Recommendation Systems.

Content based recommendation system. → based on user's similarity with the item vector. For example, a user might be represented by the average of the products they've bought so far, i.e., $\vec{u} = \frac{\sum_i P_i}{n}$, and then the similarity between a user and another product P_j is given by their cosine similarity. $\cos(u, v) = \frac{u^T v}{\|u\| \|v\|}$. Note that a user vector can also be constructed in other ways (because this way, there is a cold start problem). For example, we can just ask the user!

Collaborative filtering. (based on ratings of other products).

Item-item collaborative filtering:

$$R(v_j, u_i) = \frac{\sum_{i \in N(u_i, v_j)} S_{xi} \, r_i}{\sum_i S_{xi}}$$

(in practice, with mean subtraction)

where $N(u_i, v_j)$ → Neighborhood of item v_j for user u_i. That is, the items that user u_i has rated, and are similar to product v_j upto some threshold.

S_{xi} → Similarity of product i with v_j

r_i → User u_i's rating for product i.

$S \in$ set of products that both users u and v have rated.

Similarity here is defined in terms of either Jaccard similarity of ratings vectors for two products or the Pearson's? correlation co-efficient.

$$Jac(u, v) = \frac{|u \cap v|}{|u| + |v| - |u \cap v|}$$

Pearson's coeff =

$$P(u, v) = \frac{\sum_s (r_{xs} - \bar{r}_x)(r_{ys} - \bar{r}_y)}{\sqrt{\sum_s (r_{xs} - \bar{r}_x)^2 \cdot \sum_s (r_{ys} - \bar{r}_y)^2}}$$

Latent factor models

Given a rating matrix $R \in \mathbb{R}^{m \times n}$, where $m \to$ number of items, and $n \to$ no. of users, we use SVD to say

$$R = P \Sigma Q^T \quad (\text{or } P Q^T \quad (\text{with } \Sigma \text{ folded into } Q))$$

where $P \in \mathbb{R}^{m \times r}$ $\Sigma \in \mathbb{R}^{r \times r}$ and $Q \in \mathbb{R}^{r \times n}$.

We are essentially factorizing (expressing) each item into 'r' latent features, and similarly for users. Each element of P and Q is treated as a learnable parameter.

The rating matrix is sparse. We use the known values to obtain the elements of P and Q with a least square regression model.

↳ Some values will likely still be undetermined.
↳ What happens when new element are added?

} They don't really talk about this. I suppose you need to re-calculate after each step.

For the LSQ model, the initial point for SGD is chosen from the SVD factorization, assuming all missing values are zeroes.

With deep learning, we are most often building embeddings of items and users, and then using k-nearest neighbors to find products similar to a given product or to a given user. These embeddings are usually trained via metric learning (such as triplet loss, or pairwise contrastive loss?). Both methods assume that we have a dataset of pairs of products / users that are similar, and pairs that are <u>not</u>.

In any case, the outputs of these models are the embeddings themselves, and need to be stored (loaded into memory and we need to be able to run k-nearest neighbors efficiently on these embeddings

<u>Locality</u> <u>Sensetive Hashing</u> (LSH) ! The purpose of LSH is to efficiently look up similar vectors in a dataset. These vectors might be very high dimensional embeddings / feature vectors. There are two steps →

(a) Min-hashing → This is a step where we calculate a lower-dimensional signature of the embedding so that the similarity of the original vector is maintained.

Min-hashing contd.

Given boolean vectors, we calculate a permutation of the rows, and then for each permutation, our signature is the index of the first row at which there is a 1. ~~Tode~~ We do 'n' such permutations to obtain a n-dimensional signature for each sample.

For ex:

$$\begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 1 \end{bmatrix} \rightarrow \text{permute rows} \rightarrow [3, 2, 0, 1, 4] \rightarrow \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

← first 1 is at index 1. ↓ signature $\boxed{1}$

$$\rightarrow \text{permute rows} \rightarrow [4 \; 0 \; 1 \; 2 \; 3] \rightarrow \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

← first 1 is at index 0 ↓ $\boxed{\begin{matrix} 1 \\ 0 \end{matrix}}$

Note that we do this for all data points (the same n permutations n times are applied to all data points obviously.

Proof of similarity.

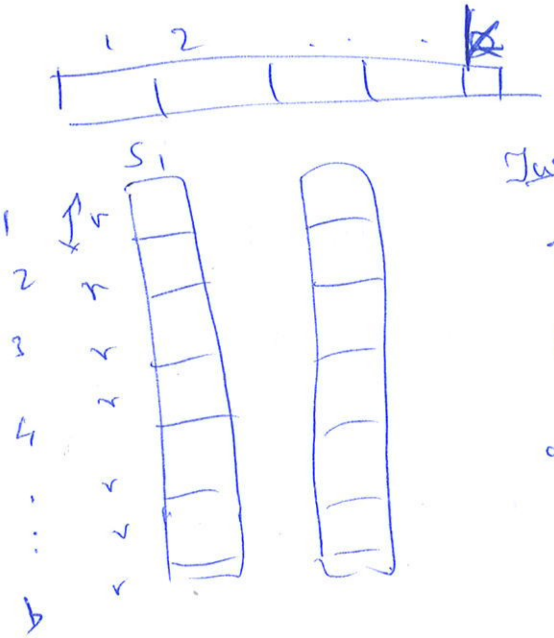
If $Sig_1 == Sig_2$, there are $|A \cap B|$ indices that would need to be on top. $Pr(Sig_1 == Sig_2) = \frac{|A \cap B|}{|A \cup B|}$

Hmm, so if we pick n hash functions, this is like n bernoulli trials, and the expected no. of positions where the signatures match is $np = n\left[\frac{|A \cap B|}{|A \cup B|}\right]$, and hence similarity = fraction that match $= \cancel{n} \frac{|A \cap B|}{|A \cup B|} / \cancel{n}$

But, of course, variance is reduced.

Bucketing

Now, given the signatures, we divide the signature into b disjoint ~~buckets~~ bands, each with width 'r', and hash each band into k buckets.



We assume that two bands fall into the same bucket only if they are an exact match.

~~Two signatures S_1 and S_2~~

Two signatures S_1 and S_2 are considered candidates if they fall into the same bucket for at least one of the b bands.

What's the probability of false negatives?

Let's say two signatures S_1 and S_2 are similar, that is $sim(S_1, S_2) >$ threshold. Now, let $sim(S_1, S_2) = s$.

Then the probability that a band matches in entirety is s^r, i.e, all r elements in the band match.

$Pr(\text{false negative}) = Pr(\text{no bands match}) = \cancel{1 - Pr(\text{some band matches})}$ $(1 - s^r)^b$

$Pr(\text{false positive}) = Pr(\text{some band matches}) = 1 - Pr(\text{no bands match})$

$Pr(\text{false positive}) = \boxed{1 - (1 - s^r)^b.}$

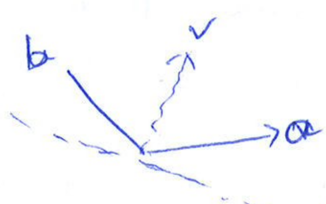
↑ S-curve used to pick r, b/

LSH for cosine similarity function

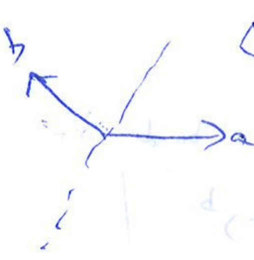
Instead of doing min-hashing (which is for Jaccard similarity), we take random projections, and then calculate the signature as

~~sign~~ $S(a) = \text{sign}(a^T v)$, where v is the random projection vector, and sign( ) is 1 if v and a are in the same direction, and 0 otherwise.

Visually.



← pick this hyperplane, check if a and b are on the same side of the plane. if yes, then the signature for both a and b would be 1. otherwise



← if you pick this plane, then the two signs will not match. Essentially the no. of matches will be proportional to the angle between the two vectors.