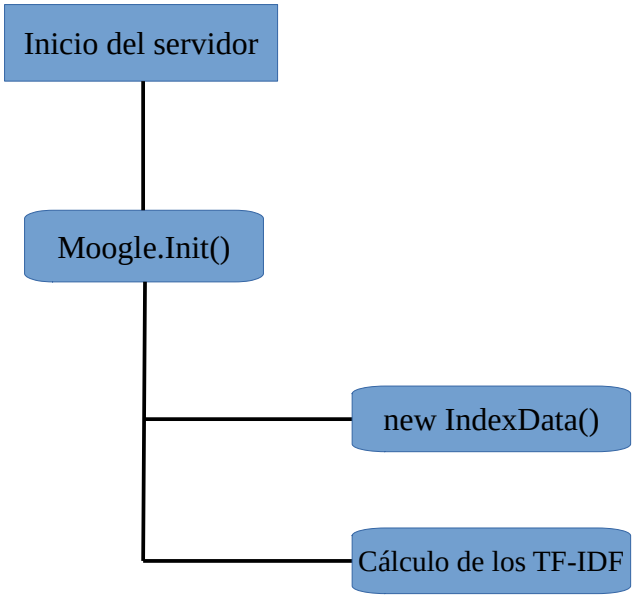
PROYECTO DE PROGRAMACIÓN CIENCIAS DE LA COMPUTACIÓN 1er AÑO

Fernando Valdés García
Grupo C-122

Curso 2021

Indexado de los documentos



Clase IndexData: Es la encargada de leer todos los documentos al iniciar el servidor y almacenar las palabras en estructuras de datos para mayor velocidad en las búsquedas.

Propiedades:

- **Words**: Dictionary<string, Dictionary<int, Occurrences>> - Para cada palabra distinta existente en los documentos, se almacenará, para cada documento en el que esta se encuentra, un objeto **Occurrences** con información sobre esa palabra en dicho documento.
- **Docs**: Dictionary<int, string> - A cada documento se le asignará un índice único. Cada índice en el diccionario apuntará a la ruta de su documento.
- **Roots**: Dictionary<string, List<string>> - Cada raíz generada por un Stemming sobre todas las palabras en los documentos, apuntará a una lista que contendrá todas las palabras cuya raíz es la actual.
- **Synonyms**: Dictionary<string, List<string>> - Para cada palabra, almacena una lista con sus sinónimos. Si una palabra no tiene sinónimos (no figura en el archivo de sinónimos), entonces no aparecerá en el diccionario.

Clase Occurrences: Almacena la relevancia (TF-IDF) y apariciones de cierta palabra en un documento específico.

Propiedades:

- **Relevance**: float – Representa el TF-IDF de la palabra en el documento. Si se le intenta asignar el valor 0 (sucedería cuando una palabra aparece en la gran mayoría de los documentos), este valor será modificado a uno muy pequeño. Esto es una medida para poder

diferenciar entre documentos sin palabras de una búsqueda y los documentos con solo palabras poco relevantes.

- **StartPos**: List<int> - Contiene las posiciones en las que aparece la palabra en el documento. Se guardará la posición en bytes del primer caracter de cada ocurrencia.

Funcionamiento del indexado:

Lo siguiente tiene lugar al inicializarse una nueva instancia de la clase **IndexData**:

1. Si no se desea precalcular los datos en los documentos, se intentará cargar la información almacenada en la carpeta **Cache** usando la clase **CacheManager**. Los datos guardados allí se almacenarán en **Words**, **Docs**, y **Roots**. Luego de esto se pasará al paso 6. Si en la llamada al programa se especificó que se deseaba precalcular los datos, o la caché almacenada es defectuosa, se pasará al paso 2.
2. Se obtiene un array de strings donde cada elemento es la ruta de un documento
3. Por cada documento:
   - Se le asigna un ID y este es guardado en **Docs** apuntando a su ruta.
   - Se llama al método **GetWords(StreamReader)**, el cual se encarga de parsear el contenido del documento. Recibe el StreamReader del documento que se está analizando, y devuelve una List<(string, int)>, en la que cada elemento representa una palabra del documento en minúsculas, y la posición en bytes del primer caracter de esta (en esta lista estarán todas las palabras del documento según el orden de aparición y sin filtrar repeticiones).
     Nota: Cuando se leen las palabras solo se considerarán como parte de una las letras y números. Para validarlos se usará el método **ArraysAndStrings.IsAlphaNum(char)**.
   - Se recorre cada par (palabra, posición) y por cada uno:
     - Si no se ha agregado la palabra a **Words** (es decir, nunca ha sido analizada), se halla su raíz con el algoritmo de Stemming y se agrega esta raíz **Roots**, y a la lista asociada a dicha raíz se le agrega la palabra actual.
     - Se añade esta ocurrencia específica a su lista **StartPos** correspondiente (se encuentra en **Words[palabra][documento].StartPos**).
3. Se ordena el diccionario tomando como criterio de comparación la longitud de las palabras.
4. Se calculan los TF-IDF de cada palabra en los documentos en que esta aparece.
5. Los datos calculados se serializan desde **CacheManager** y se almacenan en la carpeta **Cache** como archivos .json.
6. Se llama al método **LoadSynonyms()**. Este lee el archivo Thesaurus.csv y, para cada palabra que aparece en este, almacena sus sinónimos dentro de **Synonyms**, y cada uno apuntará a una lista con sus sinónimos.

Cálculo de los TF-IDF:

El TF-IDF de cada palabra es calculado mediante las siguientes fórmulas:

$$TF = \log_2(C+1) \qquad IDF = \log_2(N/K) \qquad TFIDF = TF * IDF$$

C: Cantidad de ocurrencias de la palabra en el documento
N: Cantidad total de documentos
K: Cantidad de documentos en los que aparece la palabra

Nota: En caso de que la palabra aparezca en mas del 95% de los documentos (valor almacenado en la variable estática **percentToNullify**), entonces se asignará un TF-IDF igual a 0 (el cual, como se mencionó antes, se modificará a un valor infinitesimal).

Sobre el tiempo de precálculo:

Los siguientes tiempos fueron calculados al ejecutar el proyecto 10 veces sobre un corpus de 125 documentos de 35Mb en total, con aproximadamente 125 mil palabras distintas (no totales). Se realizaron en una PC con las siguientes características:

- OS: Kali Linux GNU/Linux Rolling
- CPU: Intel Pentium 2.16GHz, Quad-Core
- RAM: 4Gb

Operación	Tiempo mínimo	Tiempo Máximo	Tiempo promedio
Indexado	55.25s	68.94s	60.22s
Ejecución (con los datos ya en la caché)	6.24s	6.75s	6.52s

Para este corpus, el contenido de la carpeta **Cache** tendrá un tamaño de 72,7Mb.

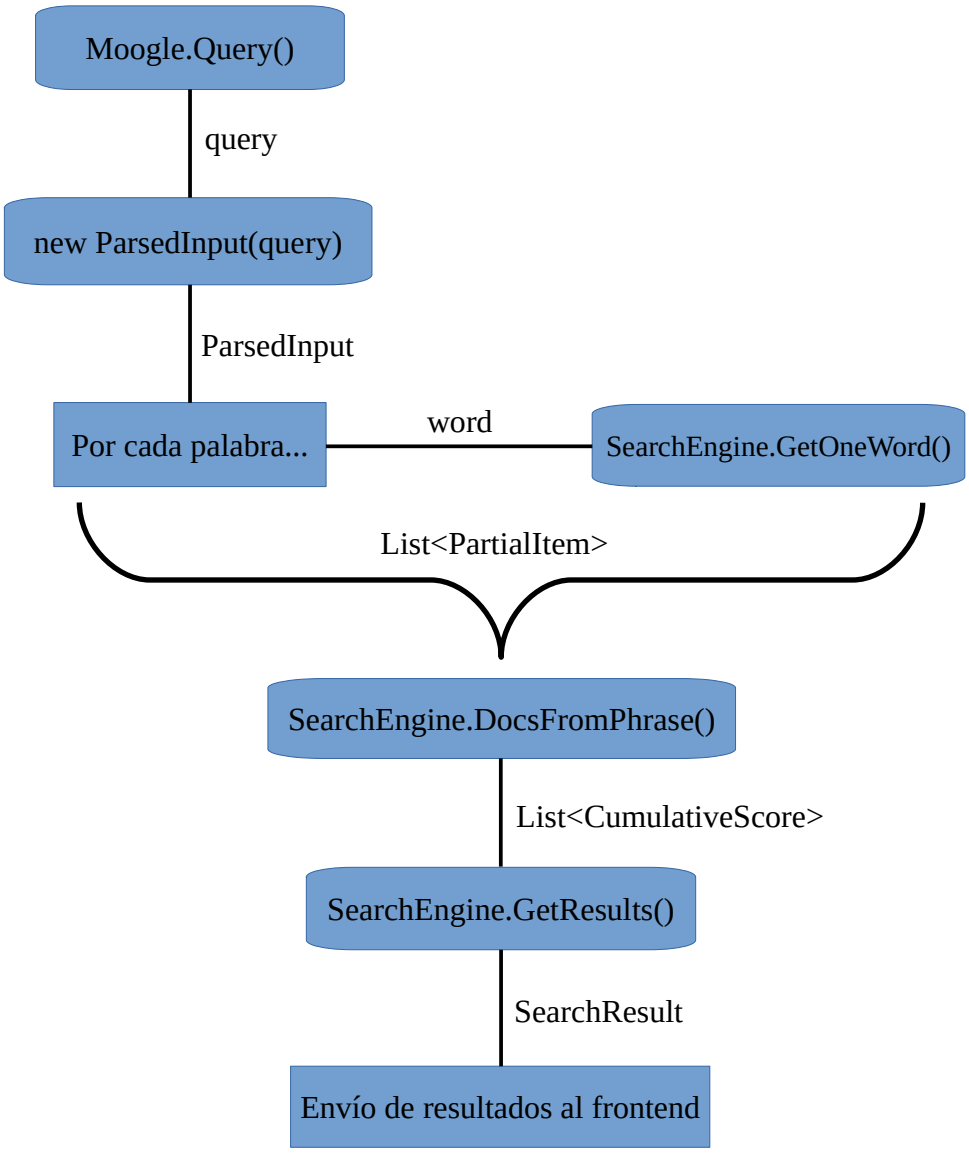
Sobre el algoritmo de Stemming:

La idea para reducir una palabra a su raíz es seguir un conjunto de reglas específicas para ir eliminando sufijos. El primer paso sería buscar y eliminar los pronombres sufijo que puedan haber. Luego se buscarían sufijos típicos de determinados sustantivos, adjetivos y adverbios. Si estos no fueron encontrados, se procede a buscar y eliminar los sufijos verbales. Finalmente se eliminarían los sufijos residuales que puedan haber quedado.
Para información más detallada sobre la lógica de este algoritmo dirigirse dirigirse a la bibliografía.
El punto de entrada de este algoritmo se encuentra en **Stemming.GetRoot(string)**. Es necesario mencionar que este algoritmo tiene cierto margen de error con ciertas palabras cuyas conjugaciones u otras derivaciones sean casos particulares de la lengua. También hay que señalar que debido a que el algoritmo no "entiende" el español, pueden haber resultados defectuosos. Por ejemplo, "morada" y "morir" tienen la misma raíz (mor), por lo que una búsqueda con una de estas palabras seguramente arrojará resultados con la otra.

Sobre los sinónimos:

En el archivo Thesaurus.csv se guarda una relación entre una palabra y sus sinónimos. No es necesario escribir una palabra más de una vez en todo el archivo, puesto que el código se encargará de, entre sinónimos, asociar todos con todos. Tampoco hay que escribir variaciones de las palabras (conugaciones, género, número, etc.). Hay que señalar que el contenido de este archivo es solo de muestra para dejar constancia de que la implementación funciona. Para un funcionamiento ideal se hace necesario agregar más sinónimos del español a mano.

Búsqueda de frases:



Clase ParsedInput: Se encarga de almacenar la query introducida por el usuario en un formato más cómodo para el trabajo con esta. Cualquier signo o símbolo (que no sea un operador) y los espacios sobrantes serán ignorados.

Propiedades:

- **Words**: List<string> - Cada elemento es una de las palabras introducidas por el usuario.
- **Operators**: List<string> - Se usa para almacenar las posiciones de los operadores "*!^". El elemento en la i-ésima posición representa los operadores que se introdujeron a la izquierda de la i-ésima palabra.
- **Tildes**: List<bool> - Se usa para almacenar los operadores '~'. Si la i-ésima posición es 'true' significa que entre la i-ésima y la (i+1)-ésima palabras, el usuario colocó un '~'.

Propiedades de solo lectura:

- **MandatoryWords**: string[] - Devuelve las palabras que tienen un operador '^'.
- **ForbiddenWords**: string[] - Devuelve las palabras que tienen un operador '!'.
- **MultipliedWords**: (string, int)[] - Devuelve las palabras que tienen operadores '*', junto con la cantidad que tiene cada una.
- **CloserWords**: List<string>[] - Devuelve los grupos de palabras que están unidas por '~'. Por ejemplo, si la query es "estudiamos ~ computación pero nadie ~ quiere ~ suspender", el conjunto devuelto sería {{estudiamos, computación}, {nadie, quiere, suspender}}.

Métodos:

- **PushOperator(char)**: Asocia un nuevo operador entre "*!^" a la próxima palabra a analizar. Una palabra puede tener más de un operador asociado.
- **PushTilde()**: Asocia un '~' a la última palabra que se haya analizado y la siguiente que se analizará. Si el usuario introdujo uno antes de todas las palabras, este será desechado. Si se agregó más de uno entre dos palabras, solo se almacenará uno.
- **TrimEnd()**: Elimina los operadores que el usuario pueda haber escrito al final.

Cuando se llama al constructor de **ParsedInput**, pasándole la query como parámetro, este la recorre caracter a caracter y según su tipo los va insertando en las propiedades correspondientes del objeto. Al final llama a su **TrimEnd()** para deshacerse de cualquier operador sobrante.

Clase PartialItem: Es una forma minimalista de guardar un resultado de búsqueda temporal. Una instancia representa que cierta palabra existe en un documento.

Propiedades:

- **Word**: string - La palabra que se buscó.
- **Document**: int - Un documento donde esta palabra existe. El valor almacenado es el ID del documento.
- **Multiplier**: float - Un multiplicador que se debe aplicar sobre el TF-IDF de la palabra en el documento. Su valor por defecto es 1.0. Este valor varía si la palabra trae operadores "*~", o si se trata de una sugerencia, sinónimo, o palabra de igual raíz.
- **Original**: string - En caso de que **Word** haya sido generado por una sugerencia, en **Original** se guardará la palabra original de la cual la sugerencia se derivó. Su valor por defecto es un string vacío "".

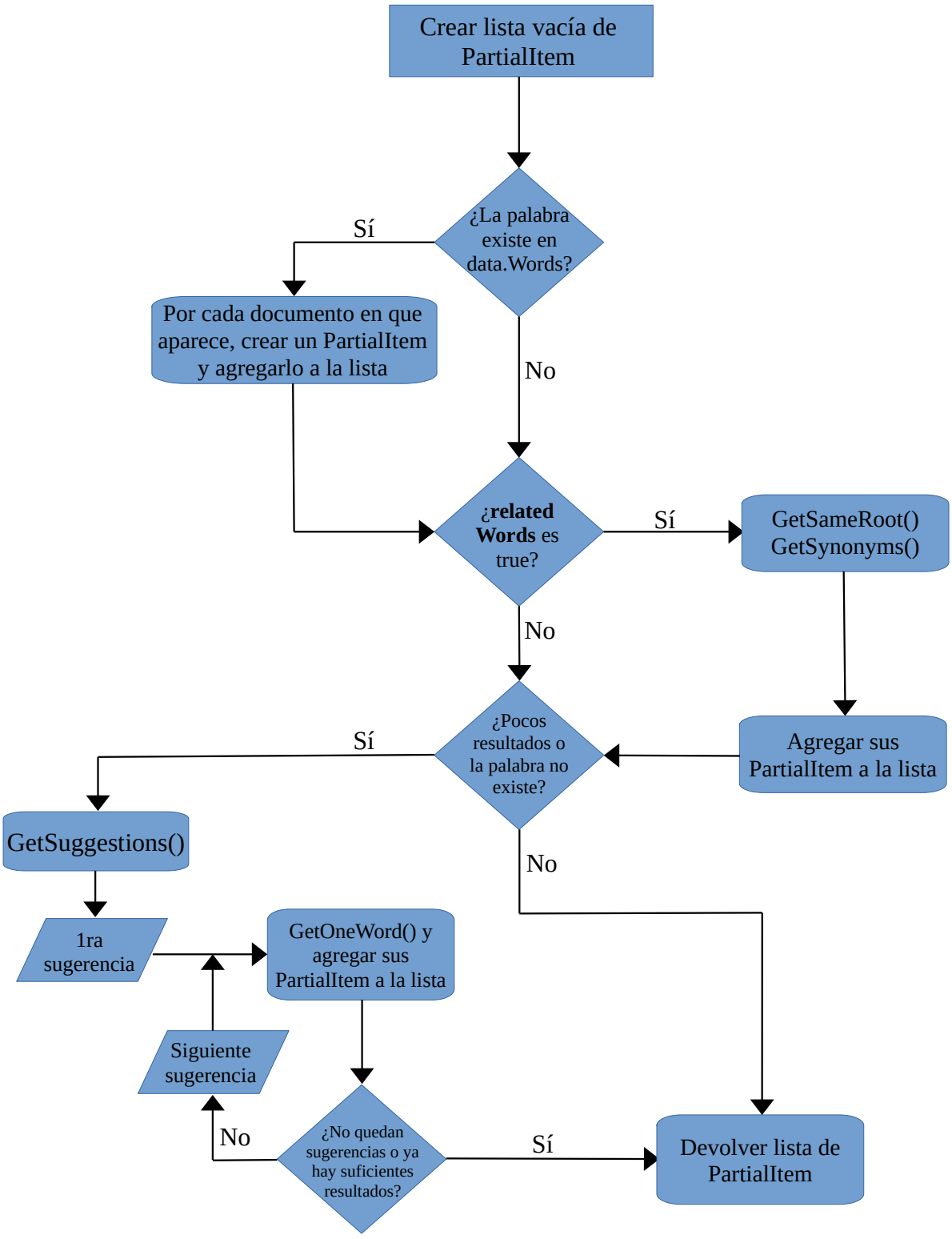
Método GetOneWord(): Recibe una palabra y devuelve los **PartialItem** asociados a esta. Si la palabra trae un operador '!', no se generarán sugerencias, sinónimos o palabras de igual raíz.

Parámetros:

- **data**: IndexData - La instancia de los datos indexados de los documentos.
- **word**: string - La palabra que se desea buscar en los documentos.
- **suggest**: bool - Si se desea generar los **PartialItem** de las sugerencias de la palabra o no.
- **multiplier**: float - El multiplicador asociado a esta palabra. Los **PartialItem** que genere se guardarán con este multiplicador.

- **original**: string - La palabra de la que se derivó la actual. String vacío por defecto.
- **relatedWords**: bool - Si se desea generar los **PartialItem** de los sinónimos de **word** y palabras de la misma familia o no.

Diagrama de GetOneWord():



Variables static relacionadas con GetOneWord():

- **minAcceptable**: int - La cantidad mínima de resultados que debe generar una palabra (incluyendo sus sinónimos y las de misma raíz) para que no se haga necesario buscar sus sugerencias. Valor por defecto: 3.
- **suggestionsByWord**: int - La cantidad máxima de sugerencias que se buscarán para una palabra. Valor por defecto: 3.
- **resultsWithSuggestion**: int - Si las sugerencias analizadas ya han generado al menos esta cantidad de resultados, no se generarán los **PartialItem** de las demás. Valor por defecto: 10.

Búsqueda de sugerencias:

El método **GetSuggestions()** recibe la instancia de **IndexData** con los datos de los documentos, y la palabra de la que se desea generar sugerencias. Devuelve una List<(string, float)>, donde cada elemento es una sugerencia junto a su multiplicador. El procedimiento consiste en iterar por todas las palabras de igual longitud a la palabra pasada en los argumentos. Luego itera por todas las palabras de longitud +1 y -1, luego las de longitud +2 y -2, y así sucesivamente (la variable estática **maxCharDiff** contiene la diferencia máxima de longitud a analizar, por defecto está en 2). Para buscar la primera palabra de la longitud deseada se usa el método **GetLengthInDict()**, el cual usa una búsqueda binaria para hallar la posición deseada en el diccionario. Entonces, por cada palabra analizada, se calcula la distancia de Levenshtein entre esta y la palabra original. Si esta distancia no supera el valor de **maxDistance** (por defecto 4), se agregará a una lista de posibles candidatos a sugerencias, con un multiplicador que depende inversamente de dicha distancia. Finalmente, la lista es ordenada descendentemente y se seleccionan las sugerencias con mayor multiplicador. Estas serán retornadas por la función.

Obtención de palabras con la misma raíz:

El método **GetSameRoot()** recibe la instancia de **IndexData** y la palabra de la que se generarán raíces, y devuelve una lista con los **PartialItem** generados por las palabras obtenidas. Primeramente se llama a **Stemming.GetRoot()** para obtener la raíz de la palabra. Luego se accede a **data.Roots[raiz]** (si esta existe) y se itera por cada una de las palabras que tienen esa misma raíz. Por cada una de estas palabras se llamara a **GetOneWord()**, pasándole un multiplicador que depende inversamente de la distancia entre la palabra original y la que se está analizando. Además los parámetros **suggest** y **relatedWords** se pasarán como 'false'.

Obtención de sinónimos:

El método **GetSynonyms()** recibe los mismos parámetros que el anterior, y devuelve una lista con los **PartialItem** de los sinónimos. Lo primero que hace es obtener la raíz de la palabra con **Stemming.GetRoot()**, e itera por cada palabra que tiene esa raíz. Luego, intenta buscar cada una de esas en **data.Synonyms**, si esta ahí, agrega todos los sinónimos a una lista.
Al terminar, itera por cada sinónimo en la lista, halla su raíz, recorre todas las palabras que tienen esa misma raíz y llama a **GetOneWord()** con cada una. Este **GetOneWord()** recibirá un multiplicador de valor 0.001, así como los parámetros **suggest** y **relatedWords** en 'false'. Todos los **PartialItem** obtenidos aquí serán retornados por la función.

El método DocsFromPhrase():

Este método recibe la instancia de **IndexData**, la lista de **PartialItem** obtenida anteriormente, el objeto **ParsedInput** de la entrada, un entero que representa la cantidad máxima de resultados que se devolverán para ser mostrados en el Front-end, y una lista de **PartialItem** vacía donde se guardarán las palabras que se obtuvieron por sugerencias. Dicha lista se usará desde la clase **Moogle**, aprovechando que esta se pasa por referencia. Este método devuelve una lista de **CumulativeScore**.

Clase CumulativeScore: Esta clase representa la relevancia total que tiene un documento según la query introducida por el usuario.

Propiedades y métodos:

- **TotalScore**: float - Es la suma de los TF-IDF de todas las palabras relacionadas al query que aparecen en el documento.
- **Content**: List<PartialItem> - Son todos los **PartialItem** que generaron las palabras relacionadas a la query que aparecen en este documento.
- **AddWord(float, PartialItem)** - Este método recibe el **PartialItem** de una palabra en este documento, junto a su TF-IDF. Agrega dicho **PartialItem** a **Content** y le adiciona a **TotalScore** el TF-IDF introducido, pero multiplicado por el multiplicador del **PartialItem**.

Funcionamiento de DocsFromPhrase():

Lo primero que hace este método es recorrer todos los **PartialItem** buscando los que se hayan obtenido por sugerencias, y los inserta en la lista que se pasó en los parámetros. Luego se filtra la lista de **PartialItem** que recibe según los operadores. Para esto llama al método **FilterByOperators()** pasándole la instancia de **IndexData**, la lista de **PartialItem**, y el objeto **ParsedInput** con la query de entrada. Este método devuelve una nueva lista de **PartialItem**, y su funcionamiento será analizado en detalle más adelante.
Luego del filtrado, se crea un diccionario de <int, CumulativeScore>. En este, la Key representa el ID de un documento, y este apuntará a un objeto **CumulativeScore** que contendrá todos los **PartialItem** asociados a este documento. Para esto se itera por cada uno de los **PartialItem** obtenidos tras el filtrado. Se accede a la propiedad **Document** de cada uno, y entonces, este **PartialItem** es agregado al Value asociado al documento de igual ID. Por supuesto, el valor **TotalScore** de dicho **CumulativeScore** también irá aumentando.
Por último, el diccionario de **CumulativeScore** es ordenado descendentemente según el **TotalScore** de cada uno. Luego, se itera por cada uno para eliminar sus Key y guardar cada **CumulativeScore** en una lista. Esta lista será devuelta por la función, y en ella cada elemento representa uno de los documentos que se le mostrará al usuario en el Front-end, ya ordenados.

El método FilterByOperators():

Lo primero que hace este método es llamar a las propiedades get del objeto **ParsedInput** (**MandatoryWords**, **ForbiddenWords**, **MultipliedWords**, **CloserWords**) y guardar sus resultados en objetos del tipo correspondiente. Luego se crea una lista vacía de **PartialItem** donde se irán guardando los documentos filtrados. También se crea un diccionario <int, float> para hacer un memoization de los documentos, debido a que un mismo documento probablemente aparezca en varios **PartialItem**. Esto permitirá evitar recalcular cada documento. Para cada documento se

almacenará un multiplicador. Si este es 0, significa que el documento fue descartado, por lo que no se agregará a la lista de resultados. Si no es 0, el multiplicador del **PartialItem** será multiplicado por este valor y luego se agregará a la lista de resultados.
Para realizar el filtrado se itera por cada **PartialItem**. Si el documento al que representa no ha sido analizado, se evalúa cada operador de la siguiente manera:

- **Operador de prioridad '*'**: Se recorre cada palabra obtenida al llamar a **MultipliedWords**, si la palabra del **PartialItem** es la misma que estamos analizando, se multiplica le asigna a la variable **priorityMult** un valor igual a la cantidad de * más 1.
- **Operador de inclusión '^'**: Se recorre cada palabra obtenida al llamar a **MandatoryWords**. Si el documento no contiene dicha palabra es descartado y memoizado con un multiplicador de 0.
- **Operador de exclusión '!'**: Se recorre cada palabra obtenida al llamar a **ForbiddenWords**. Si el documento contiene dicha palabra es descartado y memoizado con un multiplicador de 0.
- **Operador de cercanía '~'**: El funcionamiento de este operador es considerablemente más complejo que los anteriores, por lo que se debe profundizar mucho más en él. Para empezar, es necesario definir lo siguiente.

Clase WordPositions: El propósito de esta clase es recibir un conjunto de palabras junto con sus listas de apariciones en un mismo documento, y organizar todas estas ocurrencias en una estructura.

Propiedades y métodos:

- **Positions**: SortedSet<int> - Guarda todas las posiciones ocupadas por las palabras deseadas.
- **Word**: Dictionary<int, string> - El Key representa una posición en el documento, y esta apunta a la palabra que se encuentra ahí
- **Differents**: int - Devuelve la cantidad de palabras diferentes que se han guardado. Se apoya en un **HashSet** privado.
- **Insert(string, int[])**: Recibe una palabra y un arreglo con las posiciones de esta en el documento, y almacena estos datos en las estructuras anteriores.

Método SnippetOperations.GetZone():

Este método recibe un int representando una posición en el documento, un objeto **WordPositions** con la información de ciertas palabras en el documento, y un entero representando un diámetro cuyas unidades son posiciones. La utilidad de este método es devolver la cantidad de palabras diferentes (de las guardadas en el objeto **WordPositions**) que se encuentran en el diámetro pedido. Para hacer esto primero calcula los límites izquierdo y derecho cuyo diámetro sea el deseado. Luego se auxilia del método **GetViewBetween()** de la clase **SortedSet** para obtener las palabras que se encuentran en el intervalo comprendido entre los límites calculados. Después, apoyándose en un **HashSet**, cuenta la cantidad de palabras distintas en el intervalo, y retorna este valor.

Funcionamiento del operador de cercanía:

Este algoritmo itera por cada uno de los grupos de palabras relacionadas por '~' que se obtuvieron con el getter **CloserWords** de la instancia de **ParsedInput**. Es así que por cada lista de palabras se hará lo siguiente:

Se recorre cada una de las palabras del conjunto a la vez que se accede al objeto **Occurrences** de cada una en el documento que está siendo analizado. Si dicha palabra existe en el documento, se toma su propiedad **StartPos** guardada en su **Occurrences**, y dichas posiciones se insertarán en un objeto **WordPositions**. Luego de iterar por las palabras, este objeto contendrá las posiciones de todas las palabras del conjunto que aparecen en el documento. En este punto se accede a la propiedad **Differents** del objeto **WordPositions** para saber la cantidad de palabras distintas del conjunto que aparecen en el documento. Si es solo una o ninguna, omitiremos este conjunto de palabras. En caso de que la condición anterior se incumpla, se ejecuta el siguiente algoritmo:
Tenemos la variable estática **closerDiameter** la cual es un array de int donde se encuentran ordenados de menor a mayor los diámetros donde se buscarán las palabras cercanas (representando posiciones en bytes, los valores por defecto son 50, 100, 150, 300, 600). Comenzaremos recorriendo todos los diámetros. Por cada uno de estos iteramos por cada posición guardada en el **WordPositions**. Si la palabra es poco relevante y existen palabras relevantes en el conjunto, nos la saltamos. En caso contrario, llamaremos al método **SnippetOperations.GetZone()** pasándole la posición actual como pivote, el objeto **WordPositions** y el diámetro actual. Con esto sabremos la cantidad de palabras diferentes que existen en este diámetro alrededor de la posición. Con esto calculamos un posible valor para el multiplicador del conjunto, el cual depende directamente de la cantidad de palabras en el diámetro, e inversamente de la amplitud de dicho diámetro
A medida que ejecutamos lo anterior por cada posición, y en cada diámetro definido, nos iremos quedando con el mayor multiplicador que aparezca. Como medida de optimización, si en el recorrido nos encontramos con algún diámetro que contiene todas las palabras del conjunto que aparecen en el documento, usaremos el multiplicador de ese diámetro y dejaremos de buscar en este conjunto, puesto que es imposible que aparezca otro diámetro de mejor score.
Finalmente, multiplicaremos los scores de cada conjunto, y el valor obtenido será el multiplicador que nos darán los operadores '~'. Este valor se agregará al multiplicador del **PartialItem** actual, a la vez que se agrega a la lista de memoization.

El método GetResults():

Este método recibe la instancia de **IndexData**, la lista de **CumulativeScore**, el objeto **ParsedInput** con la query de entrada, y la lista de **PartialItem** con las sugerencias obtenidas anteriormente; y devuelve un objeto del tipo **SearchResult** que será enviado a la interfaz gráfica. Al inicio, este método crea una lista de **SearchItem** donde se irán almacenando los resultados en formato final de cada documento.
Luego de esto, se recorre cada elemento del **CumulativeScore** y se toman algunos datos como el ID del documento, su score total, su título y su ruta. Acto seguido se accede a su propiedad **Content** y se recorre cada **PartialItem** que aparezca ahí. Si la palabra que representa es poco relevante (y el documento contiene palabras relevantes) nos la saltaremos. Si no, accedemos a la lista **StartPos** de esta palabra en el documento actual, y la insertamos en un objeto **WordPositions**.
Una vez revisados todos los **PartialItem** de este **CumulativeScore**, llamamos al método **SnippetOperations.GetSnippet()**. Dicho método recibe la ruta de un documento y el objeto **WordPositions** asociado a este que recién se generó; para luego devolver el snippet que se mostrará. Este método recorre cada posición enviada en el objeto de posiciones, y se auxilia de **GetZone()** para encontrar el diámetro que más palabras diferentes contiene (el valor del diámetro está definido en la variable **snippetWidth**, por defecto 600). Una vez obtenido el pivote para el snippet, se calcularán los límites izquierdo y derecho, y luego se cargará el documento en ese intervalo, para ser devuelto como un string. Antes de devolver este snippet se llama al método **TrimSnippet()**. Dado que estamos trabajando con las posiciones en bytes, es posible que al escoger los límites se haya picado un caracter especial, lo cual mostraría un error visual en la interfaz gráfica. Este método se asegura que en los bordes del snippet solo hayan caracteres alfanuméricos.
Nota: Si un documento solo tiene palabras poco relevantes, no se buscará su intervalo más óptimo, ya que esto puede ser lento. Simplemente se escogerá el primer intervalo que contenga una palabra de la búsqueda.
Luego de todo esto solo queda generar el string de sugerencias, el cual es el mostrado tras el mensaje "¿Quisiste decir ...?". Para esto usaremos el método **GenerateSuggestionString()**, el cual recibe el **ParsedInput** del query, y la lista de **PartialItem** con las palabras resultantes de sugerencias que teníamos. La idea tras este método es sencilla: Tendremos un diccionario de <string, (string, int)> en el cual, para cada palabra original que se modificó, guardaremos su mejor sugerencia junto a la distancia entre ambas palabras. Para determinar la mejor sugerencia de cada palabra, simplemente nos guiaremos por la que tenga menor distancia con la palabra original. Esto lo hacemos con un recorrido por cada **PartialItem**, y vamos escogiendo la mejor sugerencia a medida que iteramos.
Una vez obtenida la sugerencia a mostrar para cada palabra, recorremos este diccionario, y buscamos cada palabra en el string del query original y la reemplazamos (para esto nos auxiliaremos de **ArraysAndStrings.Find()**, método simple que localiza la posición de un string en un array de strings). Al acabar este paso devolvemos la query modificada como un string (usando el método **ArraysAndStrings.WordsToString()**).
Cuando todo esto haya concluido, solo queda crear un nuevo **SearchResult** desde el final de **GetResults()** con todos los datos que se han obtenido. Este objeto es enviado al Front-End por el método **Moogle.Query()**, y así se da por terminada una búsqueda.

Bibliografía

1. **C# Documentation**:
   https://docs.microsoft.com/en-us/dotnet/csharp/

2. **Understanding TF-IDF: A Simple Introduction**:
   https://monkeylearn.com/blog/what-is-tf-idf/

3. **TF-IDF**:
   https://es.wikipedia.org/wiki/Tf-idf

4. **Defining R1 and R2**:
   http://snowball.tartarus.org/texts/r1r2.html

5. **Spanish stemming algorithm**:
   http://snowball.tartarus.org/algorithms/spanish/stemmer.html

6. **Edit distance | DP-5**:
   https://www.geeksforgeeks.org/edit-distance-dp-5/