# Mapping Sparse Matrix Recommender to Streams Blending Recommender

Anton Antonov
Accendo Data, LLC

February 15, 2019

## Contents

## 1 Introduction

This document provides theoretical background for the stream merging recommendations that approximate the functionalities of the Sparse Matrix Recommender (SMR); see [AAA17].

The document briefly describes SMR and then defines streams, merging operations of streams, and recommendations computed with them.

Concrete partitioning of the metadata and similarity matrices are discussed.

In order to have clear abbreviations we are going to use the name "Stream Blending Recommender" (SBR) instead of "Stream Merging Recommender". In this document the words "merging" and "blending" mean the same when we talk about to streams.

The description of the recommenders is done through the entities "product", "maker", "user" for generality. Concrete examples corresponding to those set entities are (i) music tracks, artists and listener, (ii) services, providers, and clients, (iii) articles, authors, and readers; etc. Note that we assume that a product can have multiple makers, and a maker can make multiple products.

## 2 Notation

### 2.1 Sizes notation

The number of products is going to be denoted as $|products|$, or $n_{products}$, or when we have to save space as $np$.

Similarly, the number of makers is going to be denoted as $|makers|$, or $n_{makers}$, or when we have to save space as $nm$.

When we want to speak about both products and makers we are going to use the term "items" and use the notation $|items|$ or $n_{items}$ to denote the number of items.

The total number of metadata is denoted as $n_{metadata}$ or when we have to save space as $n_{md}$.

The length of a vector $v$ is going to be denoted as $|v|$. (Note this is different from the norm of the vector, denoted as $\|v\|$.)

### 2.2 Sub-matrix notation

We are going to use the following notation for sub-matrices of the matrix $M$.

- The sub-matrix specification has the following general form:

$$M(index\ range\ specification, index\ range\ specification).$$

- An element of the vector $v$ is denoted with $v(i)$ or $v_i$ where $i$ is an integer.
- An entry of the matrix $M$ is denoted as $M(i,j)$ where $i$ and $j$ are integers.
- If we want to specify a sub-matrix formed by the rows from 1 to 10 of $M$ and all columns of $M$ we use

$$M(1:10,:) \text{ or } M(1:10)\,.$$

- If we want to specify a sub-matrix formed by the first 20 columns of $M$ we use

$$M(:,1:20)\,.$$

- A sub-matrix of $M$ formed by its first ten rows and last five columns is specified as

$$M(1:10,-5:-1)\,.$$

- If we want to specify a sub-matrix formed by the rows 1, 2, 8, 23, 26 and all columns of $M$ we use

$$M(\{1,2,8,23,26\},:)\ \textit{or}\ M(\{1,2,8,23,26\})\,.$$

- Similarly, for a sub-matrix formed by all rows of $M$ and the columns 5,7, and 26 we use:

$$M(:,\{5,7,26\})\,.$$

## 2.3 Linear algebra notation

We write $diag(a_1,\ldots,a_n)$ for a ***diagonal matrix*** whose diagonal entries starting in the upper left corner are $a_1,\ldots,a_n$.

# 3 General theory of SMR

## 3.1 The main idea

We have several types of metadata characterizing the products. If the products are music tracks, the metadata types are genres, decades, creators, acoustic features, and others. Let us denote the set of metadata types with $TT := \{type_1, type_2, \ldots\}$. ("TT" stands for "tag type".)

For each relation between the set of items (songs, shows, makers, ...) $S$ and the set $MD(t)$ of metadata from a given type $t$, $t \in TT$, we form a metadata matrix $M(t)$. The relationship (association) between an item and a tag (metadatum) can be weighted (using, for example, the inverse document frequency formula). The matrices are normalized so the maximum elements are 1.

The normalized matrices are spliced into one matrix $M^{01}$ in the following way:
for $i \in \{1,|items|\}$ the $i$-th row of $M^{01}$ is obtained by splicing the $i$-th rows of the matrices $\{M(t)\}_{t\in TT}$.

Let us denote with $n_{metadata}$ the sum

$$n_{metadata} := \sum_{t\in TT} |MD(t)|.$$

The matrix $M^{01}$ has dimensions $|items| \times n_{metadata}$. The matrix $M^{01}$ represents all items, each row corresponds to an item. $M^{01}$ is assumed to be (very) sparse.

In order to be able to tune the recommender we want to assign different significance factors to the metadata. In order to do that we select a weight $\omega_t$ for each type of metadata $t \in TT$. Then using the notation $n_t := |MD(t)|$ we construct the **tag type significance weight vector** $w$ in the following way:

$$w := \{\underbrace{\omega_{type_1},\ldots,\omega_{type_1}},\underbrace{\omega_{type_2},\ldots,\omega_{type_2}},\ldots,\underbrace{\omega_{type_{|TT|}},\ldots,\omega_{type_{|TT|}}}\}.$$

Then we form the diagonal matrix $diag(w)$ and use matrix-martix multiplication to obtain weighted sparse matrix item-metadata associations:

$$M := M^{01} diag(w) = M^{01} \left\{ \begin{array}{ccc} w(1) & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & w(n_{metadata}) \end{array} \right\}. \tag{1}$$

### 3.1.1 Similarity matrix

If we multiply the metadata matrix with its transpose we are going to obtain the item-item similarity matrix $S$:

$$S := M\ M^T. \tag{2}$$

The similarity matrix as defined in 2 is symmetric. If we divide the rows $S$ by the diagonal elements, $\widetilde{S} := 1/diag(s)\,S$, then the resulting matrix $\widetilde{S}$ is no longer symmetric.

### 3.1.2 Metadata space

We are going to call the vector space made of all metadata tags ***metadata space***.

### 3.1.3 Recommendations

The matrix $M$ is used to compute the recommendations. First, with a vector $v$ in the vector space $R^{|items|}$ we represent a consumption history of a user. The recommended items are the items with the highest coordinates in the vector:

$$S\,v = (M\,M^T)v = M(v\,M)^T = M^{01}diag(w^2)(v\,M^{01})^T. \tag{3}$$

If we have a user profile (which is a set of metadata tags) $p \in R^{|metadata|}$ then the recommendations for that user are the items with the highest coordinates in the vector $M\,p$.

### 3.1.4 Metadata sub-matrices

The metadata matrix $M$ by definition is constructed by matrices corresponding to different metadata types. Given a metadata data type $t_0 \in TT$ we are going to use the notation $M(:, t_0)$ or $M_{t_0}$ to denote the sub-matrix of $M$ corresponding to the type $t_0$. We can go further and denote the metadata matrix corresponding to two (or more) metadata types $t_1, t_2 \in TT$, with $M(:, \{t_1, t_2\})$ and $M_{\{t_1,t_2\}}$.

Let us list some of the metadata sub-matrices of a music recommender:

- $TR_{genre}$ tracks×genres metadata matrix;
- $TR_{maker}$ tracks×makers metadata matrix;
- $TR_{acoustic}$ tracks×acoustics metadata matrix;
- $TR_{release\,date}$ tracks×release-date metadata matrix;
- $TR_{trend}$ tracks×trends metadata matrix;
- $AR_{genre}$ artists×genre metadata matrix;
- $AR_{decades}$ artists×decades metadata matrix;
- $TR_{\{genre,popularity\}}$ tracks×{genre,popularity} metadata matrix;
- and others...

It is important to keep in mind what are the spaces corresponding to those matrices. In some cases it will be convenient to work with the transposed matrices. (So we can form item streams based on metadata tags, see below.)

### 3.1.5 Similarity sub-matrices

Similar to the partitioning of the metadata matrix $M$ we can partition the similarity matrix $S$ into sub-matrices that are derived from one or several metadata types.

Let us list some the similarity sub-matrices:

- $pSR_{maker}$ products×products similarity matrix based on common makers;
- $pSR_{timestamp}$ products×products similarity matrix based on timestamps;
- $mSR_{\{location,trend\}}$ makers×makers similarity matrix based on maker locations and trends,
- etc.

**Remark:** The similarity sub-matrices do not need to be derived with formula (2), they might be externally specified.

### 3.2 Mapping between metadata spaces

Consider the graph made with the associations between the products and the makers. We can look at that graph as a bi-partite graph: one set of nodes for corresponds to the products, and the other set of nodes corresponds to the makers. When we consider sub-graphs by edge types we (probably) most of the time will deal with one set of nodes. It is useful to consider the operations of mapping between the two sets.

For example, the set of makers $nn(a)$ that are the nearest neighbors of the maker $a$ can be mapped into their products, $products(nn(a))$. The function $products(a)$ maps the maker $a$ to his products, the function $makers(t)$ maps the product $t$ into its makers.

The functions *products* and *makers* are derived from the edges of the products-makers bi-partite graph. The application of these functions over streams is going to be defined below.

A good example of a product-maker bi-partite graph and a related, useful matrix-vector multiplication is given in [AAA15].

### 3.3 Graph interpretations

The considered Products-Makers Graph (PMG) allows the nodes to be connected with multiple edges. Each edge has its own attributes. We can postulate that each edge has a type. (Type is one of the attributes.) Therefore, PMG is comprised of sub-graphs corresponding to the different types of the graph edges.

The metadata sub-matrices of SMR correspond to sub-graphs defined by the edges types.

## 4 Mapping into the products-makers graph

### 4.1 The main idea

The main idea is to use operations over pre-computed scored lists of items (products or makers) in order to replace $S\,v$ in (3). These kind of scored lists are called streams. The streams are derived from disjoint, or almost disjoint, metadata spaces.

### 4.2 Streams

A ***stream*** is a scored list of items (names or item ID's) with a descending order of the scores. Here is an example of a stream:

$$\left\{ \begin{array}{ccc} 4.2 & 34322 & Get\ some \\ 4.0 & 23212 & Heads\ will\ roll \\ 3.6 & 11221 & Big\ brat \\ 2.8 & 34232 & Everythin\ Zen \\ 2.7 & 34323 & I\ can\ levitate \end{array} \right\}.$$

**Definition:** Given a stream $s$ the operators $scores(s)$ and $ids(s)$ return the scores and the ID's of $s$ respectively. ■

**Definition:** The ***merging*** of two streams $s_1$ and $s_2$ into $s_3$ is defined in the following way:

1. the items of $s_3$ are the union of the items of $s_1$ and $s_2$;
2. the items of $s_1$ are matched with the items of $s_2$;
3. the scores of the items in $s_3$ are obtained by summing the scores of the ID's that match, and using the scores from $s_1$ or $s_2$ for the ones that do not match.

Let us denote with $merge(s_1, s_2)$ the merging of the two streams $s_1$ and $s_2$. Now we can write $s_3 = merge(s_1, s_2)$. ■

It is important see the stream merging as a sum of two sparse vectors $\vec{s_3} = \vec{s_1} + \vec{s_2}$ in the space of items $R^{|items|}$ followed by the ordering of the non-zero coordinates of $\vec{s_3}$ in descending order. This observation can be used as a justification that we can approximate the SMR functionalities by using streams.

**Remark:** The sum of two sparse vectors does involve finding the intersection between the indices of the non-zero coordinates.

We can go further and represent by stream merging the ***weighted sum*** of two vectors $\alpha\,\vec{s_1} + \beta\,\vec{s_2}$: we just have to multiply the scores of $s1$ with $\alpha$ and the scores of $s_2$ with $\beta$. Because of this observation we are going to denote with $\alpha\,s$ the multiplication with $\alpha$ of the scores of the stream $s$, an we can write

$$s_3 = merge(\alpha\,s_1, \beta\,s_2).$$

### 4.2.1 *Writing remarks*

1. I think we can go further and provide mathematical and algorithmic details of the approximation.
2. It is probably a good idea to provide a dictionary appendix of all notions and notation introduced in this document.
3. *merge* probably has to be replaced with *mergesum*.

## 4.3 Stream operations

Assume that any stream $s$ can be represented as a vector $\vec{s} \in R^{|items|}$.

**Definition:** The operation $ord(\vec{s})$ takes a vector $\vec{s} \in R^n$ and orders $\vec{s}$ into a scored list in which the coordinates of $\vec{s}$ are paired with their corresponding axes and ordered in a descending order. More precisely, if

$$v := ord(\vec{s}),$$

then

$$v = \left\{ \begin{array}{cc} v_{i_1} & i_1 \\ v_{i_2} & i_2 \\ \vdots & \vdots \\ v_{i_n} & i_n \end{array} \right\},$$

$$v_{i_1} \geq v_{i_2} \ldots \geq v_{i_n}.$$

Further, we can enhance the function $ord$ to take only the top $k$ of the elements, so if $k \leq n$ then

$$v := ord(\vec{s}, k)$$

$$v = \left\{ \begin{array}{cc} v_{i_1} & i_1 \\ v_{i_2} & i_2 \\ \vdots & \vdots \\ v_{i_k} & i_k \end{array} \right\},$$

$$v_{i_1} \geq v_{i_2} \ldots \geq v_{i_k} \geq \ldots \geq v_{i_n}.$$

■

**Definition:** Using the vector representation of streams we define the following operations

$$scalarmult(\alpha, s) \equiv \alpha * s := \alpha\,\vec{s},$$

$$mergemax(s_1, s_2, k) := ord(\{max(s_1(i), s_2(i))\}_i, k),$$

$$mergesum(s_1, s_2, k) := ord(\vec{s_1} + \vec{s_2}, k),$$

$$mergemult(s_1, s_2, k) := ord(\vec{s_1} * \vec{s_2}, k) = ord(\{s_1(i) * s_2(i)\}_i, k),$$

$$mergedot(s_1, s_2) := \vec{s_1}.\vec{s_2} = \sum(\vec{s_1} * \vec{s_2})\ .$$

The stream merging functions can be naturally extended to take multiple stream arguments not just two. Here is an example of such a definition of *mergesum* with multiple stream arguments:

$$mergesum(s_1, s_2, \ldots, s_n, k) := ord(\vec{s_1} + \vec{s_2} + \ldots + \vec{s_n}, k)\ .$$

■

**Remark:** Note, that instead of the notation of equivalence $scalarmult(\alpha, s) \equiv \alpha * s$ we can assume that all stream operations can make the multiplication of streams with scalars, *scalarmult*. In both cases instead of

$$mergesum(scalarmult(\alpha, s_1), scalarmult(\beta, s_2), k)$$

we can just write

$$mergesum(\alpha * s_1, \beta * s_2)\,.$$

**Remark:** Note that *mergesum* is equivalent to *merge* defined in the previous section.

**Remark:** In order to make the exposition easier we are going to assume that stream merging functions work also over sparse vectors. Sparse vectors are very often stored in a format directly corresponding to a stream (without the descending order).

## 4.4 Stream action extensions of the graph mapping functions

The definitions of the functions *products* and *makers* can be extended so they can be applied to streams.

**Definition:** For every maker (product) ID $i$ the function *products* (*makers*) returns a stream of the products (makers) of $i$. This defines *products* (*makers*) as a function from the space of ID's to the space of streams.

■

**Definition:** Given a stream $s_a$ of makers and a stream $s_t$ of products the function applications $products(s_a)$ and $makers(s_t)$ are defined in the following way:

$$products(s_a) := mergesum(\{products(i)\}_{i \in ids(s_a)})\,,$$

$$makers(s_t) := mergesum(\{makers(i)\}_{i \in ids(s_t)})\,.$$

■

## 4.5 Using streams instead of vector-matrix multiplication

The main operation of SMR, the recommendations calculation, is done with the formula (3). Vector-vector multiplication can be implemented using the streams operation *mergedot*. Hence, we can implement vector-matrix and matrix-vector operations with streams. This means that we can implement formula (3) with streams.

In order to replace sparse matrix linear algebra operations in (3) with streams operations we need to do some profiling of both types of operations.

For example, assume that $M \in R^{m \times n}$, and $m = 100,000$ and $n = 10,000$. (We have $m$ items described by $n$ tags.) Then for the vector-matrix multiplication $v\,M$ we have to make $n = 10,000$ stream operations of the type *mergedot* each being between two sparse vectors of dimension $m = 100,000$.

Obviously, the operation *mergedot* has the same computational complexity as a sparse vector-matrix multiplication. If we denote with $nnz(v)$ and $nnz(M(:,i))$ the number of the non-zero elements of $v$ and the $i$-the column of $M$ respectively, then the computational complexity is

$$n\,O(2(nnz(v) + nnz(M(:,i)) - 1). \tag{4}$$

If we assume that $nnz(v) \approx 1000$ (a user history of thousand items) and $nnz(M(:,i)) \approx m/n = 10$ (the tags are disjoint on average) then for formula (4) we will have the value $\approx 20,190,000$.

At this point we will assume that using streams for all types of vector-matrix multiplications is too expensive.

### 4.6 Using streams of similarities

Probably the most efficient application of streams is to the product $S\,v$, in which the similarity matrix $S$ is pre-computed. Each row of $S$ is a stream. When the number of non-zero coordinates of $v$ is relative small then $S\,v$ can be done quickly with *mergesum*.

With the mapping functions (see 3.2 and 4.4) we can enhance the utilization of the metadata and similarity (sub-)matrices.

For example, given a set of makers $A = \{a_1, \ldots, a_n\}$ we find the products by temporally similar makers with

$$products(mergesum(mSR_{timestamp}(A, :)))\,. \tag{5}$$

Here is the breakdown of formula (5):

1. using $mSR_{timestamp}(A, :)$ take the sub-matrix of $mSR_{timestamp}$ defined by the rows $\{a_1, \ldots, a_n\}$ ;
2. using $mergesum(mSR_{timestamp}(A, :))$ merge the streams derived from the rows of the sub-matrix $mSR_{timestamp}(A, :)$ ;
3. using *products* we map from the space of maker streams into the space of product streams (obtaining one stream as a result).

### 4.7 Approximation by separation into primitive streams

In order to approximate SMR recommendations with streams we are going to define similarity and metadata matrices that are computed offline, and then we are going to list the stream merging operations that are computed online.

The main challenge to approximate formula (3) with stream merging is to device a way to approximate the dynamic similarity provided by (3) through the multiplication with the diagonal matrix $diag(w^2)$; see also formula (1). The tag weights $w$ can be different every time we calculate recommendations with (3).

One way to approximate the dynamic similarity is to partition the similarity matrix into sub-matrices across the columns using sets of metadata $\{md_i\}_{i=1}^{k}$ that are disjoint, as in this formula:

$$S = [S(:, md_1), S(:, md_2), \ldots, S(:, md_k)]\,.$$

Then using stream merging with different weights for the streams derived from the sub-matrices $S(:, md_i)$ we can provide dynamic similarity that approximates the one provided by (3).

#### 4.7.1 Offline computations

If we assume that $S$ is for music track$\times$music track similarities, then one natural selection of the collection of sets $\{md_i\}_{i=1}^{k}$ is the following:

1. $md_1$ all of the artist names tags;
2. $md_2$ all of the composer names tags;
3. $md_3$ all of the producer names tags;
4. $md_4$ all of the label names tags;
5. $md_5$ all of the genre tags
6. $md_6$ all of the release years tags;
7. $md_7$ all of the acoustic tags;
8. $md_8$ all of the popularity tags;
9. $md_9$ all of the trendiness tags.

The matrices $S(:, md_5) = S(:, genre)$ and $S(:, md_6) = S(:, release\ years)$ can be quite dense, and hence each with a large number of non-zero elements. To see this consider the metadata matrix $M(genre)$. Because every track-genre association weight in $M(:, genre)$ is binary (either 0 or 1) and because many products might have the same genres for every track we would have a very large number of nearest neighbors with the same similarity scores. (The similarity scores are computed with $S(:, genre) = M(:, genre)\, M(:, genre)^T$. ) Hence for every row we have to keep all non-zero entries and their number is going to be a significant fraction of $|products|$.

One way to reduce the number of nearest neighbors in a row of $M(:, genre)$ is to combine the track genres with another metadata type that provides association weights with great variance. Let us consider two cases.

1. In the metadata type "popularity" we observe an exponential distribution of the track popularities. So we can replace $S(:, genre)$ and $S(:, release\ years)$ with the similarity matrices $S(:, \{genre, popularity\})$ and $S(:, \{release\ years, popularity\})$ respectively. In the matrices $S(:, \{genre, popularity\})$ and $S(:, \{release\ years, popularity\})$ we drop the entries below a certain threshold or we keep the largest $l \in N$ entries for each row.

2. The "overall" similarity, that combines the tags of all metadata types and assigns weights to them with IDF, would be non-constant and expected to greatly vary for each item. We can compute the similarity matrix $S$ in such a way that the columns of $M$ that correspond to the metadata type "genre" have weights several orders of magnitude larger than the rest. Then for each row of $S$ we take the entries above a certain threshold or we take the largest $l \in N$ entries. Note that since the "genre" tags have large weights, the resulting similarity matrix is still going to be quite disjoint from the rest.

The separation, computation, and additional transformation of the similarity matrices are done offline.

#### 4.7.2 Online computations: the approximation formula

After the preparation of the similarity matrices we are ready to compute the top $tn \in N$ recommendations for a given product ID $tid$ using the significance factors $\{f_i\}_{i=1}^k$ corresponding to the metadata sets $\{md_i\}_{i=1}^k$:

$$mergesum(f_1 * S(tid, md_1), \ldots, f_k * S(tid, md_k), tn)\,. \tag{6}$$

Formula (6) gives the desired approximation of (3). The dynamic similarity provided by (6) is on a coarser level than (3) because (6) uses pre-computed similarity sub-matrices.

### 4.8 User profile calculation with streams

TBD...

## 5 Concrete steps

### 5.1 Verification of the proposed operations

The following stream operations can be represented/computed in the graph:

- $ord(s)$: the stream $s$, assuming it corresponds to an item $i$ whose vertex in the graph is $v_i$, will be represented in the graph as a set of edges outgoing from $v_i$, and each of such edges will have a property with the corresponding score. Titan has the ability to create vertex-centric indices on edge properties, which brings significant performance improvements. Hence, the edges that represent $s$ will be already sorted by score when retrieved from the graph. This means that there will be no need to compute $ord(s)$ on runtime. In addition, it is not store in the graph all the elements in $s$. Instead, the graph will only store $ord(s, k)$.

- $scarlarmult(\alpha, s)$: will be computed on $s$, runtime after reading $ord(s, k)$ in memory.

- $mergemax(s_1, s_2, k)$, $mergesum(s_1, s_2, k)$, $mergemult(s_1, s_2, k)$, and $mergedot(s_1, s_2, k)$ : will be computed on runtime after reading $ord(s_1, k)$ and $ord(s_2, k)$.

- Each maker or product is represented as vertex in the graph, where a maker has an edge to each of its products, and a product as an edge to each of makers. Hence, $products(s_a)$ is computed by retrieving the vertices that correspond to $s_a$, and then retrieving the list of products for each vertex. The function $makers(s_t)$ is computed in a similar fashion.

All operations can be computed/stored in the graph as long as we only deal in terms of $ord(s, k)$.

## 5.2 Which streams to create?

### 5.2.1 *Writing remarks*

- Kind of answered already at the end of the previous section.

### 5.2.2 Product centric

- products for a maker
- Albums for a maker
- products for a related set

### 5.2.3 Maker centric

# 6 Using stream merging for sequential product consumption

Let us call a sequence of products a "playlist". (Consider the case in which the products are music tracks, the makers are artists.)

## 6.1 Preliminaries

1. Playlist mining
   (a) From the playlist mining we have N-gram models or prefix tree rules (of conditional probability type).
   (b) We have several conditional probability predictors, like,
      i. top tag predictor using the previous $k_1$ top tags;
      ii. maker popularity predictor using the previous $k_2$ maker popularities;
      iii. ...
   (c) We have separation rules derived over different metadata tags.
2. Streams
   (a) We have an implementation of retrieval of makers, products, and product-sets streams based on one or several metadata tags.
   (b) We have an implementation of stream merging.
3. Notation
   (a) The iterative process of playlist generation builds the playlist $L$.
   (b) We have predictors $P_i$ $i \in [1, n_p]$ for the tag types $tt_i \in TTP$.
   (c) For each $tt_i$ we can convert $L$ into the list $L_i$ of tags of type $tt_i$. Denote the function performing that conversion as $totags(tt_i, L)$ or $totags_i(L)$ for short. (The function *totags* converts its playlist argument into a sequence of tags of type $tt_i$. )
   (d) Each predictor $P_i$ is applied to the converted current playlist $totags_i(L)$ and the wall-clock time, $t_c$. The predictors are functions defined as $P_i : \{tt_i\} \times R^+ \rightarrow tt_i$.
   (e) The separation rules $R_j$ $j \in [1, n_r]$ are represented as functions with arguments $L$ and wall-clock time $t_c$ and with return values products that cannot be inserted into the current playlist $L$ as a next product.

## 6.2 Algorithm

### 6.2.1 Next product step

The fundamental step of the algorithm is the derivation of the next product in the playlist being built.

1. For each tag type $tt_i$ with predictor $P_i$ predict the next tag with $P_i(totags_i(L), t_c)$.
2. For each $tt_i \in TTP$ form a stream corresponding to tag $P_i(totags_i(L), t_c)$.
3. Merge the streams
$$S := mergesum(stream(P_1(totags_1(L))), \ldots, stream(P_{n_p}(totags_{n_p}(L)))).$$
4. Remove from $S$ the products that are in any of the penalty boxes. I.e. find the complement of $S$ the union the products produced separation rules.
$$\bar{S} = S \setminus \bigcup_{j=1}^{n_r} R_j(L).$$
5. Using $\bar{S}$ make a random weighted choice. The stream scores in $\bar{S}$ are used as weights for the random choice.

**Remark:** The streams can have assigned weights. This needs further investigation and it is not detailed here at this time.

### 6.2.2 The playlist generation loop

# 7 Mapping stream scores to ranks

## 7.1 Problem set up

Assume we have several streams with items of the same type (makers, products). The scores of the streams are computed with different algorithms or taken from different data sources. Because of this the scores of the streams have different ranges and distributions. Without preliminary normalization the merging of these streams might produce unexpected results. But even with rescaling and normalization the merging of the streams would depend too much on the scores distributions within the streams.

In order to resolve the outlined problems for each stream we map the scores to ranks derived from a quantile partitioning of the scores. We use the same number of quantiles for each stream.

## 7.2 General strategy

Assume we have (i) $n_{items}$ number of items, and (ii) $na$ number of algorithms for stream production and these algorithms are denoted with $\alpha_1, \alpha_2, \ldots, \alpha_{na}$. We have a stream corresponding to each algorithm. The score of the item $i \in [1, \ldots, n_{items}]$ in the stream $s_k$, $k \in [1, \ldots, na]$ is denoted with $sc_k(i)$.

The description of the general strategy follows.

First, we map the scores computed by each algorithm $\alpha_k$ into a set of relatively small number of integers $[1, \ldots, nq]$. (For example, $nq = 20$.) We propose for each algorithm $\alpha_k$ to use as a mapping function a piecewise constant function $\pi_k$: $\mathbb{R}^+ \to \mathbb{N}$ generated by a set of quantiles for the set $sc_k(i), i \in [1, \ldots, n_{items}]$. The larger the score is, the larger the integer to which it is mapped to. More precisely,

$$sc_k(i) \leq sc_k(j) \Rightarrow \pi_k(sc_k(i)) \leq \pi_k(sc_k(j)).$$

Given a stream $s$ we denote with $\pi_k(s)$ the stream derived from $s$ with scores mapped by $\pi_k$.

After the mapping of all scores obtained by all algorithms, we can filter the streams.

1. From each stream $\pi_k(s_k)$, $k \in [1, \ldots, na]$ we take the top $m$ items.
2. Drop the items for which $\pi_k(sc_k(i))$, $k \in [1, \ldots, na]$ ,$i \in [1, \ldots, n_{items}]$ is too smaller than a certain (global) number.

We are going to call the mapping described above ***quantile mapping***, and call the numbers $\pi_k(sc_l(i))$ ***quantile scores***. Below is given a detailed description of the quantile scores calculation.

**Remark:** If the streams are stored in a database table then the mappings with the functions $\pi_k$ can be stored using an additional column.

## 7.3 Calculation of the quantile scores

1. For each stream $s_k$ find the set of scores $sc_k$.
2. Choose a natural number $nq \in \mathbb{N}$. This is a parameter. For each stream $s_k$ we are going to form a piecewise constant function $\pi_k : \mathbb{R}^+ \to \mathbb{N}$ of $nq + 1$ pieces.
3. For each set of scores $sc_k$ calculate $nq$ quantiles.
   (a) The quantiles of $sc_k$ form an array $p_k$ of $nq$ numbers.
4. For each stream $s_k$ (with each array $p_k$) create the mapping function:

$$\pi_k(t) := \begin{cases} 0 & t \leq p_{k,1} \\ 1 & p_{k,1} < t \leq p_{k,2} \\ \dots & \dots \\ nq & p_{k,nq} \leq t \end{cases} .$$

5. For each stream $s_k$ map the scores of $s_k$ with $\pi_k$. This can be expressed more precisely with

$$sc_k(i) \to \pi_k(sc_k(i)), i \in [1, \dots, n_{items}], k \in [1, \dots, na].$$

## References

[AAA17] Anton Antonov, “A Fast and Agile Item-Item Recommender: Design and Implementation”, 2011, URL: http://www.wolfram.com/events/technology-conference/2011/presentations/AFastAndAgileIIR.cdf .

[AAA15] Anton Antonov, “RSparseMatrix for sparse matrices with named rows and columns”, 2015, URL: https://mathematicaforprediction.wordpress.com/2015/10/08/rsparsematrix-for-sparse-matrices-with-named-rows-and-columns/ .