Practica 1

Juan López, Marc Ortiz, Pere Ayats

Noviembre 2016

Introducción

En esta práctica están implementados algoritmos para hallar la similaridad entre documentos. Cada archivo de texto es representado por medio de un conjunto de k-shingles, en nuestra implementación los shingles son una string que puede contener k caracteres o palabras consecutivas.

Los algoritmos calculan la similaridad de Jaccard, una aproximación de Jaccard por medio de firmas minhash y localización de documentos similares usando un Locality Sensitive Hashing basa en firmas minhash. están basados en el libro Massive Datasets y no se separan mucho de la descripción a excepción de algunos pequeños detalles.

Jaccard

El Jaccard es de complejidad `O(n)`, tomando como parametros los dos sets de shingles de ambos documentos. Esta hecho con tree-sets y si se toma en cuenta la creación de estos la complejidad pasa a ser `O(n lgn)`.

Para calcular los elementos comunes, se itera ambos sets de forma ordenada. Cuando un elemento de un set es menor al otro se incrementa su iterador y cuando son iguales se aumenta en uno el contador. Al final se aplica la fórmula de intersección de conjuntos y se retorna.

```
double jaccard(const set<string>& d1,
const set<string>& d2) {
    int ncommon = 0;
    set<string>::iterator it1 =
        d1.begin(), it2 = d2.begin();
    while (it1!=d1.end() and it2!=d2.end()) {
        if (*it1 < *it2) {
            it1++;
        } else if (*it1 > *it2) {
            it2++;
        } else {
            it1++;
            it2++;
            ncommon++;
        }
    }
    return ncommon/double(d1.size()+d2.size()-ncommon);
}
```

Minhash Jaccard

Comparar `n` documentos por similitud de Jaccard es costoso, para obtener una aproximación usamos un vector de signatures minhash. Mientras mas funciones de hash se usen menor sera el error con el valor real, otra ventaja es que generalmente la matriz de signatures resultante es de menor tamaño que la de shingles.

Un gran componente sin duda es el generador de funciones de hash que permite simular las permutaciones. Las funciones son generadas al azar y no producen colisiones. Para lograr que dos elementos no se mapeen al mismo lugar se aprovecha en teoremas de teoría de conjuntos. Haciendo la multiplicación del número a hashear por un coprimo al tamaño del conjunto se sabe que no habrán repeticiones `i*coprimo mod size`. El número que hace multiplicar al número coprimo para que el módulo sea cero es size, por cada numero `0 < i < size` no puede haber repeticiones o de lo contrario el coprimo seria divisor de size.

```
long long int shuffler::shuffle(int i){
    return ((i+1)*coprime % (size+1))-1;
}
```

En la versión final no se usa un paso intermedio de compresión por hash. De esta forma hay menor probabilidad de bajar la precisión de la aproximación.

Minhash/LHS

Local Sensitive Hashing permite encontrar conjuntos de documentos similares rápidamente escogiendo un threshold, sin embargo esto se paga en variabilidad de las soluciones por haber la posibilidad de falso negativos y positivos. Las signatures de cada documento se fraccionan y se comparan por separado para aumentar la probabilidad de un match entre documentos. Siguiendo las especificaciones del libro, se hace en orden los siguientes pasos:

1. Escoger un valor de k aportado por el usuario y construir un conjunto de k shingles para cada documento. No se comprime el conjunto por medio de hashing.
2. Ordenar los shingles y e indexarlos en cada documento usando un map.
3. Escoger una longitud h aportada por el usuario para las signatures. Proveer el index al algoritmo para computar las signatures.
4. Escoger, ambos dados por el usuario, el threshold o el número de bandas a usar. Por medio de una búsqueda dicotómica se busca un par de valores `(nbandas, nfilas)` para el cual la aproximación del threshold sea mayor o igual al proporcionado.
5. Hashear cada fracción de banda a un array grande de buckets separados. Se aplica un hashing muy sencillo para reducir el ruido, solo se encontraran dos elementos en el mismo bucket si son iguales. Cuando se calculan los similares de un solo documento se optimiza de hacer un hashing una simple comparación de una a todas las signatures.
6. Construir los pares de candidatos después determinar los documentos similares pertenecientes al mismo bucket.

Experimentos

Para los siguientes experimentos hemos usado distintas colecciones de ficheros de texto, todas en ingles.

La colección tweets colección de datos son tweets obtenidos mediante la API de twitter. La variabilidad en la cantidad de caracteres de los tweets es baja, podemos asumir que el número de tweets es linealmente proporcional a la cantidad total de texto. Para cada observación se ha tomado la media de cinco ejecuciones.

La colección answers fue obtenida de las respuestas de una sola pregunta de Quora, son bueno ficheros de texto para encontrar similitudes entre respuestas que deberían ser similares.

La colección novels proviene de un conjunto de novelas clásicas.

La ultima colección es de permutaciones de 50 palabras.

Experimento 1

En la siguiente prueba se compara el crecimiento del tiempo de ejecución de cada método al encontrar documentos similares a uno dado.

Para comparar establecimos los ciertos parámetros entre todas las ejecuciones:

- 5 shingles
- 10 signatures
- 2 bands

Tweets	Jaccard	MinHash	LHS
10	0.004505s	0.006492s	0.006567s
100	0.04075s	0.047604s	0.046994s
1000	0.45367s	0.51788s	0.526521s
10000	5.37817s	6.18325s	5.94365s

Se puede observar que tanto Jaccard como MinHash y LHS crecen linealmente en tiempo de ejecución. Algo importante de notar es que hashing no se ejecuta más rápidamente que Jaccard porque la cantidad de comparaciones hechas no amortiza el tiempo de crear y montar las estructuras de datos que hacen MinHash posible. Lo mismo pasa con el LHS.

Experimento 2

En este experimento se observa como como varia el tiempo de ejecución al buscar pares de documentos similares dentro de una colección. Los parámetros fijos son los mismo que en el experimento 1.

Ahora si se puede observar que aunque Jaccard y MinHash crecen cuadraticamente MinHash se ejecuta mas rápidamente por que el costo de cada comparación es menor. LHS crece linealmente dado que solo necesitara hashear `n` documentos y luego dependiendo de la carga de cada bucket generar los pares.

Tweets	Jaccard	MinHash	LHS
10	0.004971s	0.005437s	0.009931s
100	0.067914s	0.073821s	0.054155s
1000	3.56031s	2.52831s	0.517803s
10000	¿ 239.133 s	239.133s	5.79142s

Experimento 3

En este experimento se estudia la precisión con 200 funciones de hash. Se puede observar que el error es bastante alto, para poder mejorarlo la cantidad funciones tendría que crecer mucho. Lo ideal seria tener todas las permutaciones posibles de los shingles para poder obtener una perfecta aproximación.

Novels	Jaccard	MinHash	LHS
Doc 1,2	0.0110141	0.37	0.359
Doc 1,3	0.00411464	0.41	0.405
Doc 1,4	0.00961538	0.29	0.280
Average			0.348

Experimento 4

Aqui se comprueba usando MinHash como se compara encontrar similitudes entre documentos usando word-shingles y char-shingles como es descrito en Massive Datasets. Este experimento se llevo a cabo con la colección de respuestas de Quora donde obtuvimos mejores resultados debido a que se usan las palabras auxiliares que permiten al algoritmo encontrar las word-shingles. Se puede observar que generalmente son similares.

WORD: 3 shingles, 200 hash signature.

Docs	Valor
7,1	0
7,2	0
7,6	0
7,8	0.565

CHAR: 8 shingles, 200 hash signature.

Docs	Valor
7,1	0.005
7,2	0.015
7,6	0.01
7,8	0.675