Why Recommendation is Special?

Pei Wang
IntelliGenesis Corporation
and
Center for Research on Concepts and Cognition, Indiana University
Homepage: www.cogsci.indiana.edu/farg/pwang.html
E-mail: pwang@cogsci.indiana.edu

A recommendation system, in its simplest form, can be described as the following:

> There is a collection of objects stored in a computer system, and each of them is characterized by a set of attributes. The system's task is to select some objects according to the user's requirement.

A typical example of recommendation is shopping selections. Let us assume that there is a database storing information for a certain type of product, and the user provides requirements to the recommendation system, then gets back information on some products as the system's suggestion.

Described in this way, recommendation can be categorized as information retrieval. Why we cannot just use conventional mechanism, such as relational database, to handle this? It seems that what we need here is just a user-friendly interface to generate database queries. Actually it is what often happens: the user's request is put into a database query, and the result for the user is the product list returned by the query.

Obviously, this is not a recommendation system we are interested in, otherwise we are simply attaching a new label to a mature technology. To see what a recommendation system should do, let us first study when a simple database query is not enough.

If the user's requests can be accurately captured by database queries, no recommendation is needed. On the other extreme, if the user has no request at all, no recommendation is needed, neither (everything is equally good). Therefore recommendations become necessary when the user's requests cannot be properly represented as database queries.

For a single attribute, a database query can pick up entries whose attribute value falls into a certain range. However, to specify such a range is not always as easy as it seems to be.

For example, if you want to buy a computer but have little technical knowledge, you may know that you want something runs fast, and can store lots of data, but may not be able to specify intervals for CPU speed, RAM capacity and hard-disk volume.

Even in familiar domains, intervals may not serve you well, unless you know the distribution of the data. For example, if you were looking for a flight that departs around 9am, the interval [8:55am, 9:05am] might be too narrow (since there was no matching flight), while the interval [7 am, 11 am] might be too wide (since you got 50 matching flights). Even when interval [8:50 am, 9:10 am] gave you exactly one flight, you might had missed a better one at 8:49 which was much cheaper and arrived much earlier.

In summary, an interval corresponds to a binary distinction: all values inside are equally good, so they are provided to the user without additional help; all values outside are equally bad, so they are withdrawn from the user. However, our preference to an object is often a matter of degree, which is determined in the comparison with other objects. If only one attribute is under consideration, the most natural solution is to sort the objects according to the request (such as "as fast as possible" or "as close to 9 am is possible"), then show the top objects to the user.

Though a simple solution, this method already shows some fundamental features that distinguish a recommendation systems from conventional information retrieval systems:

- What really matters is the relative value, not the absolute value, of the attribute — 11 am may be considered as very close to 9 am if all other flights depart in the afternoon.
- To really help the decision of the user, the recommended objects cannot be too few or too many — there are always "other factors" that cannot be put into the request, so to return the top one object turns the recommendation system into a decision making system, while to return the top one hundred objects leaves too much for the user to do.

Intuitively, the default number of recommendations should be 5 to 10, and the user should be allowed to change this default for special situations.

The above method can be extended to handle multiple attributes. At first, each attribute is processed separately, and every object gets a score according to the relative ranking of its value, then a total score is calculated for each object, which is the weighted average of the individual-attribute scores. Here the weights are specified by the user, reflecting the relative importance of the attributes for the final decision. In this way, conflicting requirements on difference attributes can reach a reasonable overall compromise (just as we often prefer a computer which is a little slower but much cheaper).

The weighted-average is not necessarily the most natural operator to use in all situation. For example, if only two attributes are considered, where one is the amount of something, and the other is the total price, then a more reasonable ranking index is the unit price, which is the quotient of the two attribute values. However, the usual situation is that there are more attributes to be considered, and there is no obvious method to combine their values into a single ranking index. In this situation, the weighted-average operator is preferred, because it is domain-independent and well-justified.

For the above idea to work, the key technical issue is the scoring function, which needs to be

data-sensitive, so it will be adjusted automatically when the data changes (For instance, CPU is getting faster and cheaper all the time, and we do not want to manually redesign the score function with each change in the data), and

well-justified, so the user can understand the principles used to get the recommendations and the scores.

Such a recommendation system is available at
http://www.cogsci.indiana.edu/farg/peiwang/SmartRanker/SmartRanker.html
with is based on a previous paper of mine at
http://www.cogsci.indiana.edu/farg/peiwang/papers.html#fuzziness