Session 2: Programming with ElasticSearch

Q1 2022-23

Integrantes:
Pol Pérez Castillo
Daniel Ruiz Jiménez

Fecha de entrega: 2-10-2022

Índice

Introducción

En la segunda sesión del curso vamos a seguir programando ElasticSearch para aplicar diferentes filtros al crear un índice. Los tokenizadores separarán palabras según diferentes criterios, y los filtros usarán algoritmos para realizar la misma tarea. Nuestra misión será realizar muchos experimentos combinando filtros y tokenizadores sobre el conjunto de textos literarios, para ver qué combinación tiene el menor número de palabras diferentes. Más tarde implementaremos un programa que, usando el cociente tf-idf, nos permitirá calcular la similitud de 2 documentos.

Preparación previa

Al igual que con la sesión anterior, nos descargamos los 3 conjuntos de documentos: textos literarios, científicos y coloquiales. Nuestro objetivo será estudiar los literarios en esta primera parte de la sesión. Hemos usado el fichero *IndexFilesPreprocess.py* para crear múltiples índices, cada uno con un tokenizador diferente usando las opciones de ejecución —token y —filter:

- Standard: El más básico, divide en términos por fronteras de palabras. Además elimina los signos de puntuación.
- Letter: Se divide por signos de puntuación diferente a una letra. (, ! ? - ...).
- Whitespace: Divide palabras al encontrar un espacio.
- Classic: El más adecuado para el inglés, se divide por palabras que el tokenizador reconoce.

A continuación hemos ido añadiendo diferentes filtros, acumulando los que no entraran en conflicto.

- Lowercase: Aplicado por defecto, pasa todas las palabras a minúsculas.
- Asciifolding: Elimina las palabras con carácteres ASCII poco usados, considerados “extraños” por el filtro.
- Stop: Quita las stopwords del inglés (más adelante veremos que tiene sólo 33 stopwords registradas).
- Snowball, Porter_stem y Kstem: Cada uno cuenta con un algoritmo propio para el filtrado de palabras.

Modificación de los índices mediante tokenizers y filters

Una vez obtenidos los respectivos outputs con el número de palabras diferentes tokenizadores y filtros, hemos comparado los resultados por cada filtro diferente.

Lowercase+Asciifolding

Para la primera comparación, hemos utilizado tanto el filtro lowercase como el Asciifolding, y el resultado ha sido el siguiente:

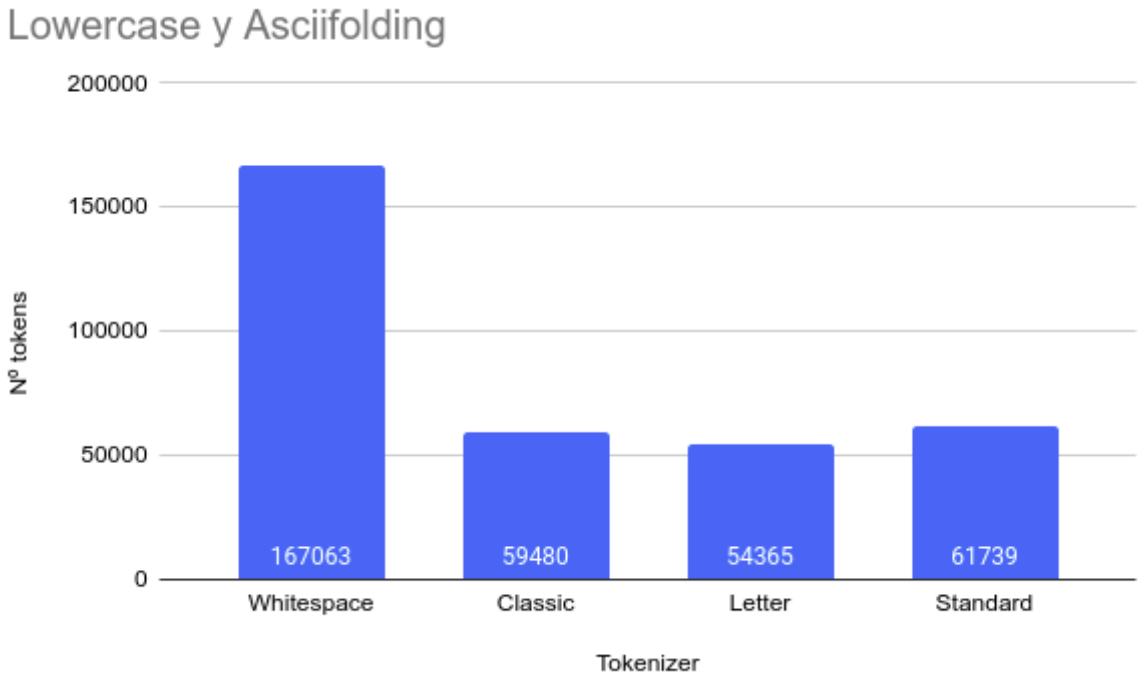

Imagen 1: Gráfica lowercase y asciifolding

Como podemos observar, el tokenizador Letter es el más agresivo, puesto que elimina palabras que contengan carácteres diferentes de una letra del alfabeto. Por otra parte, el whitespace es el menos restrictivo, ya que su condición para separar palabras son los espacios en blanco. Es por este motivo que nos fijaremos sobretodo en el tokenizador Letter para ver la mejoría de los filtros, el resto pasarán a tener carácter secundario.
La palabra más frecuente en el conjunto es “the”.

Lowercase+Asciifolding+Stop

La segunda comparación no cambia mucho respecto a la primera. Hemos añadido el filtro Stop, y lo que ha hecho ha sido quitar 33 stopwords exactas para cada tokenizador. El propio tokenizador cuenta con una base de datos con 33 stopwords que retirar, pero es customizable, por lo que podríamos añadir manualmente las stopwords que consideremos. El resultado no ha variado mucho más, Letter sigue siendo nuestra mejor opción. No obstante, ahora la palabra más frecuente no es "the", sino "i".

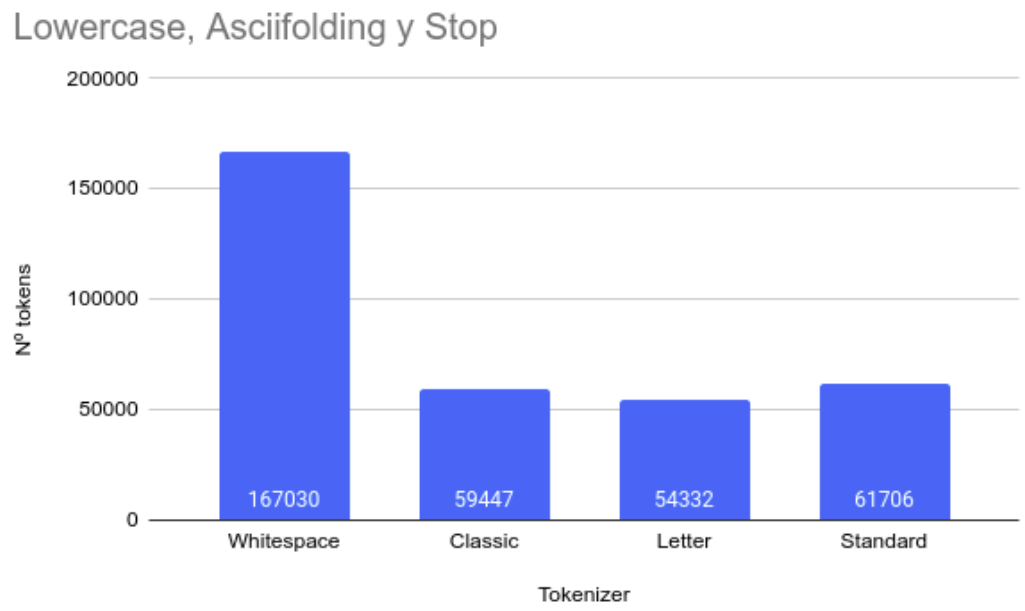

Imagen 2: Gráfica lowercase, asciifolding y stop

Lowercase+Asciifolding+Stop+Snowball

Ahora pasaremos a añadir sólo un filtro a la vez, y veremos cuál de los siguientes 3 filtros es la mejor opción para nuestro conjunto de documentos. Empezaremos por el Snowball. Aquí los resultados han mejorado mucho en general, se han reducido unas 20.000 palabras diferentes por cada tokenizador. Nuestra mejor opción es Letter una vez más, y la palabra más frecuente sigue siendo "i", ya que al no ser considerada stopword no se retirará ninguna en ningún momento.

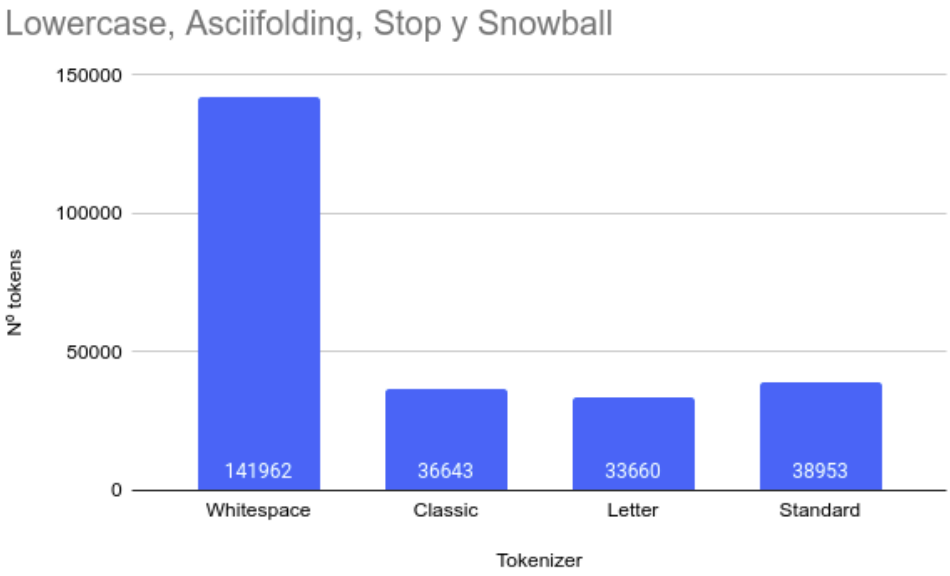

Imagen 3: Gráfica lowercase, asciifolding, stop y snowball

A partir de aquí hemos observado que para cada tokenizador sea cual sea el filtro aplicado la palabra"i" sigue siendo la más frecuente y con el mismo número de apariciones.

Lowercase+Asciifolding+Stop+Porter_stem

En el siguiente caso vamos a retirar Snowball y añadiremos Porter_stem, para evitar posibles conflictos entre los algoritmos que usan estos dos filtros. Para nuestra sorpresa, no hemos obtenido mejores resultados que con Snowball, han aumentado el número de palabras diferentes en los 4 tokenizadores. Nuestra mejor opción sigue siendo Letter.

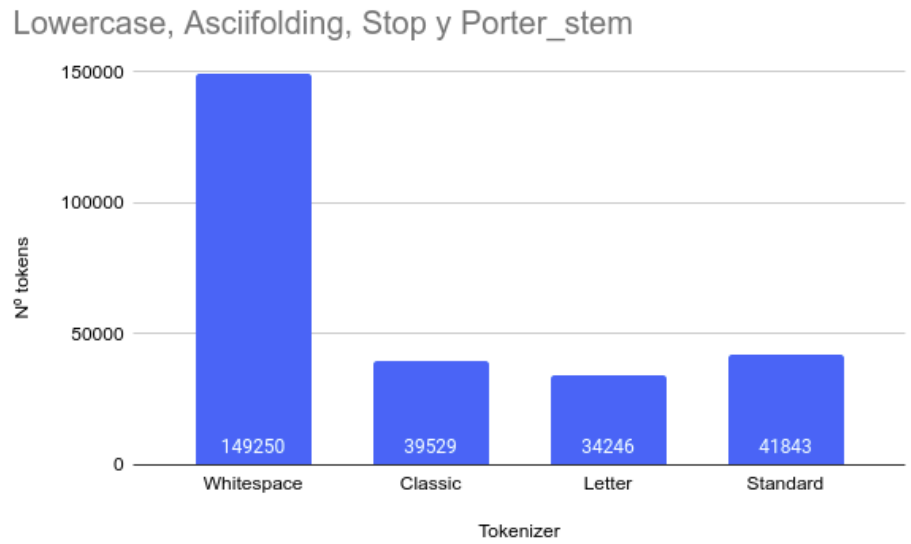

Imagen 4: Gráfica lowercase, asciifolding, stop y porter_stem

Lowercase+Asciifolding+Stop+Kstem

Pasamos ya a la última combinación de filtros, esta vez con el Kstem. Lamentablemente ni el filtro anterior ni este han conseguido superar al Snowball, y el Kstem es aún peor que el Porter_stem.

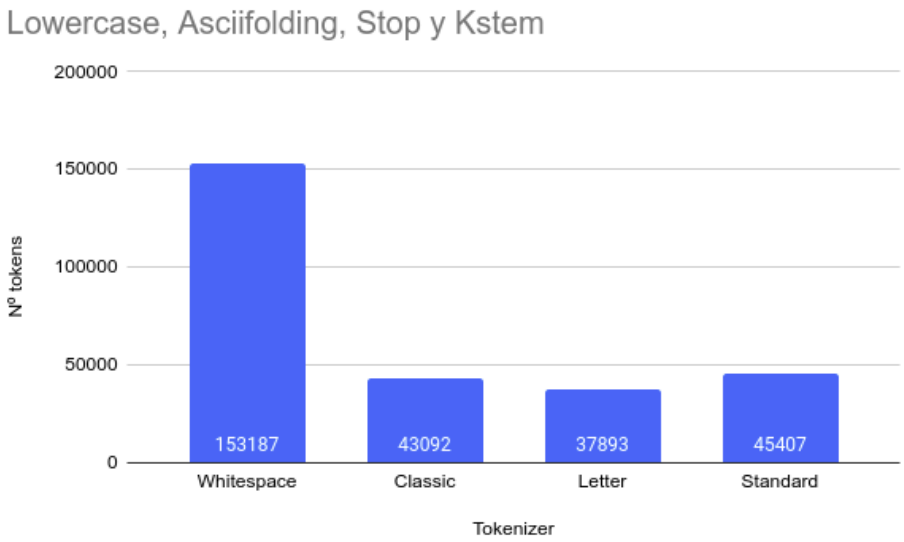

Imagen 5: Gráfica lowercase, asciifolding, stop y kstem

Comprobación mejora en la Ley de Zipf

Ahora que ya tenemos la mejor combinación de tokenizers y filtros para el conjunto de las novelas, comprobaremos si hay alguna mejoría respecto al cálculo de la ley de Zipf de la sesión anterior. Reutilizando el mismo programa, hemos obtenido esta gráfica:

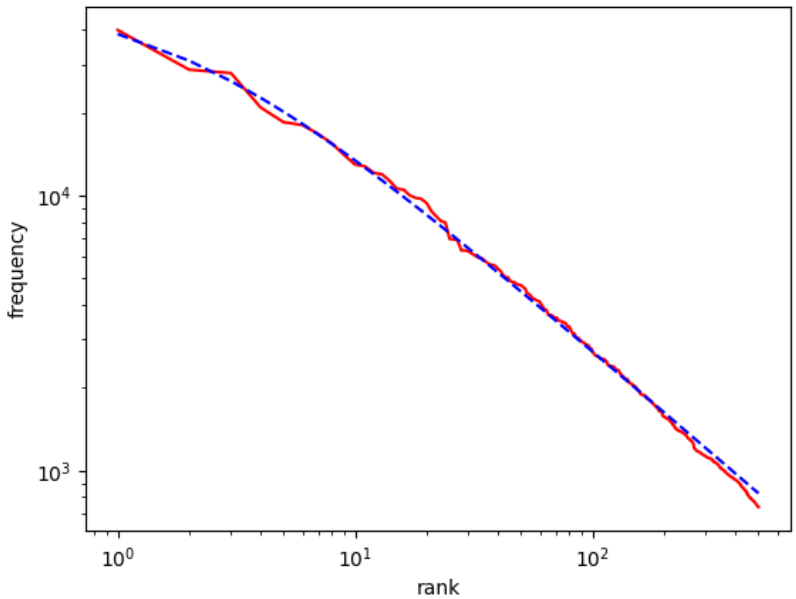

Imagen 6: Gráfica logarítmica de Novels tras aplicar los filtros y tokenizers

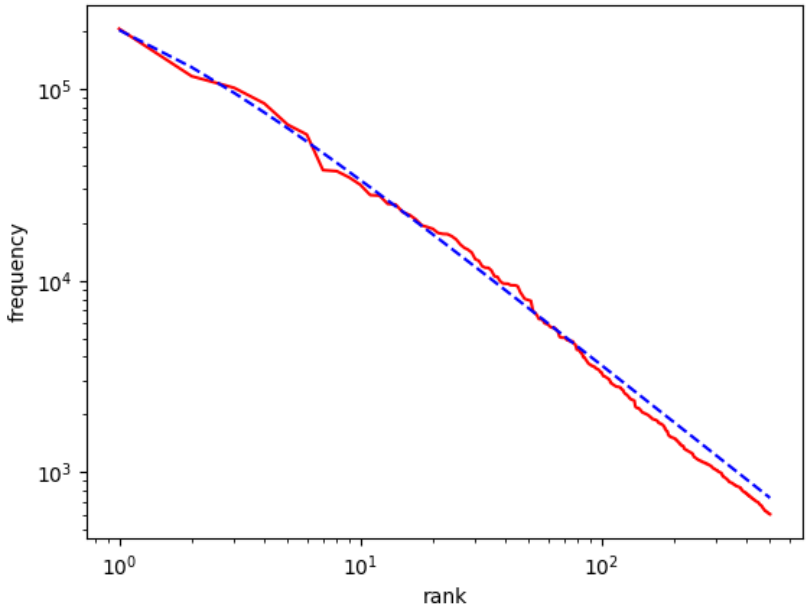

Imagen 7: Gráfica logarítmica de Novels sin filtros ni tokenizers

Podemos observar que después de aplicar la mejor combinación de filtros y tokenizers, observamos que la recta obtenida se ajusta mucho más a la propuesta por la ley de Zipf.

Cálculo tf-idf y similitud coseno

Para el cálculo del esquema de pesos tf-idf hemos modificado el programa *TFIDFViewer.py* que se nos ha proporcionado, y una a una hemos ido modificando el código de varias funciones incompletas.

Para la función *toTFIDF* se nos pedía obtener el vector que contuviera los pares (término, peso) de los documentos a comparar. Para ello simplemente hemos aplicado las fórmulas de las transparencias de teoría. Una vez obtenido este vector, hemos completado la función *normalize,* para normalizar los pesos del vector. Para ello hemos hecho la raíz cuadrada del sumatorio de todos los términos, cada uno elevado al cuadrado. Una vez hecho, a cada término le correspondía su peso normalizado, un valor entre 0 y 1. Finalmente, para la función *cosine_similarity,* hemos usado 2 vectores normalizados como parámetros, y para cumplir la condición de sólo recorrer cada vector máximo 1 vez hemos hecho un bucle while del tamaño del vector, y en cada iteración hemos accedido al término de los dos vectores y si sólo si son la misma palabra, calculamos el producto escalar con los pesos de respectivos términos; en caso contrario avanzamos el índice con palabra alfabéticamente menor. Finalmente hemos retornado la similitud coseno aplicando la fórmula vista en teoría haciendo el producto escalar de los vectores normalizados. [1]

[1] Código completo en TFIDFViewer.py

Experimentación

Una vez terminada la implementación del programa, hemos probado su correcto funcionamiento con varios casos: el primero, con los casos de ejemplo de las slides de teoría (carpeta docs), y el resultado de la ejecución del programa coincide con el calculado a mano mediante las fórmulas; el segundo, hemos calculado la similitud de un documento consigo mismo, y nos da 1, como debería ser.

Ahora, para la fase de experimentación, hemos ido haciendo pruebas con el conjunto de textos coloquiales (20_newsgroups), más concretamente en la subcarpeta ***rec.autos***, comparando documentos 2 a 2. En la siguiente tabla mostramos los resultados de la similitud de dichos documentos:
Los dos siguientes índices se han creado con todos los documentos de las dos subcarpetas (988 y 993 documentos respectivamente).

Doc1: 0006050
Doc2: 0007061
Doc3: 0007155
Doc4: 0012341
Doc5: 0007990

Documento	**Doc1**	**Doc2**	**Doc3**	**Doc4**	**Doc5**
Doc1	**1**	0,01338	0,01304	0,00613	0,00223
Doc2	0,01338	**1**	0,09017	0,01165	0,01899
Doc3	0,01304	0,09017	**1**	0,03859	0,03640
Doc4	0,00613	0,01165	0,03859	**1**	0,00314
Doc5	0,00223	0,01899	0,03640	0,00314	**1**
Media similitud	0,01085	0,04085	0,05505	0,01879	0,01951

Una vez obtenida la media de similitud de un documento al resto (quitando el máximo y el mínimo de cada comparación por columnas), calcularemos la media de estos resultados para hacer una estimación de la media entre los documentos de la subcarpeta de rec.autos. En concreto, esta media es 0,02901.

Haremos lo mismo para la subcarpeta ***sci.space***:
Doc6: 0014092
Doc7: 0014677
Doc8: 0014912
Doc9: 0018015
Doc10: 0014000

Documento	**Doc6**	**Doc7**	**Doc8**	**Doc9**	**Doc10**
Doc6	**1**	0,00675	0,03088	0,02621	0,03384
Doc7	0,00675	**1**	0,02809	0,01031	0,02788
Doc8	0,03088	0,02809	**1**	0,01498	0,01353
Doc9	0,02621	0,01031	0,01498	**1**	0,01296
Doc10	0,03384	0,02788	0,01353	0,01296	**1**
Media similitud	0,03031	0,02209	0,02465	0,01805	0,02508

Ahora calcularemos la media entre estos documentos, y el total es 0,02404.

Los resultados de ambas medias son bastante similares. Ahora compararemos 3 documentos de la primera subcarpeta con 3 documentos de la segunda subcarpeta. El índice es la suma de todos los documentos de ambas carpetas (1981 archivos).

Doc1: 0006050 Doc6: 0014092
Doc2: 0007061 Doc7: 0014677
Doc3: 0007155 Doc8: 0014912

Documento	**Doc6**	**Doc7**	**Doc8**	**Media similitud**
Doc1	0,00533	0,01585	0	0,00706
Doc2	0,00205	0,02166	0,00722	0,01031
Doc3	0,03328	0,05058	0,07426	0,05271
Media similitud	0,01355	0,02936	0,02716	**0,02336**

Conclusiones

Una vez comparadas las dos subcarpetas, la media de similitud entre las dos subcarpetas es de 0,02336. Cabe destacar las siguientes conclusiones:

1) La similitud entre el documento Doc1 y Doc8 da 0. Esto no se debe a un mal funcionamiento del programa, sino a que no hay ninguna palabra que esté presente en ambos documentos. Habiendo comprobado manualmente, vemos que las únicas palabras que coinciden son consideradas stopwords, y nosotros previamente habíamos decidido retirarlas al crear el índice. De modo que decidimos dejar el 0 antes que un número pequeño.
2) La media de las 2 subcarpetas no ha resultado ser tan inferior a las dos otras medias, como hubiéramos esperado. Puede deberse a que los documentos hablan de temas parecidos (ventas, motores tanto de autos como espaciales...). La arbitrariedad de los temas no nos deja predecir una posible similitud a priori.
3) Si en lugar de comparar dos subcarpetas de un mismo conjunto de documentos hubiéramos comparado 2 subcarpetas de conjuntos diferentes (por ejemplo textos literarios y científicos), la diferencia de similitud hubiera sido mucho más notable.

Finalmente, se nos pide comprobar por qué el path de los documentos a comparar parece no estar tokenizado a pesar de que por defecto todos los campos se tokenizan. Hemos comprobado los dos archivos *IndexFiles.py* e *IndexFilesPreprocess.py* y al final del segundo archivo hemos encontrado un fragmento de código que evita que el campo –path sea tokenizado. Adjuntamos dicho fragmento:

```
# configure the path field so it is not tokenized and we can do exact match search
client.indices.put_mapping(doc_type='document', index=index, include_type_name=True, body={
    "properties": {
        "path": {
            "type": "keyword",
        }
    }
})
```