Abstract

Implementación básica del Modelo Vectorial.

Modelo Vectorial

Memoria. Parte 1.

Victor De Gouveia - 100333516

Carlos Mayoral Lorenzo - 100284050

Kevin Geeting - 100370975

# Introducción

El objetivo de esta práctica ha sido crear un modelo vectorial aplicado a la búsqueda y recuperación de documentos web, específicamente documentos HTML, simulando de esta manera el funcionamiento que tienen los grandes motores de búsqueda como Google. El proyecto se ha implementado usando Java como lenguaje de programación, y se han incorporado diferentes métodos para la obtención de resultados usando el modelo vectorial, como por ejemplo el uso de las funciones del coseno del ángulo entre los vectores o el producto escalar entre ellos.

# Perspectiva General

En este apartado, se detallará desde una vista de alto nivel, el funcionamiento del modelo vectorial que ha sido desarrollado.

Antes de empezar, resaltaremos que el uso de los modelos vectoriales aplicados a la recuperación de información sirve para poder satisfacer la necesidad de información de un usuario (representado mediante una consulta), devolviéndole un conjunto de documentos que tienen altas probabilidades de satisfacer la necesidad mencionada. En otras palabras, funcionan para poder establecer una relación entre la consulta de una persona con los documentos que puedan satisfacer esa consulta, y de esta manera devolverle al usuario los documentos más relevantes para él.

Primero que nada, debemos definir el dominio sobre el cual estamos trabajando, descomponiendo los diferentes elementos que tendrán un rol vital en el funcionamiento del modelo. En muy resumidas cuentas, el modelo vectorial funciona de la siguiente manera:

1. Existen tanto documentos como consultas en el dominio, ambos compuestos por términos.
2. Los términos mencionados anteriormente tienen asociados una serie de pesos. Los diferentes pesos pueden ser obtenidos de diferentes maneras (peso como TF o TF\*IDF, por ejemplo).
3. Tanto los documentos como las consultas deben poder representarse como vectores. Estos vectores tienen dimensiones, donde .
4. Se aplica una función de similitud sobre la consulta y cada documento para determinar la medida de relevancia que hay entre ellos. Esta función de similitud puede ser de varios tipos.
5. Con la medida de relevancia podremos saber cuales de los documentos dentro del conjunto analizado son los más apropiados para resolver la necesidad de información.

Esta ha sido una explicación muy básica de como funciona el modelo vectorial que hemos desarrollado, y la meta de esta explicación es dar al lector un poco de contexto y conocimiento base antes de proceder a detallar las decisiones de diseño que se han tomado a la hora de desarrollar el modelo.

# Componentes Desarrollados

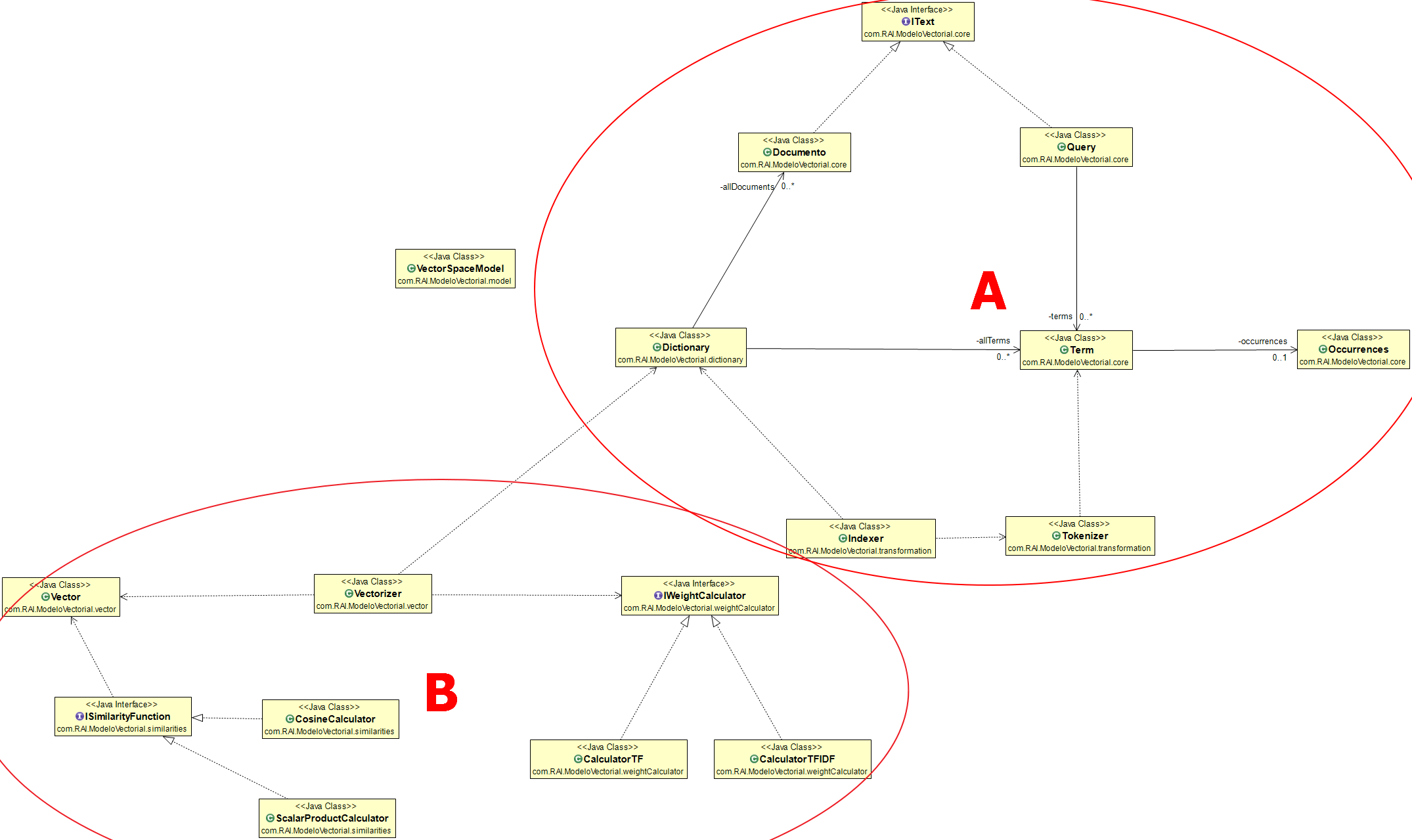
Para la implementación del modelo vectorial, hemos intentado mantener todo lo más modular posible. Para lograr esto, hemos descompuesto los elementos que participan en el dominio del modelo vectorial, y luego los hemos agrupado en componentes dependiendo de su funcionalidad. A continuación, se presenta una tabla con los diferentes componentes (modelados como *Packages* en Java), una breve descripción y las clases que los componen:

|  |  |  |
| --- | --- | --- |
| **Componente** | **Descripción** | **Clases/Interfaces** |
| **Core** | Agrupa las entidades imprescindibles y atómicas para el funcionamiento del modelo vectorial, como por ejemplo documento y consulta | IText  Documento  Query  Term  Occurrences |
| **Dictionary** | Contiene la clase que representará el índice. Es la estructura que almacenará la información relacionada al *Corpus* de documentos y términos, sobre el cual se harán las consultas. | Dictionary |
| **Transformation** | Componente encargado de realizar transformaciones de texto. Necesario para filtrar y normalizar los términos que entran al índice. | Tokenizer  Indexer |
| **WeightCalculator** | Componente encargado de obtener los pesos en sus diferentes modalidades de los términos. | IWeightCalculator  CalculatorTF  CalculatorTFIDF |
| **Vector** | Componente encargado de representar una consulta o documento como un vector de pesos de cada uno de sus términos. | Vector  Vectorizer |
| **Similarities** | Componente encargado de emplear las diferentes funciones de similitud entre vectores. | ISimilarityFunction  CosineCalculator  ScalarProductCalculator |
| **Model** | Componente encargado de representar el modelo vectorial como un entero. Dentro de este componente se encuentra el punto de entrada; el método Main. | VectorSpaceModel |

Para más información sobre los diferentes componentes, existen diagramas de componente en la ruta del proyecto **..\ModeloVectorial\_4\Documentation.** Cada componente tiene su propio directorio, con una vista a los diagramas de clase dentro de ese componente. Hay un directorio llamado **Overview** que contiene diagramas que abarcan una vista macroscópica de todo el proyecto.

Para la explicación del modelo, dividiremos el sistema en dos grandes macro-componentes.

El macro-componente A abarcará los componentes Core y Dictionary, que tienen que ver con el funcionamiento básico del modelo, mientras que el macro-componente B tendrá en cuenta el resto de componentes salvo el de Modelo, ya que estos tienen en cuenta funcionalidades auxiliares como filtrar, indizar, vectorizar, etcétera.



# Macro-Componente A: Core y Dictionary

Tomando en cuenta el proceso explicado en la *Perspectiva General* de este documento, una de las cosas más importantes para poder desarrollar un modelo vectorial que funcione adecuadamente es poder conceptualizar los elementos y estructuras que van a ser necesarias. A continuación, nos fijaremos en el subconjunto del diagrama de clases correspondiente al macro-componente A que se enfoca en las relaciones que hay entre los denominados elementos base y el diccionario.

Como se ha mencionado anteriormente, tanto las consultas **(clase Query)** como los documentos **(clase Documento)**, comparten ciertas características como el hecho de tener la capacidad de ser representados como vectores, además de que tanto los documentos como las consultas esencialmente son texto, y por esta razón ambas clases implementan una interfaz común denominada IText.

Por otro lado, a pesar de haber mencionado anteriormente que tanto los documentos como las consultas están compuestos por diferentes términos **(clase Term),** la clase Documento no tiene una relación de agregación con la clase Term. Es decir, desde una visión estrictamente conceptual y programática, los documentos en este diagrama **no** están compuestos por términos. Esta es una de las decisiones de diseño más importante que hemos tomado, y la razón de ser es la siguiente: dado que los términos de un documento se pueden obtener mediante diferentes métodos a posteriori (métodos encontrados en la clase **Indexer**, que aún no hemos mencionado), es un poco redundante tener que almacenar todos los términos de un documento, que puede llegar a ser una cantidad abrumadora, dentro de su objeto. Al almacenar la dirección en donde se encuentra el documento, somos capaces de extraer eso términos y almacenarlos en otro lugar, pero nunca dentro del mismo objeto **Documento**, porque sería redundante y solo ocuparía más espacio de lo que debería en memoria. En resumen, con solo tener la dirección del documento dentro del sistema, luego seremos capaces de extraer los términos que lo componen con el uso de otra clase, y ahorrarnos repetir información dentro del sistema y ser más eficientes.

Por otro lado, el razonamiento del por qué no hemos seguido esta misma lógica para las consultas es que generalmente las consultas son bastante cortas con respecto a los documentos. Las consultas generalmente contienen un par de términos, y estas se pueden escribir directamente, no como los documentos que pueden tener una cantidad de términos que supera los miles. Además de esto, como se explicará más adelante, los objetos **Query** poseen un conjunto de términos que debe ser diferente al conjunto de términos que se han indizado, ya que los términos de un objeto **Query** tienen sus propios pesos, independientes de aquellos pesos almacenados en el índice. Este índice al que nos referimos es la estructura que almacena la información pertinente a todos los términos y documentos que se han almacenado en el sistema y se encuentra en la clase **Dictionary.**  Tiene información como, por ejemplo:

1. La frecuencia del término, o , que nos dice cuantas veces aparece el término en el documento .
2. El IDF del término, que es un valor global con respecto a todo el índice. Se refiere al , y es una medida del poder de discriminación de un término.

Este índice está almacenado dentro de la clase **Dictionary**, que como podemos ver en el diagrama de clases, contiene un conjunto de objetos **Documento** y un conjunto de objetos **Term**. La clase **Dictionary** es fundamental para el funcionamiento del modelo. Es aquí donde se indizan los diferentes documentos y términos, creando una especie de *Corpus* de documentos, sobre el cual podemos realizar las consultas.

Por último, podemos observar la existencia de la clase **Occurrences**, que es una agregación a la clase **Term**. En otras palabras, cada término tiene una cantidad de ocurrencias en los diferentes documentos presentes en el índice o en la consulta. Esta cantidad de ocurrencias en cada documento es importante para obtener tanto el del término como el .

***Nota:*** es importante resaltar que, dado que los documentos que se tratan son documentos web y estos tienen varias etiquetas HTML, se ha utilizado la librería *Jsoup* para dejar el texto limpio.

# Macro-Componente B: Componentes Transformation, WeightCalculator, Vector y Similarities

Este componente es el que se encarga de tratar la información antes de pasar a indizarla dentro del objeto **Dictionary,** y posteriormente realizar cálculos sobre ella para poder obtener las medidas de relevancia. Para resumir un poco el funcionamiento de este componente, dado que es bastante extenso y el presente documento tiene un límite de 8 páginas, identificaremos los pasos por los cuales pasa la información antes de ser indizada:

1. Los documentos que se quieren indizar primero pasan por el componente **Transformation**, que gracias a las clases **Tokenizer** e **Indexer**, son separados en distintos tokens y filtrados. El proceso de filtrado procesa los diferentes tokens, eliminando las denomiandas *stop words* (a, the, and, or, …)**,** y realizando *stemming.* Este componente hace uso de la librería externa Lucene, dado que los algoritmos de filtrado ya se encuentran disponibles ahí.
2. La clase **Indexer** indiza un conjunto de documentos en el objeto **Dictionary**, término por término, donde se ajustan los datos correspondientes a el TF e IDF de cada uno de ellos. El diccionario realiza un conjunto de checks para determinar que hacer en cada caso (término ya existente, documento nuevo, etc).

Una vez indizado los documentos y términos en el diccionario, se puede proceder a procesar las consultas sobre ese índice y obtener las métricas de similitud.

Existen dos métricas de similitud dentro del componente **Similarities:** similitud con ángulo del coseno entre dos vectores y similitud con producto escalar de dos vectores. Ahora, lo interesante está en que realmente el modelo debe ser capaz de obtener cuatro métricas diferentes:

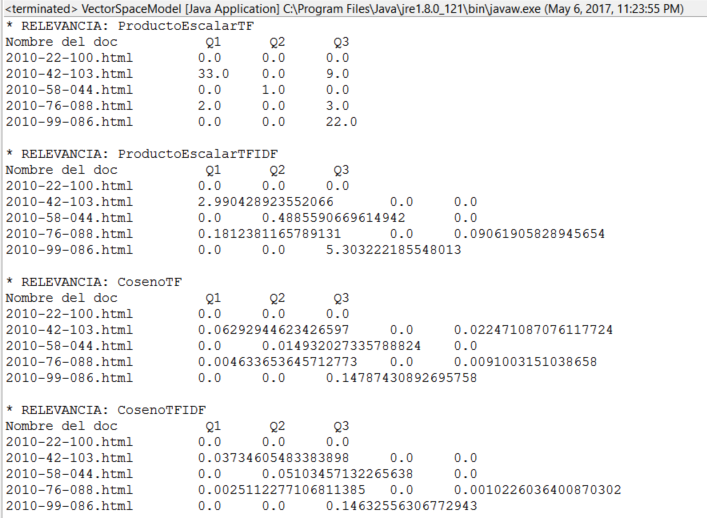
1. El ángulo del coseno de dos vectores con pesos TF
2. El ángulo del coseno de dos vectores con pesos TF\*IDF
3. El producto escalar entre dos vectores con pesos TF
4. El producto escalar entre dos vectores con pesos TF\*IDF

Lo que hemos diseñado para lograr esto, es que los objetos **Vector** sobre los cuales el componente **Similarities** trabaja, puedan ser obtenidos de dos maneras diferentes. Dado que la clase **Vectorizer**, encargada de crear objetos **Vector**, toma una interfaz de calculadora de peso **(IWeightCalculator)**, somos capaces de pasarle o bien una instancia de **CalculatorTF**, o **CalculatorTFIDF.** Dependiendo de cual calculadora pasemos, obtendremos un vector cuyos pesos son en base al o un vector cuyos pesos son en base al .

Teniendo dos posibles tipos de vectores (TF o TF\*IDF), las clases **CosineCalculator** y **ScalarProductCalculator** del componente **Similarities** son capaces de calcular las cuatro métricas que se piden.

# Resultados

Los resultados obtenidos al ejecutar la clase **VectorSpaceModel** dentro del componente **Model,** que contiene el método Main son los siguientes:



Resumiremos la explicación de estos resultados brevemente. Podemos ver que para el primer documento **2010-22-100.html,** las cuatro funciones de similitud arrojan 0 para todas las consultas. Esto se debe a que el primer documento no tiene nada que ver en lo absoluto con las consultas, y los términos que se encuentran dentro de las consultas no aparecen en el documento, causando que los cálculos den 0. Para el resto de los documentos empezamos a obtener resultados variados.

Los resultados del segundo documento **2010-42-103.html** son interesantes. Podemos ver que para las métricas que toman en cuenta el TF y la tercera consulta, tenemos resultados diferentes de 0, pero para las métricas que usan el peso TF\*IDF, nos devuelve 0. Esto se debe probablemente a que, para el segundo documento, todos los términos de las consulta aparecen en todos los otros documentos, entonces son términos que no tienen el poder de discriminación, dado que son comunes en todos los documentos.

# Mejoras Implementadas

1. **Índice en base de datos.** Se ha utilizado la base de datos SQLite dado que esta no requiere ningún servidor ni SGBD porque se encuentra integrada a la aplicación Java. En los diagramas de clase mostrados en este documento no aparecen las clases desarrolladas para lograr esto, pero son las clases **Crawler** (inserta todos los documentos web encontrados en un directorio a la BD, guardando su filepath) y **Connector** (clase que controla la lógica para operar en la BD). Al final, el índice se guarda en la BD mediante el objeto **Dictionary** que llama a los métodos de **Connector** mientras se indiza el *Corpus,* y luego se hace un clear sobre las estructuras del **Dictionary** para no tener esa información repetida en RAM, dado que ya se encontraría en almacenamiento.
2. **Tokenizer y normalización de términos.** Se ha utilizado la librería externa *Jsoup* para quitar todas las etiquetas HTML de un documento, y luego se han utilizado clases de la librería externa *Lucene* para obtener los tokens separados como tal y realizar el filtrado. Gracias a esto, palabras como *Teclado* y *teclados* son indizados como la misma palabra (en inglés, claro).
3. **Separar el cálculo de pesos de la función de similitud.** Como ha sido explicado anteriormente, hemos seguido el patrón de diseño *estrategia* para poder obtener esto. El calculo de pesos se realiza dentro del componente **WeightCalculator**, siendo posible obtener los pesos en base a TF o en base a TF\*IDF, y las funciones de similitud se encuentran dentro del componente **Similarities**, podiendo calcular usando el coseno del ángulo entre dos vectores o el producto escalar.