# Community-Assisted Software Engineering Decision Making

Gregory Gay and Mats P.E. Heimdahl
Department of Computer Science and Engineering
University of Minnesota
greg@greggay.com, heimdahl@cs.umn.edu

## 1. INTRODUCTION

The field of artificial intelligence (AI) as applied to software engineering problems is constantly growing, with conferences devoted to the topic—such as the annual Mining Software Repositories conference—and large open-source data repositories [8]. AI techniques have seen regular use in the software engineering toolbox. For instance, search-based techniques such as genetic algorithms are effectively used to find test cases that satisfy a predefined test adequacy criterion, e.g., branch coverage [4]. Nevertheless, before such techniques can be applied to problems such as test generation, there are a plethora of higher level decisions that must be made. For example, we must determine what coverage criterion will be the most effective when testing the particular system under test, we must determine which search technique will be the most effective given system characteristics and the coverage criterion, and we must determine what variables the test oracle should monitor during the testing process.

At every stage of the development process, similar high-level decisions must be made. Which design modeling notation is most suitable for the problem at hand? Which verification tool is most suitable for the current model? Will agile techniques be suitable in our situation? Facing such questions, from our experience it seems like most organizations make—in hindsight—highly questionable decisions. It would be desirable for one to get recommendations as to which processes, methods, tools, and techniques should be employed to address the software engineering challenges in a particular situation. Traditional AI approaches, such as case-based reasoning or association rule learning, have been successful at making predictions about specific questions that arise during development—questions such as how many defects to expect given certain source code attributes. To date, however, these algorithms have not been widely applied toward making recommendations about broad, high-level problems (such as our development practice dilemma). We envision a future where a *Software Engineering Recommender* can help an organization make informed development decisions even when they are facing a new—and often unique—situation.

We see two main problems in achieving our vision:

1. The factors influencing the decisions to be made are not clear. For example, even for a relatively simple decision such as which test adequacy criterion would be effective, we do not currently know what characteristics of the system affect the criterion efficacy.
2. The recommendations from our SE Recommender do not depend only on the immediate artifacts surrounding the decision at hand. For example, the choice of test adequacy criterion is dependent on the structure of the program under test [10]. However, other factors such as the function computed by the program (the shape of its state-space), the selection of hardware (affecting observability and controllability of tests), percentage of the variable space monitored, availability of tools in the organization, etc. all affect the choice of coverage criterion. A system basing its recommendations only on the program under test will—in the best case—be suboptimal, and—in the worst case—harmful.

The first problem is particularly important because solution quality is often highly dependent on the quality of the data analyzed. Small changes to data sets (such as additions, deletions, or even preprocessing) often lead to conclusion changes [3]. Data sets that include projects developed across multiple companies are infamous for missing, poorly recorded, or unhelpful attributes, and generally require a nearest-neighbor filtering [12] or feature optimization [2] before they become useful.

Inspiration for overcoming these hurdles could be drawn from the field of recommender systems [7]. Such systems often eschew traditional predictive models in favor of collections of weighted keyword vectors. The user's query is transformed into one of these vectors, and recommendations are made based on the contents of or actions taken by similar items in the data model (where similarity is decided by algorithms from the text mining and information retrieval fields).

One could imagine a data model full of project descriptions, where the user is presented with a list of practice recommendations based on the choices made by similar projects. This model could effectively circumvent our second problem by presenting a *series* of recommendations, determined by the most similar project (or, with some calculation, a series of the similar projects). By basing recommendations on textual similarities, we can set up a functioning data model at a low start-up cost. Accuracy of such a model would become an issue. One additional feature of recommender systems, however, relevance feedback, could allow us to address our first problem by *building evidence over time*.

Recommender systems often allow for an iterative process where users offer feedback on recommendations, which is then used to refine the produced results. With slight enhancements, this feedback could be factored into the underlying data model. When presented with recommendations, a user could supplant their *yes* or *no* verdicts with structured feedback on *why* or *why not*. In addition to reshaping results in the current usage cycle, this feedback could also enhance the model itself—adding evidence and context to the recommendations offered to the system. At the end of each feedback cycle, the user-input query information could even be added to the body of data stored in the model.

We propose that the transformation of static predictive models into dynamic community-driven models could assist not only with our goal of making high-level development decisions, but with many of the data problems in software engineering. Relevance feedback could lead to lower initial costs, as smaller amounts of data are needed to seed a model. Over time, these models could be continually updated with new examples and revisions based on practitioner feedback. This allows us to address data quality issues, collect new measurements, and filter out irrelevant factors.

Therefore, we believe that a future research direction in AI-based software engineering will be the usage of feedback—not just in recommender systems, but in many AI techniques—to improve and reshape both the produced analytical results and the underlying prediction models.

## 2. EXAMPLE: RECOMMENDER SYSTEMS

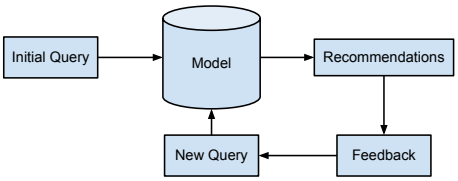


**Figure 1: Model of a typical recommender system.**

Recommender systems are information filtering systems that run a user-supplied query against a predictive model and offer new items of potential interest [7]. These systems can be broadly classified into one of two forms. The first is a *content-based* recommender system, which takes in a query and recommends items with similar features [11]. One example of a content-based system is the Pandora radio program, which seeds the data model with a chosen song or musician and then plays tracks deemed to be similar in musical style to the initially seeded item. The other common type of recommender system utilizes *collaborative* filtering—recommendations are made based on the actions taken by other users of the system. This can be seen on Amazon.com, where users that look at a particular item will see additional recommended purchases based on what other users bundled with that item. These two types of systems are not mutually exclusive—many recommender systems filter content-based recommendations based on the actions of other users. The video streaming service Netflix recommends movies based both on similarity to what you watched and the viewing habits of other users.

Recommender systems are already in use in software engineering research. An entire workshop was devoted to work on this topic recently, with demonstrations of systems that recommended design patterns given project characteristics [9] or elements in model libraries that could be reused to save coding effort [5]. One of the authors of this report previously created a recommender system for locating faults based on linking bug reports with source code [1].

Earlier, we expressed a desire for high-level recommendations on what practices to employ at various stages of development. One could envision an approach to this problem taking the form of a recommender system. At the heart of such a system could be a data model containing (ideally) dozens to hundreds of entries pairing corpora of data on past software projects (descriptions, requirements documents, design specifications, or even source code) with lists of the practice decisions made throughout development. For instance, in the testing stage, a few of these decisions may include program structure choices (e.g. whether to separate compound Boolean decisions into intermediate expressions), test suite size, test generation technique, test adequacy metric, or what percentage of the variable space to monitor during test execution.

This data model differs from many common ones slightly—it may not contain measured attributes justifying each of the individual decisions (although, if available, it certainly could). Project similarity could be calculated instead by comparing the corpora of project information. Such a model allows for two things: (1) a broader, more generalized view of the data space and (2) a lower start-up cost—initial recommendations can be made without much prior knowledge of the factors leading to decisions and, if recommendations are presented based solely on the *most similar* project, without needing to consider the dependence of one decision on another (if solutions are based on multiple similar projects, more care will need to go into recommendations). Indeed, such a model may be more palatable to privacy-conscious data sources; they could offer a list of recommendations along with as little or as much project information as they wish (of course, more data will generally lead to better solutions).

The operation of this recommender system (and of recommender systems in general) is visualized in Figure 1. The user inputs an initial set of information relating to the project under development (a description, solicited requirements, domain-specific rules, tool decisions mandated by business rules or previous development, etc). This input is transformed into a weighted vector of key terms and passed into the data model, which would output practice recommendations based on the decisions made during the development of similar projects.

The next step is of particular interest; the user can offer feedback on these recommendations. This typically comes in the form of a positive or negative response (either a binary choice or a numeric scale). This feedback is used to *alter* the user's original query, employing relevance filtering algorithms such as Rocchio [6] to increase or decrease the weights or add and remove terms. This process produces new recommendations, calculated using the altered query. This feedback and recommendation loop continues until the user is satisfied with the offered solutions.

### 2.1 Enhancing Relevancy Feedback

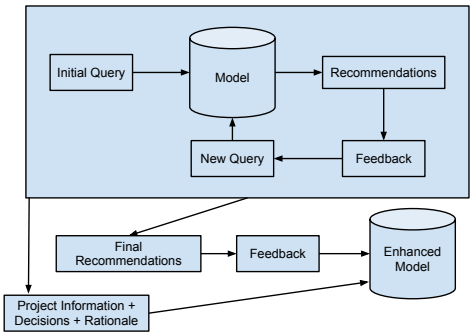


**Figure 2: Model of an enhanced recommender system.**

Although the data model proposed in the previous section allows for lower initial costs, as predictions are based on textual similarities rather than concrete knowledge of individual factors, the produced recommendations may not be accurate. Even if the recommended practices were successful in projects with similar goals, designs, or product domains, other internal conditions may require the use of different tools. Providing feedback to the recommender system in the standard "yes" or "no" format could improve accuracy, but such recommendations would still lack anything more than anecdotal evidence.

An extension to the relevancy feedback loop pictured in Figure 1 could transform the underlying static prediction model into a dynamic one where evidence is grown over time to support the produced recommendations. This enhanced loop is visualized in Figure 2.

In this new usage model, a user does not simply provide positive or negative responses to recommendations. Instead, they reply with data on why each attempted recommendation would work or why it wouldn't. This feedback would need a standardized structure. The correctness verdict (yes or no) could be accompanied by suggestions for measurements that contributed to that verdict as well as values for a series of data attributes already suggested by other users—in the form of the numeric measurements or textual responses familiar to users of standard data sets. The user could additionally provide feedback on the existing attributes.

A simple example can be seen in Table 1 where the recommender system suggests the use of MC/DC as a test suite adequacy metric. The user states that this was a relevant suggestion, and fills in values for two existing evidence-providing attributes. The user also suggests the measurement of the ratio of Boolean expressions to numeric calculations as

**Table 1: Feedback Sample**

| Recommendation | Relevant | Num. Boolean Expressions | Num. Numeric Calculations | New Attribute Suggestions and Values |
|---|---|---|---|---|
| **MC/DC for Test Suite Adequacy** | Yes | 219 | 73 | Ratio of Boolean to Numeric, 3:1 |

another potentially helpful attribute in making the recommendation of a test suite adequacy metric.

This feedback could, of course, still be used in the typical manner—reshaping the user's query to the existing predictive model. Once the recommendation cycle is finished, however, the user's feedback could be factored back into the predictive model, through a combination of automatic and manual techniques, as new evidence for the model data. With consent, the project data and chosen practices provided by the user could also be added into this enhanced data model.

In essence, this creates small predictive models underneath each of the individual recommendations that the overall data model would offer—models that could be mined to provide backing evidence for why this recommendation would be the correct choice to make. Rather than the standard static data set, this represents a model that can evolve over time in both quantity of data and the quality of the provided results.

# 3. IMPLICATIONS AND CHALLENGES

The use of feedback mechanisms to transform static prediction models into evolving ones is exciting for multiple reasons. First, it allows for lower start-up costs. Even if the data attributes to study are not well-known, a data model could be set up connecting outcomes to generalized information about software projects (things like descriptions, product domain rules, and design documents). Even if such models are not initially accurate, they have the potential to improve as new evidence is added.

Second, such models could help solve the data quality and quantity issues that are brought up frequently in AI research. Unhelpful data attributes could be automatically removed over time as negative reports mount, lessening the need for data preprocessing. New attributes will be added, creating additional opportunities for model analysis. User data could be imported, growing the number of records for others to consider.

The use of feedback to improve models could even lead to a Wikipedia-like effect on data repositories—where the continual improvements to data cause a continual increase in both the number of users and the number of edits made to the model. We envision a future where software development decisions aren't just assisted by analysis of a few data records, but by powerful community-assisted prediction models.

There, of course, exist a number of challenges to solve in implementing the ideas presented in this report:

- While recommender systems offer the inspiration for these feedback mechanisms, such techniques are by no means limited to such systems. Research will need to be pursued on how to effectively build feedback into other AI approaches. This will likely necessitate a shift from data sets as static files that sit on a desktop to cloud-based models with accompanying feedback hooks. These models could even be connected into development environments, automatically collecting data and providing advice.
- If a user suggests new attributes, when should those be added to the model and presented to other users? Similarly, when should existing attributes that are deemed unhelpful be phased out? What measures should be taken to group together and standardize similar suggestions for new attributes?
- As new attributes and evidence is added to the data model, older data in the model may lack the amount or sophistication of data that newer projects contain. Steps will need to be taken to either update older data, account for missing data, or outright remove records that becomes less useful. Previous contributors should be frequently invited to submit revisions to their data, and some amount of manual editing on the part of data-set retainers may be necessary.
- There is the additional problem of convincing users to actually provide feedback. Academic researchers can, to some extent, be convinced to contribute by the promise that others will also make contributions, creating a stronger pool of accessable data. Industrial practicioners will need to be convinced that the effort spent providing feedback will result in a worthwhile return on investment.

# 4. SUMMARY

We propose that there are a class of problems—such as deciding on a series of development practices — that challenge many AI approaches for two reasons: (1) well-defined data is necessary, but it is not clear which factors are important and (2) multiple recommendations must be made, and dependence must be considered.

Recommender systems offer inspiration. First, they can make use of data models that broadly pair a series of recommendations with generalized information like project descriptions and design documents. Second, they offer feedback mechanisms to refine the calculated recommendations. Detailed feedback could be factored back into the data model to, over time, build evidence and context for recommendations.

Existing algorithms and data models would be amenable to the addition of feedback mechanisms, and the use of these dynamic models could lower start-up costs and generate more accurate results as the model grows. We believe that feedback-driven dynamic prediction models will become an exciting research topic in the field of AI-based software engineering.

# 5. REFERENCES


[1] G. Gay, S. Haiduc, A. Marcus, and T. Menzies. On the use of relevance feedback in IR-based concept location. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 351 –360, sept. 2009.

[2] G. Gay, T. Menzies, M. Davies, and K. Gundy-Burlet. Automatically finding the control variables for complex system behavior. *Automated Software Engg.*, 17(4):439–468, Dec. 2010.

[3] M. Harman and B. Jones. Search-based software engineering. *Journal of Information and Software Technology*, 43:833–839, December 2001.

[4] M. Harman and P. McMinn. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *IEEE Transactions on Software Engineering*, pages 226–247, 2009.

[5] L. Heinemann. Facilitating reuse in model-based development with context-dependent model element recommendations. In *Third International Workshop on Recommendation Systems for Software Engineering*, 2012.

[6] C. Manning, P. Raghavan, and H. Schutze. *An Introduction to Information Retrieval*. Cambridge University Press, 2009.

[7] P. Melville and V. Sindhwani. Recommender systems. *Encyclopedia of Machine Learning*, 2010.

[8] T. Menzies, B. Caglayan, E. Kocaguneli, J. Krall, F. Peters, and B. Turhan. The promise repository of empirical software engineering data, June 2012.

[9] F. Palma, H. Farzin, and Y.-G. Gueheneuc. Recommendation system for design patterns in software development: An DPR overview. In *Third International Workshop on Recommendation Systems for Software Engineering*, 2012.

[10] A. Rajan, M. Whalen, and M. Heimdahl. The effect of program and model structure on MC/DC test adequacy coverage. In *Proc. of the 30th Int'l Conference on Software engineering*, pages 161–170. ACM, 2008.

[11] F. Ricci, L. Rokach, and B. Shapira. *Recommender Systems Handbook*. Springer, 2011.

[12] B. Turhan, T. Menzies, A. B. Bener, and J. Di Stefano. On the relative value of cross-company and within-company data for defect prediction. *Empirical Softw. Engg.*, 14(5):540–578, Oct. 2009.