Chapter 1
Recommendation Systems in-the-Small

Laura Inozemtseva, Reid Holmes, and Robert J. Walker

Abstract Many recommendation systems rely on data mining to produce their recommendations. While data mining is useful, it can have significantly implications on the infrastructure needed to support and to maintain an RSSE; moreover, it can be computationally expensive. This chapter examines recommendation systems in-the-small (RITSs), which do not rely on data mining. Instead, they take small amounts of data from the developer's local context as input and use heuristics to generate recommendations from that data. We provide an overview of the burdens imposed by data mining, and how these can be avoided by a RITS through the use of heuristics. Several examples drawn from the literature illustrate the constraints and design choices of RITSs. We provide an introduction to the development of the heuristics typically needed by a RITS. We discuss the general limitations of RITSs.

1.1 Introduction

Many recommendation systems rely on *data mining*; that is, attempting to discover useful patterns in large data sets. While such recommendation systems are helpful, it is not always practical to create, maintain, and use the large data sets they require. Even if this issue is resolved, data mining can be computationally expensive, and this may prohibit interactive use of the recommendation system during devel-

Laura Inozemtseva
David R. Cheriton School of Computer Science, University of Waterloo, 200 University Avenue West, Waterloo, ON, Canada, N2L 3G1, e-mail: lminozem@uwaterloo.ca

Reid Holmes
David R. Cheriton School of Computer Science, University of Waterloo, 200 University Avenue West, Waterloo, ON, Canada, N2L 3G1, e-mail: rtholmes@cs.uwaterloo.ca

Robert J. Walker
Department of Computer Science, University of Calgary, 2500 University Drive NW, Calgary, AB, Canada, T2N 1N4, e-mail: walker@ucalgary.ca

opment. For these reasons, RSSE researchers have developed a number of *recommendation systems in-the-small* (RITSs): systems that base their recommendations on the developer's immediate context, possibly using heuristics to compensate for the smaller amount of available data.

This chapter introduces this class of recommendation systems. Section 1.2 explains what recommendation systems in-the-small are by comparing a RITS with a system that uses data mining to generate its recommendations. Section 1.3 demonstrates the diversity of RITSs by introducing a few example systems from different application areas. Section 1.4 describes how the heuristics typically used in RITSs can be generated and the pros and cons of each method; we make the discussion more concrete by revisiting the recommendation systems discussed in Sect. 1.3. Section 1.5 discusses some of the limitations of this class of recommendation systems. Section 1.6 concludes the chapter and provides suggestions for further reading.

1.2 Recommendations with and without Data Mining

To illustrate the difference between RITSs and recommendation systems that use data mining to generate their recommendations, which we refer to as *data mining recommendation systems* (DMRSs), we examine a RITS and a DMRS that solve the same problem in different ways.

Consider the problem of finding code elements (functions, variables, etc.) that are relevant to a given task. This is an issue that software developers frequently face as they fix bugs, add new features, or perform other maintenance tasks. In fact, one study estimates developers spend 60–90% of their time reading and navigating code [3]. However, finding relevant code can be difficult. Basic lexical searches are one possible strategy, but are often insufficient since the developer may not know the correct search term to use. Even if the developer can find some of the relevant elements, it has been shown that it is difficult for a developer to faithfully find all dependencies from an entity [5]. A recommendation system that can suggest relevant code elements would therefore be helpful to developers.

Robillard [11] tackled this problem by developing a RITS called Suade. Suade takes as input a set of code elements that are known to be relevant to the developer's task and produces as output a ranked list of other code elements that are also likely to be relevant. Suade produces these recommendations by doing a localized analysis of the program's dependency graph to identify relevant elements. It does this in two steps. First, Suade builds a graph of the code elements in the project and the relationships that exist between them, such as function calls and variable accesses. When a developer initiates a query, Suade identifies elements in the graph that have a direct structural relationship to the known-relevant elements. It then uses two heuristics to rank the related elements from most to least likely to be relevant. We describe these heuristics in more detail in Sect. 1.3.

Though Suade builds a graph of relationships between code elements, it is a RITS because it does not generate recommendations by mining the full graph for

patterns. Rather, it takes a small amount of localized information as input. The graph is only used to look up information about relationships; heuristics are used to rank the recommendations.

Zimmermann et al. [14] took a different approach and built a DMRS called eROSE (originally called ROSE). eROSE mines the project's version control system to identify program elements that are frequently changed together (in the same commit). This information is used to identify relevant elements, on the assumption that elements that changed together in the past will likely change together in the future. More specifically, when a developer modifies a program element e as part of a maintenance task, eROSE presents a list of other program elements that may need to be changed, ranked by how confident the system is about the necessity of the change. The confidence value for each recommended element e' is based on two factors: first, the number of times e and e' have occurred in the same commit in the past; and second, out of the total number of times e was changed in the past, the percentage of those times that e' also changed.

eROSE is a DMRS because it must mine a version control system to identify commits in order to produce a recommendation. Given that source code is updated incrementally, it may be possible for eROSE to update its commits incrementally, but it is not possible to avoid data mining completely.

RITSs and DMRSs have different strengths. Data mining can reveal patterns that RITSs might miss. For example, eROSE's recommendations are not limited to source code: if a documentation file always appears in the same change set as a source code file, eROSE can identify that relationship. Suade cannot do this because it uses a graph of relationships that are specific to a programming language; while in principle it could be adapted to work for any programming language, this would require targeted development to accomplish.

That said, RITSs tend to be more lightweight and produce recommendations more quickly. In this case, eROSE is not particularly slow, and could be used interactively, but in general, mining a repository on every query can be computationally expensive and may not scale well to large projects. In addition, in order to use a DMRS, a repository of data must have been collected for the DMRS to mine. For eROSE, the history of commits must have been collected over time; when a project is new, there is no history and DMRSs can provide no help (this is known as the *cold start problem* [10]). A RITS is not subject to the cold start problem.

Figure 1.1 shows these differences diagrammatically. The general structure of the two diagrams is identical: the developer builds the recommendation system; then the developer uses the recommendation system, whereby queries are issued and recommendations are returned. The key difference is: for the RITS (on the left), the developer must determine one or more heuristics to build into the recommendation system before it can be used; for the DMRS (on the right), the recommendation system must mine its data source. If the data source is unchanging, in principle the DMRS would not need to repeat the mining step, but it could simply make use of the patterns it had already detected; this is effectively a caching optimization.

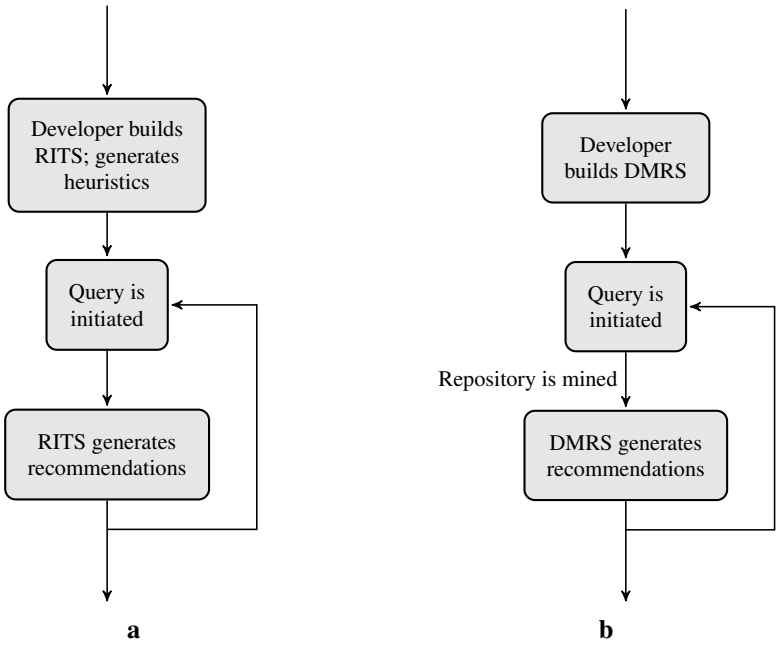

Fig. 1.1: Flowcharts illustrating how recommendation systems are built and used. **a** Building and using a RITS. **b** Building and using a DMRS

1.3 The Versatility of Recommendation Systems in-the-Small

In this section we describe several recommendation systems in-the-small from the literature to demonstrate their versatility: Suade (in more detail); Strathcona; Quick Fix Scout; an unnamed Java method-name debugger; and YooHoo. These five tools illustrate the variety of tasks that RITSs can perform; they also provide an introduction to the development and inherent constraints of RITSs.

1.3.1 Suade: Feature Location

As noted above, Suade helps developers find code elements that are likely to be relevant to a particular task. It uses two heuristics to rank the results from most to least likely to be relevant. The first heuristic, which Robillard refers to as *specificity*,[1] considers how "unique" the relationship is between a given element and the known-relevant elements. Consider Figure 1.2. Each node represents a code element. The

[1] Not to be confused with the term that is synonymous with true negative rate.

shaded elements (D, E, and F) are known to be relevant to the current task, while the unshaded elements (A, B, and C) may or may not be relevant. Arrows represent a *uses* relationship; for example, the `addListener(TestListener)` method (node A) calls the `add(?)` method (node E). In this figure, node E has three neighbors of unknown relevance: A, B, and C. However, node F only has one neighbor of unknown relevance: C. C therefore has a more "unique" relationship with a known-relevant element than A or B do, so it will be given a higher specificity score. That is, Suade thinks C is more likely to be relevant to the current task than A or B.

The second heuristic, *reinforcement*, considers how many of the neighbors of a known-relevant element are also known to be relevant. In the figure, F has two neighbors, C and D. D is known to be relevant, so it seems likely that C is also relevant, since it is the "odd one out". Node E has three neighbors, and none of them are known to be relevant, so there is no particular reason to think any of the three are in fact relevant. Suade will therefore give C a higher reinforcement score than A or B. That is, Suade thinks C is more likely to be relevant to the current task than A or B is.

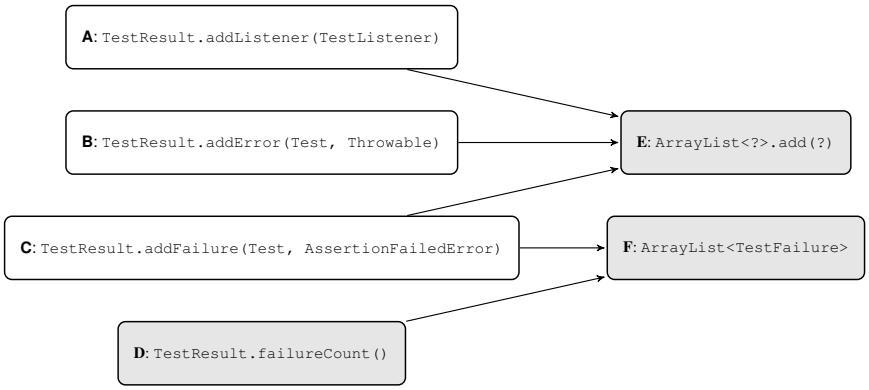

Fig. 1.2: An example showing specificity and reinforcement using relationships from JUnit's `TestResult` class. The gray elements are known to be relevant; the white elements are of unknown relevance. Arrows represent a *uses* relationship. C is both more specific and more reinforced than A and B

The results of these two heuristics are combined so that an element that is both specific and reinforced is preferred to one that is specific but not reinforced or vice-versa. Suade does not use a threshold value to limit the number of displayed results; all elements that are related to the known-relevant elements are shown to the developer in their ranked order.

Suade illustrates that RITSs can be used for code exploration and understanding tasks. Though RITSs use only the developer's context as input, they can identify useful information in other parts of the developer's project without mining its source code or associated metadata, as eROSE does.

```
public class MyView extends ViewPart {
   public void updateStatus(String msg) {
     Action run = new Action("Run", IAction.AS_CHECK_BOX);
     ...
     IStatusLineManager.setMessage(msg);
   }
}
```

Listing 1.1: An example of a code snippet that could be submitted to Strathcona as a query. Strathcona will look for examples with similar structural properties; for example, methods in the corpus that also call `Action.Action(String, int)`

1.3.2 Strathcona: API Exploration

Holmes et al. [6] noted that developers sometimes have difficulty using unfamiliar APIs. These developers often look for examples of correct API usage to help them complete their tasks. Unfortunately, such examples are not always provided in the project documentation. The authors created a RITS called Strathcona to help developers locate relevant source code examples in a large code corpus. This is done by extracting structural information from the developer's integrated development environment (IDE) and using this information to locate examples that have similar structure. Four heuristics are used to measure the structural similarity between the developer's code and a given example from the code corpus. Like Suade, Strathcona uses a data repository but does not use data mining to find patterns in the repository; it simply looks up information about examples.

The two simplest heuristics locate examples that make the same method calls and field references as the query. For example, if the developer were to query Strathcona in a file containing the code snippet shown in Listing 1.1, the *calls* heuristic would locate all methods in the corpus that call `Action.Action(String, int)` and `IStatusLineManager.setMessage(String)`. If no method calls both of these methods, then methods that call one or the other would be returned. The *references* heuristic would locate all methods in the corpus that contain a reference to `Action.AS_CHECK_BOX`.

While these two simple heuristics excel at locating examples that use specific methods and fields, they are not good at identifying more general examples. Strathcona uses two additional heuristics to address this shortcoming. Rather than focusing on specific methods and fields, the *uses* heuristic looks at the containing types of the code elements (e.g., `IStatusLineManager`, `Action`, and `IAction` in Listing 1.1) and tries to find methods that use these types in any way. The *inheritance* heuristic also works more broadly, locating any classes that have the same parent classes as the query (e.g., classes that extend `ViewPart` for the query in Listing 1.1). These heuristics are more general than the previous two heuristics and usually match many more code examples.

Strathcona does not weight the heuristics; instead, every example is assigned a score that is the sum of the number of heuristics that it matched. The top ten examples are returned to the developer.

Strathcona illustrates that RITSs can be used for API exploration and code maintenance tasks. Even without data mining, RITSs can harness the knowledge of other programmers to help a developer write new code. In this case, Strathcona's heuristics allow it to find useful examples without mining the example repository.

1.3.3 Quick Fix Scout: Coding Assistance

The Eclipse IDE has many features to make development easier, one of which is quick fix suggestions. When a program has a compilation error, these suggestions allow developers to automatically perform tasks that may resolve the error. For example, if the developer types "`public sTRING name`", the dialog box will recommend changing "`sTRING`" to "`String`". Unfortunately, for more complex errors, the outcome of these quick fix tasks is unpredictable and may even introduce new errors. Muşlu et al. [8] created an Eclipse plugin called Quick Fix Scout that uses *speculative analysis* to recommend which quick fix to use. More precisely, it applies each quick fix proposal to a copy of the project, attempts to compile the resulting program, and counts the number of compilation errors. The plugin does this for each suggested quick fix and reorders the suggestions in the quick fix dialog so that the fixes that result in the fewest compilation errors are the top suggestions. Any two fixes that result in the same number of errors are kept in their original relative order. The heuristic used here is simply that fewer compilation errors is better.

Quick Fix Scout also augments the quick fix dialog box with the global best fix. Sometimes the best suggestion to fix a given compilation error is not found at the error location itself, but at some other program point. Quick Fix Scout checks all suggestions at all error locations to identify the best fix overall, rather than just at the individual program point queried by the developer. The global best fix is shown as the top suggestion in the Quick Fix Scout dialog box.

Quick Fix Scout illustrates that RITSs can help with the day-to-day work of development: their use of heuristics and small amounts of data makes them fast enough to use interactively, which is not always the case for DMRSs. A DMRS developer would probably use static analysis for this application, while Quick Fix Scout can use the simpler "try it and see" approach.

1.3.4 A Tool for Debugging Method Names: Refactoring Support

Høst and Østvold [7] noted that Java programs are much more difficult to understand and maintain when the methods do not have meaningful names; that is, names that

accurately describe what the methods do. As a concrete example, a method named `isEmpty` that returns a string is probably poorly named.

To address this, the authors developed a tool that automatically checks whether a given method's name matches its implementation. The authors used natural language program analysis on a large corpus of Java software to develop a "rule book" for method names, i.e., a set of heuristics that are built into the tool. To return to our previous example, a possible rule is that a method named `is`*Adjective*`(...)` should return a Boolean value. The tool compares the name of a method in the developer's program to its implementation using these rules and determines if it breaks the rules, which the authors call a *naming bug*. If a naming bug is present, the tool generates recommendations for a more suitable name. Specifically, it returns two suggestions: the (roughly speaking) most popular name that has been used for similar methods without resulting in a naming bug; and the name that is (again, roughly speaking) most similar to the current name that does not result in a naming bug. The tool does not use a threshold value when displaying results: it merely displays a list of methods that have naming bugs and two recommended replacement names for each one.

This tool illustrates that RITSs can support refactoring tasks. Moreover, they can obtain many of the benefits of data mining without using it online to generate recommendations. By using data mining to generate heuristics that are built into the tool, the recommendation system can produce better recommendations without the recurring cost of data mining.

1.3.5 YooHoo: Developer Awareness

One downside of relying on external libraries is that it can be hard for developers to keep apprised of when these libraries are updated. If a library is updated in a way that changes the API, the developer's program will stop working for users who have the new version of the library, and the developer may not know about the problem until one of them files an issue report. Holmes and Walker [4] noted that if developers wish to respond proactively to changes in libraries they depend on, they must constantly monitor a flood of information regarding changes to the deployment environment of their program. Unfortunately, the amount of information they must process leads to many developers giving up: rather than proactively responding to changes in the deployment environment, they respond reactively when an issue report is filed. The delay between the change occurring and the issue report being filed may exacerbate the problem and lower users' opinion of the product. In response, Holmes and Walker developed YooHoo, a tool that filters the flood of change events regarding deployment dependencies to those that are most likely to cause problems for a specific developer or project.

YooHoo has two components: *generic change analysis engines* (GCAs) and *developer specific analysis engines* (DSAs). Each GCA is tied to a specific external repository and monitors changes that are made to that repository. When a developer

starts using YooHoo, it automatically generates GSAs for all of the projects the developer depends on. These GSAs generate the flood of change events that may or may not be of interest to the developer.

DSAs are responsible for filtering the events captured by the GCAs. YooHoo monitors the files in the developer's system to identify the set of files they are responsible for. It then determines which external dependencies are used by those files so that it can limit the flood of change events to events coming from those dependencies. In addition, every event that comes from a relevant GCA is analyzed to gauge the severity of its impact on the developer's code. Events that might break the developer's code are shown, as are informational events such as updated documentation. All other events are discarded. This filtering process means the developer will see on the order of 1% of the change events that occur, and most of them will be immediately relevant to the developer's work.

YooHoo illustrates that RITSs can be used for developer awareness and coordination tasks. Recommendation systems in-the-small have two advantages in this application area: they can generate recommendations quickly, and they are designed to make recommendations based on the developer's current context. They are therefore good at producing relevant recommendations for time-sensitive tasks. In YooHoo's case, its heuristics allow it to categorize change events quickly and with high accuracy. An alternative, data mining approach to categorization would likely increase accuracy at the price of speed.

1.4 Developing Heuristics

As we mentioned at the beginning of the chapter, recommendation systems in-the-small frequently use heuristics to compensate for having a limited set of raw data to base recommendations on. These heuristics can be developed in three ways:

- using human intuition about the problem;
- using data mining during the development of the recommender to identify patterns that are built into the recommender as heuristics; and
- using machine learning to learn the heuristics during operation.

Combinations of these approaches can also be used; for instance, human intuition could be used to generate initial heuristics that are then refined by machine learning. Heuristics can also be combined, possibly with different weights assigned to each.

In this section, we discuss the advantages and disadvantages of these different methods of generating heuristics. We also discuss how the recommendation systems described in the previous section make use of these techniques.

1.4.1 Human Intuition

When developing a RITS, the creators often use their intuition about the application domain to develop the heuristics that are built into the recommendation system. Figure 1.3 shows the general procedure for doing this. First, the recommendation system developer chooses some initial heuristics. The developer encodes these into the tool, tries it, and notes whether or not the recommendations are of satisfactory quality. The latter step might be done through manual inspection or empirical analysis. If the results are not satisfactory, the RITS developer can modify the existing heuristics, possibly by weighting them differently or using different threshold values. Alternatively, the developer can develop entirely new heuristics.

In the previous section, we saw three recommendation systems that use heuristics based on human intuition: Suade, Strathcona, and Quick Fix Scout; we examine in turn how the heuristics used by each were developed.

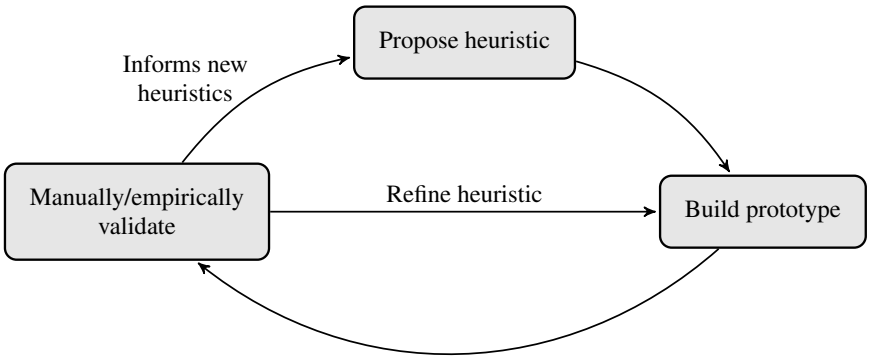

Fig. 1.3: The process that a recommendation system designer uses to develop the heuristics that the system will use

Suade's Heuristics

Suade's specificity and reinforcement heuristics are intuitive in the development setting. Specificity reflects the idea that, if an element of unknown relevance has a unique relationship with an element that is known to be relevant, it is probably relevant itself. Reinforcement reflects the idea that, if most of the elements in a group of structurally related elements are known to be relevant, then the rest of the elements in the group are probably relevant as well.

After building Suade around these heuristics, Robillard performed an empirical study on five programs to evaluate the heuristics. He chose a benchmark for each

program by identifying a group of elements that had to be modified to complete a given task. The benchmarks were either generated for the study or reused from previous work. For each benchmark, the author generated a number of subsets of the known relevant elements, fed them into Suade, and verified that the other elements in the benchmark received high ratings.

Strathcona's Heuristics

The Strathcona system includes four heuristics that Holmes et al. believed would identify relevant source code examples. As previously discussed, the authors began with two simple heuristics, but, finding them inadequate, added two more. The authors evaluated the heuristics with a small user study. They asked two developers to perform four tasks using Strathcona. For three of the tasks, Strathcona could identify relevant examples; the remaining task was deliberately chosen so that Strathcona could not help the developers. The developers were mostly successful at completing the tasks, though one developer did not finish the last task, and reported that Strathcona made the tasks easier. They also quickly realized that one of the tasks had no relevant examples, suggesting that Strathcona does not lead to wasted exploration time in the event that the example repository contains nothing relevant to the current task.

Quick Fix Scout's Heuristics

Quick Fix Scout relies on a simple heuristic: a fix that results in fewer compilation errors is better than one that results in more compilation errors. The way this is applied is trivial: try all fixes for a given error, count the number of remaining errors after the fix is applied, and sort the list to put the fix that resolves the most errors at the top. Quick Fix Scout also puts the global best fix at the top of every quick fix dialog box. This captures the intuitive knowledge that some faults manifest themselves as compilation errors in several program locations, so fixing the underlying fault can resolve several compilation errors at once.

Since it may be hard to believe that such a simple heuristic could be useful, the authors did a thorough evaluation of the tool. First, thirteen users installed the plugin and used it for approximately a year. Second, the authors did a controlled study where twenty students performed twelve compilation-error removal tasks each, using either the regular quick fix dialog or Quick Fix Scout. The results from both studies suggest that Quick Fix Scout is helpful: feedback from the thirteen long-term users was positive, and in the controlled study, students were able to perform compilation-error removal tasks 10% faster.

Advantages and Disadvantages

The main advantage of relying on human intuition is that no data set is needed. The only requirement is that the developer of the recommendation system have sufficient knowledge of the application domain to generate heuristics and evaluate the correctness of the results. It is worth repeating that these heuristics are usually developed iteratively: as the results from one or more heuristics are examined, other heuristics are often introduced to either broaden or limit the results that are already identified. Domain knowledge is again necessary for this process.

Another advantage of these heuristics is that developers are often more willing to trust them, because they tend to be easy to understand and, obviously, intuitive to people who are familiar with the problem domain.

The main disadvantage of relying on human intuition is that it may not be correct. If this is the case, the recommendations will be poorer than they would be if a data-driven approach had been used to generate the heuristics. Even if the heuristics seem to work well, it is possible that better ones exist, but the recommendation designer stopped looking for them once a fairly good set had been identified.

1.4.2 Offline Data Mining

At first glance, using data mining to generate heuristics may seem to contradict the theme of the chapter. However, there is an important distinction between using data mining offline, while building the recommendation system, and using data mining online, as part of the recommendation system. In the first scenario, the developer constructing the recommendation system uses data mining on a large corpus of software projects to identify patterns. This information is then encoded as a heuristic in the recommendation system. The recommendation system itself does *not* perform data mining and does *not* update the heuristics: the heuristics are static. In the second scenario, the recommendation system itself mines a repository of some sort as part of its recommendation generation algorithm. This is the technique used by DMRSs and the one we wish to avoid, as it can be computationally expensive and requires the maintenance of the data repository.

Using offline data mining to generate heuristics is a way of getting some of the benefits of data mining without incurring the expenses of data mining each time we want a recommendation. We saw an example of this technique in the previous section: the tool of Høst and Østvold for debugging method names used a set of rules that were derived by performing natural language program analysis on a large corpus of Java programs. More precisely, the authors identified many attributes that methods can have, such as *returns type in name*, *contains loop*, and *recursive call*. They then used offline data mining to establish statistical relationships between method names and these attributes; for instance, methods with the name `get`*Type*`(...)` nearly always have the attribute *returns type in name*. When a method in the developer's program breaks one of these rules—say, a method named `getList` that

returns an object of type `String`—the recommendation system flags it as a naming bug. The authors did a small evaluation of their rules by running the tool on all of the methods in their Java corpus, randomly selecting 50 that were reported as naming bugs, and manually inspecting them. The authors found that 70% of the naming bugs were true naming bugs, while the other 30% were false positives. Of course, this study is seriously limited, since the same corpus was used for both training and evaluating the tool, but it suggests the technique has promise.

The main advantage of offline data mining is that, if a sufficiently large and diverse repository is mined, the heuristics are likely to be more accurate and possibly more generalizable than heuristics generated with the other techniques.

The main disadvantage of this technique is that it is very susceptible to biased training data. If the repository being mined only contains software projects from, say, a particular application domain, the learned heuristics may not be applicable for other types of software projects and may lead to poor recommendations.

1.4.3 Machine Learning

The previous two heuristic generation techniques produce static heuristics. Since they cannot adapt to a developer's current context, recommendation systems that use static heuristics tend not to generalize as well; this can be mitigated by using machine learning to adapt the heuristics on the fly.

We saw an example of a RITS that uses machine learning in Sect. 1.3. YooHoo contains developer-specific analysis modules that filter the flow of incoming change events to those that are likely to impact a particular developer. This filtering relies on knowing which source files the developer owns, and since these change over time, the DSAs must update themselves to match.

The authors evaluated the generated DSAs with a retrospective study that measured how well YooHoo compresses the flow of change events and whether it erroneously filters important events. They found that YooHoo filters out over 99% of incoming change events, and on average 93% of the impactful events remain after filtering; that is, YooHoo rarely erroneously filters impactful events.

The advantage of machine learning is that it may improve the quality of the recommendations, since the heuristics can evolve with the project. This also avoids one of the issues with using human intuition, where the human finds a "good enough" set of heuristics and stops trying to improve them. In addition, the recommendations will be pertinent to a greater variety of projects, since the heuristics were not developed with a particular (possibly biased) training set, and the heuristics can change to match the characteristics of a new project.

The main disadvantage of machine learning is that it can get bogged down in excess complexity in the data, identifying spurious relationships between variables. When it is fully automated (which is common), it cannot be combined with human analysis; when it is not fully automated (as in *supervised learning*), a developer has to be trained to interact with it and has to be willing to spend the time to do so. Ma-

chine learning also adds considerable complexity to the recommendation system's implementation, which may be undesirable for the sake of maintainability, runtime footprint, and error-proneness.

1.5 Limitations of Recommendation Systems in-the-Small

Given that recommendation systems in-the-small do not use data mining as they generate their recommendations, they cannot be used in application domains where it is necessary to identify patterns in large data sets. Consider code reuse: recommendation systems for this domain help developers reuse existing code components by locating potentially useful ones in a large component repository. The recommender must mine the component repository to obtain the information it needs to recommend components, so RITSs are not suitable for this domain. That said, there are still many areas where recommendation systems in-the-small are applicable and useful, as we saw in Sect. 1.3.

RITSs that do not use machine learning may not generalize well due to their reliance on static heuristics. If the training set does not include a variety of software systems, the RITS may produce poor recommendations when a developer tries to use it on a project that was not represented in the training set. Even if a developer's project is initially similar to one in the training set, if the project changes enough, the recommendations could degrade over time. Both of these issues can be addressed by updating the heuristics, either by incorporating machine learning or by manually updating them in each new version of the RSSE.

Finally, RITSs cannot use information about non-localized trends to guide their recommendations. Since the recommendations are based on the developer's immediate context, they may not be as good as they could be if the recommender had access to information about the rest of the developer's system. Unfortunately, this is part of the fundamental tradeoff between recommendation systems in-the-small and data mining recommendation systems.

1.6 Conclusions and Further Reading

Many recommendation systems use data mining in their recommendation generation algorithms, but this can be expensive and impractical. Recommendation systems in-the-small provide a lightweight alternative and can be used in many of the same application areas. This chapter described five recommendation systems in-the-small—Suade, Strathcona, Quick Fix Scout, the Java method debugger, and YooHoo—that show the diversity that RITSs can attain. However, these tools barely scratch the surface of what is possible with RITSs. Other works of possible interest include:

- SRRS [12], which helps the user choose an appropriate way of specifying the security requirements for their project;
- DPR [9], which recommends appropriate design patterns to help developers preserve their system's architectural quality; and
- Bruch et al. [2], which compares data mining and non-data mining approaches to code completion recommendations.

RITSs often use heuristics to compensate for the smaller amount of data available to them, and we looked at three ways these heuristics can be generated: using human intuition, using offline data mining, and using machine learning. We discussed some of the advantages and disadvantages of these different heuristic generation techniques. Readers who want to learn more about data mining may want to consult a textbook such as Tan et al. [13]; readers who want to use machine learning may be interested in Alpaydin [1].

Finally, we explored some of the limitations of recommendation systems in-the-small and the situations where they cannot be used. While RITSs are in some sense more limited than data mining recommendation systems, they are often easier to develop; they also tend to be easier and more responsive to use since there is no data repository to maintain. For these reasons, recommendation systems in-the-small will continue to hold a prominent position in the field of recommendation systems in software engineering. We encourage the reader to explore this interesting area.

References

1. Alpaydin, E.: Introduction to Machine Learning. 2nd edn. MIT Press (2009)
2. Bruch, M., Monperrus, M., Mezini, M.: Learning from examples to improve code completion systems. In: Proceedings of the European Software Engineering Conference/ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 213–222 (2009). DOI 10.1145/1595696.1595728
3. Erlikh, L.: Leveraging legacy system dollars for e-business. IT Professional **2**(3), 17–23 (2000). DOI 10.1109/6294.846201
4. Holmes, R., Walker, R.J.: Customized awareness: Recommending relevant external change events. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, pp. 465–474 (2010). DOI 10.1145/1806799.1806867
5. Holmes, R., Walker, R.J.: Systematizing pragmatic software reuse. ACM Transactions on Software Engineering and Methodology **21**(4), 20:1–20:44 (2012). DOI 10.1145/2377656.2377657
6. Holmes, R., Walker, R.J., Murphy, G.C.: Approximate structural context matching: An approach to recommend relevant examples. IEEE Transactions on Software Engineering **32**(12), 952–970 (2006). DOI 10.1109/TSE.2006.117
7. Høst, E.W., Østvold, B.M.: Debugging method names. In: Proceedings of the European Conference on Object-Oriented Programming, *Lecture Notes in Computer Science*, vol. 5653, pp. 294–317. Springer (2009). DOI 10.1007/978-3-642-03013-0_14
8. Muşlu, K., Brun, Y., Holmes, R., Ernst, M.D., Notkin, D.: Speculative analysis of integrated development environment recommendations. In: Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (2012). DOI 10.1145/2384616.2384665

9. Palma, F., Farzin, H., Guéhéneuc, Y.G., Moha, N.: Recommendation system for design patterns in software development: An [sic] DPR overview. In: Proceedings of the International Workshop on Recommendation Systems for Softare Engineering (2012). DOI 10.1109/RSSE.2012.6233399
10. Robillard, M., Walker, R., Zimmermann, T.: Recommendation systems for software engineering. IEEE Software **27**(4), 80–86 (2010). DOI 10.1109/MS.2009.161
11. Robillard, M.P.: Topology analysis of software dependencies. ACM Transactions on Software Engineering and Methodology **17**(4), 18:1–18:36 (2008). DOI 10.1145/13487689.13487691
12. Romero-Mariona, J., Ziv, H., Richardson, D.J.: SRRS: A recommendation system for security requirements. In: Proceedings of the International Workshop on Recommendation Systems for Softare Engineering (2008). DOI 10.1145/1454247.1454266
13. Tan, P.N., Steinbach, M., Kumar, V.: Introduction to Data Mining. Addison-Wesley (2005)
14. Zimmermann, T., Weißgerber, P., Diehl, S., Zeller, A.: Mining version histories to guide software changes. IEEE Transactions on Software Engineering **31**(6), 429–445 (2005). DOI 10.1109/TSE.2005.72