LimeSpot Machine Learning Engineer Take-home assignment

Amir Hosein Safari

Octobor 15, 2021

1 Section 2

1.1

B and C are the most similar products. For this problem I assumed that the order of IDs is not important in both "List of Category IDs" and " List of Keyword IDs". For instance, for product A, there is no difference between categoryID 1 and categoryID 2. Considering this assumption, in order to calculate their similarity, we need to use the Cosine similarity. The reason that we choose Cosine similarity, is that this type of similarity calculation is generally used as a metric for measuring distance when the magnitude of the vectors does not matter (in our scenario we don't have any preferences for each of the features which mean that their magnitude is not important).

I calculated the cosine similarity for each pair and the results are in Figure 1.

As you can see for pairs B, C we got the highest score.

Moreover, I did another experiment, I assumed that "Vendor", "Size", and "color" are not important for calculating the similarity, since from the customer's perspective, probably he/she is not looking for only blue color. Therefore, we shouldn't consider color as a feature in this case.

You can see the results of this experiment in Figure 2. As you can see still B-C got the highest score.

1.2

1.2.1 First solution

For this section, we assumed that we don't want to use any approximation techniques and we want to find the exact similarity for each pair, not its approximation.

Therefore, we need to calculate the cosine similarity for each pair. This means we need 100k*100K/2 calculations and we need the same amount of space.

This is a huge amount of space and if we store them as a 2D matrix or list in memory, then each time that we need to read a similarity we need to load a huge amount of data and I/O is very time-consuming. Also, if we add a new sample to the dataset, we need to load, modify, and save a huge dataset as well. Therefore, we need to find a way to make it more efficient.

We need to save the cosine similarity score for each item in a separate file. It means that we have item A, and we have a file for item A that stores the cosine similarity between A and other items.

This approach has two benefits:

1) When we want to see the most similar items to A, we don't need to load the score of other items and we just load the scores of A. It saves us a bunch of time and memory.

2) If we are going to delete an item from our data, we don't need to modify a huge dataset.

	A	B	C
A	1	0.40414519	0.56428809
B	0.40414519	1	0.77005354
C	0.56428809	0.77005354	1

Figure 1: cosine similarity for section 1.1

	A	B	C
A	1	0.37796447	0.55555556
B	0.37796447	1	0.81892302
C	0.55555556	0.81892302	1

Figure 2: cosine similarity for section 1.1

However, we need to consider that for each time that we are going to add an item, we need to load all these files and add the similarity score to the new item in their list. Considering the fact that we just add an item once, but we probably use it thousands of times, we can ignore this disadvantage of this approach.

The time complexity of this approach is $O(n^2)$ in the preprocessing and O(n) in real-time to catch the most similar items.

1.2.2 Second solution

However, if we care about the preprocessing time and we can't afford $O(n^2)$ calculations, we can change the problem in order to reduce the time complexity. But this approach is slower in real-time. If we assume that each time for item X we want to find the k most similar item to it and we assume that k is much smaller than N (number of items). In this case, instead of storing each pair similarity, we can run the following algorithm:

The main idea is that each time that we want to find the most similar items to item X, except for calculating its similarity to each item, we find the most similar item to it (k times).

We know that cosine similarity is greatest when Euclidean distance is smallest after normalizing by the L2 norm. Therefore, the problem reduces to the closest pair of points problem, which can be solved in $O(n * log(n))$ time. The time complexity of this approach is $O(k * n * log(n))$ which is way smaller than $O(n^2)$ and also it requires less space since it's not storing any value on the memory.

1.2.3 Third solution

In this solution we are going to change the way that we are calculating the cosine similarity.

In Scipy, and probably in general, we are calculating the cosine similarity by this term:

$dist = 1.0 - np.dot(u, v)/(norm(u) * norm(v))$

However, we can change it to this in order to make it more efficient:

$data1 = data/np.linalg.norm(data, axis = 1)[:, None]$

$mat1 = np.einsum('ik, jk- > ij', data1, data1)$

This way, we normalize data once at the start, rather than each time. And then use einsum to calculate the whole set of dot products.

However, einsum while fast for modest size arrays, can get bogged down with larger ones. But even this works in our scenario where the data is too large to handle with one call to einsum, we could break the calculation into blocks, e.g.

$mat[n1 : n2, m1 : m2] = np.einsum('ik, jk- > ij', data1[n1 : n2, :], data1[m1 : m2, :])$

1.3

For the first approach the pseudo-code is as follow:

```
p: list of all Products
temp_matrix = matrix of shape len(p) * len(p)
for i in range(0, len(p)):
    for j in range(0, i):
        cosine_sim = calculate the cosine similarity of p[i] and p[j]
        temp_matrix[i][j] = cosine_sim
storing the similarities
for i in range(0, len(p)):
current_product = []
```

for j in range(0, len(p)):
    if $i < j$:
        $current\_product$.append($temp\_matrix[i][j]$)
    else:
        $current\_product$.append($temp\_matrix[j][i]$)
    save the $current\_product$ in memory