Pràctica Algorismia: Detecció de similituds de documents amb hashing

Quadrimestre de tardor: 2018-2019

Pol Renau i Carla Lara

Continguts:

1. Introducció

L'objectiu d'aquesta pràctica resideix en solucionar el problema de trobar el grau de similitud entre un conjunt de documents i identificarlos com a còpies.

Per tal de dissenyar una bona solució hem tingut en compte varis factors que ens porten a dur a terme varies implementacions semblants entre elles. La millor solució serà la que obtingui millors resultats en la majoria d'experimentacions que es facin sobre totes les versions. Els millors resultats tant en eficiència temporal com en correctesa de la solució.
Concretament, hem implementat varies versions dels següents algorismes: *Jaccard, min-hash, locality sensitive hashing.*

2. Descirpció del problema

Com el problema que ens ocupa es tracta de detectar documents similars, la entrada serà un conjunt de varis documents que s'hauràn de comparar tots entre tots i la sortida serà un conjunt de parells de documents tals que la semblança entre ells superi un tant per cent preestablert, suficient, com per suposar que ambdós documents són semblants.

En funció del algorisme que estiguem implementant per assolir l'objectiu, hi ha algunes variacions a l'hora de tractar-lo que poden resultar més o menys eficaces. És per aquest motiu que hem decidit realitzar aquelles que ens han semblat millors per comparar-les i escollir la millor versió de totes com a algorisme final.

En els següents apartats ens centrarem en l'explicació i definició d'aquests algorismes, de les seves respectives funcions i de l'experimentació.

3. Disseny del conjunt d'algorismes

Donada una entrada, que consisteix en el conjunt de documents que cal comparar, caldrà fer un anàlisis per parells de documents. L'ordre en que aquests documents es comparin no és important ja que el resultat serà el mateix, és a dir, no importa si es comparen primer el document 1 i el número 10, com si es comparen primer els documents 2 i 4. De la mateixa manera, fer la comparació dels documents 1 i 2 donarà el mateix resultat que fer la comparació dels documents 2 i 1. Per tant, donada una entrada de **n** documents, es faràn un total de $\sum_{i=1}^{n}(n-i)$ comparacions de documents.

Per cada parell de documents comparats s'obté el grau de similitud que proporciona l'algorisme executat, aquest serà comparat amb un llindar preestablert. Si el resultat és inferior al llindar, els documents es donaran per no similars. Altrament es classificaran com a similars i el programa mostrarà per pantalla quins són aquests dos documents.

És un requisit que els documents no tinguin accents ni símbols estranys, ja que la codificació en ASCII d'aquests caràcters complica bastant la comparació, atès que no són considerats com a únic char, sinó que són més de un char, i considerem que pot provocar errors en els resultats de manera que deixen de ser fiables.

4. Implementació dels programes

a. Estructura de dades

Fem servir varies estructures de dades per emmagatzemar la informació que trobem en els documents. Evidentment, en funció de cada implementació ens serà més útil fer-ne servir una o una altre.

- map: Per a aquells algorismes que necessitem fer un comptador de paraules, és a dir, quines paraules apareixen en un document i en quina quantitat.

- set: Per a aquells algorismes que els hi interessa saber quines paraules hi ha en els documents però la quantitat d'aquestes no és necessària.

- matriu: Per a aquells algorismes que utilitzen funcions de hash, acostumen a tenir tamany de Funcions de Hash * nº de Documents.

b. Mètodes

Tenim un conjunt de mètodes que els nostres algorismes fan servir. Aquests es poden trobar presents en més d'una implementació i tot:

- rellena y leeDocs: S'encarregan d'omplir les corresponents estructures de dades amb la informació que contenen els documents. Aquesta funció varia una mica depenent de la versió però la idea general és la mateixa en totes.

- jaccard: Tal i com diu el propi nom, es tracta de la funció que calcula la similitud de dos documents pel mètode de jaccard.

- leeTodo: Transforma tot el contingut d'un document a un únic string.

- genKsingle: Genera totes les substrings de tamany "K" a partir d'un string.

- purga: retorna la string p on només hi ha lletres a - z en minuscules i majusculas, la resta ha estat eliminat (comes , símbols puntuació, espais en blanc ...)

- afegeixParaules: donat un string, crea tots els subtrings de tamany k possible sobre aquest, i les insereix en un vector que rep per paràmetre.

- getsingleMax: dins de tot el coprus de documents, retorna el valor més gran que la funció "afegeix paraules" fa per cada document.

- setDoc: pasa totes les paraules d'un document a un Set, sense contemplar les repetides.

- noRepGen: dins de tot el corpus de documents, crea un set que conté totes les paraules d'ell, sense contemplar les repetides.

- generaCM: genera una matriu de integers, de tamany nombre de paraules total (sense repetir) * nº documents. A cada posició CM[i][j], podem tenir o bé un 1 (la paraula i , es troba en el document j) o bé un 0, que indica el contrari que 1.

- generaSM: genera una matriu de tamany nº funcions de Hash * nº de documents, on cada fila es una funció de hash, que s'ha aplicat a cada paraula de cada document. A la posició SM[i][j], tenim el Hash mínim totes les paraules que formen part d'aquell document j, aplicant la funció de hash i.

- generateCandidates: fa la comparació amb els valors que toca de cada parell de documents, i ens diu si té una probabilitat més alta que el llindar establert, de ser semblant.

c. Control d'errors

- Si la d'execució al programa no és correcte, s'informa al usuari, mostran't-li per pantalla el "usage" executable.
- En cas que la ruta al document o el document no existeixi, es mostra un error que indica que no s'ha trobat cap document corresponent a la entrada que s'ha passat.
- En el cas dels algorismes que fan servir kshingles, si el valor de la k supera la quantitat de caràcters que hi ha presents en el document es mostra l'error.

  *En tots els casos s'aborta l'execució.

d. Implementacions

i. Generador

Busquem el llibre de don Quijote i agafem les 50 primeres paraules que trobem. Aquestes les posem a un document ".txt" per tal de fer-lo servir per la generació dels 20 documents de text. Abans de poder-ho fer servir, hem hagut d'eliminar tots els accents ja que ens causarà complicacions més endavant.

La forma més ràpida i eficient que hem pensat és fer un script amb C++ anomenat "generador". Aquest llegeix el document i emmagatzema les paraules en un vector de 50 posicions, on cadascuna d'elles és una de les paraules del document. Ara, fent servir el temps actual, s'escull una seed per a l'ordenació random.

En un principi havíem pensat d'implementar nosaltres el algorisme encarregat de fer la generació random però buscant per internet hem trobat una librería que et fa un "random_shuffle" d'un vector i hem pensat que era molt millor fer-la servir. Per tant, la funció "escriu" rep el vector de 50 paraules i el re-ordena de forma random fent servir la seed escollida. Un cop reordenat s'exporta en un documentocume.txt que té per nom "**Random_1.txt**", on el número del final es el número que identifica el document creat dels 20 que s'han fet.

ii. Jaccard

Es tracta d'una semblança que no té en compte el significat de les paraules que compara per averiguar semblança. És una bona pràctica ja que ens permet detectar:

- Documents idèntics on la diferència és que les paraules estan ordenades de forma diferent.
- Documents que són una còpia entre ells però en els quals se'ls hi ha fet petits canvis que els separen de ser idèntics.

- Documents textualment similars, que contenen algún paràgraf diferent o algun d'afegit però que tenen la mateixa base.

En el cas de voler averiguar la diferencia entre dos documents, es fa el càlcul de la semblança de JACCARD i es resta a 1: 1 - “jaccard similarity” de manera que s'obté la dada complementaria a la semblança, el que implica que és la diferència entre documents.

El requisit de l’entrada d’aquesta implementació és la ruta per accedir a tot el conjunt de documents que volem que es comparin, per exemple “../FitxersFont/Randoms/Random_1.txt”.

Per cada parell de documents a comparar es generen dos mapes, on la clau primària serà una cadena de caràcters i la clau secundaria, la quantitat de vegades que aquesta cadena apareix en el document. Un cop iniciats els maps es procedeix a executar l’algorisme de comparació.

Es creen dos integers incialitzats a 0 i un float que contrindràn:

- **total**: nombre total de paraules del document, és a dir, la unió dels dos maps.
- **inter**: nombre total de paraules que s’han trobat que coincideixen entre els dos maps.
- **f**: divisió de inter/total, és a dir, la divisió de la intersecció entre unió dels dos conjunts que estem intentant comparar. De manera que en f hi haurà un valor entre 0 i 1, tal que, quant més s’apropi a 1, vol dir que més semblants són els conjunts.

Es comença recorrent un dels dos maps, és indiferent quin dels dos, el iterador que recorre el segon mapa correspondria a la paraula a la que apunta el primer iterador. Si es troba la paraula, com és possible que la mateixa paraula aparegui més cops en un dels documents, procedim a agafar el mínim valor dels dos maps d’aquesta paraula (la intersecció) sumem aquest valor al

contador de la intersecció, y las restem de cada un dels contadors de total de paraules per cada mapa. Si no es troba la paraula no es fa res i es continua amb el bucle sobre el mapa 1.

Al acabar de iterar sobre mapa 1, hem de fer la intersecció, per a fer-ho, sabem que la unió de dos conjunts es la suma de a seva intersecció, més la seva disjunció. Aleshores el que fem és: unió= inter + contador de 1 + contador de 2. Ja que en contador 1 tenim el nombre total de paraules que no existeixen a 2, y a contador 2 tenim el nombre total de paraules que no existeixen a 1.

Per acabar es fa el càlcul de f i es mostra per pantalla, com la semblança dels dos documents.

El cost de l'algorisme és: $O(\sum_{i=1}^{d}(d-i)$ *(m*log(n))). on m és el tamany del set1 (document1), n es el tamany del set 2 (document2) y d es el nombre de documents del corpus a comparar.

- ***Versió 1***

  La cadena de caràcters d'interès que es posarà en el map seràn les paraules del document, és a dir, els substrings del document que es troben entre dos blancs, o al principi o al final del document.

- ***Versió 2***

  Fem servir k-shingles per determinar les cadenes de caràcters d'interès. Aquestes es formen generant tot el conjunt de substrings possibles del document de tamany k. La k haurà de ser especificada per l'usuari.

  Per aquesta versió, les substrings que es formaràn tindran en compte espais, signes de puntuació, etc.

- ***Versió 3***

  També es fan servir k-shingles per determinar les cadenes de caràcters d'interès. La diferència entre aquesta versió i l'anterior és que ara es farà una purga inicial. Amb això volem dir que s'eliminaran espais, signes de puntuació, etc, de manera que els substrings que es formin no els tindran en compte.

iii. MinHash

El estudi de semblanças entre documents mitjançant la semblança de Jaccard podria donar alguns problemes a l'hora de tractar amb un set de dades suficientment gran. És per això que s'introdueix la tecnica minhash.
Aquesta tècnica utilitza un algoritme basat en la randomicitat per estimar ràpidament la similitud de Jaccard.

Per tal d'aconseguir-ho mostrarem les dades, és a dir, el conjunt de sets, en forma de matriu. On les files seran tots els elements, iguals o diferents, presents en els diferents sets a comparar i en les columnes hi hauran els sets.
M[i][j] → 0 si el conjunt i no es troba present en el set j

→ 1 si el conjunt i es troba present en el set j

El problema es troba en termes d'espai, ja que computacionalment és més ràpid que jaccard.

Per reduïr el cost de búsqueda a la hora de comparar, fem una matriu de Hash/Signatures, en la que reduïm tota la entrada a tenir un valor per cada document aplicant una funció de Hash, per tenir millor qualitat de predicció, tenim varies funcions de hash, que ens permetrà fer la comparativa de Jaccard dels documents.

El cost teoric calculat d'aquest algorisme es O(n*d*log(n) + n*d*hf+ $(\sum_{i=1}^{d}(d-i)$ *hf)), on n es nº total de paraules, d nº total de documents i hf nº de funcions de Hash. Aquest cost no el podem reduir més, ja que no sabem quin dels valors pot ser més gran, depén de la entrada, ja que poden haver-hi varis valors assimptòtics, i el resultat variaria depenent d'aquests.

Les funcions de hash es basen en, a partir d'una entrada, mitjançant una sèrie de modificacions, obtenir una sortida que hagi "codificat" aquesta entrada. La entrada ha de ser un conjunt finit d'elements (ja siguin strings,...) i els converteix en un rang de de sortida finit, normalment de la mateixa longitud.

La idea és que les funcions de hash serveixin per representar qualsevol entrada com una cadena finita.

Una avantatge és que té poc cost computacional i en temes d'espai, a més a mes, per una mateixa cadena d'entrada retorna exactament la mateixa cadena de sortida,

- ***Versió 1***

  Per cada document llegim les paraules que conté, i apliquem una funció de hash sobre les paraules d'aquest document, i guardem com a signatura d'aquell document, la paraula amb el valor mínim de Hash.

- ***Versió 2***

  Construim una matriu de Boleans, que ens diu si una paraula està en el document o no (tamany: nº total de paraules * nº documents). A aquesta matriu, li apliquem un hash per cada fila, de manera que "desordenem" la matriu, aleshores a la taula de hash/Signatures, guardarem la primera fila de cada columna, que té un 1 de la matriu desordenada.

- ***Versió 3***

  Aquesta versió usa la mateixa tècnica que l'anterior, però en lloc d'agafar les paraules que venen tal qual del text, creem paraules de longitud k, de la mateixa manera que ho fem per Jaccard_v3. La resta de la técnica es la mateixa que per la versió 2 d'aquest algorisme.

iv. Locality Sensitive Hashing (LSH)

Es refereix a un conjunt de funcions de hash de dades classificades per grups de manera que les aquelles que són iguals o semblants tenen una elevada probabilitat d'acabar classificades en el mateix grup, anàlogament aquelles que no tenen res a veure es troben en grups diferents.

Per tal de trobar grups de documents que son similars dintre un conjunt de documents farem servir l'algorisme LSH que separarem en 3 passos:

1. Transformem cada document en un set de caràcters de longitud k (apliquem k-Shingles)
2. Apliquem minhash, explicat anteriorment.
3. Per tal de reduir les comparacions entre cada parell de documents, apliquem un hash dins de la matriu de Hash/Signatures, que ens agrupara les paraules semblants entre elles a un mateix hash, de tal manera que en lloc de fer tantes comparacions com nombre de Funcions de Hash que tenim, per cada parell de documents, les reduïm al nombre de bandes.

Ens indica bàsicament si la semblança entre dos documents és superior o no a una cota "threshold" donada. Si és superior, l'algorisme ens indica que són documents que tenen una probabilitat bastant alta per ser sembalnts.

LSH agrupa els hashos en bandes de manera que no compares hashos individuals sino que compares bandes. Aquesta tècnica requereix de menys consultes i, a més a més, menys resultats hauràn de ser processats per cada query individual. Per tant això és un guany clar en temps respecte l'anterior algorisme que estavem tractant. El problema es que es perd precisió en els resultats, per tal de que aquesta pèrdua de precició no sigui molt gran, cal posar valors de bandes i tamany d'aquestes, que siguin valors calculats i no uns valors random.

El cost teoric calculat d'aquest algorisme es O(n*d*log(n) + n*d*hf+ ($\sum_{i=1}^{d}(d-i)$ *b)), on n es nº total de paraules, d nº total de documents, hf nº de funcions de Hash, i b es el nº de bandes. El podem desenvolupar,simplificar ja que sabem que aquest sumatori es igual aO(n*d*log(n) + n*d*hf+ $\frac{(d-1)*(d)}{2}$ *b))

5. Experimentació

a. Entorn d'experimentació

El conjunt d'experiments que s'han realitzat i que s'expliquen posteriorment, s'han realitzat sota les següents condicions:

- **Processador:** intel-core i7.
- **Memòria RAM:** 8GB
- **Sistema Operatiu:** Linux Mint
- **Compilador:** g++
- **Llibreria per mesurar el temps:** La que proporciona g++ → time.h

Totes les proves que s'han realitzat, tant de temps d'execució com per mesurar el grau de correctesa dels algorismes, s'han fet sobre les condicions esmentades per tal d'evitar al màxim les possibles variacions de resultats provocades per fer les proves en diferent ordinadors o condicions.

b. Jocs de prova

Des d'un bon començament, quan vam plantejar els algorismes que toca implementar per tal d'aconseguir detectar la semblança entre diversos documents mitjançant tres tècniques diferents, vam pensar en les diferents versions que s'han explicat en el capítol anterior. Per tal de quedar-nos amb la versió que millors resultats dóna de cada tècnica, les hem executat amb una sèrie de documents que sabem el que contenen i els resultats que haurien de sortir. El conjunt de documents i els respectius resultats són;

A. Conjunt de documents molt dispersos entre ells. Per a aquelles implementacions que necessiten un valor k, el definirem prou petit al principi i poc a poc l'anirem incrementat per veure l'efecte que té el valor d'aquest en la detecció de similituds.

   Els resultats que s'haurien d'obtindre per tenir un bon paràmetre fixat és que ningún parell de documents del conjunt és semblant, és a dir, que no supera un grau de semblança preestablert prèviament per nosaltres.

Per a valors petits de k (per exemple, 1 o 2,..) esperem que hi hagi algun parell de documents que es determinin com a semblants. A mesura que la k comença a prendre valor més grans, esperem que ningún parell de documents resulti com a similar,

B. El conjunt de 20 documents que hem generat a partir del llibre "HarryPotter" i que contenen les mateixes paraules però en ordre totalment diferent. Tal i com en el joc de proves anterior, la k començarà amb valors petits i anirà incrementant i el que hauria de resultar és que la gran majoria de parells de documents no són semblants

C. Conjunt de documents pràcticament idèntics, amb més o menys la mateixa quantitat de paraules, on les poques diferències que hi haurà entre ells serà de alguna paraula diferent. Com en els casos anteriors, la k comença amb valors petits i l'anem augmentant.

   El que esperem és que tots els parells de documents es detectin com a semblants.

D. Conjunt de documents casi idèntics, els quals tenen alguna, però poques, paraules diferents. La diferència amb el joc de proves anterior es que aquí, els paràgrafs entre els documents estan canviats d'ordre, igual que algunes frases, el que esperem que compliqui la detecció de documents similars, tot i que la gran majoria de documents haurien de ser semblants entre ells.

Després d'haver sotmès totes les versions dels algorismes a aquest conjunt de proves, hem fet una comparativa de grau d'error de cadascun d'ells. Al final, aquells que més s'aproximaven a les solucions esperades per cada versió són:

- **Jaccard** → La versió número 3
- **Minhash** → La versió número 3
- **Locality Sensitive Hashing** → només hi ha una versió i funciona com era d'esperar.

Per tant, les versions que farem servir per a les diferents experimentacions seran els esmentats.

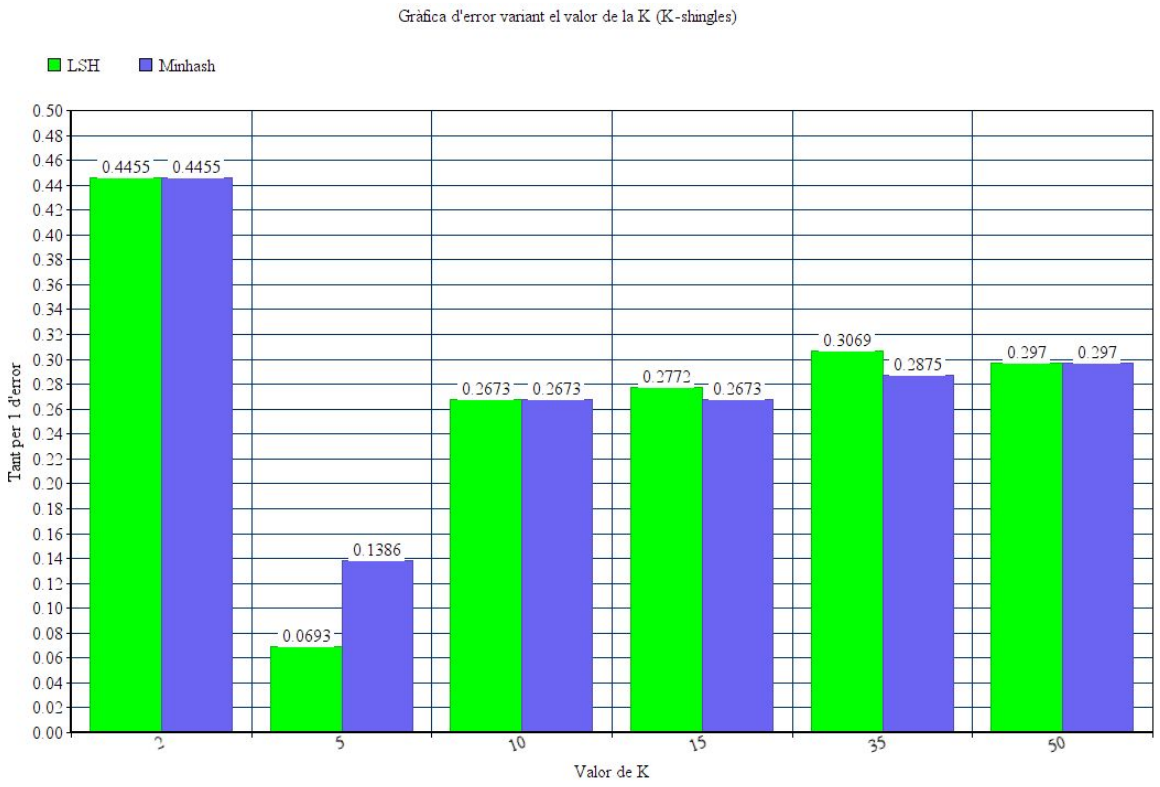
c. Escollint els millors valors del paràmetre k

Per a totes les versions hi ha paràmetres d'entrada que fan que els resultats variin en funció d'ells. Evidentment, per a alguns paràmetres obtindrem millors o pitjors resultats. El nostre objectiu és fer una sèrie de proves i quedar-nos amb el valor d'aquests paràmetres que més s'ajusti a la correctesa de les solucions.

Per aquest motiu executarem els algorismes escollits en l'apartat anterior amb els mateixos jocs de prova però en diferents condicions. S'executaran tots per als següents valors de k: 2, 5, 10, 15, 35 i 50, i per a cadascuna d'aquestes ens quedarem amb el tant per cent d'error que s'ha produït al fer les comapracions entre els documents respecte els resultats esperats. La k final escollida serà la que en majoria tingui un error mínim per a tots els jocs de proves i algorismes diferents.

Per cada execució s'ha realitzat un conjunt de proves amb valors random de nombre de funcions de Hash, s'ha mostrejat la mitjana per cada execució de K.

Aquests són els resultats:



Com era d'esperar, per valors de k tan petits om 2 es produeix un error molt elevat ja que els conjunts de cadenes de caràcters que es poden formar a partir de 2 caràcters té una probabilitat molt elevada de ser present en la gran majoria de documents. A partir de k = 10 el error comès per els dos algoritmes va augmentant. Com el que volem és un valor de k que funcioni millor per a tots els algorismes, ens surt a compte quedar-nos amb la **k = 5** ja que com veiem a la gràfica, a partir d'aquell valor es comença a disparar la probabilitat d'error en els dos algoritmes.

A partir d'ara, les següents experimentacions es faran amb aquest valor establert.

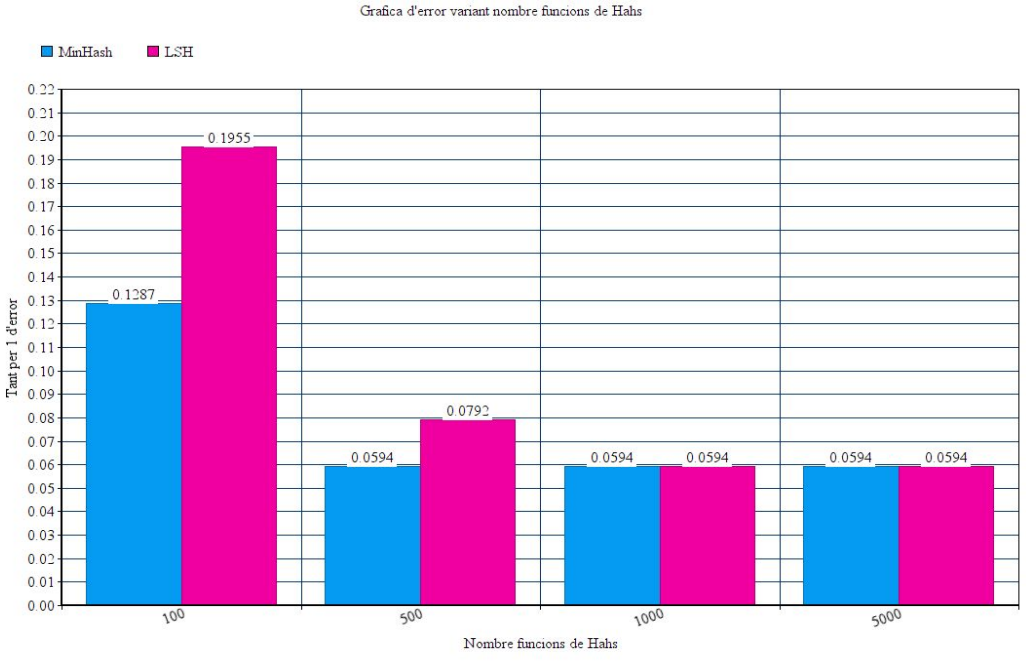
d. Escollint la millor quantitat de funcions de hash

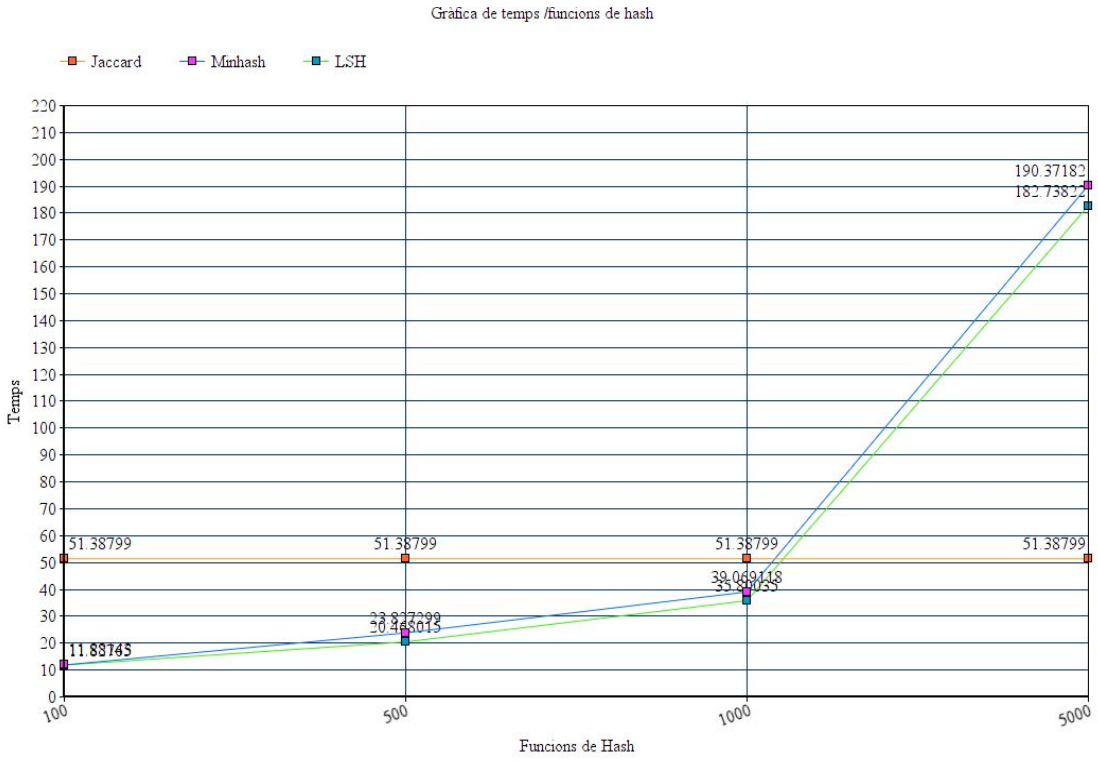
Per als algorismes de minhash y LSH s'han de fer servir una quantitat de funcions de hash qualsevol. De la mateixa manera que en l'apartat anterior, en funció d'aquesta quantitat s'obtindran millors o pitjors resultats tant temporalment com de correctesa.

L'objectiu és sotmetre els algorismes a una serie de proves de les quals esperarem un resultat. En funció d'aquests escollirem una quantitat de funcions de hash a executar a partir dels següents apartats. Aquesta experimentació es farà amb els jocs de prova i algorismes escollits ja anteriorment. La quantitat de funcions de hash a la que es sotmetràn seràn = 100, 500, 1000 i 5000.

Per cada valor de funcions de Hash s'ha executat varies vegades l'algorisme, i el que es mostra a la gràfica, es la mitjana d'aquestes execucions, a més a més, com LSH fa servir bandes i el tamany d'aquestes, per a que aquests no pertorbin les dades, hem fet que siguin valors randoms tals que (bandes* tamany = Número funcions de Hahs).

Emmagatzemem els resultats en termes de temps i l'error comes es mostren en les següents gràfiques:





Tal i com es pot veure i tal i com pensavem, a mesura que s'incrementen les funcions de hash, l'error disminueix. El problema d'això és que el temps d'execució augmenta considerablement a mesura que s'augmenta les funcions de Hash. Per tant creiem que la quantitat de funcions de hash que hem de fer servir ha de ser una que dongui un error acceptable en un temps d'execució que no es dispari.

Evidentment, quantes menys funcions de hash més error i quantes més funcions de hash més temps, per tant hem d'escollir un valor entre el 500 i el 1000. Creiem que és més raonable fer servir 500 funcions de hash ja que, tot i que l'error és una mica superior al que es produeix amb 1000, el temps és bastant inferior, per tant considerem que val la pena sacrificar una mica d'error per tal de tenir un algorisme més ràpid, ja que LSH no deixa de ser una aproximació, per tant sempre hi haurà una probabilitat d'error.

A partir d'ara es faràn servir **500 funcions de hash**.

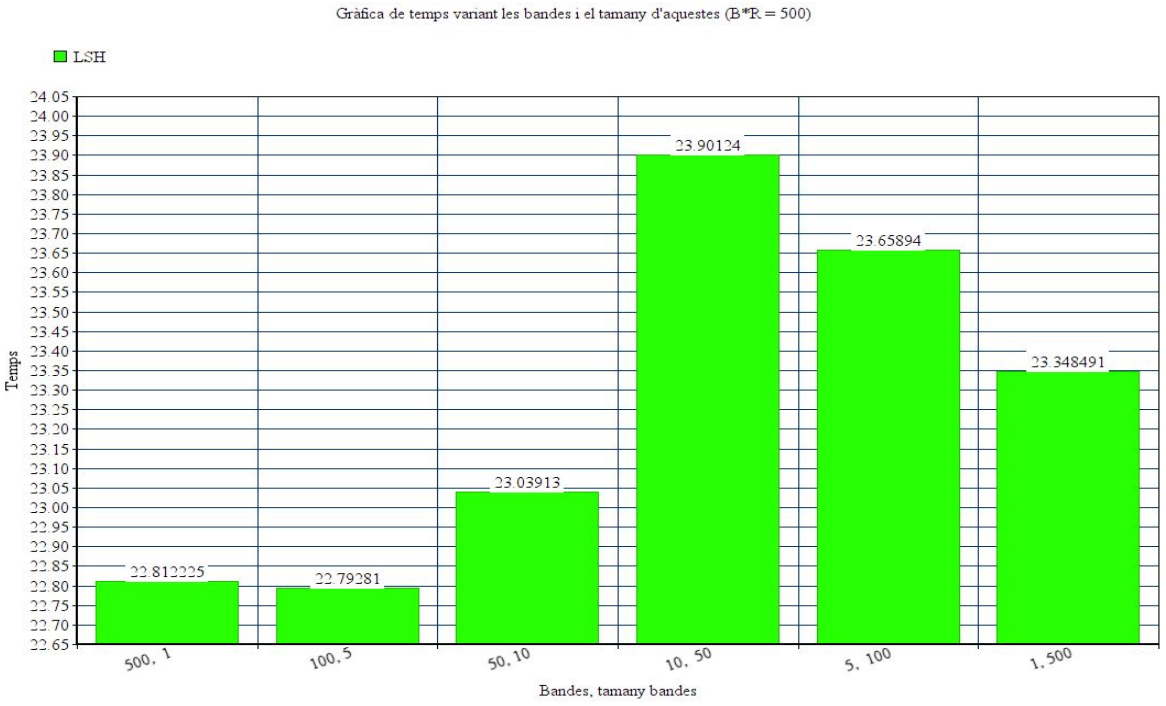
e. Escollint els millors valors de b i r

Tal i com s'ha fet en els apartats anteriors, hem de predeterminar els valors dels paràmetres sobre els quals l'usuari pot tenir influència. En aquest cas ens centrarem en els que falten per determinar del locality sensitive hashing.

Sotmetrem l'algorisme a les mateixes proves que els anteriors, és a dir, als mateixos conjunts de documents i calcularem l'error comès i el temps d'execució empleat, en funció dels següents valors de b = 500, 100, 50, 10, 5, 1 i r = 1, 5, 10, 50, 100, 500, assignats respectivament.

En aquest cas, com no queden més paràmetres lliures, no és necessari executar cap parametre de forma random, sino que es faràn amb els parametres calculats anteriorment.

Els resultats són:



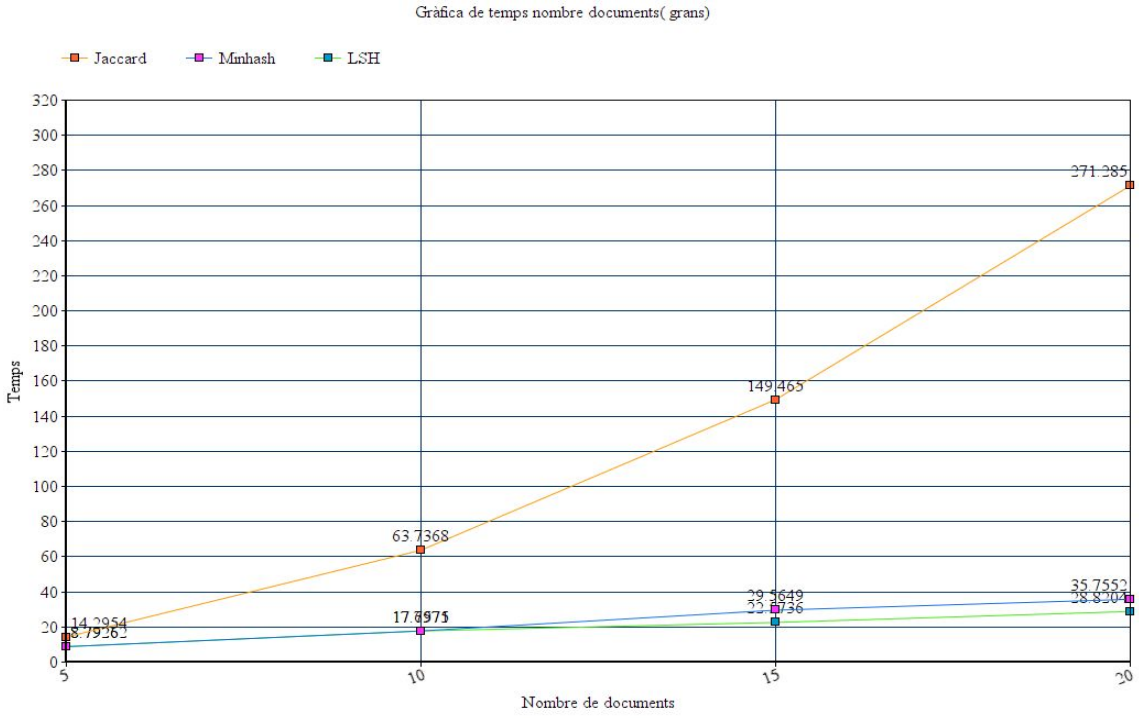
Com podem veure en la gràfica, el parell de b i r que ens mostra menor error, es el de b = 100 i r = 5. A més a més, veiem que aquest mateix parell, també es el que té un temps més reduït, tot i que realment la diferència no sigui molt gran en quan als temps d'execució. Per aquest motiu hem decidit quedar-nos amb la b = 100 i la r = 5.

f. Comparacions de temps d'execució

Hem sotmès els tres algorismes que detecten similituds a proves amb un conjunt de documents, tant els esmentats prèviament com d'altres. Hem executat 5 vegades cada algorisme amb els paràmetres definits anteriorment i el mateix conjunt de documents, aleshores hem fet la mitjana entre els resultats de temps, ja que com es fa ús de random en algun d'ells creiem que pot tenir algun efecte. Per la nostre sorpresa els valors del temps no canvien gaire.

També hem fet varies execucions amb diferents conjunts de documents, és a dir, primer comparant 5 documents, després d'un conjunt de 10, etc. Per finalment veure com afecten la quantitat i grandària d'aquests documents. L'hem executat amb documents que tenen aproximadament unes 80.000 paraules.

Els resultats són els següents:



Com es pot veure, a mesura que augmenten els documents, els temps també ho fan, però el que més repercussió té és **jaccard.** Evidentment, no només depèn de la quantitat de documents que es comparin sino de la grandària d'aquests, ja que no es el mateix fer-ho amb 20 documents que tinguin 50 paraules que amb 20 documents que en tinguin 50.000. Per tant, el temps acaba depenent de la grandària dels documents principalment.

i. Jaccard

Es podria dir que considerem que jaccard és el algorisme que dóna un valor més exacte de la similitud real entre dos documents. El problema que té aquest és el temps que triga en fer totes les comparacions corresponents per arribar a la bona solució. Per pocs documents o per conjunts de documents no massa grans es podria determinar com a algorisme òptim.

ii. Minhash

Aquest algorisme ens proporciona una aproximació de la similitud entre parelles de documents dintre d'un conjunt de documents a comparar. És per això que resulta llògic que el temps d'execució sigui mínim que en el cas de jaccard, però també és evident que això suposa sacrificar part de la correctesa de la solució proporcionada, és per aquest motiu que minhash és una aproximació.

Per tant creiem que pot ser molt útil fer servir aquests algorisme per casos en que hi ha un conjunt molt gran de documents a comparar o per quan aquests tenen un tamany molt gran, de manera que obtindrem una bona aproximació en un temps raonable.

iii. Locality Sensitive Hashing (LSH)

Aquest algorisme també ens proporciona una aproximació de la similitud entre parelles de documents dintre d'un conjunt de documents a comparar, és per això que el temps d'execució d'aquest algorisme és similar al de minhash. Tot i que sigui similar es veu que és lleugerament inferior, i això és degut a que és una aproximació menys exacte encara.

6. Conclusions

Gràcies a les comparacions de temps i de correctesa dels algorismes podem arribar a la conclusió sobre quin d'aquests creiem que és el més útil.

Amb les comparacions anteriors ens queda clar que jaccard pot ser molt exacte però en quant a temps es refereix pot arribar a ser problemàtic, per tant el destacariem de primeres.

Els altres dos algorismes estan molt a la par, això ens fa concloure que seria molt millor decantarnos per minhash, ja que en termes de temps són molt similars entre els dos i minhash dóna uns resultats lleugerament millors. Per tant, preferim resultats lleugerament millors que temps lleugerament millor.

Creiem que aquestes són bones tècniques per detectar similituds però tenen el problema de que no detectarien semblança entre dos documents que diuen el mateix però fent servir paraules diferents.

7. Bibliografia

PER EL ALGORISME D'ORDENACIO RANDOM:

- http://www.cplusplus.com/reference/vector/vector/vector/
- http://www.cplusplus.com/doc/tutorial/files/
- http://www.cplusplus.com/reference/algorithm/random_shuffle/

PER ENTENDRE JACCARD:

- https://en.wikipedia.org/wiki/Jaccard_index
- http://infolab.stanford.edu/~ullman/mmds/ch3.pdf
- https://www-users.cs.umn.edu/~kumar001/dmbook/dmslides/chap2_data.pdf

PER ENTENDRE K-SHINGLES:

- https://en.wikipedia.org/wiki/W-shingling
- http://infolab.stanford.edu/~ullman/mmds/ch3.pdf
- https://github.com/steven-s/text-shingles
- http://ethen8181.github.io/machine-learning/clustering_old/text_similarity/text_similarity.html

PER ENTENDRE LOCALITY SENSITIVE HASHING (LSH):

- https://medium.com/engineering-brainly/locality-sensitive-hashing-explained-304eb39291e4
- https://en.wikipedia.org/wiki/Locality-sensitive_hashing
- https://towardsdatascience.com/understanding-locality-sensitive-hashing-49f6d1f6134
- https://santhoshhari.github.io/Locality-Sensitive-Hashing/
- http://infolab.stanford.edu/~ullman/mining/2009/similarity1.pdf
- https://www.youtube.com/watch?v=dgH0NP8Qxa8

PER ENTENDRE LES FUNCIONS DE HASH

- https://es.wikipedia.org/wiki/Funci%C3%B3n_hash
- https://latam.kaspersky.com/blog/que-es-un-hash-y-como-funciona/2806/
- http://www.convertstring.com/es/Hash
- https://larevoluciondelbitcoin.com/funciones-de-hashing/
- https://en.wikipedia.org/wiki/Hash_function
- https://www.lifewire.com/cryptographic-hash-function-2625832

PER ENTENDRE MINHASH:

- https://en.wikipedia.org/wiki/MinHash
- https://cran.r-project.org/web/packages/textreuse/vignettes/textreuse-minhash.html
- http://mccormickml.com/2015/06/12/minhash-tutorial-with-python-code/
- http://ivory.idyll.org/blog/2016-sourmash-signatures.html
- https://www.cs.utah.edu/~jeffp/teaching/cs5955/L5-Minhash.pdf