Similitud entre Documents

Algorísmia - Facultat d'Informàtica de Barcelona

Pau Argelaguet
Rubén Marías
Victor Massagué

Continguts

1. Introducció i objectius

L'objectiu d'aquesta pràctica és estudiar diferents algorismes i implementacions utilitzades per la detecció de similitud documental. El concepte de similitud pot ser certament subjectiu, i es poden identificar almenys dos tipus de similituds entre documents: la ortogràfica i la lèxica. El treball s'ha en la identificació de la primera i no la segona, que tot i que també seria un problema molt interessant, implica tècniques diferents de les que s'usaran.

Així doncs, s'han implementat algorismes que utilitzen la similitud de Jaccard, la similitud per Minhash i LSH (*Locality Sensitive Hashing*). Aquestes implementacions s'han fet prenent la descripció proporcionada al llibre *Mining massive datasets* [1] com a base. Adicionalment, s'ha implementat una combinació de LSH i distància de Levenshtein amb la que s'han obtingut resultats que han permès aprofundir en la comprensió d'aquest tipus de tècniques.

En aquest document s'expliquen detalladament les decisions preses, així com el disseny i el procés d'implementació seguit. A més a més, es dedica un apartat a la validació experimental on s'hi detalla el disseny dels experiments per tal de determinar els paràmetres que optimitzen el funcionament i l'eficiència dels algorismes anteriorment mencionats, així com el conjunt d'indicadors utilitzats per evaluar la qualitat de les solucions. Finalment, es presenten les conclusions de la pràctica derivades dels experiment realitzats.

[1] http://infolab.stanford.edu/ ullman/mmds/ch3.pdf

2. Implementació

En aquest apartat es discutirà les decisions que s'han pres a l'hora de fer la implementació, començant per a les eines emprades, l'estructura de classes i fitxers, i finalment els algorismes proposats.

2.1 Llibreries i eines utilitzades

El llenguatge de programació ve imposat per l'enunciat de la pràctica, és a dir, C++. S'ha decidit usar-lo en la seva versió C++11, perquè ofereix un bon equilibri entre ser moderna i ser compatible amb la gran majoria de compiladors i sistemes actuals. Entre les funcions d'aquesta versió, han motivat el seu ús:

- Inferència de tipus (paraula clau `auto`).
- Flexibilitat en la sintaxi (`vector<vector<int>>`, en comptes de `vector<vector<int> >`)
- Iteradors menys *verbose* (`for(auto e : V)`).
- STL millorada.

El programa s'ha desenvolupat i provat sobre màquines amb Apple macOS i Linux Ubuntu. Com que fem ús de llibreries estàndard i multiplataforma, no hauria de suposar cap problema generar els binaris per a Microsoft Windows. Amb el codi font s'inclouen les instruccions per a generar els binaris.

S'ha fet ús extensiu de la orientació a objectes que ofereix C++ per tal d'encapsular i reutilitzar funcionalitats i codi. S'ha creat un fitxer d'utilitats comunes (`Utils`) i les tres classes, `Comparator`, `Document` i `Experiments`, que s'expliquen en els seus apartats corresponents.

A part de la STL, s'ha fet servir la llibreria Boost [1] en la seva versió 1.6.2. Boost, de fet, és un conjunt de llibreries de C++, de codi obert i molt renom, que implementa funcionalitats de tot tipus. Les funcionalitats específiques de Boost que s'han fet servir per la implementació d'aquest projecte han sigut:

- **CRC (Cycle Redundant Check)**, funció de hashing. S'explica més endavant el funcionament detallat i l'ús que se li ha donat.
- **Filesystem**, per a llegir arxius del sistema de fitxers. Es va valorar fer servir el sistema *built-in* de C++, però en aquest cas Boost ens permetia més funcionalitats amb un codi més simple.
- **Hash combine**, funció de hashing. S'explica més endavant el funcionament detallat i l'ús que se li ha donat.

Per a la compilació i enllaçat s'ha fet servir CMake [2]. CMake és un sistema de gestió de *builds* que facilita molt el desenvolupament. S'integra amb l'entorn de desenvolupament (IDE) emprat, CLion [3], i permet abstraure la compilació i l'enllaçat de la plataforma i del compilador concret que s'estigui usant.

2.2 Funcionament bàsic

El programa té dos executables, `main.cpp` i `mainExperiments.cpp`. En aquesta es procedirà a explicar el funcionament del primer, el segon és per executar els experiments que s'expliquen al pertinent apartat d'experiments.

El programa rep com a argument de línia de comandaments una direcció a un directori, que pot ser relativa o absoluta. Llavors, obrirà aquest directori, llegirà tots els fitxers amb extensió `.txt` que contingui i instanciarà un vector amb punters a objectes `Document`. Noti's que es fa servir un vector de punters a `Document` i no un vector de documents en si. S'ha pres aquesta decisió simplement per a una major eficiència a nivell de memòria (espacial).

Llavors es generarà una matriu de similituds per a aquest conjunt de documents, de tal manera que la casella `M(i,j)` contindrà la similitud, en escala de 0 a 1, entre el document *i* i

[1] http://www.boost.org/
[2] https://cmake.org/
[3] https://www.jetbrains.com/clion/

el *j*. Com que el valor de `M(i,j)` és el mateix que `M(j,i)`, realment només s'està calculant la meitat de la matriu, però s'imprimeix per la pantalla la matriu completa per facilitar-ne la comprensió.

Quan s'executa el programa principal, surten per pantalla tres matrius com la descrita anteriorment, calculades amb els tres mètodes diferents que es fan servir en la pràctica: **Jaccard, Minhashing i LSH** (detallats en els propers apartats). Per a cada una d'aquestes execucions, es controla el temps que tarda en executar-se. Per a fer aquestes comparacions, es crea un objecte `Comparator`, al qual s'associa el vector de `Document` i que retorna per a cada mètode, una matriu de similituds.

En el cas de voler modificar els valors de nombre de funcions, kshingles, nombre de bandes o rows s'han de modificar els valors per defecte de les variables globals corresponents en el codi.

2.2.1 El control de temps

Per a mesurar el temps, s'empra el paquet `ctime` de la STL de C++. Simplement es guarda en una variable el moment inicial de l'execució i el final. Aquests dos valors no són segons pròpiament dits, sinó cicles de rellotge de la CPU. Per a obtenir el valor en segons, es resta el principi al final i es divideix per una constant de conversió que incorpora C++, `CLOCKS_PER_SEC`.

2.2.2 L'objecte Document

Per a representar un document, no es llegeix el contingut del fitxer i s'aboca dins una una variable simple del tipus `string`, sinó que es crea un objecte. S'ha decidit implementar d'aquesta aquesta manera perquè un document té associats varis mètodes i propietats que seran útils en el transcurs de la pràctica que s'hi poden encapsular. D'aquesta manera, podem accedir al codi d'una manera més eficient i el manteniment i actualització del codi és més senzill.

L'estructura de Document és:

- **Booleà de validesa**, que indica si el document s'ha pogut llegir correctament de disc.
- **Dades**, un string amb el contingut del document en sí.
- **Path**, la direcció des d'on s'ha llegit el document.

2.2.3 L'objecte Comparator

L'objecte Comparator és l'encarregat de recollir un vector de punters a `Document` i fer les comparacions pertinents usant els tres mètodes ja esmentats, retornant la matriu de similituds. Efectivament, la gran part dels algorismes proposats a la pràctica estan implementats aquí (excepte algunes funcions auxiliars fortament lligades a Document).

L'estructura de Comparator és:

- Vector de punters a documents, on es guarden tots els punters dels documents que utilitzarem per analitzar la similitud.
- Threshold, valor límit utilitzat en els algorísmes LSH a partir del qual ens determina que si un valor és més gran o igual que aquest, hi ha una probabilitat molt alta de que siguin documents semblants. Més endavant en l'apartat de LSH s'explica com es calcula.
- Bands, nombre de bandes que utilitzem en els algorísmes LSH.
- Rows, nombre de files que tindrà cada banda en els algorísmes LSH.
- Hash Functions, nombre de funcions de hash que utilitzarem per l'algorísme minhash.

2.3 Algorismes

En els apartats que segueixen s'explica la implementació dels algorismes proposats a l'enunciat de la pràctica. En els dos primers apartats, Jaccard i Minhash, s'assumeix que s'està parlant de com obtenir la similitud entre dos documents, sent trivial la generació de la matriu de similituds (executar repetidament l'algorisme descrit per a les diferents parelles de documents). En l'apartat de LSH, s'explica aquest algorisme com a conjunt de documents (no té sentit emprar-lo tan sols amb dos documents).

2.3.1 Similitud de Jaccard

Per a implementar la similitud de Jaccard entre dos documents, s'ha fet servir la fórmula subministrada a l'enunciat com a base, on A i B són dos conjunts de shingles de longitud k que representen els documents A i B.

$$Jsim(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

Primerament, s'obtenen els documents i se n'extreuen els shingles de llargada k (la obtenció d'aquest valor arbitrari s'explica en apartats següents). Per a fer-ho, es va iterant per cada caracter i de la cadena de text que representa el document i se n'obté la subcadena de i a i+k. S'insereix cada cadena (shingle) a un `<set>` de la STL (s'ha triat aquest aquesta estructura de dades per la seva simplicitat a l'hora d'inserir elements i gestionar duplicats i perquè té les dues funcions explicades a continuació).

Per a calcular la similitud seguint la fórumla, es fan servir dues funcions de C++ específiques per a `<set>`: `set_intersection` i `set_union`, que donats dos sets d'entrada (els dos documents que volem comparar), genera un set de sortida amb la seva intersecció/unió. Llavors, és trivial obtenir la seva mida i fer-ne la divisió. El cost d'aquestes operacions és lineal, però el què és més interessant és que per a la implementació de la pràctica permeten un cert nivell d'abstracció amb les operacions de sets.

Els sets comentats fins ara són de strings, però per a fer els apartats següents, com que s'han de fer operacions de hashing, serà molt més còmode si són naturals. A més, exeperimentalment queda patent que el rendiment és molt més bo si es guarden els shingles hashejats com a naturals (degut a la facilitat d'una computadora per tractar naturals en comptes de cadenes de caràcters), per tant, així s'ha decidit fer, tant per a calcular la similitud de Jaccard com per als següents apartats.

Vist això, és evident que era necessària una funció per a hashejar un shingle (una cadena de text) a un número natural. Es va decidir que en comptes de haver d'exposar-se a tenir errors al dissenyar una funció de hash pròpia, era preferible fer servir un algorisme conegut i estudiat que garanteixi un número despreciable de col·lisions, una distribució uniforme i un rendiment acceptable.

Es van considerar llavors els algorismes de Cycle Reduntant Check (CRC), MD5 i SHA.
[RAONS]
CRC vs. MD5 vs. SHA: Si s'analitzen les diferents opcions es pot veure que MD5 i SHA són computacionalment més complexes que no pas CRC ja que MD5 i SHA pretenen ser més

segurs i utilitzen procediments molt més complexes.

Un cop escollit CRC, es va decidir fer servir la implementació que proveïa Boost. Concretament, es va fer servir la variant `crc_16 type`, perquè els 16 bits cabien dins de un `unsigned` (32 no éren necessaris, no aportaven una millora en els resultats significativa, i per tant proporcionava tenir millor rendiment, al ser el cost de càlcul més baix). És un typedef de `crc_optimal`, que òbviament es va fer servir per sobre de `crc_basic` (els dos generen el mateix output d'una manera transparent al desenvolupador, però el primer s'executa més ràpid).

2.3.2 Similitud per Minhash

La part essencial de la implementació d'aquest algoritme és la generació de les signatures de cada document. Un cop generades les signatures, simplement retornem la similitud entre els dos documents, és a dir, per als dos vectors de signatures V i W, quants coïncideixen ($V[i] = W[i]$) sobre la seva mida total (el número funcions de hash usades).

Per a calcular la signatura d'un vector, teòricament s'han de generar n permutacions de la matriu de representació i agafar el primer shingle que sigui present (tingui 1 a la matriu). La signatura és aquesta successió de shingles. Com s'explica al *paper*, aquesta aproximació no és pràctica, per tant s'ha optat per emprar hashing.

El hashing consisteix en aplicar una funció de hash a tots els elements presents al set i obtenir com a signatura el hash amb menor valor. A l'aplicar-se n funcions de hash, tenim una signatura de longitud n.

En aquest cas, era molt més pràctic optar per una funció de hash creada expressament per la pràctica, ja que necessitavem generar una funció que:

- Tingués n variants, ja que no és factible ni eficient usar n funcions de hash completament diferents.
- Fós ràpida de calcular.
- Mapegés un valor numèric a un altre valor dins del rang $[0 - UINT_MAX$.

La funció resultant va ser aquesta:

$$h(x) = (C1 * x + C2) \quad mod \quad UINT\_MAX$$

C1 i C2 són dues constants. [PER QUÈ FEM SERVIR DUES CONSTANTS?] Per a cada instància de `Comparator` es genera un vector de constants (bàsicament mitjançant una funció que omple un vector de mida n amb nombres aleatoris dins del rang dels enters de C++). És important notar que aquestes constants seran usades per a generar les signatures de tots els documents en una execució, ja que si fóssin variables, les sortides de la funció de hash no serien comparables.

D'altra banda, noti's que s'aplica a la funció de hash el mòdul amb la constant `UINT_MAX`. Aquesta és una constant proveïda per C++ que ens permet assegurar que el valor resultant de la funció es trobarà dins del rang dels `unsigned`.

Llavors, per a generar la signatura d'un `Document`, es crida un mètode del mateix, `get_signature`, amb els dos vectors de constants generats anteriorment i el número de funcions de hash (permutacions teòriques, efectivament la longitud del vector de signatura) que s'han d'aplicar per a obtenir la signatura. Una altra vegada, en una instància de `Comparator`, el número de funcions de hash ha de ser constant per a ser comparable (quant més alt és aquest factor arbitrari, més temps d'execució té l'algorisme, però de més precisió són els resultats).

El què emminentment aquesta funció és executar n vegades la funció de hash, cada vegada amb uns coeficients `C1[i]` i `C2[i]` per a cada element (shingle) present al conjunt que representa el document. Per a cada iteració, en guarda el valor menor (ho fem usant la funció `min` de C++). El cost de generar una signatura és doncs, O(n * |Document|).

2.3.3 LSH

La part essencial d'aquest algorísme és generar la matriu de b bandes i r files per banda, la qual permetrà centrar-se en els parells de documents que semblin ser similars, en comptes de calcular la similitud per Minhash de tots els parells de documents. Inicialment es computa una matriu que guarda totes les signatures Minhash dels documents per tal de no fer calculs innecessaris posteriorment. Cada vegada que es generen les sigantures d'un document en concret, es genera també la seva banda per tal d'agilitzar el algorísme.

Per obtenir la banda d'una signatura Minhash, es divideix la signatura en vectors de tamany ROWS (files per banda), on cada vector se li aplica una funció hash que donat un vector retorna un valor numeric. La funció hash utilitzada és la anomenada anteriorment com Hash combine de la llibreria boost que simplement fa el hash d'un vector. Aquesta funció d'obtenir les bandes retorna un vector amb el conjunt de bandes del document.

Un cop tenim la matriu de bandes, simplement es comparen totes les files de la matriu entre elles (cada fila correspon al vector de bandes d'un document) i ens guardem els elements comuns en els parells de documents comprovats. Aquest el dividim pel nombre total de bandes de la fila per obtenir un indicador de similitud. Però per tal de saber si aquest indicador és apte per aplicar la similitud de Minhash fem servir un valor threshold, que simplement aquest ens indicarà el mínim valor que ha de prendre el indicador per tal d'aplicar Minhash. El valor de threshold es calcula utilitzant la fórmula següent:

$$threshold = (1/bands)^{1/rows}$$

on bands és el nombre de bandes dels documents i rows el nombre de files per bandes. Si un indicador ens dóna més gran o igual que el threshold s'aplica la similitud de Minhash per aquell parell de documents, i en cas contrari el valor de similitud es deixa a 0 ja que voldrà dir que no són suficientment semblants i el seu valor s'aproxima a 0. En ambdos casos el resultat es guarda en una matriu de tamany (número de documents) (número de documents).
Un cop calculada la matriu de similitud amb LSH es retorna aquesta. Teóricament el cost de generar aquesta solució és menys costos que utilitzar la similitud per Minhash ja que aquí apliquem una mena de filtre que ens classifica els documents que aparentment tenen opcions de ser semblants utilitzant un nombre més petit de comparacions, el de bandes i files per banda.

2.4 Dificultats trobades

Un dels problemes que va sorgir durant la implementació és que tenia un comportament erràtic quan la longitud del document era menor a la k escollida per la llargada del shingles. Bàsicament això significa que s'intentava fer *substr* de longitud k en una cadena que no contenia suficients caràcters. Per això, es va decidir restringir els documents a tenir una longitud major que la de k. En qualsevol cas, es traca d'un cas extrem i la limitació s'ha posat

simplement per a controlar comportaments estranys.

Un altre problema que va sorgir va ser referent al algorísme Minhash. Després de realitzar diferents proves es va veure que la similitud de Minhash tardava molt més en executarse que no pas la similitud de Jaccard. Tot i veure que els resultats eren correctes obteniem un temps major al esperat en aquests casos. La solució que es va trobar va ser que hi havia una variable que es modificava de forma errònia i feia càlculs repetits que relentitzaven el algorísme.

Un altre problema que va sorgir per alguns membres del grup va ser la instalació i linkatge de la llibreria Boost en determinats sistemes operatius. Per exemple en Windows no es va aconseguir instal·lar-la i en el cas de Ubuntu problemes amb la detecció de variables d'entorn.

2.5 LSH amb Distància de Levenshtein

La distància de Levenshtein (també coneguda com distància d'edició) és una mesura freqüentment utilitzada en la comparació de textos i strings. Informalment, la distància de Levenshtein entre dos strings es defineix com el nombre mínim d'edicions d'un únic caràcter, és a dir, insercions, esborrats o substitucions.

Hi ha diverses formes de calcular la distància de Levenshtein, la més comuna és una que no utilitza cap contenidor per guardar les dades i fa càlculs repetits diverses vegades. En aquest cas s'ha utilitzat una versió més eficient evitant repetir càlculs innecessaris.

La distància de Levenshtein proporciona una mesura adequada de la similitud entre strings, doncs és evident que a una major semblança ortogràfica menys edicions caldrà fer per transformar un string en un altre. Ara bé, el cost del càlcul no és gens despreciable i per això es proposa calcular-la només d'aquells documents on hi ha un cert grau de certesa de la seva semblança. Això ens permet obtenir més informació de la similitud d'aquells documents que ja persuposem que són semblants. En canvi, dels documents que no presenten cap semblança no hi estem interessats i no és necessari calcular-ne la distància d'edició.

2.6 Generació de permutacions d'un document

Per la creació dels jocs de prova s'han generat permutacions de diferents documents. La generació de les permutacions d'ha fet de forma pseudo-aleatòria utilitzant elements pro-

porcionats per el llenguatge C++. Cal dir que per tal de generar les permutacions el més versemblants possibles les paraules s'han tractat com a objectes indivisibles, per tant mai s'ha generat una permutació que dividis una paraula en dos.

Així doncs, el primer pas per la generació de permutacions consisteix en la identificació de paraules del document. Després, es genera la permutació a partir de la llista de paraules comprovant que la permutació resultant no hagués estat generada anteriorment. Si ja ha estat generada, es descarta i se'n genera un altre fins a obtenir el nombre de permutacions desitjat.

3. Validació experimental

3.1 Descripció dels jocs de prova

Per tal de comparar els diferents algorismes d'identificació de similitud entre documents hem dissenyat 6 jocs de prova. L'objectiu dels jocs de prova no és només proporcionar una validació experimental de l'efectivitat dels algorismes, sinó també permetre'ns comprovar quins paràmetres són els que millor resultat donen amb aquests algorismes.

Cal destacar que per tal d'obtenir una mostra representativa del conjunt de documents que podem tractar amb el nostre sistema hem seleccionat jocs de prova amb diferents estils d'escriptura, doncs és evident que no és el mateix tractar un article periodístic que un text tècnic. D'aquesta manera, el funcionament i l'efectivitat dels algorismes no està condicionada a l'estil d'escriptura. Amb aquesta mateixa finalitat, els conjunts de prova triats no són tots del mateix idioma. Es presenten a continuació els 6 jocs de prova utilitzats:

- **Conjunt 1:** Permutacions generades a partir d'un article actual del diari "Financial Times" sobre la situació política dels Estats Units. En concret s'han generat 20 permutacions de 50 paraules cadascuna.
- **Conjunt 2:** Permutacions d'un fragment de l'obra de Ciceró. S'han generat 50 permutacions de 100 paraules cadascuna.
- **Conjunt 3:** Permutacions d'un article periodístic de política actual a l'estat espanyol. S'han generat 30 permutacions de 70 paraules.
- **Conjunt 4:** A partir d'un fragment de la wikipèdia sobre l'algorisme de Dijkstra, s'han generat variacions manualment a través de la reordenació de paràgrafs o la modificació d'alguns termes lèxicament equivalents amb l'objectiu de simular un exemple de possible còpia.

- **Conjunt 5:** 50 permutacions generades partir d'un fragment del llibre "Mining massive datasets".
- **Conjunt 6:** 20 permutacions d'un article d'un diari sobre la situació econòmica actual.

3.2 Similitud de Jaccard amb sets hashejats o sense hashejar

Per tal de veure si és més òptim realitzar la similitud de Jaccard amb sets de kshingles hashejats o sense es realitza el següent experiment. Els resultats obtinguts mostren que obtenim la similitud de Jaccard més ràpid utilitzant els kshingles hashejats. Això és degut a que el tamany dels kshingles és més llarg que el de kshingles hashejats. La gràfica mostra que per diferents conjunts de documents el temps d'execució de la similitud de Jaccard és sempre inferior utilitzant kshingles hashejats que no pas sense.

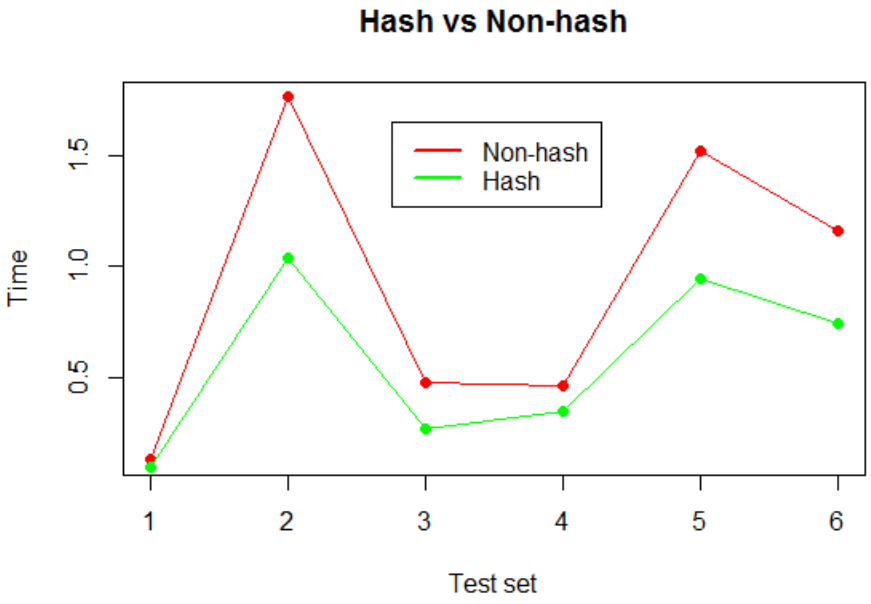

Figure 3.1: Diferència temporal entre Jaccard amb kshingles hashejats i sense hashejar

3.3 Tria del valor de k

La tria del valor de k fa referència a l'el·lecció de la mida dels *shingles* que utilitzen els diferents algorismes de manera més o menys directa. És una decisió important a considerar ja

que un valor de k incorrecte pot provocar l'aparença de similitud en documents no similars o simètricament, l'aparença de dissimilitud en documents similars. Qualsevol d'aquestes dues situacions provocaria un mal funcionament dels nostres algorismes, per tant l'establiment d'un valor de *k* és fonamental.

Per les raons comentades anteriorment, la tria del valor de *k* s'ha de fer considerant la llargada dels documents que tractarem. Tot i això, independentment de la llargada dels documents el valor de *k* ha de ser suficientment gran com per a que la probabilitat de que un *shingle* qualsevol aparegui en un document arbitrari sigui baixa.

Els jocs de prova utilitzats contenen entre 50 i 200 paraules. La llargada mitjana d'una paraula són 5 caràcters, els jocs de prova contenen entre 250 i 1000 caràcters. A més a més, dels caràcters de l'alfabet n'hi ha 20 que apareixen majoritàriament. Així doncs, 20^k és una estimació raonable del nombre de *shingles*. Per tant, el nombre de caràcters dels documents a comparar ha de ser petit en relació a 20^k; atenent a la longitud dels documents considerats, un valor de k = 6 donarà resultats satisfactoris. Tot i que s'experimenta amb diferents valors de k, s'utilitzarà aquest valor com una primera referència. Es mostra a continuació la similitud mitjana en funció del valor de k. Es pot observar que a mesura que el valor de k augmenta, la similitud decreix. Com s'ha mencionat anteriorment, s'utilitzarà el valor de k=6 ja que s'ha determinat que és el que ofereix les mesures més versemblants.

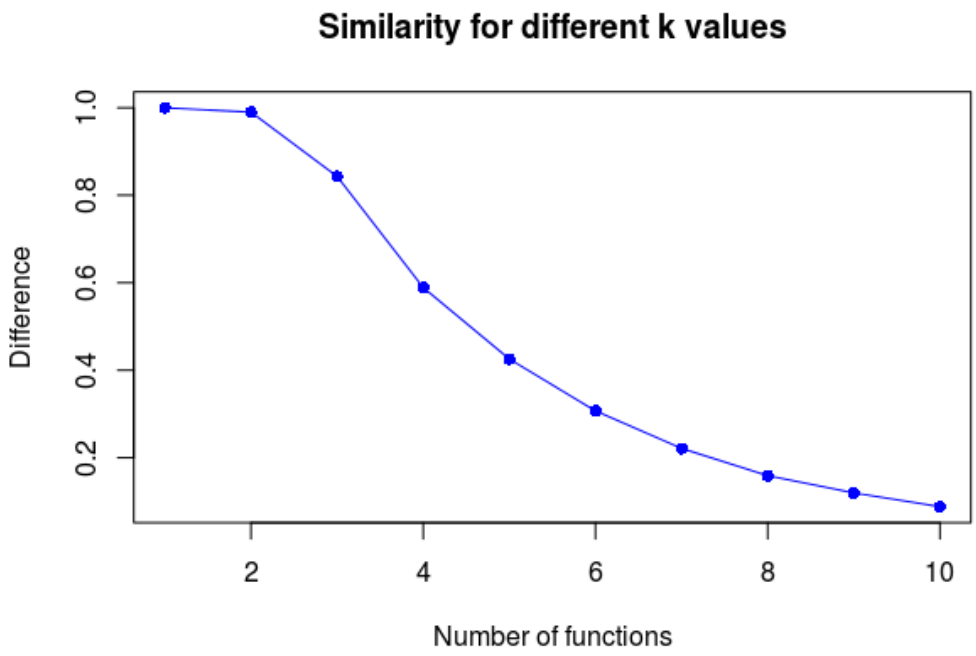

Figure 3.2: Similitud mitjana per a diferents valors de k.

Com era d'esperar, a mesura que la k augmenta la similitud entre documents disminueix. En aquest experiment s'ha obtingut la similitud mitjana per a tots els conjunts de documents. En el nostre cas valors extrems de k donarien resultats poc informatius. Pel que fa als valors mitjos tots són, a priori, igual de vàlids i s'escull k=6 que dona resultats satisfactoris.

3.4 Tria dels paràmetres en LSH

Per estudiar la influència del nombre de funcions de hash, el nombre de bandes i implícitament el nombre de fil·les per banda ja que bandes*fil·les = Nº de funcions, es considera a terme el següent experiment:

Es pren com a referència la similaritat obtinguda per Jaccard. És a dir, es considera la millor solució com aquella que s'acosti més a la similitud obtinguda per Jaccard. Per a cada conjunt de prova, es compara la similitud de Jaccard obtinguda amb la similitud obtinguda per LSH per diferents nombres de funcions de hash. A més a més, per a cada numero de funcions de hash es proven diferents combinacions de bandes i fil·les que després es promitjen. Les figures 3.1 a 3.6 mostren els resultats obtinguts per a cada conjunt de prova i la figura 3.7 el resultat de promitjar els resultats dels diferents conjunts de prova.

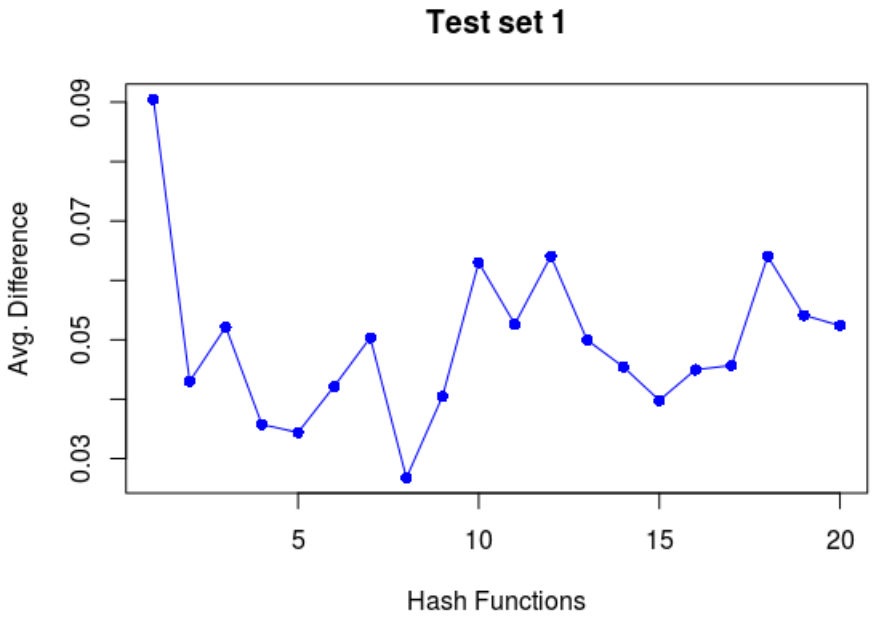

Figure 3.3: Diferència en el conjunt de prova 1 per diferent nombre de funcions de hash (en desenes).

Figure 3.4: Diferència en el conjunt de prova 2 per diferent nombre de funcions de hash (en desenes).

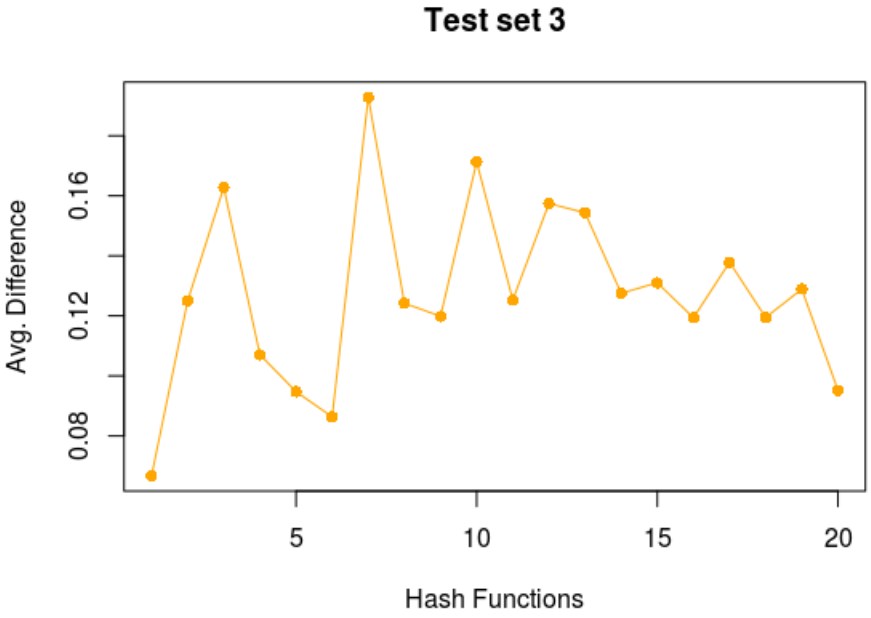

Figure 3.5: Diferència en el conjunt de prova 3 per diferent nombre de funcions de hash (en desenes).

A partir de la figura 3.7, es pot concloure que, en els conjunts de prova utilitzats, el nombre de funcions de hash que aproxima millor la similitud per LSH a la similitud de Jaccard és 50

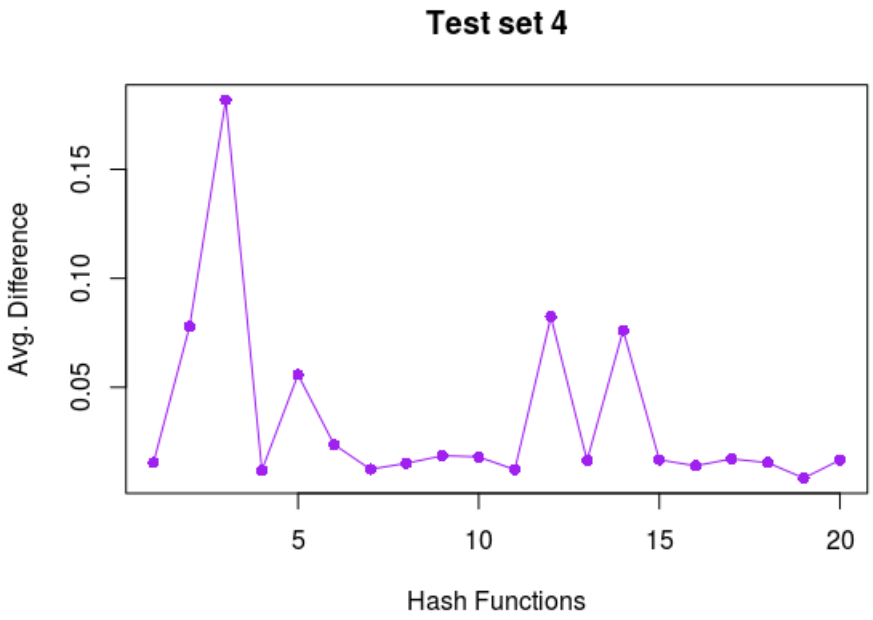

Figure 3.6: Diferència en el conjunt de prova 4 per diferent nombre de funcions de hash (en desenes).

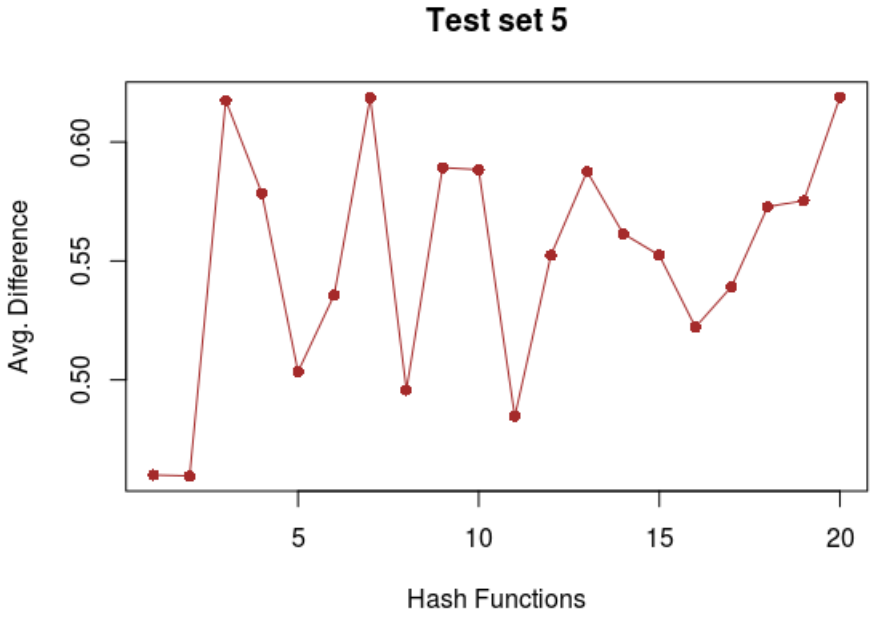

Figure 3.7: Diferència en el conjunt de prova 5 per diferent nombre de funcions de hash (en desenes).

(5 desenes). S'estudia a continuació quina és la distribució de bandes i fil·les que optimitza la similitud quan el nombre de funcions és 50.

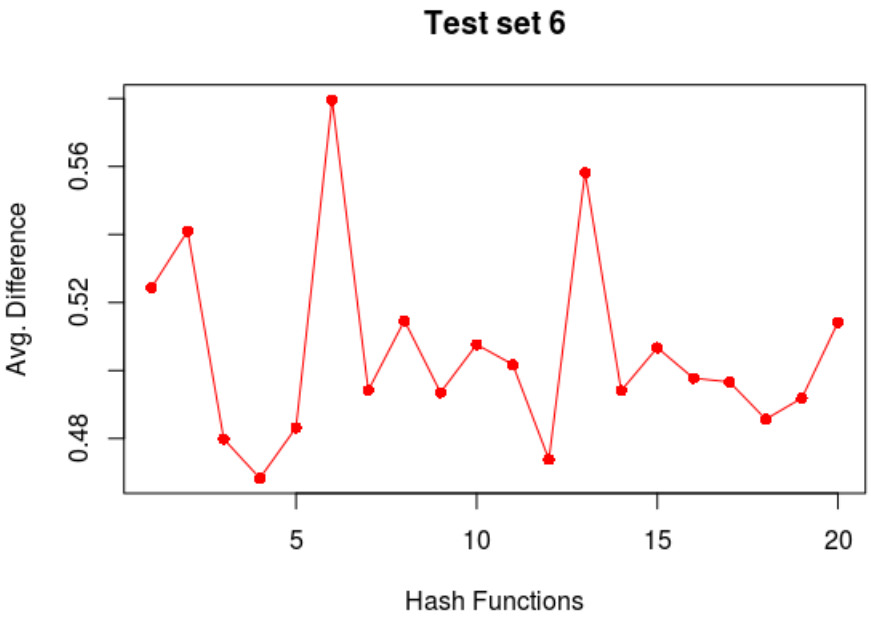

Figure 3.8: Diferència en el conjunt de prova 6 per diferent nombre de funcions de hash (en desenes).

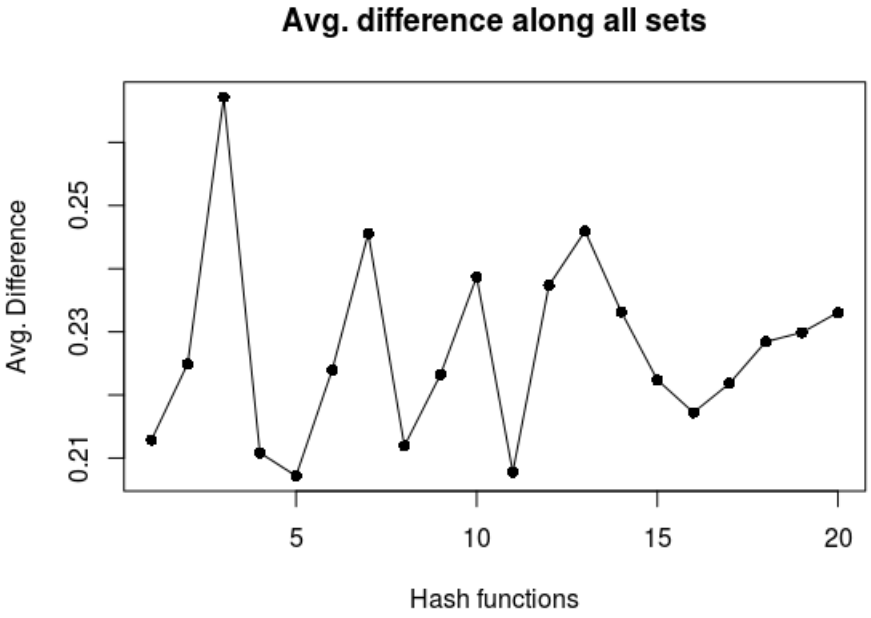

Figure 3.9: Diferència mitjana en tots els conjunt de prova per diferent nombre de funcions de hash (en desenes).

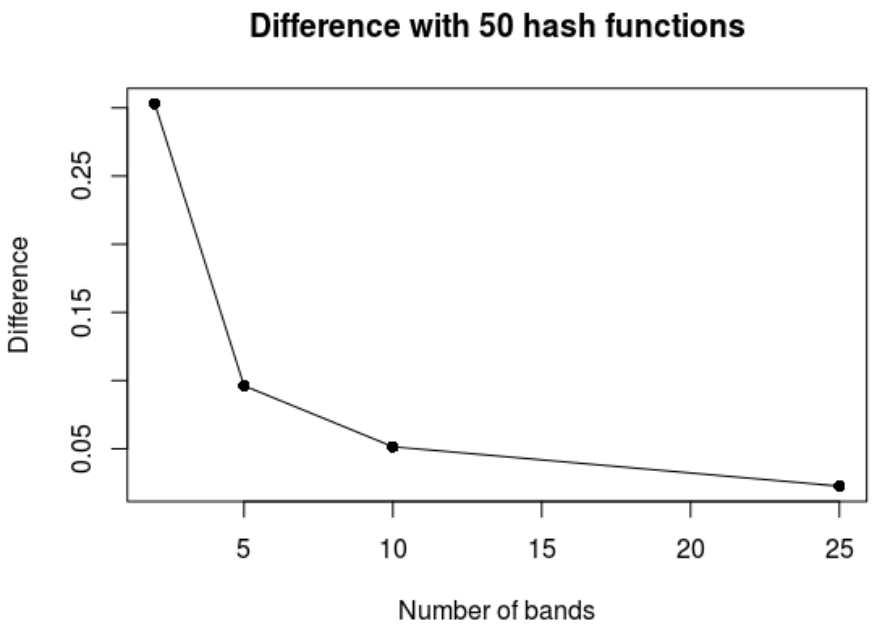

Figure 3.10: Execució de LSH amb 50 funcions de hash per a diferents valors de bandes

Un cop determinat que el nombre de funcions de hash que millor resultat dona és 50, queda per veure quina és la millor distribució de bandes i fil·les. Per veure-ho es dissenya un experiment similar a l'anterior: s'executen els diferents jocs de prova i per a cadascun es proven diferents combinacions de bandes i fil·les i es calcula la diferència del resultat obtingut amb el de la similitud de Jaccard. El resultat òptim s'obté quan s'utilitzen 25 bandes de 2 fil·les cadascuna. A més a més, la diferència promig amb la similitud de Jaccard és de 0.03.

S'estudia ara la solució desde el punt de vista temporal. Per fer-ho es compara la diferència de temps entre Jaccard i LSH (a més diferència millor) per a cadascun dels conjunts de prova i es promitja la diferència. A continuació es mostren els resultats obtinguts.

Es pot observar com el valor amb una major diferència mitjana és el 2, que correspon a 20 funcions de hash. Atenent únicament a la duració del càlcul, aquest és més ràpid quan s'utilitzen 20 funcions de hash.

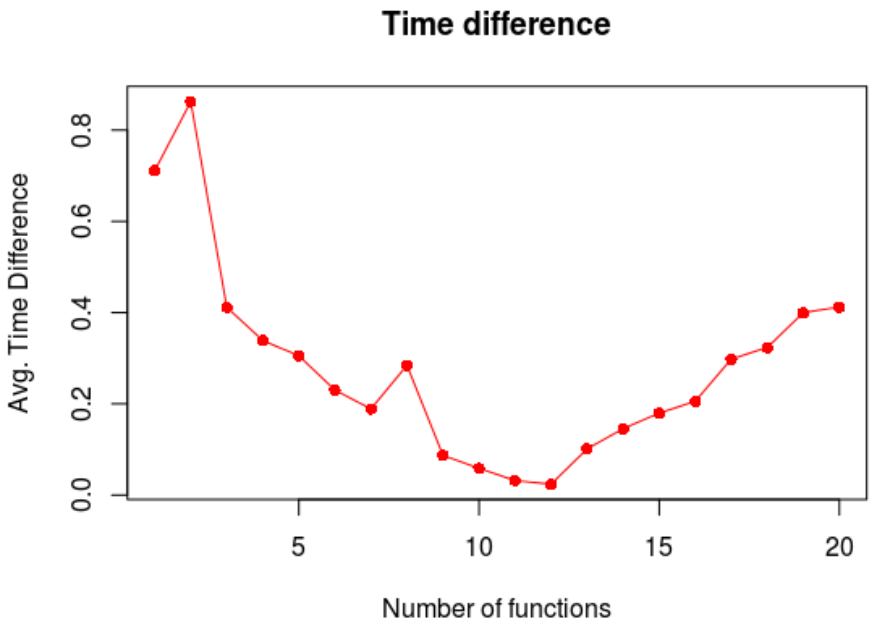

Figure 3.11: Diferència temporal entre Jaccard i LSH per a diferents valors de bandes

3.4.1 Qualitat de les solucions

Per evaluar la qualitat global de les solucions es té en compte tant el temps com la precisió, que són els dos factors més interessants i amb aplicació més directa. Per fer-ho, es combinen tant la diferència temporal com la diferència en precisió (figura 3.10) i s'observa el resultat. S'observa que la diferència de precisió no és gaire significativa globalment. Ara bé, la diferència temporal si que ho és de forma més marcada.

Pel que fa a la precisió, s'obtenien resultats òptims quan el nombre de funcions era 50. Nogensmenys, al considerar el temps, el nombre òptim de funcions és 20. Per tant, depenent del que es pretengui optimitzar caldrà utilitzar una configuració o un altra.

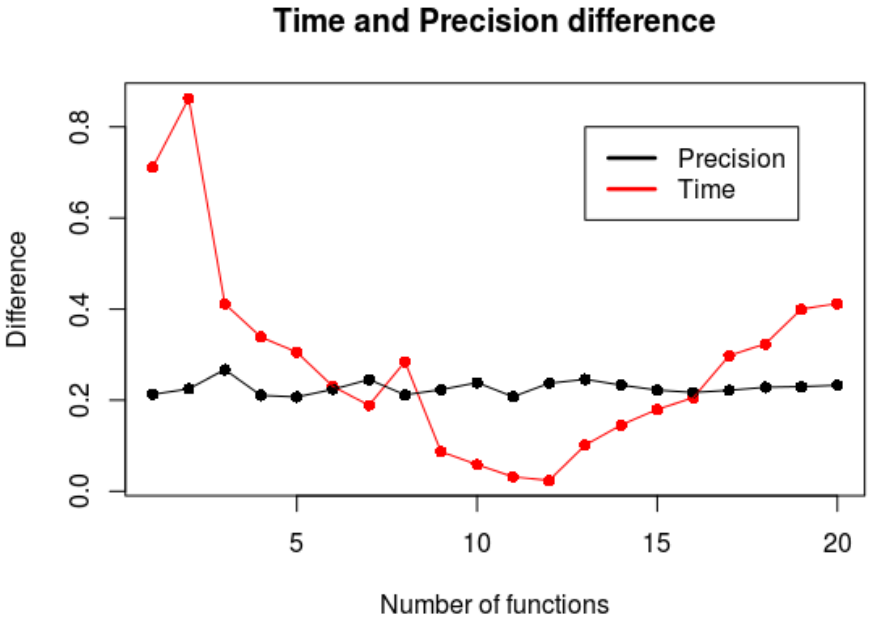

Figure 3.12: Diferència temporal entre Jaccard i LSH per a diferents valors de bandes

4. Conclusions

Les conclusions d'aquesta pràctica són diverses en naturalesa. L'objectiu de la pràctica era la implementació, l'estudi i comparativa de diversos algorismes per a la detecció de similituds en documents. Concretament se n'han implementat tres: similitud per Jaccard, per minhash i per LSH. Endemés, s'ha implementat una combinació de LSH i distància de Levenshtein que ha permès l'aprofundiment conceptual. Finalment, s'ha estudiat de forma experimental l'efecte que tenen, tant en el rendiment qualitatiu i quantitatiu, els diferents paràmetres que defineixen cadascun dels algorismes.

Concretament, s'ha justificat de forma experimental la utilització dels algorismes de minhashing i LSH (Locality Sensitive Hashing) ja que aporten un augment de l'eficiència significatiu. Pel cas de la similitud de Jaccard, s'ha estudiat l'efecte que el valor de k té en el rendiment qualitatiu de l'algorisme. S'ha conclós que valors extrems donaran un mal resultat i, dins dels valors intermitjos cal escollir el més adequat per a la longitud dels documents que es vol tractar. Endemés, s'ha estudiat l'efecte que tenen el nombre de funcions i la divisió en bandes i fil·les en el rendiment de l'algorisme tot i determinant els paràmetres que l'optimitzen per al conjunt de proves creat.

Finalment, s'ha realitzat una ampliació de l'algorisme LSH on s'ha combinat amb la implementació del càlcul de la distància de Levenshtein. S'ha conclós que tot i que el càlcul de la distància d'edició és costós, aporta una mesura força interessant del grau de similitud entre documents.