SISTEMAS DE RECOMENDACIÓN

MODELOS BASADOS EN EL CONTENIDO

Nombre: Alejandro Martín de León
Contacto: alu0101015941@ull.edu.es

Índice

Introducción

Este informe tiene como objetivo detallar el código que ha sido elaborado, así como de realizar un pequeño análisis de los datos obtenido tras los ejemplos propuestos.

Estructura del repositorio

El repositorio se compone en los siguientes directorios:

```
.
├── bin
│   └── main.o
├── csv
│   ├── file-01.txt.csv
│   ├── file-02.txt.csv
│   └── file-03.txt.csv
├── files
│   ├── file-01.txt
│   ├── file-02.txt
│   └── file-03.txt
├── include
├── makefile
├── README.md
├── sist_recomendacion
└── src
    └── main.cpp
```

Con el objetivo de comprender mejor cada uno de los directorios, se procede a realizar una breve descripción:

- **bin**: Aquí se almacenan los ficheros .o
- **csv**: Después de ejecutar el programa, si se ha elegido la opción CSV, se generará un fichero de salida por cada fichero de entrada, conteniendo la tabla resultado.
- **files**: Es el directorio que almacena los ficheros de entrada.
- **src**: Almacena el código fuente principal del programa
- **makefile**: Al ejecutar la orden "make", se encarga de compilar todo el programa principal.
- **sist_recomendacion**: Es el ejecutable creado tras compilar el programa.

Detalles sobre el código

Principalmente se ha creado el programa sobre un única fuente de código, concretamente situado en el directorio src nombrado como main.cpp.

A continuación se procede a realizar una breve descripción del programa, así como de indagar en sus principales funciones de este programa, ignorando algunas de las funciones menos importantes o más obvias:

- **Función de lectura de ficheros:** Esta función se encarga principalemente de leer el fichero pasado por argumento. Este comprueba si no se ha podido realizar la apertura del mismo, si no es así, procede a introducir toda la lectura cruda del fichero en un vector de strings. Previamente se establece la transformación del string a letras minúsculas y la eliminación de todos los carácteres posibles de puntuación, como pueden ser los puntos, las comas, los guiones, entre otros. Por este motivo, y como bien se puede ver en los ficheros que se han colocado de ejemplo en el directorio **/files**, existen palabras compuestas por guión en ingles. En este caso se ha decidido eliminar el guión y tratar a la palabra como una única palabra sin guión ni espacio. Un ejemplo de esto es:

> Black-cherry ---> blackcherry

Posteriormente, se introduce en otro vector de strings, todas las palabras pero esta vez sin repetición. Finalmente, se cierra el fichero.

```
void readFile(string filename, vector<string> &norepeated_words,
vector<string> &original_words) {

  string lines;
  string source = "./files/" + filename;
  string word;

  ifstream file(source);
  if (!file) {
    cerr << "\nERROR: file name '" << filename << "' does not exist or is
not readable." << endl;
    exit(1);
  }
  while (file >> word) {
```

```
    word = removePunctuation(word);
    transform(word.begin(), word.end(), word.begin(), ::tolower);
    if (isRepeated(norepeated_words, word) == false) {
      norepeated_words.push_back(word);
    }
    original_words.push_back(word);
  }
  file.close();
}
```

- **Función de frecuencia de palabras:** Esta función tiene como objetivo principalmente realizar una cuenta de las repeticiones que se produce de una palabra en un mismo documento. Su funcionamiento es sencillo, recorriendo el con dos bucles por un lado el vector de palabras sin repetición y comparando cada una por todas las palabras extraídas del documento. Finalizado el conteo, se introduce el mismo en un vector con las mismas dimensiones, logrando así que en cada posición del vector de frecuencia se le asocie la palabra situada en el mismo indice de posición del vector de palabras sin repetición. El código se muestra a continuación:

```
vector<unsigned int> wordFrequency(vector<string> norepeated_words,
vector<string>  original_words) {
  vector<unsigned int> word_frequencies;
  unsigned int counter = 0;
  for (long unsigned int i = 0; i < norepeated_words.size(); i++) {
    for (long unsigned int j = 0; j < original_words.size(); j++) {
      if (norepeated_words[i] == original_words[j]) {
        counter++;
      }
    }
    word_frequencies.push_back(counter);
    counter = 0;
  }
  return word_frequencies;
}
```

- **Función para el cálculo de IDF:** Esta función se encarga principalmente de analizar los vectores y matrices recogidos con anterioridad y aplicar las operaciones necesarias para hallar el vector de valores IDF asociado a cada palabra. La matriz de strings **files_words** se descompone por columnas con cada uno de los términos sin repetición y por cada fila en los diferentes documentos de entrada. De esta forma, haciendo uso de dos bucles for, será posible realizar el cálculo para cada una de las palabras que aparece en todos los documentos. Con la ayuda de otro bucle for, se procederá a realizar la búsqueda de cada una de las palabras en todos los documentos. Además, con el condicional if, se establece que, si la palabra ha sido encontrada al menos una vez en el vector, que se cuente en una variable de tipo entero. Por último, en cada posición de forma simétrica a la matriz original, se colocan asociado a las mismas posiciones, el número de repeticiones que se han encontrado en los distintos documentos por cada palabra de los mismos.

```
vector<vector<double>> IDF(vector<vector<string>> files_words) {
    string word;
    unsigned int counter = 0;
    vector<vector<double>> v_idf;
    v_idf.resize(files_words.size(), vector<double>(maxCols(files_words)));
    for (unsigned int i = 0; i < files_words.size(); i++) {
      for (unsigned int j = 0; j < files_words[i].size(); j++) {
        word = files_words[i][j];
        counter = 0;
        for (unsigned int k = 0; k < v_idf.size(); k++) {
          auto finder = find(files_words[k].begin(), files_words[k].end(),
word);
          if (files_words[k].end() != finder) {
            counter += 1;
          }
        }
        v_idf[i][j] = counter;
      }
    }

    (. . .)
```

Finalmente, se realizan dos bucles adicionales, sobre los que se realizará la operación que determinará el valor idf para cada término. La operación es la siguiente:

$$IDF(x) = log\frac{N}{dfx}$$

Donde **N** es el número de todos los documentos que pueden ser recomendados y **dfx** el números de documentos en los que la palabra elegida aparece.

```
(. . .)
  for(unsigned int i = 0; i < v_idf.size(); i++) {
    for(long unsigned int j = 0; j < v_idf[i].size(); j++) {
      if (files_words[i][j].empty() == false)  {
        float result = static_cast<float>(v_idf.size()) / v_idf[i][j];
        v_idf[i][j] = log10(result);
      }
    }
    cout << "\n";
  }
  return v_idf;
}
```

- **Funciones para calcular la Similaridad Coseno:** Para conseguir este cometido, se ha estructurado el programa en tres funciones diferentes. En primer lugar, como primer paso se necesita realizar la longitud de cada uno de los vectores asociado a cada uno de los documentos. La operación es:

$$vectorLenth = \sqrt{word_1^2 + word_2^2 + word_3^2 + ... + word_n^2}$$

Esta operación se ha de realizar por cada una de las líneas de la matriz de palabras, es decir, por cada documento. El resultado será almacenado en un vector.

```
vector<double> vectorLength(vector<vector<double>> idf) {
  vector<double> idf_length;
  double result = 0;
  for (unsigned int i = 0; i < idf.size(); i++) {
    result = 0;
    for (unsigned int j = 0; j < idf[i].size(); j++) {
      result = result + (pow(idf[i][j], 2));
    }
    result = sqrt(result);
    idf_length.push_back(result);
  }
  return idf_length;
}
```

Otra de las operaciones previas a realizar es lo que se conoce como la normalización del vector. Esto no es más que, la división de cada uno de los valores IDF de cada termino entre la longitud del vector del documento, es decir:

$$normalized = \frac{IDF(word_x)_y}{vectorLenth_y}$$

Donde **x** representa el id de la palabra y **y** el documento asociado.

```
vector<vector<double>> normalizeVector(vector<vector<double>> idf,
vector<double> idf_length) {
  vector<vector<double>> normalizedMatrix;
  normalizedMatrix.resize(idf.size(), vector<double>(idf[0].size()));
  for (unsigned int i = 0; i < idf.size(); i++) {
    for (unsigned int j = 0; j < idf[i].size(); j++) {
      normalizedMatrix[i][j] = (idf[i][j] / idf_length[i]);
    }
  }
  return normalizedMatrix;
}
```

Una vez se han realizado estos cálculos, se procede a realizar la operación necesaria para hallar la similitud entre cada par de documentos introducidos. Para ello, se hace uso de la funcion **cosineValues**, que dado la matriz normalizada y el conjunto de palabras, calculará la similtud de términos. La fórmula que representa el código es la siguiente:

$$cos(A1, A2) = wordA1_1 * wordA2_1 + wordA1_2 * wordA2_2 + ... + wordA1_n * wordA2_n$$

Donde se realiza la multiplicación del término IDF de las palabras que son iguales de entre ambos documentos. Es decir, en la fórmula, wordA1_1 y wordA2_1 representan el IDF de la misma palabra en los diferentes documentos.

Para lograr eso, se ha diseñado el siguiente código:

```
void cosineValues(vector<vector<double>> normalizedMatrix, unsigned int
size, vector<vector<string>> files_words) {
  unsigned int aux = 1;
  double result = 0;
  vector<double> cosine_values;
  for (unsigned int i = 0; i < size; i++) {
    aux = i+1;
    while (aux < size) {
        result = 0;
        for (unsigned int j = 0; j < normalizedMatrix[0].size(); j++) {
          for (unsigned int k = 0; k < normalizedMatrix[0].size(); k++) {
            if ((files_words[i][j] == (files_words[aux][k]) &&
(files_words[i][j].empty() == false))) {
              result = result + (normalizedMatrix[i][j] *
normalizedMatrix[aux][k]);
            }
          }
        }
      cout << "cos(A" << i << ", A" << aux << ") = " << setprecision(5) <<
result << endl;
      cosine_values.push_back(result);
      aux++;
    }
  }
}
```

Como se puede observar, se realiza una comprobación de que ambas palabras sean la misma y si y sólo si eso se cumple, se realiza la operación de IDF asociada a esa palabra en cada documento.

El resto de funciones que se encuentran en el código son pequeñas funcionalidades que tienen como objetivo mejorar el programa para se ejecute de forma óptima y correcta, pero se considera que su explicación queda lo suficientemente detallada en los propios comentarios que las describen en el programa main.cpp situado dentro del directorio /src.

Ejecución y uso del programa

Compilación

En primer lugar, para compilar el programa será necesario seguir cualquiera de las dos siguientes opciones:

- Hacer uso del programa **make**:

  ```
  make
  ```

  O para eliminar los ficheros csv y de ejecución:

  ```
  make clean
  ```

- Ejecutar la siguiente línea:

  ```
  g++ -g src/main.cpp -o sist_recomendacion
  ```

Ejecución

Posteriormente, se procederá a ejecutar el programa. Para ello, es necesario pasarle unos argumentos:

- **-f** ó **--file** [arg1] [arg2] [...]: Este comando permitirá establecer todos los ficheros que se han de querer referenciar para la ejecución del programa. Obligatoriamente será necesario que éstos estén situados dentro del directorio /files. Es importante colocar todos los ficheros seguidos de este comando y antes de usar cualquier otro comando, ya que produciría un error. A continuación se muestra dos ejemplos de usos:

  Uso erróneo:

  ```
  ./sist_recomendacion doc1.txt --file doc2.txt
  ```

  Uso correcto:

  ```
  ./sist_recomendacion --file doc1.txt doc2.txt -c
  ```

- **-c** ó **--csv**: (Opcional) Este comando permitirá la salida de datos a través de documentos csv que se generarán dentro del directorio /csv. Si los ficheros existían con anterioridad, serán sobreescritos.

- **-h** ó **--help**: (Opcional) Este comando muestra la ayuda. Se muestra a continuación un ejemplo de su uso:

```
ale@pc:~/Documentos/ULL2122/GCO/GCO-ModeloBasadoenContenido(master)$
./sist_recomendacion --help

Content-based Filtering
This app filters files data and find out the similarity between all
introduced txt files. Then, it prints the results on terminal and/or on a
csv file

To compile: 'make' or 'g++ -g src/main -o app_name'
```

```
usage: app_name [options] input1 input2 (...)
    -f, --file: One or more file inputs could be placed before this
option.Important: Only the files between this option and another will be
redeabled.
    -c, --csv: This option puts the output of the app into an .csv file at
the csv directory.
    -h, --help: Prints help table.
```

Un ejemplo de ejecución del programa sería el siguiente:

```
./sist_recomendacion --file file-01.txt file-02.txt file-03.txt --csv
```

Con la sentencia anterior se ejecutará el programa para esos tres ficheros y además se imprimirá la salida tanto por los ficheros csv como por la terminal.

Análisis

Dado los ficheros proporcionados en el siguiente enlace se va a realiar un breve análisis sobre los datos resultantes del programa. Tras realizar la ejecución y observar los datos almacenados en los csv, se puede apreciar que realmente son pocas las palabras que generan una repetición con el resto de documentos. Incluso para un mismo documento, se tratan de pocas las palabras que superan 2 o 3 repeticiones. Normalmente se tratan de conectores, palabras clave o determinantes y preposiciones. Es por esto, por lo que se espera que la similitud de entre los diferentes documentos sea realmente no demasiado elevada, debido que son más bien poca las palabras que coinciden. Tras realizar la ejecución del programa, al final de la terminal obtenemos las siguientes similitudes coseno:

```
cos(A0, A1) = 0.0016757
cos(A0, A2) = 0.0066367
cos(A1, A2) = 0.0068222
```

Como se puede observar, a penas se parecen los diferentes documentos, ya que el número que se obtiene se aleja bastante de lo que sería una similitud exacta. Se observa además que la pareja de documentos que más se asemejan son la del "file-02.txt" y "file-03.txt".

Volver al Inicio