Machine Learning, 15, 137–168 (1994)

Using Neural Networks to Modularize Software

ROBERT W. SCHWANKE AND STEPHEN JOSÉ HANSON
Siemens Corporate Research, Princeton, NJ 08540

Editor: Alex Waibel

Abstract. This article describes our experience with designing and using a *module architecture assistant*, an intelligent tool to help human software architects improve the modularity of large programs. The tool models modularization as nearest-neighbor clustering and classification, and uses the model to make recommendations for improving modularity by rearranging module membership. The tool learns similarity judgments that match those of the human architect by performing back propagation on a specialized neural network. The tool's classifier outperformed other classifiers, both in learning and generalization, on a modest but realistic data set. The architecture assistant significantly improved its performance during a field trial on a larger data set, through a combination of learning and knowledge acquisition.

Keywords. neural networks, software modularization, similarity classification.

1. Introduction

1.1. The cognitive task of programming

Software engineers today face a formidable cognitive challenge: understanding the interactions among thousands of procedures, variables, data types, macros, and files. Most software engineers work on large, long-lived programs. Consequently, they spend more of their time modifying existing code than they do creating new code. The engineer frequently must read and understand parts of the program that he did not write, or that he wrote months or years ago and no longer recognizes. Any documentation he might have available is almost certainly obsolete. The original designers of the system have probably moved on to new projects, or even new employers. Thus, he is left with only the code itself to give him the information he needs.

Most significant commercial software systems comprise more than 100,000 lines of code—1600 pages, thicker than a James Michener novel. Fortunately, the code is typically organized into modules,[1] so that the programmer can deal with it in larger chunks. Even so, a large system is likely to comprise more than 10,000 procedures, variables, types, macros, etc. (hereafter called software units, or units), in more than 100 modules, and is likely to involve five or more programmers. Furthermore, the system is likely to be changing rapidly, with new major releases coming out every year, each with one quarter or more of the code different from the previous release. With rapid change comes architectural drift, as each change moves the structure of the system away from its original design. To compound his woes, the programmer may be responsible for working on several different system versions simultaneously, so he must remember how the interactions among components differ from one version to another.

The goal of the current research is to help rescue engineers from the nightmare of incomprehensible code by providing them with intelligent tools for analyzing the system, reorganizing it, documenting the new structure, and monitoring compliance with it, so that significant structural changes can be detected and evaluated early, before they become irreversible.

1.2. Why software is in modules

Recent developments in programming environments have raised questions about whether modules are as important as they once were. Cross-reference aids, "smart recompilation," and hypertext facilities, for example, treat procedures, macros, and other software units individually, practically ignoring traditional file and module boundaries. However, when programming is considered as a human cognitive activity, the importance of modules becomes clear. Reviewing this activity will also motivate the heuristic analysis and reorganization methods we are proposing.

- Modules are the building blocks of a software system's technical design. One of the goals of design is to select a set of conceptual entities that have relatively few interactions between them, so that the designers can reason about the system as a whole without much reference to the details inside individual modules.
- Modules are often used to assign technical responsibility. Each programmer on a large project becomes a specialist in certain parts of the system. Limiting the interactions between modules reduces the amount of communication needed between programmers.
- Good modularity can also limit the impact of program changes. A single conceptual change generally requires changes to several software units. For example, if every module in a system contains code that directly accesses a sorted list, changing it to a hash table will be extremely difficult. However, if other modules can access the list only by calling the "insert," "retrieve," and "remove" routines of the "SortedList" module, then only these three routines will need to be rewritten.
- Modules are the basic units of system integration and testing. Good planning depends heavily on having well-defined modules with limited dependencies on other modules, and on making sure that the dependencies do not change much between writing the plan and starting integration.

A recent study reveals that at least half of the cost of a software system occurs after the software is first delivered to the customer. (cf. Chapin, 1988). The largest component of this cost is modifications to the software, including fixing bugs, adding new functions and services, and porting the softare to new computers, new operating systems, and new user interface systems. Furthermore, the programmers who make these modifications spend most of their time, not in making the changes, but in *understanding* the code that is related to the changes.

In summary, the choice of modules for organizing a large software system affects understandability, division of labor, modifiability, integratability, and testability. Of these, understandability has the largest impact on the success of a software project. Therefore,

modules are important for their role as conceptually coherent chunks of software, and improving coherence through machine-assisted reorganization is an appropriate goal.

1.3. Overview of the article

The current research is intended to form the basis for a heuristic *module architecture advisor*, which recommends organizational changes that would improve the *information-hiding* quality of the modules in a software system. This article models modularization as a categorization activity requiring similarity judgment. Similarity between software units is computed by a function of their common and distinctive features, which is fitted to training data by a neural network. Categorization is accomplished by a nearest-neighbor classifier. The model is examined by embedding it in a software classification tool and several interactive clustering tools, which make reclassification and clustering recommendations, respectively. The tools incorporate a learning component, which responds to rejected recommendations by using a neural network to adjust feature weights as necessary to make the classifier agree with the category assignments given explicitly by the user. The learner transforms the user's category assignments into more-similar-than judgments, "S is more similar to G than S is to B," selecting triples <S, G, B> such that a similarity function whose values minimize errors on those judgments also maximizes the classifier's accuracy on the given category assignments. The tool then learns an ordinal similarity function that optimally fits the more-similar-than judgments.

Learning is carried out through a special-purpose back-propagation neural network. The network directly compares the value of the similarity function computed on two pairs of inputs (<S, G> and <S, B>), and back-propagates error to increase similarity on the first pair while decreasing it on the second. The features of <S, G> and <S, B> are preprocessed and presented to the network as common and distinctive features. The similarity function computed by the network is constrained to compute a ratio of common to distinctive features, in keeping with accepted models of human similarity judgment.

The classifier-with-learner thus constructed compares favorably to more traditional category learning methods. It has also been installed in a module architecture advisor, and used successfully on several real software reorganization tasks.

We conclude from these experiences that modeling software modularization as nearest-neighbor classification, with a similarity function based on accepted models of human similarity judgment, is a viable basis for the design of a module architecture advisor. The learning method used would be useful for a wide range of applications involving nearest-neighbor classifiers. The module architecture advisor illustrates a promising approach for designing "intelligent assistants" for expert tasks.

2. The information-hiding principle

One of the earliest and most influential writers on the subject of modularity is David L. Parnas. In 1971,[2] he wrote of the information distribution aspects of software design (emphasis his),

> *The connections between modules are the assumptions which the modules make about each other.* In most systems we find that these connections are much more extensive than the calling sequences and control block formats usually shown in system structure descriptions (Parnas, 1972).

The same year he formulated the *information-hiding* criterion, advocating that a module should be

> . . .characterized by a design decision which it hides from all others. Its interface or definition [is] chosen to reveal as little as possible about its inner workings (Parnas, 1971).

According to Parnas, the design decisions to hide are those that are most likely to change later on. Good examples are

- data formats,
- user interface details,
- hardware (processor, peripheral devices)
- operating system.

In practice, the information-hiding principle works in the following way. First, the designers identify the role or service that the module will provide to the rest of the system. At the same time, they identify the design decisions that will be hidden inside the module. For example, the module might provide an associative memory for use by higher-level modules and conceal whether the memory is unsorted or sorted, whether it is all in fast memory or partly on disk, and whether it uses assembly code to achieve extra-fast key hashing.

The module description is then refined into a set of procedure and data types that other modules may use when interacting with the memory. For example, the memory module might provide operations to insert, retrieve, modify, and remove records. These four operations would need parameters specifying records and keys, and some way to determine when the memory is full. The module would declare and make public the data types "Key" and "Record," and the procedures "Insert," "Retrieve," "Modify," and "Remove."

Next, the associative memory module is implemented as a set of procedures, types, variables, and macros that together make, for example, a large in-core hash table. The implementation can involve additional procedures and types beyond the ones specified in the interface; only the units belonging to that module are permitted to use these "private" units. Thus, the information that the memory is implemented as a hash table is concealed from other modules. They cannot, for example, determine which order the records are stored in, because they cannot use the name of the table of records in their procedures. Later, if the implementor should decide to replace the hashing algorithm, or even to use a sorted tree, all the code that he would need to change would be in the associative memory module.

This example shows that many design decisions are represented by software unit declarations, such as

```
HashRecord array HashTable[TableSize]
```

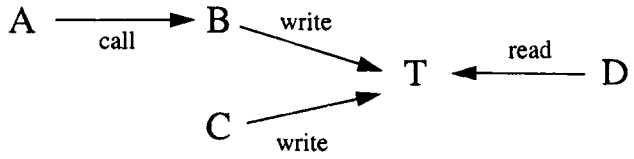
which embodies the decision to store hash records in a fixed-size table rather than, say, a linked list or tree. In most cases, procedures that depend on the design decision will use the name of the corresponding software unit, such as

```
procedure Retrieve(KeyWanted: Key)
        Index = Hash(KeyWanted)
        if HashTable[Index].Key equals KeyWanted
            return HashTable.Record
        else return FAILURE
```

This correspondence implies that

> If two units use several of the same unit-names, they are likely to be sharing significant design information, and are good candidates for placing in the same module.

A unique aspect of our research is that we measure design coupling, rather than data or control coupling. A simple example will illustrate the difference. The diagram below illustrates four procedures (*A*, *B*, *C*, and *D*) and table *T*. Procedure *A* calls procedure *B* to write information into table *T*. Procedure *D* reads information from the table. Procedure *C* also writes information into table *T*. Procedures *A* and *B* have a *control link* between them, because *A* calls *B*. Procedures *B* and *D* have a *data link* between them, because data pass from *B* to *D* through the table. Likewise, *A* and *B* are data-linked through parameters, and *C* and *D* are data-linked through *T*. However, *B* and *C* are *not* data-linked, because both of them put data into *T*, but neither one takes data out. Finally, *B*, *C*, and *D* have a *design link* among them, because all three share assumptions about the format and interpretation of table *T*. If one of the procedures ever needs to be rewritten in a way that affects the table *T*, the other two should be examined to see if they require analogous changes.



Before Parnas's work, it was commonplace to group units into modules based on control links, leaving large numbers of design dependencies between modules. Nowadays, programmers generally agree that it is more important to group together procedures that share data and type information than to group procedures that call one another.

It would be nice if the clear, simple concepts contained in a system's original design could be directly mapped into an appropriate set of implementation modules, and the mapping preserved throughout the system's lifetime. However, the implementation process always uncovers technical problems that were not apparent during the early design process, leading to changes in the design. Furthermore, design decisions are almost never so clearly separable that they can be neatly divided into subsystems and sub-subsystems. Each decision interlocks

with other decisions, so that inevitably there are some design decisions that cannot be concealed within modules, even though they are likely to change. Conversely, a module may span several loosely related decisions. In addition, there are often managerial and other non-technical influences on how a system is modularized. In the final analysis, good modularity is highly subjective.

3. A model for human software classification

We observe that programmers modularize software in much the same way that humans generally classify objects. Specifically, modules are used analogously to categories. The software units contained in a module are instances, or exemplars, of the category. The unit names appearing in an instance are its boolean-valued features. Two units can be compared by looking at their shared and distinctive features. Programmers often decide whether two units belong in the same module by such comparisons. For example, when writing a new procedure, a programmer will normally place it in the same module as other procedures that use some of the same data types and data structures in the same way. (Unlike some domains, such as thyroid assay interpretation (cf. Horn et al., 1985), we make a sharp distinction between shared "true" features and shared "false" features.)

Modules must be described by exemplars because there are many cases in which a well-designed, useful module contains two units (instances) that do not share any features. Therefore, there can be no necessary-and-sufficient feature list to describe the category. Feature diversity is intrinsic in the problem, because the information-hiding principle implies that there should be very few widely used unit names, and therefore very few features that are common to large numbers of instances. Nonetheless, a module is often designed to surround several related data structures and other private software units. Some procedures access only one or two of these data structures, while others may access all of them. Consequently, many modules contain no single "typical" member, although in some cases two or three procedures together represent the principal types of procedures contained in the category.

These observations are consistent with the literature of human classification. Humans classify things according to a few simple heuristics. First, people tend not to behave as if categories are defined by necessary and sufficient conditions; rather, they treat them as probabilistic (cf. Smith & Medin, 1981), or, more generally, as if they possess a feature "polymorphy" (cf. Hanson & Bauer, 1989). Much of the natural world promotes this view: cups, chairs, birds and so forth are labeled as such because they possess smaller feature variance within each category than between categories. Cups are cups because they possess more "cupness" than, say, "bowlness." Consequently, categorization of like objects arises partly as a contrast between clusters of objects. Another important heuristic about human classification is that not all exemplars within a category have equal status—categories are not equivalence classes. Some members of a category are better representatives of the whole category; some are more typical or more central to the "definition" of the category (Posner & Keele, 1968; Homa, 1978). And finally, humans tend to use multiple strategies when classifying, depending on the frequency of the candidate and its closeness to the most typical case. Categories can be extended either by comparisons to an aggregate pattern, prototype, or "average" or by nearest match to an exemplar (Medin & Schaffer, 1978; Homa, 1978).

To turn this qualitative model of modularization into an operational one, we require

- a way to describe software units as sets of features,
- a way to measure similarity in terms of those features, and
- a classification rule based on that similarity measure.

3.1. Software implementation features

The information-hiding principle led us to the observation that the names used in a program unit are good clues about the design assumptions on which its implementation depends, and therefore are good indicators of which module it belongs in. For similar reasons not elaborated here, the names of other units in which a unit is used are also good clues about where it belongs, although the correlation is not as strong. Therefore, the names a unit uses, and the names of places where a unit is used, are appropriate features representing the design characteristics of that unit.[3]

This information can be extracted from the code itself. First, a conventional cross-reference extractor analyzes the code to produce a relation, $NamesUsed \subset Unit \times Unit$, where $\langle x, y \rangle \in NamesUsed$ if and only if the name of unit y is used in unit x. Then,

$$UserNames = \{\langle y, \hat{x} \rangle\} \text{ such that } \langle x, y \rangle \in NamesUsed, \text{ and}$$

$$HasFeatures = NamesUsed \cup UserNames.$$

Notice that *UserNames* introduces the notation $\hat{x}$, denoting a synthetic name derived from x. This represents the difference between a name used in a unit and the name of a place where the unit is used. The distinction is made so that, when *HasFeatures* is computed, $\langle y, \hat{x} \rangle$ and $\langle y, x \rangle$ are distinct tuples.

Experience has shown the importance of obtaining cross-reference information that is as fine-grained as possible, e.g., the individual field names of structures and the individual literals of enumeration types. Such details are what distinguish code that implements an abstract data type from code that merely uses it.

Although cross-reference analysis generates a rich set of implementation features, some important design decisions do not correspond to any particular identifier. Therefore, the relation *HasFeatures* may be expanded with tuples supplied by the human architect.

3.2. Measuring similarity

In order to design an appropriate similarity function, we first describe several important properties that the function should satisfy, and then introduce a function that satisfies them.

3.2.1. Matching and monotonicity

When a programmer judges the similarity of two procedures, she looks both at the features they have in common and the features that are distinctive to one or the other procedure.

Adding a common feature increases similarity; adding a distinctive feature decreases it. She also judges the relative importance of different features. A feature representing a localized, volatile design decision deserves a greater weight than a feature representing a widely used, stable design decision. One type of feature she does not look at is one that is absent from both procedures. Identifiers that do not occur in either procedure have no impact on their similarity; they are simply irrelevant.

These properties of software similarity judgment correspond to general models of human similarity judgment, such as proposed by Tversky (1977). Tversky's model treats object descriptions as sets of features, and similarity functions as functions of common and distinctive features, defined as follows.

Let $A, B, C, \ldots$ be objects described by sets of features $a, b, c, \ldots$, respectively. When comparing two objects, the following computed feature sets are significant:

$a \cap b$ The set of features that are *common* to A and B.

$a - b, b - a$ The sets of features that are *distinctive* to A or B, respectively.

The matching property restricts similarity functions to those that are functions of the common and distinctive features, and that are independent of the features that neither object has. (The property is apparently so named because it matches up features in the two sets.) A similarity function, *SIM*, has the *matching* property if there exists a function F such that

$$SIM(X, Y) = F(x \cap y, x - y, y - x)$$

The monotonicity property embodies the idea that similarity should increase in proportion to common features and decrease in proportion to distinctive features. A similarity function, *SIM*, has the *monotonicity* property if $SIM(A, B) \geq SIM(A, C)$ whenever

$$a \cap b \supseteq a \cap c$$

$$a - c \supseteq a - b$$

$$c - a \supseteq c - b$$

and, furthermore, the inequality is strict whenever at least one of the set inclusions is proper.

3.2.2. Other desirable properties

The software domain suggests some additional characteristics that a similarity function should have:

- No maximum value. Two identical procedures with many features should be more similar than two identical procedures with few features.

- A minimum value, obtained when there are no common features. Two unrelated procedures should be just as dissimilar as two other unrelated procedures.
- The function should be defined when there are no common and no distinctive features. This surprising requirement arises because real-world software sometimes contains "stub" procedures that have no bodies and no callers, and hence no features.

3.2.3. A ratio model of similarity

We have designed a matching, monotonic similarity function with the above properties, called *Ratio*, similar to one proposed by Tversky, of the same name. We define

$$Ratio(S, N) = \frac{Weight(s \cap n)}{1 + c \cdot Weight(s \cap n) + d \cdot (Weight(s - n) + Weight(n - s))}$$

This function satisfies the matching and monotonicity properties as long as c and d are positive, and *Weight* increases monotonically with the set membership of its argument. This requirement is satisfied by defining

$$Weight(X) = \sum_{x \in X} w_x, \text{ where } w_x > 0.$$

Weight computes the combined significance of a set of features. Although Tversky defined the function to be linear, we admit the possibility that it might be non-linear, representing correlations among features.

Ratio satisfies the requirements of software similarity mesurement described above. It is also symmetric, which is not necessary to model human similarity judgment, but makes it possible to use the function in standard clustering algorithms.

There still remains the problem of how to assign weights to the features, and values to c and d. Giving all features the same weight causes high-frequency features to dominate clustering performance at the expense of rare features. Intuitively, feature weight should vary inversely with its frequency, since rarely occurring features have a better chance of being encapsulated within a module. Therefore, we have been estimating the significance of a feature by its Shannon information content, $w_f = -\log P(f)$, where $P(f)$ is the probability that a unit in the system being studied has feature f. In a later section we will describe how to learn better estimates of these weights.

3.3. A nearest-neighbor classification rule

Because programmers assign procedures to modules by finding a small group of highly similar procedures, modularization can be modeled by a nearest-neighbor classification rule. To describe the rule, we use the following definitions:

Subject. A unit being considered for inclusion in a category.

Good Neighbor. A unit belonging to the category for which the subject is being considered.

Bad Neighbor. A unit belonging to any category other than the one for which the subject is being considered.

k. A parameter of the classification rule, denoting the minimum group size to which a subject might be added.

Since more than one module may be a good candidate to receive a given subject, the classification rule below incorporates a *confidence measure* for each of the possible categories:

Confidence. Subject S fits in category X with confidence C if and only if assigning S to X would result in S having exactly C bad neighbors more similar to it than its k'th good neighbor.

Note that C is zero when S's k nearest neighbors are all members of category X. Greater values of C imply that the immediate neighborhood of S is "polluted" with units from other categories.[4] With the confidence measure so defined, the classification rule is straightforward:

Classification Rule. A software unit belongs in the category for which its confidence rating is best (closest to zero).

Confidence ratings also provide a sensitive way to measure how well a classifier implementing this rule conforms to a given set of classification data. If performance were measured only in terms of classification errors, the classifier might compute the correct categories, but with marginal confidence. Therefore, it is useful to measure a classifier's performance in relation to its confidence that the subjects are correctly classified as labeled in the data. However, such a measure would still be sensitive to the cluster size parameter, k. Experience has shown that setting k equal to *1* causes problems when some of the data are mislabeled. If just one unit is mislabeled, any unit for which it is the nearest neighbor will appear to be misclassified. Similarly, any particular value for k may be inappropriate for a specific unit, because highly cohesive software clusters occur in many sizes. Therefore, performance is actually measured over a range of values of k, as follows:

Classifier Performance Measure. A nearest-neighbor classifier conforms to a data set D with rating R,

$$R = \sum_{\langle S,C\rangle \epsilon D} \sum_{k=1}^{K} \mathit{Confidence}(S,\ C,\ k)$$

where $\langle S,\ C\rangle$ denotes unit S assigned to cluster C. Typical values of K range from *2* to *5*.

4. Learning architectural judgment

The classification rule described in the previous section has been installed in a module architecture advisor and used profitably to help reorganize real software systems (including

the tool itself). These experiences will be discussed in section 5. However, the classifier's accuracy has been limited by the arbitrary way that values are assigned to the constants and feature weights in the similarity function. "Hand tuning" those values has been tedious and unenlightening, although doing so does improve performance.

The goal of the present research is to replace hand tuning with an automatic tuning process, in which the tool learns from its mistakes. To achieve this goal, the learning task is divided into the following steps:

1. Advise the architect until the architect disagrees with the advice (indicating that learning is needed).
2. Identify a subset of the units that the architect believes are correctly classified, for use as a training set.
3. Construct a set of more-similar-than judgments that if correctly modeled by the similarity function would produce perfect classifier performance and confidence on the training data.
4. Train the similarity function, by back propagation, to maximally fit the more-similar-than judgments.
5. Ask the architect for additional features to explain any category assignments that the classifier has not learned.

The rest of this section assumes that the advisor can extract a training set from the current tool state. It describes how more-similar-than judgments are constructed, and describes the back propagation network used to train the similarity function. Training data selection and feature acquisition from the architect are deferred to section 5.

4.1. Constructing more-similar-than judgments

From the definition of the classifier performance measure, one can see that only similarity judgments between a subject and its near neighbors are relevant to classifier performance. In particular, *Confidence*(S, C, k) is proportional to the number of cases in which S is more similar to one of its bad neighbors than to its k-th nearest good neighbor. Aggregating over all values of S and k, classifier performance is equal to the number of cases, $\langle S, G, B \rangle$, for which a subject, S, is more similar to a bad neighbor, B, than to one of its K nearest good neighbors, G. Therefore, only a subject's K nearest good neighbors are relevant to it. The relevant bad neighbors are those more similar to the subject than its K'th nearest good neighbor. Therefore, the more-similar-than judgments that should be learned are all possible combinations of a subject and its relevant good and bad neighbors. Optimizing these judgments optimizes classifier performance according to the specified measure.

The optimization process brings in one additional problem: initialization. Since the goal is to learn a similarity function, and training data are selected using that same function, one must assume that, initially, the estimated similarity function is a poor predictor of actual similarity. Therefore, limiting consideration to the "K'th good neighbor" -hood might arbitrarily screen out units that are actually highly similar to the subject.

This problem is solved in the present research by setting K very large at the beginning of training, while the similarity function is poorly estimated, and gradually decreasing it as learning progresses. Initially, all weights and constants are drawn from random distributions, and K is set greater than the size of the largest module. After each training epoch, K is reduced, by exponential decay, toward a predetermined asymptotic value and the triples $\langle S, G, B \rangle$ are reconstructed.

4.2. Backpropagating similarity judgment errors

Similarity judgment is learned by computing more-similar-than judgments in a feedforward neural network, and backpropagating errors. However, rather than being a general hidden-layer network, the network is designed to mirror the model of similarity judgment discussed above. It is described here mathematically first, and then as a network.

4.2.1. Inputs

For each triple $\langle S, G, B \rangle$, the inputs to the network are the corresponding feature sets s, g, and b.

4.2.2. Error function

The network computes the sigmoid function of the difference of two similarities:

$$Error(s, g, b) = (\sigma(Sim(s, g) - Sim(s, b)) - threshold)^2$$

where the threshold is typically 0.95, and

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

The value of *Error* will be near zero whenever S is much more similar to G than to B, and near $threshold^2$ when the opposite is true.

4.2.3. Similarity function

The similarity function is defined as described in section 2.3:

$$Sim(s, n) = \frac{Weight(s \cap n)}{1 + c \cdot Weight(s \cap n) + d \cdot (Weights(s - n) + Weight(n - s))}$$

where $Weight(X) = \sum_{x \in X} w_x$, and $w_x > 0$.

4.2.4. Implementation as a network

Figure 1 shows the topology of the network. The Focuser uses the current estimate of the similarity function to select a set of input triples for one training epoch. Each triple is presented to the network as a triple of feature vectors. A preprocessing stage computes the needed sets of common and distinctive features. The next stage computes *Weight* for all six sets, using the same feature weights each time. These six *Weights* are fed into two identical sub-networks, which compute *Ratio* on $\langle s, g \rangle$ and $\langle s, b \rangle$, using the same values for c and d each time. Finally, the comparison stage of the network computes the sigmoid function of the difference of the two similarities. Back propagation is carried out with a simple delta rule.

4.2.5. Novel aspects of the network

The most significant aspect of the network design is that it learns a similarity function by comparing the value computed by the function on two related pairs of inputs. This works

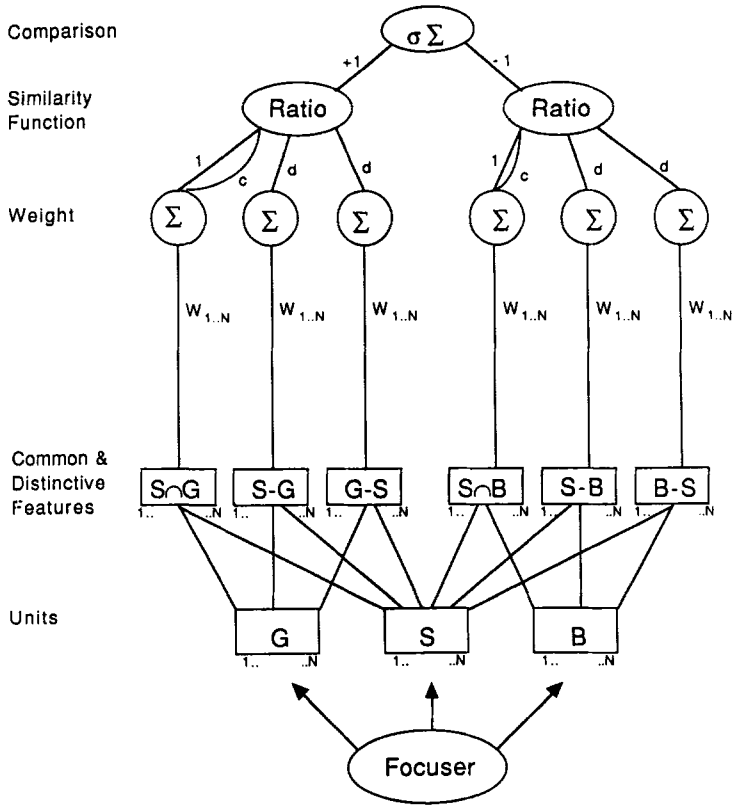


Figure 1. Comparison network for learning relative similarity judgments.

because the absolute value computed by the function is unimportant; only the relative order of the values it computes matters. Therefore, instead of training the network to compute a specified value, it must compute two values with a specified relative order. The implementation method was suggested by Tesauro, who had invented it to teach his Neurogammon system (Tesauro & Sejnowski, 1989) an ordinal move evaluation function by training it on pairs of alternative moves from the same board position and dice roll, one of which was known to be an expert's choice. He calls the approach the *comparison paradigm.*

To implement the comparison paradigm, the network uses the same set of link weights for both computations of *Sim*. When error is propagated back, each weight receives error assignments for its roles in both computations. The order of the inputs does not need to be randomized, and no training is signal needed, because the symmetric network design prevents the weights from learning a bias toward the "left" or "right" neighbor. The "training signal" is always the same: S should be more similar to G than to B.

By implementing the ratio formula directly, the network restricts the class of similarity functions to a subset of the monotonic, matching functions. This restriction was motivated by the problem domain and has proven to be acceptable in practice. However, further research could examine other monotonic, matching functions.

The network also specifies a linear *Weight* function. Nonlinear, but monotonic, functions could be substituted, if necessary to model feature correlations, but experience has shown the linear function to be satisfactory.

Two other implementation details are worth mentioning. The weights are bounded greater than zero, so that the similarity function remains monotonic. When the weights are updated during backpropagation, any weights that drop below a small positive threshold are reset to that threshold. The other detail is that the common and distinctive feature sets are represented as feature vectors, with 1's for set members and 0's for non-members. Zero has the special property that back propagation will assign no portion of the error to a link from a zero input, meaning that error can only be assigned to weights for features that were present in at least one of the input units. This reinforces our working hypothesis that similarity is unrelated to absent features.

4.3. Learning and generalization performance

To obtain a preliminary estimate of the tool's capabilities, we applied it to a small but realistic data set. The sample data come from a real software system, which is actually an early version of our batch-clustering tool. It comprises 64 procedures, grouped into seven modules. Membership in the modules is distributed as described in table 1.

The software is written in the C programming language. To create the sample data set, we applied a cross-reference analysis tool to it, collecting every occurrence of a non-local identifier, including procedures, variables, macros, typedefs, and the individual field names of structured data types. Each such name was given a unique identification number, so that there would be no confusion when, for example, two different record types have a field with the same name.

Table 1. Module sizes

Members	Module
10	attr
12	hac
5	massage
7	node
4	objects
12	outputmgt
14	simwgts

Each distinct non-local name occurring within a procedure was then recorded as a feature of that procedure. In those cases where one procedure called another, two features were recorded: the callee's name became a feature of the caller, and the caller's name became a feature of the callee, but marked to distinguish "called by X" from "calls X."

This process produced 152 distinct feature names. However, many of these features occurred in only one procedure each, and were therefore greatly increasing the size of the problem without contributing to the similarity of two procedures. Therefore, we eliminated all such singly occurring features, leaving 95.

We expected the given data to contain classification errors, because the software had not been carefully modularized. However, we wanted to measure generalization performance on "clean" data, so that generalization errors would not be confused with training data errors.

Therefore, we established two criteria for eliminating units from the data set, both of which had to be met before the unit was eliminated:

1. The classifier must fail to classify the unit correctly, even after learning.
2. The feature data must show evidence that the learning failure was due to a modularization error.

Twelve procedures met both criteria and were removed, leaving 52. The network was able to learn to classify all 52 correctly.

To test the network's generalization ability, we contructed a jackknife test, in which the 52 procedures were divided into a training set and a test set, to determine how often the similarity measure, used in a nearest-neighbor classifier, could correctly classify procedures that were not in the training data. The test consisted of 13 experiments, each using 48 procedures for training and 4 for testing, such that each procedure was used for testing exactly once. Each experiment consisted of training on triples constructed from the training set, and then using the learned similarity function to identify the nearest neighbor of each procedure in the test set. K had a final value of 3. The test procedures were classified with k equal to *1*.

The results of the jackknife test are shown in table 2. Each row gives the number of procedures that were in that module and how many of them were classified into each module during the jackknife test. Only one unit out of 52 was misclassified.

Table 2. Classifier performance on unseen units.

		Classifed as Module						
Actual Module	No.	outputmgt	simwgts	attr	hac	node	massage	objects
outputmgt	11	11						
simwgts	11		10	1				
attr	9			9				
hac	8				8			
node	7					7		
massage	4						4	
objects	2							2

From these results we conclude that the given data, the classification and similarity models, and the learning method were very well matched to one another. Naturally, it could be that the data were highly redundant, so that a jackknife test that removed only four procedures was not really withholding any knowledge. Or it could be that the process of removing errors from the data biased the results. The experiment was simply too small to tell.

4.4. Comparisons to simpler classifiers

To study the properties of software categories, we first tried clustering them in feature space. Similarity in feature space is measured by distance metrics with small distance corresponding to great similarity. Consequently, units possessing common features will be more similar than those that do not. Furthermore, those features that appear in one category but not in the other will augment similarity of units within a group. Typical kinds of similarity measures include Euclidean, Hamming, and inner product (cosine). Such measures used with agglomerative hierarchical clustering algorithms can only produce cluster groups that were originally at least linearly separable, i.e., clusters that could be separated by a line in 2-space or a hyperplane in four or higher dimensions.

We studied the category properties in feature space by attempting to cluster the example data set in feature space, using a hierarchical, agglomerative clustering algorithm under several different similarity measures.

Shown in figure 2 is a Euclidean, centroid clustering of the software units by their cross-references (see above). At the end of each leaf is a label indicating which module of a possible seven the unit is defined in. Note that separation of the category modules does not result; less than half of the group members are assigned to their proper modules. Other measures fare no better. Figure 3 and figure 4 show cluster diagrams for other similarity measures.

4.5. Comparison to neural network classifier

Neural networks represent a class of function approximating methods that can create similarity as a function of the data to which they are exposed. Within any network is a

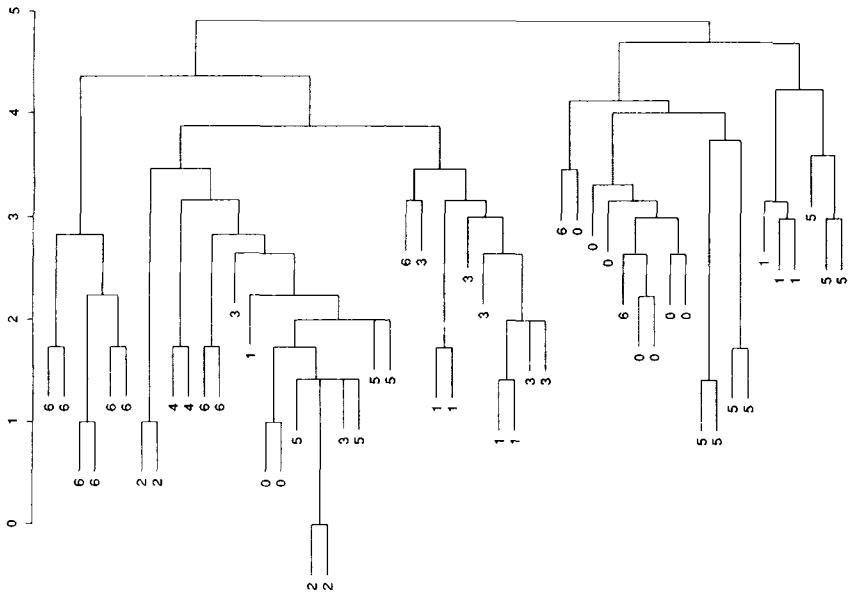


Figure 2. Euclidean, centroid clustering.

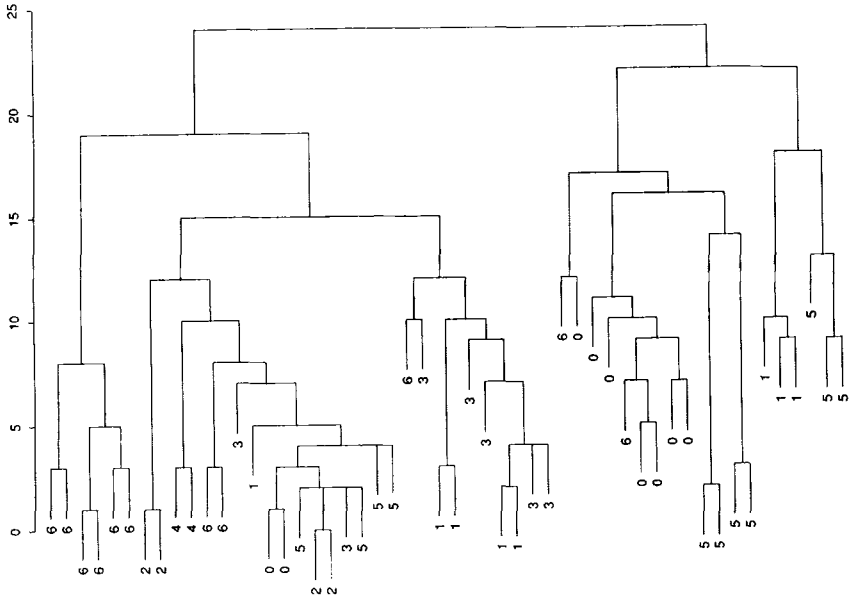


Figure 3. Hamming, centroid clustering.

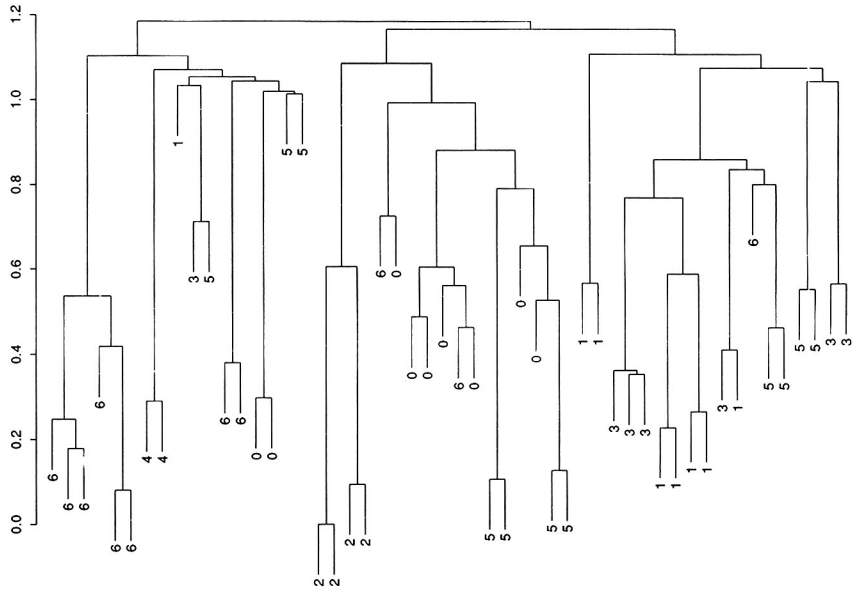


Figure 4. Hierarchical clustering with cosine correlation.

basis set of functions (see Hanson & Burr, 1990) that allow arbitrary similarity functions to result as the network learns to correctly label data points. Such supervision is also critical for category problems in which the initial feature space is required by the category labels to transform nonlinearly. Consequently, although the similarity measure may cause units with shared and distinctive features to be closer in similarity space, category labels as determined by membership in modules may require transformation of similarity by moving units that are initially far apart closer together in similarity space.

Nonetheless, having an effective similarity measure and supervision from labels still involves a complex induction problem. Methods like neural nets share with other learning approaches the problem of generalizing correctly from a limited sample of example cases. In theoretical learning work, this is currently an intense area of research with many unanswered questions (cf. Somplinsky & Tishby, 1990; Baum & Haussler, 1988; Rivest, Haussler, & Warmuth, 1989). Consequently, it is possible to learn perfectly all examples from the domain and still incorrectly classify new examples as they appear.

We found that this phenomenon was indeed present in our sample data. We used a simple feed-forward, back-propagation network with one hidden layer of sigmoidal activation units, and trained it to classify the given objects. We found that, with just four hidden units, this network was able to classify the training data perfectly. However, this perfect learning prevented it from classifying any novel units correctly. These results are shown in figure 5.

We believe that one of the reasons neural network generalization performance is so poor is that the categories are sparsely populated. There are generally more relevant features than units to be classified. Each unit has typically only seven features, which it shares

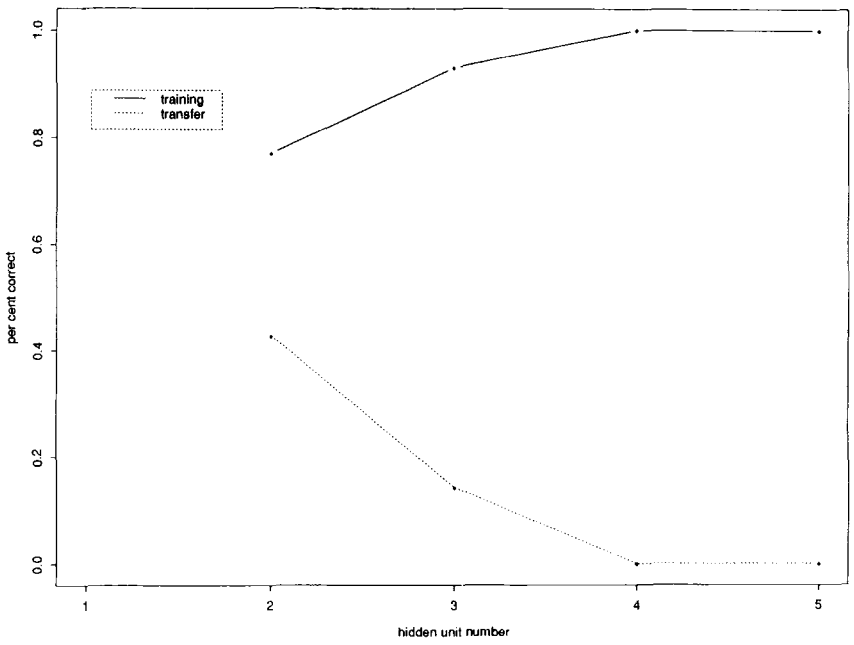


Figure 5. Training and transfer performance by number of hidden units.

with only a handful of other units. The majority of instances have no exact duplicates in the training set. Therefore, the network is not able to make generalizations about salient features, but can only memorize the category memberships.

5. Advising architects

Since preliminary experiments on small data sets showed the classification and learning methods to be promising, we proceeded to incorporate them in a heuristic architecture advisor (Schwanke & Platoff, 1989) and to try them out on real reorganization tasks. This section describes

- the working styles supported by the tool,
- how the tool uses the classification model to give advice,
- a case study showing that the advice is useful, and
- a case study showing how the quality of the advice improves with learning.

5.1. Working styles

The tool supports three different (although overlapping) styles of work:

- Incremental change: the software is already organized into high-quality modules. The engineer wishes to identify individual weak points in the architecture, and repair them by making small changes.
- Moderate reorganization: although the software is already organized into modules, their quality is suspect. The engineer wishes to reorganize the code into new modules, but with an eye to preserving whatever is still good from the old modularity.
- Radical (re)organization: Either the software has never been modularized, or the existing modules are useless. The engineer wishes to organize the software without reference to any previous organization.

All these styles can be applied to a whole system or to an individual subsystem or module. The tool supports these activities with two kinds of service: clustering and maverick analysis.

5.2. Clustering

This service organizes software units into a tree of categories. It is actually a group of clustering services, each of which interacts with the user in a different way:

Batch clustering supports radical reorganization. It uses a hierarchical, agglomerative clustering (HAC) algorithm to form a category tree. The given similarity measure is used to derive a group similarity measure. The algorithm starts by placing each unit in a group by itself. It then repeatedly combines the two most similar groups. Some variations of it heuristically eliminate useless interior nodes in the category tree, so that the tree has varying arity appropriate to the data.

Incremental clustering supports incremental to moderate reorganization. It is based on the same HAC algorithm, but allows the user to apply it to any node of his cateogry tree at any time, and to review each clustering action before it is carried out. The user selects a node in the category tree, and asks for either the nearest sibling or the two most similar children of that node. Based on the answer, he may decide to combine the indicated groups, or to make some other change in the organization. Thus clustering is carried out manually, but with advice whenever requested.

Interactive reclustering supports moderate reorganization. It starts with a given set of original categories, but tries to build a fresh classification tree out of the units in them. It uses the original category labels to decide which clustering steps should be reviewed by the user, and which can be carried out automatically, without review. It uses the hierarchical, agglomerative clustering algorithm, but before combining two groups, it checks to see whether all members of both groups were in the same original category. If so, it combines the two groups automatically. If not, it pauses and asks the user whether to combine them. To help the user decide, it presents several other relevant groups. For each of the two groups that it is recommending combining, it also presents the second-nearest neighbor as well as the nearest neighbor whose members all belonged to the same group originally. The user can then choose to combine the recommended pair, to combine some other subset of the presented groups, or to make any other organizational change he wishes. The interactive clustering algorithm then resumes its work using whatever clusters the user has formed.

Neighborhood clustering can be used as a prelude to incremental clustering. Given a set of units, this service forms the smallest clusters for which each unit is in the same cluster as its k nearest neighbors.

The batch-clustering algorithm appears not to be useful, because a mistake early in the clustering process often makes all subsequent clustering decisions wrong. Also, an architect is not likely to accept a new set of categories that is radically different from the set of modules with which he started. The interactive clustering methods are much more useful, because in each of them the architect has opportunities to override the tool's recommendations, and the tool then continues its work based on the architect's decisions. Also, the interactive tools present the architect with several good alternatives, rather than just one best choice, thus giving the architect powerful guidance without preempting his options. The neighborhood clustering algorithm is quite powerful: even specifying a neighborhood size of 1 often creates clusters with an average of four members. This means that three quarters of the clustering decisions (1 decisions combines two clusters) have already been made, leaving only one quarter to be made by other methods. However, all the reclustering methods suffer the same weakness: they cannot yet learn from their mistakes.

5.3. Maverick analysis

This service heuristically identifies software units that appear to violate the *information-hiding* principle. A unit is deemed to be in violation if it appears to share more implementation characteristics with units in other modules than with units in its own module. This violation is detected by using the classifier to compute the "correct" module assignment. If the computed assignment is different from the module in which it resides, the unit is listed as a *maverick*. (Like a stray calf on the western plains, the unit must be returned to the proper herd.) Such a misplaced unit usually indicates a conceptual weakness in the architecture. The correct repair is usually not to simply move the unit. More often, it indicates that one or more units are mixing design decisions from different modules, and that the units should be rewritten.

The maverick analyzer evaluates the category assignment of each unit. It uses the classifier both to identify the best category for the unit and to give confidence ratings for both the present and recommended category assignments. A unit is considered a maverick if it fits into another category with a better confidence rating than its rating for the category to which it is currently assigned. Because the heuristic nature of the analysis leads to a substantial number of false positives, the mavericks are presented to the architect "worst first." They are sorted in order of confidence in the recommended category (strongest first), and, among mavericks with equal reclassification confidence, in order of confidence in the current classification (weakest first).

The maverick analyzer has been used to review the organization of a modest-sized industrial-strength software system (Lange and Schwanke, 1991). The system studied contained 300 procedures, organized into 27 modules, and 900 distinct cross-reference feature names. At the time of the analysis, the programmer maintaining the code was already planning to clean up its structure, but had not yet done so.

Maverick analysis yielded 51 procedures that were apparently misclassified. An analyst unfamiliar with the code used the maverick list to uncover 24 specific ways in which the code modularity could be improved. Recommended improvements included the following:

- Move a procedure
- Repartition a set of modules
- Add methods to an abstract data type, and use them instead of accessing the representation
- Introduce an interface layer to separate low-level from high-level functionality
- Split a procedure that is straddling two modules
- Replace an erroneous variable reference with the correct one
- Remove dead code

The original maintainer reviewed each of these 24 recommendations and responded in one of four ways:

- Correct (5 cases)
- Helpful (6 cases): the problem was correctly identified, but a more appropriate repair was found
- Redundant (7 cases): the identified problem was due to one of the previous 11 cases
- Incorrect (6 cases)

To assess the potential benefit of learning from mistakes, the analyst then hand tuned the maverick analyzer to improve its performance, by adding five user-defined features, adding four syntactic features that were not derived from cross-references, changing eight feature weights, and marking two procedures as unquestionably belonging to the modules in which they resided. This reduced the maverick list to 23 procedures, including the 11 correct and helpful cases.

These results are promising, but not completely satisfactory. Although maverick analysis provided real benefit to a real project, 80% of the mavericks were useless or redundant. Although hand tuning reduced this number to 50%, the tuning process was difficult and unenlightening. Therefore, it seemed worthwhile to try an automatic tuning process, which, in combination with focused knowledge acquisition from the user, might produce a more useful tool.

5.4. Maverick analysis with learning

Although presenting the mavericks worst-first mitigates the problem of false positives, the architect must eventually review all the mavericks on the list or worry that he has overlooked a problem. What is worse, if he makes changes to the code because of the real mavericks he finds, he may have to re-review the whole maverick list to see if any of the old false positives have become real mavericks. Adding a learning capability to the tool could overcome these problems in two ways:

1. Translating the human's classifiction decisions into similarity judgments will make them applicable to other potential mavericks after the code has been changed or reorganized. Instead of re-reviewing all mavericks, the human would only have to review those that were not screened out by his previous judgments.
2. By using the architect's judgments as "relevance feedback" during an analysis session, the tool could reorder the maverick list to bring the real mavericks closer to the top of the list.

These two hypotheses have been investigated in a case study using the learning method to improve the quality of maverick analysis for a real architect reorganizing a real system. This section first describes the case study itself, then reports subsequent analsyis of data taken from the study.

5.4.1. A case study

The Arch tool was applied to the code of a real software system called TSL (Balcer, Hasling, & Ostrand, 1989). This system is in production use in Siemens operating divisions, and is undergoing active maintenance. It comprises 470 procedures in 33 modules, ranging in size from 1 to 29 procedures. Three modules with fewer than four procedures were excluded from the experiment. The architect was asked to perform four kinds of actions:

1. Move those mavericks which were actually in the wrong module.
2. Identify false-positive mavericks.
3. Identify those units for which nearest-neighbor classification was not the right model to explain his modularization decisions.
4. Supply user-defined features, as necessary, to identify shared design properties that were not represented by cross-reference features.

Time did not permit us to have the programmer actually rewrite any code.
The experimental procedure went as follows:

1. Initialize the tool's classifier with feature weights based on the information content of features, as described in section 3.3.
2. Have the architect review each of the 10 worst mavericks, either specifying the best module assignment explicitly or removing it from further analysis.
3. Move the mavericks to their new modules as specified.
4. Review the common and distinctive feature data for explicitly classified mavericks, checking for learnability.
5. Have architect supply features to explain "unlearnable" cases.
6. Extract training data from the user session log and transmit it to the learning component.
7. Train the neural network, as described in sections 4.1 through 4.2, until classification performance on the training data stops improving.
8. Transmit the learned weights back to the maverick analyzer.
9. Generate a revised maverick list.
10. If the list is non-empty, go back to step 2.

Notes:

- Step 4. Experience has shown that, for a module assignment to be learnable, there usually must be at least one feature that the unit shares with some good neighbors that it does not share with its nearest bad neighbors.
- Step 5. In principle, we could have waited until the tool failed to learn the correct classification for the unit before asking for a user-defined feature, but time constraints prevented us from waiting for this verification. From the user's point of view, supplying the feature provides useful documentation anyway, so it is not an unreasonable procedure. This step is actually a carefully focused knowledge acquisition procedure. The architect is asked to focus her attention on just the difficult situations, and to supply just enough information to explain them. This strategy is far more desirable for the architect than supplying large amounts of knowledge a priori, because she can see the immediate benefit of supplying the knowledge: it corrects the tool's mistake.
- Step 6. To extract training data without burdening the architect unduly, most of the training set should be identified automatically. Therefore, the training set was constructed by collecting all the units that appeared to be correctly classified already, with strong confidence, before the first user session began, and adding to them all the mavericks that the architect had reviewed and explicitly classified. By this procedure we hoped to avoid putting "false negatives" in the training data, but this assumption was never directly verified.

The architect required 10 cycles through the experimental procedure before all the mavericks had been examined. His actions on each cycle are summarized in table 3.

Table 3. Architect's actions during maverick analysis.

Session	Mavericks	False	Move	Remove	Repeat	User Features	Checked
1	125	5	5				10
2	92	6	4				10
3	82	5+6	3	1	1	2	16
4	71	6+3	1+2				15
5	65	6+5	3	1+1		6	16
6	56	7+9		1	1	4	18
7	45	6+5		1		6	12
8	27	5+1		5+3		3	14
9	12	6		4	1	2	10
10	3	1			2	1	1
Totals	125	53+29	16+2	15+4	5	24	114

Session: During each session, the architect set out to check the ten worst mavericks. The actual count was sometimes more and sometimes less.
Mavericks: The number of mavericks on the maverick list for each session.
False: The number of mavericks that the architect indicated were false positives.
+6: Sometimes the architect explicitly labeled procedures other than the top ten mavericks, such as when he was providing user-defined features for the maverick and the other members of its cluster. The number of non-top-ten mavericks so labeled is shown as "+digit."
Move: The number of mavericks that the architect reassigned to a new module.
Remove: The number of mavericks that the architect indicated were not appropriate for maverick analysis.
Repeat: The number of mavericks that showed up in the top ten after having been checked previously.
User features: The number of user-defined features added to explain "unlearnable" cases.
Checked: Total number of False, Moved, Removed, and Repeated mavericks.

Several observations about the study are worth mentioning:

- All of the Moved mavericks were found in the first five sessions. The last five sessions served only to teach the tool not to report the false positives. A real user, after finding no more real mavericks in the top ten, might well choose to stop analysis, satisfied that he has found nearly all of the problem procedures.
- In the first five sessions, all of the top-ten mavericks had a reclassification confidence of 2 or less. All of the Moved mavericks had a reclassification confidence of 0 or 1.
- User-defined features were not needed until session 3.
- Nineteen of the mavericks were judged to be procedures for which the nearest-neighbor classification heuristic seemed inappropriate. For example, some of the cases involved one-of-a-kind procedures. There were another 19 procedures that were excluded by very simple heuristics, such as those belonging to modules with three or fewer members, belonging to an imported module, or with fewer than three good or bad neighbors having at least one common feature.
- Eight of the 24 mavericks requiring user-defined features were type-destructor procedures that shared more implementation information with one another than they did with other procedures defined on their own type. Possibly these eight should have been excluded rather than kept and given an extra feature.
- One hundred and fourteen of the 125 mavericks had to be checked explicitly before the tool learned to completely agree with the architect. This seems to indicate that very little generalization was taking place.
- Despite these concerns, the tool did eventually learn to incorporate all the architect's classification judgments.

5.4.2. Analyzing generalization performance

The fact that 114 mavericks had to be checked explicitly before perfect learning was achieved seemed to be due to a very liberal definition of *maverick*. In order to minimize the risk of false negatives, a large number of false-positive mavericks were occurring. Therefore, to measure learning performance, we decided to treat the maverick list like the result of an information retrieval (IR) query, measuring its performance on a precision/recall graph.

When an IR system produces a set of documents that approximately match a query, some of the retrieved documents are likely to be irrelevant, as judged by the end user. Since retrieval is based on the similarity between the query and each of the documents in the collection, the retrieved list is typically sorted in order of decreasing similarity, since the most-similar documents are presumed to be most likely to be relevant. Depending on the stamina of the end user, she may look at only the first few entries, or half the list, or the whole list.

Information scientists have defined two standard concepts, precision and recall, to measure the quality of a retrieved set of documents. Precision is the fraction of documents in the retrieved set that are relevant. Recall is the fraction of relevant documents that are in the retrieved set. When the retrieved set of documents is ordered by their estimated likelihood

of relevance, that ordering can be evaluated by measuring precision and recall for successively longer prefixes of the list. Specifically, data are collected for each prefix of the list that ends with a relevant document. Precision and recall are plotted for each such prefix in a precision/recall graph. Perfect performance, where all the relevant documents were recalled and bunched at the top of the list, would produce a horizontal line at $Y = 1.0$. Random performance, where all relevant documents were recalled but uniformly distributed throughout the list, would produce a horizontal line at $Y = relevant/(relevant + nonrelevant)$.

Two methods of estimating relevance can be compared by comparing their precision/recall graphs. However, precision and recall depend strongly on both the document collection and the specific query, so comparisons must use the same collections and queries to evaluate both methods. Normally, precision and recall are averaged over a large number of queries. However, it is also valid to compare the performance of two methods on a single query, as long as one does not generalize too much from a single example.

Precision/recall measurement can be used to measure the quality of maverick analysis by treating the maverick list as a sorted list of retrieved documents, sorted by estimated relevance. Relevance is estimated by reclassification confidence and lack of confidence in the current classification. We will compare two sets of parameters for the similarity function by comparing the maverick lists they generate for the same set of data.

Various small procedural changes during the course of the case study prevented us from performing meaningful analysis directly on the experimental protocol. However, the protocol did produce a complete list of relevant documents, allowing us to analyze precision and recall with and without learning.

First, figure 6 shows why learning was needed. Out of 125 mavericks, only 16 were actually relevant. Although precision was relatively high near the top of the list, beyond the first five real mavericks, precision dropped off rapidly.

In the first iteration of the experiment, the architect's analyses of the 10 worst mavericks were used to learn new feature weights. To compare performance with and without learning, we performed the specified module reassignments, removed the original top 10 from further consideration as mavericks, and compared the maverick lists constructed with and without the learned feature weights. The results are plotted in figure 7. Here we see that, without learning, precision is low at all recall levels, not much better than "random." Precision with learning is better in all cases, and 3 to 5 times better for recall levels up to 0.63.

To compare the contribution of feature weights versus gross coefficient values in the similarity function, we tried varying the gross coefficients, both with and without learned feature weights. We found the variation in performance due to the gross coefficients to be negligible compared to the variation due to learning feature weights.

6. Discussion

6.1. Modeling modularization

The case studies reported here support the following hypotheses:

- Nearest-neighbor classification, with similarity measured based on common and distinctive features, is an effective model for a large fraction of the modularization decisions in software systems.

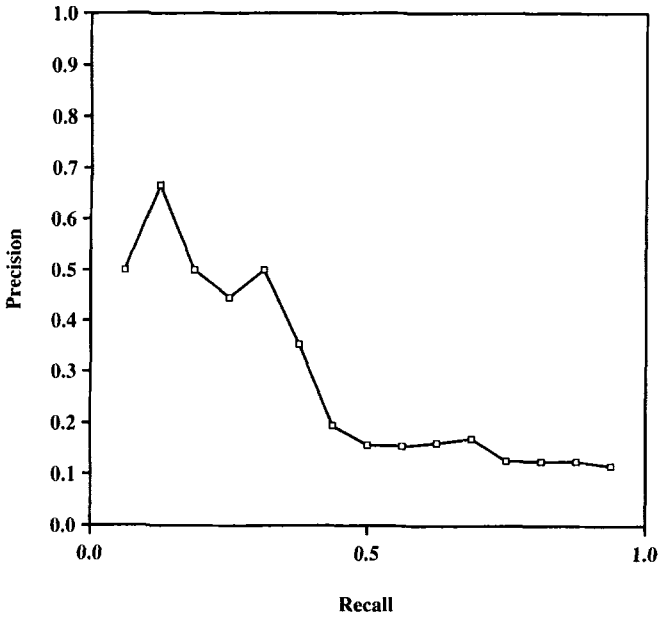


Figure 6. Retrieval performance of the initial, untrained maverick list.

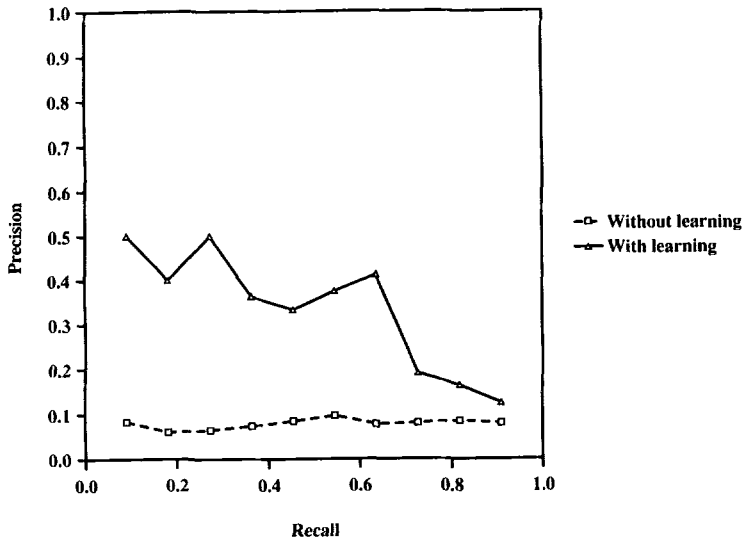


Figure 7. Retrieval performance of revised maverick list.

- The accuracy of the model is highly sensitive to individual feature weights.
- The accuracy of the model is relatively insensitive to the gross coefficients of the similarity function.

  Model performance can be substantially improved by learning from its biggest mistakes, without need for additional features.
- With a modest number of user-defined features, the learning component can adapt perfectly to the architect's judgment.

Future research combining the nearest-neighbor classification heuristic with other kinds of heuristic modularization advice may produce a practical module architecture advisor.

6.2. Related work on software similarity and modularity

Other software engineering research related to the present work falls into the following categories:

- Software similarity: work by Maarek and Kaiser (1987), for example, uses similarity and clustering to group software units in a reuse library. They rejected implementation features as irrelevant to reuse.
- Module and subsystem synthesis: Belady and Evangelisti (1982), Hutchens and Basili (1985), and Chen et al. (1990) have investigated clustering units into modules based on data bindings and data flow connection strength. None of these papers reports validation of the clusters by real maintainers. Maarek and Kaiser (1988) looked at clustering for the purpose of identifying software units that are likely to be modified at the same time. They proposed measuring affinity between software units by a combination of connection strength and how often in the past the units have been changed as part of the same task. Choi and Scacchi (1990) propose synthesizing subsystems based on articulation points in the cross-reference graph.
- Module quality analysis: Selby and Basili (1988) have measured the maintainability of a module by measuring its internal cohesion and external coupling to other modules. Porter and Selby (1990) successfully use this information to help predict the module's error proneness and cost to repair. They automatically construct decision trees from large volumes of real project data. This work is closest in spirit to the present work, in that it applied machine learning techniques to real-world data and measures success by real-world standards. However, the methods of these authors do not identify specific flaws in module quality, nor do they make reorganization recommendations.

6.3. Learning similarity vs. learning classification

The classifier and similarity learning method worked significantly better in the given problem domain than simpler methods that learn categories directly. We attribute the failure of simpler methods to the feature sparsity and category diversity inherent in the problem domain, and to the small number of examples in the training data. The information-hiding

principle predicts that very few features will occur widely; most will be rare, indicating that exemplar-style category models would be more appropriate than probabilistic models. The small number of examples of each category also suggested that the actual category members be used as exemplars, rather than a small set of synthesized prototypes representing a larger set of category members. Explicitly representing and learning the similarity function also permitted us to force it to be monotonic and matching, which probably also contributed to its success.

6.4. Heuristic design advisors

Several characteristics of the architecture advisor may be useful in other heuristic design advisors:

- The advisor embodies a model of the judgment-based human reasoning process it is advising, rather than merely checking a mechanistic design rule. While a practical tool would incorporate as many such design rules as are useful, adding judgment-based advice extends the usefulness of the tool considerably.
- By providing advice in the form of an ordered list of good alternatives, the tool was actually more useful than if it had given a single recommendation. By examining the alternatives, the architect could better understand why one was recommended over another, even if she chose one of the "lesser" alternatives.
- By providing interactive analysis on the architect's work in progress, the tool played the role of a subservient assistant rather than a demanding master. Whether or not the architect acted in accordance with or in opposition to the tool's advice, the tool could analyze the new situation, sometimes revise its judgment based on the architect's actions, and identify good alternatives for the next step.
- By acquiring most of its feature data from the design artifacts themselves, the tool was able to provide a useful level of service even before it acquired additional user-defined features and learned from its mistakes.
- Further knowledge acquisition was mistake-driven. The user could know much better what information to supply when she knew what mistake needed correcting. The alternative would have been to expect the user to supply much more information, not knowing which parts of it were important.
- By using its mistakes as "relevance feedback" to reorder the priorities of its other recommendations, the tool significantly improved the quality of its advice.

7. Future work

Clearly, more case studies are needed to confirm or contradict what was found in the TSL study. Such studies are somewhat expensive because they require an engineer knowledgeable about the code to comment on all mavericks, both real and false-positive. We find that an effective combination is to team up the knowledgeable engineer with an experimenter well versed in the information-hiding principle. The experimenter pre-screens the mavericks to point out the ones that are obviously true or false, thereby reducing the engineer's effort considerably.

Next, a method is needed for incorporating learning into the interactive clustering tools. The main difficulty is to automatically extract training data from the user session.

We believe that learning similarity rather than learning classification makes the knowledge learned more transferable to new problems. The similarity judgment learned during maverick analysis could be used to

- cluster the members of a module into smaller sub-modules.
- cluster modules into subsystems, and
- reanalyze the structure of the system after adding major enhancements.

Use in reanalysis would require a few extensions to the present work. Suppose that an architect did maverick analysis on a system, including several rounds of learning from relevance feedback, and saved the learned weights. After a major revision to the system, the architect's specific relevance judgments could not be reused without review, because the restructuring might have invalidated some of them. However, the learned weights represent his judgment of the relative significance of different implementation features, and that judgment would not be likely to change radically. Therefore, the learned weights could be used to compute the initial maverick list after reorganization, and then could be readjusted according to further relevance feedback. Some procedure would be needed for calibrating the a priori weight estimates for new features introduced by the enhancements, so that as a group they would be neither more nor less significant than the learned weights carried over from before the enhancements.

Substantial additional research is needed to create a practical architecture re-engineering tool. Advisory heuristics are needed that apply to variables, types, and macros. Representations are needed for the role of each module in a system, and for the conceptual relationships among modules. Heuistics are needed that relate individual units to the roles of modules, and explanation techniques are needed to present the analyses to the architect.

8. Conclusions

We conclude from the work described here that it is effective to model software modularization as a nearest-neighbor clustering and classification activity. The model supported effective heuristic assistance with that activity, and effective performance improvement by learning and knowledge acquisition. Naturally, similarity-based clustering is not the only principle used in software modularization. However, research on other heuristics can now be based on identifying those cases where the nearest-neighbor heuristic does not apply.

We also conclude that nearest-neighbor classification can be learned effectively by converting classification data to more-similar-than judgments and training the similarity function by back-propagation using the comparison paradigm. We did not do any efficiency studies, whether in time, space, or features needed. However, we believe that feature sparsity is an important characteristic of the problem domain and that nearest-neighbor classification captures more information about the sparse features than is collected in statistical classifiers.

Finally, we hope that the ideas about modeling judgment, giving advice, and acquiring knowledge will prove useful in the creation of other intelligent design assistants.

Notes

1. When the programming language does not have an explicit *module* construct, the programmer typically uses files to represent his modules. However, our use of the term *module* specifically does not cover systems where every module or file contains only one procedure.
2. The conference proceedings were not actually published until the following year.
3. The astute programmer will be worrying about the problem of duplicate variable names in different scopes, such as i, j, and k, which are declared many times in large systems, but with unrelated meanings. We restrict our features to *non-local* names, so that private variables are not considered, and give each distinct software unit a unique name system wide, so that there is no problem with duplicate names.
4. Since greater values of C imply poorer confidence, one might more intuitively call this a measure of doubt.

References

Balcer, M.J., Hasling, W.M., & Ostrand, T.J. (1989). Automatic generation of test scripts from formal test specifications. *Proceedings of the ACM SIGSOFT 1989 Third Symposium on Software Testing, Analysis, and Verification.* Key West, FL: ACM Press.

Baum, E., & Haussler, D. (1988). What size net gives valid generalization? In D. Touretzky (Ed.), *Advances in neural information processing systems* (Vol. 1). Morgan-Kaufmann.

Belady, L.A., & Evangelisti, C.J. (1982). System partitioning and its measure. *Journal of Systems and Software, 2(2).*

Chapin, N. (1988). Software maintenance life cycle. *Proceedings of the Conference on Software Maintenance—1988.* IEEE Computer Society Press.

Chen, Y.-F., Nishimoto, M., & Ramamoorthy, C.V. (1990). The C information abstraction system. *IEEE Transactions on Software Engineering, 16(3).*

Choi, S.C., & Scacchi, W. (1990). Extracting and restructuring the design of large software systems. *IEEE Software, 7(1),* 66-73.

Hanson, S.J., & Bauer, M. (1989). Conceptual clustering, categorization, and polymorphy. *Machine Learning, 3,* 343-372.

Hanson, S.J., & Burr, D.J. (1990). What connectionist models learn: Learning and representation in connectionist networks. *Behavioral and Brain Sciences, 13,* 471-518.

Homa, D. (1978). Abstraction of ill-defined form. *Journal of Experimental Psychology: Human Learning and Memory, 4,* 407-416.

Horn, K.A., Compton, P., Lazarus, L., & Quinlan, J.R. (1985). An expert system for the interpretation of thyroid assays in a clinical laboratory. *Australian Computer Journal, 17(1),* 7-11.

Hutchens, D.H., & Basili, V.R. (1985). System structure analysis: Clustering with data bindings. *IEEE Transactions on Software Engineering, 11(8).*

Lange, R., & Schwanke, R.W. (1991). Software architecture analysis: A case study. *Third International Workshop on Software Configuration Management.* ACM Press.

Maarek, Y.S., & Kaiser, G.E. (1987). Using conceptual clustering for classifying reusable Ada code. *Using Ada: ACM SIGAda International Conference.* Special issue of *Ada Letters.* ACM Press.

Maarek, Y.S., & Kaiser, G.E. (1988). Change management for very large software systems. *Seventh Annual International Phoenix Conference on Computers and Communications.* Scottsdale, AZ.

Medin, D., & Schaffer, M.M. (1978). Context theory of classification learning. *Psychological Review, 85,* 207-238.

Parnas, D.L. (1972). Information distribution aspects of design methodology. *Information Processing 71.* Amsterdam: North-Holland.

Parnas, D.L. (1971). On the criteria to be used in decomposing systems into modules (Technical Report No. CMU-CS-71-101). Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA.

Porter, A.A., & Selby, R.W. (1990). Empirically guided software development using metric-based classification trees, *IEEE Software, 7(3).*

Posner, M.I., & Keele, S.W. (1968). On the genesis of abstract ideas. *Journal of Experimental Psychology, 77,* 353–363.

Rivest, R., Haussler, D., & Warmuth, M.K. (1989). *Proceedings of Second Annual Workshop on Computational Learning Theory.* Morgan-Kaufmann.

Schwanke, R.W., & Platoff, M.A. (1989). Cross references are features. *Second International Workshop on Software Configuration Management. Software Engineering Notes, 17*(7), ACM Press.

Selby, R.W., & Basili, V.R. (1988). Error localization during software maintenance: Generating hierarchical system descriptions from the source code alone. *Conference on Software Maintenance—1988.* IEEE Computer Society Press.

Smith, E.E., & Medin, D.L. (1981). *Categories and concepts.* Cambridge, MA: Harvard University Press.

Sompolinsky, H., & Tishby, N. (1990). Learning from examples in large neural networks. *Siemens Computational Learning Theory and Natural Learning Systems Workshop,* September 5 & 6, Princeton, NJ.

Tesauro, G., & Sejnowski, T.J. (1989). A parallel network that learns to play Backgammon. *Artificial Intelligence, 39,* 357–390.

Received September 24, 1990
Accepted July 24, 1992
Final Manuscript December 15, 1992