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:

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

2 Pretprocesiranje

Bitan segment analize teksta je pretprocesiranje. Pretprocesiranje počinje s podjelom dokumenta na riječi (engl. *tokenization*). Implementiran je jednostavni tokenizator koji prvo zamijenjuje posebne 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*). Svođenje riječi na jednostavniji oblik je potrebno, primjerice, zato da bi se riječ *testing* svela na *test* (ako upit sadrži riječ *test*, dokument u kojem se nalazi riječ *testing* je poprilično bitan dokument). 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 ∼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 izgubila informacija (opet subjektivna procjena). Kada bi postojala potreba za lematizatorom tada bi se uz Python koristio lematizator Spacy i to zato što je brži u odnosu na, primjerice, lematizator NLTK.

3 Algoritmi

3.1 Bag of words

Dvije su osnovne strukture podataka 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 omogućuju 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 pojavljivanja 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 raditi sporije od očekivanog, ali pretpostavka je da se to neće dogoditi. Na danom skupu dokumenata, vrijeme izvođenja algoritma je reda 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 $TF \cdot IDF$. Inicijalno je to napravljeno tako da se CSR matrica pretvorila u DOK, nakon toga joj se povećala veličina i onda se pretvoritla u LIL matricu kako bi se efikasno dodali novi elementi. Na kraju se LIL matrica pretvorila natrag u CSR matricu. Takva implementacija, za dani testni skup podataka, trajala je ∼6s. U trenutnoj implementaciji prilikom dodavanja novog dokumenta na TF i IDF dodaje se sa *scipy.sparse.vstack scipy.sparse.hstack* primjereni broj redaka i stupaca, te se nakon toga matrice konvertiraju u LIL tip kako bi se popunile s odgovarajućim vrijednostima. Takva implementacija je spustila vrijeme dodavanja jednog dokumenta s ∼6s na ∼1s.

3.3 Binary independence

Temelj algoritma binarne nezavisnosti 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.

Implementacijska složenost algoritma ovisi o broju dokumenata nad kojima se vrši rangiranje. Konkretno, za dani upit potrebno je obići sve dokumente kako bi se dobila procjena vjerojatnosti $p(D_t|q, \bar{r})$. Danu procjenu u teoriji je moguće pretprocesirati, no budući da bi pretprocesiranih podataka bilo ($broj\_dokumenata \cdot broj\_tokena$), to je u praksi nemoguće napraviti. Problem bi se mogao riješiti tako da se skup dokumenata podijeli

na dovoljno male podskupine unutar kojih bi se onda radilo rangiranje. Ekvivalentan postupak mogao bi se provesti za svaki od promatranih algoritama.

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

3.4 Vrijeme izvođenja

Pravilno mjerenje vremena izvođenja ovakvih algoritama nije jednostavan zadatak. Ovdje je provedeno mjerenje vremena u kojem se dodaje jedan dokument od 10 nasumično odabranih riječi i mjerenje u kojem se rangiraju dokumenti za upit koji ima 3 riječi. Kao rezultat prikazana su približna vremena na testnom računalu.

Algoritam	**Dodavanje dokumenta (10 rijči)**	**Upit s 3 rijči**
bag of words	∼1ms	∼1ms
vector space	∼1s	∼25ms
binary independence	∼1ms	∼30ms

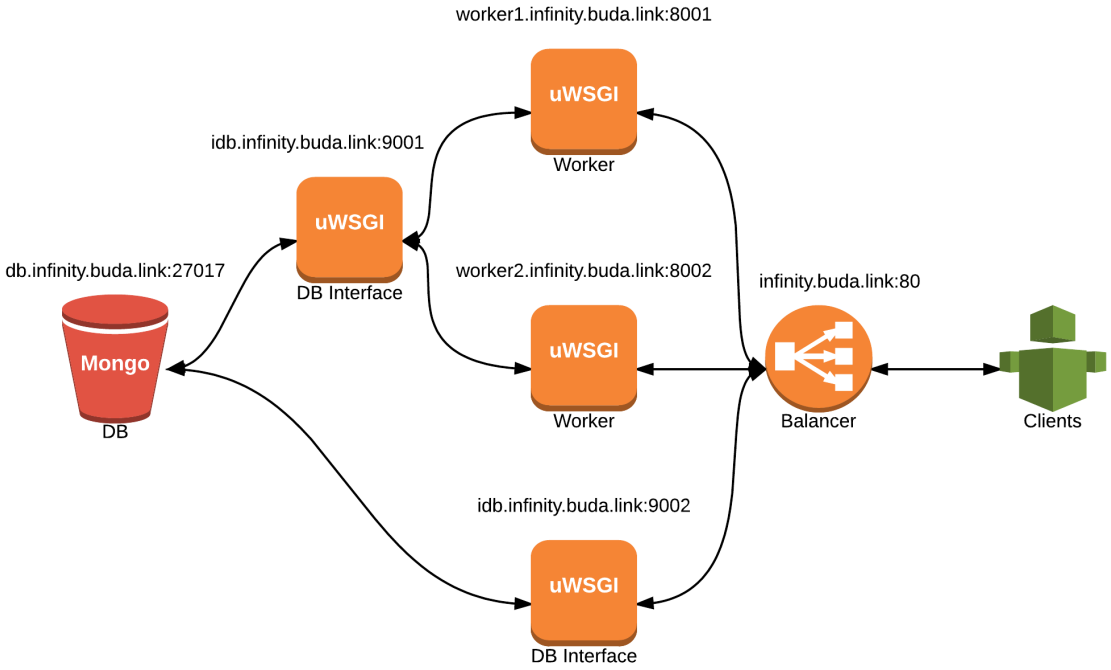
4 Implementacijski detalji

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 pretprocesiranje svih dostupnih dokumenata. *preprocess_ one* putem koje se radi pretprocesiranje samo jednog novog dokumenta. I, *run* koja vraća konkretne rezultate. Kako bi se izbjeglo pokretanje pretprocesiranja 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 pretprocesiranja. Varijabla *prepared_ algorithms* se prilikom svakog postavljanja novih dokumenata kroz *setter files* postavi na prazni riječnik. Na taj način se postiže da se kod idućeg dohvaćanja nekog algoritma ponovno forsira proces pretprocesiranja. 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 pretprocesiranja 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 Produkcijska okolina

Kao što se vidi na slici 1, svi upiti klijenata dolaze na Nginx load balancer koji ima izrazito veliku propusnost. Nakon toga load banancer prosljeđuje upit na worker instance koje, svaka za sebe, imaju cijeli dataset i znaju vratiti odgovarajući rezultat. Dataset worker instance preuzimaju od DB interface instance, a ne direktno iz baze. Trenutno je u deployment-u samo jedna instanca sučelja prema bazi, ali u produkciji tu može biti opet load balancer i više instanci sučelja prema bazi podataka. Baza podataka je MongoDB, također samo jedna instanca, no u praksi tu može doći Mongo replica set ili Mongo shard cluster.

Sve instance imaju simbolička imena (FQDN). To je također izrazito bitno jer se time postiže transparentnost pristupa i migracijska transparentnost. U konkretnoj implementaciji sva imena su definirana u /etc/hosts datoteci na deploy stroju (Digital ocean instanci), no u realnoj produkciji će DNS imena biti definirana na redundantnim DNS serverima.



Slika 1: Produkcijska okolina

Prilikom testiranja opterećenja 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	1915,63

Sve komponente sustava mogu biti pokrenute untar Docker kontejnera. Prednosti Docker kontejnera su: automatizirani proces izgradnje kontejnera i lakši postupak pokretanja procesa koji je untar Docker kontejnera. Na taj način poslužiteljsko računalo nije direktno ovisno o procesu kojeg pokreće. Docker kontejner, se u apstraktnom smislu, može shvatiti kao omotač oko procesa koji je unutar njega pokrenut. Cijeli sustav je robusniji i lakši za održavanje zbog toga što su procesi pokrenuti unutar Docker kontejnera.

Web sučelje implementirano je uz pomoć `https://angularjs.org` (MVC web radnog okvira) i `http://materializecss.com`. Dakle, napravljena je klijentska MVC aplikacija koja uzima podatke u JSON formatu sa servera. Postupak prikaza HTML-a i CSS-a ostavljen je na klijentskoj strani. Nginx može veoma brzo posluživati statične datoteke i ne postoji opterećenje generiranja cijelokupnog HTML-a na serverskoj strani.

Cjelokupni sustav postavljen je na Digital ocean poslužitelju. Prednosti Digital oceana nad ostalim VPS pružateljima su: SSD diskovne jedinice, optimizirana KVM virtualizacija, povoljna cijena, podatkovni centri u Americi, Aziji i Europi, te jako intuitivna kontrolna ploča (engl. management panel).

6 Upute za pokretanje

Kako bi se instaliralo sve što je potrebno za pokretanje konzolne aplikacije potrebno je na računalu imati python, pip i virtualenv, tj. sve za rad s programskim jezikom python3. Kroz konzolnu aplikaciju moguće je pokretati sva tri algoritma.

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

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

Literatura

[1] FER, Text Analysis and Information Retrieval (TAR), 2013/2014, TAR-03-IR.pdf

[2] Tian Xia, Yanmei Chai; An Improvement to TF-IDF: Term Distribution based Term Weight Algorithm; 2011 ACADEMY PUBLISHERE

[3] Mingyong Liu, Jiangang Yang, An improvement of TFIDF weighting in text categorization; 2012 International Conference on Computer Technology and Science, Singapore

[4] Za sve ostale implemtacijske detalje: www