SRC Technical Note

1997 – 023

October 1, 1997

Each to Each Programmer's Reference Manual

Paul McJones and John DeTreville

Systems Research Center
130 Lytton Avenue
Palo Alto, CA 94301

http://www.research.digital.com/SRC/

Copyright © Digital Equipment Corporation 1997. All rights reserved

Each to Each Programmer's Reference Manual

Paul McJones and John DeTreville

Contents

Introduction

This document describes how to incorporate the Each to Each recommendation technology in a complete application. Each to Each applies collaborative filtering techniques to the problem of making subjective recommendations to consumers faced with "infoglut". The basic idea is to ask people to vote for items on a numeric scale, then perform a statistical analysis of the collection of all people's votes, and use the results of the analysis to predict additional items of potential interest to a particular person. Unlike some competitive approaches, the Each to Each technology separates prediction from analysis, allows predictions to be made using compact "models" produced by the analysis, and provides meaningful predictions after a person has provided just a few votes.

The general goal of the APIs presented in this document is to separate core recommendation functionality from application-dependent features such as choice of platform, database, communication mechanism, etc. Implementing the "glue" code to connect these APIs into a complete application is straightforward.

The APIs are written in C++, but avoid advanced C++ features (e.g., no multiple inheritance, templates, or exceptions). Bindings for other languages (e.g., Java, C) would not be difficult to design.

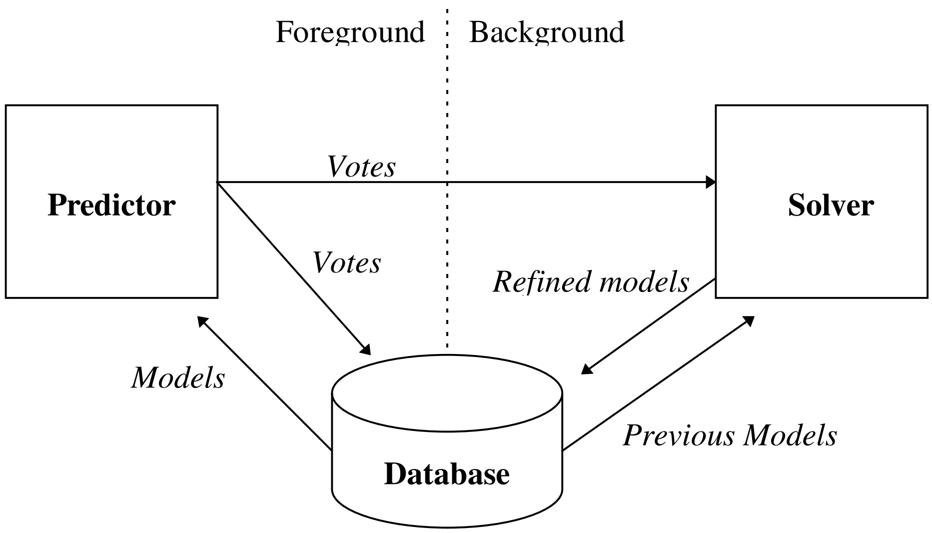
This document describes everything needed for pure vote-based predictions of person-item and person-person affinity (the latter to be used for choosing reviews of potential interest to a person). The appendix describes an enhancement called categories, which allow the use of demographic information and item categorization information in predictions. We have little experience with categories.

The technology is available for licensing; see **Appendix B: Technology Availability**.

Overview of an Each to Each application

The Each to Each technology separates prediction from analysis. Prediction involves interacting with a person to record his votes or ratings of specific items, computing predicted ratings, and providing them back to the person. Analysis involves applying a statistical algorithm to the collection of votes gathered from all people, producing a compact set of *models* used to drive future predictions.

The prediction component, or *predictor* for short, is inherently interactive. The analysis component, or *solver* for short, does not need to have low-latency communication with the prediction function or the user interface. In a typical internet application, the predictor runs as an application gateway, accessed via an interface such as CGI, NSAPI, or ISAPI, while the solver runs as a stand-alone process, perhaps on a different server computer. In a retail kiosk application, the predictor runs in the kiosk computer while the solver runs at a central data center; periodic dial-up communication is used to send votes to the solver and updated models to the kiosks. Finally, in a CD-ROM application, the predictor runs on a personal computer, using pre-computed models stored on the CD-ROM. Votes are stored on the local hard drive, and optionally sent to a central data center via dial-up communication and/or the Internet. Here is a diagram showing the overall data flow:



Terminology

We have already introduced the terms predictor and solver. We should point out that we use the term "predictor" to refer to two different things: the `ee_predict` API in the Each to Each SDK and the component of an application that records new votes and generates predictions. Similarly, we use the term "solver" to refer both to the `ee_solve` API and the component of an application that calls this API.

There are a few other terms with special meanings in this document:

Person: someone who votes for items and asks for recommendations.

Item: something that can be voted on.

Vote: actual or predicted assessment of an item by a person, consisting of a score and a weight.

Score: a numeric value between 0.0 and 1.0, where higher numbers mean more positive assessments.

Weight: a nonnegative floating-point value. On input to Each to Each, weights have a linear interpretation (e.g., .5 means half as confident as 1.0), but the particular scale (e.g., 0.0 to 1.0 or 0.0 to 10.0) doesn't matter. On output from Each to Each, weights are approximate and ordered but not linear and will be between 0.0 and 1.0. Determining appropriate thresholds upon which to base recommendations typically requires a bit of application-dependent tuning.

Model: a block of data (currently 128 bytes in length) computed by the solver for each person and for each item. The predictor functions use the model in fast algorithms for predicting votes and correlations.

APIs

There are two APIs, one for the predictor and one for the solver, each defined in a separate header file. As mentioned earlier, these APIs perform the core prediction and analysis algorithms, but do not provide application-specific functionality such as nonvolatile storage or communication. The next section of this document, **Sample Application**, shows how to integrate the APIs with a complete application.

Both APIs are "thread-safe", that is the implementations do all necessary locking so that concurrent calls by multiple threads function correctly.

Note: We assume that the target platform's mutual exclusion primitives are fast relative to the time for a call on the prediction procedures.

eetypes.h

This header file gives the types used in the ***Schema***, ***eepredict.h***, and ***eesolve.h*** sections. We say that the types used to identify persons and items, `ee_person` and `ee_item` respectively, are *opaque*, meaning that the Each to Each code doesn't assume anything about them, such as whether or not there are "gaps" in the range of values in use. The one exception is that the values `ee_person_null` and `ee_item_null` should never be assigned to a real person or item; they are reserved for use as a special indicator value, such as indicating when an "iterator" has no more values to return (see `ee_solve::person_iterator` and `ee_solve::item_iterator` in ***eesolve.h***).

```
typedef int ee_bool;

class ee_person {                           // an opaque uid for a person
public:
    int uid;
    ee_person(int uu = 0): uid(uu) {};
};

inline int operator==(const ee_person& x, const ee_person& xx)
{ return x.uid == xx.uid; }

const ee_person ee_person_null(0);          // unused "null" value

class ee_item {                             // an opaque uid for an item
public:
    int uid;
    ee_item(int uu = 0): uid(uu) {};
};

inline int operator==(const ee_item& y, const ee_item& yy)
{ return y.uid == yy.uid; }

const ee_item ee_item_null(0);              // unused "null" value

class ee_vote {                             // a vote
public:
    float s;                                  // score
    float w;                                  // weight
    ee_vote(float ss = 0.0f, float ww = 0.0f): s(ss), w(ww) {};
};

const ee_vote ee_vote_null(0.0f, 0.0f);     // unused "null" value

const ee_model_ints = 32;

class ee_model {                            // a model
public:
    int a[ee_model_ints];
};

const ee_model ee_model_null = { { 0 } };  // initial "null" value
```

Schema

The Each to Each solver and predictor require nonvolatile storage of some information about each person, each item, and each vote on an item by a person. For reasons of portability and flexibility, this storage is not managed directly by the Each to Each software. Instead, it is managed by application-specific code and is made accessible to the Each to Each software as part of the APIs defined in the next section of this document.

Here we define the logical schema as if this information is stored in a relational database with five tables, `Persons`, `Items`, `Votes`, `VotesLog`, and `Generations`. Of course, the application programmer is free to merge these tables with existing tables, or to implement them in a different storage medium such as a file system, subject to performance considerations.

The `Persons` table contains records of the form:

```
ee_person person;    // unique key
ee_model model;
```

The `Items` table contains records of the form:

```
ee_item item;        // unique key
ee_model model;
```

The `Votes` table contains records of the form:

```
ee_person person;    // unique key: person, item
ee_item item;
ee_vote vote;
```

The `VotesLog` table contains records of the form:

```
ee_person person;    // unique key: person, item, time
ee_item item;
ee_vote vote;
timestamp time;      // (e.g., DATE) indexed by time
```

The `VotesLog` table helps the solver process detect updates to the `Votes` table made by the predictor process as it records new votes. Every transaction that modifies (inserts, updates, or deletes a record in) `Votes` must also insert a corresponding record in `VotesLog`. A deletion to `Votes` is indicated by a record in `VotesLog` with `v` equal to `ee_vote_null`. The `time` field determines the order in which the solver processes the entries, and so must have enough precision to resolve successive updates to a vote, e.g., when a person decides to change or delete a vote.

The `Generations` table contains a single record of the form:

```
int id;              // always equal to one
int generation;      // initially equal to zero
```

The `Generations` table helps the predictor process notice updates to the `Items` table made by the solver process. Each time the solver has completed a set of updates to the `Items` table, it must update the record in `the Generations` table by increasing the generation field by one.

Consistency

The Each to Each APIs were designed in such a way that very few consistency constraints on the data are required:

1. The models in the `Persons` and `Items` tables don't have to be updated atomically; they may be from different runs of the solver.
2. If there is a vote for a person or item without a model, a model will be generated internally and will be available from `get_*_model`, etc. If there is a person or item without any votes, its model will be very near the null model.
3. The `VotesLog` table, if applied to the `Votes` table (doing the insertions, modifications, deletions), gives the "truth" from which the system works. It is therefore okay for the `VotesLog` table to contain votes that have already been applied to the `Votes` table.

eepredict.h

This header file defines a single class `ee_predict`, which provides functions to generate person-item predictions and also person-person correlations. The `ee_predict` class doesn't have direct access to the nonvolatile storage in which votes and models are stored. Instead, when it needs a particular piece of information, it calls one of two *virtual functions*, `get_person_votes` or `get_item_model`. These functions are implemented in a class derived from `ee_predict` that is implemented by the client. See the **Sample Application** section for details.

```
#include "eetypes.h"

class ee_predict_impl;                          // private implementation class

class ee_predict {
public:
    ee_predict();                               // constructor
    /* Multiple distributed predictors can increase performance and
       availability; changes to the person/item/votes model database
       should be propagated via reset_vote() to each instance. */

    virtual ~ee_predict();                      // destructor

    ee_vote predict_item(const ee_person x, const ee_item y);
    /* Return person x's predicted vote (score, weight) for item y. */

    ee_vote predict_person(const ee_person x, const ee_person xx);
    /* Return person x's predicted correlation (score, weight) with
       person xx.  predict_person is not necessarily symmetric with
       respect to x and xx; it expresses the interest of person x in
       person xx. */

    void reset_vote(const ee_person x, const ee_item y, const ee_vote v);
    /* Note v as person x's new or revised vote for item y, or delete
       if v is null. To delete a whole person or item, delete all the
       votes for that person or item.  This call should follow a
       change in the nonvolatile votes table; it resets any cache the
       predictor may have kept. */

    void reset_all_models(void);
    /* Reset the internal state to include no knowledge of any item's
       model. This should be called each time new models are available
       from the solver.  This call should follow any mass changes to
       the nonvolatile model data and resets any cache the predictor
       might have kept. */

    void reset_all(void);
    /* An extension of reset_all_models that also clears any
       cache of votes. */

/* Pure virtual functions, to be implemented by the client in a class
   derived from ee_predict: */

    virtual void get_person_votes(
        const ee_person x,
        void (*setvote)(
            const ee_person x, const ee_item y, const ee_vote& v, void* a),
        void* arg) = 0;
    /* Retrieve all <person, item, vote> records with person==x from
       nonvolatile storage, calling (*setvote)(person, item, vote, arg)
       for each. */

    virtual void get_item_model(const ee_item y, ee_model& m) = 0;
    /* Retrieve item y's model from nonvolatile storage, and return it
       in m; if there is no model, return the null model. */

private:
    // Disable the copy constructor and assignment operator:
```

```
    ee_predict(const ee_predict& rhs);
    ee_predict& operator=(const ee_predict& rhs);

    ee_predict_impl *impl;                  // instance data
};
```

eesolve.h

This header file defines a single class `ee_solve`, which provides functions to analyze a set of votes, producing a compact set of models that can be used by `ee_predict` to generate predictions. Like `ee_predict`, the `ee_solve` class doesn't have direct access to the nonvolatile storage in which votes and models are used. Rather than using virtual functions, `ee_solve` provides ordinary functions to supply it with votes and (optionally) models, and to retrieve refined models.

`ee_solve` uses an iterative refinement algorithm. The longer this algorithm runs, the more accurate the predictions that are generated from the models it produces. The progress of the algorithm is reflected in a quantity called the *residue*, which measures the remaining opportunity for further refinement. The residue is proportional to the count of votes, decreases with time, and approaches an asymptote greater than zero. Functions are provided to retrieve the current residue and the count of votes.

It's possible to do a "cold start" of `ee_solve`, giving it only a set of votes. However, it will more quickly reach low residue values if it is given models from a previous run as well as the set of votes. As shown in the **Sample Application** section, it can be run continously by supplying it with a log of changes to the `Votes` database.

```
#include "eetypes.h"

// Private implementation classes:
class ee_solve_impl;
class ee_solve_person_iterator_impl;
class ee_solve_item_iterator_impl;

class ee_solve {              // the solver, normally with exactly one instance

public:
    ee_solve();               // constructor
    ~ee_solve();              // destructor

    void restart(void);
    /* Reset the internal state to that following a constructor. */

    void set_vote(const ee_person x, const ee_item y, const ee_vote v);
    /* Set or update person x's vote for item y to be v, or delete if
       v is null. */

    void set_person_model(const ee_person x, const ee_model& m);
    /* Set the model for person x to be m. Before initialization, a
       model is treated as null. */

    void set_item_model(const ee_item y, const ee_model& m);
    /* Set the model for item y to be m. Before initialization, a
       model is treated as null. */

    void solve_step();
    /* Update models (a potentially lengthy operation!).  solve_step
       is meant to be called repeatedly until approximate convergence
       is reached; see get_residue. */

    double get_residue();
    /* Return the current value of the "residue", a non-negative
       value.  Calling solve_step decreases in the residue; setting
       votes or models can increase it. The residue grows with the
       number of votes, and the client can use the residue per vote to
       determine approximate convergence. */

    int get_vote_count();
```

```
    /* Return the current number of votes; overridden votes are not
       counted. */

    void get_person_model(const ee_person x, ee_model& m);
    /* Set m to person x's model. */

    void get_item_model(const ee_item y, ee_model& m);
    /* Set m to item y's model. */

    class person_iterator {
    public:
        person_iterator(ee_solve& s); // constructor
        ~person_iterator();           // destructor
        ee_person get_next();         // iterate over persons
    private:
        // Disable the copy constructor and assignment operator:
        person_iterator(const person_iterator& rhs);
        person_iterator& operator=(const person_iterator& rhs);

        ee_solve_person_iterator_impl *impl; // instance data
    };
    /* A person_iterator is used to discover all the persons known to
       the solver.  A person is known to the solver if one or more
       votes by that person have been registered via set_vote and not
       subsequently deleted, or if a model for that person has been
       registered via set_person_model and not subsequently deleted.

       Each call to get_next returns another person not previously
       returned since the iterator was constructed, or ee_person_null
       if none remain.  The result of interleaving calls to get_next
       with calls to set_vote or set_person_model is undefined. */

    class item_iterator {
    public:
        item_iterator(ee_solve& s);   // constructor
        ~item_iterator();             // destructor
        ee_item get_next();           // iterate over items
    private:
        // Disable the copy constructor and assignment operator:
        item_iterator(const item_iterator& rhs);
        item_iterator& operator=(const item_iterator& rhs);

        ee_solve_item_iterator_impl *impl;  // instance data
    };
    /* An item_iterator is used to discover all the items known to the
       solver; see person_iterator for details. */

    friend class person_iterator;
    friend class item_iterator;

private:
    // Disable the copy constructor and assignment operator:
    ee_solve(const ee_solve& rhs);
    ee_solve& operator=(const ee_solve& rhs);

    ee_solve_impl *impl;                    // instance data
};
```

Sample Application

In this section we sketch out how to use the APIs in an application consisting of a predictor process and a solver process. The predictor process gathers and records votes and makes predictions. The solver process reads votes and computes successively more accurate models, periodically writing the models back to the database so they can be used by the predictor.

We start with the solver process, whose organization changes little from application to application. We then show a typical predictor process; here, the specifics will vary more between applications. Code that needs to be adapted to a particular environment (database, OS, etc.) is typeset `<like this>`.

Solver process

```
#include "eetypes.h"
#include "eesolve.h"
```

We create a single instance of the solver class:

```
ee_solve ee;
```

A real solver process might take command-line parameters, or use a "GUI" to accept control parameters. Here we're keeping it simple:

```
main() {
```

When the process first starts, we read in the entire `Votes`, `Persons`, and `Items` tables.

```
    <SELECT * FROM Votes>
    while (<records remain>) {
        ee_person p;
        ee_item i;
        ee_vote v;
        <get p, i, v from current record>
        ee.set_vote(p, i, v);
        <move to next record>
    }
    <SELECT * FROM Persons>
    while (<records remain>) {
        ee_person p;
        ee_model m;
        <get p, m from current record>
        ee.set_person_model(p, m);
        <move to next record>
    }
    <SELECT * FROM Items>
    while (<records remain>) {
        ee_item i;
        ee_model m;
        <get i, m from current record>
        ee.set_item_model(i, m);
        <move to next record>
    }
```

Everything from here to the end of the program is a loop that is repeated until shutdown:

```
    while (1) {
```

First, process the `VotesLog` table to find out about recent updates to the `Votes` table:

```
        <SELECT * FROM VotesLog ORDER BY time>
        while (<records remain>) {
            ee_person p;
            ee_item i;
            ee_vote v;
            <get p, i, v from current record>
            ee.set_vote(p, i, v); /* deletes if v is ee_vote_null */
            <delete current record>
            <move to next record>
        }
```

Now run the solver algorithm. It is based on iterative refinement, so running it longer gives increased accuracy. In determining the number of steps to run it, we need to balance response time, accuracy, and the cost of writing the models back to the database.

```
#define RESIDUE_THRESHOLD <tune for application>

        do ee.solve_step();
        while (RESIDUE_THRESHOLD <= ee.get_residue() / ee.get_vote_count()
               && <time limit not exceeded>);
```

After running the solver algorithm, write back the updated models to the `Persons` and `Items` table. This checkpoints the state across shutdown/restart, and makes the new item models available to the predictor process.

```
        <SELECT * FROM Persons>
        while (<records remain>) {
            ee_person p;
            ee_model m;
            <get p from current record>
            ee.get_person_model(p, m);
            <update p's model to m in current record>
            <move to next record>
        }
        <SELECT * FROM Items>
        while (<records remain>) {
            ee_item i;
            ee_model m;
            <get i from record>
            ee.get_item_model(i, m);
            <update i's model to m in current record>
            <move to next record>
        }
```

The code above depends on the `Persons` and `Items` tables having been initialized with an entry for each person and item, respectively. To avoid this dependency, we could have taken this alternate approach using *iterators*:

```
        ee_solve::person_iterator persons;
        for ( ; ; ) {
            ee_person p = persons.get_next();
            if (p==ee_person_null) break;
            ee_model m;
            ee.get_person_model(p, m);
            <update p's model to m in current record>
        }
        ee_solve::item_iterator items;
        for ( ; ; ) {
            ee_item i = items.get_next();
            if (i==ee_item_null) break;
            ee_model m;
            ee.get_item_model(i, m);
            <update i's model to m in current record>
        }
```

Note that this approach using iterators performs the database updates in an arbitrary order, which may not perform as well as performing the updates in index order.

The final step is to signal the predictor process that it should reset its cache, causing it to reload the updated item models from the database:

```
        <SELECT * FROM Generations WHERE id==1>
        <update generation = generation+1 in current record>
    }
}
```

Predictor process

From the point of view of the Each to Each APIs, there are three parts of the predictor process:

1. Providing database access to Each to Each functions.
2. Gathering votes and recording them in the database.
3. Making predictions.

Here we package all three parts in a single C++ program file.

Our first task is to provide the code that connects the Each to Each prediction functions with the votes and item models stored in the application-provided database. We do this by writing a new class, `my_ee_predict`, that is derived from `ee_predict`. In the new class, we will implement the virtual functions of the `ee_predict` class, which in C++ terminology is an "abstract" class. This means that it contains one or more "pure virtual functions" which aren't defined by the class itself, but which are called by other member functions of the class. We will also implement two additional functions: one to record a vote, and one to check for new data from the solver process. (In a real application, you might want to put `my_ee_predict` in a separate file.)

```
#include "eetypes.h"
#include "eepredict.h"

class my_ee_predict : public ee_predict {
public:
    virtual void get_person_votes(
        const ee_person x,
        void (*setvote)(
            const ee_person x,
            const ee_item y,
            const ee_vote& v,
            void* arg),
        void* arg) = 0;
    virtual void get_item_model(const ee_item y, ee_model& m);
    void record_vote(ee_person x, ee_item y, ee_vote v);
    /* Update stable storage and the cache to indicate that v is person x's
       new or revised vote for item y, or delete person x's vote for item y if v
       is null. */
    void test_generation();
    /* Test whether the solver process has recently written updated item models
       into stable storage, and if so flush all item models from the cache. */
}
```

First we provide the implementations of the two virtual functions:

```
void my_ee_predict::get_person_votes(
    const ee_person x,
    void (*setvote)(
        const ee_person x,
        const ee_item y,
        const ee_vote& v,
        void* arg),
    void* arg)
{
    ee_item item;
    ee_vote vote;
    <SELECT * FROM Votes WHERE person==x>
    while (<records remain>) {
        <get item, vote from record>
       (*setvote)(x, item, vote, arg);
    }
}

void my_ee_predict::get_item_model(const ee_item y, ee_model& m) {
    ee_model model;
    <SELECT * FROM Items WHERE item==y>
    <get model from record>
    m = model;
}
```

Now we implement the `record_vote` function, which needs to update the `Votes` and `VotesLog` tables and also flush any previous vote that may have been cached. This function can be used for the case when a person wants to erase his vote, by passing `ee_vote_null` for `v`:

```
void my_ee_predict::record_vote(ee_person x, ee_item y, ee_vote v) {
    timestamp t = now();
    <begin transaction>
    reset_vote(x, y, v);
    <SELECT * FROM Votes WHERE person==x AND item==y>
    if (<record exists>)
        if (v==ee_vote_null)
            <delete current record from database>
        else
            <update vote=v in current record>
    else
        if (v==ee_vote_null)
            ; // nothing to do
        else
            <insert new record in Votes Table with x, y, v>
    <insert new record in VotesLog table with x, y, v, t>
    <commit transaction>
}
```

Since these virtual functions access the database, calling them is relatively expensive. Thus the `ee_predict` implementation caches information in main memory to speed up subsequent predictions for the same persons and/or items. In the mean time, the solver process is computing more accurate models (based on recent votes), which it periodically writes back to the database. At that point, we'd like the predictor process to flush the cached information and once again call the virtual functions to load the latest information. The way we arrange to do this is to periodically (say, once an hour) poll the `Generations` table in the database. So the predictor process needs to call this function every so often, say by forking a thread that loops forever, sleeping for a half hour and calling this:

```
void ee_predict::test_generation() {
    static int last_generation = 0;
    int generation;
    <SELECT * FROM Generations WHERE id==1>
    <get generation from current record>
    if (generation < last_generation) {
        reset_all_models();
        last_generation = generation;
    }
}
```

That ends the implementation of `my_ee_predict`. Now we create a single instance of it to use throughout the predictor process:

```
my_ee_predict ee;
```

We're ready to do predictions. Here's an example of how to recommend up to five new Chinese restaurants in Paris:

```
#define WEIGHT_THRESHOLD <tune for application>
#define SCORE_THRESHOLD <tune for application>

ee_person p;      // input: person desiring prediction
<query database for Chinese restaurants in Paris that p has
    not already rated>
while (<records remain>) {
    ee_item i;
    <get i from current record>
    ee_vote v = ee.predict_item(p, i);
    if (v.w < WEIGHT_THRESHOLD) continue;
```

```
    if (v.s < SCORE_THRESHOLD) continue;
    <add [i, v] to list of possibilities>
    <move to next record>
}
<sort list of possibilities in descending order of v>
return <first five restaurants from list>;
```

Here's an example of how to present up to three reviews of a hotel. You might consider treating user-written reviews about items as additional "first-class items". However, we recommend using person-person correlation (`ee_predict::predict_person`) to rank order a set of reviews in terms of the similarity of interests between the current user and the authors of the reviews.

```
ee_person p;       // input: current user
ee_item i;         // input: subject of review
<query database for reviews of hotel i>
while (<records remain>) {
    ee_person q;  // author
    <get q, review from current record>
    if (q == p) continue;
    ee_vote v = ee.predict_person(p, q);
    if (v.w < WEIGHT_THRESHOLD) continue;
    if (v.s < SCORE_THRESHOLD) continue;
   <add [review, v] to list of possibilities>
   <move to next record>
}
<sort list of possibilities in descending order of v>
return <first three reviews from list>;
```

Appendix A: Categories

The APIs described so far provide recommendations based solely on the (subjective) votes of items recorded by persons. This appendix describes additions to that functionality, extending the recommendation mechanism to use demographic information known about the persons and/or descriptive information known about the items.

The approach we've taken is to define "categories", to allow persons to be associated with categories of demographic information, and to allow items to be associated with categories of descriptive information. Here are examples of possible demographic categories:

```
age_0_20
age_21_35
age_36_54
age_55_up
job_doctor
job_lawyer
job_teacher
zip_9xxxx
```

And here are examples of item description categories for a travel information application:

```
restaurant_French
restaurant_Chinese
music_blues
music_classical
recreation_bicycling
```

Note: We do not allow a person to explicitly rate a whole category, e.g., it is not possible to say "I like Chinese restaurants in general." Instead, such information is inferred. We also do not allow the specification of structural relationships between categories, such as "Chinese restaurants are a kind of Asian restaurant".

Here are the API additions to eetypes.h:

```
    class ee_category {                   // an opaque uid for a category
public:
```

```
    int uid;
    ee_category(int uu = 0): uid(uu) {};
};

inline int operator==(const ee_category& x, const ee_category& xx)
{ return x.uid == xx.uid; }

const ee_category ee_category_null(0); // unused "null" value
```

For each category, there is a model (just as with persons and items). The schema for nonvolatile storage needs to be extended with a `Categories` table to hold the category models:

```
ee_category category;               // unique key
ee_model model;
```

There must also be a mechanism (tables or procedures) to map information about persons and items to categories.

Here are the additions to eepredict.h:
  (a) Add two member functions:

```
void reset_person_category(
    const ee_person x, const ee_category c, const float w);
/* Assert c as a category for person x with weight w; remove any
   such assertion if w==0. This call should follow any change in
   the nonvolatile database; it resets any cache the predictor may
   have kept. */

void reset_item_category(
    const ee_item y, const ee_category c, const float w);
/* Assert c as a category for item y with weight w; remove any such
   assertion if w is 0. This call should follow any change in the
   nonvolatile database; it resets any cache the predictor may have
   kept. */
```

  (b) Add two pure virtual functions (to be implemented by the application programmer):

```
virtual void get_person_categories(
    const ee_person x,
    void (*setcategory)(
        const ee_person x, const ee_category c, const float w, void* a),
    void* arg) = 0;
/* Retrieve all categories (and weights) for person x from nonvolatile
   storage, calling (*setcategory)(x, category, weight, arg) for each. */

virtual void get_person_category_model(
    const ee_category c,
    ee_model& m) = 0;
/* Retrieve category c's model from nonvolatile storage, and return
   it in m; if there is no model, return the null model. */
```

Finally, here are the additions to eesolve.h:
  (a) Add four member functions to class `ee_solve`:

```
void set_category_model(const ee_category c, const ee_model& m);
/* Set the model for category c to be m. Before initialization, a
   model is treated as null. */

void set_person_in_category(
    const ee_person x, const ee_category c, const float w);
/* Assert that x is a member of c with weight w; a zero weight
   clears an assertion. */

void set_item_in_category(
    const ee_item y, const ee_category c, const float w);
```

```
/* Assert that y is a member of c with weight w; a zero weight
   clears an assertion. */

void get_category_model(const ee_category c, ee_model& m);
/* Set m to category c's model. */
```

(b) Add a new class:

```
class category_iterator {
public:
    category_iterator(ee_solve& s); // constructor
    ~category_iterator();           // destructor
    ee_category get_next();         // iterate over categories
private:
    // Disable the copy constructor and assignment operator:
    category_iterator(const category_iterator& rhs);
    category_iterator& operator=(const category_iterator& rhs);
    ee_solve_category_iterator_impl *impl; // instance data
};
friend class category_iterator;
/* A category_iterator is used to discover all the categories known
   to the solver.  A category is known to the solver if one or more
   persons or items have been registered in that category via
   set_person_in_category or set_item_in_category and not deleted
   via restart, or if a model for that category has been registered
   via set_category_model and not deleted via restart.

   Each call to get_next returns another category not previously
   returned since the iterator was constructed, or ee_category_null
   if none remain.  The result of interleaving calls to get_next
   with calls to set_*_in_category or set_category_model is undefined. */
```

Appendix B: Technology Availability

The Each to Each recommendation technology was developed in a research project at the Systems Research Center of Digital Equipment Corporation. The technology was extensively tested in EachMovie, a free web-based movie recommendation service. EachMovie was operated for eighteen months. During this time, some 72,916 users entered a total of 2,811,983 numeric ratings for 1,628 different movies (films and videos). Based on unsolicited comments from users and the number of repeat users, EachMovie was successful.

Digital Equipment Corporation is offering the technology for licensing. If you are interested, contact:

Ted Faigle
Licensing Executive
Digital Equipment Corporation
MSO2-3/H25
111 Powdermill Road
Maynard, MA, 01754-1418

telephone: 1 508 493-8348
email: faigle@mail.dec.com
http://www.digital.com/info/corporate-licensing/