Tutorial on How to Fit Latent Factor Models

Bee-Chung Chen

January 5, 2012

This document describes how you can fit latent factor models using the open source package developed in Yahoo! Labs.

```
Stable repository: https://github.com/yahoo/Latent-Factor-Models
Development repository: https://github.com/beechung/Latent-Factor-Models
```

1 Preparation

Before you can use this code to fit any model, you need to install R (with R version $\geq$ 2.10.1) and compile the C/C++ code in this package.

1.1 Install R

To install R, go to `http://www.r-project.org/`. Click CRAN on the left panel. Pick a CRAN mirror. Then, install R from the R source code.

Alternatively, you can install R using linux's package management software. In this case, please install `r-base`, `r-base-core`, `r-base-dev`, `r-recommended`.

After installing R, install the following R packages: `Matrix` and `glmnet`. Notice that these two packages are not required if you do not need to handle sparse feature vectors or matrices. To install these R packages, use the following commands in R.

```
> install.packages("Matrix");
> install.packages("glmnet");
```

1.2 Be Familiar with R

This tutorial assumes that you are familiar with R, at least comfortable reading R code. If not, please read
`http://cran.r-project.org/doc/manuals/R-intro.pdf`.

1.3 Compile C/C++ Code

This is extremely simple. Just type `make` in the top-level directory (i.e., the directory that contains LICENSE, README, Makefile, Makevars, etc.).

2 Bias-Smoothed Tensor Model

The bias-smoothed tensor (BST) model [2] includes the regression-based latent factor model (RLFM) [1] and regular matrix factorization models as special cases. In fact, the BST model presented here is a bit more general than the model presented in [2]. In the following, I demonstrate how to fit such a model and its special cases. The R code of this section can be found in `src/R/examples/tutorial-BST.R`.

2.1 Model

We first specify the model in its most general form and then describe special cases. Let y_{ijkpq} denote the *response* (e.g., rating) that *source node* i (e.g., user i) gives *destination node* j (e.g., item j) in *context* (k, p, q), where the context is specified by a three dimensional vector:

- *Edge context* k specifies the context when the response occurs on the edge from node i to node j; e.g., the rating on the edge from user i to item j was given when i saw j on web page k.
- *Source context* p specifies the context (or mode) of the source node i when this node gives the response; e.g., p represents the category of item j, meaning that user i are in different modes when rating items in different categories.
- *Destination context* q specifies the context (or mode) of the destination node j when this node receives the response; e.g., q represents the user segment that user i belongs to, meaning that the response that an item receives depends on the segment that the user belongs to.

Notice that the context (k, p, q) is assumed to be given and each individual response is assumed to occur in a single context. Also note that when modeling a problem, we may not always need all the three components in the three dimensional context vector.

Because i always denotes a source node (e.g., a user), j always denotes a destination node (e.g., an item) and k always denotes an edge context, we slightly abuse our notation by using $\boldsymbol{x}_i$ to denote the feature vector of source node i, $\boldsymbol{x}_j$ to denote the feature vector of destination node j, $\boldsymbol{x}_k$ to denote the feature vector of edge context k, and $\boldsymbol{x}_{ijk}$ to denote the feature vector associated with the occasion when i gives j the response in context k.

Response model: For numeric response, we use the Gaussian response model; for binary response, we use the logistic response model.

$$y_{ijkpq} \sim \mathcal{N}(\mu_{ijkpq},\ \sigma_y^2) \text{ or } y_{ijkpq} \sim \mathit{Bernoulli}((1 + \exp(-\mu_{ijkpq}))^{-1}),$$

where $\mu_{ijkpq} = \boldsymbol{x}'_{ijk}\boldsymbol{b} + \alpha_{ip} + \beta_{jq} + \gamma_k + \langle \boldsymbol{u}_i, \boldsymbol{v}_j, \boldsymbol{w}_k \rangle$. Note that $\langle \boldsymbol{u}_i, \boldsymbol{v}_j, \boldsymbol{w}_k \rangle = \sum_\ell \boldsymbol{u}_i[\ell]\, \boldsymbol{v}_j[\ell]\, \boldsymbol{w}_k[\ell]$ is a form of the tensor product of three vectors $\boldsymbol{u}_i$, $\boldsymbol{v}_j$ and

$\boldsymbol{w}_k$, where $\boldsymbol{u}_i[\ell]$ denotes the ℓth element in vector $\boldsymbol{u}_i$. For ease of exposition, we use the following notation to represent both the Gaussian and logistic models.

$$y_{ijkpq} \sim \boldsymbol{x}'_{ijk}\boldsymbol{b} + \alpha_{ip} + \beta_{jq} + \gamma_k + \langle \boldsymbol{u}_i, \boldsymbol{v}_j, \boldsymbol{w}_k \rangle \,, \tag{1}$$

where $\boldsymbol{b}$ is the regression coefficient vector on feature vector $\boldsymbol{x}_{ijk}$; α_{ip} is the latent factor of source node i in source context p; β_{jq} is the latent factor of destination node j in destination context q; γ_k is the latent factor of edge context k; $\boldsymbol{u}_i$, $\boldsymbol{v}_j$ and $\boldsymbol{w}_k$ are the latent factor vectors of source node i, destination node j and edge context k, respectively. Note that these latent factors and regression coefficients will be learned from data.

Regression Priors: The priors of the latent factors are specified in the following:

$$\alpha_{ip} \sim \mathcal{N}(\boldsymbol{g}'_p\boldsymbol{x}_i + q_p\alpha_i,\ \sigma^2_{\alpha,p}), \qquad \alpha_i \sim \mathcal{N}(0,1) \tag{2}$$

$$\beta_{jq} \sim \mathcal{N}(\boldsymbol{d}'_q\boldsymbol{x}_j + r_q\beta_j,\ \sigma^2_{\beta,q}), \qquad \beta_j \sim \mathcal{N}(0,1) \tag{3}$$

$$\gamma_k \sim \mathcal{N}(\boldsymbol{h}'\boldsymbol{x}_k,\, \sigma^2_\gamma I), \tag{4}$$

$$\boldsymbol{u}_i \sim \mathcal{N}(G'\boldsymbol{x}_i,\, \sigma^2_u I), \quad \boldsymbol{v}_j \sim \mathcal{N}(D'\boldsymbol{x}_j,\, \sigma^2_v I), \quad \boldsymbol{w}_k \sim \mathcal{N}(H'\boldsymbol{x}_k,\, \sigma^2_w I), \tag{5}$$

where $\boldsymbol{g}_p$, q_p, $\boldsymbol{d}_q$, r_q, G, D and H are regression coefficient vectors and matrices. These regression coefficients will be learned from data and provide the ability to make predictions for users or items that do not appear in training data. The factors of these new users or items will be predicted based on their features through regression.

2.2 Toy Dataset

In the following, we describe a toy dataset. You can put your data in the same format to fit the model to your data. This toy dataset is in the following directory:

```
test-data/multicontext_model/simulated-mtx-uvw-10K
```

Please read the README file there to better understand this dataset, which was created by running the following R script. Please do not rerun this R script.

```
src/unit-test/multicontext_model/create-simulated-data-1.R
```

This is a simulated dataset; i.e., the response values y_{ijkpq} are generated according to a ground-truth model. To see the ground-truth, run the following commands in R.

```
> load("test-data/multicontext_model/simulated-mtx-uvw-10K/ground-truth.RData");
> str(factor);
> str(param);
```

Response Data: The response data, also called observation data, is in `obs-train.txt` and `obs-test.txt`. Each file has six columns:

1. `src_id`: Source node ID (e.g., user i).
2. `dst_id`: Destination node ID (e.g., item j).
3. `src_context`: Source context ID (e.g., source context p).
4. `dst_context`: Destination context ID (e.g., destination context q).
5. `ctx_id`: Edge context ID (e.g., edge context k).
6. `y`: Response (e.g., the rating that user i gives item j in context (k, p, q)).

Note that all of the above IDs can be numbers or character strings. To read `obs-train.txt`, run the following commands in R.

```
> input.dir = "test-data/multicontext_model/simulated-mtx-uvw-10K"
> obs.train = read.table(paste(input.dir,"/obs-train.txt",sep=""),
  sep="\t", header=FALSE, as.is=TRUE);
> names(obs.train) = c("src_id", "dst_id", "src_context",
  "dst_context", "ctx_id", "y");
```

It is important to note that the **column names** of an observation table have to be exactly `src_id`, `dst_id`, `src_context`, `dst_context`, `ctx_id` and `y`. The model fitting code does not recognize other names.

Source, Destination and Context Features: The features vectors of source nodes ($\boldsymbol{x}_i$), destination nodes ($\boldsymbol{x}_j$), edge contexts ($\boldsymbol{x}_k$) and training and test observations ($\boldsymbol{x}_{ijk}$) are in
type-`feature-user.txt`,
type-`feature-item.txt`,
type-`feature-ctxt.txt`,
where *type* = "dense" for the dense format and *type* = "sparse" for the sparse format.

For the dense format, take `dense-feature-user.txt` for example. The first column is `src_id` (the `src_id` column in the observation table refers to this column to get the feature vector of the source node for each observation). It is important to note that the **name of the first column** has to be exactly `src_id`. The rest of the columns specify the feature values and the column names can be arbitrary.

For the sparse format, take `sparse-feature-user.txt` for example. It has three columns:

1. `src_id`: Source node ID
2. `index`: Feature index (starting from 1, not 0)
3. `value`: Feature value

It is important to note that the **column names** have to be exactly `src_id`, `index` and `value`.

```
sparse-feature-user.txt       dense-feature-user.txt
SPARSE FORMAT            <=>  DENSE FORMAT
src_id  index   value         src_id   feature_1  feature_2  feature_3
    15      2  -0.978             15           0     -0.978      0.031
    15      3   0.031
```

Observation Features: The features vectors of training and test observations ($\boldsymbol{x}_{ijk}$) are in

type-`feature-obs-train.txt`,

type-`feature-obs-test.txt`,

where *type* = "dense" for the dense format and *type* = "sparse" for the sparse format.

For the dense format, take `dense-feature-obs-train.txt` for example. The nth line specifies the feature vector of observation on the nth line of `obs-train.txt`. Since there is a line-by-line correspondence, there is no need to have an ID column. Each column in this file represents a feature and the column names can be arbitrary.

For the sparse format, take `sparse-feature-obs-train.txt` for example. It has three columns:

1. `obs_id`: Line number in `obs-train.txt` (starting from 1, not 0)
2. `index`: Feature index (starting from 1, not 0)
3. `value`: Feature value

It is important to note that the **column names** have to be exactly `src_id`, `index` and `value`.

```
obs_id  index  value   # MEANING
     9      1   0.14   # 1st feature of line  9 in obs-train.txt = 0.14
     9      2  -0.93   # 2nd feature of line  9 in obs-train.txt =-0.93
    10      1  -0.91   # 1st feature of line 10 in obs-train.txt =-0.91
```

2.3 Model Fitting

See Example 1 in `src/R/examples/tutorial-BST.R` for the R script. For succinctness, we ignore some R commands in the following description.

Step 1: Read training and test observation tables (`obs.train` and `obs.test`), their corresponding observation feature tables (`x_obs.train` and `x_obs.test`), the source feature table (`x_src`), the destination feature table (`x_dst`) and the edge context feature table (`x_ctx`) from the corresponding files. Note that if you replace these tables with your data, you must not change the column names.

```
input.dir = "test-data/multicontext_model/simulated-mtx-uvw-10K"
obs.train = read.table(paste(input.dir,"/obs-train.txt",sep=""),
            sep="\t", header=FALSE, as.is=TRUE);
names(obs.train) = c("src_id", "dst_id", "src_context",
```

```
                     "dst_context", "ctx_id", "y");
x_obs.train = read.table(paste(input.dir,"/dense-feature-obs-train.txt",
              sep=""), sep="\t", header=FALSE, as.is=TRUE);
obs.test = read.table(paste(input.dir,"/obs-test.txt",sep=""),
           sep="\t", header=FALSE, as.is=TRUE);
names(obs.test) = c("src_id", "dst_id", "src_context",
                    "dst_context", "ctx_id", "y");
x_obs.test = read.table(paste(input.dir,"/dense-feature-obs-test.txt",
             sep=""), sep="\t", header=FALSE, as.is=TRUE);
x_src = read.table(paste(input.dir,"/dense-feature-user.txt",sep=""),
        sep="\t", header=FALSE, as.is=TRUE);
names(x_src)[1] = "src_id";
x_dst = read.table(paste(input.dir,"/dense-feature-item.txt",sep=""),
        sep="\t", header=FALSE, as.is=TRUE);
names(x_dst)[1] = "dst_id";
x_ctx = read.table(paste(input.dir,"/dense-feature-ctxt.txt",sep=""),
        sep="\t", header=FALSE, as.is=TRUE);
names(x_ctx)[1] = "ctx_id";
```

Step 2: Index the training and test data. Functions `indexData` and `indexTestData` (defined in `rc/R/model/multicontext_model_utils.R`) convert the input data tables into the right data structure. In particular, they replace the original IDs (`src_id`, `dst_id`, `src_context`, `dst_context` and `ctx_id`) by consecutive index numbers, and convert feature tables (data frames) into feature matrices.

```
data.train = indexData(
    obs=obs.train, src.dst.same=FALSE, rm.self.link=FALSE,
    x_obs=x_obs.train, x_src=x_src, x_dst=x_dst, x_ctx=x_ctx,
    add.intercept=TRUE
);
data.test = indexTestData(
    data.train=data.train, obs=obs.test,
    x_obs=x_obs.test, x_src=x_src, x_dst=x_dst, x_ctx=x_ctx
);
```

We then describe some input parameters to function `indexData`.

- `src.dst.same`: Whether source nodes and destination nodes refer to the same set of entities. For example, if source nodes represent users and destination nodes represents items, `src.dst.same` should be set to `FALSE`. However, if both source and destination nodes represent users (e.g., users rate other users) and `src_id` $= A$ refers to the same user A as `dst_id` $= A$, the `src.dst.same` should be set to `TRUE`.

- `rm.self.link`: Whether to remove self-edges. If `src.dst.same=TRUE`, you can choose to remove observations with `src_id` = `dst_id` by setting `rm.self.link=FALSE`. Otherwise, `rm.self.link` should be set to `FALSE`

- `add.intercept`: Whether you want to add an intercept to each feature matrix. If `add.intercept=TRUE`, a column of all 1s will be added to every feature matrix.

Because `data.train` is passed into `indexTestData`, the above parameters do not need to be passed into `indexTestData` and the parameter setting used to create the test data will be the same as the setting used to create the training data.

The output of `indexData` and `indexTestData` primarily consists of the following three components:

- `obs`: This is the observation table (data frame) with the new numeric index IDs. The columns are: `src.id`, `dst.id`, `src.context`, `dst.context`, `edge.context` and `y`, where `src.id` corresponds to `src_id`, etc., and `edge.context` corresponds to `ctx_id`.
- `IDs`: This list of vectors contains the mapping from new numeric index IDs to the original IDs.
- `feature:` This is a list of four feature matrices. `x_obs`, `x_src`, `x_dst` and `x_ctx` correspond to $\boldsymbol{x}_{ijk}$, $\boldsymbol{x}_i$, $\boldsymbol{x}_j$ and $\boldsymbol{x}_k$, respectively.

For example, assume the mth row of `data.train$obs` is

src.id	dst.id	src.context	dst.context	edge.context	y
i	j	p	q	k	y_{ijkpq}

Then, we have the following correspondence:

- `data.train$IDs$SrcIDs[`i`]` is the original source node ID of this observation. Similarly, `DstIDs[`j`]`, `SrcContexts[`p`]`, `DstContexts[`q`]` and `CtxIDs[`k`]` are the original IDs of the destination node, source context, destination context, edge context of this observation.
- `data.train$feature$x_obs[`m`,]` is the observation feature vector of this observation. Similarly, `x_src[`i`,]`, `x_dst[`j`,]` and `x_ctx[`k`,]` are the feature vectors of the source node, destination node and edge context of this observation.

Step 3: Fit the model(s). We first specify the settings of the models to be fitted.

```
setting = data.frame(
    name          = c("uvw1", "uvw2"),
    nFactors      = c(     1,      2),
    has.u         = c(  TRUE,   TRUE),
    has.gamma     = c( FALSE,  FALSE),
    nLocalFactors = c(     0,      0),
    is.logistic   = c( FALSE,  FALSE)
);
```

In the above example, we specify two models to be fitted.

- `name` specifies the name of the model, which should be unique.

- `nFactors` specifies the number of interaction factors per node; i.e., the number of dimensions of $\boldsymbol{v}_j$, which is the same as the numbers of dimensions of $\boldsymbol{u}_i$ and $\boldsymbol{w}_k$. If you want to disable or remove $\langle \boldsymbol{u}_i, \boldsymbol{v}_j, \boldsymbol{w}_k \rangle$ from the model specified in Eq 1, set `nFactors = 0`.

- `has.u` specifies whether to use $\langle \boldsymbol{u}_i, \boldsymbol{v}_j, \boldsymbol{w}_k \rangle$ in the model specified in Eq 1 or replace this term by $\langle \boldsymbol{v}_i, \boldsymbol{v}_j, \boldsymbol{w}_k \rangle$ (more examples will be given later). Notice that the latter does not have factor vector $\boldsymbol{u}_i$; thus, it corresponds to `has.u=FALSE`. It is important to note that if `has.u=FALSE`, you must set `src.dst.same=TRUE` when calling `indexData` in Step 2.

- `has.gamma` specifies whether to include γ_k in the model specified in Eq 1 or not. If `has.gamm=FALSE`, γ_k will be disabled or removed from the model.

- `nLocalFactors` should be set to 0 for most cases. Do not set it to other numbers unless you know what you are doing.

- `is.logistic` specifies whether to use the logistic response model or not. If `is.logistic=FALSE`, the Gaussian response model will be used.

In the following, we demonstrate a few different example settings and their corresponding models.

- The original BST model defined in [2]: Set `has.u=FALSE`, `has.gamma=FALSE`, and set all the context columns to be the same in the input data; i.e., before Step 2, set the input observation tables `obs.train` and `obs.test` so that the following holds.

  ```
  obs.train$src_context = obs.train$dst_context = obs.train$ctx_id
  obs.test$src_context  = obs.test$dst_context  = obs.test$ctx_id
  ```

  This setting gives the following model:

  $$y_{ijk} \sim \boldsymbol{x}'_{ijk}\boldsymbol{b} + \alpha_{ik} + \beta_{jk} + \langle \boldsymbol{v}_i, \boldsymbol{v}_j, \boldsymbol{w}_k \rangle$$

  Notice that since all the context columns are the same, there is no need for using a three dimensional context vector (k, p, q); instead, it is sufficient to just use k to index the context in the above equation. Also note that you must set `src.dst.same=TRUE` when calling `indexData` in Step 2.

- The RLFM model defined in [1]: Set `has.u=TRUE`, `has.gamma=FALSE`, and before Step 2, set:

  ```
  obs.train$src_context = obs.train$dst_context = obs.train$ctx_id = NULL;
  obs.test$src_context  = obs.test$dst_context  = obs.test$ctx_id  = NULL;
  x_ctx = NULL;
  ```

  This setting gives the following model:

  $$y_{ij} \sim \boldsymbol{x}'_{ij}\boldsymbol{b} + \alpha_i + \beta_j + \boldsymbol{u}'_i\boldsymbol{v}_j$$

  Notice that setting the context-related objects to `NULL` disables the context-specific factors in the model.

Step 4: Run the model fitting procedure.

```
out.dir = "/tmp/unit-test/simulated-mtx-uvw-10K";
ans = run.multicontext(
    obs=data.train$obs,         # training observation table
    feature=data.train$feature, # training feature matrices
    setting=setting, # setting specified in Step 3
    nSamples=200,    # number of Gibbs samples in each E-step
    nBurnIn=20,      # number of burn-in samples for the Gibbs sampler
    nIter=10,        # number of EM iterations
    test.obs=data.test$obs,         # test observation table (optional)
    test.feature=data.test$feature, # test feature matrices  (optional)
    reg.algo=NULL,    # regression algorithm; see below
    reg.control=NULL, # control parameters for the regression algorithm
    IDs=data.test$IDs,  # ID mappings (optional)
    out.level=1,        # see below
    out.dir=out.dir,    # output directory
    out.overwrite=TRUE, # whether to overwrite the output directory
    # initialization parameters (the default setting usually works)
    var_alpha=1, var_beta=1, var_gamma=1,
    var_v=1, var_u=1, var_w=1, var_y=NULL,
    relative.to.var_y=FALSE, var_alpha_global=1, var_beta_global=1,
    # others
    verbose=1,      # overall verbose level: larger -> more messages
    verbose.M=2,    # verbose level of the M-step
    rnd.seed.init=0, rnd.seed.fit=1 # random seeds
);
```

Most input parameters to `run.multicontext` are described in the above code piece. We make the following additional notes:

- `nSamples`, `nBurnIn` and `nIter` determine how long the procedure will run. In the above example, the procedure runs 10 EM iterations. In each iteration, it draws 220 Gibbs samples, where the first 20 samples are burn-in samples (which are thrown away) and the rest 200 samples are used to compute the Monte Carlo means in the E-step of this iteration. In our experience, 10-20 EM iterations with 100-200 samples per iteration are usually sufficient.

- `reg.algo` and `reg.control` specify how the regression priors will to be fitted. If they are set to `NULL`, R's basic linear regression function `lm` will be used to fit the prior regression coefficients $\boldsymbol{g}, \boldsymbol{d}, \boldsymbol{h}, G, D$ and H. Currently, we only support two other algorithms `GLMNet` and `RandomForest`. Notice that if `RandomForest` is used, the regression priors become nonlinear; see [3] for more information.

- `out.level` and `out.dir` specify what and where the fitting procedure will output. If `out.level` ¿ 0, each model specified in `setting` (i.e., each row in the `setting` table) will be output to a separate directory. The output directory name of the mth model is

```
paste(out.dir, "_", setting$name[m], sep="")
```

In this example, the output directories of the two models specified in the `setting` table are:

```
/tmp/unit-test/simulated-mtx-uvw-10K_uvw1
/tmp/unit-test/simulated-mtx-uvw-10K_uvw2
```

If `out.level=1`, the fitted models are stored in files `model.last` and `model.minTestLoss` in the output directories, where `model.last` contains the model obtained at the end of the last EM iteration and `model.minTestLoss` contains the model at the end of the EM iteration that gives the minimum loss on the test observation. `model.minTestLoss` exists only when `test.obs` is not `NULL`. If the fitting procedure stops (e.g., the machine reboots) before it finishes all the EM iteration, the latest fitted models will still be saved in these two files. If `out.level=2`, the model at the end of the mth EM iteration will be saved in `model.`m for each m. We describe how to read the output in Section 2.4.

2.4 Output

The two main output files in an output directory are `summary` and `model.last`.

Summary File: It records a number of statistics for each EM iteration. To read a summary file, use the following R command.

```
read.table(paste(out.dir,"_uvw2/summary",sep=""), header=TRUE);
```

Explanation of the columns are in the following:

- `Iter` specifies the iteration number.
- `nSteps` records the number of Gibbs samples drawn in the E-step of that iteration.
- `CDlogL`, `TestLoss`, `LossInTrain` and `TestRMSE` record the complete data log likelihood, loss on the test data, loss on the training data and RMSE (root mean squared error) on the test data for the model at the end of that iteration. For the Gaussian response model, the loss is defined as RMSE. For the logistic response model, the loss is defined as negative average log likelihood per observation.
- `TimeEStep`, `TimeMStep` and `TimeTest` record the numbers of seconds used to compute the E-step, M-step and predictions on test data in that iteration.

Sanity Check:

- Check `CDlogL` to see whether it increases sharply during the first few iterations and then oscillates at the end.
- Check `TestLoss` to see whether it converges. If not, more EM iterations are needed.
- Check `TestLoss` and `LossInTrain` to see whether the model overfits the data; i.e., TestLoss goes up, while LossInTrain goes down. If so, try to simplify the model by reducing the number of factors and parameters.

You can monitor the summary file when the code is running. When you see `TestLoss` converges, kill the running process.

Model File: The fitted models are saved in `model.last` and `model.minTestLoss`, which are R data binary files. To load the models, run the following command.

```
load(paste(out.dir,"_uvw2/model.last",sep=""));
```

After loading, the fitted prior parameters are in object `param` and the fitted latent factors are in object `factor`. Also, the object `IDs` contains the ID mappings described in Step 2 of Section 2.3.

2.5 Prediction

To make predictions, use the following function.

```
pred = predict.multicontext(
    model=list(factor=factor, param=param),
    obs=data.test$obs, feature=data.test$feature, is.logistic=FALSE
);
```

Now, `pred$pred.y` contains the predicted response for `data.test$obs`. Notice that the test data `data.test` was created by call `indexTestData` in Step 2 of Section 2.3.

2.6 Other Examples

In `src/R/examples/tutorial-BST.R`, we also provide a number of additional examples.

- Example 2: In this example, we demonstrate how to fit the same models as those in Example 1 with sparse features and the glmnet algorithm.
- Example 3: In this example, we demonstrate how to fit RLFM models with sparse features and the glmnet algorithm. Note that RLFM models do not fit this toy dataset well.

References

[1] D. Agarwal and B.-C. Chen. Regression-based latent factor models. In *KDD*, 2009.

[2] B.-C. Chen, J. Guo, B. Tseng, and J. Yang. User reputation in a comment rating environment. In *KDD*, 2011.

[3] L. Zhang, D. Agarwal, and B. Chen. Generalizing matrix factorization through flexible regression priors. In *RecSys*, 2011.