

Informe

De

Entrega

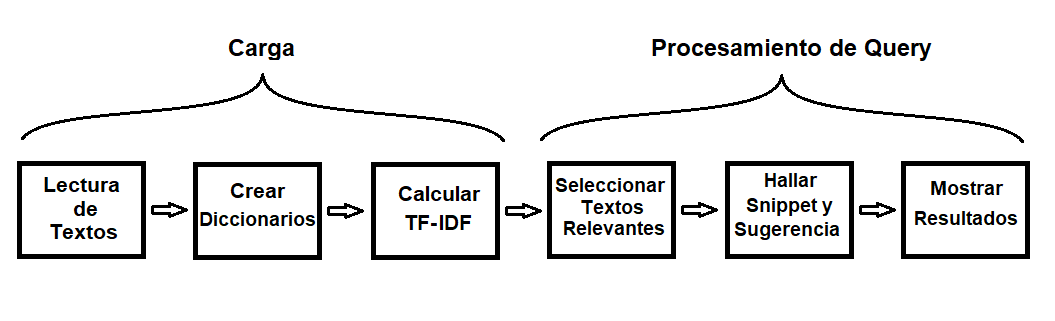
Alumno: Jonathan de Armas Avila

Grupo: C-111

Hi! Es un placer presentarles mi proyecto “ Moogle!”. Describiré sus aspectos y cualidades más importantes y daré una breve explicación de cómo funciona.

“Moogle!” intenta recrear una pequeña versión de lo comúnmente conocido como “buscador de google” (para nada deducible por el nombre) donde al introducir una palabra o frase, la página devuelva una serie de documentos de la base de datos que deben ser los más “relacionados” de alguna forma con el texto propuesto por el usuario.

La directriz a seguir por mi solución es la siguiente:



1 – Proceso de carga

* 1. – Lectura de textos:

Para esto creé un script “MoogleReader.cs” en el cual itero por todos los \*.txt de la carpeta Content, los convierto en strings. No tiene mucha ciencia.

* 1. – Creación de diccionarios:

En mi script “MoogleDictionary.cs” voy a tokenizar los strings que representan a cada texto(eliminar caracteres que no sean alfanuméricos) para luego dividirlo en arreglos de strings que contienen las palabras de cada texto. Luego itero por cada arreglo y genero los “diccionarios” de cada documento. Para esto necesitaba una estructura de datos donde pudiera guardar una pareja de <string,int>. Decidí utilizar “Dictionary<string,int>” ya que esta pincha por detrás como un AVL (un árbol binario balanceado)1 lo que me permite que las preguntas de si ya existe un cierto string así como la extracción de los datos de este tenga una complejidad de O(logn), lo que aumenta la eficiencia de mi solución. También necesito un diccionario general de todas las palabras que hay entre todos los textos, el cual creo de la siguiente manera: mientras voy rellenando los diccionarios locales, cada vez que me encuentro una palabra por primera vez en un documento, pregunto si está en mi diccionario global, si existe, aumento en uno la cantidad de textos en que aparece esta palabra, si no existe, la añado a mi “GlobalDictionary” con valor 1, que es la cantidad de textos en los que me la he encontrado hasta ahora. Esto será necesario para el cálculo del IDF más adelante.

* 1. – Calcular TF-IDF

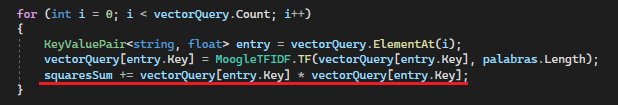
En esta parte utilicé la fórmula convencional de TF\*IDF, simplemente adicioné 1 en el numerador y el denominador para evitar casos como divisiones entre 0. En el script “MoogleTFIDF.cs” tengo una función estática vacía que denominé “GetValues()” la cual genera al inicio de su constructor un objeto del tipo “MoogleDictionary” que genera los diccionarios; esta, a su vez, crea en su constructor un objeto tipo “MoogleReader” que lee los documentos. Esto se ejecuta de forma “recursiva”, primero se leen los textos, luego se crean los diccionarios y al final se calculan los tf-idf. Esta función “GetValues()” es llamada en el script “Program.cs” de MoogleServer la cual realiza estos precálculos mientras carga la página en tu navegador.

Con esto termina el proceso de carga de la solución. En este momento la página muestra su interfaz y el usuario introduce su búsqueda, a lo que procede el procesado de la query:

2- Procesamiento de la Query

2.1 – Selección de textos relevantes:

Se llama a la función query a la que se le pasa como parámetro un string: la búsqueda. Este string es procesado como si fuera un texto más, se tokeniza, se pica, se crea el diccionario de la query y se calcula el tf-idf de cada palabra de esta (en este paso utilizo una variable int squaresSum en la que al terminar el ciclo voy a tener la sumatoria de los cuadrados de los tf-idf de cada palabra de la query(lo que abstractamente llamamos el módulo del vector query):



esto nos resultará útil en el cálculo de la similitud de los cosenos.

Luego de esto voy a comenzar a crear los elementos “SearchItem”, los cuales serán los resultados de la búsqueda. Genero cada uno con el título de cada documento (utilizando un substring de la dirección del documento en almacenamiento), el snippet (que por el momento lo igualaré a “ “), y el score, el cual nos dará la relevancia de cada documento con respecto a la query. Para esto utilizo la fórmula de similitud de coseno convencional, sin más. El score es bastante acertado de esta forma. Teniendo la lista de SearchItems ordenada solo basta quedarse con los 5 primeros que serán los resultados de nuestra búsqueda.

2.2- Hallar Snippet y sugerencia

Ya con nuestros SearchItems ordenados necesitamos un Snippet, un pedazo del documento que tenga cierta relevancia con respecto a nuestra query. En mi caso lo hice de la siguiente forma: me quedé con la palabra más valiosa de la query con respecto a cada texto y devolví un rango de palabras donde apareciera esta. Realmente comencé a desarrollar un snippet más personalizado el cuál devolviera el rango de 15 o 20 palabras donde apareciera la mayor cantidad de palabras de la query, pero necesitaba más información de cada texto la cual debí haber extraído a priori en mi solución, por lo que comenté esta función y utilicé la otra que entendí la podría realizar más rápido. Pienso poder terminar este snippet más potente en un período de tiempo.

Con esto hemos terminado nuestros SearchItems y estamos listos para darle respuesta a búsqueda del usuario pero… ¿Qué pasa si el usuario nos introduce palabras o una frase entera que no está en nuestro diccionario de palabras? ¿O si simplemente lo escribe mal ? ¿No sería tremenda talla preguntarle lo que quizás quiso decir, utilizando palabras cercanas a las que puso, pero que estén dentro de nuestro diccionario ? Aquí entra en juego la sugerencia, Fiel escudera de la RAE y protectora del reino de la gramática. Utilizando el conocido algoritmo denominado Distancia de Levenshtein, el cual con un poquito de DP nos permite hallar la “distancia mínima” entre dos palabras y luego cambiar la palabra de la query que no aparezca, por la más cercana que tengamos en nuestro diccionario( no adjunto el código porque es el usual, sin modificación alguna). Ya solo queda..:

2.3- Mostrar resultados

Y con esto termina la ejecución de nuestro código, mostrando los 5 resultados con su Snippet y una sugerencia en caso de haber palabras inexistentes en nuestro diccionario.

3- Extras

Tengo un script “Utilities.cs” que creé para poner las cosas que utilizaría en diferentes clases en el cual solo terminé poniendo la distancia de Levenshtein y el pequeño algoritmo para tokenizar y picar los textos.

También añadí algunas funcionalidades extras:

- Si presionas la sugerencia te mostrará nuevos resultados tomando como búsqueda la sugerencia.

- El título del resultado es un enlace que abre en otra ventana el \*.txt referido.

- El botón Enter ejecuta la búsqueda (que tedioso ir a dar clic en buscar cada vez que escribo algo).

- Modifiqué un poco el frontend de la página con temática de pokemón pues porque me gusta XD (¿ por qué no, eh ?).