UA

Buscador

DanielAsensiRochDNI : **48776120C**

May 29, 2022

Contents

1 Introducción

En la presente práctica de la asignatura de Explotación de la información se nos plantea la problemática de finalizar con nuestro buscador, de tal manera que para el correcto funcionamiento del mismo antes deberemos tener plenamente operativas las funciones de tokenizar y indexar los términos que compondrán nuestra base de datos de búsqueda, así como los documentos en los que buscaremos.

Para poder realizar el proceso de búsqueda se nos plantea implementar los modelos **DFR y BM25**, los cuales nos posibilitarán la obtención de pesos en relevancia a la query insertada para la búsqueda.

2 Análisis de la solución implementada

Para implementar mi solución he optado por realizar una función angular, la cual se encargará de decidir cual de los modelos se ejecutará en cada momento dependiendo de los parámetros pasados a la misma de tal manera que será llamada por la función Buscar(), la función ha sido diseñada de la siguiente manera:

```
bool Buscador::SeleccionarMetodo(const int &numDocumentos, const int &numPregunta)
{
    // Tenemos que comprobar que el indice de preguntas no esta vacio
    // En el caso de que este vacio no podemos realizar la busqueda y devolvemos false
    if (indicePregunta.empty())
        return false;
    else
    {
        // cout << "Seleccionando metodo de busqueda...\n";
        if (this->formSimilitud == 0)
        {
            // cout << "Metodo de busqueda: DFR\n";
            realizarDFR(numDocumentos, numPregunta);
        }
        else
        {
            // cout << "Metodo de busqueda: BM25\n";
            realizarBM25(numDocumentos, numPregunta);
        }
        return true;
    }
}
```

En el caso de que hubiera **varias preguntas** en las que buscar se llamará a la función mediante un bucle el cual irá pasándole a la selección del método la pregunta y el número de documentos donde debe buscar.

En cuanto a los **algoritmos de similitud**, la solución que he optado por implementar ha sido la de una función sencilla en la que se recorrerá la colección de documentos que ha sido previamente indexada de tal manera, que para cada documento se recorrerá la query indexada, sacando los valores pertinentes del documento sobre las palabras de la query, una vez sacados los valores, deberemos aplicar su formula de la similitud, y por último una vez calculada la similitud para el documento con todos lo valores lo empujaremos dentro de la cola de prioridad.

De tal manera también se han utilizado librerías matemáticas para la realización de los logaritmos en base 2, así como para la obtención de los valores absolutos en los resultados.

2.1 Algoritmo utilizado para DFR

```
void Buscador::realizarDFR(const int &numDocumentos, const int &numPregunta)
{
    double vSimilitud = 0.0;
    int k = infPregunta.getNumTotalPalSinParada();
```

```
double wiq = 0.0; double wid = 0.0; int nt = 0;       int ft = 0.0;     int ftd = 0.0;
double ftq = 0.0; double lambda = 0.0;   double first = 0.0; double avg = 0.0;
int ld = 0.0;
priority_queue<pair<double, long>> aux;

avg = 1.0 * (double)informacionColeccionDocs.getNumTotalPalSinParada()
/ (double)informacionColeccionDocs.getNumDocs();

// cout << "Calculando similitud...\n";
// cout << indiceDocs.size() << " documentos indexados\n";
for (auto informacionDoc = indiceDocs.begin();
informacionDoc != indiceDocs.end(); ++informacionDoc)
{
    informacionDoc->second.getNumPalSinParada(ld);
    //  Para cada documento sacamos su similitud
    vSimilitud = 0.0;

    for (auto terminoPregunta = indicePregunta.begin();
    terminoPregunta != indicePregunta.end(); ++terminoPregunta)
    {
        ftq = terminoPregunta->second.getFt();
        wiq = 1.0 * (double)ftq / k;

        // Hay que sacar el id del documento, buscarlo en el indice de documentos y
        //sacar ft del termino en el documento
        int idDoc;
        informacionDoc->second.getIdDoc(idDoc);
        ftd = indice[terminoPregunta->first].getFtDoc(idDoc);
        indice[terminoPregunta->first].getFtc(ft);
        nt = indice[terminoPregunta->first].sizeL_docs();

        if (ftd == 0 || ft == 0 || nt == 0)
        {
        }
        else
        {
            double lambda = 1.0 * ft / indiceDocs.size();
            double ftdd = (double)ftd * log2(1.0 + (c * avg) / ld);
            wid = (log2(1.0 + lambda) + ftdd * log2((1.0 + lambda)
            /(lambda))) * ((ft +1) /(nt *(ftdd + 1)));
            vSimilitud += wiq * wid;
        }
    }

    if (vSimilitud != 0.0)
    {
        int idDoc;
        informacionDoc->second.getIdDoc(idDoc);
        aux.push(make_pair(vSimilitud, idDoc));
    }
}
for (int i = 0; i < numDocumentos && !aux.empty(); i++)
{
    docsOrdenados.push_back(ResultadoRI(aux.top().first, aux.top().second, numPregunta));
```

```
        aux.pop();
    }
}
```

2.2 Algoritmo utilizado para BM25

Para el algoritmo de BM25 he aplicado exactamente la misma estrategia que para el model DFR, ya que se realiza de forma rápida y sencilla

```
void Buscador::realizarBM25(const int &numDocumentos, const int &numPregunta)
{
    double vSimilitud = 0.0;double idf = 0.0;double ft = 0.0;double fd = 0.0;
    double avg = 0.0; // media en palabras (no de parada) de los documentos
    priority_queue<pair<double, long>> aux;

    avg = (double)informacionColeccionDocs.getNumTotalPalSinParada() / (double)informacionColeccionDocs

    for (auto informacionDoc = indiceDocs.begin(); informacionDoc != indiceDocs.end(); ++informacionDoc)
    {
        // Para cada documento sacamos su similitud
        vSimilitud = 0.0;
        for (auto infTermPregunta = indicePregunta.begin(); infTermPregunta != indicePregunta.end(); ++i
        {
            // Para cada termino de la pregunta lo buscamos en el indice
            auto informacionTermino = indice.find(infTermPregunta->first);
            // Comprobamos que no se haya acabado el indice
            if (informacionTermino != indice.end())
            {
                int idDoc;
                informacionDoc->second.getIdDoc(idDoc);


                if (informacionTermino->second.existeDocu(idDoc))
                {
                    int ft;
                    ft = informacionTermino->second.getFtDoc(idDoc);

                    int ftc;
                    informacionTermino->second.getFtc(ftc);
                    // Si el termino aparece al menos una vez en el documento aplicamos la formula
                    if (ft != 0)
                    {
                        int numSinParada;
                        informacionDoc->second.getNumPalSinParada(numSinParada);

                        idf = log2(((double)informacionColeccionDocs.getNumDocs() -
                        (double)informacionTermino->second.sizeL_docs() + 0.5) /
                        ((double)informacionTermino->second.sizeL_docs() + 0.5));
                        fd = (ft * (k1 + 1.0)) /
                        (ft + k1 * (1.0 - b + (b * abs((double)numSinParada /avg))));

                        vSimilitud += idf * fd;
                    }
                }
```

```
            }
        }
        if (vSimilitud != 0.0)
        {
            int idDoc;
            informacionDoc->second.getIdDoc(idDoc);
            aux.push(make_pair(vSimilitud, idDoc));
        }
    }

    for (int i = 0; i < numDocumentos && !aux.empty(); i++)
    {
        docsOrdenados.push_back(ResultadoRI(aux.top().first, aux.top().second, numPregunta));
        aux.pop();
    }
}
```

2.3 Estructura de datos

Las estructurar de datos elegidas para recorrer los documentos y los terminos de la query, es la misma que las implementadas en las prácticas anteriores, las tablas hash ya que el acceso directo a los datos de las mismas mediante clave tiene una complejidad de O1.

En cuanto a los datos utilizados para realizar los cálculos de los pesos, he optado por utilizar números enteros casteados a doubles para por optar a obtener más números decimales y por lo tanto una mayor precisión en los resultados obtenidos.

Se ha mantenido el uso de las colas de prioridad para almacenar los resultados obtenidos por los cálculos de manera auxiliar, estos resultados luego serán transformados a un vector.

3 Justificación de la solución elegida

Finalmente he optado por la anteriormente explicada solución, porque es la que menores tiempo de búsqueda me ha arrojado y con la que mejores valores de similitud y más redondeados he obtenido, de tal manera a continuación explicaré las soluciones probadas y finalmente descartadas.

3.1 Alternativas probadas y descarte

El uso de las **colas de prioridad para el almacenamiento de datos** de manera continuada fue descartado por el difícil acceso a los mismos, ya que estos una vez realizado el pop se destruyen y por lo tanto si queríamos recorrer una lista de prioridad para imprimir sus datos debíamos realizar una copia defensiva de la misma.

El **cálculo del peso de un termino** en un documento mediante el tamaño en bytes que ocupaba el termino y multiplicado por las veces que aparecía el mismo, dividido por el peso total del documento, esta solución fue descartada ya que aunque es una manera valida de calcularlo los resultados eran muy dispares a los realmente necesitados.

Intento de **ignorar documentos**, este intento consistió en realizar una función la cual nos devolviera un booleano informándonos si un documento contenía o no el termino, esta función fue descartada simplemente porque ralentizaba el código de sobremanera.

4 Análisis de las mejoras implementadas en la práctica

La mejora más destacable implementada fue el uso de un vector en vez de la cola de prioridad inicialmente propuesta, esta mejora fue implementada ya que para recorrer una cola de prioridad durante la impresión debía realizar una copia defensiva de la misma, lo cual generaba un gran gasto de memoria y computacional,

de esta manera al iterar sobre un único vector es más sencillo y mejor computacionalmente que la cola de prioridad, de tal manera esta manera las colas de prioridad no fueron descartadas completamente, ya que se puede apreciar su implementación en la realización de los modelos DFR y BM25, estas colas auxiliares implementadas **priority_queue pair double, long aux;** nos ayudan a guardar la información de manera temporal.

Otra de las mejorar fue utilizar las funciones **getFt(), getFtc()** para obtener las frecuencias de los términos y ahorrar costo computacional a la hora de realizar operaciones en la obtención de la similitud.

Además una mejora que implemente de manera involuntaria pero la cual fue muy útil a la hora de insertar los documentos en el vector fue la comprobación del valor **vSimilitud**

5 Análisis de eficiencia computacional

En el siguiente análisis de eficiencia computacional he utilizado el siguiente main:

```
int main() {
        double aa = getcputime();

        IndexadorHash b("./StopWordsIngles.txt", ". ,:",
        false, false, "./indicePruebaIngles", 0, false, false);
        b.IndexarDirectorio("CorpusTime/Documentos/");
        b.GuardarIndexacion();

        Buscador a("./indicePruebaIngles", 0);
        a.IndexarPregunta("KENNEDY ADMINISTRATION PRESSURE
        ON NGO DINH DIEM TO STOP SUPPRESSING THE BUDDHISTS . ");
        a.Buscar(423);
        //a.ImprimirResultadoBusqueda(423);

        double bb=getcputime()-aa;
        cout << "\nHa tardado " << bb << " segundos con DFR:\n\n";

        time_t inicioB, finB;
        time(&inicioB);
        double aaB=getcputime();
        a.Buscar("CorpusTime/Preguntas/", 423, 1, 83);
        //a.ImprimirResultadoBusqueda(423);
        double bbB=getcputime()-aaB;
        cout << "\nHa tardado " << bbB << " segundos con BM25\n\n";

        return 0;
}
```

De tal manera que el tiempo arrojado por las función de Buscar es el siguiente:

```
Ha tardado 2.13522 segundos con DFR:
Ha tardado 0.439238 segundos con BM25
```

6 Precisión y cobertura

Para el análisis de precisión y cobertura se ha ejecutado el buscador un total de cuatro veces con distintos modelos y stemmings. En concreto hemos ejecutado la función de **Buscar()**con el modelo DFR y el modelo Okapi BM25, a su vez indexando gran cantidad de documentos, con y sin stemming, obteniendo la siguiente tabla gracias a la aplicación **trec_eval** proporcionada por el profesor. Como podemos observar la tendencia de los modelos ya sea aplicando o no el stemming es de que a cuanto mayor recall menor precisión en los

Recall	DFR (stemming)	DFR	BM25 (stemming)	BM25
0	0,6163	0,7083	0,6132	0,7054
0.1	0,6163	0,7083	0,6077	0,7054
0.2	0,6054	0,6909	0,5971	0,6939
0.3	0,5887	0,6747	0,5709	0,6689
0.4	0,5703	0,6516	0,5522	0,6315
0.5	0,5471	0,6330	0,5343	0,6150
0.6	0,4738	0,5658	0,4631	0,5315
0.7	0,4432	0,5256	0,4260	0,4868
0.8	0,4268	0,5092	0,4196	0,4758
0.9	0,3694	0,4368	0,3578	0,4086
1	0,3614	0,4286	0,3514	0,4019

Figure 1: Tabla obtenida

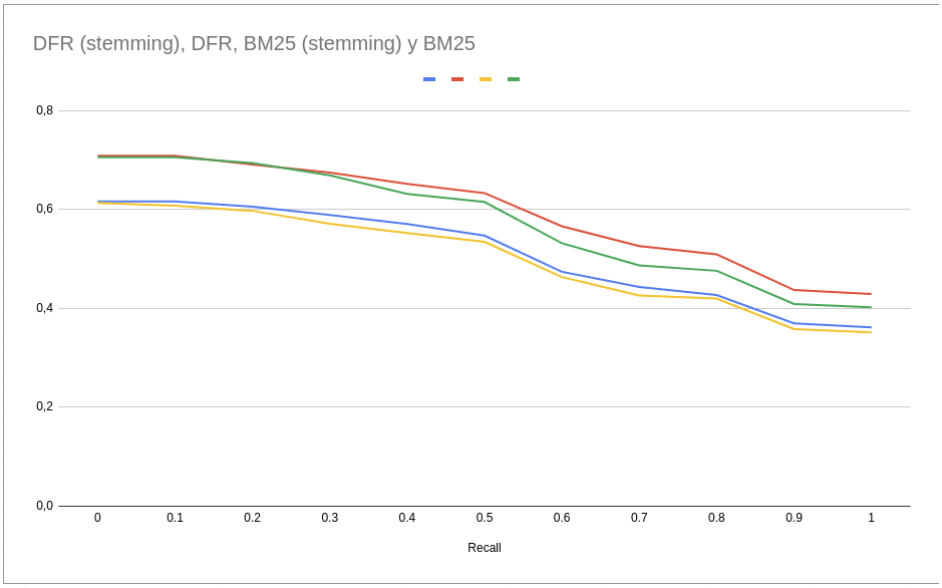

Figure 2: Gráfica de precisiónes obtenidas

resultados de la búsqueda. Como podemos observar en la tabla de resultados ambos modelos se encuentran muy a la par en cuanto a la precisión de los resultados refiere independientemente del Recall asignado, pero cabe destacar que el modelo **DFR en cuanto a mi práctica se refiere ha sido el que mayor precisión ha adquirido**, y el que **peor precisión ha sido el modelo BM25 aplicando el stemming.**