și updatat la fiecare pas.



4. Construcția Sistemului

Proiectul a fost dezvoltat pe platforma Ruby on Rails. Ruby este un limbaj de programare care nu a fost studiat in facultate. Am ales acest limbaj de programare deoarece momentan dezvolt și alte proiecte în acest limbaj. O altă cauză care m-a determinat să aleg rails a fost lipsa in momentul de față a unui crawler stabil în acest limbaj care să ofere și alte funcționalități decât crawlarea de bază, în afara de câteva librării care sunt destul de vechi și nu au fost updatate in ultima vreme.

Există o singură librărie care în acest moment este updatată, este vorba despre MetaInspector, dar aceasta nu este o librărie care să acopere web crawling-ul bine sau cel puțin nu mi-a satisfăcut mie toate cerințele. Din cauza aceasta am apelat la o librărie mai veche, numită Anemone dar care are dependințele la zi, cum ar fi fake-web folosit pentru requesturile către URL-uri, Nokogiri folosit pentru stocarea efectivă a paginilor și altele. Aceasta poate fi considerată o primă versiune de bază pentru următoarele modificări și îmbunătățiri care pot fi aduse proiectului. Pentru baza de date am folosit Mysql2 și pe partea de view este folosit jquey si javascript.

4.1 Crawler

Pentru construirea crawler-ului după cum am spus și mai sus am folosit o librărie numită Anemone. Această librărie a oferit mijloacele necesare pentru a face crawl pe un site random. Pe lângă Anemone am folosit și MetaInspector ca să reușesc să acopăr cazul in care URL-ul oferit de către utilizator avea un redirect inițial. Exemplu: www.infoiasi.ro va accesa link-ul www.infoiasi.ro/bin/Main .

Anemone își bazează crawler-ul pe modelul de bază al unui web crawler. Un prim pas pe care crawler-ul îl realizează este acela de a verifica fișierul robots.txt, dacă acesta există si de a reține ce url-uri trebuie sa evite, în caz că fișierul este prezent. După cum am precizat și anterior, un crawler dacă nu respectă fișierul robots.txt se transformă intr-un crawler invadativ.

Primele rezultate ale crawling-ului arătau in forma următoare:

http://www.****.ro/ Queue: 0

http://www.****.ro/customer/account/ Queue: 57

http://www.****.ro/customer/account/login/ Queue: 57

http://www.****.ro/wishlist/ Queue: 57

După cum se poate observa mai sus, la început se inițializează o coadă empty după care se adaugă link-uri în coadă care așteaptă să fie procesate. Căutarea este realizată cu un algoritm BFS accesând paginile prin intermediul URL-urilor anterioare cum este descris mai jos, si recursiv si noile pagini sunt accesate în același mod.

```
doc.search("//a[@href]").each do |a|
  u = a['href']
  next if u.nil? or u.empty?
  abs = to_absolute(u) rescue next
    @links << abs
end
```

*codul de sus este extras din librăria folosită anemone care a fost modificată ulterior de mine

Având in vedere că nu descărcăm fișierele de pe servere și nu le parsăm, o primă filtrare a fost realizată la nivel de url-uri, în care a trebuit sa eliminăm url-urile care conțineau in url, formate de tipul: flv swf png jpg gif asx zip rar tar 7z gz jar js css dtd xsd ico raw mp3 mp4 wav wmv ape aac ac3 wma aiff mpg mpeg avi mov ogg mkv mka asx asf mp2 m1v m3u f4v pdf doc xls ppt pps bin exe rss xml. În următoarele variante putem să eliminăm din listă fișierele de tipul doc și pdf, care pot fi parsate și extrase cuvintele cheie și adăugate la lista cu cuvinte cheie și procesate.

Primele probleme de crawling au apărut foarte repede, având in vedere că foarte multe pagini aveau link-uri care nu duceau la nici o pagină sau care întorceau timeout și era întreruptă conexiunea cu crawler-ul și în același timp si conexiunea cu serverul. Pentru a elimina această problemă o soluție a fost să introducem un alt filtru la url-uri, dar de data aceasta a fost un filtru pe partea de coduri de return la pagini. Orice pagină care nu a returnat

un cod între 200-399 când i s-a făcut fetch (pas inițial realizat de crawler-ul web pentru a lua paginile de pe web) va fi ignorată. Prin această soluție am scăpat de TCP/IP connection closed, de erorile de tipul bad URI și multe altele.

Următoarele probleme apărute au fost legate de redirect-urile către alte pagini. Aici au existat mai multe dificultăți deoarece crawler-ul trebuia să permită accesul pe paginile care erau în subdomeniul url-ului inițial, dar să nu permită urmărirea url-urilor din alte domenii, deoarece acest lucru ar duce la căutări imense.

O primă soluție a fost să mai fac o filtrare a url-urilor: Am creat liste cu url-urile care conțineau redirect-uri și în timp ce realizam crawling-ul verificam ca url-urile introduse să fie unice. Am implementat acest lucru, dar după ce am adăugat încă un crawler pentru redirecturi, a devenit evident că o să fie foarte ineficient luând în considerare cazurile în care există foarte multe redirect-uri pe un site → rezultă foarte multe crawlere care o sa ruleze în același timp, ceea ce a dus foarte rapid la o lipsă de memorie și încetarea rulării serverului.

O altă soluție a fost să adăugăm url-urile de redirect in set-uri diferite și să fie rulate după ce se termina crawler-ul început. Și această soluție creștea timpul foarte mult având in vedere că trebuia să stocăm foarte multe informații temporal și trebuiau înlocuite foarte des, așa că am decis să facem Fork la gem-ul Anemone, să facem o copie locală, ulterior adăugată pe GitHub ca si repository public (https://github.com/ValentinGeorge27/anemone) și după am modificat librăria cu dependințele necesare pentru a realiza crawling-ul cu cerințele noastre.

O primă adăugare în librărie a fost posibilitatea unui crawler de a adăuga la url-urile permise și subdomeniile. Acest lucru a fost realizat cu un regex care a utilizat librăria PublicSuffix și URI. Regexul este de forma:

@valid_domains = @urls.map{|u| [u.host, PublicSuffix.parse(URI.parse(URI.encode(u.to_s)).host).domain]}.flatten.compact.uniq

-encode a fost folosit deoarece erau multe url-uri care nu aveau forma necesara parsării

-host, domain → returneaza o host-ul urlului respectiv domeniul

-flatten → include toate array-urile rezultate în unul singur

-compact → concatenează toate array-urile reurnate

-uniq → păstrează doar domeniile unice

Acest array a fost folosit mai departe în cod și prin modificările necesare și dupa ce am adăugat optiunea de :crawl_subdomains => true , metodă care a fost adăugată în librărie ca să adauge la crawler modificările de mai sus am obținut rezultatele următoare:

http://www.piciorugras.ro/checkout/cart/ Queue: 55

http://www.piciorugras.ro/customer/account/login/ Queue: 54

http://www.piciorugras.ro/concursuri/ Queue: 53

După cum se poate observa sunt adăugate si blog-urile și restul de url-uri care aparțin aceluiași domeniu.

O altă problemă destul de îngrijorătoare în legătură cu acest crawler a fost lipsa memorie RAM, având în vedere că în următoarele adăugări de funcționalitate va crește timpul de procesare. O problemă a intervenit la coada folosită de crawler având în vedere că la site-urile foarte mari se adunau foarte multe link-uri în coadă de oridinul zecilor de mii și procesările la fiecare link durau destul de mult, a trebuit să introducem o altă modificare în librăria anemone care să modifice coada creată de crawler cu un SizedQueue care cere un parametru ceea ce reprezintă memoria RAM maximă cozii. Prin acest procedeu am reușit să menținem crawler-ul la un nivel optim și care nu mai cauzează opriri neașteptate din cauza memoriei.

Cu această ultimă problemă rezolvată, având un url dat pentru un site de shopping de haine pentru copii care conține inițial un număr de 1283 url-uri după procesările specificate mai sus, am obținut un număr de 775 link-uri valide un timp de 2.4122 min.

După acest pas am adaugat la crawler și un delay time, deoarece sunt multe site-uri care nu conțin în fișierul robots.txt opțiunea de a pune între link-uri delay time și din cauza aceasta multe dintre serverele care erau sub crawler cădeau la un moment dat sau erau inaccesibile din exterior. Deoarece mai bine așteptăm puțin mai mult decât să nu mai avem pe ce aplica crawler-ul.

4.2 Filtrarea Cuvintelor

Următorul pas în aplicație a fost adăugarea de funcție de filtrare a cuvintelor. Acest stagiu al aplicației presupune în principiu o filtrare selectivă a unui string ce reprezintă eliminarea cuvintelor care nu au ce căuta în string-ul final. Câteva seturi importante de cuvinte care trebuie eliminate reprezintă scripturile javascript, părtile de css introduse in paginile HTML și tag-urile HTML. Versiunea nouă de Rails, 4.2 a venit în ajutor în acest caz deoarece are implementate 4 metode pentru a scăpa de toate tag-urile html, javascript si css. Cele 4 metode sunt sanitize, sanitize_css, strip_tags și strip_links . Noi vom folosi doar primele 3 metode pentru a curăța fiecare fișier în parte, deoarece de link-uri avem nevoie.

După aceste transformări trebuie să eliminăm si cuvintele de legătură; am ales să eliminăm și cuvintele de legătură din limba engleză deoarece in multe pagini apar foarte multe citate în limba engleză. Momentan pe lângă limba engleză se elimină doar cuvintele de legătură, deoarece multe din site-uri nu adaugă limba care trebuie la metadatele paginilor, iar crearea un sistem de detectare a limbilor dintr-un fisier html nu era inclusă în proiectul inițial. Pe lângă cuvintele de legătură o altă filtrare a fost eliminarea pronumelor din documente. O ultimă filtrare a fost legată de titlul paginii, daca avem o pagină exemplu www.exemplu.ro cuvântul exemplu va fi eliminat din lista de cuvinte deoarece utilizatorul îsi dă seama dacă acel cuvânt este un cuvânt specific site-ului sau nu, nu are nevoie de sistemul nostru să îi mai spună încă o dată lucrul acesta.

4.3 Aplicare tf-idf

Pentru algoritmul tf-idf am folosit o librărie din Rails numită tf-idf-similarity, care ne-a permis să folosim algoritmii tf-idf într-un mod simplu și concis. Având în vedere că aplicația trebuie să returneze date în timp real la client, la fiecare url valid trebuie să creăm un document tf-idf, după care trebuie să creăm un model corespunzător pentru fiecare url în parte căte un model updatat cu valorile anterioare noi. Acest lucru s-a dovedit destul de costisitor din punct de vedere al timpului consumat.

În aplicarea algoritmului TF-IDF folosim frecvența augmentată. În acest sens documentele sunt normalizate după frecvența maximă a fiecărui termen în parte.

Algoritmul aplicat pentru TF este ca în cel descris jos, care reprezintă TF calculat în mod normal.

```
def term_frequency(document, term)
    tf = document.term_count(term)
    sqrt(tf)
end
```

O altă implementare posibilă este de a calcula prin dublă normalizare 0.5. Observăm că avem întâi calculată suma pentru toate documentele, după care se calculează frecvența pentru termeni. Această metodă previne înclinarea rezultatelor pentru un termen dacă există documente extrem de mari în care apare foarte des:

```
function tf(term, document) {
    maxTermCount = argmax(t => count(t,document));
    return 0.5 + 0.5*count(term,document)/maxTermCount;
}
```

Această implementare este disponibilă în librărie și este bazată pe modelul de calcul Lucene. La un pas al aplicației am implementat această funcție, dar pentru calculul complet al scorului Lucene aveam nevoie de date pe care nu le puteam lua de pe site-urile web decât cu prelucrări și crawling în prealabil.

Algoritmul aplicat pentru IDF calculează un logaritm pentru a afla câtă informație este dată de către un cuvânt în toate documentele existente.

```
def inverse_document_frequency(term)
    df = @model.document_count(term)
    1 + log(documents.size / (df + 1.0))
end
```

La prima rulare pentru site-ul anterior cu 700+ url-uri crawling-ul si procesarea tuturor datelor a durat in jur de 7h30 min. După această prima rulare am modificat putin modul de a realiza toate modelele și am încercat sa eliminăm toate url-urile care nu aveau cuvinte rămase după toate prelucrările, adăugând și o metodă prin care eliminăm url-urile

care fac doar anumite call-uri ajax și nu influențează codul din html. După toate aceste schimbări am ajuns la un crawling stabil care dureaza între 40-60 min pe laptopul meu, dar pe o unitate care are I7 si 8gb RAM, acest procedeu a fost realizat în aproximativ 20 min.

Acest gem pe lângă faptul că folosește tf-idf pentru calcularea scorurilor, mai are implementată scorul Lucene, cum a fost prezentat mai sus.

Flow-ul pentru implementarea tf-idf unui URL este următorul:

-crearea documentului tf-idf si adăugarea lui într-un set de documente.

```
doc = TfIdfSimilarity::Document.new(page_for_doc)
```

-calcularea modelului Tf-Idf pentru documentele prezente până la url-ul curent

```
model = TfIdfSimilarity::TfIdfModel.new(docs, :library => :narray)
```

-aflarea scorurilor fiecărui termen din document și trimiterea acestora către utilizator

-realizarea unei sume a termenilor pentru adăugarea în baza de date a primilor 20 de termeni semnificativi din lista cu termeni calculată pentru viitoarele dăți când utilizatorul va încerca să crawleze același site.

Pentru că avem matrici foarte mari si date foarte multe, am folosit în loc de matrici obișnuite, o librărie care include o matrice Narray.

Narray este un vector pe N dimensiuni. Suportă tipuri de elemente de tip Integer care ocupă 1/2/4-bytes, precizie simplă/dublă Reală/Complexă și obiecte Ruby. Această extensie încorporează calcule rapide și manipularea rapidă a vectorilor numerici foarte mari în limbajul ruby. Pe lângă această librărie, tf-idf-similarity mai recomandă si Library (GSL) si Nmatrix, dar dintre cele trei librării, Narray are cea mai bună performanță.

4.4 Trimitere date către server prin call-uri asyncrone

O primă variantă pe care am încercat să o implementăm a fost prin folosirea unui Live controller care a fost implementat în ultima versiune Rails. Live Controller-ul folosește SSE (Server Side Events) care au fost implementate in versiunea HTML5. SSE definește

următorul flow pentru un live controller:

-eveniment: Dacă este specificat, un eveniment cu numele lui va fi specificat în browser

-reîncercare: Timpul de reconexiune în milisecunde utilizat când se încearcă trimiterea unui eveniment

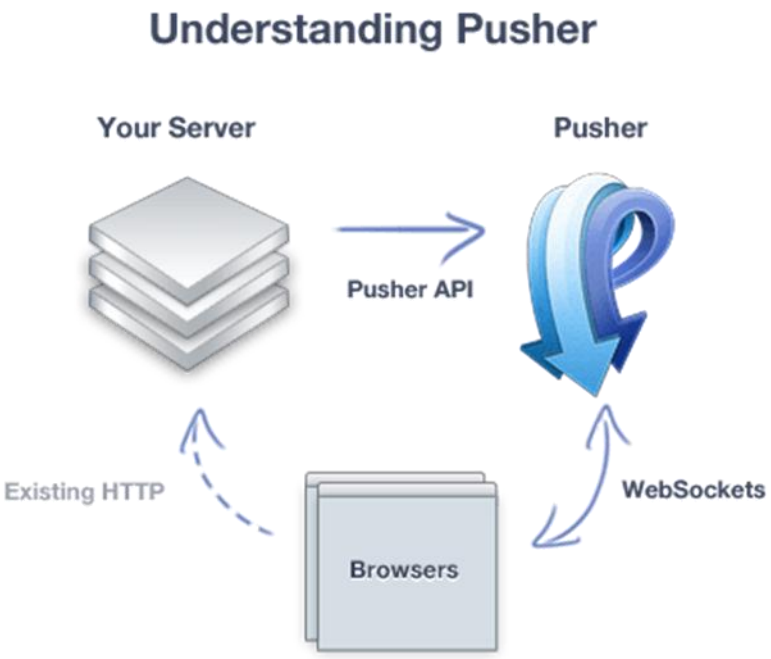
-id. În caz că se întrerupe conexiunea în timp ce un SSE se trimite la browser, atunci server-ul va primi un id al ultimului eveniment care va avea valoarea id

Având în vedere că sunt la început aceste metode in Rails, momentan ca să trimitem date la server avem nevoie de o metoda care să facă conexiunea cu browserul și să aibă setate headerele necesare pentru a trimite date prin streaming (text/event-stream) și un loop, de preferabil cât mai îndelungat având în vedere că putem avea call-uri care durează foarte mult timp.

Momentan, o problemă este că headerele se pot seta doar o singură dată, nu mai putem schimba headerele o dată ce un răspuns a fost trimis. Dacă vrem să trimitem date la server trebuie să le trimitem in metoda respectivă sau să salvăm temporar in fișiere/baza de date. Problema aceasta putea fi rezolvată ușor, dar între timp a mai apărut încă o problemă. Având în vedere că stream-ul este rulat la infinit și a trebuit să schimbăm serverele pe care rulăm aplicația pe un server nou denumit Puma, deoarece Puma este un server care suportă call-urile concurente. Pe lângă problema de server, a reapărut problema cu lipsa memorie și foarte multe thread-uri deschise în același timp, având în vedere că deja consumăm foarte multă memorie cu procesările tf-idf si cu crawler-ul.

Din aceste motive folosim o altă abordare în trimiterea datelor la browser. Este vorba despre un Third Party API denumit Pusher.

Pusher este o api pentru a integra real time ușor si sigur call-uri bi-direcționale prin WebSockets de la server către server sau telefoane mobile.



Pe partea de View datele sunt primite de la Pusher și sunt prelucrate la fiecare pas, pentru ca utilizatorul să știe în timp real care sunt primii 10 termeni dominanți în url-ul dat. Această afișare s-a executat cu ajutorul unor librării javascript care vor fi prezentate în continuare.

4.5 Afișarea datelor pe partea de client

După cum am precisat și mai sus datele vin asyncron de la server. Pe partea de view cel mai bun mod de a arăta rezultatele utilizatorului este printr-un cloud de cuvinte. Acest lucru se poate realiza printr-un update constant al acelui cloud.

Pentru a realiza acest cloud am creat o variabila globală care reprezintă o însumare totală pe partea de client a tuturor termenilor care vin de la server. Pentru termenii care se repetă vom însuma doar scorurile lor.

Cu ajutorul JavaScript si underscore, o librărie ce oferă un o modalitate de a aplica programarea funcțională pe obiectele JavaScript, am modificat tot ce era nevoie pentru ca în funcție de librăria pe care o folosim să afișăm conținutul necesar.

4.5.1 JQCloud

O primă încercare de a realiza aceste afișări pe partea de client au fost prin utilizarea librăriei JQCloud. Cu această librarie se contruia un cloud de cuvinte bazat doar pe HTML + CSS și era bazată pe jQuery.

Pentru a construi acest cloud a trebuit să modificăm formatul obiectelor primite de la server într-un vector de obiecte de forma {text,weight}. Aceste două elemente erau necesare pentru librărie pentru a calcula mărimea fontului unui cuvânt în comparație cu restul cuvintelor din vector, luând în considerare scorul cuvântului.

Pe lângă aceste două elemente, aveam ca și elemente opționale și html → cu care puteam să îi atribuim opțiuni html pe lângă ce oferă librăria default, link → putem adăuga unui cuvânt un link care va fi deschis la clic pe cuvânt și un titlu aferent și alte opțiuni.

Totodată puteam atribui și opțiuni pentru div-ul unde ajungeau cuvintele cum ar fi weight/height în caz că nu s-au atribuit aceste opțiuni în fișier sau în caz că dimensiunile la cloud depășeau dimensiunile inițiale ale div-ului. Un exemplu de apel la cloud este : $(“#exemplu”).JQCloud(words); unde exemplu reprezintă id-ul div-ului unde va fi atașat cloud-ul.

După cum s-a văzut și mai sus există doar o modalitate de a afișa datele și principalul meu obiectiv a fost să fac update la cuvintele din cloud live, nu la fiecare apel să fac refresh la div cu alte valori. Librăria avea o metoda de update, dar din cauza erorilor am renunțat la folosirea ei.

4.5.2 D3-cloud

D3-cloud este o librărie scrisă în JavaScript inspirată de la Wordle. Wordle este un tool care îți permite să creezi un cloud de cuvinte de la un set dat bazat pe scorurile precisate în set.

Pe acest principiu funcționează si D3-cloud. Creatorul librăriei oferă si un tool pentru librărie unde poti adăuga un link si îți întoarce un cloud de cuvinte. O funcționalitate extinsă este oferită de librăria javascript wordcount2.js. Aceasta a fost și a doua încercare de a afișa cuvintele care sunt primite de la server. Cu ajutorul underscore.js am transformat obiectele primite de la server pentru a putea fi folosite pentru noua librărie. Wordcount2 extinde WordCount și folosește canvas, un element HTML5 pentru a afișa imagini dinamice și de a crea grafuri în imagini.

WordCloud(elements, options); → cu ajutorul acestui apel inițiam un nou cloud având cuvinte de la server. La acest call la opțiuni se pot adăuga de la liste de cuvinte la call-uri după randări, dar și dimensiuni, timing, rotații, forme, dacă este interactiv sau nu și evenimente. Cu toate aceste opțiuni am renunțat la aceast tool deoarece avea și el multe

restricții pentru afișare și am decis să folosim librăria d3-cloud și să contruim flow-ul personal direct din librărie, fără a folosi nici un tool.

Pentru a procesa datele am folosit aceeleași metode de la underscore, după care am creat un obiect de tip d3.layout.cloud(), care ne permite să apelăm la o metodă de desenare în imagine. Pentru imagine am folosit tagul HTML svg. Scalable Vector Graphics sunt folosiți pentru a defini grafice pentru partea de Web. Pe lângă aceasta, mai se pot adăuga imagini, texte, forme geometrice. Mai jos avem o forma a unui cloud de cuvinte, in care sunt adaugate proprietatile de baza pentru realizarea cloudului.

```
layout = d3.layout.cloud()
            .timeInterval(2)
            .size([800, 600])
            .words(arr.map(function (d) {
                return {text: d.text, size: d.weight, url: d.url};
            }))
            .padding(5)
            .rotate(function () {
                return 0;
            })
            .text(function(d) { return d.text; }) // THE SOLUTION
            .fontSize(function (d) {
                return wordScale(d.size);
            })
            .on("end", draw);
    layout.start();
```

Functia “draw” va fi responsabila pentru adaugarea/modificarea si stergerea cuvintelor din cloud.

După ce am selectat forma svg adăugăm cuvintele care vin de la server in trei modalități-dacă sunt deja în formă ii schimbăm scorul, dacă nu sunt în formă adaugăm cuvântul cu scorul aferent și la sfârșit verificăm restul cuvintelor care erau în formă și nu au mai venit de la server in topul cuvintelor să le eliminăm, să arătăm utilizatorului doar cuvintele semnificative. Cuvintele încep cu un font minim 8 si urcă în timp ce le crește scorul.

După cum se observă și în imagine cuvintele au mărimi diferite și nu sunt intersectate deloc. Aceasta a fost altă problemă întâmpinată la update-ul cuvintelor de la server fară a face nici un fel de refresh. Pentru a evita aceste suprascrieri, la fiecare pas cuvintele își schimbă poziția și în spate se calculează o poziție pentru fiecare element din listă să fie cât mai bine poziționat. Pe langa acest lucru la fiecare pas sunt adaugate si url-urile la top 10 cuvinte care au cel mai mare scor in acel moment.

5. Rezultatele evaluării

Sistemul de crawling a fost programat să elimine toate informațiile inutile primite de la site-ul web. Sistem primește ca input un URL specific de la utilizator. Crawler-ul este contruit în așa fel încât să primească mai multe opțiuni:

-max_page_queue_size → reprezintă memoria maxim acordată listei de stocare a url-urilor,

-obey_robots_txt → una din cele mai importante optiuni, care nu ar trebui să lipsească în nici un crawler

-depth_limit → cât de adânc sa crawleze site-ul, acum este setată la 5 pentru testări,

-skip_query_strings → când este setat true crawler-ul face skip la url-urile care contin query-uri la server

-read_timeout → setăm timpul care ar trebui să asteptăm un răspuns la un url

-crawl_subdomains → setăm dacă să adăugăm si subdomeniile din domeniul in linkuri

-delay → setăm această opțiune dacă dorim să avem o pauză între cereri

Aceste opțiuni ne ajută să primim date cât mai concludente si mai rapide de la crawler pentru a putea fi modificate în continuare de către tf-idf și trimise la utilizator în timp real.

Dacă în primă fază crawler-ul nostru dura aproximativ 7h+ pentru un crawling la un site care conține aproximativ 1200 de link-uri dintre care 700 link-uri valide, adică care au returnat un cod de success între 200-399, după aplicarea tuturor schimbărilor peste url-uri și conținutului am ajuns la un timp de sub 1h. După introducerea pe partea de web a afișării și introducerea unui delay de 2 secunde pe partea de server pentru protejarea serverelor, am ajuns la un timp final de 70-90 min .

5.1 Direcții ulterioare

Acest proiect poate avea foarte multe continuări, fiecare modul în partea poate avea o parte aprofundată și după cum am mai precizat s-ar putea contrui un API pentru o librărie nouă și up to date pentru a putea fi folosită în rails.

O primă evoluare, poate fi considerată pe partea de baze de date, având în vedere că momentan în baza de date ținem doar date generale asupra site-urilor pe care facem crawl.

La partea de crawling putem să adăugam un nou environment de lucru, mai exact am putea adăuga proiectul pe Amazon EC2, atribuind proiectului 2 sau mai multe mașini virtuale. În caz că una din mașini rămâne fără memorie o alta care este în stagiul de ready preia procesul și continuă crawling. În acest fel, scăpăm de mai multe probleme întâlnite pe partea de server crawling.

La partea de tf-idf și sortări, se poate implementa partea de Lucene, dacă vom salva în baza de date mai mult decât date generale despre site, atunci vom avea și date de test care pot fi folosite pentru Lucene queries.

Pe partea de eliminare a cuvintelor din text. O soluție care să ne dea rezultate mai precise ar fi eliminarea unor cuvinte bazate pe limba prezentă în text, Goole Cloud Platform este un tool care ar putea detecta ce limbi ajung de la url la server și să ne ofere și cuvintele de legătură care trebuie eliminate.

Pe partea de client, o primă modificare ar fi adăugarea de url-uri pe cuvinte pentru tot istoricul parcurgerilor si formarea unui top al url-urilor, dar acest lucru este posibil doar dacă baza de date este updatată cu toate link-urile pentru a calcula scoruri diferite pentru anumite seturi de date pentru cuvinte specifice.

6. Concluzii

În realizarea acestui proiect am reușit să îmi ating țintele propuse, mai exact să ofer unui client date referitoare la un site random oferit de ei. Prin realizarea aceastui proiect am acumulat cunoștințe puternice legate de crawling, de tipuri de site-uri, conexiuni realizate și multiple probleme ce pot apărea în timpul conexiunilor, aplicarea de tf-idf asupra cuvintelor, javascript mai precis despre d3 și despre afișarea datelor in grafuri, imagini.

7. Bibliografie

[1] http://lucidworks.com/blog/crawling-in-open-source-part-1/

[2] http://ijrise.org/asset/archive/15SANKALP36.pdf

[3] https://en.wikipedia.org/wiki/Web_mining

[4] https://en.wikipedia.org/wiki/Robots_exclusion_standard

[5] http://www.scaleunlimited.com/about/web-mining/

[6] http://dmr.cs.umn.edu/Papers/P2004_4.pdf

[7] http://www.grepsr.com/blog/importance-of-web-crawling-in-the-age-of-big-data/

[8] http://aisel.aisnet.org/cgi/viewcontent.cgi?article=3785&context=cais

[9] https://en.wikipedia.org/wiki/Tf%E2%80%93idf

[10] http://www.tfidf.com/

[11] https://pusher.com/docs

[12] http://api.rubyonrails.org/classes/ActionController/Live/SSE.html

[13] http://lucene.apache.org/core/4_3_0/core/org/apache/lucene/search/similarities/TFIDFSimilarity.html