FUNCIONAMIENTO DEL CÓDIGO BASELINE KDD 2020: Ejemplo e ideas.

Este documento se centra en explicar cómo funciona el código Baseline de la KDD 2020.

Una resumida y sintetizada idea de lo que realiza este código es:

- Carga y combina las tablas de las fases
- Utiliza las siguientes 3 funciones para la obtención de la recomendación final:

1. Genera una matriz de correlaciones (similitudes) entre todos los items.
2. Recomienda, basándose en la anterior matriz, a cada usuario los 50 items más apropiados.
3. Realiza una predicción final donde se queda los 50 mejores, mezclando los más apropiados que se obtuvo antes con los más populares.

- Emplea las 3 funciones anteriores en un breve código del que se exporta el resultado en el formato de subida de la KDD.

Para facilitar la comprensión de la primera función (probablemente, la más importante), ilustraremos su funcionamiento con un ejemplo recogido en el archivo Excel adjunto.

NOTA: En el ejemplo, simplemente se utilizan la tabla train y test_clic de la fase 0. Además, por comodidad, se utiliza el usuario cuyo user_id es 12, ya que únicamente hace 3 clics.

0. PREPARACIÓN:

El código comienza cargando todas las tablas de train y test_click que estén disponibles hasta la fase actual y reuniendo todas las observaciones en esa gran tabla llamada whole_click, **que está ordenada por tiempo en orden ascendente.**

1. USO DE LA 1ª FUNCIÓN:

Es importante ver lo que devuelve la matriz para tener claro en qué consiste su funcionamiento:

- Una gran matriz (realmene, para Python por dentro es un diccionario) que recoja el parecido existente entre los objetos de la tabla whole_click creada antes, ya veremos con qué medida de similitud.
- Un diccionario de Python que asigna, a cada usuario, todos los objetos que clica.

¿Cómo funciona esta función y cómo crea esas similitudes entre objetos?

1. La función crea 2 diccionarios fijos, uno que empareja cada usuario con todos los objetos que clica (user_item_dict) y otro que relaciona, cada usuario con todos los tiempos en los que hace click (user_time_dict), que recordemos que están ordenados cronológicamente.
2. Crea 2 diccionarios vacíos que irá rellenando: uno es donde guardará, para cada objeto, su similitud con los demás (item_sim_list) y el otro cuenta todas las veces que aparece un item (item_cnt). Esto último será útil para una ponderación final.
3. Entra en una anidación de hasta 3 bucles donde, para cada usuario, va viendo sus items y sumándoles una similitud según dos factores:

- El orden en el que el usuario clica en ellos (lo que llama loc1 y loc2)
- La distancia temporal que hay entre ellos (lo que llama t1 y t2)

4. Termina reajustando las similitudes obtenidas, según las veces que aparece cada item
5. Devuelve el diccionario de similitudes entre objetos y el diccionario user_item_dict.

--> Ir al Excel y ver el ejemplo de funcionamiento más claro:

OBSERVACIONES:

Lo más importante que hay que tener en cuenta es que **la similitud entre dos objetos va a depender del orden en el que son clicados por el usuario y del tiempo que transcurre entre un clic y otro.** Así, no hay simetría, puesto que el código pondera con un 0'7 la similitud que tiene un objeto A con un objeto B que ha sido clicado anteriormente a él, mientras que pondera con un 1 a los objetos clicados después.

Esto tiene sentido, ya que **puntúa mejor los clics posteriores de un objeto**, que son los que queremos predecir.

Nótese que **esta similitud se va acumulando**, y que, en general, para que un item A tenga mucha similitud con un item B, debe ocurrir muchas veces que este item B aparezca clicado después de A (o muy cerca) y en un lapso temporal nulo (o muy pequeño).

En cualquier caso, siempre podríamos plantearnos qué efecto tendría calibrar estas ponderaciones de 0,7 y 1, dando más peso a clics futuros y menos a pasados.

También hay que tener en cuenta que, **la única forma de que dos objetos tengan similitud es la de que aparezcan en el historial de clic de algún usuario**, aunque estén muy separados. Sería extraño que ocurriera, pero podría haber dos objetos que se parecieran (en términos de embedding, por ejemplo) pero que, como no coinciden en ningún usuario, no tendrán similitud (OJO: no es que tengan similitud = 0, sino que, directamente, no tienen similitud, es decir, no aparecen recogidos en el diccionario item_sim_list, de forma que sería imposible predecir con ellos).

2. USO DE LA SEGUNDA FUNCIÓN.

Una vez que ya sabemos lo bien o mal que se llevan los items entre sí en ese gran diccionario *item_sim_list*, ahora la segunda función se encarga de hacer una recomendación a cada usuario en función de esa matriz de similitudes.

Esta función recibe los siguientes elementos de entrada:

- Diccionario de similitudes (*item_sim_list)*
- Diccionario de usuario con sus items *(user_item_dict)*
- El user_id del usuario al que hacer la recomendación
- Cantidad de objetos a estudiar *(top_k)*
- Cantidad final de objetos a recomendar *(item_num)*

La idea de la función es sencilla: recibe un usuario y le da una recomendación basándose en su historial de clics y las similitudes halladas anteriormente. ¿Cómo funciona paso a paso?

1. Inicializa rank={}, el diccionario que, dado un usuario, guarda su recomendación.
2. Para el usuario dado, crea el array interacted_items con sus items clicados.
3. Invierte el orden de este último vector, **poniendo en primer lugar el clic más reciente.**
4. Entra en un doble bucle que va cogiendo cada item clicado (i) junto a su posición de clicado (loc) y busca entre los top_k primeros objetos (se coge top_k=500) de la columna del objeto i, ordenados según su item_id (esta ordenación es irrelevante).
5. Ahora bien, **para aquellos objetos j que NO estén en el historial de clicks del usuario** (los que no haya visto), inicializa su puntuación a 0 y se les va sumando una puntuación o rank como sigue:

   Rank += wij * (0’7^loc)

   Como wij representa la similitud entre el par de objetos (i,j), **esta medida rank va puntuando peor a los objetos (i,j) a medida que crece la variable loc, es decir, a medida que el objeto i se aleje más de ser el último click del usuario.**

   Así, únicamente se obtienen puntuaciones para objetos NO clicados previamente por el usuario, y estas son mejores cuanto más reciente es el click. Nótese que como los objetos van apareciendo más de una vez, estas puntuaciones se van acumulando.

6. Finalmente, ordena todos estos rank y devuelve exclusivamente los 50 primeros.

MEJORAS:

- Como para algunos usuarios el click que hay que predecir se encuentra en medio de su historial de clicks, podemos asumir que no hará click en algo en lo que después clickó, con lo que habría que implementar que no se puntúe a los items futuros, ganando precisión.
- Para puntuar los objetos, se usa la matriz de similitudes y se pondera según la cronología de los clicks del usuario, pero no se tiene en cuenta el embedding para una posible ponderación más.

3. USO DE LA 3ª FUNCIÓN

IDEA:

Realizar una predicción final para cada usuario, mezclando sus objetos más apropiados (obtenidos con la función recommend anterior) con los objetos más populares.

La función recibe los siguientes parámetros de entrada:

- Un dataframe, que será el de cada usuario con sus 50 recomendaciones.
- Una columna que calcular, que será la de la puntuación de recomendación final.
- Un listado de los 50 objetos más populares (los que más clicks tienen).

Una vez recibidos estos parámetros la función actúa de la siguiente manera:

1. Divide los 50 objetos más populares en un único array, separados por comas.
2. Crea un vector scores de 50 elementos, de la forma [-1, -2, -3, …, -50]
3. Crea un listado “ids” con los user_id únicos que aparecen en el dataframe dado
4. Crea un nuevo dataframe llamado fill_df, en la que aparecen 50 veces cada uno de los ids anteriores. Seguidamente, los ordena, obteniendo un mismo id en las primeras 50 filas, un nuevo id en la fila 51 hasta la 100, el tercero a partir de la 101, y así sucesivamente.
5. A ese dataframe fill_df le añade 2 columnas: la primera (‘tem_id’) coloca, al lado de cada una de las 50 filas de cada usuario, los 50 items más populares, la segunda (‘sim’) añadiría el vector de puntuaciones negativas creado en el paso 2. Con todo ello, el dataframe creado tendría esta pinta:

	user_id	item_id	sim
58739	11	113569	-1
2197	11	52766	-2
77032	11	87107	-3
10512	11	31005	-4
5523	11	87254	-5
...	...	...	...
16153	35398	51182	-46
42761	35398	14666	-47
7838	35398	57900	-48
51076	35398	20208	-49
34446	35398	7709	-50

83150 rows × 3 columns

(No prestar atención a la columna de la izquierda, pues son los índices originales de las observaciones y no tienen relevancia alguna).

6. Une este último dataframe al dataframe que le pasamos inicialmente con las recomendaciones.
7. Ordena esta gran tabla según la columna ‘sim’, que representa la similitud, de más a menos, y elimina lo duplicados con keep=first. Esto es importante, puesto que con la unión del paso 6, a cada usuario le hemos añadido los 50 objetos más populares con una medida de similaridad negativa. Entonces, en caso de que alguno de esos objetos ya hubiera estado desde antes recomendado para un usuario, sería un duplicado (pero tendrían distinto valor ‘sim’) y con keep=first, al haber ordenado la tabla por ‘sim’, nos quedaríamos con la observación de más similitud.
8. Crea una nueva columna ‘rank’ que, tras haber agrupado el dataframe por usuarios, puntúa de primera a última las observaciones según su ‘sim’. Quedaría así:

	user_id	item_id	sim	rank
578759	10362	113569	1.716005	1.0
283515	4829	88058	1.683048	1.0
591074	29898	88058	1.636380	1.0
578760	10362	116437	1.600590	2.0
225254	8844	113569	1.569474	1.0
...	...	...	...	...
48222	25344	7709	-50.000000	182.0
49326	3443	7709	-50.000000	550.0
21468	25355	7709	-50.000000	538.0
67120	25377	7709	-50.000000	309.0
34446	35398	7709	-50.000000	539.0

838063 rows × 4 columns

9. Se queda con aquellos items de 'rank' menores o iguales que 50.
10. Reestructura el dataframe para darle el formato de la KDD.

En resumen, la función recoge las recomendaciones anteriores e inserta, para cada usuario, items populares con similitud negativa, que, al reordenar, se colarán dentro de los 50 finales antes que otros items que tengan similitudes muy negativas.

COMENTARIOS DEL CÓDIGO DONDE USA LAS FUNCIONES:

```
# Código:
item_sim_list, user_item = get_sim_item(whole_click, 'user_id', 'item_id', use_iif=False)

for i in tqdm(click_test['user_id'].unique()):
    rank_item = recommend(item_sim_list, user_item, i, 500, 500)
    for j in rank_item:
        recom_item.append([i, j[0], j[1]])

# find most popular items
top50_click = whole_click['item_id'].value_counts().index[:50].values
top50_click = ','.join([str(i) for i in top50_click])

recom_df = pd.DataFrame(recom_item, columns=['user_id', 'item_id', 'sim'])
result = get_predict(recom_df, 'sim', top50_click)
result.to_csv('baseline_phase_4_LB_0_2435.csv', index=False, header=None)
```

Las 3 funciones anteriores se combinan en este código para llegar a la recomendación final.

En la primera línea se obtiene el diccionario de similitudes entre los objetos, así como el diccionario que empareja cada usuario con sus objetos (será necesario para le función recommend).

El siguiente bucle consiste en ir recorriendo los usuarios de la tabla test, para los cuales se realiza una recomendación, que se guarda en ese dataframe recom_item.

En top50_click simplemente se almacenan los 50 items que más aparecen (más populares).

Finalmente, se genera un nuevo dataframe a partir del recom_item anterior, que se pasa a la función get_predict para llegar al resultado final, que se exporta en formato .csv.

IDEAS Y POSIBLES MEJORAS PARA EL CÓDIGO:

Como hemos podido ver, el código se basa claramente en una idea de filtro colaborativo, es decir, busca concurrencias entre los items según los historiales de clicks de todos los usuarios y se basa en las similitudes calculadas para realizar las recomendaciones a los usuarios.

Esto es bueno en el sentido de que favorece la diversidad de items en las predicciones, pero tiene la pega de que no atiende al historial de cada usuario para realizar una recomendación más personalizada, y quizás ahí se esté perdiendo información. Esto podría ser un aspecto clave a la hora de mejora el código.

Por ejemplo, para darle a la predicción un toque más personal según el cliente, podríamos pensar en ponernos en su punto de vista. Si pensamos en los usuarios que compran en páginas web de comercio online, con las facilidades que se tienen, podríamos distinguir entre dos tipos: quienes compran por necesidad y quienes compran por consumismo.

- Aquellos usuarios que compren por necesidad, en general, **van a tener sesiones de clicks más cortas**, en las que **se centrarán mucho en lo que está buscando** para comprar (por ejemplo, una funda para su móvil) y, como mucho, visitarán algunos objetos parecidos, como podrían ser unos auriculares o un cargador más rápido.
- Los usuarios más asociados al consumismo serían aquellos con **sesiones de mayor cantidad de clicks** seguidos en las que puede haber una **gran diversificación de items** en la búsqueda, pudiendo haber, a priori, poco parecido entre los primeros y los último.

En esa línea, habría que pensar cómo estudiar la diferencia temporal entre los clicks que se tienen de un usuario y su click a predecir, de forma que **una distancia "corta" podría darle más relevancia a su historial de items que a las similitudes de los demás usuarios.** En este caso, podríamos utilizar para la recomendación aquellos items que tengan el mayor parecido en términos de embedding con los items de su historial de los que se disponga de embedding, midiéndolo con alguna distancia (coseno, producto escalar, etc.).

En el caso de usuarios que tengan un perfil más consumista, **quizás haya que ponderar más aquellos items populares o más visitados** (que suelen tener más publicidad y suelen ser más fáciles de acceder), antes que concurrencias de items del historial.

En cualquier caso, la mejora de este código se basa en alternar o mejorar esa forma de recomendar basada en lo que hacen los demás.