Report Progetto Algoritmi per i Big Data

Progetto Scelto:
Num 1, Tutto il mondo è LSH

Studenti:

Matteo Di Gioacchino
Email:
matteo.digioacchino@alumni.uniroma2.eu
Matricola: 0350191

Maurizio Fiusco
Email:
maurizio.fiusco@alumni.uniroma2.eu
Matricola: 0350127

Indice

Introduzione

Si è deciso di implementare la pipeline di lavoro per il riconoscimento di documenti simili, in particolar modo, la pipeline è suddivisa in 3 macrosteps:

1) K-shingling dei documenti e relativa costruzione della matrice binaria degli shingles.
2) Costruzione degli sketches per ogni documento utilizzando la tecnica del MinHashing tramite l'applicazione di un numero "h" di funzioni Hash al fine di ridurre notevolmente la dimensione della matrice di shingling.
3) Dato un certo parametro "s", rappresentante una Jaccard Similarity desiderata, applicazione della tecnica dell'LSH (Locality Sensitive Hashing) al fine di ridurre notevolmente il numero di confronti effettuato.
4) Analisi dei risultati ottenuti tramite grafici, osservazioni e confronto tra calcoli teorici e risultati ottenuti. Inoltre, il codice è stato eseguito su due macchine dalla potenza computazionale diversa per poterne valutare l'efficienza ed i tempi di esecuzione.

Dataset scelto

Come dataset sono stati scelti gli abstract di diversi articoli che trattano argomenti di carattere biologico, e le loro rielaborazioni tramite il comando "rearrange" dato a chatGPT. L'obiettivo era vedere se fossimo riusciti a individuare nuovamente le coppie abstract-rielaborazione.

Abbiamo complicato il tutto inserendo articoli "simili" tra loro, ovvero:

- 7 articoli che trattano di predizione di risposta farmacologica ai tumori, di riviste, gruppi e autori uguali (potrebbero esserci stili di scrittura simili) e tecniche di apprendimento automatico e non.
- 8 articoli relativi a TFEB (fattore di trascrizione), tutti a firma dello stesso group leader
- 5 articoli sul rene policistico autosomico dominante, 4 dei quali trovati nei simili al primo dall'algoritmo di PubMed
- A questi si aggiungono altri articoli spuria, alcuni facenti capo ai medesimi gruppi dei precedenti

Implementazione:

Linguaggio di programmazione utilizzato

Come linguaggio di programmazione utilizzato per l'implementazione è stato scelto Python a causa della sua facilità di utilizzo e vastità di librerie utili per la risoluzione dei problemi implementativi riscontrati (dei quali si tratterà in una sezione a parte). Inoltre, Python rende più facile suddividere il codice in moduli separati, indipendenti tra di loro e facilmente esportabili, in questo modo, è stato possibile modificare, valutare ed analizzare ogni step della pipeline in modo semplice ed efficiente.
Infine, le implementazioni di alcune strutture dati fornite da Python si sono rivelate fondamentali per la risoluzione di alcune problematiche emerse durante l'implementazione di ogni step della pipeline.

Primo Step della Pipeline: K-Shingling

Librerie utilizzate

Per questo step della Pipeline, le liberie utilizzate sono:

LIBRERIA	MOTIVAZIONE
os	Ricerca ed utilizzo della directory contenente i files ".pdf" utilizzati come dataset.
pypdf	Merging dei files utilizzati come dataset al fine di facilitare la costruzione dell'insieme dell'insieme di tutti gli shingles.
PyPDF2	Lettura ed analisi del testo di ogni documento utilizzato nel dataset.
string	Doublecheck sui singoli shingles prima di inserirli nell'insieme degli shingles totali
primepy	Scelta di un numero primo da utilizzare come size della tabella degli shingles al fine di poter aggiornare la presenza o meno di uno shingle tramite una funzione di hash deterministica.
bitarray	Ridurre al minimo lo spazio utilizzato da ogni colonna della tabella degli shingles.

Problematiche riscontratte e soluzioni

Caratteri o parole

La prima problematica affrontata è stata il determinare se lo shingling dovesse essere effettuato sui singoli caratteri dei documenti o sulle parole; dopo una ricerca su internet, è stato osservato che nella maggior parte delle implementazioni di questa pipeline vengono scelte le parole, e non i singoli caratteri, come tokens per lo shingling, quindi, anche in questa implementazione verrà utilizzata la stessa logica.

Dimensione della matrice binaria per gli shingles

Per determinare la dimensione della matrice binaria per gli shingles e quindi il numero di shingle distinti si è usata la struttura dati **Set()**.

Set è una struttura dati in Python gestita come una HashTable senza ripetizione di elementi, in questo modo, riesce a garantire tempi di inserimenti *medi* nell'ordine di **O(1)**. Dunque, per determinare la size della tabella di shingling, è stato inizializzato un Set() inizialmente vuoto, dopodiché, è stato effettuato il merge di ogni documento del dataset in un documento a parte (quindi senza modificare i files originali) al fine di facilitarne la loro analisi, è stato effettuato il K-Shingling di ogni pagina del documento ed infine è stato inserito ogni shingle all'interno del Set di tutti gli shingles. Naturalmente, anche se ogni inserimento ha tempo di esecuzione *medio* nell'ordine di *O(1)*, ne vengono effettuati un numero proporzionale al numero di shingles complessivi, ripetuti e non. Da qui, ricavare la dimensione della tabella di Shingling è stato facile: molte strutture dati in Python, tra cui Set(), mantengono un puntatore ad un valore intero corrispondente alla loro dimensione, il quale viene aggiornato ogni qual volta si inserisce o si cancella un nuovo elemento, perciò, ricavarne la dimensione è un operazione effettuabile in tempo **O(1)**. Questa sarà la dimensione di una singola colonna (corrispondente ad un singolo documento) della matrice di shingling, quindi andrà moltiplicata per il numero di documenti totali al fine di determinare la dimensione complessiva della nostra tabella di shingling.

Aggiuntivamente a ciò abbiamo elaborato la tesi di Matteo per utilizzare Flajot-Martin per contare il numero di shingles distinti in modo da fornire un approccio differente nel momento in cui il numero di documenti e conseguentemente di shingle fosse stato più elevato che in questo caso.

Memoria occupata dalla matrice di shingling

Poiché un intero in Python viene trattato come un vero e proprio oggetto, oltre al numero di bits utilizzati viene inserito un ulteriore overhead molto grande alla quantità di memoria utilizzata per rappresentarlo, in particolare modo, utilizzando la funzione *getsizeof()* della libreria *sys*, è possibile osservare come la memorizzazione di un valore intero *"1"* o *"0"* in Python utilizzi ***28 bytes = 224 bits***; dunque, utilizzare una lista in Python contenente *"1"* o *"0"* per indicare la presenza o meno di uno shingle avrebbe portato ad un numero di bit complessivo pari a: ***224*|colonna|*|documenti|*** **bits**, un numero fin troppo grande. Si è quindi pensato di utilizzare una matrice contenente variabili booleane *"True"* o *"False",* ma anche esse in Python vengono gestite come *"1"* o *"0",* tornando quindi alla problematica precedente. E' stata quindi utilizzata una struttura dati particolare, il *"bitarray",* che riduce drasticamente il numero di bytes utilizzati, in particolar modo, costruendo un bitarray di size *7853* (numero primo utilizzato come dimensione di una singola colonna della tabella di shingling), si ottengono i seguenti risultati:

```
Workspace > sizes.py > ...
1  from bitarray import bitarray
2  import sys
3
4  Matrix = [[],[],[]]
5  N = 7853 # numero primo utilizzato come lunghezza di una singola colonna della matrice di shingling
6  Matrix[0].append(bitarray('0'*N))
7  for _ in range (0,N):
8      Matrix[1].append(1)
9      Matrix[2].append(True)
10 print(f"Numero di bytes utilizzati per un singolo valore intero in Python: {sys.getsizeof(1)}")
11 print(f"Numero di bytes utilizzati per il 'bitarray': {sys.getsizeof(Matrix[0])}")
12 print(f"Numero di bytes utilizzati per una lista di interi: {sys.getsizeof(Matrix[1])}")
13 print(f"Numero di bytes utilizzati per una lista di variabili booleane: {sys.getsizeof(Matrix[2])}")

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\Users\matte\Workspace> & C:/Python312/python.exe c:/Users/matte/Workspace/sizes.py
Numero di bytes utilizzati per un singolo valore intero in Python: 28
Numero di bytes utilizzati per il 'bitarray': 88
Numero di bytes utilizzati per una lista di interi: 67224
Numero di bytes utilizzati per una lista di variabili booleane: 67224
PS C:\Users\matte\Workspace>
```

Quindi, in totale utilizziamo ***88 bytes = 704 bits*** per ogni colonna (e quindi per ogni documento), abbiamo 48 documenti, per un totale di ***704*48 = 33792 bits.***

Secondo Step della Pipeline: MinHashing

Librerie utilizzate

Per questo step della Pipeline, le liberie utilizzate sono:

LIBRERIA	**MOTIVAZIONE**
random	Generazione di due numeri, "a" e "b", compresi tra 1 ed N (dove N è la size di una colonna della tabella di shingling, ossia 7583), al fine di poter generare una famiglia di funzioni Hash del tipo "(ax+b) mod N".

Problematiche riscontratte e soluzioni

Generazione di una famiglia di funzioni Hash

Come già detto nelle librerie utilizzate, la famiglia di funzioni Hash sfruttata per generare gli sketches di ogni colonna della matrice di shingling è una famiglia di funzioni Hash del tipo:

$$(ax + b)\ mod\ N$$

Per generarla, ogni qual volta si voleva calcolare un nuovo sketch di ogni documento, per tutti i documenti (quindi applicando UNA funzione Hash a TUTTI i documenti ad ogni step, dove il numero di step è pari al numero di funzioni hash totali), si sceglievano due numeri, "a" e "b", compresi tra 1 ed N (dove N è la size di una colonna della tabella di shingling, ossia 7583), e si eseguivano le necessarie operazioni.

Trasposizione della matrice per prepararla all'LSH

Al fine di preparare meglio la matrice per prepararla all'LSH, è stato necessario trasporre la matrice delle signature finale, in modo tale da rendere più facile la sua successiva divisione in bande e righe.

Terzo Step della Pipeline: Locality Sensitive Hashing

Librerie utilizzate

Per questo step della Pipeline, le liberie utilizzate sono:

LIBRERIA	**MOTIVAZIONE**
copy	Copia della matrice di MinHashing al fine di suddividerla successivamente in bande e righe per poi effettuarne il Locality Sensitive Hashing.

Problematiche riscontratte e soluzioni

Scelta di "b" ed "r" per l'LSH

Ci si è trovati nella condizione di dover scegliere due valori interi, *"b"* e *"r"*, per determinare il numero di bande (*"b"*), composte ognuna da righe (*"r"*). Inoltre, per effettuare l'LSH, serve anche un valore di input *"s"* indicante la Threshold di Jaccard Similarity desiderata. In definitiva, serviva la costruzione di una funzione che, presi in input *"n"* (equivalente al numero di funzioni hash utilizzate durante il MinHashing) ed *"s"*, restituisse *"b"* e *"r"* tali che:

$$\begin{cases} \boldsymbol{n = b * r} \\ \boldsymbol{s \sim \left(\frac{1}{b}\right)^{\frac{1}{r}}} \end{cases}$$

La seconda equazione è stata estratta da calcoli per minimizzare la quantità di falsi negativi seguendo la seguenti equazioni:

$$\Pr\{TUTTE\ le\ righe\ in\ una\ banda\ sono\ uguali\} = s^r$$

$$\Pr\{\exists\ righe\ diverse\ in\ una\ certa\ banda\} = (1 - s^r)$$

$$\Pr\{\nexists\ righe\ uguali\} = (1 - s^r)^b$$

Scelta del numero di Buckets C per l'LSH

Per rendere il più uniforme possibile la distribuzione delle bande all'interno della matrice di LSH (e di conseguenza ridurre il numero di falsi positivi), è stato scelto un numero di buckets *"C"* pari a *1000*, poiché la scelta di *"C"* doveva essere tale che *C >> r*, e dalle prove effettuate *"r"* non andava mai oltre *10* o *25*.

Scelta di una funzione di hash per l'LSH e valore sul quale effettuare l'hashing

La funzione di hash scelta è una semplice funzione del tipo:

$$h(x) = \big(hash(x)\big)\ mod\ C$$

Dove *hash(x)* è la funzione *hash()* propria di Python, che mappa un numero intero in un valore di hash compreso fra $\{0, 2^{32}\}$.
Per determinare quale fosse il valore di *x* sul quale effettuare l'hashing, serviva un valore che rappresentasse tutti gli sketches contenuti all'interno delle *r* righe di una banda, per far ciò, sono stati concatenati tutti i valori degli sketches di una singola banda, creando così un numero intero sul quale effettuare l'hashing.

Per esempio:
Sketches in una banda = [15, 234, 120, 78, 17]
x = 152341207817

Link alla Repository di GitHub

https://github.com/MatteoDiggioacca/DocSimilarityImplementation/tree/main/DocumentSimilarity

Come utilizzare il Software

Per eseguire il Software, si veda il file *"test.py"* contenuto nella Repo di cui sopra. I parametri da modificare per eseguire il Software in maniera propria sono i seguenti:

PARAMETRO	**PORZIONE DEL FILE**	**FUNZIONALITA'**
n_docs	IMPORTS	Indica il numero di documenti utilizzati
k	SHINGLING	Indica la size di ogni singolo shingle.
dir_path	SHINGLING	Indica il path della directory che contiene i file da utilizzare come dataset.
result_file	SHINGLING	Indica il nome del file di merge dei documenti del dataset (si guardi la porzione del report riguardante il Primo Step).
result_file_path	SHINGLING	Indica il path del nome del file di merge di cui sopra
num_hashes	MIN_HASHING	Indica il numero di funzioni di hash da utilizzare per il MinHashing
treshold	LSH	Indica la Jaccard Similarity desiderata per l'LSH.

I file del dataset vanno nominati *"1.pdf", "2.pdf", ... , "n_docs.pdf"*, all'interno della directory "coppie" si potranno trovare anche dei file nominati *"copia_1.pdf", "copia_2.pdf"*, ... , "copia_n_docs.pdf" sono le rielaborazioni effettuate con chatGPT.

Alcune idee provenienti dalla biologia computazione.

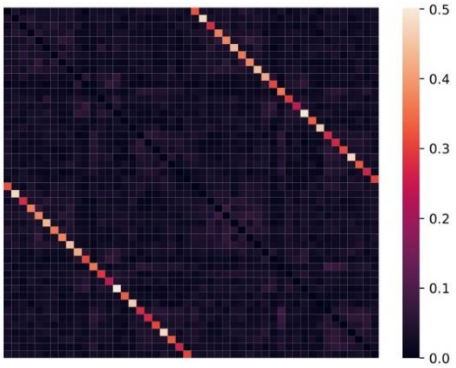
Il k-shingle funziona?

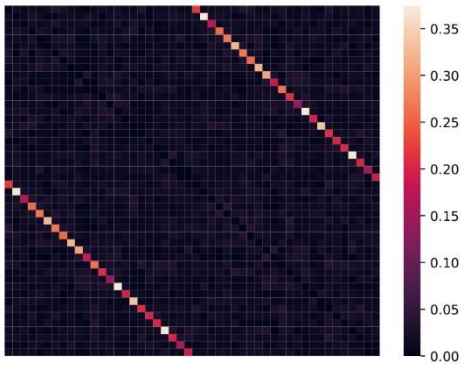
Per verificare che il k-shingle funzionasse e che tramite esso riuscissimo a distinguere effettivamente i documenti ci siamo avvalsi di due strumenti.

Heatmap

Le heatmap sono una rappresentazione grafica dei dati dove i singoli valori contenuti in una matrice sono rappresentati da colori. Risultano essere particolarmente comode quando si vogliono rappresentare graficamente grandi dataset, visualizzare e riassumere risultati di diverso tipo.

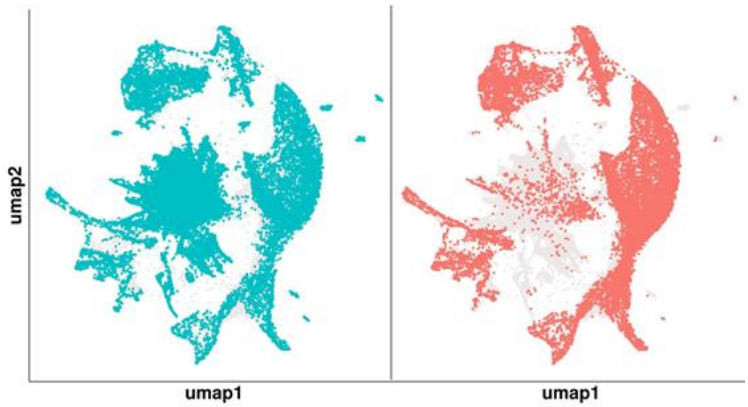
Calcolate le jaccard, non è facile accorgersi visivamente se ci fosse differenza nei valori tra documenti simili e distinti e quanto questa differenza fosse accentuata. In questo modo è possibile farlo graficamente. Sulle righe e sulle colonne abbiamo i documenti, all'intersezione tramite il colore è rappresentata la similarità della coppia. La diagonale che va da sinistra a destra rappresenta la jaccard del documento con se stesso, il suo valore è 1 ma nel grafico risulta 0 altrimenti avrebbe reso con la scala di colore, meno accentuate le differenze tra le coppie significative. Come si può vedere la distinzione è netta tra gli articoli simili e quelli distinti a livello di tonalità. Si riportano i risultati con k = 2 e k = 3



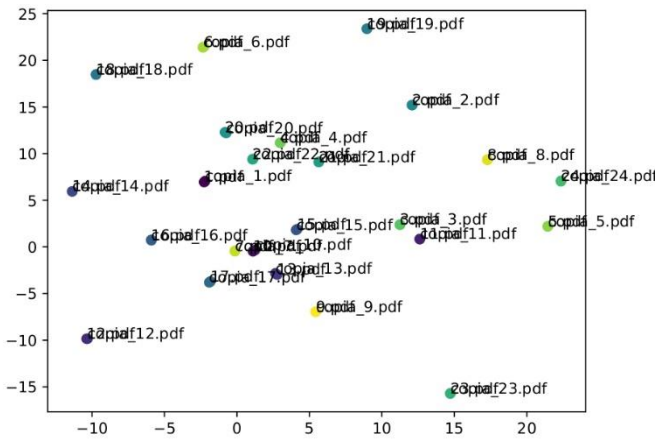


Umap

Si tratta di una tecnica per la riduzione di dimensionalità non lineare. L'obiettivo è la trasformazione dei dati da uno spazio di dimensione più alta a uno di dimensione inferiore, in modo che questa rappresentazione mantenga alcune proprietà significative dei dati di origine. Molto utile quando si hanno matrici particolarmente sparse che sono computazionalmente sconvenienti, situazione tipica dei dati di sequenziamento a singola cellula. Solitamente le matrici sono molto grandi e vedono per ciascuna cellula il livello di espressione di ciascun gene. Parliamo di migliaia di righe e di colonne che rendono difficile il confronto e la scoperta di relazioni tra le varie cellule a livello visivo, per tale ragione, è applicata anche per la visualizzazione delle cellule in uno spazio a due o tre dimensioni, con ottimi risultati in termini di consevazione dell'informazione. In seguito due esempi di risultato tratti da Identification of Early Transcriptional Markers of Autosomal Dominant Polycystic Kidney Disease Cystic Epithelial Cells



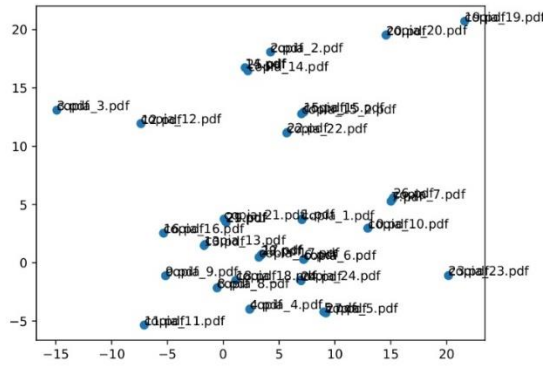
L'idea è la stessa: rappresentare i documenti su due dimensioni, naturalmente settando appositamente I parametri e verificare se le coppie riescono a essere individuate. E così è:



Doc-similarity tramite riduzione di dimensionalità

Ma se la riduzione di dimensionalità funziona così bene, perché non usarla per la doc similarity? Così abbiamo preso le coordinate calcolate di ciascun punto e ristretto il confronto ai documenti entro una certa distanza. Con risultati a dir poco ottimi a livello di qualità, anche aggiungendo documenti senza la loro rielaborazione con chatGPT (singletons) e documenti con due rielaborazioni anziché una.

Con matrici di dimensioni superiori i tempi possono essere ridotti con una riduzione di dimensionalità preventiva di tipo lineare, solitamente si usa la PCA.


20
15
10
5
0
−5
−15
−10
−5
0
5
10
15
20

Risultati

L'esecuzione è stata testata sui nostri pc personali (Matteo, Maurizio). Per problemi di compatibilità di versioni, l'elaborazione con umap è stata effettuata solo su il pc con prestazioni inferiori. Come metodo di paragone si possono prendere le versioni con 150 e 250 funzioni hash, effettuate su entrambe le macchine.

Algoritmo classico, k = 3, 48 documenti (24 coppie)

LEGENDA
b: numero di bande in cui è stata suddivisa la tabella di MinHash **r:** numero di righe che compongono ogni banda **s:** Jaccard's Similarity desiderata **num_hashes:** numero di funzioni Hash utilizzate

150 FUNZIONI HASH PER IL MIN-HASHING

b: 30, r: 5, s: 0.4, num_hashes: 150

Numero di confronti prima di LSH: 1176
Numero di confronti dopo LSH: 816
Tempo di esecuzione Shingling: 0.9980 - 4.5353 secondi
Tempo di esecuzione MinHashing: 1.6201- 2.6198 secondi
Tempo di esecuzione LSH: 0.0043 - 0.5686 secondi
Tempo di esecuzione totale: 2.6224 - 7.7237 secondi
Numero di Falsi negativi: 0

b: 25, r: 6, s: 0.6, num_hashes: 150

Numero di confronti prima di LSH: 1176
Numero di confronti dopo LSH: 699
Tempo di esecuzione Shingling: 0.9588 – 3.7124 secondi
Tempo di esecuzione MinHashing: 1.5879 – 3.1189 secondi
Tempo di esecuzione LSH: 0.0042 – 0.5889 secondi
Tempo di esecuzione totale: 2.5510 – 7.4202 secondi
Numero di Falsi negativi: 2

b: 15, r: 10, s: 0.8, num_hashes: 150

Numero di confronti prima di LSH: 1176
Numero di confronti dopo LSH: 254
Tempo di esecuzione Shingling: 0.9553 - 3.5949 secondi
Tempo di esecuzione MinHashing: 1.5620 - 2.5727 secondi
Tempo di esecuzione LSH: 0.0039 - 0.6123 secondi
Tempo di esecuzione totale: 2.5212 - 6.7799 secondi
Numero di Falsi negativi: 13

b: 6, r: 25, s: 0.9, num_hashes: 150

Numero di confronti prima di LSH: 1176
Numero di confronti dopo LSH: 34
Tempo di esecuzione Shingling: 0.9664 – 3.7211 secondi
Tempo di esecuzione MinHashing: 1.5949 – 2.6091 secondi
Tempo di esecuzione LSH: 0.0038 – 0.5625 secondi
Tempo di esecuzione totale: 2.5650 – 6.8927 secondi
Numero di Falsi negativi: 23

250 FUNZIONI HASH PER IL MIN-HASHING

b: 50, r: 5, s: 0.4, num_hashes: 250

Numero di confronti prima di LSH: 1176
Numero di confronti dopo LSH: 1095
Tempo di esecuzione Shingling: 1.2161 – 3.7309 secondi
Tempo di esecuzione MinHashing: 2.6420 – 4.1896 secondi
Tempo di esecuzione LSH: 0.0064 – 0.5817 secondi
Tempo di esecuzione totale: 3.8645 – 8.5023 secondi
Numero di Falsi negativi: 0

b: 25, r: 10, s: 0.6, num_hashes: 250

Numero di confronti prima di LSH: 1176
Numero di confronti dopo LSH: 572
Tempo di esecuzione Shingling: 0.9730 – 4.0044 secondi
Tempo di esecuzione MinHashing: 2.6346 – 4.3798 secondi
Tempo di esecuzione LSH: 0.0051 – 0.6747 secondi
Tempo di esecuzione totale: 3.6126 – 9.0589 secondi
Numero di Falsi negativi: 4

b: 25, r: 10, s: 0.8, num_hashes: 250

Numero di confronti prima di LSH: 1176
Numero di confronti dopo LSH: 572
Tempo di esecuzione Shingling: 0.9691 – 3.7705 secondi
Tempo di esecuzione MinHashing: 2.6847 – 4.1466 secondi
Tempo di esecuzione LSH: 0.0056 – 0.6097 secondi
Tempo di esecuzione totale: 3.6593 – 8.5268 secondi
Numero di Falsi negativi: 8

b: 10, r: 25, s: 0.9, num_hashes: 250

Numero di confronti prima di LSH: 1176
Numero di confronti dopo LSH: 108
Tempo di esecuzione Shingling: 0.9678 – 3.6838 secondi
Tempo di esecuzione MinHashing: 2.7050 – 4.5428 secondi
Tempo di esecuzione LSH: 0.0046 – 0.5882 secondi
Tempo di esecuzione totale: 3.6774 – 8.8148 secondi
Numero di Falsi negativi: 19

500 FUNZIONI HASH PER IL MIN-HASHING

b: 100, r: 5, s: 0.4, num_hashes: 500

Numero di confronti prima di LSH: 1176
Numero di confronti dopo LSH: 1128
Tempo di esecuzione Shingling: 0.9507 secondi
Tempo di esecuzione MinHashing: 5.3321 secondi
Tempo di esecuzione LSH: 0.0111 secondi
Tempo di esecuzione totale: 6.2940 secondi
Numero di Falsi negativi: 0

b: 50, r: 10, s: 0.6, num_hashes: 500

Numero di confronti prima di LSH: 1176
Numero di confronti dopo LSH: 1067
Tempo di esecuzione Shingling: 0.9786 secondi
Tempo di esecuzione MinHashing: 5.4540 secondi
Tempo di esecuzione LSH: 0.0090 secondi
Tempo di esecuzione totale: 6.4416 secondi
Numero di Falsi negativi: 2

b: 25, r: 20, s: 0.8, num_hashes: 500

Numero di confronti prima di LSH: 1176
Numero di confronti dopo LSH: 518
Tempo di esecuzione Shingling: 0.9713 secondi
Tempo di esecuzione MinHashing: 5.2471 secondi
Tempo di esecuzione LSH: 0.0075 secondi
Tempo di esecuzione totale: 6.2259 secondi
Numero di Falsi negativi: 8

b: 20, r: 25, s: 0.9, num_hashes: 500

Numero di confronti prima di LSH: 1176
Numero di confronti dopo LSH: 378
Tempo di esecuzione Shingling: 0.9593 secondi
Tempo di esecuzione MinHashing: 5.6026 secondi
Tempo di esecuzione LSH: 0.0074 secondi
Tempo di esecuzione totale: 6.5693 secondi
Numero di Falsi negativi: 19

3 ESECUZIONI PARTICOLARI:
Numero di Buckets utilizzati per LSH: 10.000

b: 100, r: 10, s: 0.6, num_hashes: 1000

Numero di confronti prima di LSH: 1176
Numero di confronti dopo LSH: 1648
Tempo di esecuzione Shingling: 0.9829 secondi
Tempo di esecuzione MinHashing: 10.8691 secondi
Tempo di esecuzione LSH: 0.0273 secondi
Tempo di esecuzione totale: 11.8794 secondi
Numero di Falsi negativi: 0

b: 100, r: 10, s: 0.7, num_hashes: 1000

Numero di confronti prima di LSH: 1176
Numero di confronti dopo LSH: 1630
Tempo di esecuzione Shingling: 0.9748 secondi
Tempo di esecuzione MinHashing: 10.9280 secondi
Tempo di esecuzione LSH: 0.0294 secondi
Tempo di esecuzione totale: 11.9321 secondi
Numero di Falsi negativi: 0

b: 50, r: 20, s: 0.8, num_hashes: 1000

Numero di confronti prima di LSH: 1176
Numero di confronti dopo LSH: 552
Tempo di esecuzione Shingling: 0.9725 secondi
Tempo di esecuzione MinHashing: 10.5357 secondi
Tempo di esecuzione LSH: 0.0241 secondi
Tempo di esecuzione totale: 11.5323 secondi
Numero di Falsi negativi: 20

SI OSSERVI COME, NON APPENA CAMBINO I VALORI DI b ED r (dipendenti da s), IL NUMERO DI FALSI NEGATIVI AUMENTI DRASTICAMENTE.

UMAP 48 documenti (24 coppie), k = 3, n_neighbors = 2, distance 0.2

Numero di confronti prima di umap: 1176
Numero di confronti dopo umap: 24 (verifica solo documento con la sua copia)
Tempo di esecuzione Shingling: 3.4893 secondi

Tempo di esecuzione UMAP: 2.4168 secondi

Tempo di esecuzione totale: 5.9061 secondi
Numero di falsi negativi: **0**

UMAP 53 documenti (23 coppie, 1 tripla e 4 singleton), k = 3, n_neighbors = 2, distance 0.2

Numero di confronti prima di umap: 1431
Numero di confronti dopo umap: 29 (solo 4 falsi positivi)
Tempo di esecuzione Shingling: 8.3335 secondi
Tempo di esecuzione UMAP: 2.9275 secondi
Tempo di esecuzione totale: 11.261 secondi
Numero di falsi negativi: **0**

Il collo di bottiglia del pc è lo shingling, dovuto ad una ram di piccole dimensioni, bypassata questa fase, lo UMAP è più che comparabile con i tempi dell'altro pc. I risultati sono invece notevolmente migliori a livello di qualità.

Conclusioni

I risultati portano ad un risparmio nel numero di confronti, le prestazioni della riduzione di dimensionalità risultano essere però migliori con i documenti in esame. In particolare risulta ottenere performance superiori sia in qualità che in tempi di esecuzione. Nel caso di UMAP, risulta essere ormai standardizzato per quanto riguarda i suoi parametri in biologia computazione, non ci sono invece tracce in rete di utilizzi in questo settore, per cui c'è largo spazio per l'ottimizzazione. L'aumento del numero di documenti può chiaramente comportare un aumento dei tempi di esecuzione, che però risulta essere lento e soprattutto può essere notevolmente mitigato tramite una preliminare riduzione di dimensionalità di tipo lineare con una PCA che cerca di preservare informazione per ciascun documento in un numero di componenti che solitamente viene mantenuto non superiore a 10. Abbiamo condotto alcune prove e le prestazioni sono qualitativamente comparabili. Può essere interessante tramite modifiche dei paramentri UMAP cercare di clusterizzare i documenti che trattano argomenti simili, modificando quindi il task, così come comparare i risultati con tecniche differenti ma con prestazioni tipicamente peggiori come t-SNE.

Per quanto riguarda il task che ci eravamo prefissati, risulta che i documenti rielaborati da chatGPT sono identificabili e riconducibili all'originale tramite metodi computazioni.