Automatically Identifying Valid Web Form Inputs

Patrick Mutchler
pcm2d@stanford.edu

***Abstract*—In this paper we provide a classifier that can accurately predict if a set of inputs is valid for a given web form. Our classifier is based on the assumption that text found near an input field gives information about what kind of input that field expects. We use a Naive Bayes classifier to determine if form inputs are valid. We collected forms from 20 of the most visited websites in America and hundreds of inputs to those forms. We found that our classifier had a high success rate for determining the validity of an input/field pair as well as determining if a field is a "confirmation" field. Using our estimates from the Naive Bayes classifier, we classified whole form inputs with high accuracy.**

I. Introduction

Achieving complete or near-complete site coverage is an essential feature of a web crawler. This task is made all the more difficult by the complexity of modern web application design. Crawlers can no longer blindly follow hyperlinks and hope to achieve any success. Instead, they must be able to interact with a variety of technologies (AJAX, Flash, etc.), monitor application state, and traverse behind web forms [1] [2]. The latter challenge is know as *deep crawling*. In order to deep crawl effectively, a crawler must be able to submit form data that the application will accept. Unfortunately, applications often expect specific input formats for individual form fields (valid addresses, emails, phone numbers, etc.).

Top of the line crawlers make an attempt to solve this problem by including a dictionary of common words that identify specific kinds of inputs. When the crawler must fill out an input field, it checks the id or name fields of the input tag against its dictionary. If there is a match, then the crawler inserts the appropriate input. This appraoch is somewhat successful, but suffers from being very brittle. If the name or id fields are not in precisely the expected format, then the system fails. For example, the ids "Passwd" and "user[password]" don't trigger the system to input a proper password.

To help address this problem we propose a classifier that can distinguish valid and invalid web form data. By testing inputs with our classifier before submitting them to the web application, a crawler can traverse an application with fewer requests to the application. This makes the crawling process faster and places a smaller burden on the application server.

Our classifier takes advantage of two observations about web site design. The first is that web applications are designed to be immediately usable by anybody with basic web browsing experience, and therefore must follow general patterns that are common to most web applications. The pattern that we utilize is *form fields must have nearby text that tell the user what sort of data is expected, and that the format of this text is relatively consistent across web applications*. The second observation is that web applications are designed so that the code is as simple and readable as possible. This means that the names used to reference page elements reflect their functionality. Concretely, the reference names of form fields often contain information about what sort of data the field expects.

Using these observations, we developed a Naive Bayes classifier that can accurrately predict if a set of inputs is valid for a given web form. We collected data from 20 of the most popular websites (based on Alexa). We chose to only collect forms to create accounts, since these forms tended to have a large number of of input fields that expect particular kinds of inputs. In all, we collected 300 complete form inputs, which amounted to approximately 2000 total input/field pairs. We found that our classifier had a high success rate for determining the validity of an input/field pair as well as determining if a field is a "confirmation" field. Using our estimates from the Naive Bayes classifier, we were able to classify whole form inputs with high accuracy.

II. Formulating the Problem

Before we can model an entire web form, we must model individual form fields. We model each field f_k as a set of strings $\{f_{k0}, f_{k1}, ...f_{kn_k}\}$ corresponding to the id and name attributes of the HTML tag as well as any relevant strings found near the field on the rendered page. The id and name attributes capture how the input tag is referenced by the application while the "nearby" strings capture information that a human user is given before being asked to fill out the form. Precise definitions of "nearness" and "relevance" are discussed in section 3.

Next we must model the input to the form field. It is not enough to model the input, i_k, directly as a string since it is not likely that a particular input string will appear in our training set. In addition to the input string, we also model the input as a *regular expression* that matches the input string. In particular, we choose the most exclusive regular expression from a dictionary of regular expressions that matches the input string. This is an appropriate choice because all of the common input filters found on websites are regular expressions. We let the term r_k be the regular expression for input i_k.

We can now model a form F as follows. $F = \{\{i_0, r_0, f_0\}, \{i_1, r_1, f_1\} \dots \{i_n, r_n, f_n\}\}$. An obvious approach would be to say that a form is valid if and only if each of its input/field pairs are valid. This, however, does not capture the requirement that the inputs to different fields be identical e.g. "password" and "confirm password" fields. To account for this, we include the term d_i for each $i \in \{1 \dots n\}$. If $d_i = 1$, then for F to be valid, i_i must be identical to

i_{i-1}. Note that this only allows for adjacent fields to expect identical inputs. In practice, it is extremely unusual for two non-adjacenet fields to expect identical input strings.

Finally, we have a complete model of a form F:

$$F = \{\{i_0, r_0, f_0, d_0\}, \{i_1, r_1, f_1, d_1\} \dots \{i_n, r_n, f_n, d_n\}\}$$

And we have an expression for valid(F)

$$\text{valid}(F) = \forall i \in 0 \dots n, \text{valid}(r_i, f_i) = 1 \wedge d_i \Rightarrow i_i = i_{i-1}$$

III. Data Collection

As far as we are aware, no public data sets exist for valid and invalid form inputs. This meant that we had to collect our own data specifically for this project. We built two tools for automated data collection. The first tool extracts and parses forms from raw HTML files while the second allows for quickly creating valid and invalid inputs to a particular form. In all, we collected 300 manual inputs to 20 forms from some of the most visited web sites in America (based on Alexa). This is around 2000 total input/field pairs.

A. Extracting forms from HTML

The two challenges in extracting forms from raw HTML are determining what strings are "relevant" and determining what strings are "near to" a given input field.

In order to decide what makes a particular string "relevant", we have a dictionary of strings that commonly appear in web forms and provide identifying information about form inputs. For example, the string "zip" appears in our dictionary because it is common for address forms to include the word "zip" and the word is a strong indication that a field expects a zip code. Our dictionary was collected manually and then trimmed by removing strings and seeing if they affected our experimental results. We also perform some normalization during this step. For example, if a web form contains the word "zipcode", we normalize it to "zip". This gives us greater uniformity between different web forms and leads to better results.

Computing nearness on a page directly is extremely difficult since it requires us to correctly render the page and then determine where each element is positioned on screen. Instead, we make the assumption that elements which are close to eachother in the HTML parse tree will be close to eachother on the rendered page. We say that text is "near" an input field if their least upper bound has exactly one input tag and zero form tags below it. See Figure 1 for a visual explanation.

We built our tool using HTMLCleaner, a java library that provides methods for parsing raw HTML.

B. Collecting valid and invalid inputs

In order to collect inputs, we built a simple web application that allows us to upload a form and then record inputs to that form. The application displays the form as it appears on the original page except it includes checkboxes for the user to assert that an input is valid or invalid. We assume that all of the assertions about the validity of inputs are accurate.

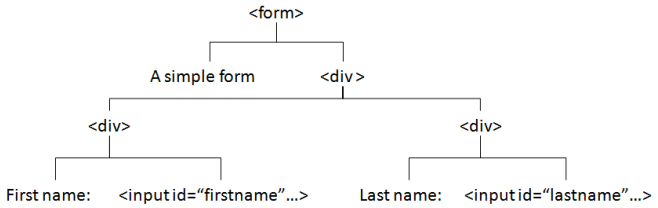

Fig. 1. HTML parse tree demonstrating "nearness". The string "First name" is near the input with id "firstname" and the string "Last name" is near the input with id "lastname" but the string "A simple form" is not near to either of the inputs.

C. Dictionaries

Our dictionaries of relevant strings (for modeling fields) and relevant regular expressions (for modeling inputs) were created manually based on what seemed appropriate. We then adjusted our dictionaries (adding or removing elements), reran our experiments, and kept changes that led to better results. This means that our dictionaries are fitted to the specific forms we collected. This is perhaps an area of experimental bias, but we feel that since forms are so uniform across web sites that this approach was not inappropriate.

IV. Implementation

A. Computing validity for a single input/field pair

First we show how to compute the probability that a single input/field pair is valid. For now we can ignore our form based model and treat our training data as a set of input/field pairs. This means that in the following section, m refers to the number of input/field pairs in the entire training set.

Let r correspond to a regular expression matching some input and f correspond to a set of strings near some field. This means that r is a signle integer matching an index in our dictionary of regular expressions and f is a set of integers $\{f_0, f_1 \dots f_n\}$ where each f_i matches an index in our normalized string dictionary. Finally let v be whether or not an input/field pair is valid.

We use Bayes' rule and the Naive Bayes assumption to compute $P(v|r, f)$ [3].

$$P(v|r, f) \sim P(v)P(f|v)P(r|f, v)$$

where

$$P(f|v) = \prod_{k=1}^{n} P(f_k|v) \text{ and } P(r|f, v) = \prod_{k=1}^{n} P(r|f_k, v)$$

By solving for the maximum likelihood (and using Laplace smoothing) we find that the ML estimates are:

$$P(v) = \frac{1}{m} \sum_{i=1}^{m} 1\{v^{(i)} = v\}$$

$$P(f_k|v) = \frac{\sum_{i=1}^{m} \sum_{j=1}^{n_i} 1\{f_j^{(i)} = f_k \wedge v^{(i)} = v\} + 1}{\sum_{i=1}^{m} n_i\{v^{(i)} = v\} + |D_{string}|}$$

$$P(r|f_k, v) = \frac{\sum_{i=1}^{m}\sum_{j=1}^{n_i} 1\{r^{(i)} = r \wedge f_j^{(i)} = f_k \wedge v^{(i)} = v\} + 1}{\sum_{i=1}^{m}\sum_{j=1}^{n_i} 1\{f_j^{(i)} = f_k \wedge v^{(i)} = v\} + |D_{regex}|} \quad (1)$$

Since our string and regular expression dictionaries are relatively small (fewer than 100 values each), it is possible to precompute $P(v)$, $P(f_k, v)$, and $P(r|f_k, v)$ for each r, f_k, and v. This makes computing $P(v|r, f)$ extremely fast since we just need to look up 6 values and do some additions. This speed is essential since we need our classification time to be much faster than actually sending an input to a website and interpreting the results.

B. Computing whether a field is a "confirmation field"

We compute whether a field is a "confirmation field" using a similar method. Again let f be the set of integers $\{f_0, f_1 \ldots f_n\}$ where each f_i matches an index in our normalized string dictionary and d be whether or not the field is a "confirmation field". Note that in this section m refers to the number of unique fields in the entire training set.

We use Bayes' rule and the Naive Bayes assumption to compute $P(d, f)$ [3].

$$P(d|f) \sim P(d)P(f|d)$$

where

$$P(f|d) = \prod_{k=1}^{n} P(f_k|d)$$

Again, we solve for the maximum likelihood to find the ML estimates:

$$P(d) = \frac{1}{m}\sum_{i=1}^{m} 1\{d^{(i)} = d\}$$

$$P(f_k|d) = \frac{\sum_{i=1}^{m}\sum_{j=1}^{n_i} 1\{f_j^{(i)} = f_k \wedge d^{(i)} = d\} + 1}{\sum_{i=1}^{m} n_i\{d^{(i)} = d\} + |D_{string}|}$$

Just like in the previous section, we precompute $P(d)$ and $P(f_k|d)$ for all d and f_k in order to speed up classification time.

C. Computing validity for an entire form

Rather than trying to compute the probabilities that a form input is valid or invalid, we instead take our estimates for $valid(r_i, f_i)$ and d_i and compute $valid(F)$ using the expression given in section 2. This gave us very good results, which are discussed in section 5.

V. Results

To test our system, we collected three kinds of results: how successful it is at determining if an input/field pair is valid (Figure 2), how successful it is at determining if a field is a "confirmation" field (Figure 3), and how successful it is at determining if an entire form input is valid (Figure 4).

To test how successful our system is at determining if an input/field pair is valid, we computed the percentage of correct classifications when training on the entire dataset, training on all the data except the particular input, and training on all the data except the inputs to the page that the input we are testing belongs to. When we hold out an entire web page we see a significant drop in accuracy. This is typically due to a page having an unusual field. For example, our error rate when testing on eBay's signup form is high because it asks for a phone number in an unusual manner. If we train on even a few inputs to a web page, then our success rate is very close to our success rate when training on the entire data set.

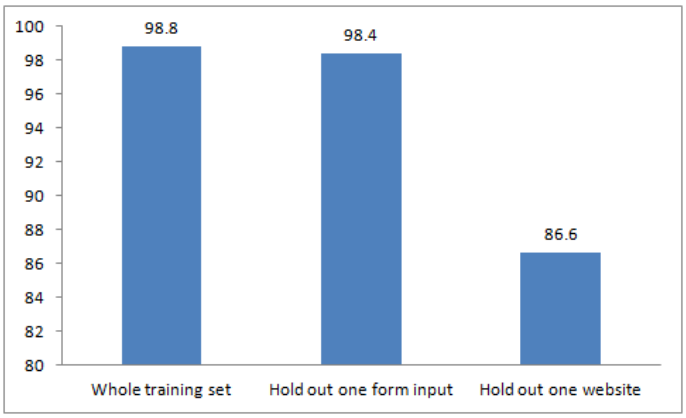

Fig. 2. Percent successful classifications for input/field pairs

We performed similar tests to see how succesful our system is at determining if a field is a confirmation field. We either train on the entire data set, the entire data set except one field, or the entire data set except one web page and get similar results to our previous experiement. We were surprised by the significant decline in accuracy when holding out an entire web page, since it would seem that each of the fields on a page are somewhat independent. Interestingly, all of our errors were estimating that a field was a confirmation field when it wasn't. This suggests that there is something wrong with the model we are using.

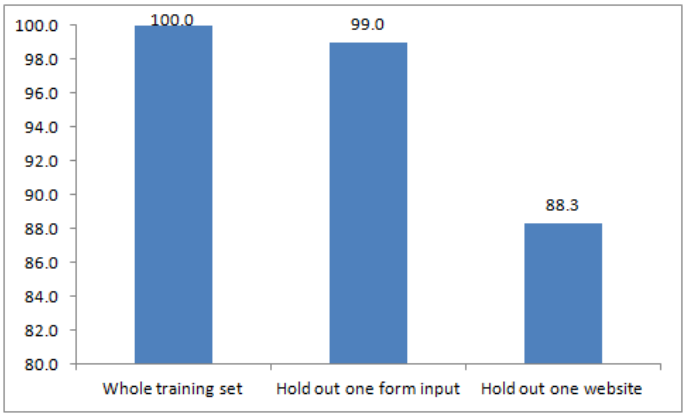

Fig. 3. Percent successful classifications of "confirmation" fields

Finally, we tested how successful our system is at determining the validity of an entire form input. We used our simple method of computing the validity of a form given our estimates for the validity of each input/field pair and our estimates for

whether each field is a confirmation field. We got excellent results, being able to correctly classify form inputs 90% of the time even if we have not trained on the form at all. These results are a little misleading, however, because the vast majority of our errors come from misclassifying a valid form input as invalid. This is because it only takes a single invalid input/field pair to make the entire form invalid. A single error on a valid form input leads to a misclassification. For an invalid form input to be misclassified, the system must make an error for each invalid field/input pair and no errors for each valid field/input pair. In particular, this means that it is possible to make many errors and still correctly classify every form input.

Fig. 4. Percent successful classifications of entire forms

VI. Conclusion

In this paper we presented a classifier that can determine whether or not an input to a web form is valid. We collected forms from 20 of the most visited sites in America and 300 inputs to these forms. Using a Naive Bayes classifier, we were able to identify valid input/field pairs and "confirmation" fields with high accuracy. Using our estimates from the Naive Bayes classifier, we classified whole form inputs with high accuracy.

References

[1] Doupé, A. and Cova, M. and Vigna, G. Why Johnny can't pentest: an analysis of black-box web vulnerability scanners. In: Detection of Intrusions and Malware, and Vulnerability Assessment. pp 111-131 (2010).

[2] Bau, J. and Bursztein, E. and Gupta, D. and Michell, J. State of the art: Automated black-box web application vulnerability testing. In: IEEE Symposium on Security and Privacy. pp 332-345 (2010).

[3] Ng, A. *Lecture Notes 2 - Generative Learning Algorithms*. Available at http://cs229.stanford.edu/notes/cs229-notes2.pdf