Recommendation Engine: Semantic Cold Start

Pablo Rodriguez Bertorello

July 2020

1 Abstract

In this document, we explore methods to provide useful recommendations until activity data is gathered. We refer to this situation as the cold start: users have no activity history, and items have no ratings. Particularly we explore how to successfully cold start recommendations by applying natural language processing. We assume that text describing users and items exists at the onset.

2 Introduction

One typical way that long-tail items get discovered, is by being associated with hot items. However, even for that, data is required to make such an association. In this document, we assume that such data is not available, and come to specify how it may need to be processed.

Sometimes, we want to provide **sequential recommendations**. Our objective is to guide the user through a sequence of activities. Alternatively, our objective may be to provide **spot recommendations**. The general types of recommendations are:

- **Editorial**: hand curated
- **Aggregates**: top 10, most popular, recent
- **Tailored**: to specific users. The bulk of the discussion will be around these

For tailored recommendation, formally we say:

- **Users**: X = { Xi }
- **Items**: S = { Sj }
- **Utility function**: u = X x S $\mapsto$ R
- **Ratings**: { Rij }

The plots of Figure 1 displays, even though in 2D, are a fair semantic representation of n-dimensional Word2Vec vectors.

	Avatar	LOTR	Matrix	Pirates
Alice	1		0.2	
Bob		0.5		0.3
Carol	0.2		1	
David				0.4

Figure 1: Sample user-item ratings Matrix[11]

The over-arching issue with this approach is the sparseness of this matrix:

- Eventually gathering **known ratings**. This may be done requiring explicit user feedback, or implicitly from user actions. The latter is a key aspect of our solution to the cold start
- Extrapolating **unknown ratings** from the known ones. These include: **content-based**, **collaborative**, and **neural latent factor**
- Evaluating extrapolation methods

3 Extrapolating Unknown Ratings

3.1 Content-based Recommendation Systems: TF-IDF

The idea is to recommend to user Xi, similar to previous items rated highly by Xi.

To quantify the similarity, one typical approach when text data is available is **TF-IDF**: term frequency, inverse document frequency. It quantifies how discriminative a particular word is.

The **term frequency** is:

- f_{ij} for term i, document j
- Max-normalized for longer documents

$TF_{ij} = f_{ij} \ / \ max_k f_{kj}$

The inverse document frequency is:

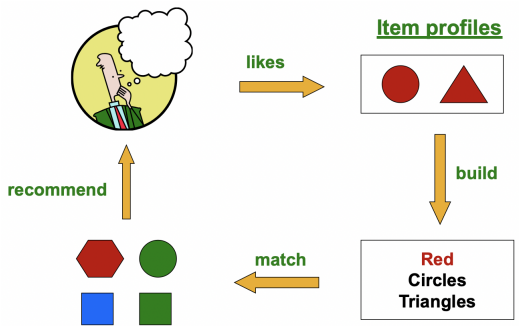

Figure 2: User-likes driven recommendations

- n_i, the number of docs that mention term i
- N the total number of docs

IDF_i = log N / n_i

Thus the TF-IDF score: $w_{ij} = TF_{ij}$ / IDF_i

An item's profile can then be formed as a vector with TF-IDF scores for each word.

Then, a user profile can then be formed from items she is associated with: The **term frequency** is either:

- Vector of average of items
- Vector of difference from the mean item rating, to compensate for variability in mean ratings across domains
- In both these cases, we can weight the vectors using any known rating by the user of the item

And to predict items the user may like, the cosine similarity of a user x to an item i can be quantified:

$$U(\vec{x}, \vec{i}) = cos(\vec{x}, \vec{i}) = \frac{\vec{x}.\vec{i}}{\|\vec{x}\|_2 * \|\vec{i}\|_2}$$

Pros and cons:

- No cold start problem, unless we know nothing about the user. If we do know, no first rater problem
- It provides good recommendations for users with unique tastes. On the other hand, it over-specializes and only recommends from what it knows about the one user

- The main drawback is that these recommendations do not benefit about any knowledge from any user other than Xi. As a benefit, no data is necessary on other users
- For text it is easy to implement, for movies/music/images it is more difficult to score
- The recommendations are explainable
- In high dimension, it is expensive to establish the similarity of every user to every item

3.2 Content-based Recommendation Systems: Word2Vec

In the same as Section3.1, instead of using TF-IDF, Word2Vec[8] could be used. In the context of this paper, we will loosely call **Word2Vec+**, the following family of algorithms:

- The two shallow neural architectures in the original paper, which can be used to learn distributed representations of words: **Continuous Bag of Words** and **Continuous Skip-gram**
- Bags of tricks for single-instance performance, including: hierarchical softmax, n-grams[3]
- Techniques that can be used to enrich word vectors with sub-word information[5]
- Methods for parallelism to compute Word2Vec[7] and [6]

Once an embedding has been learned for a word, likewise one can be computed for an item or for a user. As described above, cosine similarity may be performed. The learned vectors should be stronger than with TF-IDF. However, largely the same pros and cons apply.

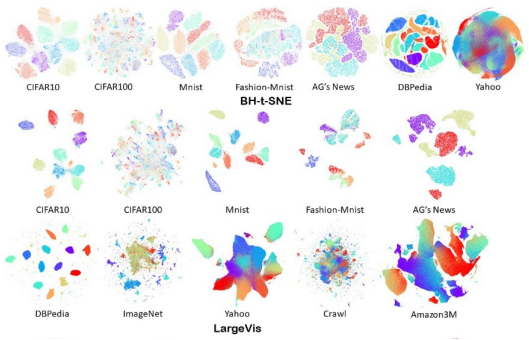

Figure 3: t-SNE plot for a variety of data sets

3.3 Collaborative Filtering

In collaborative filtering, an unsupervised method, recommendations are made on the basis of user similarities. This overcomes one of the key weaknesses of content-based systems, which recommend on the basis of previous ratings of the one user.

Consider a vector of ratings by a user x: $\vec{r}_x$. We then need to find the right measure of similarity between user rating vectors:

- Jaccard similarity is not suitable if the rating is a real number, and not just binary
- Cosine similarity has a major drawback: in a sparse vector, with most items unrated, a zero would be the default rating. For example in the context of a 5 star rating system, a zero would equate to a very bad review. However, there was no review
- With the Pearson Correlation Coefficient, the similarity is computed only on items rated by both users, and as a delta from the mean rating of each user

Figure 4: User-similarity driven recommendations

To address the compute weaknesses of content-recommendation, we can reduce the consideration set to the N users most similar to a user x. When predicting how much this user will like an item i she's never rated before we can do it by:

- Estimate by averaging the nearest-neighbor ratings
  $\hat{r}_{xi} = \frac{1}{k} \sum_{y \in N} r_{yi}$
- Or preferably estimate by weighting the nearest-neighbor rating using each user's mutual similarity score sim(x,y)
  $\hat{r}_{xi} = \frac{\sum_{y \in N} sim(x,y) * r_{yi}}{\sum_{y \in N} sim(x,y)}$

- In practice, predicted ratings include a bias term, that takes into account: overall mean ratings, as well as rating deviations of by user x, and of item i, in comparison to all users and items

In the discussion above leverages the similarity of a user to its nearest-neighbor users to predict an item rating. There is an analogous approach in which the similarity of nearest-neighbor items is leveraged to predict the rating of an unrated item. It has been observed that item-item based recommendations work better than user-user because people change over time.
Pros and cons:

- It suffers the cold start problem, need enough activity in the system before a recommendation can be made
- Works for any kind of item, no knowledge about the user/item other than a rating is necessary
- Suffers first-rating problem, can't find similarity to something not previously rated
- Tends to recommend biased by popularity, not able to perform to unique taste

Note that it is too compute expensive to find the k most similar users in real-time. Thus, it needs to be pre-computed. Nearest neighbor discovery is linear O(k *$|X|$). To make it sub-linear, **Locality Sensitive Hashing** may be applied, which reduces similarity estimates to only within clusters of items. Likewise with clustering approaches like kmeans, and dimensionality reduction approaches like SVD (further discussed below).

3.4 Neural Latent Factor Model

Here we increase performance by turning recommendations into a supervised problem[10]. We combine:

- **Local effects via Collaborative Filtering** per the solution in Section 3.3, which can be improved by machine-learning the optimal similarity measure
- **Global effects via Bias Adjustments** to take into account overall bias in user-item ratings
- **Regional effects via Neural Latent Factorization**, which also reduces dimensionality, and enables us to focus on high-ratings over low-ratings

We can improve on all the similarity measures mentioned above, and take into account global adjustments, by posing the rating estimate in this form:
$\hat{r}_{xi} = b_{xi} + \sum_{j \in N(i,x)} w_{ij}(r_{xj} - b_{xj})$

- **Baseline estimate** adjusting for bias $b_{xi} = \mu + b_x + b_i$. For a moment we assume that average is a good bias estimator. Then, μ is the overall mean rating, b_x is the user's rating deviation from user mean rating, and b_i is the item's mean rating deviation from item mean. For example: if the mean item rating is 3.7, and the rating of item i is 0.5 above average, if user x on average rates 0.2 below average...then predict $r_{xi} = 3.7 + 0.5 - 0.2 = 4.0$
- It is computed for nearest neighbors N, by item or user similarity, for compute scalability
- w_{ij} is the rating is the extrapolation coefficient to machine-learn
- $\hat{r}_{xi}$ can be learned with a neural network that minimizes RMSE, where the error is $\hat{r}_{xi} - r_{xi}$, regressing over all the ratings in the user-item matrix to find all w_{ij}

To add regional knowledge, we would like to produce our estimates as a function of singular vectors/values, i.e. per SVD. We seek to estimate our ratings matrix as follows:

$\hat{R} = QP^T$

- R's shape is items x users
- Q's shape is items x factors
- P^T shape is factors x users
- Thus a rating can be estimated as $\hat{r}_{xi} = \vec{q}_i.\vec{p}_x$
- Or, including global effects: $\hat{r}_{xi} = b_{xi} + \vec{q}_i\vec{p}_x$

SVD allows us to place any user or item in principal component space. Importantly the components are discovered, and need not be feature-engineered. A caricature of this is shown in Figure 5. Recall in SVD, a matrix is factorized as:
$\hat{R} = U\Sigma V^T$

- U has left singular vectors
- V^T has right singular vectors
- Σ has singular values
- This can be adapted to our purposes with Q = U, and $P^T = \Sigma V^T$

However, standard SVD is not a good fit for our problem, because it tries to fit all the cells in our sparse recommendations matrix. Again, as discussed above, we do not want no-rating to be interpreted as low/negative rating.

Instead, we can apply neural networks to add a twist on SVD[2], which here we'll call **SVD+**. We accomplish it by minimizing this objective function, which optimizes only for rated user-item cells:

$\min_{P,Q} \sum_{rated(i,x) \in R} (r_{xi} - \vec{q}_i . \vec{p}_x)^2$

- Alternating least squares: P $\leftarrow P - \eta \nabla P$, then Q $\leftarrow Q - \eta \nabla Q$
- Regularized for the norm of $\vec{q}_i$
- Regularized for the norm of $\vec{p}_x$
- Each gradient descent training epoch is expensive, consider using stochastic gradient descent to converge with less compute effort

Now we also seek to learn the global effect bias, instead of assuming that average is a good estimator for them. Thus, the objective function becomes:

$\min_{P,Q} \sum_{rated(i,x) \in R} [r_{xi} - (\mu + b_x + b_i + \vec{q}_i . \vec{p}_x)]^2$

- Alternating least squares: P $\leftarrow P - \eta \nabla P$, then Q $\leftarrow Q - \eta \nabla Q$
- Regularized for the norm of $\vec{q}_i$
- Regularized for the norm of $\vec{p}_x$
- Regularized for the norm of b_x
- Regularized for the norm of b_i
- Each gradient descent training epoch is expensive, consider using stochastic gradient descent to converge with less compute effort
- For now, we assume that biases are not a function of time[9]

3.5 Ensemble Models

Instead of using one best known method, combine several models. For brevity, this is not discussed further.

4 Evaluating Predictions

By performing the usual splits of data into train/test/validate, we're able to arrive at key metrics like RMSE: actual rating vs predicted rating.

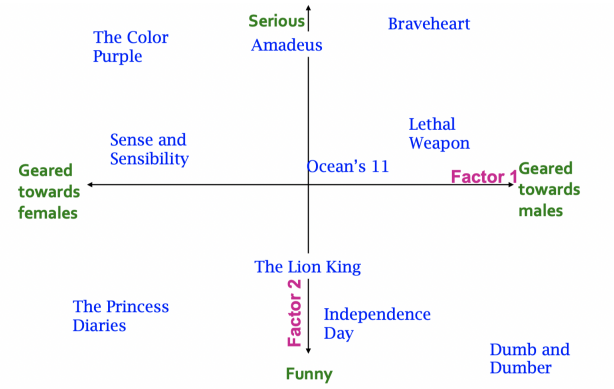

Figure 5: Sample users and items in principal component space

While this is useful for many machine learning problems, for recommendation engines this is not good enough. Particularly, we're interested in performing better for high-rating items, which users are most likely to act on, than on low-rating items.

A supplemental metric that can be used is precision on a binary classification predicting relevant/not, applied to the top k items recommended to a user. It may be challenging to make this a useful metric, as it could potentially point away from **serendipity**, which is a desired aspect of recommendation engines.

Per the **0/1 model** criterion, we need to track the percentage of users for whom we're able to produce a recommendation, on/after the day they're onboarded.

Measuring the percentage of recommendations that are in context (i.e. winter boots while browsing shoes) would tell us how often we're off left field.

The order of the predictions is also important, but since we're only discussing spot predictions in this document, we'll leave it aside.

5 Cold Start Recommendations

5.1 Minimum Viable Product 1

Because of the different pro/cons profiles of content-based and collaborative/latent systems, the best solutions tend to be a hybrid.

Because not much progress has been made on recommendation systems[4] since the well understood methods above were invented, it is reasonable to assume

these methods constitute a good starting point for a minimum viable product.

Since we are solving for the cold start spot recommendations, we do not have any history of user rating or engagement with our items. However, when we we do have text corresponding to a user and an item, an approach could be as follows:

- During the cold start, assume that a user's rating of an item can be estimated with their text, generally as in Section 3.1.
- But instead of TF-IDF, learn Word2Vec vectors, training them with the corpus of ingested text, per Section 3.2. Follow the Word2Vec+ set of best practices
- Do not build a Collaborative Filter model, dscribed in Section 3.3
- Instead, train a Latent Factor model, as described in Section3.4
- Seed the ratings matrix with r_{xi} as the dot product of the user and item vectors in Word2Vec embedding space

A word of caution: in cases with extensive use of technical terms, Word2Vec embedings should probably be trained from scratch, with the known user/item text corpora. Without this, using a general embedding as the TensorFlow Universal Encoder[1] is bound prove insufficient. It is very likely that users and items would appear generally dissimilar to each other6, and thus a poor proxy for user-item ratings.

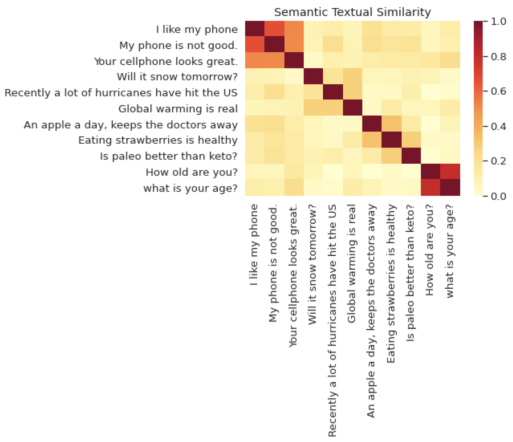

Figure 6: General encoder resulting in overall dissimilarity of text

5.2 Data Gathering for Minimum Viable Product 1

In order to evaluate the efficacy of our predictions, we need to log application activity tuples:

- $(user\_id, top\_k\_recommendations)$
- $(user\_id, item\_id, usage)$, for example the minutes a user spent watching a video
- $(user\_id, item\_id, user\_rating)$
- $(user\_id, word2vec\_vector)$
- $(item\_id, word2vec\_vector)$

These tuples can later be map-reduced for the kind of analytic described in Section 4.
One approach could be to cluster users and items and actions in embedding recommendation space, per SVD+. For this, we need to log application activity tuples:

- $(user\_id, cluster\_id)$
- $(item\_id, cluster\_id)$

This clustering information can be joined, then and map-reduced with the data of Section 5.2, to estimate any necessary transition probabilities, for example for reinforcement learning.

6 Conclusion

To cold-start a recommendation engine, a semantic natural language processing approach is recommended. User-item ratings are predicted via the cosine similarity of the Word2Vec vectors of the user and item descriptions. For technical text that cannot be approximated well by general application sentence encoders, fine-tuning Word2Vec encodings with the corpora at hand is strongly encouraged.

References

[1] *Semantic Similarity with TF-Hub Universal Encoder.* Google.

[2] Pablo Bertorello. *Latent Factor Model.* GitHub.

[3] Armand Joulin et al. *Bag of Tricks for Efficient Text Classification.* Facebook.

[4] M. Ferrary et al. *Are we Really Making Much Progress?* RecSys.

[5] Piotr Bojanowski et al. *Enriching Word Vectors with Subword Information.* Facebook.

[6] Shaurabh Gupta et al. *BlazingText: Scaling and Accelerating Word2Vecc using multiple GPUs.* Amazon.

[7] Shihao Ji et al. *Parallelizing Word2Vec in Shared Memory and Distributed Memory.* Intel.

[8] Thomas Mikolov et al. *Efficient Estimation of Word Representations in Vector Space.* Google.

[9] Y. Koren. *Collaborative Filtering with Temporal Dynamics.* KDD.

[10] Yehuda Koren. *The BellKor Solution to the Netflix Grand Prize.* Netflix.

[11] Jure Leskovec. *Mining Massive Datasets.* Stanford.