CAIM Lab Session 2: Programming with ElasticSearch

Moreno Oya, Daniel
Cidraque Sala, Carles

Índice

The index reloaded

Lo primero que hacemos es proceder con la creación de 4 índices diferentes sobre el corpus de noticias y cada uno de estos índices utilizará una opción de tokenizado diferente (las 4 opciones siendo Word Oriented Tokenizers) mediante el script IndexFilesPreprocess.py que nos permite darle valor a dos parámetros adicionales (--token y --filter) respecto la versión anterior del script de creación de índices de la sesión 1.

En concreto:

newsws utiliza **whitespace**

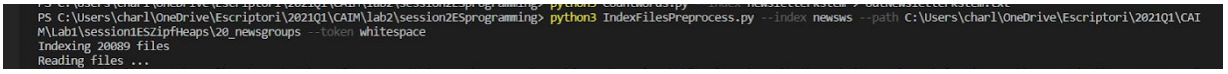

fig1. Creación del índice con whitespace

newsclassic utiliza **classic**

fig2. Creación del índice con classic

newsstandard utiliza **standard**

fig3. Creación del índice con standard

newsletter utiliza **letter**

fig4. Creación del índice con letter

De momento no hemos utilizado ninguna opción de filtrado, es decir, no le damos valor al parámetro --filter, dejamos el que está por defecto que es el *lowercase*.

Una vez tengamos los índices anteriormente mencionados creados nos podemos apoyar en el script CountWords.py para contar cuantos términos hay en cada uno de estos índices. Vemos cuantos tokens se han generado para cada una de las técnicas de tokenizado:

- Para el índice *newsws* utilizando como opción de tokenizador el *whitespace* no hemos conseguido que nos funcione el script CountWords.py.

- Para el índice *newsclassic* encontramos que hemos obtenido un total de **132956** términos distintos utilizando como opción de tokenizador el *classic*.

- Para el índice *newsstandard* encontramos que hemos obtenido un total de **130290** términos distintos utilizando como opción de tokenizador el *standard*.

- Para el índice *newsletter* encontramos que hemos obtenido un total de **94003** términos distintos utilizando como opción de tokenizador el *letter*.

Vemos que el que obtiene un número menor de términos es la opción *letter* de tokenizado. Este *word oriented tokenizer* divide el texto en términos cuando encuentra algo que no sea

una letra (en el idioma inglés). Por lo tanto, como podemos intuir, es obvio que esta sea la opción de tokenizar que obtiene un número menor de términos, ya que elimina todo lo que no sean letras, se hacen más divisiones pero el número de de términos diferentes disminuye ya que son palabras más cortas y el número de coincidencias es mayor.

Dónde deberíamos encontrar un mayor número de términos diferentes con mucha diferencia es con la opción *whitespace*, y es lógico, ya que no discriminamos ninguna palabra o signo, cogemos absolutamente todos los términos siempre que estén separados por un espacio en blanco. Por pequeña que sea la diferencia entre dos palabras o por extraña que sea la composición de esa palabra en términos de caracteres se computará como un término más en en el proceso de tokenización y habrá más variedad.

A continuación cogemos la opción de tokenización más agresiva utilizada, es decir, la que genera más tokens y como hemos visto esta es la opción de *letter*. Con ella aplicaremos los filtros mediante el parámetro *--filter*. En este caso utilizaremos los filtros de *lowercase* que ya se utilizaba en las sesión 1 implícitamente y también la opción de filtrado *asciifolding* que recorre el texto y cada vez que no encuentre un carácter que no es ASCII, que no se contempla en el idioma inglés pues se eliminará. La creación del nuevo índice *novelsletterfilter* es la siguiente:

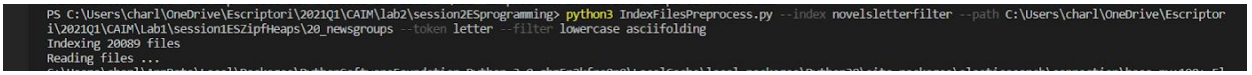

fig5. Creación índice --token letter y --filter lowercase asciifolding

Nos ayudamos del CountWord.py para ver que términos son los más frecuentes:

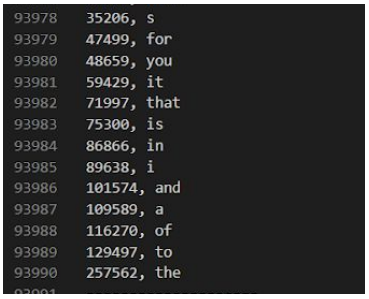

fig6. términos más frecuentes con stopwords

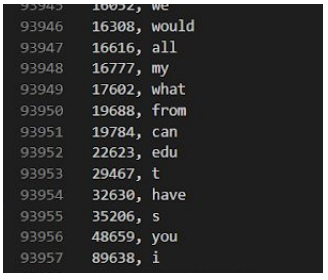

fig7 términos más frecuentes sin stopwords

Vemos que los términos más frecuentes son los llamados stopwords del idioma inglés (adverbios, artículos, preposiciones), términos que no son útiles semánticamente. Esto ha ocurrido porque no hemos puesto el filter *stop*. Si lo aplicamos obtenemos los resultados de la figura *fig7*.

Para ver las diferencias entre las diferentes opciones de filtrado es interesante analizarlas por separado. Para ello hemos elaborado la gráfica de la *fig8*, donde vemos que el # términos varía mucho según el filtro utilizado. Un factor a tener en cuenta es que el orden en que ponemos como parámetro los diferentes filtros (en caso de que queramos aplicar más de uno) es muy determinante, ya que cada técnica de filtración se aplicará sobre el resultado de aplicar la técnica de filtrado de la opción de filtro que le precede (no es una combinación de todas las técnicas pasadas como parámetro aplicadas a la vez). Cuando obtenemos un menor número de palabras es usando el filtro *lowercase*, es por lo tanto el filtro que "homogeneiza" más todos los términos de nuestra prueba con el corpus de noticias y por lo tanto obtenemos menos variedad de términos. Eso no quiere decir que obtengamos menos número de tokens totales, por ejemplo con el filtro stop, que es un tipo de filtro que elimina tokens (no como el de *lowercase* mencionado que es de modificación de token, los pasa a minúscula) eliminamos muchas palabras, concretamente las más utilizadas en el inglés (*stopwords*). Gracias al programa escrito en la sesión 1 hemos podido comprobar que por ejemplo utilizando solamente *lowercase* obtenemos un número total de palabras de 5.291.784, mientras que utilizando solo *stop* obtenemos 3.886.363.

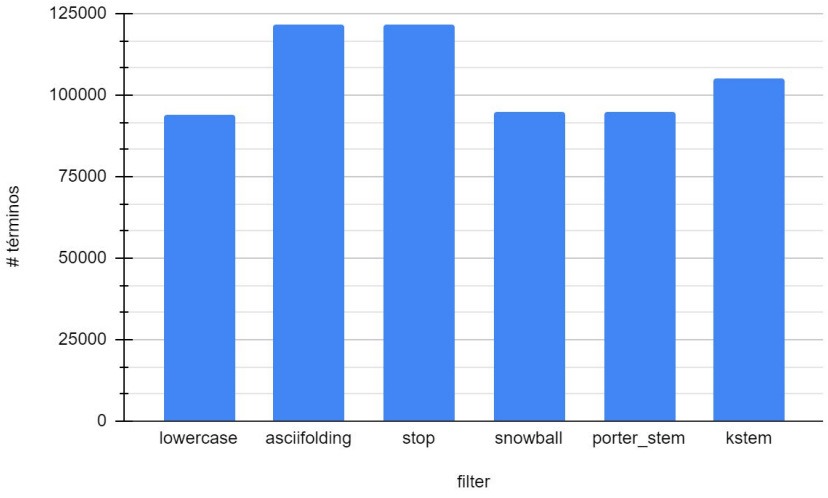

fig8 . Diferencias entre las diferentes opciones de filtrado

Probamos el mismo tokenizado y filtraje (--token letter --filter lowercase asciifolding stop) pero ahora con el arxiv corpus.

```
87643   35408, time
87644   36303, n
87645   37036, s
87646   41339, data
87647   45392, model
87648   51692, our
87649   54145, can
87650   64172, which
87651   73294, from
87652   210217, we
87653   ----------
```

fig9. tokenizado y filtraje (--token letter --filter lowercase asciifolding stop) con el arxiv corpus.

Computing tf-idf ’s and cosine similarity

En este apartado teníamos que completar el código de TFIDFViewer.py para que compare la similitud de dos documentos.
Primero debíamos completar la función toTFIDF, la cual retorna una lista de pares término y peso. Primero debíamos obtener una lista con pares término y frecuencia en el documento y después otra lista de pares término y frecuencia en el índice. Seguidamente debíamos obtener el número de documentos en el índice y después calcular el tfdif para cada término. Esta función no representa demasiada dificultad ya que solo había que calcular el tfidf para cada término.
Seguidamente debíamos completar la función normalize, la cual primero obtiene la normal del vector y después divide cada elemento del vector por ella.
Después debíamos completar la función print_term_weigth_vector, la cual imprime una línia por cada término y su peso.
Por último teníamos que escribir el código de la función cosine_similarity, la cual calcula la similitud entre dos vectores. Hemos calculado la similitud como el sumatorio de la multiplicación de los pesos de los términos que coinciden en ambos documentos. Para ello hemos creado dos iteradores que recorrían el vector y dado que los vectores estaban ordenados alfabéticamente podíamos ir recorriendo los dos vectores simultáneamente. El coste de esta función es O(n), donde n es el tamaño de la lista más corta.

Experimenting

Una vez hecho el programa anteriormente descrito vamos a proceder a hacer un seguido de experimentos, tests sobre los mismos corpus utilizados tanto en esta sesión como en la anterior.
Para empezar comparamos un documento consigo mismo. En concreto comparamos por ejemplo consigo mismo la noticia sobre motorcycles (folder “rec.motorcycles”) “0008000”. En concreto con el índice *newsletter* creado al inicio de la sesión. El resultado que deberíamos obtener es una Similarity = 1, ya que para conseguir el máximo de similitud entre dos documentos se tiene que tener un número alto de términos iguales y como más frecuentes sean estos términos iguales conlleva a una mayor similitud (al ser el mismo documento obtenemos el valor máximo de similitud).

fig10. Similitud entre un mismo fichero

El siguiente experimento queremos responder si las noticias de un mismo tema se parecen realmente más entre ellos que con temas diferentes. Para ello analizamos 5 posibles parejas: **P1:** (tema hockey1 (“0010000”)-tema hockey2 (“0010001”)), **P2:**(tema hockey1(“0010000”)-tema hockey3 (“0010002”)), **P3:**(tema hockey1 (“0010000”)- tema hardware sys mac (“0003009”)), **P4:**(tema hockey1(“0010000”)- tema charlas politicas sobre mideast (“0016011”)), **P5:**(tema hockey1(“0010000”)- tema ciencia espacio (“0014000”)).
En la gráfica *fig 11* vemos como una noticia de un deporte en concreto como el hockey tiene mucha más similitud con otras noticias de hockey que no con noticias de otros temas como

por ejemplo hardware mac, charlas politicas sobre el mideast o noticias científicas sobre el espacio.

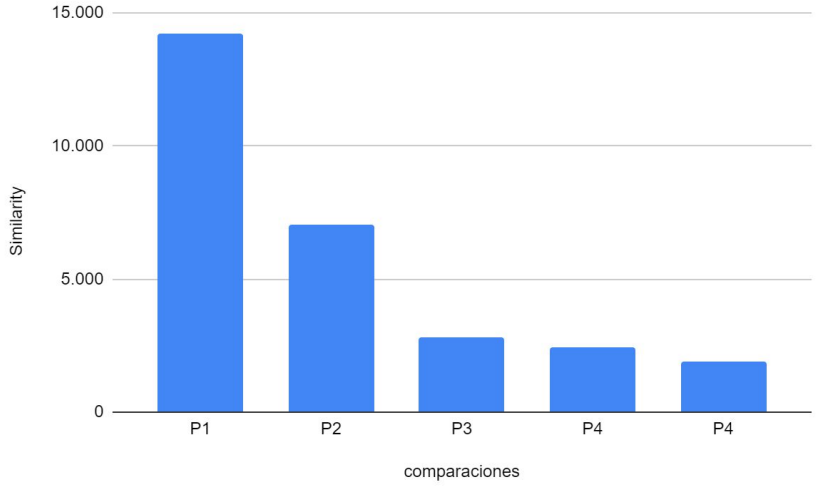

fig11. Comparación de similitud entre tipos de noticias

Otro experimento que podemos hacer es ver si hay temas que se parecen más entre ellos que otros. Para ello analizamos 6 parejas (dos parejas de “soc.religion.christian”, dos parejas de “rec.autos” y otras dos de “sci.med”), en este caso utilizamos un índice con un filtrado que permita obtener tokens con números, ya que hay seguramente noticias como las científicas que hacen un mayor uso de estos términos y no queremos eliminarlos.
Depende mucho de los ficheros que comparemos ya que hemos podido comprobar que por ejemplo en “rec.autos” hay diferentes estructuras de documentos, en la primera comparación nos sale que tiene una mayor similitud que en la segunda porque las dos son descripciones de vehículo siguiendo una estructura casi igual y palabras más parecidas, en cambio, en la segunda comparación vemos que obtenemos menos similitud y es porque hemos podido comprobar que ahora estábamos comparando una descripción como la anterior con una review más general y extensa de otros vehículos. Lo mismo pasa con los otros temas. Es difícil saber si hay temas con mayor relación entre ellos que otros (aunque con los ejemplos que hemos puesto podríamos decir que el tema “rec.autos” tiene mayor similitud entre sus documentos que los otros dos temas seleccionados).

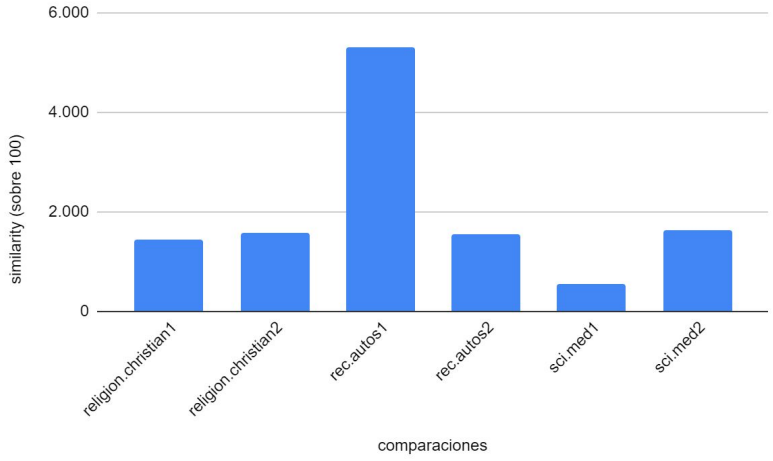

fig12. Comparación de similitud dentro de un mismo tema del corpus de noticias

En el la *fig13* también podemos observar que la novelas de Dickens y Poe tienen más similitud entre ellas que con el documento de Darwin, ya que muy seguramente es un texto de ámbito más científico y los otros dos son novelas literarias.
También hicimos una comparación entre dos novelas de Poe (entre el Vol1 y Vol2) y nos dió una similarity = 0,24176. Vemos que es una similarity muy superior a los vistos en la *fig13* ya que como podemos intuir el Vol2 es una continuación del Vol1 y nos puede indicar que utiliza un lenguaje similar en ambos volúmenes a parte de ser el mismo autor.

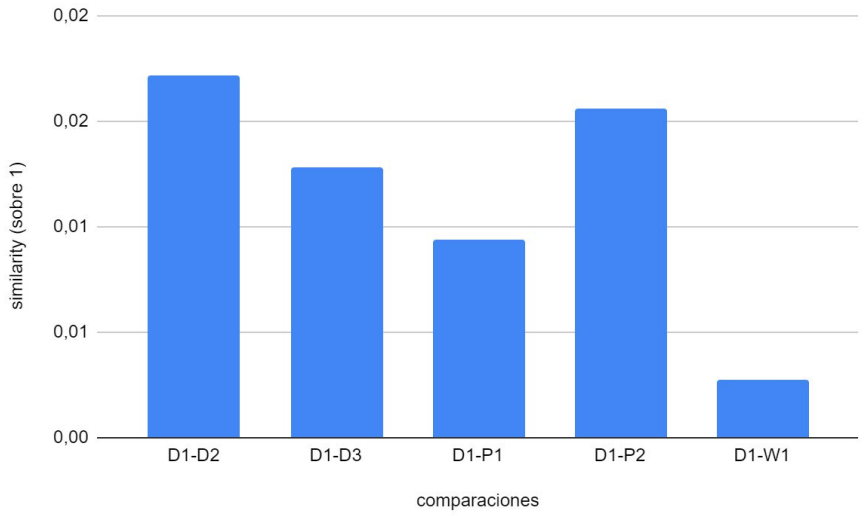

fig13. D1,D2 y D3 (Dickens), P1 y P2 (Poe) y W1 (Darwin)

Como observación final tenemos que en la estructura de datos que hay en el índice no está solo el texto, también está la path del fichero, esa información (el path) no hay que tokenizarla ni filtrarla porque eliminaríamos la path y luego no podríamos recuperar el fichero. Le decimos a elasticsearch que la path es una *keyword* y que ese término no se toca, se inserta y se deja intacto.

```
# 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",
        }
    }
})
```

fig14. Fragmento de IndexFilesPreprocess.py en el que se indica que el path es un keyword