

Chapter 10

Outlier Finding

Testing and debugging are important problems for any kind of automation, and pattern matching is no exception. Users may write incorrect patterns that inadvertently match some regions that were not meant to be matched (*false positives*) or fail to match some regions that should have been matched (*false negatives*). Inferred patterns may also need debugging. Even though 84% of inferred selections in the simultaneous-editing study (Section 9.3.1) needed only one example, the remaining 16% were *wrong* after the first example. Users have to notice that an inferred selection is incorrect before they can fix it by giving additional examples. Unfortunately, as noted in the discussion of that study, users did not always notice incorrect inferences, and made errors as a result.

This chapter¹ presents a new technique for debugging pattern matches: *outlier finding*. In statistics, an outlier is a data point which appears to be inconsistent with the rest of the data [BL84]. Applied to text pattern matching, an outlier is a pattern match (or mismatch) which differs significantly from other matches (or mismatches). Automatic outlier finding helps the user focus attention on the cases that are most likely to be problematic.

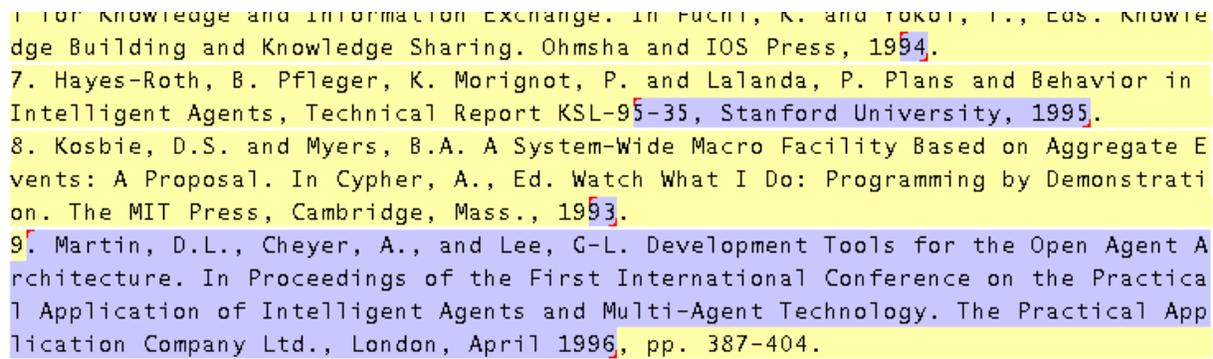
Briefly, the outlier finding algorithm takes any region set, generates a vector of binary-valued features representing each region, and then sorts the regions based on their weighted Euclidean distance from the median in feature vector space. Regions that lie far from the median are considered outliers.

In the LAPIS user interface, outlier finding is used in two ways. First, during selection inference, outliers in the system’s inference are highlighted automatically to draw the user’s attention to them. Second, in any mode, the user can use an “Unusual Matches” visualization to view and explore the currently-selected regions in outlier order.

Outlier finding depends on two assumptions. First, most matches must be correct, so that errors are the needles in the haystack, not the hay. This assumption is essential because the outlier finder has no way of knowing what region set the user actually intends the pattern to match. Unless the region set is roughly correct to begin with, the outlier finder’s suggestions are unlikely to be helpful. Fortunately, outlier finding has more value when errors are rare, since it means the errors are harder for the user to find by a manual search.

Second, erroneous matches must differ from correct matches in ways that can be captured by the features available to the outlier finder. Although the outlier finder uses all the domain

¹Portions of this chapter are adapted from an earlier paper [MM01b].



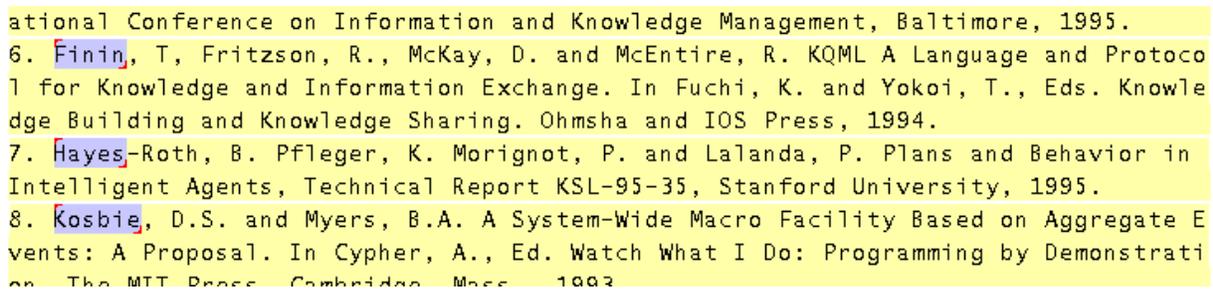
1. For Knowledge and Information Exchange. In Fuchi, K. and Yokoi, T., Eds. Knowledge Building and Knowledge Sharing. Ohmsha and IOS Press, 1994.

7. Hayes-Roth, B. Pflieger, K. Morignot, P. and Lalanda, P. Plans and Behavior in Intelligent Agents, Technical Report KSL-95-35, Stanford University, 1995.

8. Kosbie, D.S. and Myers, B.A. A System-Wide Macro Facility Based on Aggregate Events: A Proposal. In Cypher, A., Ed. Watch What I Do: Programming by Demonstration. The MIT Press, Cambridge, Mass., 1993.

9. Martin, D.L., Cheyer, A., and Lee, G-L. Development Tools for the Open Agent Architecture. In Proceedings of the First International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology. The Practical Application Company Ltd., London, April 1996, pp. 387-404.

Figure 10.1: Incorrect inference of the last two digits of the publication year. The selection is visibly wrong, and all users noticed and corrected it.



ational Conference on Information and Knowledge Management, Baltimore, 1995.

6. Finin, T, Fritzon, R., McKay, D. and McEntire, R. KQML A Language and Protocol for Knowledge and Information Exchange. In Fuchi, K. and Yokoi, T., Eds. Knowledge Building and Knowledge Sharing. Ohmsha and IOS Press, 1994.

7. Hayes-Roth, B. Pflieger, K. Morignot, P. and Lalanda, P. Plans and Behavior in Intelligent Agents, Technical Report KSL-95-35, Stanford University, 1995.

8. Kosbie, D.S. and Myers, B.A. A System-Wide Macro Facility Based on Aggregate Events: A Proposal. In Cypher, A., Ed. Watch What I Do: Programming by Demonstration. The MIT Press, Cambridge, Mass., 1993.

Figure 10.2: Incorrect inference of the author's name. "Hayes-Roth" is only partially selected, but no users noticed.

knowledge found in the LAPIS pattern library, the knowledge base inevitably has gaps, and the feature language may be incomplete. Any learning system has this problem, since it is impossible to infer a concept that cannot be represented. Fortunately, the LAPIS pattern library is easy to extend.

The next section delves into the kinds of errors made by users in the simultaneous editing study, in order to better motivate the outlier-finding solution. Subsequent sections describe outlier highlighting and the Unusual Matches dialog, which are the user interface techniques in LAPIS that rely on outlier finding. The fourth section details the outlier finding algorithm. The chapter concludes with a user study of outlier highlighting.

10.1 Motivation

Observations from the simultaneous-editing user study showed that some incorrect inferences are far more noticeable than others. Figure 10.1 shows an incorrect inference that was easy for users to notice. The user has selected "89", the last two digits of the first record's publication year, from which the system has inferred the description from end of first "9" to end of first year. This inference is drastically, visibly wrong, selecting far more than two digits in some records and nearly the entire last record. All eight users in the study noticed this error and corrected it by giving another example.

The mistake in Figure 10.2, on the other hand, was much harder to spot. The user has selected the last name of the first author, “Aha”. The system’s inference is `1st CapitalizedWord`, which is correct for all but record 7, where it selects only the first half of the hyphenated name “Hayes-Roth”. The error is so visually subtle that all seven users who made this selection or a related selection completely overlooked the error and used the incorrect selection anyway. (The eighth user luckily avoided the problem by including the comma in the selection, which was inferred correctly.) Although three users later noticed the mistake and managed to change “[Hayes 95]” to the desired “[Hayes-Roth 95]”, the other four users never noticed the error at all. A similar effect was seen in another task, when some users failed to notice that the two-word baseball team “Red Sox” was not selected correctly.

The difference between an inference whose errors were noticed by *all* users, and another whose errors were noticed by *none*, is telling. The selection errors in Figure 10.1 are too prominent for users to ignore, but the errors in Figure 10.2 are too subtle to be noticed.

10.2 Outlier Highlighting

In an effort to make incorrect selections like Figure 10.2 more noticeable, outlier highlighting was added to LAPIS. Whenever the system makes an inference in either selection guessing mode or simultaneous editing mode, it passes the resulting set of selected regions to the outlier finder, which rank the regions by their distance from the average selection. Using this ranking, the system highlights the most unusual regions in a visually distinctive fashion, in order to attract the user’s attention so that they can be checked for errors.

Two design questions immediately arise: how many outliers should be highlighted, and how should they be highlighted? Outliers are not guaranteed to be errors. Highlighting too many outliers when the selection is actually correct may lead the user to distrust the highlighting hint. On the other hand, highlighting more outliers means more actual errors may be highlighted. But highlighting a large number of outliers is unhelpful to the user, since the user must examine each one. Ideally, the outlier finder should highlight only a handful of selections when the selection is likely to have errors, and none at all if the selection is likely to be correct.

The following heuristic seems to work well. Let d be the distance of the farthest selection from the average, and let S be the set of selections that are farther than $d/2$ from the average. If S is small – containing fewer than 10 selections or fewer than half of all the selections, whichever is smaller – then highlight every member of S as a outlier. Otherwise, do not highlight any selections as outliers. This algorithm puts a fixed upper bound on the number of outlier highlights, but avoids displaying useless highlights when the selections are not significantly different from one another.

The second design decision is how outlier highlighting should be rendered in the display. One possibility is the way Microsoft Word indicates spelling and grammar errors, a jagged, brightly colored underline. Experienced Word users are accustomed to this convention and already understand that it’s merely a hint. In LAPIS, however, outlier highlighting must stand out through selected text, which is rendered using a blue background. A jagged underline was found to be too subtle to be noticed in this context, particularly in peripheral vision.

Instead, an outlier selection is highlighted by changing its color from blue to red. In simultaneous editing mode, the highlighting is further enhanced by coloring the outlier’s entire record red. The scrollbar is also augmented with red marks corresponding to the highlighted outliers. LAPIS

ational Conference on Information and Knowledge Management, Baltimore, 1995.

6. Finin, T, Fritzon, R., McKay, D. and McEntire, R. KQML A Language and Protocol for Knowledge and Information Exchange. In Fuchi, K. and Yokoi, T., Eds. Knowledge Building and Knowledge Sharing. Ohmsha and IOS Press, 1994.

7. Hayes-Roth, B. Pflieger, K. Morignot, P. and Lalanda, P. Plans and Behavior in Intelligent Agents, Technical Report KSL-95-35, Stanford University, 1995.

8. Kosbie, D.S. and Myers, B.A. A System-Wide Macro Facility Based on Aggregate Events: A Proposal. In Cypher, A., Ed. Watch What I Do: Programming by Demonstration. The MIT Press, Cambridge, Mass. 1993.

Figure 10.3: Incorrect inference with outlier highlighting drawing attention to the possible error.

already augments the scrollbar with marks corresponding to the selection (Section 7.3.4), so the red outlier marks are simply painted on top of the blue selection marks. Figure 10.3 shows the resulting display, highlighting an outlier in the erroneous author selection.

10.3 Unusual Matches Display

In simultaneous editing, outlier finding is used behind the scenes to direct the user's attention to possible errors. Some users may want to access the outlier finder directly, in order to explore the outliers and obtain explanations of each outlier's unusual features. For example, suppose a user is writing a pattern to replace all occurrences of a variable name in a large program, and the user wants to test and debug the pattern before using it. For this kind of task, LAPIS provides the Unusual Matches window (Figure 10.4).

The Unusual Matches window works in tandem with the LAPIS pattern matcher. Normally, when the user enters a pattern, LAPIS highlights all the pattern matches in the text editor. When the Unusual Matches window is showing, however, LAPIS also runs the outlier finder on the set of matches.

Unlike the outlier highlighting technique described in the previous section, the Unusual Matches window does not use a threshold to discriminate outliers from typical matches. Instead, it simply displays all the matches, in order of increasing *weirdness* (distance from the median), and lets the user decide which matches look like outliers. Each match is plotted as a small block. Blocks near the left side of the window represent typical matches, being very close to the median, and blocks near the right side represent outliers, far from the median. The distance between two adjacent blocks is proportional to their difference in weirdness. Strong outliers appear noticeably alone in this visualization (Figure 10.4).

Matches with identical feature vectors are combined into a cluster, shown as a vertical stack of blocks. Matches that lie at the same distance from the median in feature space, but along different vectors, are not combined into a stack. Instead, they are simply rendered side-by-side with 0 pixels between them.

The user can explore the matches by clicking on a block or stack of blocks, which highlights the corresponding regions in the text editor, using red highlights to distinguish them from the other pattern matches already highlighted in blue. The editor window scrolls automatically to display the highlighted region. If a stack of blocks was clicked, then the window scrolls to the first region in the stack and displays red marks in the scrollbar for the others. To go the other way, the user

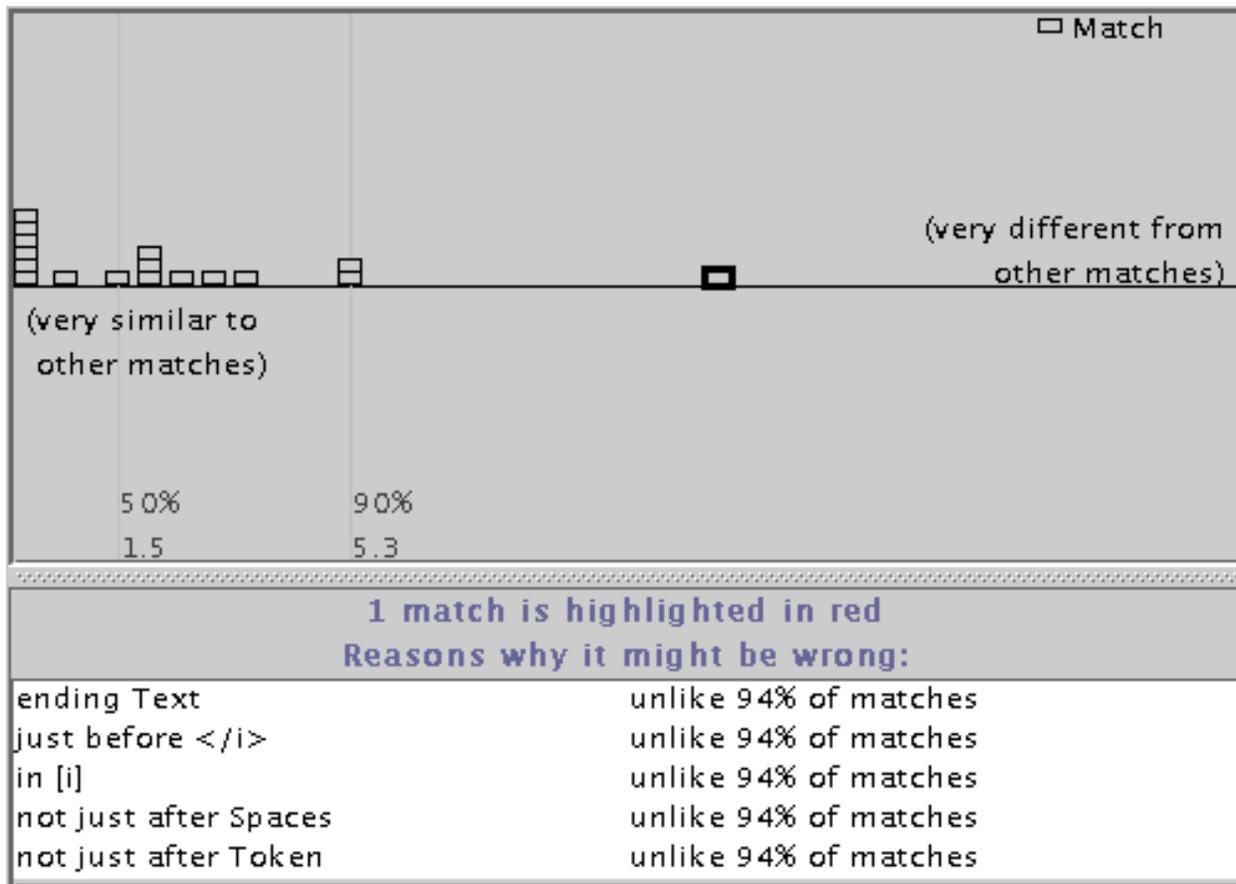


Figure 10.4: The Unusual Matches window showing occurrences of “copy” in a document. The most prominent outlier, which is selected, is found in an italicized word, “r*copy*”.

can right-click on a selected region in the editing window and choose “Locate in Unusual Matches Window”, which selects the corresponding block in the Unusual Matches window.

When a match is selected in the Unusual Matches window, the system also displays an explanation of how it is unusual (bottom pane in Figure 10.4). The explanation consists of the highest-weighted features (at most 5) in which the region differs from the median feature vector. If two features are related by generalization, such as `starts with Letters` and `starts with UpperCaseLetters`, only the higher-weighted feature is included in the explanation. Next to each feature in the explanation, the system displays the fraction of matches that agree with the median value — a statistic which is related to the feature’s weight, but easier for the user to understand. The explanation generator is still rudimentary, and its explanations sometimes include obscure or apparently-redundant features. Generating better explanations is a hard problem for future work.

The Unusual Matches window can also show mismatches in the same display (Figure 10.5). When mismatches are displayed, the user can search for both kinds of bugs in a pattern: *false negatives* (mismatches which should be matches) as well as *false positives* (matches which should not be).

The tricky part of displaying mismatches is determining a set of candidate mismatches. The search space for pattern matching is the set of all substrings of the document. A naive approach would let the set of mismatches be the complement of the matches relative to this entire search space. Unfortunately, this set is quadratic in the length of the document. There are at least three reasonable ways to reduce the search space. Currently, LAPIS only implements the first:

1. **Negated constraint.** Many patterns are written by appending one or more constraints to a library pattern. For example, `Line containing "Truman"` constrains the `Line` pattern. If the user’s pattern follows this scheme, then the constraint can be negated to find a set of candidate mismatches: `Line not containing "Truman"`. This technique effectively restricts the search space to the unconstrained library pattern, `Line`. The rules for finding this unconstrained pattern are the same as the rules used by the `Keep` command to find its record set (Section 8.1.3).
2. **All substrings between matches.** Since most applications of pattern matching (like `find-and-replace`) require nonoverlapping matches, a mismatch might be defined as any substring that does not overlap a match. Even though this set may still be quadratic, it can be represented compactly using region rectangles. This strategy has not been implemented in LAPIS.
3. **Approximate matches.** If the user specifies a literal string or regular expression pattern, then a set of mismatches can be generated by approximate string matching [WM92], which allows a bounded number of errors in the pattern match. This strategy has not been implemented either.

Regardless of how the possible mismatches are defined, the Unusual Matches window plots each mismatch on the same graph as the matches. Mismatches are colored white and plotted below the horizontal midline to clearly separate them from matches. Like matches, mismatches with identical feature vectors are clustered together into a stack. Clicking on a mismatch highlights it in the text editor and displays an explanation of why it should be considered as a possible match. The explanation consists of the highest-weighted features in which the mismatch agrees with the median match. Figure 10.5 shows the explanation for a mismatch.

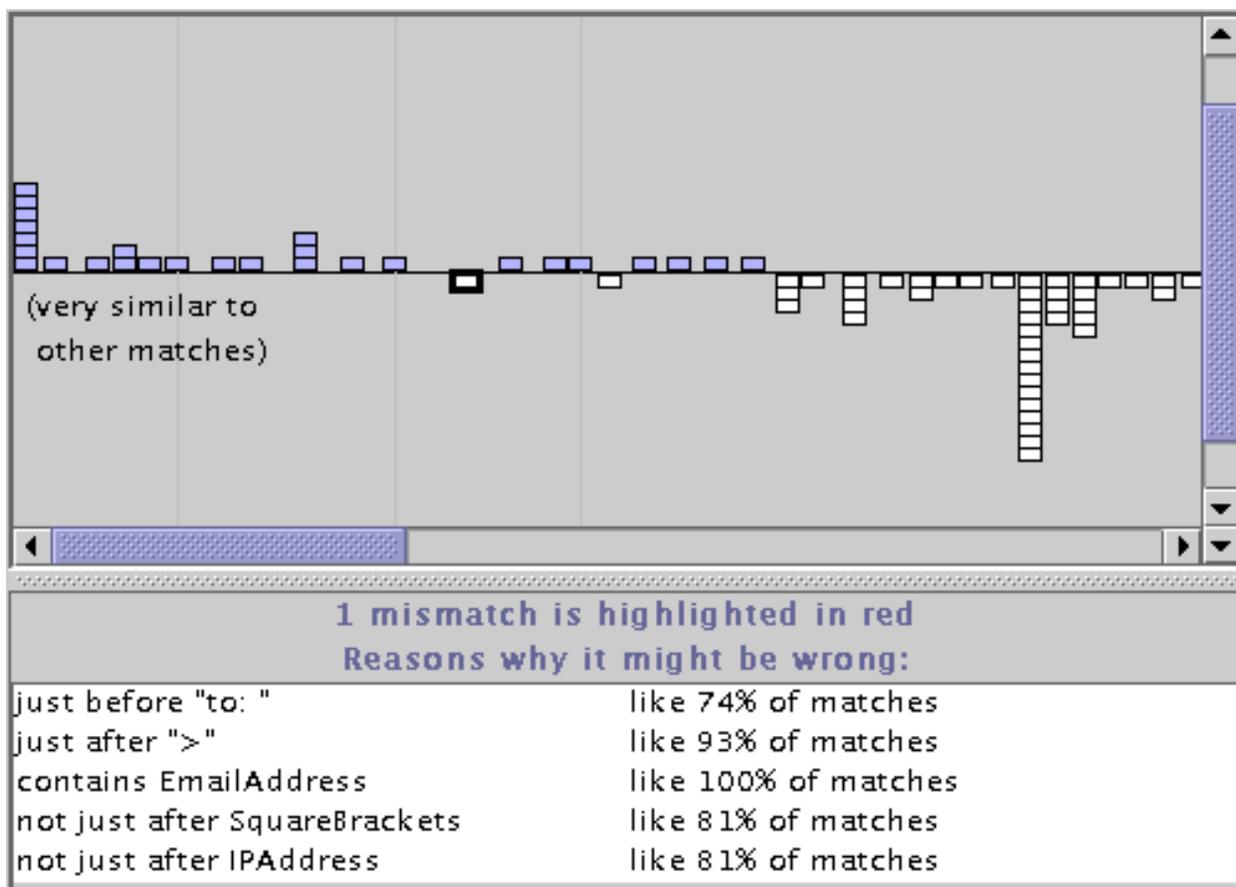


Figure 10.5: The Unusual Matches window showing both matches and mismatches to the pattern Line starting "From:" in a collection of email message headers. The most prominent mismatch, which is selected, is a Sender line which appears where the From line would normally appear in the message.

10.4 Implementation

This section describes the outlier finding algorithm. The algorithm takes as input a set of regions R and returns a ranking of R by each region's degree of similarity to the other members of R . Similarity is computed by representing each region in R by a binary-valued feature vector and computing the weighted Euclidean distance of each vector from the median vector of R . The distance calculation is weighted so that features which are more correlated with membership in R receive more weight.

Borderline mismatches are found by a related algorithm that takes two disjoint sets, R and \bar{R} , where R is the set of matches and \bar{R} is the set of mismatches. The algorithm then ranks the elements of both sets according to their similarity to R . Since this algorithm is used to rank both matches and mismatches, it will be referred to below as the *two-sided outlier finder*, while the algorithm that ranks only the matches will be called the *one-sided outlier finder*. Outlier highlighting uses one-sided outlier finding. The Unusual Matches window uses the two-sided outlier finder when the user's pattern can be negated using the "negated predicate" technique described previously. Otherwise, the Unusual Matches window falls back to one-sided outlier finding. The discussion below focuses on one-sided outlier finding, mentioning the two-sided algorithm only where it differs.

The only part of these algorithms that is specific to text substrings is feature generation. Applying the algorithm to other domains would entail using a different set of features, but otherwise the algorithm would remain the same.

10.4.1 Feature Generation

The features used by the outlier finder take the same form as the features used by the selection guessing algorithm (Section 9.2.1). A feature is a TC expression of the form $op\ F$, where op is one of the TC operators `equals`, `just before`, `just after`, `starting`, `ending`, `in`, or `contains`, and F is either a literal or a pattern identifier.

Features based on pattern identifiers are generated by searching the document for every pattern in the pattern library and then applying the seven TC operators to each of the resulting region sets.

Literal features are generated by examining the text of the regions in R . The literal feature generation algorithms are similar to those used by selection guessing (Section 9.2.1), with one important difference. Selection guessing only needs features that describe all the positive examples. Outlier finding, on the other hand, needs features that match some, but not all, of the regions in R . Literal features that match only a few regions in R are not useful either. For example, `contains "apple"` is not a useful feature if only one region in R contains the string `apple`. Admitting a feature like this would require us to admit *all* substrings as features — e.g., `banana`, `pear`, `passionfruit` — which would greatly expand the feature list without gaining any particular leverage for finding outliers. Thus, it is necessary to set some threshold on the frequency of a literal feature. Based on the assumption that outliers are rare, the threshold used in LAPIS is simple majority. A literal feature must match at least half of the regions in R to be used for outlier finding.

At present, only four kinds of literal features are generated. (In the discussion below, the regions in R are denoted by $x_i[y_i]z_i$, for $1 \leq i \leq |R|$, in some arbitrary order. For any region i , the whole document is the string $x_i y_i z_i$ and y_i is the part of the document selected by region i .)

- `starting`: Find the longest common prefixes of every pair of y_i . This can be done efficiently by sorting the y_i and comparing every adjacent pair of strings. For each longest common prefix p , generate `starting "p"` if and only if p is a prefix of at least half the y_i .
- `ending`: find the pairwise longest common suffixes of the y_i that are suffixes of at least half the y_i , using an algorithm analogous to `starting`.
- `just before`: find the pairwise longest common prefixes of the z_i that are prefixes of at least half the z_i .
- `just after`: find the pairwise longest common suffixes of the x_i that are suffixes of at least half the x_i .

The remaining three operators, `equals`, `in`, and `contains`, are not used to generate literal features. Literal features using `equals` would be redundant with `starting` and `ending` features that are already generated. Literal features using `in` are either redundant with `starting` or `ending` or not useful. Literal features using `contains` are hard to generate efficiently. The suffix tree approach used by selection guessing only generates `contains` features that match all the regions in R , which would be useless for distinguishing outliers. Developing an efficient algorithm to find substrings that are common to at least half of R is an interesting problem for future work.

The two-sided outlier finder generates literal features by sorting both R and \bar{R} together, so that it considers literal features shared by any pair of matches or mismatches. However, a literal feature must be shared by at least half of R or at least half of \bar{R} to be retained as a feature.

10.4.2 Feature Weighting

After generating a list of features, the next step is determining how much weight to give each feature. Without weights, only the number of unusual features would matter in determining similarity. For example, without weights, two members of R that differ from the median in only one feature would be ranked the same by the outlier finder, even if one region was the sole dissenter in its feature and the other shared its value with 49% of the other members of R . We want to prefer features that are strongly skewed, such that most (but not all) members of R have the same value for the feature.

The one-sided outlier finder weights each feature by its inverse variance. Let $P(f|R)$ be the fraction of R for which feature f is true. Then the variance of f is

$$\sigma_f = P(f|R)(1 - P(f|R))$$

The weight for feature f is $w_f = 1/\sigma_f$ if $\sigma_f \neq 0$, or zero otherwise. With inverse variance weighting, features that have the same value for every member of R ($\sigma_f = 0$) receive zero weight, and hence play no role in the outlier ranking. Features that are evenly split receive low weight, and features that differ on only one member of R receive the highest weight ($|R|/(|R| - 1)$).

Two-sided outlier finding uses not only R but also \bar{R} to estimate the relevance of a feature. We want to give a feature high weight if it has the same value on most members of R , but the opposite

value on most of \bar{R} . To do this, the two-sided outlier finder uses *mutual information* to estimate the weights [Pap91]. The mutual information between a feature f and the partition R, \bar{R} is given by

$$MI_f = H(R) - H(R|f)$$

where $H(R)$ is the entropy of R and $H(R|f)$ is the conditional entropy of R given f :

$$\begin{aligned} H(R) &= -P(R) \log P(R) - P(\bar{R}) \log P(\bar{R}) \\ H(R|f) &= P(f) (-P(R|f) \log P(R|f) - P(\bar{R}|f) \log P(\bar{R}|f)) \\ &\quad + P(\bar{f}) (-P(R|\bar{f}) \log P(R|\bar{f}) - P(\bar{R}|\bar{f}) \log P(\bar{R}|\bar{f})) \end{aligned}$$

Mutual information is related to the information gain heuristic used to induce decision trees [Qui86].

10.4.3 Feature Pruning

After computing weights for the features, the next step is pruning out redundant features. Two features are *redundant* if the features match the same subset of R (and \bar{R}) and one feature logically implies the other. For example, in a list of Yahoo URLs, the features `starting URL` and `starting "http://www.yahoo.com"` would be redundant. Keeping redundant features gives them too much weight, so the system keeps only the more specific feature and drops the other one.

Features are tested for redundancy by first sorting them by weight and comparing features that have identical weight. Since features are represented internally as region sets, the system can quickly compare the two region trees to test whether the matches to one feature are a subset of the matches to the other. Thus the system can find logical implications between features without heuristics or preprogrammed knowledge. It doesn't need to be told that `LowercaseLetters` implies `Letters`, or that `starting "http"` implies `starting URL`. The system discovers these relationships at runtime by comparing the region sets that these features match.

Pruning does not eliminate all the dependencies between features. For example, in a web page, `contains URL` and `contains Link` are usually strongly correlated, but neither feature logically implies the other, so neither would be pruned. The effect of correlated features could be reduced by using the covariances between features as part of the weighting scheme, but it is hard to estimate the covariances accurately without a large amount of data. Accurately estimating the n^2 covariances among n features would require $O(n^2)$ samples. Another solution would be to carefully design the feature set so that all features are independent. This might work for some domains, at the cost of making the system much harder to extend. One of the benefits of the current approach is that new knowledge can be added to the outlier finder simply by writing a pattern and putting it in the library. Thus a user can personalize the outlier finder with knowledge like `CampusBuildings` or `ProductCodes` or `MyColleagues` without worrying about how the new patterns are related to existing patterns.

10.4.4 Ranking

The last step in outlier finding is determining a typical feature vector for R and computing the distance of every element of R from this typical vector.

To form the typical feature vector, the outlier finder computes the *median* value of each feature over all elements of R . Another possibility is the mean vector, but the mean of every nontrivial feature is a value between 0 and 1, so every member of R differs from the mean vector on most features and looks a bit like an outlier as a result. The median vector has the desirable property that when a majority of elements in R share the same feature vector, that vector is the median.

After computing the median feature vector \vec{m} , the outlier finder computes the weighted Euclidean distance $d(\vec{r})$ between every $\vec{r} \in R$ and \vec{m} :

$$d(\vec{r}) = \sqrt{\sum_f w_f (r_f - m_f)^2}$$

R is then sorted by distance $d(\vec{r})$. Elements of R with small $d(\vec{r})$ values are typical members of R ; elements with large $d(\vec{r})$ values are outliers.

The two-sided outlier finder also computes $d(\vec{r})$ for members of \bar{R} . Members of \bar{R} with small $d(\vec{r})$ values share many features in common with R , and hence are outliers for \bar{R} .

10.5 User Study

To evaluate the effectiveness of outlier highlighting, the simultaneous-editing user study from Section 9.3.1 was repeated with new subjects. The only difference between the original study and the new study was the presence of outlier highlighting. The new study's tutorial discussed what outlier highlighting looks like and what it means.

For the new study, six new users were found by soliciting campus job newsgroups (`cmu.misc.jobs` and `cmu.misc.market`). All were college undergraduates with substantial text-editing experience and varying levels of programming experience (3 described their programming experience as “little” or “none,” and 3 as “some” or “lots”). Users were paid \$5 for participation.

10.5.1 Results

Comparing the two groups of users, one with outlier highlighting and the other without, showed a reduction in uncorrected inferences, although the sample size was too small for statistical significance. The system's incorrect inferences on the three tasks fall into four categories:

- Year (task 1): selection of the last two digits of the year (Figure 10.1)
- Author (task 1): selection of the author's name or the position just after it, which errs on “Hayes-Roth” (Figure 10.2)
- Winner (task 3): selection of the winning team's name or just after it, which errs on “Red Sox”
- Loser (task 3): selection of the losing team's name or just after it, which errs on (a different instance of) “Red Sox”

Selection	Corrected selections	
	No outlier highlighting	Outlier highlighting
Year	8/8 (100%)	7/7 (100%)
Author	0/7 (0%)	5/8 (63%)
Winner	1/8 (13%)	4/7 (57%)
Loser	4/7 (57%)	5/6 (83%)

Table 10.1: Fraction of incorrectly-inferred selections that were noticed and corrected by users (number corrected / number total). The denominators vary because some users never made the selection and others made it more than once.

Task	Tasks completed with errors	
	No outlier highlighting	Outlier highlighting
1	4/8 (50%)	2/6 (33%)
2	1/8 (13%)	2/6 (33%)
3	3/8 (38%)	1/6 (17%)

Table 10.2: Fraction of tasks completed with errors in final result (number of tasks in error / number total).

Only tasks 1 and 3 have incorrect inferences. All selections in task 2 are inferred correctly from one example. For all the incorrect inferences, outlier finding was perfect in the sense that every error was highlighted as an outlier, so overlooked errors are not due to the outlier finding algorithm. In these tasks, the outlier finder also highlighted regions that were not errors.

All users in both groups noticed that the Year selection was inferred incorrectly and corrected it, probably because the inference is dramatically wrong (Figure 10.1). For the other two kinds of selections, the outlier highlighting algorithm correctly highlighted the errors in the selection. As a result, users seeing the outlier highlighting corrected the Author, Winner, and Loser inferences more often than users without outlier highlighting (Table 10.1). In particular, the Author inference, which was *never* noticed or corrected without outlier highlighting, was noticed and corrected 5 out of 8 times (63%) with the help of outlier highlighting. Users confirmed the value of outlier finding by their comments during the study. One user was surprised that outlier highlighting was not only helpful but also conservative, highlighting only a few places.

Because outlier highlighting encouraged users to correct bad inferences, it also reduced the overall error rate on tasks 1 and 3, measured as the number of tasks finished with errors in the final result (Table 10.2). Note that editing with an incorrectly-inferred selection did not always lead to errors in the final output, because some users noticed the errors later and fixed them by hand.

Although outlier highlighting reduced the number of errors users made, it did not eliminate them entirely. One reason is that the system usually took 400-800 milliseconds to compute an inference, with or without outlier highlighting, and users did not always wait to see the inferred selection before issuing an editing command. For example, in the outlier-highlighting condition, 2 of the 3 uncorrected Author inferences went uncorrected because the user issued an editing command before the inferred selection and outlier highlighting even appeared. After the user study, outlier highlighting was changed so that records containing outliers remain highlighted in red through subsequent editing operations, until the user makes a new selection. As a result,

Task	Equivalent task size	
	No outlier highlighting	Outlier highlighting
1	8.4 recs [2.1–12.2 recs]	11.3 recs [8.1–18.0 recs]
2	3.6 recs [1.9–5.8 recs]	4.7 recs [3.2–7.2 recs]
3	4.0 recs [1.9–6.2 recs]	3.4 recs [2.5–4.0 recs]

Table 10.3: Equivalent task sizes (mean [min–max]) for each task in each condition.

even if the user doesn't notice an incorrect selection before editing with it, the persistent outlier highlighting hopefully draws attention to the error eventually.

Outlier highlighting also draws attention to correct inferences, undeservedly. Several users felt the need to deal with false outliers, to “get rid of the red” as one user put it. The design inadvertently encouraged this behavior by erasing the red highlight if the user provided the outlier as an additional example. As a result, several users habitually gave superfluous examples in order to erase all the outlier highlighting. Of the 143 total selections made by users with outlier highlighting, 16 were overspecified in this way, whereas no selections were overspecified by users without outlier highlighting. To put it another way, the tasks in the user study required an average of 1.26 examples per selection for perfect inference. Without outlier highlighting, users gave only 1.13 examples per selection, underspecifying some selections and making errors as a result. With outlier highlighting, users gave 1.40 examples per selection, overspecifying some selections. Giving unnecessary examples is not only slower but also error-prone, because the extra examples may be inconsistent with the desired selection. This happened to one user in task 2 — a correct inference became incorrect after the user misselected an outlier while trying to erase its highlight.

After the study, several design changes were made to mitigate the problem of overspecified selections. First, selecting an outlier as an additional example no longer erases its outlier highlighting. Instead, users who want to “get rid of the red” must right-click on an outlier to dismiss its highlight, eliminating the danger of misselection. This design was inspired by Microsoft Word, which uses the context menu in a similar fashion to ignore or correct spelling and grammar errors. Second, the outlier highlighting for simultaneous editing was changed so that only the record containing the outlier selection is colored red. The selections themselves remain blue, in order to make it clearer that the entire selection can be used for editing even if it contains outliers. There can be no ambiguity about which selection is the outlier, because each record in simultaneous editing contains exactly one selection.

Because outlier highlighting encourages users to attend to possible errors and give more examples, it also tends to slow down editing somewhat. Table 10.3 compares the equivalent task sizes with and without highlighting. The average equivalent task size with outlier highlighting was slightly larger for tasks 1 and 2, indicating that outlier highlighting made simultaneous editing slower on average. On task 1, extra editing time was well spent, because users corrected bad inferences. On task 2, all the inferences were correct after one example, so the extra editing time was wasted attending to false outliers. On task 3, however, the average equivalent task size actually decreased slightly. At least part of this decrease is explained by the fact that it takes less time to correct a selection error when it is noticed *before* editing with it (which happened more often in the outlier-highlighting condition) than it does to correct errors at the end of the task (which happened more often in the no-outlier-highlighting condition). A stitch in time saves nine.

