Sharing, finding and reusing end-user code for reformatting and validating data

Christopher Scaffidi
cscaffid@eecs.oregonstate.edu

School of Electrical Engineering and Computer Science, Oregon State University
1148 Kelley Engineering Center, Oregon State University, Corvallis, OR 97331-4501
541-737-5572 (phone)
360-935-7708 (fax)

To help users with automatically reformatting and validating spreadsheets and other data sets, prior work introduced a user-extensible data model called "topes" and a supporting visual programming language. However, no support has existed to date for users to exchange and reuse topes. This functional gap results in wasteful duplication of work as users implement topes that other people have already created.

In this paper, a design for a new repository system is presented that supports sharing and finding of topes for reuse. This repository tightly integrates traditional keyword-based search with two additional search methods whose usefulness in repositories of end-user code has gone unexplored to date. The first method is "search-by-match," where a user specifies examples of data, and the repository retrieves topes that can reformat and validate that data. The second method is collaborative filtering, which has played a vital role in repositories of non-code artifacts.

The repository's search functionality was empirically tested on a prototype repository implementation by simulating queries generated from real user spreadsheets. This experiment reveals that search-by-match and collaborative filtering greatly improve the accuracy of search over the traditional keyword-based approach, to a recall as high as 95%. These results show that search-by-match and collaborative filtering are useful approaches for helping users to publish, find, and reuse visual programs similar to topes.

Keywords: end-user programming; end-user software engineering; data; reuse; spreadsheets

1. Introduction

Everyday tasks often require reformatting and validating short human-readable strings such as phone numbers and employee ID numbers. For example, an administrative assistant at a university might gather professors' contact information from multiple web pages into a spreadsheet. Since data on different sites are often formatted differently, putting values into a consistent format can take some effort. For instance, some phone numbers might be formatted like "(541) 737-5572" while others might have a university-specific format like "7-5572," calling for tedious and error-prone reformatting into a consistent format. The user might accidentally make mistakes (e.g., dropping a digit as in "7-557"), and noticing such errors can be difficult when they are buried amid hundreds or thousands of strings.

To help users with these tasks, prior work introduced a user-extensible data model called "topes" and a supporting visual programming language to automate the reformatting and validation of strings [20][21][22][23]. Each tope is an abstraction that describes how to reformat and validate instances of one data category. For example, a user might create a tope for phone numbers. Topes have proven highly reusable across spreadsheets and even across applications (e.g., reusing a tope initially created for spreadsheets to validate databases) [21][23]. Studies have shown that topes can precisely reformat and validate many kinds of strings [23], that people can use the toolset's visual language to quickly and correctly create topes [21][22], and that users are extremely satisfied with the toolset and would like to see it deployed commercially [21][22].

However, no support has been provided to date in order to help people reuse one another's topes. That is, while a user could search through and reuse his or her own topes, there was no way for people to reuse each other's topes. This has been a major limitation because many people use the same kinds of data, and it is inefficient for each person to essentially repeat one another's work in creating a tope or grammar from scratch. Moreover, this limitation has been present in other similar systems, including tools for creating context-free and similar grammars [1][12][14][17].

Outside of grammars, a common approach for tackling this problem is to provide a repository where end users can publish, find, and reuse code. Repositories typically provide mechanisms so users can try out code before downloading it, and they provide minimalist support for browsing and search. Specifically, search is generally based on keywords, tags and categories (e.g., [15][24][25]). However, the effectiveness of these search approaches is limited by the well-known "vocabulary problem," in which different people use widely different labels and tags for the same things [9].

The objective in the current paper is to present and evaluate a repository approach for publishing and reusing topes, with a strong emphasis on discovering effective search algorithms that are well-suited for finding topes, context-free grammars, and similar programmatic descriptions of string data.

Two key insights drive this investigation. The first is that users probably will not want to reuse a tope until they have some data that they need to reformat or validate. For example, a user might want a tope for reformatting phone numbers because he has some phone numbers that need to be reformatted. Consequently, it would be ideal if it were possible to search the tope repository by specifying examples of the strings to match; this approach will be referred to as "search-by-match."

The second insight is that although some kinds of data (such as URLs) could be called "generic" or "universal" as they are used by virtually everyone, many kinds of data are specific to industries, organizations, regions, or other groups of people. For example, the spreadsheets of stock traders might have stock ticker symbols ("GOOG") and stock exchange symbols ("AMEX"), while the spreadsheets of college teachers might have grades ("B+") and course numbers ("COS-101"). Some kinds of data will even be specialized within a particular school or other organization. This insight drives a decentralized repository architecture, where users can "subscribe" to specific repositories that contain topes particular to their organization or industry. In addition, since some public repositories might contain topes relevant to more than one group of users, this insight calls for empirically evaluating whether a dynamic clustering mechanism such as collaborative filtering [3] can be used to improve the accuracy of search results, by matching topes to people based on what other topes have previously been used by "similar" people.

The central hypothesis is that it is possible to quickly and accurately find a tope in a repository based on three pieces of information: keywords, example strings that the tope should match, and a set of recently used topes. To evaluate this hypothesis, a prototype tope repository called "TopeDepot" has been implemented. The repository's search functionality was then empirically tested by simulating queries generated from real user spreadsheets whose data might need to be reformatted or validated.

This experiment has shown that the search-by-match approach is substantially more accurate than simply searching based on keywords and/or tags as in most existing end-user code repositories. Moreover, collaborative filtering further increases accuracy, to a recall as high as 95%. These results suggest that search-by-match and collaborative filtering will support reuse of topes and similar grammars at least as well as many existing code repositories that have already proven successful in practice.

The remainder of this paper is organized as follows. Section 2 discusses related work highlighting the need for mechanisms to facilitate sharing and reuse of grammars like topes. Section 3 presents the topes data model and summarizes prior experiments showing that users can successfully create topes through a visual programming language and use them to reformat and validate data. Section 4 discusses the TopeDepot prototype, including its subscription-based architecture and search interface. Section 5 specifies the search problem to be solved within a topes repository and describes three candidate algorithms that each combine aspects of search-by-match with collaborative filtering. Section 6 presents the empirical evaluation of the search algorithms, revealing that search-by-match and collaborative filtering improve search accuracy over the traditional keyword-based approach. Section 7 discusses implications for repository design. Section 8 summarizes the key conclusions.

2. Related work

Topes are the first abstraction to integrate support for *reformatting* and *validating* the short, human-readable strings of everyday life. Other prior research initiatives addressed either reformatting or validating alone.

In the area of reformatting, Potluck [11] and Lapis [14] support simultaneous editing, where a user edits one string, and the system automatically makes similar changes to other strings. These edits are instantaneous—not stored as rules that can be shared and later executed on other data. Nix's editing-by-example technique is similar, and it infers a macro-like program that records edits and can be replayed later [18], but no mechanism was provided so users could exchange macros.

In the area of validation, Grammex [12], Lapis [14], and Apple Data Detectors [17] each enable users to create context-free or similar grammars using textual programming languages assisted with a visual interface. In addition, SWYN presents a visual language of colored boxes and bubbles for creating regular expressions [1]. None of these systems includes a repository where users can publish and find each other's grammars. Outside of these research initiatives, small online repositories of regular expressions exist, the largest of which has eight categories each containing several dozen regular expressions [19]; only category-based browsing and keyword-based search are supported.

There has been one initiative to develop a heuristic-based search-by-match algorithm for validation code [4]. This algorithm centers on a novel data structure called RE-Trees for indexing regular expressions. While the accuracy of this algorithm was not evaluated, RE-Trees were found to greatly speed up search compared to trying out every regular expression in the repository to see which ones match example strings.

Outside of systems aimed at reformatting and validating data, the information extraction research community has long focused on the "named entity tagging" problem [3]. In this problem, an algorithm identifies words in a piece of text that are instances of particular categories. For example, the algorithm might scan a newspaper article to identify company names. Such algorithms resemble search-by-match in terms of identifying data categories based on example strings (though they also take advantage of contextual features—for example, the words "worked for" might precede a company name in natural language). All of these algorithms were highly tuned by researchers; there is no way for ordinary users to create and exchange such algorithms.

The efficiency of RE-Trees and the success of machine learning in information extraction research provide preliminary evidence that search-by-match might be useful for finding topes and similar grammars in a repository. However, more evaluation (the focus of the current paper) is necessary to explore how accurately search-by-match identifies search results, how well it generalizes to more complex descriptions of strings (such as topes) that support validation, and how it could be integrated effectively with other search approaches (including keyword-based search and collaborative filtering [3]).

3. Preliminary work

A tope is an abstraction which describes strings in a particular data category independently of any particular software application. For example, one tope might describe phone numbers, while another might describe URLs. A tope is a graph, where each node corresponds to a format and each edge corresponds to a transformation between formats. For instance, a phone number tope might recognize strings like "541-737-5572," "(541) 737-5572," and "541.737-5572"; for each format, the tope would include a function for recognizing instances of that format, and it also would include functions for transforming strings from one format to another. There is no "universally right" way to implement such a tope. Italians, for instance, would implement a different phone number tope than Americans would. Even within one country, an organization might need specialized versions of a tope.

A visual programming language and supporting toolkit called the Tope Development Environment (TDE) has been provided to help users create topes. With the TDE, a user actually edits a "data description," from which the TDE automatically generates a tope. Plug-ins to Excel and other end user software then pass strings into the tope's functions to reformat and validate data. Experiments have shown that topes are a powerful and effective abstraction for users, who can quickly and correctly create topes, then use them to reformat and validate data.

3.1. Tope model

Each tope describes the formats, and the relationships among those formats, for one kind of data [23]. For example, Fig. 1 is a sketch of a tope describing phone numbers. Each tope is a directed graph (F, T) with a function associated with each node and edge.

For each node, $f \in F$, isa_f is a fuzzy set membership function isa_f: $\Sigma^* \rightarrow [0, 1]$ that indicates whether a string is an instance of that format (where Σ is Unicode). Each isa function is modeled as a fuzzy set membership function because for many kinds of data, it can be impossible to conclusively determine if a string is a valid instance. In these cases, it is ideal to identify strings that are "questionable" so that a person can double-check them. For example, valid American phone numbers rarely have "555" in the exchange (the second set of three digits), since such numbers had been reserved for use in television shows (though they are now used in regions that are running out of phone numbers). Identifying such a string as questionable (neither definitely valid nor definitely invalid) makes it possible to flag it for review based on human judgment.

The "validation function" ϕ_τ of a tope $\tau = (F, T)$ ranges from 0 to 1 and indicates how well a string matches any of the tope's formats:

$$\phi_\tau(s) = \max_{f \in F}[\mathtt{isa}_f(s)]$$

For each edge $(x, y) \in T$, called a "transformation," trf_{xy} is a function trf_{xy}: $\Sigma^* \rightarrow \Sigma^*$ that converts a string from format x to format y. These functions could theoretically be chained at runtime, but in practice, this is typically not necessary because the TDE automatically implements a complete graph connecting each format to each other format.

3.2. Creating and using topes

A user actually does not implement a tope directly. Instead, he or she uses a visual programming tool to create a set of rules, called a "data description," from which the TDE automatically implements a tope.

To create a data description, a user typically starts by providing several examples of the data to be reformatted or validated [20][22]. For example, an Excel user could highlight a column of cells and click a button in the TDE's Excel plug-in, which appears in a toolbar on-screen. From these examples, the TDE infers a "boilerplate" data description that includes most or all of the strings. This tope implementation is then presented on-screen, where the user can review and customize it. For example, Fig. 2 shows a data description for phone numbers. It specifies two formats (one per row), with a rectangular icon representing each part of a phone number. Each icon contains an example string for that part (e.g.: "383" as an example area code), and the user can click on an icon to edit the name of that part as well as its rules. To start with, the inferred boilerplate data description has generic names like "PART2," but in the example figure, the user has already given a name to the first part ("area code"), has edited a rule to specify that the area code should be between 200 and 999, and has added a rule indicating that the area code cannot end with 11.

The TDE supports many different kinds of rules, which have proven quite adequate for describing a wide range of different kinds of data [21][23]. The most useful rules specify that a particular part should fall in a numeric range and that it should or should not contain certain characters or quantities of characters. Another kind of rule specifies that a part matches another data description. A final kind of rule specifies a "whitelist" indicating that parts of the string (or the string in its entirety) should be in a certain set. Each whitelist can also indicate that certain strings are synonyms of one another (that is, appearing in different formats). With few exceptions, each kind of rule can be configured to be "soft" in that it is sometimes violated even by valid strings. For example, a rule could be configured to specify that the exchange rarely is "555".

From the data description, the TDE automatically generates the tope's isa and trf functions [21][22]. Each isa is implemented by generating a context-free grammar based on the format's rules, with constraints attached to the grammar's productions. For example, one constraint on a phone area code production might check that the phone number's exchange is between 200 and 999, another might check that it does not end with 11, and a third might check that it is not 555. If a string satisfies the grammar and its constraints, then the generated isa function returns 1; otherwise, the isa function returns a lower score, depending on how many constraints were violated and their softness [21]. For each edge in the graph, the TDE implements a trf function that parses input strings using the starting format's grammar, then reformats the parts of the string to generate an output string in the target format. For example, reformatting a phone number from "541-737-5572" to "(541) 737-5572" requires parsing the string, changing the separators between the three parts of the string, and concatenating the parts with the new separators to generate the output. Since all of these functions are automatically implemented, users rarely need to worry about the details of the tope model. (However, users with skill in JavaScript can override these functions by writing custom code.)

Finally, the user can select a tope for reformatting or validating data. In Excel, the user can highlight cells and right-click, then select from a list of "recommended" topes that the user had already created (Fig. 3). The algorithm powering this recommendation list serves as a starting point for a repository search algorithm.

After the user selects a desired format, the TDE calls the transformation functions to put all strings into that format. In addition, it calls the format's isa function to validate all of the strings, since reformatting generally does not "fix" invalid strings—the transformed version of an invalid string is typically still invalid. In addition, under certain circumstances, the automatically-generated transformations can produce invalid strings, though this is generally not a problem in practice [22]. The TDE flags each invalid string with a small triangle and a tooltip listing constraints violated by each string (Fig. 4).

Once a tope has been created, the user can reuse it on other spreadsheets, web forms, and other programs. For example, a web form validated with a tope will show an error message alongside invalid inputs, and a database trigger could call a tope to check values (and throw an exception) before insertion into a table [20].

Topes are not actually embedded in these spreadsheets or databases, but rather stored in external files that are referenced by the spreadsheets and databases. There are two reasons for this. First, many data files (such as Excel spreadsheets) provide no practical mechanism for embedding topes. Second, and more importantly, topes are stored externally so that if their specification needs to be debugged or modified, then the requisite changes can be made by the user in a single place and immediately applied to spreadsheets and other data on the computer that reference this tope. (If every spreadsheet instead contained a copy of the tope, then the user would need to fix up all of these copies, which would be tedious and error-prone.) We recognized that users might not always want edits to be propagated everywhere, of course, so the TDE also allows the user to explicitly copy and customize separate versions of a data description. In software engineering terms, this is equivalent to forking separate versions of a tope by explicitly creating a clone. Thus, the default behavior is to keep a single version of a tope on a user's computer that can be easily maintained and applied to many spreadsheets and databases, but the user still can choose to use different versions for different data sets when so desired.

3.3. Summary of prior experimental results

Prior experiments have shown that topes are extremely effective at helping users to describe rules for reformatting and validating strings.

In a between-subjects experiment, half of the participants used the TDE to validate strings, while the other half of the participants used an alternate system [21]. Each participant was asked to create a tope or a Lapis grammar [14], respectively, then to use it to find typos in a list of strings taken from spreadsheets. This task was repeated for two other sets of strings taken from other spreadsheets. Overall, TDE users found three times as many typos as Lapis

users, and in half the time. The results also compared favorably with statistics provided by an earlier study involving the SWYN regular expression editor [1] (though the tasks were not identical, which limited comparability).

In a within-subjects experiment, participants created topes and used them to reformat 100 spreadsheet cells; they then had 60 seconds to manually reformat as many of these strings as possible [22]. This task pair was then repeated by each participant for two other spreadsheet columns. On average, participants took 4.5 minutes to create and use a tope, in comparison to an average time of 0.096 minutes to manually reformat a single string (for a comparable error rate), implying that the topes approach is faster when at least 47 strings need to be reformatted. This was (intentionally) a *conservative* estimate. Specifically, participants were already beginning to tire and slow in their manual edits before the 60 seconds ended (implying that they would be even slower when manually reformatting for longer periods of time), and participants were not counter-balanced (meaning that participants were more familiar with tasks when they did them manually than they were when they did them using topes).

Each user study included a satisfaction survey, which confirmed that users are extremely satisfied with the topes-based approach and eager to see it deployed in commercial applications. For example, in the latter experiment, the survey asked users to rate on 5-point Likert scale how hard the TDE was to use, how much they trusted it, how pleasant it was to use, and if they would use it if it was integrated with spreadsheets or word processors. Every participant but one gave a score of 4 or 5 on every question (the good end of the scale). Moreover, some people spontaneously gave detailed, unprompted descriptions of how they wished a tool like this had been available in office environments where they previously had worked, in order to do reformatting. The survey for the first experiment also showed statistically significant user preference for the TDE over Lapis.

In summary, topes and the supporting TDE constitute a highly effective and usable approach for helping people to validate and reformat strings. Moreover, these experiments did not even consider any additional benefits yet to be gained through reuse of topes.

4. TopeDepot prototype

Reuse of tope implementations could have a number of benefits. First, it could be faster for people to reuse an existing implementation, rather than creating one from scratch. Second, if a certain person (such as a new employee) is not yet familiar with a certain kind of organizational data (such as part numbers), then it might be impossible to implement a tope for that data, so reusing an existing tope implementation might be the only way to acquire one. Third, if different people reuse the same tope, then their computers would reformat and validate the corresponding data with the same rules, thereby reducing inconsistency and possibly the potential for miscommunication. Fourth, some people in an organization (such as system administrators, or highly-skilled end users sometimes called "gardeners" [16]) might have special skills or knowledge needed to design high-quality topes, particularly for especially complex data. For example, a system administrator would probably be more cognizant than the average user that a URL's top-level domain could be a two-character country code. If all workers could reuse these good tope implementations, then they would not need to create potentially-buggy tope implementations on their own. Finally, existing data descriptions could serve as raw materials helping to speed the creation of custom new data descriptions.

For all of these reasons, it is desirable to provide a mechanism whereby users can publish, find, and reuse topes. The typical approach taken in research aimed at end-user programmers is to provide a repository (e.g., [2][15]). For topes, a particular kind of repository is appropriate. Specifically, because so many topes are organization-specific, a subscription-based architectural approach has been taken in crafting the topes repository: that way, a user can subscribe to repositories that provide topes of interest. For example, an administrative assistant at Oregon State University might subscribe to his university's repository server (in order to download university-related topes) as well as a United States server (that provides topes tied to geography) and perhaps a global server (for topes to validate universal kinds of data).

Compared to simply assuming the existence of one single repository server, this subscription architecture has four advantages. First, allowing each user to focus on specific repositories helps to prevent distraction caused by topes

that are completely irrelevant due to organizational and societal factors. For example, there is no need for French users to be distracted by topes for American phone numbers. Second, delegating control of repository servers to organizations allows those organizations to secure and protect topes for proprietary data. For example, a particular organization might want to create a tope that includes a whitelist containing account numbers that should only be known within the organization. Third, while the prototype repository server does not require users to pay any money in order to subscribe to the server and download topes, some repositories could be created that do charge users for access to high-quality topes created by extremely skilled programmers. For example, Oracle or SAP could create a repository for common enterprise data and only allow access for customers. Finally, and perhaps most importantly, decentralizing the control of repository servers creates an opportunity for different organizations to provide innovative new repository server features. That is, if there had been just a single repository server, then only a single entity (the server's owner) would have been able to deploy new repository features. In contrast, the decentralized subscription-based architecture resembles the web, in that different repository owners could develop innovative new search engines and other features, and users could select and subscribe to different repositories based on which repositories offer the most usable user interface features.

TopeDepot is a prototype implemented to facilitate evaluation search algorithms and to provide a proof-of-concept view at what such a subscription-based tope repository might look like. The next three subsections, respectively, describe how a user subscribes to a repository, how a user publishes a data description, and how a user searches through a repository. The fourth subsection identifies opportunities to extend the prototype.

4.1. Subscribing to a repository

Subscribing to TopeDepot requires three steps. First, the repository's owner puts a copy of a repository descriptor file on any web server. Each repository descriptor file, formatted in XML, specifies a URL for publishing and unpublishing data descriptions using a standard REST protocol [6] (where an HTTP POST or PUT operation publishes a new object, DELETE unpublishes it, and GET retrieves the latest copy). The repository descriptor file

also specifies a “human interface” URL (discussed below) that a regular web browser can use to access the repository. Finally, the repository descriptor specifies a URL pointing to an HTML web form where the user can authenticate on the repository. (If the repository does not require authentication, then the descriptor omits this URL.)

Second, when a user who has the TDE installed clicks on a hyperlink to a repository descriptor file, Windows alerts the TDE, and the TDE prompts the user to confirm that he wants to add the referenced repository into the user’s list of recognized repositories. If the user confirms that this repository should be recognized, then the TDE stores the descriptor on the local computer, adding the repository to the user’s subscriptions. (The TDE also provides a window for removing subscriptions.)

Finally, the TDE checks to see if the descriptor file specifies an authentication URL. If it does, then the TDE displays a small web browser window for the user to authenticate. At this point, if the user does not have an account on the repository server, then TopeDepot prompts the user to create an account.

4.2. Publishing a data description

Later, after saving a data description locally, the user open the “Publish / Un-publish” dialog in the TDE (Fig. 5). The list of repositories is populated from the list of repository descriptor files. The “Publish” link allows the user to upload a data description to the repository. The TDE accomplishes this by performing an HTTP POST operation to the REST URL specified in the repository’s descriptor file. It immediately performs a GET to verify that the tope implementation was uploaded correctly. The “Unpublish” link allows the user to remove the data description from the repository. The toolset accomplishes this by performing an HTTP DELETE operation using the same URL.

4.3. Finding and downloading data descriptions

In addition to the REST interface provided for programmatic access to the repository, TopeDepot includes a web application allowing users to search through the data descriptions currently stored on the server. The user can access

the web application through a dialog window inside of the TDE's data description editor or through a traditional browser.

The web application's search engine allows users to find data descriptions based on a simple keyword search and/or by specifying examples of the strings that the user needs to validate and reformat. For example, the user could specify that she wants a "name" tope that can validate and reformat the strings "Oscar de la Renta" and "Mary Jane Phillips" (Fig. 6). Using one of the three alternate algorithms described in Section 5, the search engine then retrieves a list of data descriptions whose topes match the query. The repository uses icons to show how well each example string matches each data description's tope (with green up arrows flagging good matches, yellow squares flagging questionable matches, and red down arrows flagging poor matches).

If the user clicks on a search result, the web application displays detailed information, including the tope's name, author, and comments by other users (Fig. 7). With each comment, users can specify example strings that the data description's tope should match. If the user clicks the "Download" button, then the TDE retrieves the data description via the REST URL specified in the repository descriptor file. It saves this data description on the local computer, just as if the user had created the data description.

At this point, the user can customize the data description if desired, and the TDE can generate a tope for reformatting and validating strings. For example, a university system administrator might download the data description for an American phone number, customize it to include a university-specific format (e.g., "6-7890"), then publish the customized version into an organizational repository. From there, another user could download the data description and customize it still further (e.g., adding "home" to the data description's whitelist, as a synonym to a specific 10-digit number). The problem of customization will not receive further attention in this paper, since it can largely be handled with the system described in prior work [22].

4.4. TopeDepot as a basis for future enhancements

The simple three-step subscription process opens the way for enhancements by repository owners in future versions. First, repositories could restrict access only to users on an intranet (or range of internet addresses), thereby allowing an organization to prevent non-employees from accessing proprietary data descriptions. Second, commercial repositories could require users to enter a credit card number when creating an account. Third, to prevent users from copying or modifying proprietary data descriptions, repositories could allow users to create accounts only after downloading and installing digital rights management software. Finally, the current TDE embeds references to topes in spreadsheets and other datasets. When a user opens up a spreadsheet on a computer where a referenced tope has not been installed, the TDE shows a small visual error indicator in the spreadsheet. More sophisticated repository clients might proactively check to see if the missing tope is available in the organization's local repository, or in the other repositories where the user has a subscription, then download a copy. Of course, it would be desirable if such a client could be made configurable, so that the user can control what code would be invisibly downloaded to the computer.

Just as the repository client could be enhanced, the current TopeDepot provides the basis for further server enhancements. In particular, the prototype is a simple wiki-style content management system. Any user can edit or delete any other user's content. Future versions could provide more sophisticated content management. For example, an organizational repository might only allow system administrators to publish new topes (perhaps signed with a public key). Likewise, a repository for a highly regulated industry such as banking could provide sophisticated logging features.

In short, the details of the client and the server will vary based on contextual factors. What many repositories will have in common is the need for a fast, accurate search algorithm for finding topes, which is the central focus of this paper.

5. Search algorithms for finding topes in repositories

The centerpiece of almost any repository is a search engine that complements simple browsing. An effective search engine will also be a valuable part of a topes repository, since organizations might be expected to have up to several dozen topes [22][23], and it would be a waste of time to have hundreds or thousands of organization employees browsing through the entire repository every time that they need to download a tope. But organization-specific repositories are not the only ones that need a good search engine. In a repository of more "generic" or "universal" topes, such as a repository of topes for the dozens of kinds of data defined by the W3C, a search engine could impact billons rather than just thousands of users. Thus, there is a strong need for a suitable repository search engine.

5.1. The search problem

When a user wants to reuse an existing tope from a selected repository, he or she would go to the chosen repository and specify a guess at the tope name. Since the user presumably wants a tope because he or she has some strings to validate or reformat, the search engine would also accept examples of these strings. The user might have previously used that repository to retrieve topes, so the repository might have logs about prior uses of topes. Based on these few pieces of information, the repository must attempt to find a tope that meets the user's need. Since search engines often cannot perfectly identify the correct choice, in practice they return a list of search results, and the best choice in the list will ideally appear as the first item in the list.

In other words, the user presents a query with *three components* (where < > indicates a vector):

$q = < K, E, R >$

K is a set of keywords, E is a set of example strings, and R is a set of recently-used topes. The problem is to design an algorithm that will reliably compute a search result s from q (where [ ] indicates a sorted list):

$s = [\tau_1, \tau_2, \ldots \tau_n]$

5.2. Problem transformation

The candidate search algorithms will be easier to explain below if the query is first transformed to an intermediate form. This transformation takes a query q and produces a new query q' that is a vector of five subvectors:

$$q' = < < k_1, \ldots, k_T >, < w_1, \ldots, w_T >, < c_1, \ldots, c_T >, < i_1, \ldots, i_T >, < r_1, \ldots, r_T > >$$

Here, *T* equals the total number of topes in the repository. Each entry in each part of q' (such as c_2) will be referred to as a *feature* of the query, defined as follows.

Keyword features k

Each feature k_τ in the first part of q' records how many keywords in the query are matched by tope τ:

k_τ = number of keywords in K that are matched case-insensitively by the name or folksonomy tags for tope τ

For example, if the keywords are "phone number", then a tope named "phone number" would have k_τ of 2, while a tope named "phone" would have k_τ of 1, and a tope named "part number" would k_τ of 1.

Whitelist features w

Each feature w_τ in the second part of q' records how many examples in the query are matched by the whitelist of tope τ:

w_τ = # of examples in E that are case-insensitively matched by the whitelist of tope τ

For example, suppose that the examples provided by the user are "7-8080" and "258-9979". Then a tope with a whitelist containing "7-8080", "258-9979", and "6-7070" would have w_τ of 2. A tope with a whitelist containing just "6-7070" would have w_τ of 0, since neither of the examples appears in the tope's whitelist.

CharSig features c

Each feature c_τ in the third part of q' records how many examples in the query are matched by the "CharSig" of tope τ:

c_τ = # of examples in E that match the CharSig of tope τ

A CharSig records the character content of strings matched by a tope [22]. Specifically, a CharSig records the range of how many digits might occur, the range of how many letters might occur, the range of how many alphanumeric characters may occur, and the range of how many times each other character may occur. For example, if a phone number tope had formats to match values like "541-737-5572" and "541.737-5572", its CharSig would be 10 digits, 10 alphanumerics, 0-2 hyphens, and 0-2 periods. Previous work has shown how to compute the CharSig of a tope based on its data description [22].

So, for example, suppose that the examples provided by the user are "7-8080" and "258-9979". Suppose that a certain tope allowed 5-10 alphanumerics, 5-10 digits, and 1 hyphen; this CharSig matches both examples, so the tope would have c_τ of 2. Suppose that another tope only allowed 5 alphanumerics, 5 digits, and 1 hyphen; this would only match one example, so the tope would have c_τ of 1.

Isamatch features i

Each feature i_τ in the fourth part of q' records how well all of the examples in the query are matched by the validation function of tope τ:

$$i_\tau = \prod_{e \in E} \phi_\tau(e)$$

When a string poorly matches all of the formats in a tope, then all of the isa functions return values close to zero, resulting in a low value for the validation function. Whereas other features in q' are computed using sums, the validation function values are multiplied to generate i_τ features. The purpose for this multiplication is to support a

minor performance optimization when many example strings are present: if the product falls below a certain threshold (0.01), then the computation is terminated and i_τ rounded down to zero.

Suppose that the examples provided by the user are "7-8080" and "258-9979". If a tope returned a validation score of 1.0 for both examples, then it would have an i_τ of 1. If a second tope returned validation scores of 0.9 and 0.9, then it would have an i_τ of 0.81. If a third tope returned 0.9 and 0.0001, then the product of these would be rounded down to an i_τ of 0.

Recent-use features r

Finally, each feature r_τ in the fifth part of q' records whether each tope τ appears in the user's recent-use set:

$r_\tau = 1$ if $\tau \in R$, otherwise 0

For example, if a user's recent-use set R contained a "part number" tope and a "taxpayer id" tope, then those topes would have r_τ of 1, while all other topes in the repository would have r_τ of 0.

Notice that if a user is looking for τ, then τ presumably does *not* appear in the user's recent-use set, so this r_τ would equal 0 rather than 1. Instead, the values for r_i should hypothetically be 1 for topes that are *related* to the desired tope.

5.3. Implementation details

Initializing the query vector does not actually require loading all topes from the database. Instead, the query is represented as a sparse vector with entries equaling zero unless initialized otherwise, and except in the case of initializing the isa features i_τ, all non-zero vector entries are initialized solely through indexes rather than by loading topes from the database.

To determine k_τ values, a standard inverted index maps from keywords (case-insensitively) to tope identifiers in the repository database. These topes' entries in the sparse query vector are then incremented for each keyword matched.

Likewise, an inverted index is used to map from whitelist entries (case-insensitively) to tope identifiers. These topes' entries w_τ in the sparse query vector are then incremented for each example matched to the whitelist.

CharSigs make it possible to index topes based on their character content [22]. At runtime, identifiers are selected for the topes could contain the characters demonstrated by each example. For each character class `cc` (digits, letters, alphanumerics, and each other character) and each `i` in the range 0-10, an index records the set $S^{\mathtt{i}}_{\mathtt{cc}}$ of topes that could have i instances of character class `cc`. It also records $S^{\infty}_{\mathtt{cc}}$, indicating topes that could have 11 or more instances. These sets are intersected to identify topes that could possibly match each user-specified example. For instance, to match the phone number "541-737-5572", it is only necessary to increment the query entries c_τ for topes in $S^{10}_{\text{digit}} \cap S^{10}_{\text{alnum}} \cap S^{2}_{-}$

Moreover, pre-filtering with the CharSig index helps to reduce the number of topes that must be loaded in order to compute the isa feature entries i_τ. More precisely, suppose that for a string s, some tope's validation function ϕ returns a non-zero value. Then the tope must accept the combination of characters contained in s, which implies that the tope must also be in the intersection of $S^{\mathtt{i}}_{\mathtt{cc}}$ sets for that string. Therefore, if a tope does *not* appear in the intersection of $S^{\mathtt{i}}_{\mathtt{cc}}$ sets, it is safe to conclude that $\phi(s) = 0$ (and therefore that $i_\tau=0$), without ever loading the tope from the database.

Finally, the r_τ features are simply initialized by keeping a log of the recently used topes for each user. Different approaches can be used for determining when topes should be "aged" out of these sets.

5.4. Candidate search algorithms

The central hypothesis explored in this paper is that it is possible to accurately search for a tope based on the three query components—keywords, examples, and recent-use set—as represented in the five classes of query features. Several algorithms were designed in order to assess the marginal value of each query component, as well as to explore the tradeoff between algorithm complexity and search accuracy.

Baseline algorithm (BASE)

The Baseline algorithm (BASE) is a collaborative-filtering extension of the algorithm originally developed for searching a single user's computer [22]. The original algorithm sorts all topes on the user's computer based on their k_τ features, then uses w_τ features to break ties, then uses c_τ features to break remaining ties, then uses i_τ features to break any remaining ties. The original algorithm makes no use of the r_τ query features, since this original algorithm only was meant to search a single computer (as sharing of topes was not of concern when the algorithm was designed).

BASE extends this single-user algorithm by using a "tope relatedness" collaborative filtering model to break any ties that were left after using the k, w, c, and i features as in the original algorithm (Table 1). The tope relatedness model is a square matrix **M** with one row and one column for each tope in the repository. Each cell M_{xy} is proportional to the number of people who have used both x and y (normalized so each column sums to 1). This is essentially the same model that Amazon.com uses to make statements like, "Customers Who Bought Items in Your Recent History Also Bought…" [13]. Here the "items" are topes that are "bought" by making downloads. The model is driven by implicit item ratings (reflected in downloads) rather than explicit ratings specified by users, which eliminates the need for users to expend extra effort for training the system.

In practice, BASE is configured with **M** based on a log of download events for every user. Later, each query identifies a user's recently-used topes. BASE multiplies **M** by the vector $<r_1, \ldots, r_T>$ to compute a vector showing

what topes are related to the person's recently-used topes. BASE uses the cells of this resulting vector to break any ties that were left by the original algorithm.

Rank Addition algorithm (RADD)

The Rank Addition algorithm (RADD) resembles BASE, except that it applies no arbitrary dominance ordering among the query features. Specifically, this algorithm uses the five classes of query features to generate five separately sorted lists of topes. Each tope receives a certain number of "points" inversely proportional to its rank in a list, and RADD then sorts topes based on total points garnered (Table 2).

Essentially, the five lists each indicate how strongly each class of query feature "argues" or "votes" in favor of returning each tope. For example, suppose that sorting topes according keyword matches produced a list $[\tau_1, \tau_2]$, sorting according to recent use produced $[\tau_2, \tau_3, \tau_4]$, and no topes matched any example strings. Then τ_1 would receive $^1/_1 = 1$ point from being first in the keyword list; τ_2 would receive ½ point from being second in the keyword list and 1 point from being first in the recent-use matches, yielding 1.5 points and beating out τ_1. Topes τ_3 and τ_4 would receive $^1/_2$ and $^1/_3$ points, respectively. The resulting list would be $[\tau_2, \tau_1, \tau_3, \tau_4]$. (Iif several topes are tied in a list, their ranks are averaged so they receive equal points.)

Machine-learning-based algorithm (MACH)

MACH delegates the generation of search results to a supervised machine learning model (Table 3) [3]. That is, MACH is first configured with a set of example queries along with the "correct" answers for these queries. A machine learning algorithm then determines how to accurately map from queries to answers. In practice, training queries could be obtained by observing what topes a user eventually chooses to download after performing a query and browsing through the results. Thus, the BASE or RADD algorithms could be used to bootstrap MACH.

The machine learning model used by MACH is a Support Vector Machine (SVM) model [3]. The features in a transformed query q' serve as machine learning features, and the target tope serves as the target label that SVM learns. SVM was selected for three reasons. First, SVM implementations not only learn a "best" search result, but

they also can return a value between 0 and 1 to indicate how closely other options came to being selected as the best result. For MACH, this means that the SVM model can map a query q' to a list of topes [τ_1, τ_2, … τ_n] directly. The second reason why SVM was selected is because it is one of the most successful models and has proven particularly useful for collaborative filtering problems, especially when tested in the face of data containing the sparsity and other imperfections typical of real-world data [5][10]. Finally, algorithms exist for incrementally retraining the SVM model when new data arrive [8]. Therefore, while the prototype implementation described below makes use of a simpler offline SVM library, these incremental algorithms could be used in the future to efficiently maintain an accurate model as new topes are uploaded to the repository and as users perform downloads.

6. Search evaluation

As framed in Section 5, a user query provides three components: keywords, example strings, and an implicit recent-use set of topes. In order to assess the marginal value of each component, as well as to explore the tradeoff between algorithm complexity and search accuracy, the three candidate algorithms were tested with sample queries on a repository of topes for the most common kinds of data. Five questions framed the investigation:

- How does accuracy vary by algorithm? (Section 6.2)
- How does accuracy vary when query components are missing? (Section 6.3)
- How does accuracy vary based on example string features used? (Section 6.4)
- How does accuracy vary when queries provide multiple example strings? (Section 6.5)
- How does speed vary by algorithm and query features? (Section 6.6)

6.1. Method

The algorithms were evaluated on a hypothetical task in which a user wanted to reformat a column of data in a spreadsheet. The column would contain example strings beneath a header cell indicating a set of keywords. The user might have recently used topes for other columns in the spreadsheet. The user's hypothetical search would be served

from a single repository containing topes for all of the most common kinds of data—representing a "worst case" kind of scenario, in the sense that this test task mixes all topes together into one repository, rather than separating topes into several domain-specific repositories.

For this experiment, sample data were extracted from the EUSES Spreadsheet Corpus [7]. Analyses in prior work identified the 32 most common kinds of data, and topes had been implemented for each of these [23]. For each tope, 100 columns were randomly selected that were known to match that tope.

Each column was used to generate a query, in order to test how well each search algorithm could pinpoint the correct tope. Specifically, the query's keyword set K was generated from the column's first cell, the example strings E were generated by randomly selecting cells from the rest of the column, and the recent-use set R was generated from the other topes known to be present in the spreadsheet. In each column, either one or two keywords were typically present. Except where otherwise noted below (Section 6.5), each example set was populated with one string. In general, one or two other topes were present in other columns; generating R in this manner essentially tests how well the search algorithms could find the correct tope for a spreadsheet column after all of the other columns had already been labeled with their correct tope (though see discussion in Section 6.3).

Several configurations were tested with the 3200 queries, in order to answer the questions outlined above. All algorithms require a training phase, in order to build up the tope relatedness model (the matrix **M** in the case of BASE and RADD, or the SVM model in the case of MACH). Therefore, the usual tenfold validation common to machine learning research was used [3]. That is, each column served as a training column 9 times and a test column 1 time, for a total of 3200 test queries. In particular, each algorithm was tested with subsets of the query features. For example, each algorithm was tested in a configuration where R was omitted, essentially testing how well the search algorithms could find the right tope when no other columns had yet been labeled. This provides an upper and a lower bound on the accuracy that could be expected when *some* of the other columns had already been labeled.

Accuracy was judged using the standard machine learning measure of recall, which equals the number of queries that generate the right answer divided by the total number of queries. Recall is equal to the true positive rate. It rises as result sets grow in size and are therefore more likely to include the right answer. However, in our experiment, there is only one right answer for each query, so once the result set has already grown to include this correct answer, increasing it further by 1 item also increases the number of false positives by 1. Therefore, in order to assess this tradeoff between recall and result set size, Section 6.2 tests the recall of each algorithm for several different result set sizes. Then, subsequent subsections 6.3 through 6.5 compare the recall of different model variants while holding the result set size constant at 5 (so that the configurations are compared with nearly identical false positive rates). The Appendix provides additional charts showing more details of these tradeoffs among recall, false positives, and configurations.

Speed was judged in Section 6.6 using milliseconds per query. The experiment was performed using a single CPU of a 2.40 GHz Intel Core Duo system with 3GB of RAM (far more than enough to prevent swapping). Code was implemented in C#, then compiled and run with Microsoft.NET 3.5 on Windows Vista.

In general, the evaluation emphasizes *relative* differences in accuracy and speed, rather than absolute numbers, since the absolute numbers can be expected to change over time as *repositories grow* or as *new computers* become available. The focus is on the relative strengths of different algorithms and query components, particularly in terms of comparing search-by-match and collaborative filtering features with more traditional keyword-only search.

6.2. How does accuracy vary by algorithm?

As shown by Fig. 8, MACH consistently outperformed RADD and BASE by 10 to 15 percentage points. The latter two algorithms performed nearly equally, indicating that the dominance ordering used by BASE does not really affect accuracy. The outperformance of MACH shows that the accuracy of a tope search engine can be enhanced through machine learning on training queries.

In addition, recall increased noticeably as result sets included more topes, though accuracy began to level out after 3 topes: for each algorithm, the accuracy increased no more than 3 percentage points when result sets increased from 4 to 5 topes. In particular, MACH leveled out at approximately 95%. These results indicate that for this small repository, generating short 5-item result sets is sufficient for identifying the right tope for 19 out of 20 queries.

6.3. How does accuracy vary when query components are missing?

It is important to assess how much accuracy drops when query components are missing, since there are some situations where keywords, example strings, or recent-use data might be unavailable. For example, the recent-use set R would be missing in a young repository before many downloads have been performed (commonly referred to as the "cold start" problem). Also, in the context of the hypothetical task, perhaps the user might not have reformatted any other columns with a tope—this could be the first column in the spreadsheet. Keywords and example strings might be missing from other queries for various reasons.

As shown by Fig. 9, accuracy slipped by only a few percentage points when a query component was missing. The KER bars in this figure show accuracy when all three components were included, with an average of 90% over all three algorithms. The ER, KR, and KE bars correspond to queries missing one component; these generally rise to within 10 percentage points of the corresponding KER bars. Results do not drop sharply until two components are missing (especially when R is missing). These results suggest that tope search engine results did not deteriorate rapidly when only two out of three query components were present.

Fig. 9 also shows that the recent-use and example string components are each somewhat more useful than the keyword component. Averaged, the R bars are 75%, the E bars are also 75%, and the K bars are only 55%. In fact, the ER bar for MACH is 94%, scarcely lower than the 95% achieved in the presence of all three query components—that is, for the best available algorithm, keywords offered negligible additional information beyond that of the recent-use and example string components. Consequently, tope search engines that incorporate recent-use

and example string data are likely to be much more effective than a traditional search engine that works based on keywords or tags alone.

6.4. How does accuracy vary based on example string features used?

As shown by Fig. 10, the most useful query features populated from example strings were the isamatch features i, followed by CharSig matches c and trailed by whitelist matches w. Averaged over all three algorithms, the i features alone achieved 65% accuracy, the c features achieved 47%, and the w features achieved only 36%. This indicates that whitelist and CharSig features each, alone, are likely to be far less useful for searches than doing just isa checks.

On the other hand, none of these feature classes was essential if the other two were still provided to MACH. The wc, wi, and ci bars rose to within 2 percentage points of the wci bar (at 81%), indicating that two out of three classes of features together provided approximately the same information as all three put together. Therefore, any one class of these features (such as the isa checks, as in Section 6.5) could be omitted without greatly reducing the accuracy of the best available algorithm.

6.5. How does accuracy vary when queries provide multiple example strings?

Fig. 11 shows that using additional example strings helps to increase accuracy a bit. Specifically, with just w and c features populated from the query, increasing the number of examples to 5 raises SVM's accuracy by 14 percentage points and the other algorithms' by 8 percentage points—though RADD and BASE obtain most of this improvement by the time that the third example is provided. These results suggest that there is some benefit to obtaining 4 or 5 example strings, but there is little additional benefit to be gained by bothering the user to provide still more examples.

6.6. How does speed vary by algorithm and query features?

Performing a query requires two steps. First, the query features (k, w, c, i and r) are populated from the query components (K, E and R). The cost of this feature-population is the same regardless of the search algorithm. Second, a search algorithm uses the query features to find topes in the repository.

Step 1: Populating features

As shown by Table 4, populating the isamatch features i required far longer than the other features. For example, populating all of the CharSig features c required 0.08 ms (total, for all the c features together); it was even faster to populate the keyword, whitelist, and recent-use features. These four classes of features are populated using indexes, but populating i features requires testing example strings against topes. Internally, the current tope isa implementation uses a relatively slow context-free parsing algorithm (GLR) [21]. These results imply that an area for future optimization would be to improve this parser implementation. In the meantime, omitting the i features could offer an enormous improvement in search performance, since the other four classes of features can be populated in a total of only 0.13 ms.

Step 2: Looking for topes

As shown by Table 5, looking for topes in the repository is generally much faster with BASE or RADD than with MACH. For BASE and RADD, rows of this table can be added to fairly accurately estimate the time required to run the algorithm on queries that contain more than one class of feature. In contrast, MACH requires a fixed cost of approximately 1 ms to initialize an SVM query, and the remaining execution time is then linear in the number of features. The bottom line is that all three algorithms scale linearly, but the additional accuracy afforded by MACH comes at the cost of a significantly higher scaling coefficient.

7. Discussion

The first key finding was that the Machine-learning-based algorithm MACH consistently outperformed the other two algorithms by at least 10 percentage points. This indicates that the accuracy of a tope search engine can be noticeably enhanced through machine learning on example queries. Configuring MACH requires first collecting training data. Thus, it would be desirable to run a tope repository for a time with a simpler algorithm, such as BASE without recent-use features, to collect training data. This data can then be used to train SVM in order to upgrade the search engine to MACH.

The second key finding was that the recent-use set and example strings proved to be more useful than the keywords used by traditional repository search engines. Consequently, tope search engines that incorporate these query components are likely to be more effective than a traditional search engine that works based on keywords or tags alone. Therefore, these are worthy features to be considered for any future tope repositories.

However, the isamatch features were a much more computationally costly way to represent example strings than just whitelist and CharSig features. Moreover, the isamatch features offered no marginal improvements to accuracy over these other example string features when MACH was in use. Therefore, though these features might be included while a repository is in its "training phase" (with BASE in operation), it might be sensible to eliminate them and just represent example strings with whitelist and CharSig features when a repository is transitioned to MACH.

Based on a straightforward analysis of algorithms, all the numbers related to speed are likely to scale linearly with the number of topes in the repository. While further measurements on a larger repository could be performed to empirically confirm this scaling, the numbers above provide a baseline for estimating the time to search a larger repository. For example, searching a repository of 320 topes would require 1.3 ms to populate the non-isa features and only 0.9 ms to query with BASE, for a total cost of approximately 2.2 ms. MACH would take somewhat longer, at approximately 50 ms, though still well below human perception (and, most likely, network latency).

Scalability of accuracy is much harder to ascertain without obtaining a larger collection of topes. Note, however, that accuracy is likely to be just as limited by the quality of data descriptions as by the quantity. Unless if the people creating topes are well-trained and careful, they may populate the repository with many dysfunctional topes (just as other existing repositories are similarly populated with many pieces of dysfunctional user code, e.g., [2]). Therefore, obtaining a truly reliable measure of accuracy's scalability would require acquiring numerous data descriptions implemented by actual users in real world situations. Such a field test is deferred to future work.

8. Conclusions and future research opportunities

This paper has shown that it is possible to quickly and accurately find a tope in a repository based on three pieces of information: keywords, example strings that the tope should match, and a set of recently used topes. In particular, the search-by-match approach proved substantially more accurate than simply searching based on keywords and/or tags as in most existing end-user code repositories. Collaborative filtering further increased accuracy, to a recall as high as 95%. These results suggest that search-by-match and collaborative filtering will support reuse of topes and similar grammars at least as well as many existing code repositories that have already proven successful in practice.

These results reveal a path toward more effectively helping users to publish, find, and reuse visual programs similar to data descriptions. In particular, the recent-use, whitelist, CharSig, and keyword query features could all be generalized beyond just topes to support searching for regular expressions, context-free grammars and similar descriptions of strings. (The isamatch features might not generalize as directly, since unlike topes, these other grammars are binary recognizers.) For example, whitelists could be bootstrapped in a repository of regular expressions by providing a search-by-match algorithm and tracking the queries that users enter prior to downloading a selected regular expression. CharSigs represent the "character content" of strings and could be computed in a straightforward bottom-up fashion for context-free grammars. Therefore, the algorithms presented in this paper should perform just as well for helping users to find and reuse regular expressions and similar grammars.

Beyond the realm of grammars, these results also highlight the value that collaborative filtering and organization-centric repositories can potentially offer in facilitating reuse of end-user code. Collaborative filtering has long played a crucial role in online repositories of non-code artifacts, such as items offered by Amazon.com and movie databases. To date, its usefulness in repositories of end-user code has gone unexplored. Topes demonstrate that code can sometimes be particular to organizations, industries, or other groups of people with domain-specific concerns. Reflecting this domain-specificity in the design of a repository can be extremely helpful for putting reusable code at users' fingertips.

Acknowledgements

The reviewers of this paper provided several helpful suggestions for improving the wording and presentation in this paper; in addition, they recommended adding the figures that are included in the Annex. This work was funded in part by NSF via the EUSES Consortium (ITR-0325273) and by NSF under grants CCF-0438929, CCF-0613823 and CCF-0811610. Opinions, findings, conclusions or recommendations expressed in this material are not necessarily those of the sponsors.

References

[1] A. Blackwell, SWYN: A visual representation for regular expressions, in: H. Lieberman (Ed.), *Your Wish Is My Command: Programming by Example*, Morgan Kaufmann, 2001, 245-270.

[2] C. Bogart, M. Burnett, A. Cypher, C. Scaffidi, End-user programming in the wild: A field study of CoScripter scripts, in: *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, 2008, 39-46.

[3] S. Chakrabarti, *Mining the Web: Discovering Knowledge from Hypertext Data*, Morgan Kaufmann, 2002.

[4] C. Chan, M. Garofalakis, R. Rastogi, RE-Tree: An efficient index structure for regular expressions, in: *International Journal of Very Large Data Bases 12*(2) (2003), 102-119.

[5] K. Cheung, J. Kwok, M. Law, K. Tsui, Mining customer product ratings for personalized marketing, in: *Decision Support Systems 35*(2) (2003), 231-243.

[6] R. Fielding, *Architectural Styles and the Design of Network-based Software Architectures*, PhD Thesis, Department of Information and Computer Science, University of California - Irvine, 2000.

[7] M. Fisher II, G. Rothermel, *The EUSES Spreadsheet Corpus: A Shared Resource for Supporting Experimentation with Spreadsheet Dependability Mechanisms*, Technical Report 04-12-03, University of Nebraska—Lincoln, 2004.

[8] G. Fung, O. Mangasarian, Incremental support vector machine classification, in: *Proceedings of the Second SIAM International Conference on Data Mining*, 2002, 247-260.

[9] G. Furnas, T. Landauer, L. Gomez, S. Dumais, The vocabulary problem in human-system communication, in: *Communications of ACM 30*(11) (1987), 964-971.

[10] M. Grčar, B. Fortuna, D. Mladenič, M. Grobelnik, KNN versus SVM in the collaborative filtering framework, in: *Proceedings of WebKDD*, 2005, 21-24.

[11] D. Huynh, R. Miller, D. Karger, Potluck: Data mash-up tool for casual users, in: *Web Semantics: Science, Services and Agents on the World Wide Web 6*(4) (2008), 274-282.

[12] H. Lieberman, B. Nardi, D. Wright, Training agents to recognize text by example, in: *Proceedings of the 3rd Annual Conference on Autonomous Agents*, 1999, 116-122.

[13] G. Linden, B. Smith, J. York, Amazon.com recommendations: Item-to-item collaborative filtering, in: *IEEE Internet Computing 7*(1) (2003), 76-80.

[14] R. Miller, *Lightweight Structure in Text*, PhD thesis, Computer Science Department, Carnegie Mellon University, 2002.

[15] A. Monroy-Hernández, M. Resnick, Empowering kids to create and share programmable media, in: *Interactions 15*(2) (2008), 50-53.

[16] B. Nardi, *A Small Matter of Programming: Perspectives on End User Computing*, MIT Press, 1993.

[17] B. Nardi, J. Miller, D. Wright, Collaborative, programmable intelligent agents, in: *Communications of ACM 41*(3) (1998), 96-104.

[18] R. Nix, Editing by example, in: *ACM Transactions on Programming Language Systems 7*(4) (1985), 600-621.

[19] Regular Expression Library, http://regexlib.com/DisplayPatterns.aspx

[20] C. Scaffidi, Unsupervised inference of data formats in human-readable notation, in: *Proceedings of the 9th International Conference on Enterprise Information Systems - HCI Volume*, 2007, 236-241.

[21] C. Scaffidi, B. Myers, M. Shaw, Fast, accurate creation of data validation formats by end-user developers, in: *Proceedings of the 2nd International Symposium on End-User Development*, 2009, 242-261.

[22] C. Scaffidi, B. Myers, M. Shaw, Intelligently creating and recommending reusable reformatting rules, in: *Proceedings of the 2009 International Conference on Intelligent User Interfaces*, 2009, 297-306.

[23] C. Scaffidi, B. Myers, M. Shaw, Topes: Reusable abstractions for validating data, in: *Proceedings of the 30th International Conference on Software Engineering*, 2008, 1-10.

[24] G. Stahl, T. Sumner, A. Repenning, Internet repositories for collaborative learning: Supporting both students and teachers, in: *Proceedings of the 1st International Conference on Computer Support for Collaborative Learning*, 1995, 321-328.

[25] R. Walpole, M. Burnett, Supporting reuse of evolving visual code, in: *Proceedings of the IEEE Symposium on Visual Languages*, 1997, 68-75.

Annex: Tradeoff in the repository between true positives and false positives

As explained in Section 6.1, increasing the size of a result set generally increases the true positive rate (recall) as well as the false positive rate. For this experiment, the false positive rate (FP) can be computed from the true positive rate (TP) as

$$\mathrm{FP} = \frac{\#\text{ of topes included in result set that should not have been included}}{\#\text{of topes in repository that should not be included in result set}}$$

$$= \frac{\text{result set size} - \#\text{ of topes in result set that were correctly included}}{\#\text{of topes in repository} - \#\text{ of topes in repository that should be included in result set}}$$

$$= \frac{\text{result set size} - (\#\text{ of topes in repository that should be included in result set}) \bullet \mathrm{TP}}{\#\text{ of topes in repository} - \#\text{ of topes in repository that should be included in result set}}$$

$$= \frac{\text{result set size} - 1 \bullet \mathrm{TP}}{32 - 1}$$

$$= \frac{\text{result set size} - \mathrm{TP}}{31}$$

Since the result set varied in this experiment between 1 and 5, while TP can only vary between 0 and 1, FP is much more strongly affected by the result set size than by TP. Thus, holding the result set size constant at 5 (as in Sections 6.3 through 6.5) essentially holds FP constant, making it possible to fairly compare different system configurations at a given false positive rate. These considerations also show that result set size is a good proxy for FP (as in Section 6.2).

In order to provide more precise information about the tradeoffs among TP, FP, and configurations, Figures 12 through 15 depict the results of the experiment of Section 6 at a higher level of detail. Plots of TP versus FP are sometimes called “Receiver operating characteristic” (ROC) curves. Each ROC curve in the figures has five points,

each showing to the TP and FP when the result set has a certain size. Note that the points on these curves generally do not line up vertically, since FP is a function of not only the result set size, but also the TP (as derived above).

Fig. 12 is analogous to Fig. 8 (in Section 6.2), with FP used on the horizontal axis rather than result set size. For example, the leftmost point of the MACH curve has a TP of 0.8 at a result set size of 1. Thus, FP = (1-0.8)/31= 0.0065 in Fig. 12. The curves have exactly the same shape as those shown in Fig. 8, with a slight horizontal offset as explained in the preceding paragraph. As noted in Section 6.2, BASE and RADD show essentially identical accuracy.

Fig. 13 shows the tradeoff between TP and FP when different query components are available. As in Fig. 9 (in Section 6.3), it is clear that recall generally increases as more query components are available, and this is generally true for all three algorithms at virtually all values of FP. As mentioned in Section 6.3, it is particularly noteworthy that the recall is extremely low when only keywords are used (as in traditional repositories), and this is again true at essentially every level of FP.

Fig. 14 shows the tradeoff between TP and FP when different features are used to represent the example-strings component of a queries. As in Fig. 10 (in Section 6.4), the tight clustering of the MACH curves is apparent, with a high recall for all configurations except those relying solely on CharSigs (c) or whitelists (w). As long as both CharSigs and whitelists are used, the accuracy of this algorithm is essentially just as high as that of using all three sets of query features (that is, CharSigs c, whitelists i, and isa features i). This is true at every level of FP.

Fig. 15 depicts the relationship between TP and FP when different numbers of examples are used to populate the w and c features (with no keywords or recent-use sets present, as in Section 6.5). For each value of FP, these curves demonstrate approximately equal TP as soon as the number of examples rises to 3 or 4. This indicates that regardless of the level of FP, there is little marginal value to querying with more than 4 example strings, as explained in Section 6.5.

In summary, these figures demonstrate that the major findings from Section 6 are not especially sensitive to FP, and that the results described in this paper are generalizable across result sets that vary in size from 1 through 5.

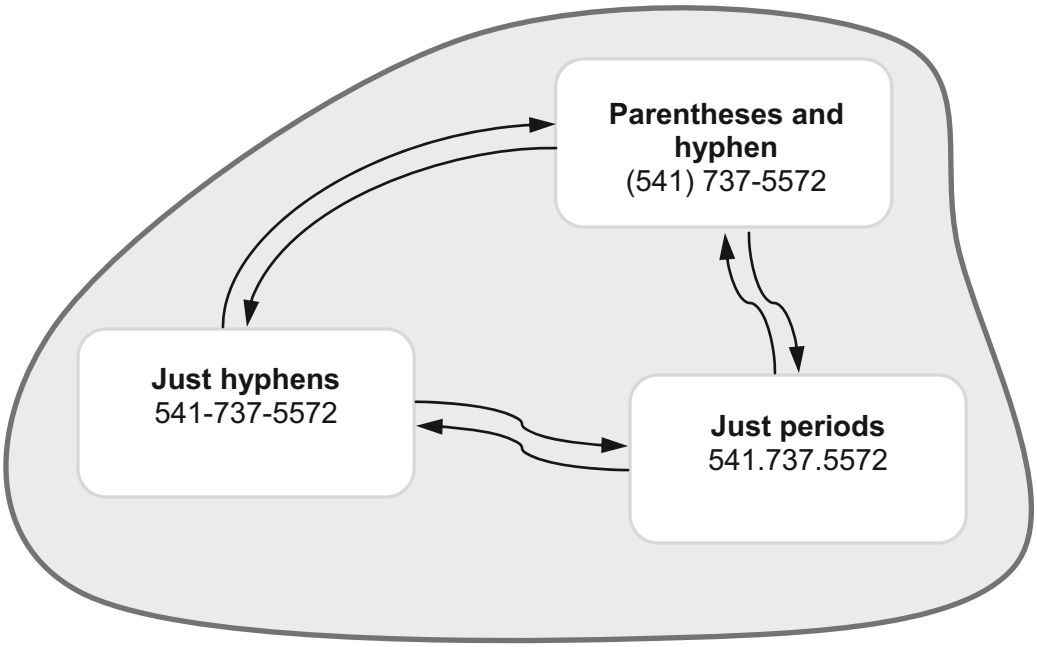
Configured with:	Tope relatedness model **M**
Input:	$< < k_1, \ldots, k_T >, < w_1, \ldots, w_T >, < c_1, \ldots, c_T >, < i_1, \ldots, i_T >, < r_1, \ldots, r_T > >$
Output:	$[\tau_1, \tau_2, \ldots \tau_n]$
Algorithm:	Use k_τ features to sort all topes in repository Use w_τ features to break ties Use c_τ features to break ties Use i_τ features to break ties Use entries in **M** · r to break ties Return the top n topes in the sorted list

Configured with:	Tope relatedness model **M**
Input:	$< < k_1, \ldots , k_T >, < w_1, \ldots , w_T >, < c_1, \ldots , c_T >, < i_1, \ldots , i_T >, < r_1, \ldots , r_T > >$
Output:	$[\tau_1, \tau_2, \ldots \tau_n]$
Algorithm:	Let X = empty set Use k_τ features to sort all topes; add the list to X Use w_τ features to sort all topes; add the list to X Use c_τ features to sort all topes; add the list to X Use i_τ features to sort all topes; add the list to X Use entries in **M**·r to sort all topes; add list to X Let $p_\tau = 0$ for all topes for each list x in X for each tope τ in x $p_\tau = p_\tau + 1/$(rank of τ in x) Sort all topes according to p_τ and return list

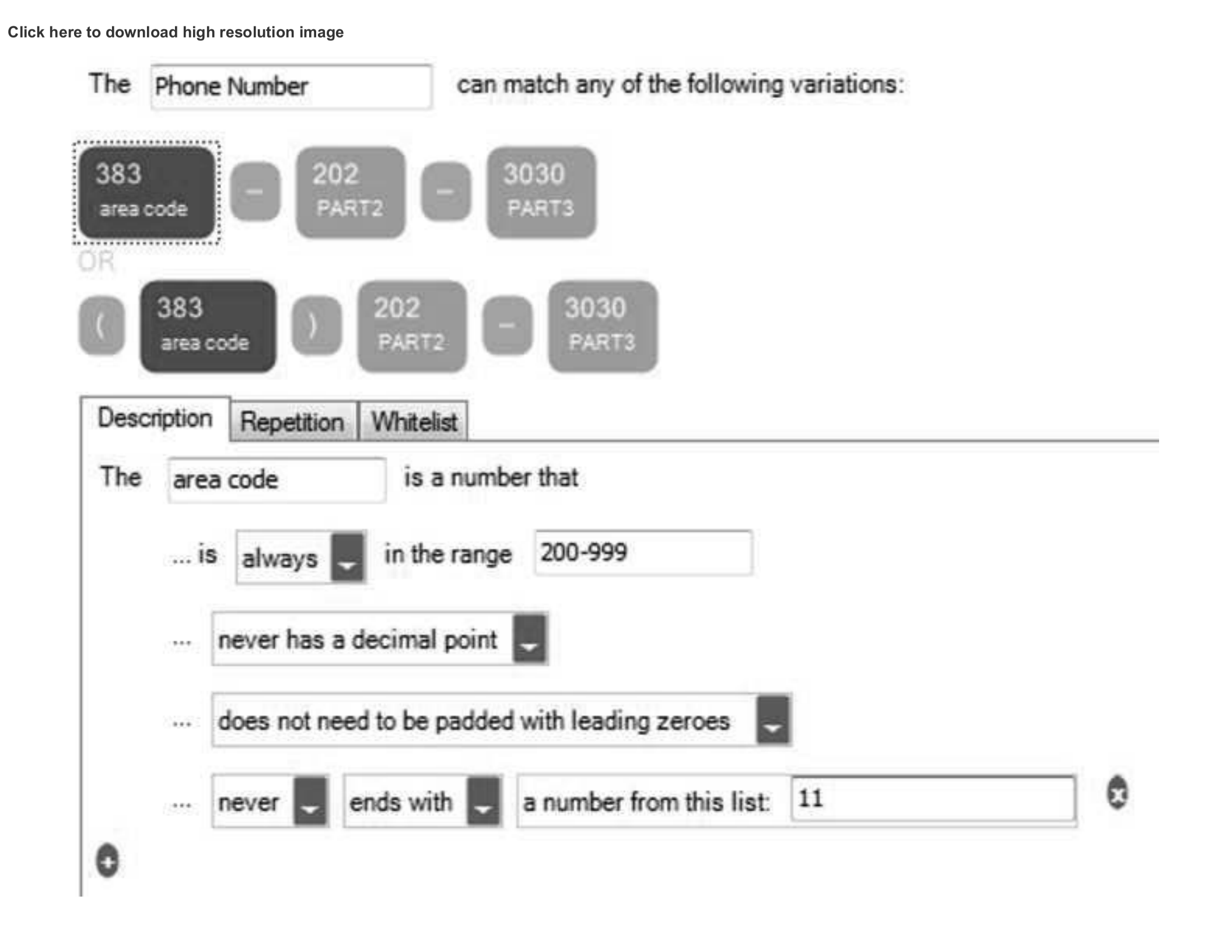
Configured with:	Tope relatedness SVM model
Input:	$< < k_1, \ldots , k_T >, < w_1, \ldots , w_T >, < c_1, \ldots , c_T >, < i_1, \ldots , i_T >, < r_1, \ldots , r_T > >$
Output:	$[\tau_1, \tau_2, \ldots \tau_n]$
Algorithm:	Pass query into SVM model, which returns the output $[\tau_1, \tau_2, \ldots \tau_n]$

	Feature-pop. time (ms)
k	0.01
w	0.01
c	0.08
i	55.98
r	0.03

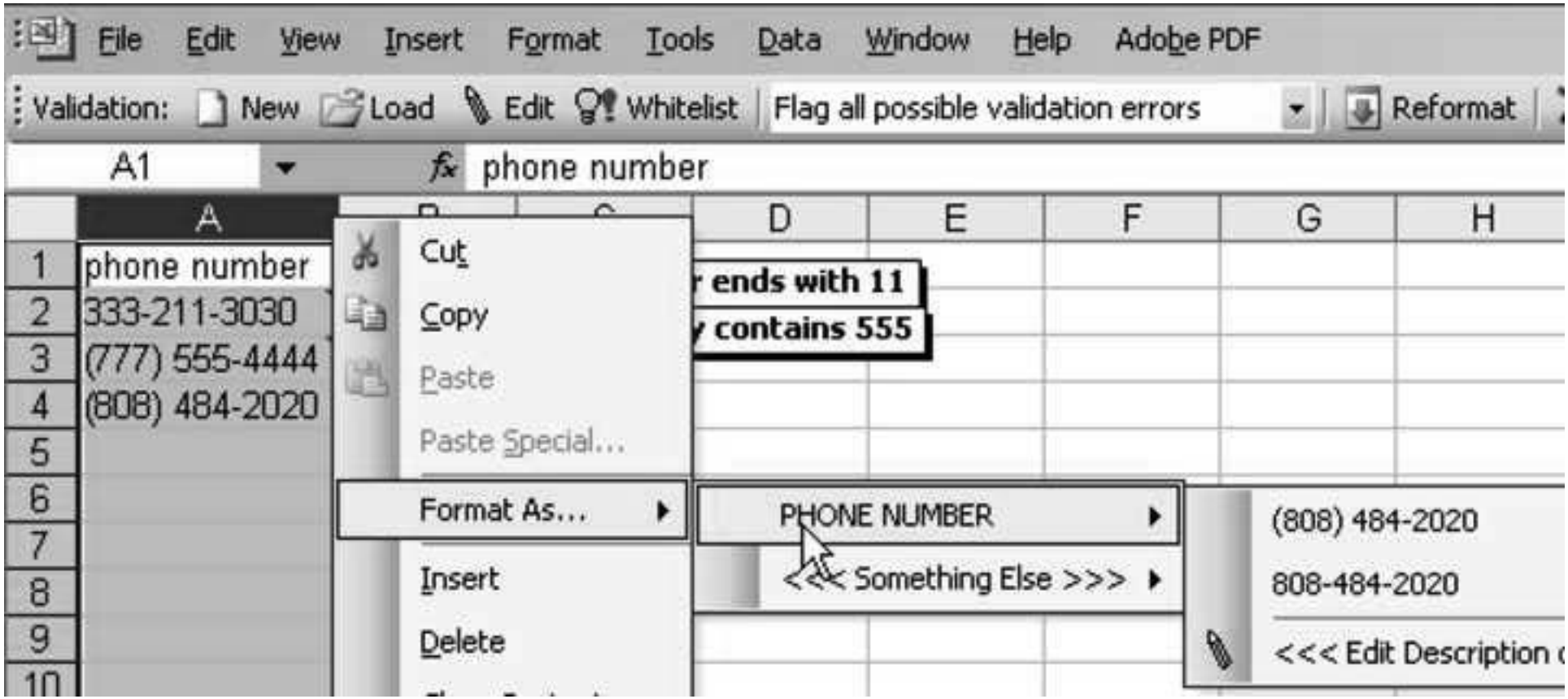
	Time to look for topes (ms)		
	BASE	**RADD**	**MACH**
k	0.01	0.01	2.21
w	0.01	0.01	1.35
c	0.02	0.03	2.53
i	0.01	0.02	2.30
r	0.06	0.13	1.88


Parentheses and hyphen
(541) 737-5572
Just hyphens
541-737-5572
Just periods
541.737.5572

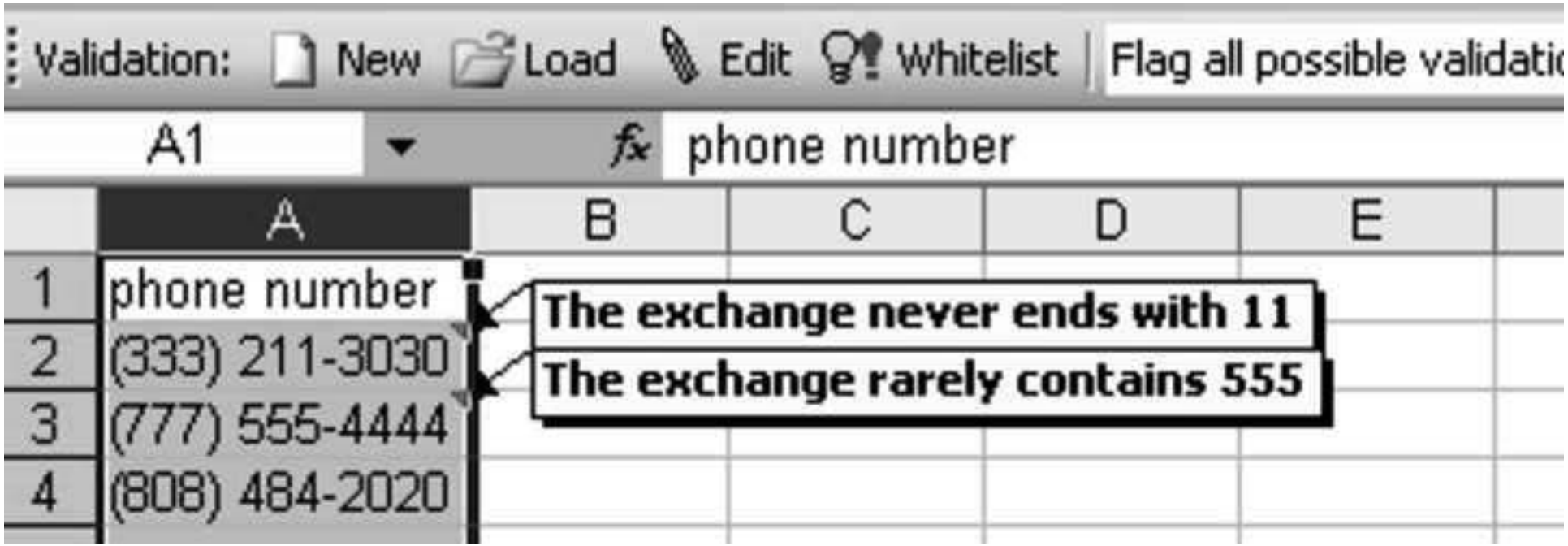
Click here to download high resolution image



Click here to download high resolution image



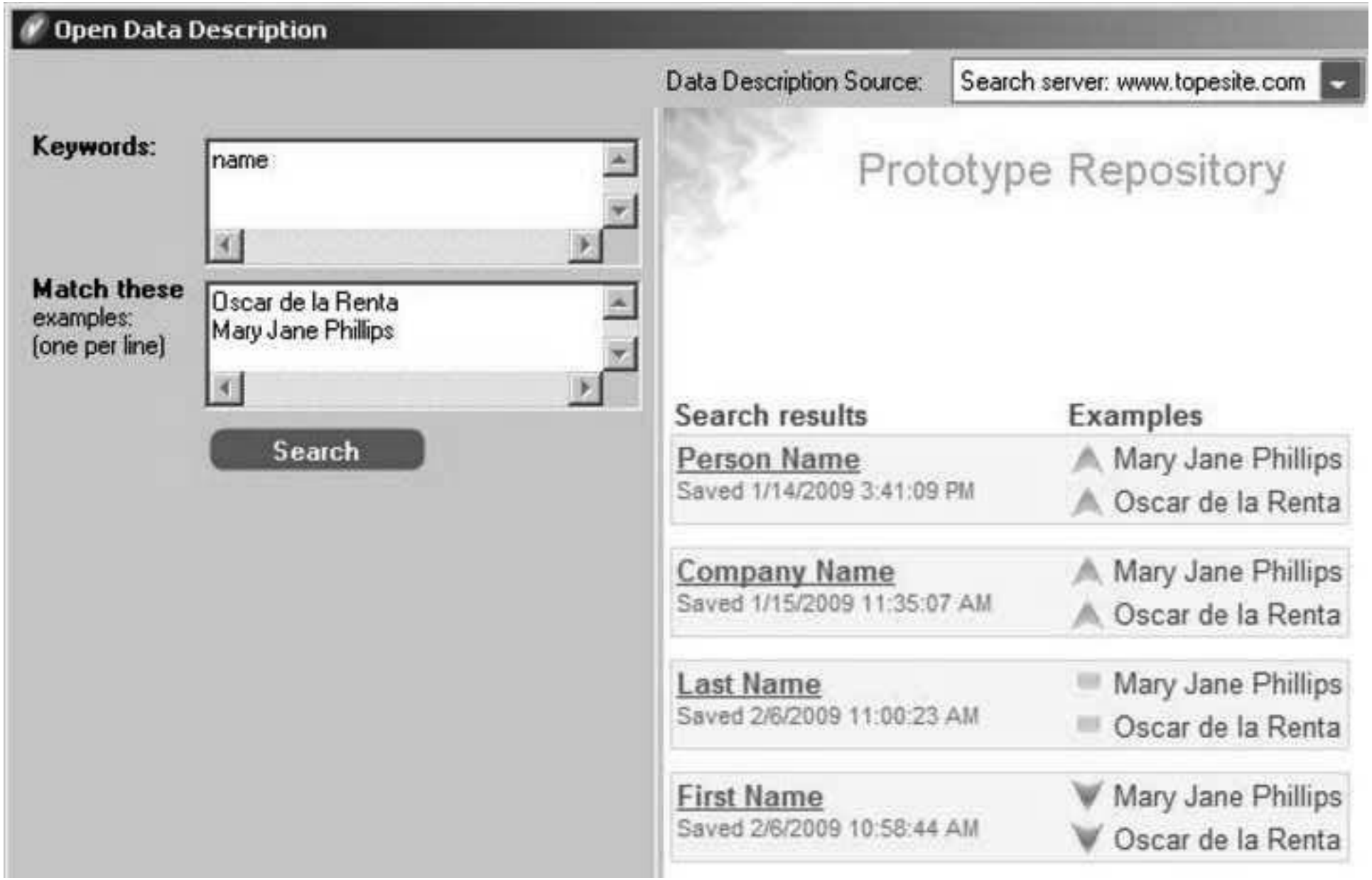
Click here to download high resolution image



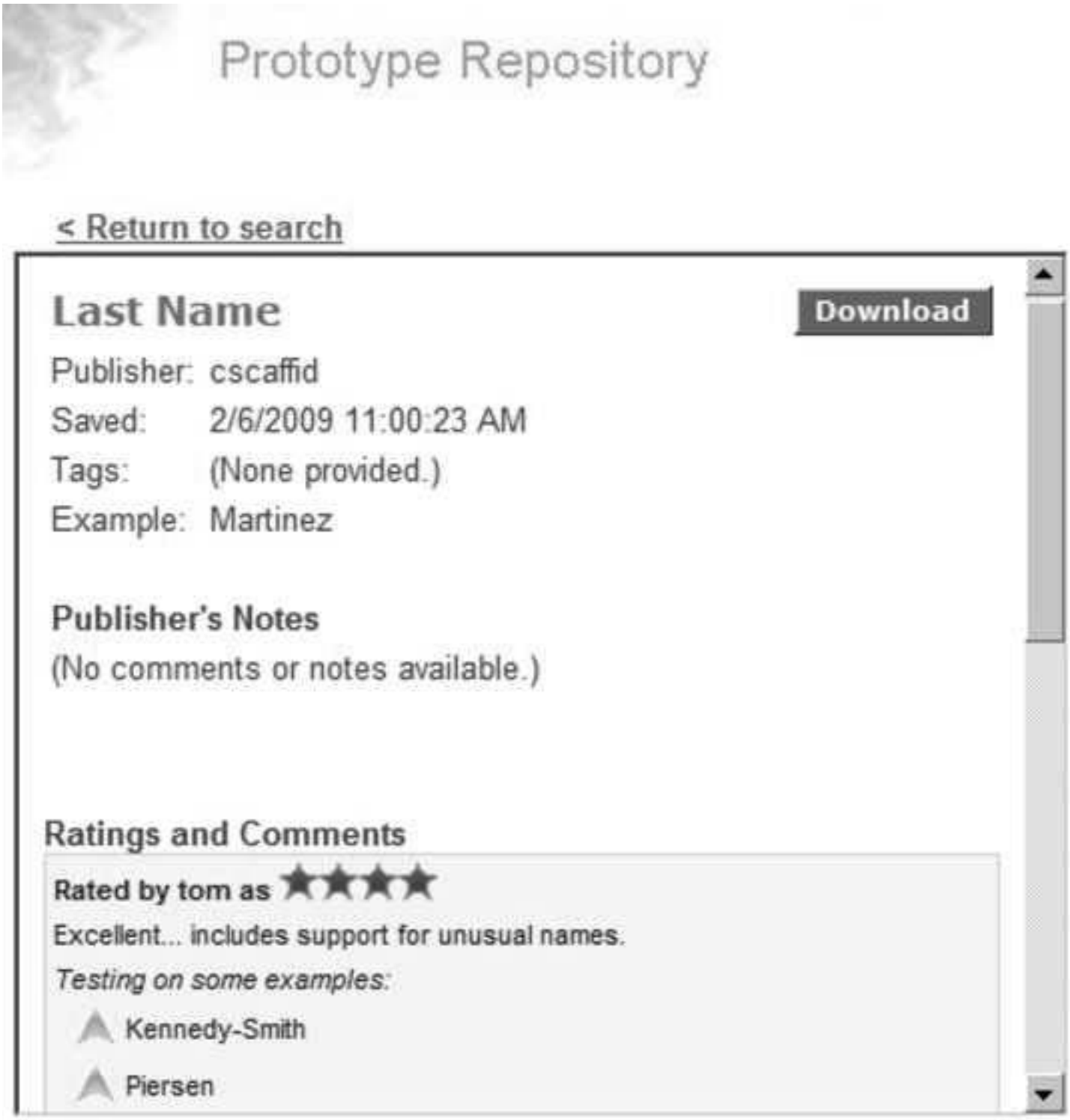

Publish Data Description
Server: data.descriptions.cmu.edu
Publication Status
Not currently published on this server.
Publish to server
Close

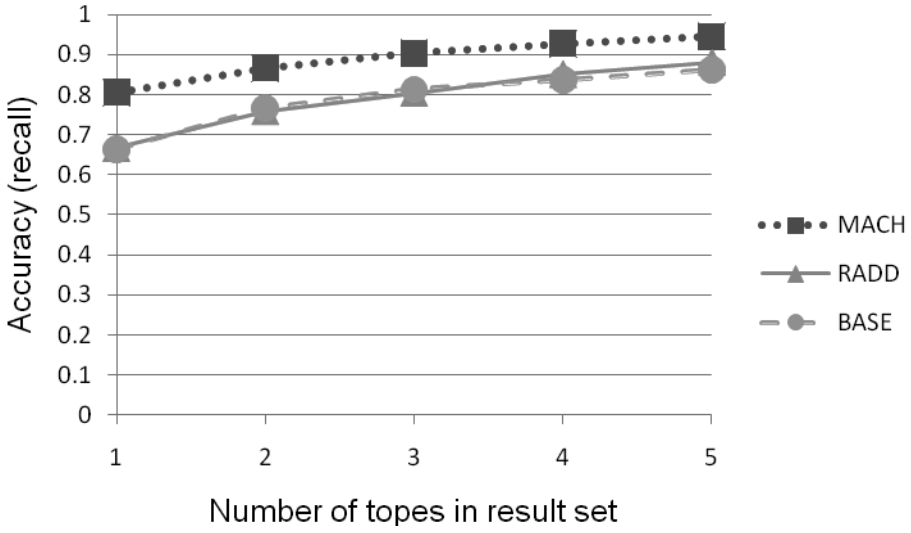

Publish Data Description
Server: data.descriptions.cmu.edu
Publication Status
The version published on the server
was saved 2 seconds ago
Publish updated version
Unpublish from server
Close

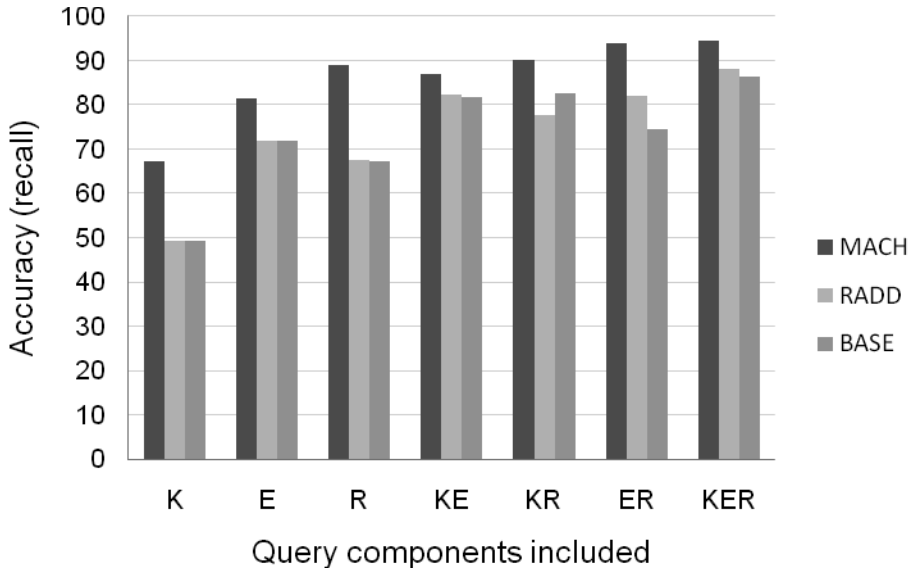
Click here to download high resolution image

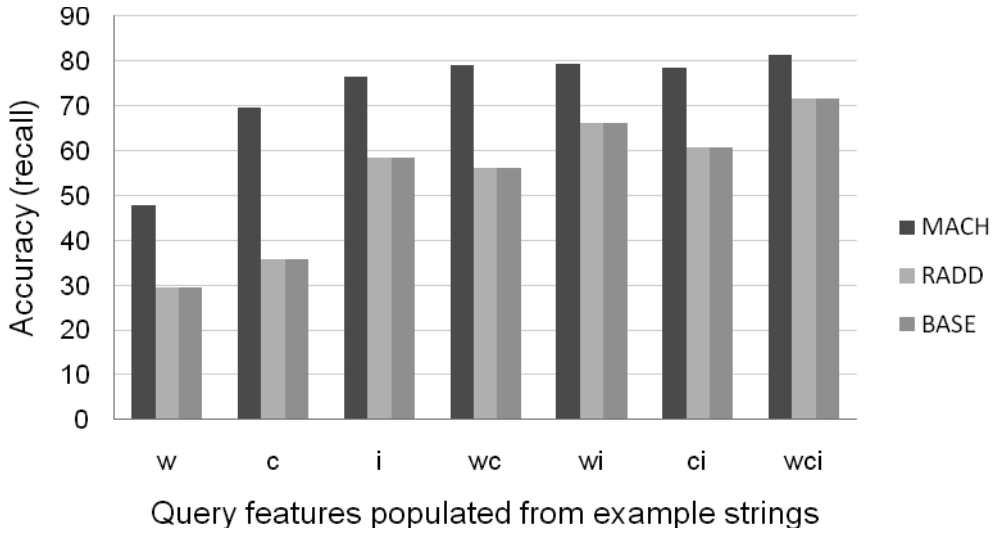


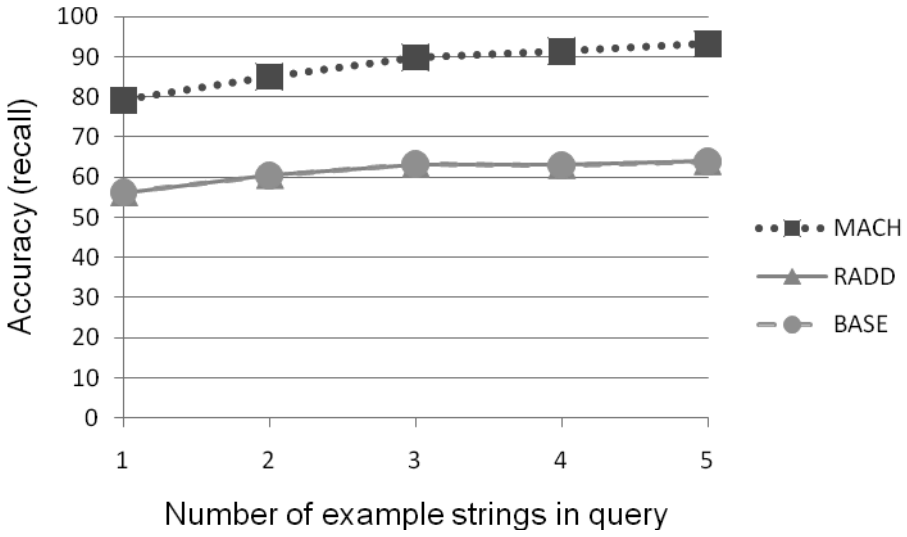
Click here to download high resolution image

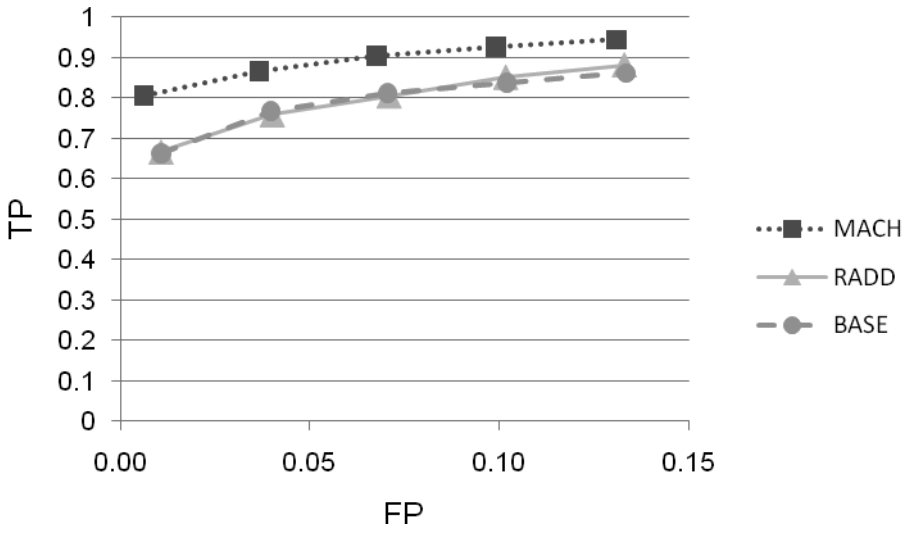


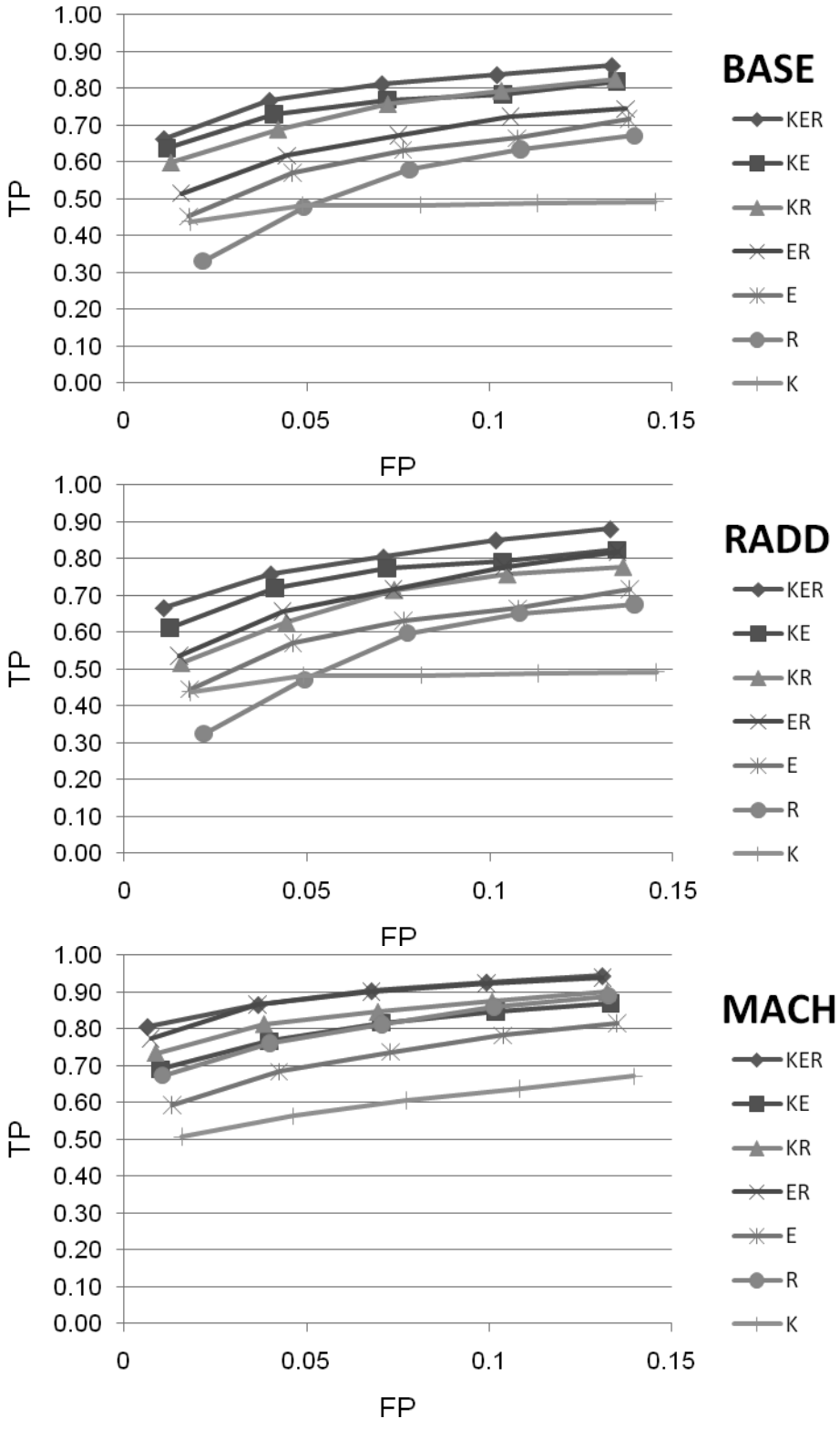

1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
Accuracy (recall)
1
2
3
4
5
Number of topes in result set
MACH
RADD
BASE

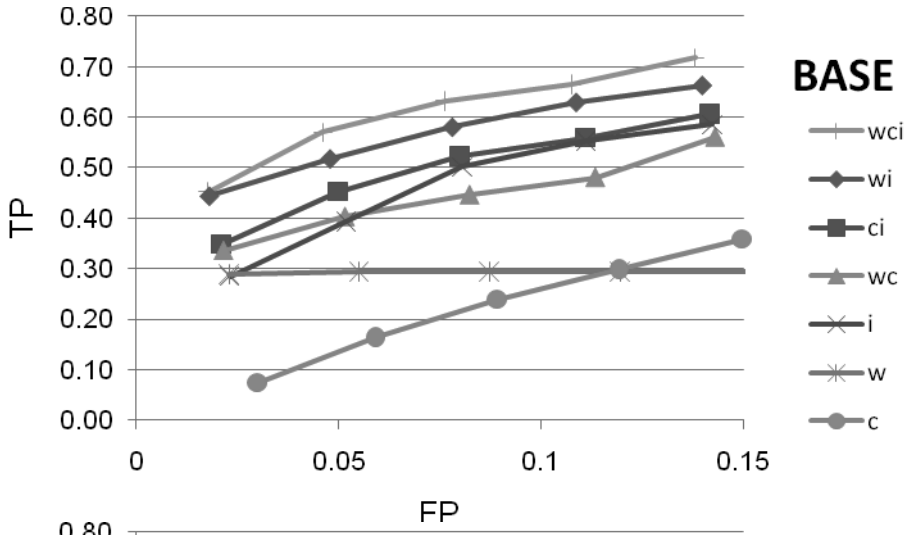

100
90
80
70
60
50
40
30
20
10
0
Accuracy (recall)
K
E
R
KE
KR
ER
KER
Query components included
MACH
RADD
BASE

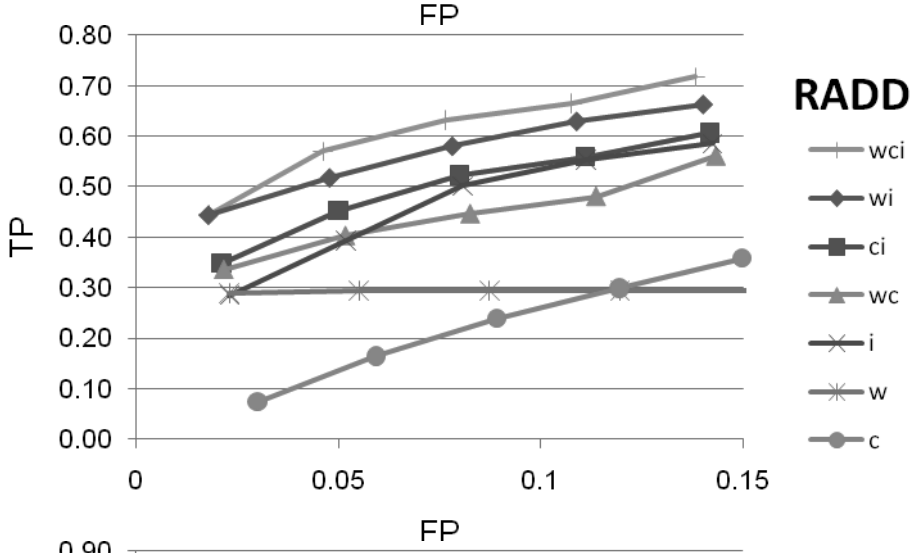

90
80
70
60
50
40
30
20
10
0
Accuracy (recall)
w
c
i
wc
wi
ci
wci
Query features populated from example strings
MACH
RADD
BASE

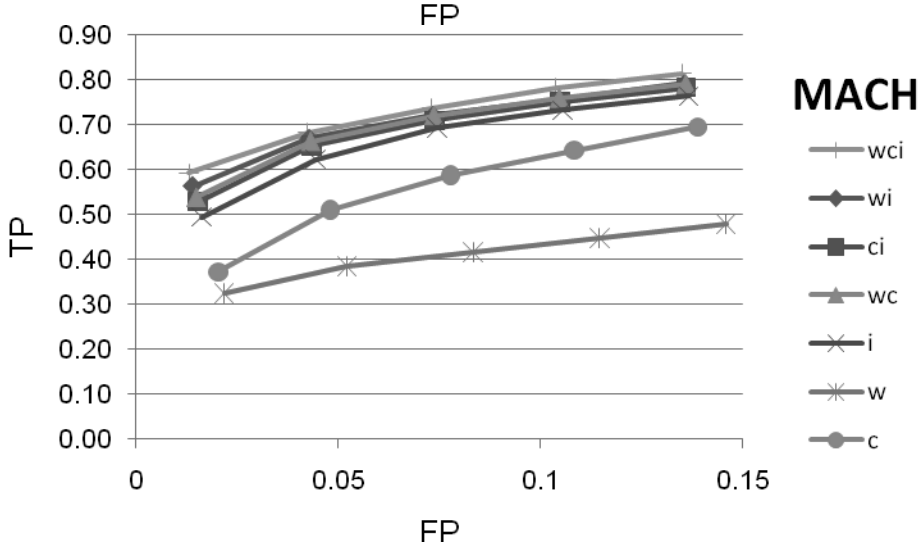

100
90
80
70
60
50
40
30
20
10
0
Accuracy (recall)
1
2
3
4
5
Number of example strings in query
MACH
RADD
BASE

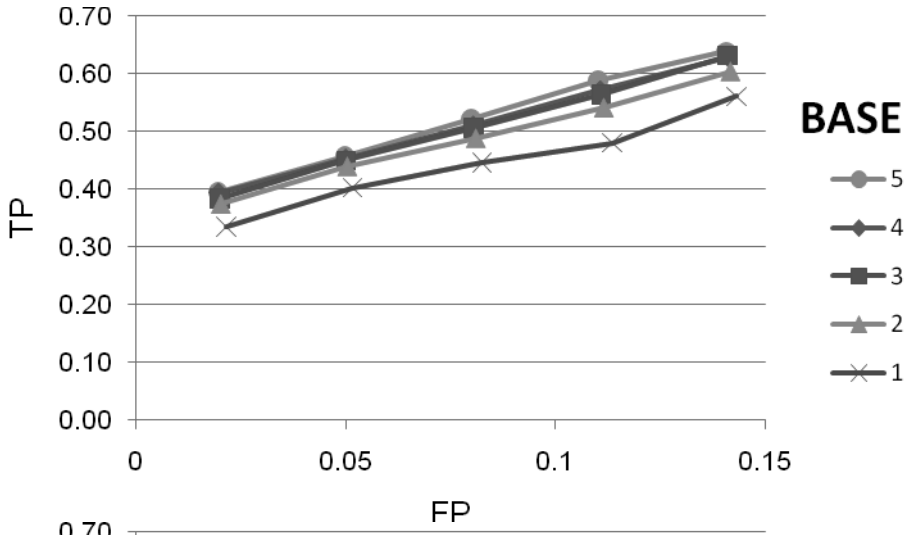

1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
TP
0.00
0.05
0.10
0.15
FP
MACH
RADD
BASE

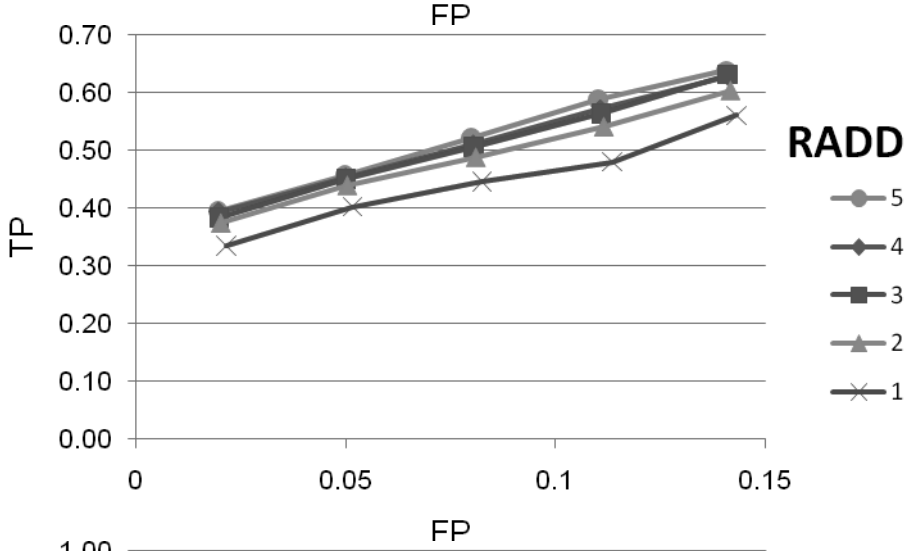

BASE
RADD
MACH
KER
KE
KR
ER
E
R
K
TP
FP
1.00
0.90
0.80
0.70
0.60
0.50
0.40
0.30
0.20
0.10
0.00
0
0.05
0.1
0.15

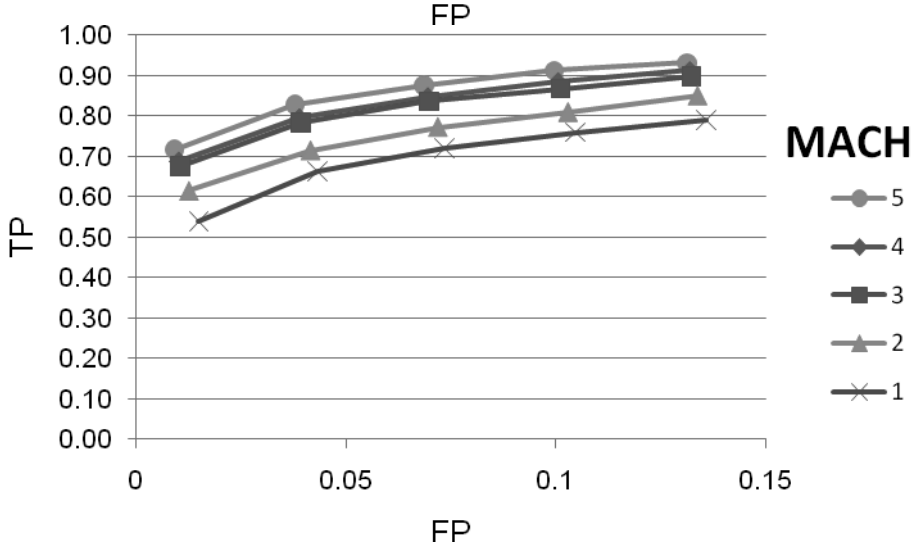

BASE
TP
FP
0.80
0.70
0.60
0.50
0.40
0.30
0.20
0.10
0.00
0
0.05
0.1
0.15
wci
wi
ci
wc
i
w
c


RADD
TP
FP
0.80
0.70
0.60
0.50
0.40
0.30
0.20
0.10
0.00
0
0.05
0.1
0.15
wci
wi
ci
wc
i
w
c


MACH
TP
FP
0.90
0.80
0.70
0.60
0.50
0.40
0.30
0.20
0.10
0.00
0
0.05
0.1
0.15
wci
wi
ci
wc
i
w
c


0.70
0.60
0.50
0.40
0.30
0.20
0.10
0.00
TP
0
0.05
0.1
0.15
FP
BASE
5
4
3
2
1


0.70
0.60
0.50
0.40
0.30
0.20
0.10
0.00
TP
0
0.05
0.1
0.15
FP
RADD
5
4
3
2
1


1.00
0.90
0.80
0.70
0.60
0.50
0.40
0.30
0.20
0.10
0.00
TP
0
0.05
0.1
0.15
FP
MACH
5
4
3
2
1