Rangirajući pretraživač dokumenata - Infinity

Marko Budiselić

30.11.2015.

1 Zadatak

U sklopu zataka potrebno je implementirati rangirajući pretraživač dokumenata (engl. *ranking search engine*) prikladan za uporabu na news portalima sa sadržajem na engleskom jeziku.

Pretraživač treba imati sljedeće odlike:

- mogućnost rada s velikim brojem dokumenata ($>$ 1 milijun)
- rangiranje rezultata pretrage
- nisko vrijeme obrade upita (sposobnost rada pod velikim opterećenjem)
- optimalno korištenje računalnih resursa
- jednostavnost implementacije i korištenja

Sučelje sustava:

- pretraživač se inicijalizira dokumentima pohranjenim u proizvoljnom obliku
- u pretraživač se naknadno (tijekom rada) mogu dodavati dokumenti
- kao upit prihvaća slijed riječi
- kao odgovor vraća indekse dokumenata

Testni skup dokumenata:

- `http://qwone.com/~jason/20Newsgroups`

Testno računalo:

- RAM: 7880944 kB; 1600MHz
- CPU: Intel(R) Core(TM) i3-4010U CPU @ 1.70GHz

2 Pretprocesiranje

Bitan segment analize teksa je predprocesiranje. Predprocesiranje počinje s podjelom dokumenta na riječi (engl. *tokenization*). Implementiran je jednostavni tokenizator koji prvo zamijenjuje posebna znakove s prazninama, nakon toga odbacuje riječi duže od 25 znakova (25 je nastao subjektivnom procjenom) i na kraju se radi svođenje riječi na jednostavniji oblik (engl. *stemming*). Korišten je Snowball (Porter2) stemmer zato što je manje agresivan od Lancaster stemmer-a, a brži od Porter stemmer-a. Uvođenjem stemmer-a naraslo je vrijeme izvođenja pretprocesiranja i to za $\sim$4 puta (koristi se stemming Python modul). Trebalo bi koristiti neku brzu C/C++ implementaciju stemming postupka. Nije se koristio lematizator zato što je bitno očuvati riječi što je moguće više onakve kakve jesu. Primjerice, ne bi bilo dobro zamijeniti *is* s *be*, jer je korisnik možda baš htio dokumente gdje se pojavljuje *is*. Lematizator bi *is* riječ zamijenio s riječi *be* i tu bi se izbugila informacija (opet subjektivna procjena). Kada bi postojala potreba za lematizatorom tada bi se uz Python koristio spacy lematizator i to zato što je brži u odnosu na, primjerice, nltk lematizatora.

3 Algoritmi

3.1 Bag of words

Dvije su osnovne strukture podatatak unutar algoritma. *Bag of words* unutar svakog dokumenta broji koliko se puta pojedina riječ pojavila unutar tog dokumenta. *Bag of documents* za svaku riječ broji koliko se puta pojavila u pojedinom dokumentu. Stvari su slične, ali omogučuju da se algoritam izvede u složenosti *(broj riječi upita * broj dokumenata u kojima se pojavljuje riječ)*. Pristigli upit najprije se pretprocesira, potom se za svaku riječ i za svaki dokument unutar kojeg se ta riječ pojavljuje računa normalizirana težina. Za svaku riječ gleda se koliko puta se ona pojavljuje unutar dokumenta i onda se ta vrijednost normalizira s veličinom dokumenta. Na taj način se postiže da kraći dokumenti u kojima broj pojavljivanje pojedine riječi čini veći postotak unutar dokumenta imaju snažniji utjecaj nego dokumenti u kojima se rijeđe pojavljuje dotična riječ. Na kraju se sortiraju dobiveni dokumenti prema izračunatim težinama. Dobre strane algoritma su brzina, jednostavnost, normalizacija, a mana je što se ne uzima nikakva globalna informacija o zastupljenosti riječi unutar cijelog skupa dokumenata. Uvijet koji bi trebao biti zadovoljen je da ne postoji dokument unutar kojeg se nalaze sve riječi iz korpusa. U suprotnom će algoritam sporije raditi od očekivanog, ali pretpostavka je da se to neće dogoditi. Na danom skupu dokumenata, vrijeme izvođenja algoritma je rada veličine 1ms i ne bi trebalo postati drastično veće na skupu od, primjerice, reda veličine 1M dokumenata.

3.2 Vector space

Temelj algoritma vektorskog prostora (engl. *vector space*) su izrazi:

$$w_{ij} = TF(t_i, d_j) \cdot IDF(k_i, D) \tag{1}$$

$$TF(t_i, d_j) = \frac{freq((t_i, d_j)}{max(freq(t, d_j)|t \in D)} \tag{2}$$

$$IDF(k_i, D) = log\Big(\frac{|D|}{|\{d \in D|t_i \in D_j\}|}\Big) \tag{3}$$

Inicijalna implementacija vector space modela bila je iterativna. Dakle za svaki dokument pretprocesiranjem se odredio vektor $TF \cdot IDF$, a prilikom rangiranja, za svaki dokument se izračunala mjera udaljenosti. Takva implementacija, za dani skup dokumenata dala je rezultat za red veličine 10s, što je neprihvatljivo. Trenutna implementacija algoritma koristi množenje sparse matrice i vektora (po svakom retku) i takva implementacija daje rezultat u vremenu od reda valičine 10ms, što je puno prihvatljivije. Prilikom izgradnje matrica koristio se *lil* tip matrice. Kada je postupak izgradnje završio *lil* matrice su prebačene u *csr* tip kako bi se što brže provelo množenje. Kao mjere udaljenosti razmatrane su euklidska i kosinusna udaljenost. Kosinusna udaljenost dala je vrijeme izvođenja upita ∼25ms, dok je euklidska udaljenost dala vrijeme izvođenja upita ∼40ms. Stoga je odabrana euklidska udaljenost. Prilikom dodavanja jednog novog dokumenta ne provodi se cijeli postupak pretprocesiranja od početka. Ideja je da se samo ažurira matrica $TF \cdot IDF$. Inicialno je to napravljeno tako da se CSR matrica pretvorila u DOK, nakot toga joj se povečala veličina i onda se pretvoritla u LIL matricu kako bi se efikasno dodali novi elementi. Na kraju se LIL matrica pretvorila nazad u CSR matricu. Takva implementacija, za dani testni skup podataka, trajala je ∼6s.

3.3 Binary independence

Temelj algoritma binarne nezavisnoti je formuala:

$$\sum_{t \in q} log \frac{p(D_t|q, r)}{p(D_t|q, \bar{r})} = \sum_{t \in q} w_t \tag{4}$$

gdje se $p(D_t|q, r)$ procjenjuje na 0.5, a $p(D_t|q, \bar{r})$ na n_t/N_d, gdje je pak n_t broj dokumenata u kojima se nalazi riječ t, a N_d ukupan broj dokumenata.

Implementacija složenost algoritma ovisi o broju dokumenata nad kojima se vrši rangiranje. Konkretno, za dani upit potrebno je obići sve dokumente kako bi se dobila procijenjena vjerojatnosti $p(D_t|q, \bar{r})$. Danu procjenu u teoriji je moguče predprocesirati, no budući da bi predprocesiranih podataka bilo $(broj\_dokumenata \cdot broj\_tokena)$, to je u praksi nemoguče napraviti. Problem bi se moglo riješiti tako da se skup dokumenata podijeli na dovoljno male podskupine unutar kojih bi se onda radilo rangiranje.

Problem kod algoritma je i da ako je ulaz jedan token, procijenjene težine za svaki dokument su iste i onda je nemoguče napraviti kvalitetno rangiranje. To bi se moglo riješiti tako da se na težinu za pojedini token doda normalizirani broj njegovog pojavljivanja unutar dokumenta.

4 Implemntacija

Svaki algoritam tipa je *IRAlgorithm*. Taj tip posjeduje četiri metode. *config* putem koje se prima nekakva konfiguracija algoritma, ako takva za dani algoritam postoji. *preprocess_ all* kroz koju se radi predprocesiranje svih dostupnih dokumenata. *preprocess_ one* putem koje se radi predprocesiranje samo jednog novog dokumenta. I, *run* koja vrača konkretne rezultate. Kako bi se izbjeglo pokretanje predprocesiranja svaki put kada se treba pokrenuti neki algoritam uveden je razred AlgorithmBox koji posjeduje instance svih dostupnih algoritama u varijabli *available_ algorithms*. U varijabli *prepared_ algorithms* nalaze se samo one instance algoritama za koje je već proveden proces predprocesiranja. Varijabla *prepared_ algorithms* se prilikom svakog postavljanja novih dokumenata kroz *setter files* postavi na prazni riječnik. Na taj način se postiže da se kod idučeg dohvačanja nekog algoritma ponovno forsira proces predprocesiranja. Ukoliko se dodao samo jedan dokument onda se samo pozove metoda *preprocess_ one* nad svim algoritmima koji su unutar riječnika *prepared_ algorithms*. Mana ovog pristupa je što svaki algoritam za sebe drži kopiju svih dokumenata i ponavljanje predprocesiranja za svaki algoritam posebno. Prednost je, svojstvo da svaki algoritam ima izolirani skup podataka s kojim može činiti što je već potrebno kako bi algoritam valjano radio. Odlučio sam se za taj pristup zato što je memorija manji problem od nekakve jednostavnosti korištenja i važnosti da je svaki algoritam za sebe izoliran. Jedna od mana trenutne implementacija je i što se dokumenti pamte unutar riječnika. Optimalnije bi bilo pamtiti dokumente unutar liste, te imati potporne strukture podataka kao što su riječnici u kojima bi se pamtilo primjerice na kojoj poziciji u listi je određeni dokument.

5 Upute za pokretanje

Izvorni kod: `https://github.com/gitbuda/infinity.git`

```
$ git clone https://github.com/gitbuda/infinity.git
$ cd infinity
$ source setup.py
$ python console.py
```

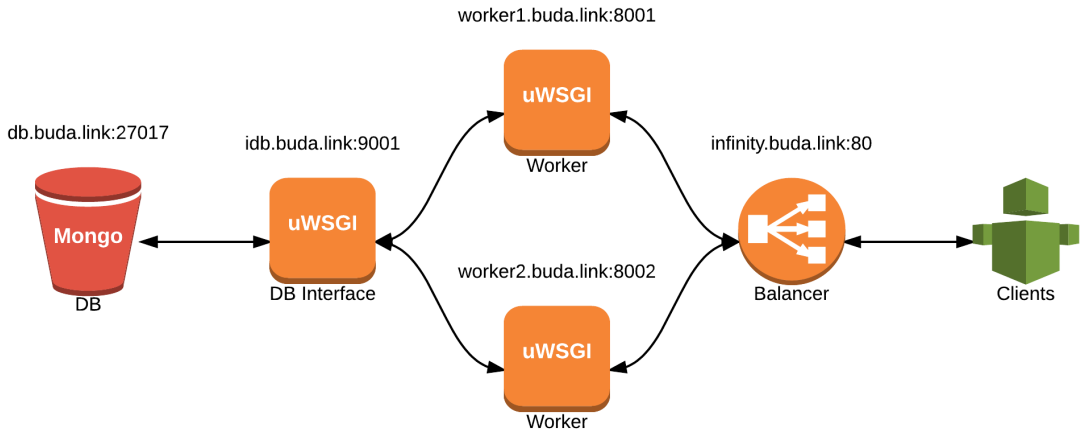
Web sučelje: `https://infinity.buda.link`

6 Produkcijska okolina

Svi upiti klijenata dolaze na Nginx load balancer koji ima izrazito veliku propusnost. Nakon toga load banancer prosljedjuje upit na worker instance koje svaka za sebe imaju cijeli dataset i znaju vratiti odgovarajuci rezultat. Dataset worker instance preuzimaju od db interface instance, a ne direktno iz baze. Trenutno je u deploymentu samo jedna instanca sucelja prema bazi, ali u produkciji tu moze biti opet load balancer i vise instanci

interface-a prema bazi podataka. Baza podataka je MongoDB, takodjer samo jedna instanca, no u praksi tu moze doci mongo replica set ili mongo shard cluster.

Kao sto se vidi na slici 1, sve instance imaju simbolicka imena (FQDN). To je takodjer izrazito bitno jer se time postize transparentnost pristupa i migracijska transparentnost. U konkretnoj implementaciji sva imena su definirana u /etc/hosts file-u na deploy stroju, no u produkciji ce imene biti definirana na redundantnim DNS serverima.



Slika 1: Produkcijska okolina

Prilikom testiranje opterečenja instanci radnika, u obzir su uzeta dva najpopularnija WSGI servera gunicorn i uWSGI, testiranje je provedeno s bag of words algoritmom, a rezultati su vidljivi u tablici ispod. Najbolje rezultate daje konfiguracija u kojoj je uWSGI iza Nginx-a u socket načinu rada.

konfiguracija	req/s
gunicorn	1245,64
uWSGI	1767,23
Nginx + uWSGI	1767,23

7 Web sučelje

TODO: Angular + materializecss

TODO: nekakva slika sučelja

8 PS

infinity > gugol