Recommender systems, Part 2: Introducing open source engines

Explore software for building a recommendation capability

M. Tim Jones

December 12, 2013

Part 1 of this series introduces the basic approaches and algorithms for the construction of recommendation engines. This concluding installment explores some open source solutions for building recommendation systems and demonstrates the use of two of them. The author also shows how to develop a simple clustering application in Ruby and apply it to sample data.

View more content in this series

Part 1 of this series explores some of the core approaches and specific algorithms that recommendation systems use. This installment starts by describing the context for a recommendation engine. Then it shows you some open source solutions for building recommendation systems. You'll see how to develop a simple clustering application in Ruby and apply it to sample data from Part 1. Then you'll experiment with two open source engines that implement similar algorithms but use markedly different approaches. Finally, you'll get an overview of other recommender, machine-learning, and data-mining solutions available in the open source domain.

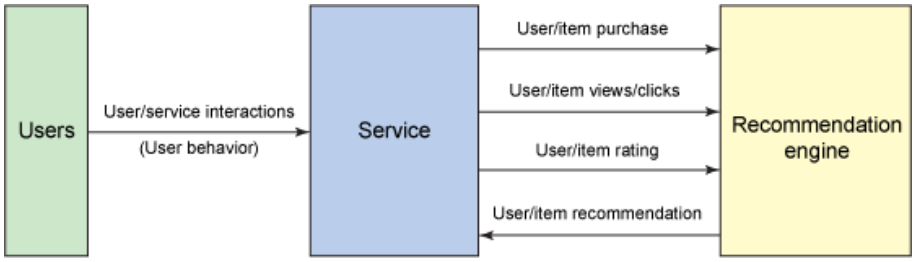
Recommendation engines operate on available data related to the behavior of a set of users for a service. In most cases, the service is a website that exposes individual items to a population of users and tracks users' behavior relative to those items (such as viewing, purchasing, and rating them). This kind of data forms the fundamental requirements of a recommendation engine that uses user-based or item-based collaborative filtering.

Figure 1 illustrates a simple ecosystem of a recommendation system. The ecosystem consists of:

- A small population of users

- A service
- The ability to differentiate the behavior of an individual user within the service (purchases, views, and so on)
- A recommendation engine that computes and store artifacts of recommendations

Figure 1. Simple ecosystem of a recommendation system



Next, I'll introduce a simple clustering implementation that uses Ruby. The implementation groups like users based on their individual behaviors.

Clustering for user-based collaborative filtering

Recall that Part 1 explores user-based collaborative filtering in a simple population of four users. To make recommendations in a population, you must first classify those users based on their behaviors. This first example explores classification of users by means of a simple clustering algorithm, *k*-means.

As you know from Part 1, *k*-means partitions items into *k* clusters, randomly at first. Then a *centroid* is calculated for each cluster as a function of its members. The distance for each item is then checked against the clusters' centroids. If an item is found to be closer to another cluster, it's moved to that cluster. Centroids are recalculated each time all items are checked. When no items move during an iteration, the algorithm ends.

Figure 2 shows the sample data set for four users who read blogs.

Figure 2. Sample data for users' blog-reading behavior

Blogs	Marc	Megan	Elise	Jill
Linux	13	3	11	-
OpenSource	10	-	-	3
Cloud Computing	6	1	9	-
Java Technology	-	6	-	9
Agile	-	7	1	8
	Articles read per user			
Cluster	1	2	1	2

Listing 1 shows the Ruby implementation of a `Cluster` class, which represents a cluster for the *k*-means algorithm.

Listing 1. `Cluster` class written in Ruby

```
class Cluster

  # Initialize the class instance
  def initialize
```

```
    @people = Array.new
    @vcentroid = Hash.new( "0" )
  end

  # Add a feature vector to the cluster
  def add( person )
    @people.push( person )
  end

  # Remove a feature vector from the cluster
  def remove( person )
    @people.delete( person )
  end

  # Return the cluster centroid
  def get_people
    return @people
  end

  # Calculate the centroid vector from the cluster members
  def calculate_centroid

    # Clear the centroid vector
    @vcentroid.clear
    tcentroid = Hash.new( "0" )

    # Iterate through each feature vector
    @people.each do |person|

      # Sum the feature vectors in this cluster
      person.each do |key,value|
        tcentroid[key] = tcentroid.delete(key).to_i + value.to_i
      end

    end

    # Compute the average for the centroid
    tcentroid.each do |key,value|
      @vcentroid[key] = value.to_f / @people.length
    end

  end

  # Calculate the geometric distance
  def calculate_gd( person )

    gd = 0.0

    person.each do |key,value|
      gd += (@vcentroid[key].to_f-value.to_f) * (@vcentroid[key].to_f-value.to_f)
    end

    gd = Math.sqrt(gd)

    return gd

  end

end
```

The `Cluster` class contains the members of the cluster as an array and the centroid for the cluster as a hash (with the blog name as the key and the value as the articles viewed). The `initialize` method creates both the array and the hash.

Three methods manage the members of the cluster:

- The `add` method adds a `person` to the cluster.
- The `remove` method deletes a `person`.
- The `get_people` method extracts the array for the purpose of iterating through it.

Two methods manage the cluster. The `calculate_centroid` method updates the mean of the cluster based on its current members. The `calculate_gd` method returns the Euclidean distance between the cluster centroid and a passed `person` object.

Listing 2 shows a Ruby implementation of *k*-means that operates on the sample data in Figure 2.

Listing 2. Implementing the basic *k*-means algorithm

```
def kmeans

  # Sample user hashes
  marc = { 'linux' => '13', 'oss' => '10', 'cloud' => '6',
           'java' => '0', 'agile' => '0' }
  megan = { 'linux' => '3', 'oss' => '0', 'cloud' => '1',
            'java' => '6', 'agile' => '7' }
  elise = { 'linux' => '11', 'oss' => '0', 'cloud' => '9',
            'java' => '0', 'agile' => '1' }
  jill = { 'linux' => '0', 'oss' => '3', 'cloud' => '0',
           'java' => '9', 'agile' => '8' }

  # Define our two clusters and initialize them with two users
  cluster1 = Cluster.new
  cluster1.add(marc)
  cluster1.add(megan)

  cluster2 = Cluster.new
  cluster2.add(elise)
  cluster2.add(jill)

  changed = true

  # Repeat until no further membership changes occur
  while changed do

    changed = false

    # Recalculate each cluster's centroid (mean)
    cluster1.calculate_centroid
    cluster2.calculate_centroid

    # Get the membership of each cluster
    people1 = cluster1.get_people
    people2 = cluster2.get_people

    # Check members of cluster 1 against the cluster centroids
    puts "Checking cluster 1"
    people1.each do |person|
      if cluster2.calculate_gd(person) < cluster1.calculate_gd(person) then
        puts "Feature vector moved from cluster 1 to cluster 2"
        cluster1.remove(person)
        cluster2.add(person)
        changed = true
      end
    end

    # Check members of cluster 2 against the cluster centroids
    puts "Checking cluster 2"
```

```
    people2.each do |person|
      if cluster1.calculate_gd(person) < cluster2.calculate_gd(person) then
        puts "Feature vector moved from cluster 2 to cluster 1"
        cluster2.remove(person)
        cluster1.add(person)
        changed = true
      end
    end

  end

  puts

  puts "Cluster 1 contains"
  people = cluster1.get_people
  people.each do |person|
    person.each do |key,value| print "#{key}=#{value} " end
    puts
  end
  puts

  puts "Cluster 2 contains"
  people = cluster2.get_people
  people.each do |person|
    person.each do |key,value| print "#{key}=#{value} " end
    puts
  end
  puts

end
```

Listing 2 begins by adding the sample data to the clusters. Next, it calculates the centroid for each cluster (as the mean of its members), then checks each member's distance from the two clusters. If the member is farther away from the centroid in its current cluster, then it's removed from its original cluster and added to the other cluster. After all members are checked, you recalculate the centroids and check the members again. The algorithm finishes when no further changes occur.

To cluster the sample users into *k* = 2 clusters, invoke the `kmeans` method, as shown in the console session in Listing 3.

Listing 3. Running the simple *k*-means implementation

```
$ ruby kmeans.rb
Checking cluster 1
Feature vector moved from cluster 1 to cluster 2
Checking cluster 2
Feature vector moved from cluster 2 to cluster 1
Checking cluster 1
Checking cluster 2

Cluster 1 contains
cloud=6 java=0 linux=13 oss=10 agile=0
cloud=9 java=0 linux=11 oss=0 agile=1

Cluster 2 contains
cloud=0 java=9 linux=0 oss=3 agile=8
cloud=1 java=6 linux=3 oss=0 agile=7

$
```

The algorithm successfully clusters Marc and Elise into cluster 1 and Megan and Jill into cluster 2. With the clustering complete, a recommendation engine can use the differences among a cluster's members to form recommendations.

A do-it-yourself approach isn't your only option for building a recommendation capability. Next, I'll explore two open source recommendation engines in depth and briefly describe some other available open source solutions.

SUGGEST and Top-N recommendations

SUGGEST is a Top-N recommendation engine, implemented as a library. SUGGEST (which was developed by George Karypis at the University of Minnesota) uses several collaborative-filtering algorithms and implements user-based and item-based collaborative filtering. The algorithm can be specified when a particular dataset is initialized.

Data is provided to SUGGEST through a set of user-item transactions. In this article's example, the data represents the blogs each user has read. In a more accurate model, you would present the individual historical transactions, such as the particular articles read. In this case, I'll keep it simple and continue using the existing data set (four users, five blogs).

SUGGEST exposes a simple API. You need only three functions to build a recommendation engine in the C programming language:

- `SUGGEST_Init` loads the historical transactions, defines the particular recommendation algorithm, and initializes the recommendation instance.
- `SUGGEST_TopN` computes a recommendation based on passed sample data.
- `SUGGEST_Clean` frees the recommendation engine instance that `SUGGEST_Init` creates.

A fourth function, `SUGGEST_EstimateAlpha`, defines the optimal alpha value for the probability-based algorithm (if that algorithm is used).

Listing 4 presents a simple recommendation implementation that uses SUGGEST.

Listing 4. Sample application (written in C) that uses the SUGGEST recommendation engine

```
#include <stdio.h>
#include "suggest.h"

int nusers = 4;
#define MARC   0
#define MEGAN   1
#define ELISE   2
#define JILL   3

int nitems = 20;   // Marc  Megan  Elise  Jill
#define LINUX1   0  //  1      0      1     0
#define LINUX2   1  //  1      0      0     1
#define LINUX3   2  //  1      0      1     0
#define LINUX4   3  //  1      1      1     0
#define OSS1   4  //  1      1      0     1
#define OSS2   5  //  0      0      1     1
#define OSS3   6  //  1      0      1     0
#define CLOUD1   7  //  0      1      1     0
```

```
#define CLOUD2   8  //  1      1      1     1
#define CLOUD3   9  //  0      0      0     1
#define CLOUD4   10 //  1      0      1     1
#define CLOUD5   11 //  0      0      0     0
#define JAVA1   12 //  0      1      0     1
#define JAVA2   13 //  0      1      0     0
#define JAVA3   14 //  1      1      0     1
#define JAVA4   15 //  0      1      1     1
#define JAVA5   16 //  0      0      0     1
#define AGILE1   17 //  0      1      0     1
#define AGILE2   18 //  0      1      0     0
#define AGILE3   19 //  0      0      1     1

#define NTRANS   40

// Historical transactions of users and items.
int userid[NTRANS] =
  { MARC , MARC , MARC , MARC , MARC , MARC , MARC , MARC , MARC,

    MEGAN, MEGAN, MEGAN, MEGAN, MEGAN, MEGAN, MEGAN, MEGAN, MEGAN, MEGAN,

    ELISE, ELISE, ELISE, ELISE, ELISE, ELISE, ELISE, ELISE, ELISE,

    JILL, JILL, JILL, JILL, JILL, JILL, JILL, JILL, JILL, JILL, JILL, JILL,
  };

int itemid[NTRANS] =
  { /* Marc Blog Reads */
    LINUX1, LINUX2, LINUX3, LINUX4, OSS1, OSS3, CLOUD2, CLOUD4, JAVA3,

    /* Megan Blog Reads */
    LINUX4, OSS1, CLOUD1, CLOUD2, JAVA1, JAVA2, JAVA3, JAVA4, AGILE1, AGILE2,

    /* Elise Blog Reads */
    LINUX1, LINUX3, LINUX4, OSS2, OSS3, CLOUD1, CLOUD2, JAVA4, AGILE3,

    /* Jill Blog Reads */
    LINUX2, OSS1, OSS2, CLOUD2, CLOUD3, CLOUD4, JAVA1, JAVA3, JAVA4, JAVA5, AGILE1, AGILE2
  };


int main()
{
  int *rcmd_handle;
  int behavior[7]={LINUX1, LINUX2, LINUX4, OSS1, OSS2, CLOUD1, CLOUD3 };
  int recommendation[2];
  int result, i;

  rcmd_handle = SUGGEST_Init( nusers, nitems, NTRANS, userid, itemid,
                              3, 2, 0.0 );

  result = SUGGEST_TopN( rcmd_handle, 7, behavior, 2, recommendation );

  if (result) {
    printf("Recommendations (%d) are %d and %d\n", result,
            recommendation[0], recommendation[1]);
  } else printf("No recommendation made.\n");


  SUGGEST_Clean( rcmd_handle );

  return 0;
}
```

Listing 4 begins with a definition of your historical data, defined in two arrays (`users` and `items`). The two arrays are related insofar as `user [index]` defines the user-item transaction (`users[0]`

bought `items[0]`, and so on). For simplicity and readability, the arrays are defined in user-item order.

Use of the SUGGEST solution appears in the short `main` function in Listing 4. There, begin by initializing your recommender with the sample user-and-item data (taken together to represent multiple users' transactions over a set of items). Pass the `user` and `items` array into `SUGGEST_Init`, along with their limits and the possible values (user and item identifiers). Also, specify the item-based Top-N recommendation algorithm (user-based Top-N with cosine-based similarity function) and the size of the neighborhood for recommendation. Next, fill your "basket" with items to calculate the recommendation request, and call `SUGGEST_TopN` to request the recommendation. Finally, free the resources associated with the recommender and exit. Note in this example that the website offers 20 articles over a range of five topics, with historical data on four users.

Execution of the sample SUGGEST recommender is shown in the console session in Listing 5.

Listing 5. Running the SUGGEST recommendation engine with sample data

```
$ ./topn
Recommendations (2) are 2 and 8
$
```

In this case, the engine recommends that the particular reader might like LINUX3 and CLOUD2. These recommendations are reasonable, considering the user's prior behavior and the frequency of that user's views. Note that the result of the call to `SUGGEST_TopN` is the number of recommendations in the recommendation array.

SUGGEST is a simple but effective implementation of Top-N collaborative filtering algorithms.

RESTful web services and easyrec

The easyrec open source recommendation engine takes a novel approach to the construction of a recommendation service. (easyrec is developed and maintained by Research Studios Austria through funding from the Austrian Federal Ministry of Science and Research.)

easyrec exposes a Representational State Transfer (REST) interface to detach it entirely from the developer's language of choice. This approach enhances developers' ability to integrate the service with end-user applications, and it improves the service's scalability.

The easyrec API exposes a rich set of RESTful interfaces that cover all of the possible actions required for a recommendation system, including item purchasing, viewing, and rating. These actions are recorded within the easyrec database. Recommendations are provided through a specific set of interfaces, such as "related items to a given item," "other users also viewed these items," "specific recommendations for a given user," and "other users also purchased." These fundamental recommendation actions cover the most common cases found in many prediction scenarios. The easyrec API supports both XML and JSON for responses.

Now I'll show you an example of requesting a recommendation for a given user with easyrec. The easyrec solution is offered as a package you can integrate into your own setting, but a demo

server is also available to use for testing. Next you'll use this server to request recommendations by using easyrec's test data.

You can use the simple Ruby script in Listing 6 to request a recommendation for a specified user. The users and a list of items are predefined in the easyrec database. Listing 6 defines a Ruby function for interacting with the remote web service. This function builds a URL that represents the request to easyrec. (In this case, `recommendationsforuser` is the requested command.)

Listing 6. Ruby function to interact with easyrec for a recommendation

```
#!/usr/bin/ruby

require 'rubygems'
require 'json'
require 'net/http'

def recommend_for_user( userid )

  puts "Request for recommendations for user #{userid}"

  apikey = "3d66b20f7cbf176bf182946a15a5378e"

  request = "http://intralife.researchstudio.at:8080" +
            "/api/1.0/json/recommendationsforuser?" +
            "apikey=#{apikey}&" +
            "tenantid=EASYREC_DEMO&" +
            "userid=#{userid}&" +
            "requesteditemtype=ITEM"

  resp = Net::HTTP.get_response(URI.parse(request))

  parsed = JSON.parse(resp.body)

  parsed2 = parsed['recommendeditems']

  parsed2['item'].each do |item|
    puts item['description']
  end

end
```

As shown in Listing 6, in addition to the REST namespace and command, you define:

- The `apikey` (using the sample key from the easyrec website)
- The `tenantid`
- The user ID of interest for the recommendation

You also specify that you're interested in the `ITEM` type, to permit all recommended products. With your Uniform Resource Identifier (URI) defined, you use Ruby's `Net:HTTP` client API to send the request and return and store the response. This response is parsed, then it's iterated on each returned item.

Listing 7 illustrates the use of the `recommend_for_user()` function within the Interactive Ruby Shell (IRB) for a particular user. In this case, the recommendations consist of music selections based that user's preferences.

Listing 7. Using easyrec to make a recommendation

```
irb(main):027:0> recommend_for_user( "24EH1723322222A3" )
Request for recommendations for user 24EH1723322222A3
Beastie Boys - Intergalactic
Gorillaz - Clint Eastwood
irb(main):028:0>
```

Listing 7 demonstrates the simplicity of the interface easyrec provides, which can be implemented in any language that supports an HTTP client and a JSON or XML parser.

Other open source offerings

Several other recommender solutions or elements of solutions are available as open source. This section explores some of these offerings. (See Related topics for these and other options.)

LensKit

LensKit (from the University of Minnesota) is a framework for the construction of recommender systems that is often used in the study of collaborative filtering. The goal of LensKit is a high-quality implementation that is both readable and accessible for integration into web applications.

Crab

The Crab recommender-engine framework is built for Python and uses some of the scientific-computing aspects of the Python ecosystem, such as NumPy and SciPy. Crab implements user- and item-based collaborative filtering. The project plans to implement the Slope One and Singular Value Decomposition algorithms in the future, and eventually to use REST APIs.

MyMediaLite

MyMediaLite is a lightweight library of several recommender-system algorithms you can use to develop recommendation engines. MyMediaLite implements rating prediction and item prediction based on user-item rating feedback. The library is implemented in C# running on the Microsoft .NET platform (with Mono support in Linux®).

Waffles

Waffles is a machine-learning toolkit developed in C++ at Brigham Young University by Mike Gashler. Waffles constructs a collection of CLI-based tools. These tools implement granular tasks for machine learning, including recommendation, thereby supporting the ability to script machine-learning tasks at a higher level than API integration requires. By including a large number of parameters for the available tasks, Waffles enables operations to be finely tuned to the task at hand.

Duine

Duine is a software library for prediction-engine development from the Telematica Instituut, Norway. Duine uses the Java™ language to implement collaborative filtering techniques in addition

to information filtering. The last commit for this framework was in 2009, so it's likely that the project is now inactive.

Recommenderlab

Recommenderlab is a collaborative filtering extension for the R environment. Recommenderlab provides a general infrastructure for research into and development of recommender systems. The Recommenderlab environment facilitates both algorithm development and evaluation and comparison among multiple algorithms.

Other offerings

The open source domain has several other offerings that serve as development environments for algorithm construction and evaluation. The Shogun machine-learning toolbox focuses on large-scale kernel methods (such as the Support Vector Machine [SVM]) in addition to a number of other methods and models. Milk is a machine-learning toolkit for Python that focuses on supervised classification methods (SVMs, k-NN, and so on). Finally, Weka provides data-mining software written in the Java programming language.

Conclusion

As the web increases in scale, the push for personalization continues to engender new algorithms and rich, varied solutions within the recommendation-engine space. A wide range of projects in several programming languages — from environments for the development of collaborative filtering algorithms to frameworks for algorithm evaluation all the way to full-featured solutions — are available in the open source domain.

Related topics

- "A Survey of Collaborative Filtering Techniques" (Xiaoyuan Su and Taghi M. Khoshgoftaar, *Advances in Artificial Intelligence*, 2009): Get a deeper introduction to collaborative filtering algorithms, including user-based and item-based Top-N algorithms.
- developerWorks Premium: Provides an all-access pass to powerful tools, curated technical library from Safari Books Online, conference discounts and proceedings, SoftLayer and Bluemix credits, and more.
- *k*-means clustering: Check out the Wikipedia page on *k*-means, which includes numerous links to implementations and visualizations.
- Recommender Systems: Visit this wiki to access information about recommender systems from a wide range of sources.
- SUGGEST: Learn more about this simple but effective library for the development of C-based user and item collaborative filtering.
- Top-N recommendation algorithms: Read more about Top-N algorithms.
- easyrec: Visit the easyrec project site for more information about this web-service-based recommendation engine.
- Explore these sites to learn more about the open source recommender tools and solutions that this article briefly introduces:
  - MyMediaLite
  - LensKit
  - Duine
  - Crab
  - Waffles
  - Recommenderlab
- For a closer look at open source software for data mining and machine learning, check out:
  - Shogun
  - Milk
  - Weka
  - Apache Mahout
  - likelike
  - OpenSlopeOne