RJ10438 (A0811-005) November 4, 2008
Computer Science

IBM Research Report

Interpreting Hand-Written How-To Documentation

Tessa Lau, Clemens Drews, Jeffrey Nichols
IBM Research Division
Almaden Research Center
650 Harry Road
San Jose, CA 95120-6099
USA

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

Interpreting Hand-Written How-To Documentation

Tessa Lau, Clemens Drews, and Jeffrey Nichols
IBM Almaden Research Center
650 Harry Rd
San Jose, CA 95120 USA
{tessalau,cdrews,jwnichols}@us.ibm.com

ABSTRACT
Hand-written instructions are a common way of disseminating how-to information. However, studies have shown that written instructions are difficult to follow. Users could benefit from a system that understands hand-written instructions and provides users with assistance in following them. While general natural language understanding is extremely difficult, we believe that understanding should be possible in the more limited domain of how-to instructions. In this paper, we present an investigation of parsing and understanding hand-written instructions. We began by collecting a corpus of instructions for 43 web-based tasks. A qualitative study of these instructions revealed that despite a wide variation in quality, there is a common set of verbs and nouns that are used to describe tasks on web sites. We then implemented and compared three how-to instruction interpreters: one based on keyword matching, one based on a grammar, and one using machine learning. The best of these interpreters achieved 53% accuracy in interpreting instructions in our corpus.

Author Keywords
instructions, how-to, documentation, parsing, interpretation

ACM Classification Keywords
H.5.2 Information Interfaces and Presentation: User Interfaces

INTRODUCTION

Written how-to documentation is everywhere, from the instruction manuals that accompany software packages to FAQ documents available online. Like a cooking recipe, these instructions tell us in step by step order how to accomplish a task. Today, many instructions are distributed in electronic form. For example, in the enterprise, employees are routinely sent instructions via email on how to update their emergency contact information, classify their IT assets, transfer employees into a different division, or search for job openings. In the consumer web space, webmasters are publishing tutorials on how to use their applications.

Unfortunately, studies have found that people have difficulty following written instructions [8, 14]. Some people report difficulty mapping from the written instruction to the target on the screen, and having to systematically search the on-screen interface for the named button. Others lose track of where they are in the instructions, resulting in following instructions out of order, or missing steps entirely. Such problems can be exacerbated if users are distracted in the middle of the tasks and must return to them later having lost most of their context.

While guided help systems such as Coachmarks [2], Stencils [5], and CoScripter [9] partially address this problem, most such systems rely on documentation that has already been authored for use with their system. They cannot take advantage of the vast array of existing how-to literature already available on the web that has been written for humans by humans. If we could automatically parse and interpret these instructions, we could augment these guided help systems with a much broader library of documentation and improve the user experience of how-to instructions.

Ultimately, the problem of understanding hand-written instructions amounts to understanding natural language. However, we believe that most instructional material uses a more structured subset of language that may be amenable to automated processing, particularly if we limit the domain of interest to a specific platform. In the web domain, which is the focus of this paper, much documentation uses a common set of terminology to describe operations, using verbs such as “click” and “type” and objects such as “link” and “textbox”.

In this paper, we formalize the problem of understanding how-to documentation and contrast three different approaches to the problem. We evaluate our results on several datasets of hand-written instructions collected from the internet and email. Our best algorithm, based on machine learning, is able to understand 53% of the usable statements in our dataset.

In summary, this paper makes the following contributions:

- A formalization of the problem of understanding hand-written how-to instructions;
- A qualitative analysis and manually-labeled corpus of hand-written instructions collected from the internet; and
- A comparison of three algorithms for interpreting written instructions automatically.

Submitted for review to IUI 2009.

We begin with a brief overview of related work. We then introduce the data we have collected and present an initial qualitative analysis of the data. Next, we introduce the problem of understanding how-to instructions, and describe our three approaches to the problem. We then report the results of a comparative study of the performance of our three approaches, and discuss the implications for system design. Finally, we conclude with a discussion of future work.

RELATED WORK

Previous studies have illustrated some of the limitations of how-to documentation, and proposed various approaches to fix it. Lau *et al*'s Sheepdog project [8] found that people have difficulty following directions online, including difficulty mapping from the written text to the on-screen widget and difficulty following instructions in the correct sequence. Prabaker *et al*'s DocWizards system [14] presents users with a guided walkthrough of demonstrationally-authored documentation, highlighting steps as necessary in the Eclipse interface to visually lead users through the instructions. The CoScripter system [9, 10] provides a similar guided walkthrough interface for web tasks. All of these systems assume that instructions have previously been recorded in a proprietary format; progress on automatically interpreting hand-written instructions could enable each of these systems to work with a much wider library of existing instructions.

Our work has been inspired by previous systems that automatically parse hand-written requests in email. For example, Tomasic *et al*'s VIO system [15] parses hand-written instructions in email and automatically extracts commands to fill out a form to effect a change on a web site. Where VIO is designed to work with a pre-specified set of possible forms, our system aims to work with arbitrary web pages, where the set of possible actions is not known in advance. Lockerd *et al*'s Mr. Web system [12] found that users utilize a constrained form of English when communicating with a webmaster over email. We believe a similar effect occurs in how-to documentation, where a constrained form of English is used to describe interactions with web applications.

Our work follows in the footsteps of Perkowitz *et al*'s work on mining models of human activity from the web [13]. Their PROACT system uses RFID tags in the local environment to sense objects being interacted with, and then mines the web for human-written activities involving these objects (e.g., a pot and the water faucet are used to make tea). PROACT then uses these activity descriptions to do activity recognition based on RFID sensor input. In contrast, we are mining how-to instructions in order to actually carry them out on a user's behalf, rather than simply recognizing when then are being done.

Little and Miller's Keyword Commands system [11] was one of the first to interpret user-generated instructions as actions on web pages. Their algorithm for parsing hand-written instructions is similar to one of the three interpreters we examine in this paper. While keyword-based approaches are great at interpreting underspecified commands, they can produce many false positives when given instructions that do not contain any actionable commands (which occur frequently in the datasets we have studied).

	scenarios	# steps	# segments
MT1	24	158	274
MT2	19	142	289

Table 1. Summary of datasets

DATASETS

In order to evaluate our algorithms on hand-written instructions, we wanted to collect a corpus of instructions written by people describing how to accomplish a task on the web. We crowdsourced the problem by creating a set of tasks on Amazon's Mechanical Turk marketplace, asking volunteers to write how-to instructional text in exchange for USD$0.25 per set of instructions. As Kittur *et al* found [6], we needed to iterate several times on our task description in order to get our workers to produce the desired output.

In our first iteration, we gave users a single text box and asked them to write up instructions, including the goal of the task and the website to which it was applicable. The responses we received for this task were very high-level. For example, one response began with: to search for free samples on the internet, go to a search engine and type in "free samples" as a keyword. We were hoping to have more specific instructions explaining exactly what page to visit, which buttons to click, which form fields to fill in, etc.

So we reformulated the task in a way that we hoped would encourage workers to provide more structured, detailed instructions. We explicitly asked participants to write instructions for novice computer users. Instead of a single textbox, we provided a sequence of ten individual textboxes labelled "Step #1", "Step #2", and so on. An overflow box was given for additional steps. We also asked them to fill in the URL to which these instructions referred (and included a JavaScript verifier that would not let them submit their instructions unless it contained a URL). Finally, we offered bonus payments for more detailed instructions.

The results from the reformulated task were much more specific and in line with what we had been expecting. Our workers wrote up instructions for a wide range of websites and tasks. Their scenarios included how to create a gmail account, how to buy a fan on Amazon.com, and how to create a level 70 WarCraft character using a web-based character-editing tool. Most of the scenarios described tasks on publically-accessible websites, though several required authenticated access to perform, such as logging in to Wachovia's online banking site, or accessing a university's internal website.

We collected two sets of instructions using the revised Mechanical Turk task and formed two datasets, MT1 and MT2. MT1 contains 24 scenarios, with a total of 158 steps. MT2 contains 19 scenarios, with 142 steps.

QUALITATIVE ANALYSIS

We began this project with a number of hypotheses about the nature and structure of how-to documentation. However, once we began analyzing the dataset, we were surprised by a number of features. First, there were a large number of *spelling and grammar mistakes* in the collection. For example, this step was part of a set of instructions on how to use Google search:

I 1. *Click "I'm Felling Luck' to go direclty to a website that involves your description.*

The three misspellings and pair of mismatched quotes are representative of the mistakes we found in the larger corpus. For the results described in this paper, we have left the data intact and not done any spelling or grammar correction on the data. Any excerpts quoted in this paper preserve the original mistakes made by the authors.

Second, there was a large *variation in complexity* of each individual step description. Some participants described one action per step, such as the example above. Others grouped together many actions into a single step:

I 2. *After you have filled in the information on this page, click the bottom button that says "Next." Now a form that asks for more information will pop up. If the bank already has information about your payee (such as the electric company), then an address will pop up. Check to see that the address is correct. Fill in the address if the form doesn't have one already. All the spots that have asterisks next to them need to be filled in or you won't get to have your payee confirmed. Hit the "Next" button on the bottom of the page after you have filled in all the parts of the form.*

This step includes multiple instructions as well as commentary about those instructions. For example, the author advises the reader that the starred fields are required, and to double-check that the address shown is correct.

Another feature illustrated by this example is the presence of *verification steps*. These are instructions that help the reader keep in sync by advising the reader what to look for or what to expect. In the previous quote, "now a form that asks for more information will pop up" is an example of a verification step. No action needs to be taken for this instruction; it merely advises the reader that in order to proceed, this form must now be visible on screen. While our current algorithms do not attempt to do anything with this type of instruction, we envision future systems being able to understand these verification steps and use them to ensure the user is on the right path.

We also assumed that writers would use a simple verb-object grammar to describe actions on web pages. However, we often found *out-of-order instructions* – instructions where the description of the object to be interacted with came before the instruction to operate it, such as in the following example:

Action	Value	Target type
Go to	URL	-
Click	-	link
Enter	text	textbox
Select	listitem	listbox
Turn on/off	-	checkbox/ radiobutton

Table 2. Supported web commands

I 3. *Just underneath where is says "Step 1" there is a button saying "browse", click on it.*

We also observed the presence of many *conditional* steps, such as the following:

I 4. *If you would like the Wikipedia page to remember your user id everytime you go there, then click the 'Remember Me' box. If you are on a public computer (i.e. people other than use use the computer) and you do not want others to be able to edit data in your name, then do not click this box; you will be required to sign-in everytime you would like to add/edit something on Wikipedia.*

Finally, we noticed that several steps contained implicit language:

I 5. *You can use the "Preview" button to see how it looks.*

There is no explicit instruction to click the Preview button, only advice that one can click it if they wanted to. A human reading this step may or may not choose to click the button, depending on their needs.

Labeling the data

Because our goal is to be able to automatically understand commands in handwritten documentation, we manually labeled our datasets.

Based on reading the instructions and manually following them ourselves, we identified a number of "web commands" that describe operations on web pages (Table 2). These commands collectively cover the vast majority of instructions on web pages, with `click` being the most frequent action.

In addition, we observed a number of different methods of identifying the interaction target on each page. While people occasionally used color or location to identify targets, the most common descriptor seemed to be a textual string representing the target's label or caption (e.g., "the Register link" or "the Preview button"). Thus, we have made the assumption that the combination of action, value, target label, and target type suffices to uniquely specify the desired interaction within each command.

Some steps contained more than one web command. In order to simplify the problem, we split each step into one or more segments, such that each segment contained at most one command. Commentary/advice were split into separate segments. Table 1 shows how many segments were identi-

Action type	MT1	MT2
Go to	18	14
Click	101	100
Enter	51	40
Select	1	0
Check	3	0
No action	100	135

Table 3. Segment types in each dataset

fied in each dataset.

Then we labeled each segment with the type of command (go to, click, enter, select, check, or no action). Table 3 shows the breakdown of command types in each corpus. Clicks were the most commonly-occurring action, followed by entering text. A large fraction of segments did not describe an action; these represent the commentary/advice segments discussed earlier.

UNDERSTANDING HOW-TO INSTRUCTIONS

At a high level, we have formalized the problem of understanding how-to instructions into two parts: segmentation and interpretation (Figure 1). Given a document containing written instructions, we first *segment* the document into individual segments $S_1, S_2, ...S_n$ such that each segment contains at most one command. For a first pass, we can leverage the structure of the document; many how-to documents use bulleted or numbered lists to organize instructions into a sequence of steps.

However, within each step, the text may describe one or more commands to perform. For example, the text "enter your name and click go" contains the two individual commands "enter your name" and "click go". Each command has a different action type (enter, click) and is applied to a different target in the application. The second phase of segmentation splits each step into the individual segments contained within the step.

Second, given a step segment and the context in which that instruction is to be followed (e.g., the web page to which the instruction applies), we want to *interpret* that instruction relative to the web page and formulate an executable action. The executable action should contain enough information for a machine to programmatically perform this action on the given web page.

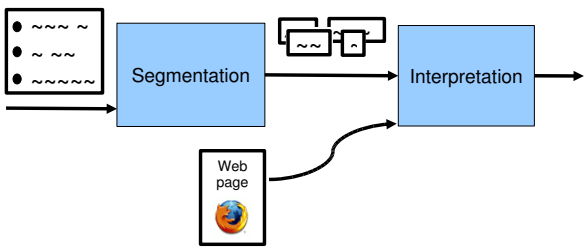

Figure 1. Understanding how-to instructions

We represent an executable action as a tuple $EA = (A, V, T)$ containing:

- **A - Action**: the type of action to perform, such as going to a URL, clicking a link, entering text, selecting an option from a listbox, or turning on/off a checkbox or radio button
- **V - Value**: the URL to go to, the text to enter in a textbox, or the value to be chosen from a listbox
- **T - Target**: the uniquely specified target on which to operate

A target T can be identified in a number of ways. For instance, the XPath of a node within an HTML DOM can be an unambiguous identifier. However, textual references in written instructions often employ a more user-friendly identification scheme based on a target's label and type. Therefore, we use two properties to specify the target T:

- $\mathbf{T}_L$ **- Target label**: a textual description of a target, which may be its label, caption, or other descriptive text
- $\mathbf{T}_T$ **- Target type**: the type of a target (e.g., link, textbox, listbox)

A target label and target type is not always sufficient to uniquely identify a target if there is more than one such target fitting that description (e.g., two buttons labeled search). One solution is to specify an ordinal (e.g., the second button) or a qualifier (e.g., the button under the Windows logo) for disambiguation. While we saw a few instances of these disambiguators, they occurred rarely in our datasets. We have chosen not to use them as target specifiers.

The next two sections describe our approach to the problems of segmentation and interpretation.

SEGMENTATION

Examination of the MT1 dataset revealed that compound steps (steps containing more than one segment) were very common. Authors frequently grouped together multiple instructions to be performed on the same page into the same step. However, in order to programmatically execute these instructions, we need to split each step into its component instructions so that they can be interpreted individually.

We experimented with two different approaches to segmentation. Our first approach, the Simple Segmenter (S_S), used a simple tokenization scheme to split each step on occurrences of the word `and`, the word `then`, periods, and newlines.

Our second approach, a Machine Learning-based Segmenter (S_{ML}), used machine learning to predict segment regions. We used MinorThird [3] to train a CRF learner [7] to extract segments from steps. The CRF learner classifies each token in an input document into whether it begins, continues, or ends a segment region. The features used for classification were the default features used by Minorthird, and include the token value, values of tokens to the left and right, character

	Precision	Recall	F1
S_S	0.42	0.57	0.48
S_{ML}	0.58	0.47	0.52

Table 4. Comparison of segmentation algorithms

types (e.g., punctuation), and the classification of previous tokens in the document. We trained S_{ML} on the MT1 training set, and report results for both algorithms on the MT2 dataset.

The results are summarized in Table 4. Both S_S and S_{ML} have fairly low precision and recall. The low performance is primarily due to the difficulty of the segmentation problem and the challenge of doing shallow segmentation without access to a language model.

The S_S algorithm suffered from the expected problems of using surface-level features for segmentation. For example, the word "and" sometimes is used to delimit segments, and sometimes is used in a label or as a conjunction within a single segment. For example, S_S mistakenly splits the following step on the word "and":

I 6. *In the upper left corner, enter in the account number and password associated with the account*

Similarly, splitting on periods is often the correct choice, but fails on cases where periods are used within a sentence, such as in the expression "e.g.". Using a language model would likely improve our results significantly.

Another challenge for the segmenter was verification steps, such as "now the picture should appear on the screen." Although we labeled verification steps as separate segments, instruction writers often combined these with commentary and actions, making it difficult to automatically split those steps into segments. For example, both S_S and S_{ML} misclassified the following steps as each being a single segment (curly braces indicate true segment boundaries):

I 7. {*When your inbox has come up*}, {*you can see how many new emails you got*}

I 8. {*The website will display the payment information for you to review*}, {*if everything looks correct click on confirm.*}

INTERPRETATION

As shown in Figure 2, interpreters take as input a segment and its associated context (such as a web page) and output an executable action EA that can be taken on that web page.

We have developed three different approaches to interpreting how-to instructions. The sloppy interpreter (SL) is a keyword-based algorithm that uses the web page context and the words in the instruction to directly generate executable actions for that web page. The structured interpreter (ST) uses a grammar-based parser to extract the action, value, target label, and target type from the instruction. It then uses a resolver to search the current web page for a target with this label and type, and outputs an executable action composed of the predicted action, value, and target. Both SL and ST have been used in the CoScripter system; SL in the original version [10] and ST in the current version. The machine learning-based interpreter (ML) is similar to ST except that instead of the grammar-based parser, it uses trained document classifiers and information extractors to predict the action and extract the value, target label, and target type from each instruction. It then uses the same resolver used in ST to resolve the target descriptor into a target on the current web page.

The intermediate representation (A, V, T_L, T_T) is produced by both ST and ML, which is then resolved into an executable action (A, V, T) on a specific web page. However, because SL does not produce this intermediate representation but directly produces the executable action, we cannot compare its parsing performance to the other two approaches. We can only compare all three approaches on the executable actions they produce.

Sloppy Interpreter

The SL algorithm has been described in more detail previously [10], however we will briefly discuss it here.

SL interprets a segment of text in the context of a web page. It begins by enumerating all elements on the page with which users can interact (e.g., links, buttons, text fields). It then generates a label for each element using the `getLabel` function described below. SL then assigns a score to each element by comparing the words in the input instruction with those in the element's label. The highest scoring element is output as the predicted target.

The scoring process works as follows. Each element starts with an initial score of 0. For each word in the instruction that matches a word in the element's label, SL increases the score of that element. The increase is uniform for most matching words, however certain common words (such as "the") have been downweighted. An element's score is also increased if a verb appears in the instruction that can be applied to the element. For example, if `click` appears in the

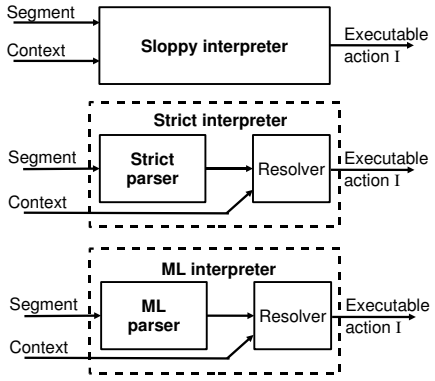

Figure 2. Three interpreters: sloppy (SL), structured (ST), and ML

instruction then all link elements would have their score increased, and similar increases would happen for text fields if `enter` appeared. If target type appears in the instruction, such as `link` and `text box`, then elements of that type will also have their score increased. If after scoring all elements on the page, two elements have the same score, SL picks the one that appears first on the page.

Once an element has been found, it determines the type of action to be performed on it. For example, links are always imply `click` actions; textboxes imply `enter` commands. Certain actions, such as `enter`, also require a value. This value is extracted by removing words from the instruction that are closely associated with the element label; the remaining words are likely to be the value. Additional heuristics are used to improve accuracy, such as preferring words that follow a preposition.

The sloppy parser was originally designed with the assumption that its input will always contain an action, and thus it will always produce an action that can be executed on the current web page. We expect SL to generate a number of unwanted results with the datasets we collected as they contain a number of steps that are not meant to describe an action that is to be executed.

Structured Interpreter

ST uses an $LL(1)$ parser that parses imperative English instructions that are applicable to the web domain. This limits the flexibility of the instructions that can be recognized, but has the advantage that it is simple, fast, and precise. If the structured parser accepts an instruction we can be very sure that it is a well formed statement. Statements it successfully accepts are of the form:

I 9. *go to www.gmcard.com*

I 10. *click on the "Save and Submit" button*

The verb at the beginning of each statement describes the action. Optionally a value can be specified, followed by a description of the target label and an optional target type. Figure 3 shows an excerpt of the BNF grammar recognized by this parser, for `click` actions. The result of a successful parse of a human readable instruction is the extraction of the tuple Action A, Value V, Target Label T_L and Target Type T_T from the parsed instruction.

If the instruction is successfully parsed by ST, then we make sure that the instruction is valid in the context of the current web page. This process is shared with the machine-learning based interpreter and is discussed later.

Click ::= click {on} the TargetSpec
TargetSpec ::= TargetLabel {TargetType}
TargetLabel ::= "String"
TargetType ::= link | button | item | area

Figure 3. Excerpt from structured parser's grammar

Machine learning-based interpreter

Interpreting written instructions can be formulated as a machine learning problem. By using state-of-the-art information extraction techniques, we hypothesized that we could build a parser that is more tolerant of variations in instruction wording (one of the weaknesses of the structured parser) yet achieves a higher precision than the keyword-based sloppy parser.

Using the MT1 dataset, we trained a multi-class document classifier on the set of segments. For each segment, this classifier predicts a class label representing the action type of the segment (goto, click, enter, select, check, or no action). We used the OneVsAll learner from the MinorThird package [3], which builds N different classifiers, each of which discriminates between instances of one class and all the remaining classes. The results of the N individual classifiers are combined to form the final prediction. We used the default MinorThird settings, which specify a Maximum Entropy learner as the inner classifier, and used the default set of features.

Once each segment has been classified with an action, we trained individual extractors to identify the crucial parts of each segment necessary to execute it. For goto actions, we extract the URL. For clicks and checks, we extract a target type and a target label – enough to specify which target to click on. For enter actions, we extract the target type, target label, and (where possible) the text to enter into the text field. For select actions, we extract the target type, target label, and the name of the item to select from the dropdown.

For each of these 11 extraction tasks, we trained a separate annotator using the MinorThird toolkit. Each annotator was trained on the labelled data in the MT1 dataset. For example, the goto-value annotator was trained to extract the values from the 18 goto segments in MT1. We used the default MinorThird settings to create the annotators, which use a VPHMM learner [4].

Once the target label and target type have been identified for each action, we use the same algorithm as the structured parser to map this description to a specific target on the web page.

Resolving targets

Our current system resolves the target label and target type into the target element using a function called `findTarget`. The parsing stage of the interpreter will have extracted an action, a target label and a target type. These latter two values are passed to `findTarget`, and, if successful, `findTarget`returns a HTML DOM element as the target. The extraction is done in three steps.

First, `findTarget`identifies all targets of the desired target type on the current web page. For example, the target type "link" will enumerate all link elements. For the type `button`, the enumeration contains the union of all elements of type `button`, all `input` elements with a type attribute of `submit`, and all link elements that contain an image.

Next, `findTarget` generates one label for each element in the enumeration using a `getLabel` function. `getLabel` takes a DOM node as a parameter and returns a label for that element. The label is generated from text the author of the web page explicitly or implicitly used as a label, including the HTML `label` element, `alt` text and other accessibility features of HTML. If none of these are available, `getLabel` will extract visible text in close proximity to the element to use as a label. Heuristics are used to generate the best possible label. For links, an easy heuristic is to use the text enclosed in the link element. For a `checkbox`, any text immediately to the right will be used first, and text to the left is used only if no text is found to the right.

Finally, `findTarget` compares the labels generated by `getLabel` for each element in the enumeration against the target label. If an exact match is found, then that element is returned as the target. It is important to note that this implementation of `findTarget` will fail to find a target if it is given a partial label, such as "Search" for a button actually labeled "Google Search". We will discuss the implications of this behavior for our results later in the Discussion section.

Contrasting the interpreters

We expect the three different interpreters to have different behavior on our dataset. SL performs a keyword-based match between the instruction and the web page, which allows it to predict an action even when the instruction is underspecified. However, this same characteristic makes it unpredictable on text that does not contain a direct command; SL will always predict *some* action, even if the instruction does not contain a command.

In contrast, the ST parser relies on a grammar to parse incoming statements. Because few people write their instructions exactly according to our grammar, we do not expect ST to achieve a high recall – we expect very few of the instructions in the dataset to match the ST parser's syntax exactly. However, when a statement does match the grammar, we expect ST to have a very high precision, and parse the statement correctly.

We hope that the ML parser combines some of the best features of SL and ST. By examining features of the entire input segment, we expect it to be able to predict the action type with fairly high precision, despite variances in wording that would trip up the ST parser. The extraction model learned by the ML parser will also hopefully be better at recognizing target labels and target types than the ST parser, which can only recognize target descriptors when the entire command obeys the correct syntax.

The ultimate performance of the ST and ML interpreters will depend on the capabilities of the resolver component and its `findTarget` function. For the purposes of this paper we describe a very simple resolver, which is only capable of finding targets given their exact label. Therefore the instruction must mention the full label of each target (and be spelled correctly) in order for the ST/ML resolver to correctly resolve that target on a web page. In contrast, by combining parsing and resolving into a single step, SL's algorithm is capable of locating targets given few or no words of the target label. This feature should give SL an advantage when the target is underspecified.

	Action	Value	Target Label	Target Type	Total
ST, all	0.50	0.88	0.66	0.71	0.43
ML, all	0.84	0.88	0.76	0.78	0.50
ST prec	0.91	1.00	0.75	0.50	0.82
ST rec	0.06	0.08	0.06	0.01	0.06
ML prec	0.82	0.32	0.67	0.79	0.27
ML rec	0.84	0.24	0.34	0.26	0.28

Table 5. ST/ML parser comparison. "All" reports the accuracy of each system on the full test set. "prec" is the fraction of correctly predicted items out of the set of segments with a prediction. "rec" is the fraction of items for which the correct prediction was made, out of the total number of segments with an action. The "Total" column reports when the system gets all possible fields correct.

EVALUATION

We compared the performance of our interpreters using two different experiments. First, since the ST and ML interpreters both generate an intermediate parse of the input segment, we were able to directly compare the performance of the parser components by comparing their intermediate representations.

However, parsing is only the first step to understanding; we would also like to know whether the interpreter produces the correct executable action given an input segment. Our second experiment evaluates the performance of all three interpreters and compares their ability to produce complete executable actions given instructional segments.

Parsing performance

The ST and ML parsers both produce intermediate parses of each instruction segment prior to resolving them into full-fledged executable actions on a web page. Due to the large variation in writing styles, we expected the structured parser to be able to parse only a small subset of the handwritten instructions (the subset that conforms to the syntax we have defined for web commands).

As the first two lines in Table 5 show, the ST parser and the ML parser exhibit nearly comparable performance on the MT2 dataset. However, the ML parser is much more effective at recognizing actions than the ST parser. This is due to the fact that the ST parser rarely recognizes any actions as being grammatically correct, and its default is to predict `noaction`. Since nearly half of our dataset consists of `noaction` segments, a default prediction of `noaction` causes the ST parser to do fairly well. However, a system that always predicted not to do anything with instructions would not be a very useful system.

So we measured the performance of the ST and ML parsers on only the subset of segments that contain actions. The

precision and recall figures in Table 5 reflect these metrics. Precision measures the fraction of times the parser made the correct prediction, out of the total number of labelled instances (i.e., segments that contain an action, ignoring the `noaction` segments as unlabeled). Recall measures the fraction of correctly predicted labels out of the total number of labelled instances. These metrics effectively ignore the `noaction` segments in the dataset and focus on the system's performance with segments that do specify an action.

In this condition, the ST parser displays extremely high precision (91%) but abysmally low recall (6%). Of the 154 segments that contain an action, the ST parser made a prediction for only 11 of those. These numbers are consistent with our hypothesis that the ST parser would not be able to parse most of the instructional segments due to the wide variation in language used by participants to express web commands. On the other hand, where it is able to parse a segment, it nearly always parses it correctly. The value is extracted with 100% precision and target label with 75% precision. The low target type precision (50%) is correlated with the extremely low target type recall (1%), which basically means that very few of the statements contained a target type in the syntax expected by the ST parser. Of those two, one was correct and one was incorrect.

By comparison, the ML parser performs much better. Recall that the ML parser uses a multi-class classifier to predict the action type given the entire input segment. It correctly predicts the action 82% of the time, with an 84% recall, showing that the ML parser is generally able to classify segments with the correct action. The ML parser's performance on the extraction tasks (extracting the value, target label, and target type) is still comparable with the ST parser overall; but focusing on the actionable steps, its recall is much higher. This shows that the ML parser is able to extract these fields from the unstructured text much more often than the structured parser can, and shows the benefits of the ML approach over the use of a strict grammar.

Interpretation performance

We evaluate the interpretation performance of SL, ST, and ML interpreters by comparing their performance each to the performance of a test subject. The test subject was given an instruction segment and the matching web page. The subject noted the action that would result from following this instruction. We compare the human subject's action against the executable action predicted by our interpreters, and count our prediction as correct if they match.

We evaluated the interpreters on the MT2 dataset. Some instruction segments had to be discarded to assure that we could deterministically compare the actions generated by the human subject with the ones generated by the SL, ST, and ML interpreters. We could only verify actions on web pages we had access to. Some instructions that required accounts on financial institutions' websites were discarded. Other instruction segments had to be discarded because the web page had changed in such a way that the instruction was no longer applicable. For example, facebook.com underwent a major update of their user interface, and many of the instructions for facebook.com are no longer applicable to the new interface design. Instruction that required a human interpretation, knowledge only a human could provide, or that specified multiple targets were also discarded. Table 6 illustrates the segment classes that were discarded.

Total in MT2	289
Inaccessible	122
Requires Human Interpretation	30
Segments for study	137

Table 6. Classification of segments for interpretation experiment

The results of the overall interpretation performance experiment is shown in Table 7. As we expected, ST performed very poorly on everything but `noaction` instructions in the the MT2 dataset and very well on `noaction` instructions. The overall performance of ST therefore looks much better than it really is. Since the dataset used contains nearly 50% `noaction` instructions, the interpretation performance of ST can be expected to be at least that high. SL performed well on segments that contained an action, but since it lacked the ability to predict `noactions`, its overall performance is diminished.

	action				
Interpreter	click	enter	goto	noact.	total
SL	0.46	0.22	0.77	0.00	0.23
ST	0.03	0.00	0.08	0.99	0.50
ML	0.31	0.06	0.23	0.84	0.53
Total	39	18	13	67	137

Table 7. Percentage correctly-interpreted segments for each action class

Overall the ML interpreter was able to successfully recognize twice as many instructions as SL, and that without the problems that SL has when it comes to `noaction` instructions. It recognized more than 80% of those. Its performance on interpreting `enter` instructions is low. This is due to the fact that it had trouble extracting the exact value component that was meant to be entered in the target.

DISCUSSION

The sloppy interpreter used in the original CoScripter system (formerly known as Koala) was amazingly simple, and performed well on a wide variety of instructions. However, the downside to its lax approach to parsing is that it often incorrectly parses statements and suggests incorrect actions. When used to automate script execution, scripts would often go awry, clicking the wrong links and leading the user to unpredictable web pages.

The interpretation performance reported for SL agrees with our field observations. While SL correctly interpreted many of the statements, it never correctly predicted `noaction`.

The structured interpreter was designed in an attempt to mitigate the problems created by the sloppy interpreter. By parsing only a limited subset of instruction syntax, we hoped

that the structured parser could improve the precision of the sloppy interpreter. While this has been useful at increasing the robustness of the CoScripter system, it is not up to the task of parsing arbitrary hand-written instructions.

As we expected, the ML interpreter performs better than either the SL or ST interpreters. Its high performance is largely due to correct recognition of `noaction` instructions, for which the SL interpreter would have incorrectly predicted an action. However, the ML interpreter is also able to correctly interpret many of the actionable commands nearly as well as the SL interpreter. Thus it provides a reasonable tradeoff between the looseness of the SL interpreter and the strictness of the ST interpreter.

IMPLICATIONS FOR DESIGN

The empirical results in the previous section show that our ML interpreter is capable of correctly interpreting 53% of instructional segments in a dataset of hand-written instructions collected from the web.

Given the difficult nature of the problem, we are encouraged by this result. We anticipate incorporating these results into a system that can import legacy instructions directly from the web, and guide the user through them step by step, automating steps where applicable. Within the context of such a system, correctly identifying when no action is required is nearly as important as predicting the correct action when one is required. Based on our experience with CoScripter, a system that interprets instructions incorrectly can be extremely disorienting to users.

We also wish to point out that a guided walkthrough system can be useful even if not all the components of an executable action are recognized correctly. For example, if the system recognized an `enter` action on a specific target, but could not guess the value to fill in to the field, it could still highlight the target field and prompt the user to type in their desired value. Such a system could still be useful to users even though some steps are not interpreted completely accurately given our current metrics.

To shed more light on misclassification failures and how they might impact the performance of a system built on our algorithms, we present a confusion matrix comparing the actual action and the predicted action for the ML interpreter (Table 8).

Many of the `noaction` instructions could have been interpreted as other types of instructions. In some cases, this was due to commentary referencing buttons on the page without explicit commands to activate them, which caused the classifier to mistake a `noaction` for a `click`:

I 11. *You can use the "Preview" button to see how it looks.*

I 12. *A window should pop up with a big button marked "Print" and the map.*

A system that instructed the user to click on each of these buttons would probably not be considered in error.

The language used to describe pressing keys on the keyboard often resembled the language for clicking buttons with the mouse. We made a (somewhat arbitrary) decision to classify the former as `enter` commands and the latter as `click` commands. Often the classifier would mistakenly classify these `enter` instructions as `click`s, as in the following examples:

I 13. *press enter*

I 14. *hit the tab button to move from space to space*

Correctly classifying these commands seems to require common sense knowledge about keyboards, for example knowing that the button labeled "Tab" is a physical button on the keyboard, and probably not a link on the web page. However, if the system mistakenly predicted that a link labeled "tab" were to be clicked, it is likely that such a link did not exist on the current web page, so the system would be unable to interpret this instruction, and present it the user for manual interpretation. Because of the "fail-soft" nature of these types of errors, we feel that a system that attempted to automatically interpret how-to instructions would perform relatively well in practice.

	Predicted				
Actual	**click**	**enter**	**goto**	**select**	**noaction**
click	**90**	0	0	0	10
enter	6	**29**	0	0	5
goto	0	2	**11**	0	1
select	0	0	0	**0**	0
noaction	11	7	3	0	**114**

Table 8. ML parser confusion matrix

CONCLUSION

In summary, we have presented an analysis of and solutions to the problem of automatically understanding hand-written how-to instructions. We reported on the collection of a dataset of hand-written instructions for 43 web tasks, and presented a qualitative analysis of these instructions. We have compared two approaches to segmenting instructions, and showed that it is a difficult problem that likely requires language model analysis. We presented and compared three different approaches to instruction interpretation, showing that the machine learning-based algorithm outperforms both the keyword-based and grammar-based algorithms, achieving 53% accuracy at interpreting instructional statements. Our results indicate that automatic how-to instruction understanding shows promise, and may soon enable the creation of systems that assist users in following hand-written instructions.

Many directions remain for future work. We would like to combine the three approaches into a single meta-interpreter that combines the precision of the ST interpreter with the enhanced recall of the SL and ML interpreters. We plan

to incorporate our best approach into the CoScripter system so that users encountering how-to documentation online can quickly import and execute it in the system. Finally, we plan to validate our results using more data, including some written by professional technical writers, to confirm that higher-quality documentation exhibits more structure and is easier to understand.

REFERENCES

1. M. Bolin, M. Webber, P. Rha, T. Wilson, and R. C. Miller. Automation and customization of rendered web pages. In *UIST '05: Proceedings of the 18th annual ACM symposium on User interface software and technology*, pages 163–172, New York, NY, USA, 2005. ACM.

2. Designing Coachmarks. `http://developer.apple.com/documentation/mac/AppleGuide/AppleGuide-24.html`, 1996.

3. W. W. Cohen. Minorthird: Methods for Identifying Names and Ontological Relations in Text using Heuristics for Inducing Regularities from Data. `http://minorthird.sourceforge.net`, 2004.

4. M. Collins. Discriminative Training Methods for Hidden Markov Models: Theory and Experiments with Perceptron Algorithms. In *EMNLP '02: Proceedings of the 2002 conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2002.

5. C. Kelleher and R. Pausch. Stencils-based tutorials: design and evaluation. In *CHI '05: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 541–550, New York, NY, USA, 2005. ACM.

6. A. Kittur, E. H. Chi, and B. Suh. Crowdsourcing user studies with Mechanical Turk. In *CHI '08: Proceeding of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*, pages 453–456, New York, NY, USA, 2008. ACM.

7. J. D. Lafferty, A. McCallum, and F. C. N. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *ICML '01: Proceedings of the Eighteenth International Conference on Machine Learning*, pages 282–289, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

8. T. Lau, L. Bergman, V. Castelli, and D. Oblinger. Sheepdog: learning procedures for technical support. In *IUI '04: Proceedings of the 9th international conference on Intelligent user interfaces*, pages 109–116, New York, NY, USA, 2004. ACM.

9. G. Leshed, E. M. Haber, T. Matthews, and T. Lau. CoScripter: automating & sharing how-to knowledge in the enterprise. In *CHI '08: Proceeding of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*, pages 1719–1728, New York, NY, USA, 2008. ACM.

10. G. Little, T. A. Lau, A. Cypher, J. Lin, E. M. Haber, and E. Kandogan. Koala: capture, share, automate, personalize business processes on the web. In *CHI '07: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 943–946, New York, NY, USA, 2007. ACM.

11. G. Little and R. C. Miller. Translating keyword commands into executable code. In *UIST '06: Proceedings of the 19th annual ACM symposium on User interface software and technology*, pages 135–144, New York, NY, USA, 2006. ACM.

12. A. Lockerd, H. Pham, T. Sharon, and T. Selker. Mr.web: an automated interactive webmaster. In *CHI '03: CHI '03 extended abstracts on Human factors in computing systems*, pages 812–813, New York, NY, USA, 2003. ACM.

13. M. Perkowitz, M. Philipose, K. Fishkin, and D. J. Patterson. Mining models of human activities from the web. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 573–582, New York, NY, USA, 2004. ACM.

14. M. Prabaker, L. Bergman, and V. Castelli. An evaluation of using programming by demonstration and guided walkthrough techniques for authoring and utilizing documentation. In *CHI '06: Proceedings of the SIGCHI conference on Human Factors in computing systems*, pages 241–250, New York, NY, USA, 2006. ACM.

15. A. Tomasic, J. Zimmerman, and I. Simmons. Linking messages and form requests. In *IUI '06: Proceedings of the 11th international conference on Intelligent user interfaces*, pages 78–85, New York, NY, USA, 2006. ACM.