"Then Click *OK!*" Extracting References to Interface Elements in Online Documentation

Adam Fourney afourney@cs.uwaterloo.ca
Ben Lafreniere bjlafren@cs.uwaterloo.ca
Richard Mann mannr@uwaterloo.ca
Michael Terry mterry@cs.uwaterloo.ca

Cheriton School of Computer Science
University of Waterloo

ABSTRACT

This paper presents a recognizer for identifying references to user interface components in online documentation. The recognizer first extracts phrases matching a list of known components, then employs a classifier to reject coincidental matches. We describe why this seemingly straightforward problem is challenging, then show how informal conventions in documentation writing can be leveraged to perform classification. Using the features identified in this paper, our approach achieves an average F1 score of 0.81, and can correctly distinguish between actual command references and coincidental matches in 93.7% of test cases.

Author Keywords

Named Entity Recognition; Online Documentation

ACM Classification Keywords

H.5.2 Information Interfaces and Presentation: Miscellaneous

INTRODUCTION

The Internet contains a wealth of reference material, tutorials, and other documentation related to the use of interactive systems. Written for the benefit of users, this documentation is expressed in natural language. However, from the perspective of a software system, this documentation is opaque and unactionable.

Recently, the CHI community has demonstrated interest in the problem of automatic identification of references to user interface components within online documentation. For example, work by Ekstrand *et al.* demonstrates how a custom search engine can extract commands mentioned in software tutorials, so that those commands can be listed in the snippets presented as part of enhanced search result pages [1]. Similarly, query-feature graphs, by Fourney *et al.* , pair high-level search query terms with the corresponding user interface elements mentioned in collections of technical documentation [3]. Also closely related is work by Lau *et al.* [4], which has demonstrated the possibility of extracting *action-target-value* triples from how-to instructions, with the goal of advancing machine-guided help systems. In their work, *targets* correspond to interface widgets.

The problem of identifying user interface elements mentioned within documentation is a domain-specific instance of *named entity recognition*. In the context of technical documentation, we are interested in the problem of extracting named entities that refer to interface components, such as commands, menu items, dialogs, settings, and tools within the interface.

In this paper, we introduce a named-entity recognizer for detecting user interface elements mentioned within HTML documents. We call this specific problem *named widget recognition*. In the following sections, we enumerate the specific challenges for this problem, then discuss how certain informal conventions in tutorial writing can be leveraged to better detect named entities in this context. From these conventions, we derive a set of features and a general classification strategy that leads to accurate recognition of user interface elements referenced within text.

CHALLENGES AND RECOGNITION STRATEGIES

Web-based tutorials and documentation use natural language to describe how to perform specific tasks with interactive applications. To be clear and unambiguous, authors typically refer to user interface elements by their given, visible names (i.e., captions or labels) [4, 3]. For example, an author may write that the user should invoke the "Undo" command from the "Edit" menu.

Given this practice, an obvious strategy to named widget recognition is to simply search the documentation for strings matching known widget labels. This tact requires a complete list of widget captions, but numerous, reliable approaches exist for automatically enumerating and extracting the captions of widgets in both web [5] and desktop [3, 6] applications. We refer to this overall strategy as the *baseline* approach to named widget recognition. This baseline approach has been employed in the past by the query-feature graphs work by Fourney *et al.* in [3], and by the enhanced search results work by Ekstrand *et al.* in [1].

While simple and straightforward, the baseline strategy is prone to error. Consider, for example, a tutorial describing various options for converting a colour image to black and white with the GIMP raster graphs software[1]. In the full tutorial, approximately 170 distinct phrases match the labels or captions of components found in the GIMP user interface.

[1] http://www.gimp.org/tutorials/Color2BW/

CHI'12, May 5–10, 2012, Austin, Texas, USA.

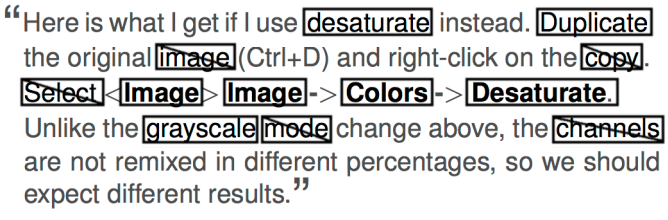

Figure 1. An excerpt from a tutorial describing how to convert a colour image to black and white using GIMP [1]. Substrings matching the names of GIMP commands are outlined in rectangles. False positive matches are crossed out.

However, upon inspection of the tutorial, only approximately 50 phrases are actual references to GIMP operations, making the false positive rate of this approach greater than 70%. An excerpt from this tutorial is listed in Figure 1, with all matches outlined with rectangles and false positives crossed out.

Online documentation also often includes its own set of menus, menu items, and controls for navigating and interacting with the website. These documentation widgets can also generate false matches when employing the baseline approach.

In the following sections, we identify a number of sources of information that can be employed to improve upon this baseline accuracy.

Leveraging Prior Beliefs

In the previous example, the term "desaturate" can be found in the first sentence (Figure 1). This term is highly technical, and rarely occurs in more general writing. Without additional evidence, we would expect that the use of this term refers to GIMP's "Desaturate" command. Conversely, in the second sentence, we encounter the term "image". This term is very generic, and occurs in many contexts beyond GIMP tutorials. Without additional evidence, we assume that it represents a false detection. Thus, by estimating how often a phrase refers to a widget in a given corpus, it is possible to correctly label many named widget references without any further consideration of the context in which the matches occur.

Leveraging Informal Conventions

In sentence 3 of Figure 1, the term "Image" twice refers to actual, named widgets. In such situations, further evidence is needed to overcome the prior expectation that generic phrases do not represent commands. This additional evidence is provided by informal conventions that are often employed in tutorial writing. For example, command names are often capitalized, and are occasionally styled to appear differently than the surrounding text (in this case, with a bold weighting). Special punctuation is also often employed to specify menu hierarchies (e.g., "->" in "File->Open"). The use (or lack of use) of these conventions can also factor into the classification of terms.

Leveraging Page Context

Page context can provide additional evidence to help correctly identify named widgets in cases where the online documentation contains links and website navigation with names matching those of the software it is documenting (e.g., "Help", "About", etc.). Specifically, matches that occur in regions of the page that resemble site navigation reduce the confidence that the matching term directly references a user interface widget.

Given this foundation, we now describe a feature set and a general classification framework that enables accurate recognition of named widgets.

CLASSIFICATION FRAMEWORK

Identifying UI components referenced in web documentation is a three step process. First, the document is examined for substrings matching the names of known interface components or commands (the baseline approach described above) to yield a set of *candidate matches*. In the second step of the process, a set of features is extracted from the context of every candidate match (where the features are derived from the cues and informal conventions discussed above). In the final step, a classifier determines which substring matches are indeed references to widgets.

Feature Extraction

To more correctly classify candidate named widgets, we employ the following set of features.

Capitalization

Recognizing the tendency for authors to capitalize references to commands, our classifier employs two related capitalization features: 1) Whether a candidate match's first token is capitalized, and 2) the total number of capitalized tokens within the match sequence. For example, "Save Selection to File" has three of its four tokens capitalized, including the first.

Next and Previous Tokens

The *next and previous tokens* features record the tokens immediately preceding and following the candidate named widget. These features are designed to model common phrases in which named widgets appear (e.g., "the File menu"). They also model conventions for describing menu hierarchies, such as the use of ">" or "->".

Next and Previous Candidates

Given a candidate match, the *next candidate* and *previous candidate* features record the candidate matches found before and after the current match. All tokens occurring between candidate matches are ignored. These features are designed to recognize and leverage common command sequences (e.g., "Copy and Paste") as well as parent-child relationships expressed in menus (e.g., "Edit > Paste As > New Layer").

Element Occupancy Ratio and Element Type

In web-based tutorials, references to interface elements are often expressed with some styling that makes the text visually distinct from the surrounding text. The *element occupancy ratio* feature models these markup differences as follows. First, we identify the HTML element directly enclosing the candidate match. We then compute the ratio between the number of tokens making up the candidate and the total number of tokens enclosed by the HTML element. For example, the word "pencil" exhibits a $1:4$ ratio in the phrase

" Select the pencil tool ", but a 1: 1 ratio in the phrase "Select the pencil tool". In addition to the element occupancy ratio, we extract a feature recording the *type* of the enclosing element.

Text-to-Tag Ratio and Location (wrt. the Start of the Page)
Tutorial content, and hence named widgets, often make up the "main content" of the enclosing HTML document (as opposed to secondary content such as navigation, headers, and advertisements). In the information extraction literature, the *text-to-tag ratio* has been found to be a good feature for discriminating between main and secondary content [8]. Specifically, main content is typified by regions of the HTML document containing many text tokens, but few HTML tags. These regions are said to have a high text-to-tag ratio. Similarly, main article content is often found in the central region of an HTML file, somewhere between header content and footer content. Thus, we use the *location* of the match within the HTML document to help rule out candidates that are likely part of a tutorial's site navigation.

Notably absent from this list of 10 features is any representation of our prior beliefs regarding the likelihood that a candidate match represents a named entity. Prior beliefs are those held before considering evidence (i.e., features), and are modelled by the classifier directly. We describe the classifier next.

A Naive Bayes Classifier with Witten-Bell Smoothing

The features outlined above are compatible with many modern text classification and information extraction techniques. In this work, we elected to construct a recognizer that employs naive Bayes classification. While potentially less effective than more complex techniques (e.g., [2]), naive Bayes classifiers have nonetheless proven to be effective general purpose classifiers, and are certainly sufficient for demonstrating the feasibility of named-widget recognition. Moreover, naive Bayes classifiers confer a number of unique advantages. First, the number of parameters in a naive Bayes model scales linearly with the number of features and classes [7]. As a result, comparatively less training data is required to produce an effective classifier. Additionally, the "naive" independence assumption affords the ability to independently learn the feature distributions, thus enabling efficient training in the types of distributed systems typical of existing web indexing platforms.

In order to classify commands, we employ a separate binary naive Bayes classifier for each UI component we would like to recognize. When classifying a candidate match, we invoke only the classifier corresponding to that named component. For example, substrings matching the text "File" invoke the classifier corresponding to the system's *File* menu.

By treating commands separately rather than collectively via a single class or classifier, we ensure that the framework is able to directly model the subtle differences in context in which named entities are expected to occur (e.g., enabling the classifier to learn menu structures). Unfortunately, the strategy aggravates the problem of sparse data, and the training data may not include mentions of every available widget. To overcome the sparse data problem, we use Witten-Bell smoothing [9]. Witten-Bell smoothing uses the training data to estimate the likelihood of novel events. It then uses this estimate, together with a more general "backoff" model, to redistribute probability mass, thus filling in missing information. In our case, the backoff model is constructed by pooling the training data for all named entities into a single "generic widget" class. As a concrete example, suppose that the training data does not include any examples of the "Cut" command. With Witten-Bell smoothing, we can use what we know about commands in general to predict how references to the "Cut" command might appear in text.

EVALUATION

To evaluate the accuracy of the classification framework, we trained classifiers for each of the software applications listed in Table 1. In the following sections, we describe how the necessary training data was collected, how the classifiers were evaluated, and report the results of the evaluation.

Generating Training Data

In the course of conducting previous research with query-feature graphs [3], we had previously amassed a corpus of thousands of web documents pertaining to each of the software applications listed in Table 1. These corpora were collected using standard web crawling procedures. To generate training data for a particular application, we randomly sampled 35 documents from the associated document collection. We then identified candidate matches within each document, and manually labeled each as either referring to a command or not. Since terminology varies in generality from application to application (e.g., the names of commands in GIMP tend to be technical), and because we sampled an equal number of pages for each application, the number of labeled examples in each training set differ.

Evaluation and Results

In order to measure the accuracy of the classifiers, we employed leave-one-out validation on a per-page basis: in each round, the classifier is trained with items found in all but one of the web documents, and is evaluated using the withheld document. We report the results of this experiment in Table 1. To provide a point of comparison, the table also presents the accuracy achieved when using only the baseline approach.

The results suggest that the classification framework performs well. The tested classifiers are able to accurately label an average of 93.7% of all candidate named entities, surpassing the baseline accuracy by a wide margin. Viewed as a retrieval problem, the classifier returns an average of 75% of all actual named widgets (i.e., recall), with an average true-positive rate of 87% (i.e., precision). This yields an overall F1 score of 0.81.

To verify the classifiers were modelling the phenomena as expected, we manually inspected the conditional probability tables making up the various naive Bayes classifiers. Through this inspection, we found that the classifiers were indeed modelling various structural aspects of the target interface. For example, our approach automatically learns that the most likely token to follow a reference to GIMP's "Fuzzy Select" command is the word "tool", which is expected since "Fuzzy

	GIMP	Inkscape	Thunderbird	Total
Classifier:				
True +	598	405	445	1448
False +	53	91	71	215
True -	2040	2668	4215	8923
False -	133	181	164	478
Precision	0.92	0.82	0.86	0.87
Recall	0.82	0.69	0.73	0.75
F1 score	0.87	0.75	0.79	0.81
Accuracy	93.4%	91.9%	95.2%	93.7%
Baseline:				
F1 score	0.41	0.30	0.22	0.30
Accuracy	25.9%	17.5%	12.4%	17.4%

Table 1. Evaluation results characterizing the performance of the proposed classification framework. For comparison, the final two rows of the table present performance characteristics of the baseline method.

Feature Set	F1 score
Next and Previous Candidate Matches	0.78
Next and Previous Tokens	0.63
Capitalization	0.63
Text-to-Tag Ratio / Location	0.59
Element Occupancy Ratio / Element Type	0.50
No features (i.e., using the prior distribution directly)	0.50

Table 2. Classification performance when using only the features named in the leftmost column.

Select" is a tool that appears in GIMP's toolbox. Similarly, the words "window" or "dialog" are the most likely tokens to follow references to GIMP's "Channels" window.

Finally, to determine the relative importance of the various features employed by the classifier, we re-ran the experiment with classifiers employing various subsets of the available features. For example, we found that classifiers employing *only* the "Next and Previous Candidates" features achieve an overall F1 score of 0.78 across the three applications. Similarly, the classifiers utilizing only the "Capitalization" features achieve an overall F1 score of 0.63. The difference between these scores reflects the relative importance of the associated features. Of all features, "Element Occupancy Ratio" and "Element Type" were the least effective (F1 score of 0.50), providing less inferential leverage than we would have liked. A summary of the F1 scores for the various feature sets is listed in Table 2.

DISCUSSION AND FUTURE WORK

In this paper, we have demonstrated a system that draws upon informal practices in tutorial writing to detect references to named widgets in online documentation. The resulting named widget recognizer provides the foundation for a number of new interaction possibilities, including: (1) new means of indexing and searching tutorials, (2) the ability for users to invoke commands directly from within tutorial text, (3) the creation of summaries that highlight important steps in long tutorials (i.e., generating "Quick Start" guides), and (4) the automatic generation of macros or macro templates.

As with any complex system, there are a number of limitations of our recognizer worthy of further discussion.

First, our recognizer compares online documentation against a list of known widgets, and disambiguates between meaningful references and spurious/coincidental string matches. Our recognizer is both trained and evaluated using automatically generated lists of all strings in a user interface (e.g., captions, labels, tooltips, etc.). A limitation of this approach is that it does not consider cases where online documentation refers to a widget using alternative text (e.g. "click the 'No' button" when the button is actually labeled "Cancel"). Consideration of such novel synonyms can be expected to lower the recall scores as compared to the results reported in Table 1. This situation can be partially ameliorated by adding common synonyms to the list of widget names, but a more general solution for detecting novel synonyms remains a topic of future work.

Additionally, our system cannot disambiguate between distinct widgets that share a common name. For instance, in the GIMP 2.6 interface, both a menu and a panel share the caption "Layers". Fortunately, this does not significantly impact the applicability of the proposed technique to applications such as improved tutorial indexing or tutorial summarization. Moreover, we can often rely on the user to differentiate between multiple possible interpretations. In the future, we hope to leverage additional context to automatically disambiguate between identically named widgets.

Despite the aforementioned considerations, the proposed recognizer can be immediately applied to existing research, including query-feature graphs [3], the work of Lau *et al.* [4], and the work of Ekstrand *et al.* [1].

REFERENCES

1. M. Ekstrand, W. Li, T. Grossman, J. Matejka, and G. Fitzmaurice. Searching for software learning resources using application context. In *Proc. UIST'11*, pages 195–204, New York, NY, USA, 2011. ACM.
2. J. R. Finkel, T. Grenager, and C. Manning. Incorporating non-local information into information extraction systems by gibbs sampling. In *Proc. ACL'05*, pages 363–370, Stroudsburg, PA, USA, 2005. Association for Computational Linguistics.
3. A. Fourney, R. Mann, and M. Terry. Query-feature graphs: bridging user vocabulary and system functionality. In *Proc. UIST'11*, pages 207–216, New York, NY, USA, 2011. ACM.
4. T. Lau, C. Drews, and J. Nichols. Interpreting written how-to instructions. In *Proc. IJCAI '09*, pages 1433–1438, 2009.
5. G. Little and R. C. Miller. Translating keyword commands into executable code. In *Proc. UIST '06*, pages 135–144, New York, NY, USA, 2006. ACM.
6. V. Ramesh, C. Hsu, M. Agrawala, and B. Hartmann. Showmehow: translating user interface instructions between applications. In *Proc. UIST'11*, pages 127–134, New York, NY, USA, 2011. ACM.
7. S. Russell and P. Norvig. *Artificial Intelligence A Modern Approach*, chapter 20, page 718. Prentice Hall, 2nd edition, 2003.
8. T. Weninger and W. H. Hsu. Text extraction from the web via text-to-tag ratio. *Database and Expert Systems Applications, International Workshop on*, 0:23–28, 2008.
9. I. Witten and T. Bell. The zero-frequency problem: estimating the probabilities of novel events in adaptive text compression. *Information Theory, IEEE Transactions on*, 37(4):1085 – 1094, July 1991.