

Pràctica A:
Detecció de similitud de documents amb
hashing

Noms: Yeray Bosoms, Guillem Gili i Ferran Noguera

Data: 15/11/2016

Curs: Tardor 2016

Descripció de l'algorisme

Hem implementat dos algorismes, el bàsic de Jaccard i funcions de hash generades mitjançant nombres aleatoris i aplicant la XOR.

Primerament separem els documents donats, que es llegeixen en forma de k-shingles de strings (frases). El k-shingle serà la nostra unitat d'informació. Un k-shingle consisteix en una agrupació de strings consecutius.

En el nostre algoritme llegeix el document k cops i llegeix cada cop 1/k k-shingles del document. Abans de guardar-los aplica la funció de hash que els converteix enters, ja que un enter ocupa molt menys a memòria. Aquesta funció ve donada per defecte de C++.

Seguidament calculem la similitud de Jaccard amb els valors obtinguts dels k-shingles de cada document que volem comparar. La funció de Jaccard consisteix en la divisió del mòdul de l'intersecció d'aquests dos documents i el mòdul de l'unió d'ambdós.

Després, calculem la similitud mitjançant Minhash i LSH. Minhash ens converteix dos sets a signatures simplificades que els identifiquen fent servir t funcions de hash. LSH o Local-Sensitive Hashing, ens permet classificar elements semblants en el mateix slot de diferents taules.

Estimem que el resultat de Jaccard serà més precís que l'obtingut mitjançant Minhash-LSH. Això serà perquè Minhash-LSH ens donarà més falsos positius que Jaccard, Jaccard només ens donarà els falsos positius que ens doni la funció de hash que fem servir com a base, mentre que Minhash-LSH tindrà més falsos positius si escollim una taula més petita que el domini de Minhash-LSH.

Canvis que hem realitzat durant la pràctica:

- Inicialment havíem considerat que les signatures havíem de ser del mateix tamany com que havien de representar el mateix nombre d'elements. Per a això havíem executat un algoritme que extreia elements dels sets més grans fins que tots els sets tinguessin el mateix tamany.

Això ha demostrat ser una mala implemmentació, ja que a més d'extreure important teníem molt poc rang per a comparar. A la nostra versió final feiem una taula de booleans per cada document, de tamany (`num_funcions_hash x 0x0000FFFF`).

- Inicialment feiem una lectura aleatòria, que era problemàtica en documents petits, i molt poc eficient respecte a informació. Tampoc transformava els valors llegits de forma immediata a enters, cosa que ens va donar problemes de memòria i ens evitava la execució del programa.

- Vam experimentar fent servir dues formes de generar funcions de hash:

1. Generació de valors mitjançant una xor amb valors de enters random

2. Generació de valors mitjançant una xor amb valors de enters random i rota els valors (posa els 8 bits finals al principi)

Experimentalment vam decidir la primera implementació ja que ens donava una millor variabilitat (tot i que la majoria de webs comentaven que la 2a era millor).

- Vam plantejar una lectura més ràpida que consistia en anar llegint el document per k strings i cada vegada eliminant el primer i afegint-ne el següent, per tant el cost algorítmic de llegir un document de n strings en k-shingles seria de fita de n i el cost seria n. Això milloraria el cost respecte a $k*n$, però al no tenir un coll d'ampolla a la lectura vam desestimar-ne la importància.

Cost asimptòtic del codi:

Consideracions generals:

- d és el número de documents
- t és el número de taules de hash a generar
- k és el tamany de k-shingle
- n és el número de elements que té un set(n_1, n_2, \dots per múltiples sets)
- ranghash és el número de slots a les taules de hash(és el mateix en totes)
- $\text{randomint}()$ és constant

Part 1:

Iterem sobre tots els documents i afegim al seu set corresponent els k-shingles necessaris, per tant tardarem $d \cdot (\text{suma de crida a funcions})$.

```
while(cin >> s1){
    ///PARTE 1
    // Creem un nou set on tindrem la signatura del fitxer s1
    set<unsigned int> nouset;
    //posem la signatura de s1(els valors de string hashejats)
    readFile(&s1[0], &nouset);
    //quan tenim el set l'afegim
    documents.push_back(nouset);
    //readfile actualitza el valor de MINSIGNATURA
    nomdocs.push_back(s1);
}
```

$O(d)$

Funcions a destacar:

Readfile:

Llegeix el document, aplica la funció de hash a un k-shingle n/k vegades. Ho fa k vegades per a tenir la representació de tot el document.

Així doncs el nostre temps serà $k \cdot n = n$

```
void readFile(char *target, set<unsigned int> *mochila){
    ifstream file;
    file.open(target);
    if (!file.is_open()) return;
    string word;
    string var;
    int max = 0;
    bool top = false;
    int pick;
    int choiced;
    while (!file.eof()){
        pick = 0;
        cout << "entra al primer bucle" << endl;
        choiced = randomint(); //gillrandom(0, random);
        cout << "choiced" << endl;
        //cout << choiced << endl;
        for(int i = 0; i < choiced and (top==true); i++){
            cout << "entra al for del primer bucle" << endl;
            file >> var;
            if (file.eof()) max++;
            else{
                file.clear();
                file.seekg(0, ios::beg);
                top = true;
            }
        }
        word = var;
        while(pick < k and !top){
            file >> var;
            if (pick < 0) word += " ";
            word += var;
            pick++;
            if (file.eof()) max++;
            else{
                cout << "hey, oh lets go" << endl;
                file.clear();
                file.seekg(0, ios::beg);
                top = true;
            }
        }
        if (pick == k) mochila->insert(myhash(word)); //cout << word << "    per el 18" << endl; //guarda var;
        cout << "picks" << mochila->size() << endl;
    }
    //float conversion = max*0.05;
    //int i_numdenhash = int(conversion + 0.5);
    //cout << "Top" << i_numdenhash << endl;
    for(int i = 1; i < i_numdenhash; i++){
        cout << "on while inside" << endl;
        file.clear();
        file.seekg(0, ios::beg);
        for(int j = 0; j < i; j++){
            file >> var;
        }
        choiced = randomint(); //gillrandom(0, max-5);
        cout << "choiced" << endl;
        cout << "initial" << set << endl;
        for(int i = 0; i < choiced; i++) file >> var;
        while(!file.eof()){
            word = var;
            pick = 0;
            while(pick < k){
                file >> var;
                if (pick < 0) word += " ";
                word += var;
                pick++;
                if (file.eof() and pick < k) pick += k;
                cout << "lipad" << endl; //guarda word;
            }
            if (pick == k) mochila->insert(myhash(word));
            cout << "lipad" << endl; //guarda word;
            //cout << mochila->size() << endl;
        }
    }
    if (MINSIGNATURA == -1 or MINSIGNATURA > mochila->size()) MINSIGNATURA = mochila->size();
}
```

$O(n)$

Push_back:


Es una funció de vectors que reserva memòria per a fer-los créixer i inserta un nou element. Aquesta funció té un cost de n on n és la quantitat de elements, però per la seva implementació s'amortigua (la part costosa és la reserva de memòria, que només és crida quan és estrictament necessari i quan es fa és sobrepassa la reserva necessària en funció del tamany actual del vector, per no haver de fer més reserves en els següents push_back). Així doncs el seu cost real acaba seguint constant, la seva contribució a l'algoritme en temps és d .

Així doncs el cost globalment a la part 1 es fita de $(d * n)$

Part 2:

Aquí calculem la similitud de Jaccard de cadascun dels sets obtingut dels documents, així que iterem sobre ells, fita de d .

```
//PARTE 2
//Fem la comparació Jaccard ara perquè perderem precisió fent-la més tard
for(int i=1;i<NUM_DOCS;i++){
    jacobianas[i-1]=Jsim(&documents[0],&documents[i]);
}
```



Funcions a destacar:

Jsim:

Crida a dues funcions que calculen la funció i la intersecció, i en divideix els valors, així que el valor que afegeix és constant, les funcions a les que crida si que aporten un cost temporal.

Unió-> Recorrem un cop cada set així que és fita de n_1+n_2

```
//Pre: k-shingles de dos docs a A i els altres a B
//Post: Retorna el modul de l'unio d'aquests sets
float opUnion(set<unsigned int> *a, set<unsigned int> *b) {
    set<unsigned int> unionSet;
    set<unsigned int> ::iterator it;

    for (it = a->begin(); it != a->end(); it++) {
        unionSet.insert(*it);
    }

    for (it = b->begin(); it != b->end(); it++) {
        if (unionSet.count(*it) == 0) {
            unionSet.insert(*it);
        }
    }

    return unionSet.size();
}
```

Interseccio->Recorrem cada set un cop així que és fita de $n_1 + n_2$

```
float opInterseccion(set<unsigned int> *a, set<unsigned int> *b) {
    set<unsigned int> interseccion;
    set<unsigned int> *mayor, *menor;
    set<unsigned int> ::iterator it;

    if (a->size() > b->size()) {
        mayor = a;
        menor = b;
    } else {
        mayor = b;
        menor = a;
    }
    for (it = menor->begin(); it != menor->end(); it++) {
        if (mayor->count(*it) > 0) {
            interseccion.insert(*it);
        }
    }
    return interseccion.size();
}
```

Així doncs el cost globalment a la part 2 es fita de $(d * n)$

Part 3:

Aquí aplicavem una funció per treure valors aleatoris dels sets que representaven els documents, pensavem que ens feia falta que per comparar dos sets tinguessin el mateix tamany. Això ens feia tenir la quantitat de semblança en percentatge fàcilment, però la feia molt poc fiable, així doncs hem decidit de no incloure-la a la versió final del programa. A més a més era el coll de botella del programa a nivell asimptòtic.

Part 4:

Genera els valors per aplicar el hash XOR (fita de t , on t és el numero de taules). Seguidament generem un vector tridimensional, de tamany $d * t * \text{ranghash}$ (fita de $d * t * \text{ranghash}$).

Després tenim un bucle que es repetirà $d * t$ vegades. (A la imatge MINSIGNATURA hauria de ser hashrang, el màxim valor de hash que volem)

```
//PARTE 4
//es el moment de generar les signatures Minhash(matrius booleanes)
//el primer index és el document, el segon es el numero de funció i el tercer es el valor que s'ha hashejat

//genera los num de las XOR
for(int i = 0; i<numtaules;i++){
    hash_xor_values[i] = randomint();
}

signatures = vector<vector<vector<bool>>>(NUM_DOCS, (vector<vector<bool>>(numtaules, vector<bool>(MINSIGNATURA, false))));
//print(&documents[0]);
for(int d = 0; d<NUM_DOCS; ++d){
    for(int f = 0; f<numtaules;++f){
        xor_hash(&documents[d],signatures[d][f],hash_xor_values[f]);
    }
}
```

Funcions a destacar:

xor_hash:

Recorrerà tot el set un cop, així que tindrem fita de n

```
//Pre: A conte els w k-shingles, n es un numero random
//Post: Retorna el set havent aplicat un xor entre cada posicio d'a i n
void xor_hash(set<unsigned int> * a, vector<bool>& s, int n) {
    set<unsigned int> ::iterator it;

    //vector<bool> result(a->size(),false);
    for (it = a->begin(); it!=a->end(); ++it) {
        s[( *it xor n)%s.size()] = true;;
    }
    //return result;
}
```

Així doncs el cost globalment a la part 4 es fita de $\max(d * t * n, d * t * \text{ranghash})$

Part 5:

Aquí calculem les similituds del primer document amb cada un dels altres, així doncs afegim fita de d .

```
//PARTE 5
//ara comparem el document D1 amb [D2...DN] fent servir LSH
vector<vector<int> > sim_taula(NUM_DOCS-1, vector<int>(numtaules));

for(int i=0;i<NUM_DOCS-1; i++){
    sim(signatures[0],signatures[i+1],sim_taula[i]);
}
```

Funcions a destacar:

sim:

Està format per dues subfuncions, itera per el número de elements que hi ha donat un document i t taules de hash(t) i per cada element a cada signatura(ranghash), així doncs té cost $(t * \text{ranghash})$

```
//aquesta va de ma amb comparacio_hashes
int com_semlants(vector<bool> prim, vector<bool> seg){
    int eresante=0;
    for(int i =0;i< prim.size();i++){
        if(prim[i]==seg[i])eresante ++;
    }
    return eresante;
}

//Pre:sign1 i sign2 tenen la mateixa llargada i amplada
//post retorna la quantitat de semblances entre la signatura de dos documents(matriu de booleans)
void sim(vector<vector<bool> > & sign1, vector < vector<bool> > & sign2,vector<int> & result ){
    for(int i= 0;i<sign1.size();i++){
        //cout << "hai" << endl;
        result[i]=com_semlants(sign1[i], sign2[i]);
        //cout << result[i] << endl;
    }
}
```

Així doncs el cost globalment a la part 5 es fita de $(d * t * ranghash)$

Part 6:

Recorrem els resultats de cada document per cada taula de hash i per Jsim així que recorrem $d*(t+1)$

```
//PARTE 6
//printem resultats
cout << "document" << '\t' << "Jsim" << '\t';
for(int i= 0;i<numtaules;++i) cout << "hash" << i << '\t';
cout << endl;
for(int i=0;i<NUM_DOCS-1;i++){
    cout << nomdocs[i+1] << '\t' << jacobianas[i] << '\t';
    for(int j=0;j<numtaules;++j){
        cout << (float)sim_taula[i][j]/(float)MINSIGNATURA << '\t';
    }
    cout << endl;
}
cout << "MINSIGNATURA" << '\t' << MINSIGNATURA << endl;
```

Activate Windows
Go to Settings to activate Windows.

Així doncs el cost globalment a la part 5 es fita de $(d * t * ranghash)$

Finalment, el nostre cost asimptòtic serà:

Part 1+ Part 2 + Part 4 + Part 5 + Part 6=

= $fita(k*n) + fita(d*n) + fita(\max(d * t * n, d * t * ranghash)) + fita(d * t * ranghash) + fita(d * t * ranghash) =$

$fita(k*n) + fita(d*n) + fita(\max(d * t * n, d * t * ranghash)) + fita(d * t * ranghash) =$

$= fita(\max(d * t * n, d * t * ranghash, k * n))$

Anàlisi de Resultats obtinguts

En el .zip hem adjuntat una carpeta anomenada samples en la qual trobaràs tots els .txt amb els resultats obtinguts i als que ens referim.

Les dades que utilitzem per fer els samples són: lorem (Document base i del qual es fan totes les comparacions amb la resta, és el lorem ipsum en espanyol), lore (lorem ipsum en turc), lorechi (lorem ipsum en chino), test-0 (document generat random a partir de permutacions de 50 paraules), HP (El llibre Harry Potter i la Pedra filosofal en espanyol), HP2 (El llibre Harry Potter i la Cambra secreta en espanyol), test-1 (document generat random a partir de permutacions de 50 paraules), test-3 (mateixa filosofia que el test-1) i un altre test-3 idèntic a l'últim mencionat.

Sample-0

La k dels k-shingles és 4. En Jsim tots seràn 0 degut a que cap document és còpia idèntica de lorem.

Los lorem ipsum no tenen cap mena de semblança degut a que són en idiomes diferents pel que no haurien de coincidir en pràcticament res.

Els llibres de Harry Potter parlen d'un tema similar al ser d'una mateixa saga, per tant el resultat de les funcions de hash en el lsh són molt similars o inclús el mateix. Com es compara amb un document en espanyol i els llibres de Harry Potter també, donen més col·lisions que la resta, generats en llengües diferents.

test-0, test-1 i test-3 no comparteixen ja que $k=4$ i aquests documents estan generats a partir de permutacions de paraules, així que la probabilitat de que exactament 4 paraules coincideixin en el mateix ordre a dos documents diferents és enormament baixa. Però al tenir dos test-3 podem observar que coincideixen al 100% entre ells, pel que podem determinar que són iguals, com clarament sabem, ja que és el mateix document.

Sample-1

La k dels k-shingles és 1 en aquest cas. En Jsim seguiran essent tots iguals a 0 pel mateix motiu. En aquest cas al agafar-ho per $k=1$ donen números més alts degut a que la probabilitat de col·lisions és molt més alta.

Els lorem ipsum segueixen sense tenir cap mena de semblança pel mateix motiu que al sample-0.

Els llibres de Harry Potter segueixen la mateixa tendència pel mateix motiu explicat anteriorment.

En aquest cas test-0, test-1 i test-3 comparteixen els valors al 100%, degut a que com agafem $k=1$ i aquests documents estan formats per les mateixes paraules, amb això podem determinar que aquesta k no ens serveix, ja que clarament són documents diferents.

Sample-2 i Sample-3

El Sample-2 s'executa per $k=10$ i el Sample-3 s'executa per $k=25$.

L'informació obtinguda en aquesta execució no és més rellevant que l'obtinguda en el Sample-0, per tant podem determinar que no és molt útil augmentar la k (en aquest cas), de fet únicament ens augmenta és el temps, com podem comprovar en el graf següent.

Sample-4

Aquest sample és el cas base demanat a l'enunciat, generar 20 documents per mitjà de permutacions de 50 paraules, en el README hem explicat una manera per la que es puguin generar més exemples a mode de test.

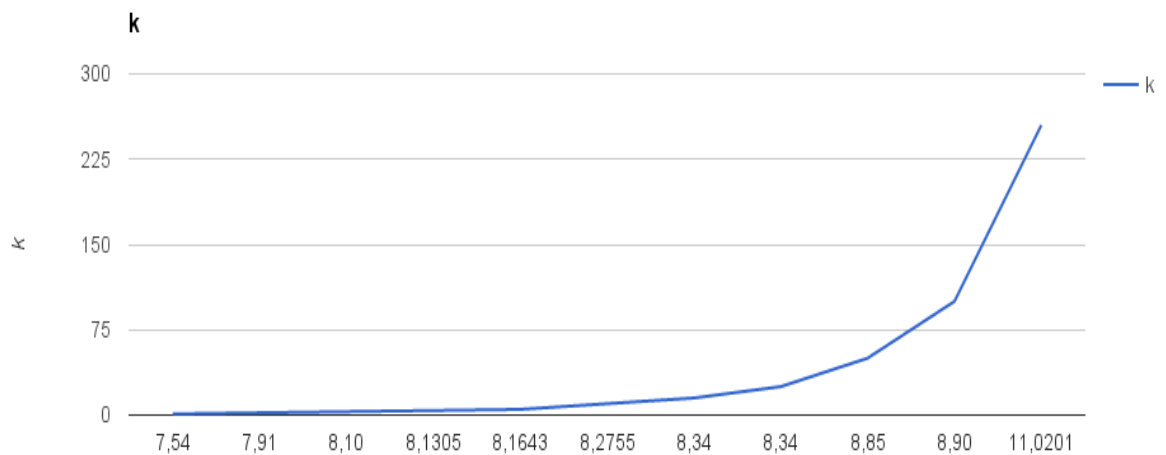
Aquests test's estan generats per permutacions de les mateixes paraules, per tant, com s'ha explicat anteriorment $k = 1$ no seria una bona k ja que donaria que tots son idèntics. Per aquest sample hem triat $k = 3$, ja que hem pogut comprovar amb les proves prèvies que la seva relació precisió/cost és bona per determinar semblances entre documents i triar una mes gran nomes augmentaria el cost.

Com s'ha explicat anteriorment, cap dels test es pot determinar com a còpia d'un altre.

Sample-5

Realitzat per $K=3$. En aquest cas hem agafat cinc texts, entre ells fent permutacions de frases o inclús deixant-los idèntics, per comprovar que Jsim funcionarà bé i hash també si es realitzen permutacions d'aquest tipus.

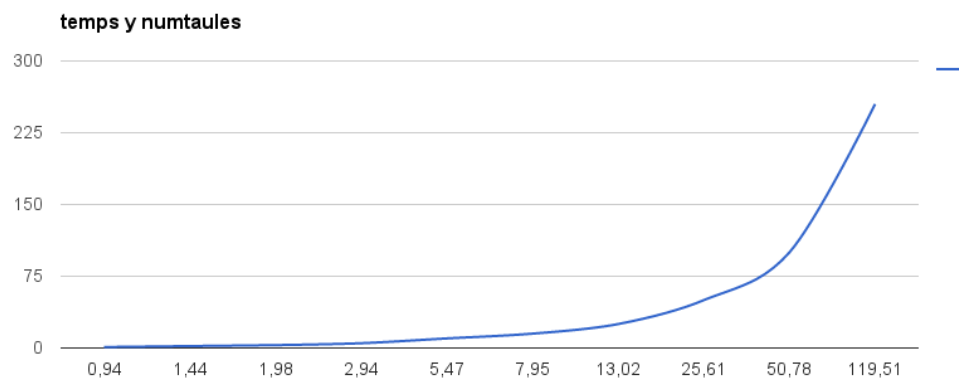
Com podem comprovar en el resultat la Jsim es molt pròxima a 1 en els 5 casos i els hash molt similars per cada compareX.txt.



En aquesta funció, agafant els documents comentats a l'inici de l'anàlisi dels resultats, n'hem anat augmentant la k per així poder veure com augmenta el temps en funció. Podem observar que en aquest cas la funció augmenta exponencialment respecte de k , pel que no ens convendrà agafar una k excessivament gran.

Hem de partir que utilitzem documents no petits, en els quals es troben llibres de unes quantes pàgines.

Com a conclusió n'extraïem que una k òptima seria entre $k = 3$ (considerem suficient per muntar una frase) i una k suficient com perquè el temps no creixi desmesuradament. En aquest cas una k entre $3 \leq k \leq 40$ seria òptim.



Aquest gràfic està format per el número de funcions de hash creades i el temps, com podem comprovar el temps augmenta exponencialment depenent de l'augment de funcions de hash, un altre com s'ha de tenir en compte que aquest gràfic ha estat realitzat amb documents força grans.

Com a conclusió n'extraïem que l'informació proporcionada per l'augment de taules de hash no és tan rellevant com per l'esforç temporal que suposa, per tant, en aquest cas, un total de 30 funcions de hash seria suficient, així més o menys tarda el mateix temps que amb la k proposada a l'anterior gràfic.