Widget Classification with Applications to Web Accessibility

Valentyn Melnyk[1], Vikas Ashok[1], Yury Puzis[2], Andrii Soviak[2], Yevgen Borodin[2], and I.V. Ramakrishnan[2]

[1] Computer Science Department, Stony Brook University, Stony Brook, NY, USA
{vmelnyk,vganjiguntea}@cs.stonybrook.edu
[2] Charmtech Labs LLC, 1500 Stony Brook Rd., Stony Brook, NY, USA
{yury.puzis,and.soviak,borodin,ram}@charmtechlabs.com

Abstract. Once simple and static, many web pages have now evolved into complex web applications. Hundreds of web development libraries are providing ready-to-use dynamic widgets, which can be further customized to fit the needs of individual web application. With such wide selection of widgets and a lack of standardization, dynamic widgets have proven to be an insurmountable problem for blind users who rely on screen readers to make web pages accessible. Screen readers generally do not recognize widgets that dynamically appear on the screen; as a result, blind users either cannot benefit from the convenience of using widgets (e.g., a date picker) or get stuck on inaccessible content (e.g., alert windows). In this paper, we propose a general approach to identifying or classifying *dynamic* widgets with the purpose of "reverse engineering" web applications and improving their accessibility. To demonstrate the feasibility of the approach, we report on the experiments that show how very popular dynamic widgets such as date picker, popup menu, suggestion list, and alert window can be effectively and accurately recognized in live web applications.

Keywords: web applications, reverse engineering, widget classification, widget localization, dynamic widgets, screen reader, web accessibility, ARIA.

1 Introduction

The Web has permeated many aspects of our lives; we use it to obtain and exchange information, shop, pay bills, make travel arrangements, apply for college or employment, connect with others, participate in civic activities, etc. A 2012 report by the Internet World Stats shows that Internet usage has skyrocketed by more than 566% since 2000, to include over a third of the global population in 2012 (over 2.4 billion people) [23]. However, over this time period, the Web has evolved from text-based web pages to interactive web applications, becoming less accessible to blind people.

Many popular websites such as Blackboard, Gmail, Linked-In, Google Drive, eBay, Kayak, YouTube, etc. have turned into sophisticated web applications that utilize dynamic (appearing and disappearing) widgets such as dropdown menus, date pickers, suggestion boxes, etc. to enhance user experience by adding convenient tools.

S. Casteleyn, G. Rossi, and M. Winckler (Eds.): ICWE 2014, LNCS 8541, pp. 341–358, 2014.

According to the W3C definition [31], a *widget* (called "widget" thereafter) is defined as a "discrete user interface object with which the user can interact". Widgets can be simple objects including standard HTML controls with a single value or operation (e.g., buttons and textboxes) or they can be complex objects (e.g., trees).

Hundreds of web development libraries and toolkits (e.g., [10, 13, 28] to mention a few) are providing an ever growing number of ready-to-use web widgets that can be further customized by web developers to fit the needs of individual web sites. Unfortunately, the diversity of the libraries and a lack of standardization and enforcement of W3C specifications have proven to be an insurmountable problem for blind users.

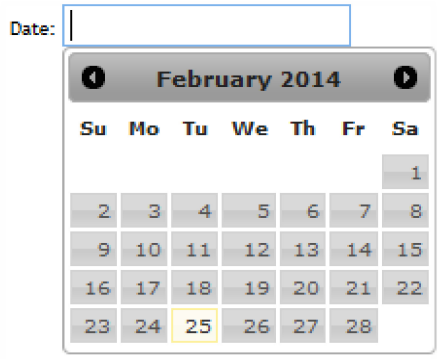
For web browsing, blind people employ screen readers (*e.g.*, JAWS [17], Windows-Eyes [33], VoiceOver [30], Dolphin [27], Sa To Go [26], NVDA [24], etc.), which convert the Web to speech, generally ignoring layout and graphics, and reading aloud all the textual content in web pages. Screen readers enable their users to listen to and navigate web content sequentially in the order it is laid out in the HTML source code, which often does not correspond to visual layout. Screen readers provide many shortcuts to navigate among elements of a particular type, e.g., links, buttons, edit fields, etc. Although not very efficient for web browsing [3], screen readers enable visually-impaired people to browse the Web and perform online activities.

Unfortunately, screen readers do not recognize widgets that dynamically appear on the screen, so the user has no easy way to find them; at best, a dynamically appearing widget will be "navigable," meaning that it can be found and narrated by the screen reader, but without giving any indication as to what kind of widget it is. As a result, blind users either cannot benefit from the convenience of using widgets (e.g., use a date picker) or they even get stuck on inaccessible content (e.g., an HTML alert window). So, while sighted people can enjoy Rich Internet Applications (RIA), blind people either cannot access them at all (e.g., cannot use Google Docs) or have to use basic versions of the websites (Gmail and Facebook).

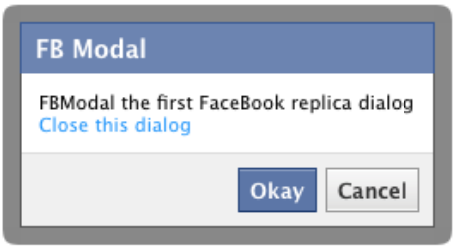
To make web applications more accessible, web developers have to follow Accessible Rich Internet Applications (ARIA) specifications [32]. For instance, ARIA allows developers to mark up live regions where the content may update, specify the importance of those updates, and provide simple roles such as "progress meter." Unfortunately, web developers do not follow ARIA specifications consistently, and ARIA does not have predefined roles for complex widgets such as date picker.

In this paper, we propose an ARIA-independent approach towards improving the accessibility of dynamic widgets. Specifically, the contribution of this paper is a scalable machine learning approach to identification/classification of *dynamic* widgets. To demonstrate the feasibility of the approach, we demonstrate high accuracy in classifying popular dynamic widgets such as date pickers, popup menus, suggestion lists, and HTML alert windows. Sample screenshots of these widgets are shown in Fig. 1.

The identification of *static* widgets (the ones that are already in the web page) and the design and evaluation of accessible user interfaces for widgets are beyond the scope of this paper, as these have been well explored in the literature, e.g., [7, 29].



a) Date Picker



b) Alert Box

c) Suggestion Box

d) Popup Menu

Fig. 1. Sample screen shots of the four widget types collected in the corpus: a) Date Picker, b) Alert window, c) Suggestion Box, and d) Popup menu

In the remainder of this paper, we provide some background on the complexities of dealing with dynamic widgets as well as on the specifics of using ARIA in Section 2; we review prior work relevant to widget localization and classification in Section 3; we describe the setup of the experiments in Section 4; we present and discuss the obtained results in Section 5, and, finally, we conclude by providing the broader impact view of this work and propose future directions of research in Section 6.

2 Background

In order to help understand the technology behind dynamic widgets, and thus the complexity of identifying them, we give a short overview of a variety of methods employed for making the widgets "appear" and "disappear" in web pages. We also provide background on how ARIA [32] can be used to mark up widgets in a way that would make them accessible with screen readers.

2.1 Methods for Displaying Dynamic Widgets

The most straightforward approach to displaying a new widget on the screen is to use JavaScript to insert a new sub-tree representing the widget into the HTML DOM tree. However, in order for this newly inserted widget, or any other widget that is already a part of the DOM tree to appear on the screen, the web developer must set a long list of widget's properties. Modifying any one of them can make the widget invisible or visible for the user. The following is a non-exhaustive list of "tricks" that web developers use to make the widget "appear" or "disappear" (this list was collected from HTML specification [16], numerous developer forums, and confirmed by observing the behavior of hundreds of widgets in the wild):

- Coordinates can be set to a negative value (off-screen) to hide a widget, and set to a positive value to show a widget within the webpage viewport;
- Coordinates can be set to place a widget within or outside of the viewport of a containing object, which can also be another widget;
- Dimensions of a containing object's viewport can be set in a way that the contained widget is hidden or shown on the screen;
- The width and height of a widget can be set to a value that would make it big enough to be visible or make it so small that it disappears;
- The widget's color can be set to blend in with the background or make it stand out;
- The font can be set to a human-readable size or to 0 to make the textual content of widgets appear or disappear;
- The opacity of the widget can be set to make it transparent or opaque;
- The "visibility" or "display" style of the widget can be set as desired;
- The Z-index of the widget can be set to place it behind or in front of other objects.

All these properties can be modified directly in the HTML code of the widget or indirectly, e.g., by changing the widget's "style" attribute. Further, some of these properties (e.g., "display") are inherited from ancestor DOM nodes, so adding a widget as a sub-tree to the DOM will immediately result in the application of ancestor's properties unless they are overridden.

The code and content of widgets can be in the HTML, Cascading Style Sheets (CSS), in JavaScript, or it can be delivered to the webpage via AJAX from the web server. Given the variety of ways to hide and show a widget and deliver it to the web page, the only good way to detect the appearance of dynamic widgets is by monitoring the changes in the DOM tree [12], as we describe in Section 4.1.

2.2 Screen Readers and Accessible Rich Internet Applications (ARIA)

As the blind user is "navigating" on a webpage, screen readers maintain the virtual cursor pointing at the currently narrated object. When dealing with widgets, screen-readers have three main tasks: 1) allow the user to move a virtual cursor from widget to widget, and from one atomic object on the webpage to another; 2) announce relevant information at the current (new) virtual cursor position, including: textual content of the widget, semantics or purpose of the visited widget (e.g., text box, alert window, etc.), its state (e.g., "blank", "disabled", etc.), and the results of user actions affecting the widget (e.g., typed letter, selected item, etc.), and 3) announce important changes to the webpage not localized at the current virtual cursor position.

Unfortunately, the best screen readers can only accomplish the tasks above if widgets are marked up following the Accessible Rich Internet Applications standard (ARIA) [32]. And the responsibility for making web content accessible lies entirely on web developers most of whom take the off-the-shelf ready-to-use widgets from a widget library. Alas, ARIA is not widely followed by web developers or even by screen reader developers. Furthermore, due to its complexity, it is often incorrectly implemented by web developers, which makes the accessibility problem even worse.

Making a Widget Accessible with ARIA. ARIA markup is not necessary for making standard widgets (e.g., textbox, link, listbox, button, etc.) accessible, because screen readers already have a long list of sophisticated hardcoded interaction rules that enable: a) virtual cursor's movement between widgets, b) user interaction with the widgets, and c) relevant announcements that are made to the user when the cursor arrives or leaves the widget or changes its value.

For custom widgets which model the standard widgets, web developers can use the ARIA roles, states, and properties that enable the screen reader to map the custom widget to a standard widget and activate corresponding hardcoded rules. For example, a checked checkbox implemented using JavaScript and html tag "div" can be marked up with attributes role="checkbox" and aria-checked="true". When the checkbox is unchecked, the author needs to use JavaScript to set aria-checked="false".

For custom widgets that model a combination of standard widgets web authors can use a complex combination of ARIA roles, states and properties. ARIA authoring practices can be found in [19], and a good overview of accessible widgets in [21].

For custom widgets that, due to their functional complexity or uniqueness, cannot be mapped to any standard widget (e.g., calendar), web developers need to: a) inform the screen-reader that this is a custom widget without explicit mapping to a standard widget by specifying the widget's role as "application"; b) process relevant keyboard commands, including those that would normally be handled by the screen-reader (e.g., arrow keys), and map those keys to the widget's functionality; c) if the widget consists of multiple interactive components (e.g., a grid with editable grid cells), the web author needs to assign some shortcuts to move the browser's focus between all those components (this will hint the screen-reader to follow the focus with the virtual cursor, and narrate the relevant content).

To announce changes to the webpage not localized at the current virtual cursor position the authors can use ARIA Live Regions; designated by attribute “live”, it can be set to one of four politeness levels (“off”, “polite”, “assertive”, “rude”) determining the urgency (importance) of the changes in the region. The attribute “relevant” is used to set the relevance of specific types of DOM changes within the live region.

The Problems with Using ARIA. Using standard widgets or a custom implementation of standard widgets enables the screen-reader to implement a powerful, accessible interface to a very limited set of widgets. This requires no or little overhead for the web developer, but, in case of custom widgets, this often depends on the diligence in assigning roles and dynamically modifying attributes as a result of user interactions.

Using a complex combination of ARIA roles and states to implement a single widget has the potential to expand the range of functionality and types of widgets supported by the screen-readers. However, this approach requires advanced understanding of ARIA by the developer, increases cost of implementation and support, depends on advanced support of ARIA by screen-readers, which is not yet available for the full ARIA specification, and is limited by the scope of ARIA specifications that do not cover all possible types of functionality. As a result, the use of ARIA in complex widgets (e.g., date picker, etc.) to make them accessible is not widely used.

The use of the “application” role in combination with JavaScript code for processing keyboard events and controlling browser focus has the potential to enable even more complicated widgets than the other approaches. However, predictably, this requires even more effort on the part of the developers than the purely ARIA approach, and, hence, this approach is not widely deployed either.

Issues with ARIA live regions, including difficulties with determining causality of the region updates, giving developers the ability to combine technically discrete but semantically atomic updates, handling interim updates, and providing higher-level abstractions for web developers, are well summarized in [29].

Overcoming the Widget Accessibility Problems. In order to make a widget accessible, it needs to be first localized and identified. Once localized, the widget needs to be enabled for screen reader interaction which can be accomplished either by injecting ARIA into the widget components [9, 18] or by enabling the screen-reader to map the widget (or its components) to the hardcoded interaction rules that will make this widget accessible, similar to the way it is done with the standard widgets. The latter approach is both more powerful than ARIA injection (because it is not restricted by the expressively of the ARIA specifications) and is less prone to errors since it is not restricted by the ability of other screen-readers to correctly interpret ARIA.

In this paper, we focus on the classification or identification of dynamic widgets, which is the first step toward making them accessible. Customizing and evaluating user interfaces for widgets [7, 29] is not in scope of this paper. However, it is notable that the proposed method does not limit the way widgets will be made accessible.

3 Related Work

Many researchers have recognized the accessibility problems caused by dynamic content and dynamic widgets (those that dynamically appear and disappear on the screen), as well as the deficiency of screen readers in handling this problem [1, 2, 4, 6, 8, 9, 11, 14, 18, 20]. Several approaches were proposed to make dynamic content and widgets accessible, localize widgets, and classify the widgets into types.

3.1 Making Dynamic Widgets Accessible

The majority of the approaches described in the literature focused primarily on retrofitting the existing web application by adding ARIA markup [32] to the web page, while a few attempted to provide a custom screen-reader interface.

An early approach proposed by the authors [4] enabled users to review any dynamic updates on the web page using a special layered view isolating the changes. While useful, that approach, however, did not help identify the type of changes or their importance; and, without the ability to isolate and classify the type of widgets, it is impossible to provide a usable interface that works best with a particular widget.

For example, if a slideshow widget changes, unless the screen-reader user is on it, it is a low priority event that should not interrupt what the user is listening to. On the other hand, if an alert window appears, the event has to be audibly announced, screen-reader navigation should only be available within the scope of the window, and the text of the window needs to be read out. If a date picker appears, current date has to be announced, and table navigation (left/right/up/down) has to be enabled in the calendar.

AxsJax [9] was one of the pioneering approaches that showed (using the example of Google chat) how web developers can make their web applications more accessible by injecting ARIA metadata into AJAX (Asynchronous JavaScript) responses sent from the server to the web browser. The injection would happen on the server-side and would have to be enabled by the web developers. However, if the developers were motivated to make their websites accessible, they would have ARIA from the start. So, AxsJax, does not fix the accessibility problem, but rather provides web developers with an alternative method for making their web applications accessible.

Single Structured Accessibility Stream for Web 2.0 Access Technologies (SASWAT) [18] is a project performing the AJAX injection on a proxy server. This releases the web developers of responsibility and puts it into the hands of the SASWAT supporters, who would have to provide the ARIA metadata describing the dynamic content and store it into a repository. Then, if any of the specified webpages were loaded through the proxy, the proxy would inject the ARIA metadata into the webpages, thus making them accessible. Of course, this approach breaks down if the web pages are changed by the developers and it is not scalable across websites, because any variation of a widget would require modifications of the scripts. A truly scalable approach would require automatic widget localization and classification as we discuss in the following section.

3.2 Widget Localization and Classification

In [1], authors propose an approach to widget security vulnerability detection. For this purpose, the authors locate widgets using DOM dynamic updates analyses. They track the parts of the DOM, which were changed as a result of some user action or JavaScript event. Later, they test identified widgets for inter change, i.e. when one widget is updated automatically by another widget. While there is some similarity in tracking mutation events used in this paper, authors do not care about the types of widgets.

The approach proposed in [2] utilizes end-user-programming to enable automation and customization of web application. Based on the selected keywords, one can build a pattern to localize a widget on the web page, which could potentially be reused across websites. However, this approach only works for simple widgets (search box or a button) and requires manual effort for selected keywords for a particular widget.

Several research teams have looked into the possibility of recognizing complex dynamic widgets. For instance, [7] has explored the possibility of detecting calendar widgets; however, the approach was a special-case heuristic classification that detected the calendar based on user interaction, which is not scalable to other widget types.

An early attempt to classify different widgets was made in [8]. The proposed algorithm analyzed webpage sources (HTML, JS, and CSS) using regular expressions to find the display window (the area containing the widget) and then matching the content to an ontology with predefined widget features. Unfortunately, the classification was done by constructing a hierarchy of widgets based on widget implementations in specific libraries, which means that the classification will work only for the widgets taken from these libraries. Also, the ontology has to be created manually, and the approach will not be able to detect the (dis)appearance of a dynamic widget.

A more dynamic approach to desktop widget classification is proposed in [11]. The main goal of the paper is to simplify a desktop interface and make it "visually accessible". The system is built as an extension of the Prefab pixel-based recognition system. The system recognizes some simple widgets such as "Windows 7 steal buttons" based on the visual markers. Besides requiring compute-intensive vision based analysis of the screen, unfortunately, this approach is also limited by the visual distinction between widgets. For example, it will not able to distinguish a suggestion box from the popup menu because both look like popups. Neither will it be able to find a widget if it is hidden behind some other control such as a drop down menu.

In contrast, the method proposed in this paper overcomes the limitations listed above by using a more general and lightweight approach to widget classification. It employs machine learning to classify widgets automatically regardless of the library they come from. Since a variety of features can be extracted from the DOM tree (Section 4.2), the approach can classify widgets even if they are visually similar, e.g., based on trigger events. Furthermore, once the widgets are classified, this information can be used either to inject ARIA metadata into the webpage or provide this information directly to the screen reader to provide customized interaction with the widget.

The limitation of the proposed approach is that it can only identify *dynamic* widgets that appear and disappear in web pages. However, some of the reviewed methods [2, 8, 11] can be employed to detect *static* widgets that are already on the web page.

4 Experimental Setup

4.1 The Corpus and Widget Localization

To experiment with widgets, we collected a corpus with four types of popular widgets (suggestion list, HTML alert window, popup window, and date picker, shown in Fig. 1) with 50 examples of each widget type. To this corpus, we added an additional 50 examples of other randomly selected widget types; we refer to this generic widget type as "others". The corpus was collected from widget libraries and live websites using custom tool developed specifically for this purpose.

The data collection tool, based on the Capti Narrator (www.captivoice.com) for Mac/Windows [5], consists of a Firefox browser extension and a Java application. The browser extension listens to all DOM mutation events (updates), and communicates them over an open socket to the Java application. Java application uses the updates to construct a timeline of DOM mutations, reconstructs the DOM at any given point of time, and displays it in a separate window, in the form of a tree. Once a website with a widget has loaded, the process of collecting the data of that single widget is semi-automatic:

1. Press a button control shortcut to start "recording" all DOM mutation events;
2. Trigger opening of the widget and wait for it to open (usually instantaneous), e.g., press a button opening an alert window, focus on a textbox with a date picker, etc.;
3. Press the same button to terminate the "recording" of DOM mutations;
4. Verify that the recorded DOM mutations represent the target widget;
5. Save the resulting timeline of DOM mutations into the corpus of widgets.

The recorded timeline spans the period of "recording" and includes only the events in two sub-trees: one representing the trigger object and the other representing the widget that opened as a result of the trigger; all other events are ignored. The data collection tool was designed with the following considerations in mind.

A webpage may have many scheduled DOM mutation events, so it is very important to localize the relevant DOM mutations. While JavaScript is executed synchronously in most browsers, the exact localization of the mutation events relevant to a particular widget is an unsolved problem; multiple unrelated mutation events can happen immediately one after another. So, automated analysis of the underlying JavaScript would be required to understand the relationship between the user action (e.g., pressing a shortcut) and the system reaction (e.g., displaying a widget).

However, in practice, a heuristic approach that uses both temporal and spatial information helps minimize the risk of collecting irrelevant mutation events. Specifically, any user event such as focus change or control activation can be considered to be the potential trigger event, starting an observation period. Any subsequent DOM mutation events occurring within time t of the trigger event can be considered candidates for the widget. If more than one DOM sub-tree has updated, the collected mutation events can be further filtered by their spatial proximity to the trigger object.

4.2 Features for Widget Classification

The selection of features was inspired by the observations made during a manual inspection of the corpus. The vector representation of features was assembled from the features extracted from four different categories listed in Table 1. Most of the examined features are binary with the exception of the "Proportion of text nodes with only numbers in them" and the "Number of text nodes".

Table 1. Feature space for widget classification

Feature	Description	Binary
PRESENCE OF HTML TAGS & KEYWORDS		
P_{table}	Presence of table tag <table> in the HTML associated with widget	Yes
P_{list}	Presence of list related tags like , , etc, in the HTML associated with widget	Yes
$P_{textbox}$	Presence of textboxes (e.g. <input type= "text">) in the HTML associated with the widget	Yes
P_{name}	Presence of widget name in "class" attribute of any tag in the HTML associated with the widget	Yes
P_{date}	Presence of "date" as the value of "type" attribute in any tag in the HTML associated with the widget	Yes
P_{image}	Presence of an image (<img>) in the HTML associated with the widget	Yes
CONTEXT RELATED		
T_{link}	Widget appears due to click of a button or link	Yes
T_{input}	Widget appears due to a keyboard entry in an input box	Yes
CHARACTERISTICS OF NODES IN WIDGET DOM SUBTREE		
$C_{text.num}$	Number of text(<text>) nodes	No
$C_{text.prop}$	Proportion of text (<text>) nodes containing only numbers in them	No
$C_{table.list}$	A table (<table>) or list ( ) is present and over 80% of its content are links	Yes
DISPLAY PROPERTIES OF WIDGET		
D_{widget}	Widget appears right below the "triggering" element	Yes

Presence of HTML Tags and Keywords (P). Analysis of the corpus revealed that, in some widgets, certain HTML elements are almost always present. For example, a list of links (
…) can be found with high probability in a popup menu, a table (<table>) is likely to be present in a date picker widget to format the calendar, and an input textbox is always a part of suggestion list. In addition to HTML tags, attribute values can also be used to identify the widget. For example, we observed that, in many cases, the "class" attribute of one of the or HTML tags contained the name of the widget (e.g., "suggestion-box", "date-picker", etc.).

Context-Related Features (T). The local context surrounding a widget provides valuable information and important cues for identifying that widget. We refer to any HTML element that causes the appearance of a widget as a "trigger". The fact that different widgets are triggered in different ways by different HTML elements can be exploited to improve widget classification performance. For example, a suggestion box is almost always triggered when a user types something in an input text box; a menu popup appears on-screen when a user presses the corresponding button; a date picker widget appears when the user goes in focus on the textbox, etc.

Characteristics of Nodes in Widget DOM Sub-tree (C). This set of features was crafted after an extensive analysis of the DOM sub-trees corresponding to different widgets in the corpus. These features strive to leverage differences in the composition of DOM sub-trees corresponding to different widgets. For example, we observed that the DOM sub-tree of a suggestion box contains relatively higher number of <text> nodes compared to the DOM sub-tree of an alert box. Variation in composition also includes the type of content stored in the DOM nodes. For example, a list in a menu pop-up widget is likely to contain a high percentage of links, whereas a list in a suggestion box widget is likely to contain a large number of text nodes.

Display Characteristics of Widget (D). The position of a widget on the screen provides an important cue for its classification. Specifically, we are interested in the screen location of the widget relative to its triggering HTML element. This is based on our observation that different widgets exhibit different display patterns relative to their corresponding trigger. For example, in our corpus, the suggestion box widget appeared right below the triggering input textbox 100% of the time, and the menu pop widget always appeared either below or to the right of the corresponding triggering button or link. This feature is especially useful to filter out "irrelevant" dynamic mutations that can happen in the same time frame when the widget appears.

Having identified the salient features that could help distinguish different widgets, we conducted experiments to identify which combination of these features with which machine learning tools would yield the best widget classification results.

5 Experiments and Results

To conduct the experiments, we used the Weka [15] toolkit. We considered several popular machine learning classifiers and selected the following classifiers that yielded

the best performances: Support Vector Machine, frequently used for benchmarking, and the J48 Decision Tree classifiers, which is a simple rule-based classifier that is appropriate for the mostly binary widget features.

As described in the previous section, we divided the features into five thematic categories according to the type of the feature. In order not to evaluate each feature separately, we combined the categories into five groups (Groups 1-5 in Table 2) in a way that would allow us to evaluate the impact of each individual category on the results. For example, Presence (P) features are absent in Group 1, Context (T) features are absent in Group 2 and so on. Group 5 has all the feature sets, and hence the performance of group 1-4 can be compared with Group 5 to assess the importance of the corresponding missing feature set.

Table 2. Feature groups used for widget classification

Group	Features
Group 1	Context (T) + Characteristics (C) + Display (D)
Group 2	Presence (P) + Characteristics (C) + Display (D)
Group 3	Presence (P) + Context (T) + Display (D)
Group 4	Presence (P) + Context (T) + Characteristics (C)
Group 5	Presence (P) + Context (T) + Characteristics (C) + Display (D)

Finally, we ran both classifiers on each of the five groups of features. We used 5-fold cross validation: 200 widget examples (40 of each type) were used for training and 50 (10 of each type) for testing, repeated 5 times with different divisions into folds. The results of the experiments are detailed in Table 3; the winning group-widget type combinations are in bold.

The absolute winners (J48 on Groups 4 and 5) have shaded background. As can be seen from the averages in Table 3, SVM and J48 classifiers yielded similar performance: SVM showed highest performance: 86% recall and precision in groups 4 and 5, while J48 won by a single percent point in both precision and recall. In all groups, J48 was performing better than SVM on average. In general, SVM yielded slightly better performance than J48 in identifying Date Pickers, but J48 was better at distinguishing Alert Boxes and Popup Menus. The absolute best performance was shown by J48 in Groups 4 and 5, yielding precision: 87% and recall: 87%, beating the SVM by 1% in both precision and recall.

Group 5 yielded the best average performance (Precision: 91%, Recall: 94%) when the generic widget type "*Others*" was excluded from the analysis, beating Group 4 (Precision: 90%, Recall: 94%) by a narrow margin of 1% in precision. In all of the feature groups, the average precision excluding "*Others*" is only slightly better than average precision with "*Others*". However, the average recall excluding "*Others*" is significantly better than the overall average recall, in all of the feature groups. These performance results demonstrate the effectiveness of our models in accurately identifying the 4 core widget types considered in our work.

Table 3. Widget classification results, Notation: P - Precision, R - Recall, J48 - Decision Tree, SVM - Support Vector Machine; the values in bold indicate the best performances per group

Widget	Classifier	Group 1		Group 2		Group 3		Group 4		Group 5	
		P	R	P	R	P	R	P	R	P	R
Suggestion box	SVM	**0.96**	**0.96**	0.81	0.88	0.76	0.68	**0.96**	**0.96**	**0.96**	**0.96**
	J48	**0.96**	**0.96**	0.80	0.96	0.94	0.58	**0.96**	**0.96**	**0.96**	**0.96**
Alert Box	SVM	0.65	0.76	0.73	0.90	0.67	0.84	0.74	0.94	0.77	0.94
	J48	0.92	0.78	0.84	0.78	0.76	0.84	**0.83**	**0.90**	0.85	0.88
Menu Popup	SVM	0.47	0.68	0.82	0.82	0.82	0.82	0.82	0.82	0.82	0.82
	J48	0.61	0.62	0.79	0.88	0.80	0.80	0.83	0.90	**0.85**	**0.90**
Date Picker	SVM	**0.98**	**1.00**	**0.98**	**1.00**	0.61	0.64	**0.98**	**1.00**	0.97	1.00
	J48	0.96	1.00	0.96	1.00	0.61	0.82	0.96	1.00	0.96	1.00
Others	SVM	0.70	0.24	0.59	0.36	0.64	0.44	0.79	0.54	0.80	0.56
	J48	0.59	0.60	0.58	0.42	0.62	0.54	**0.78**	**0.60**	0.74	0.60
Overall Avg.	SVM	0.75	0.73	0.78	0.79	0.70	0.68	0.86	0.85	0.86	0.86
	J48	0.81	0.79	0.79	0.81	0.75	0.72	**0.87**	**0.87**	**0.87**	**0.87**
Avg. excl. Others	SVM	0.77	0.85	0.83	0.90	0.72	0.75	0.88	0.93	0.88	0.93
	J48	0.87	0.84	0.85	0.91	0.78	0.76	0.90	0.94	**0.91**	**0.94**

Group 3, missing the Characteristics (C) features had the worst results both with the SVM (average P: 70%, R: 68%) and J48 (average P: 75%, R: 72%) classifiers. This shows that Characteristics (C) features were very important for classification of widgets in general. The results of both Group 1 and Group 2 are also significantly worse than Group 5, thereby demonstrating the importance of Presence (P) and Context (T) features. However, the absence of Display (D) feature (Group 4) did not cause any significant drop in performance (compared to Group 5 containing D feature). Overall, it can be inferred from Table 3 that feature sets P, T and C are critical for high performance, while the feature set D has minimal impact on the performance.

Notice in Table 3 that the accuracy in identifying Suggestion Boxes is heavily dependent on the T and C features (there is a significant drop in performance in Groups 2 and 3 compared to Group 5), while the performance is least influenced by the P and D features (Groups 1 and 4 yield the same performance as Group 5). Similarly, in case of Date Pickers, the C features were seen to contribute the most towards performance improvement compared to P, T and D features.

Table 4 lists the top discriminatory features for each widget as determined by the SVM classifier. As expected, T_{input} is the topmost predictive feature for Suggestion Box widget since all suggestion boxes are always activated when a user types something in an input textbox (often a search box). Also, observe that the feature P_{image} is also highly predictive of Suggestion Boxes, which indicates that a lot of websites provide suggestion boxes that contain image icons in addition to the suggested links or text. An example of such a suggestion box is shown in Fig. 1(c).

Table 4. Top discriminatory features as determined by the SVM classifier weights. According to [25], high positive weights indicate high predictiability of the corresponding class.

Widget	Top Discriminatory Features
Suggestion Box	T_{input}, P_{list}, P_{image}, $C_{table.list}$
Alert Box	P_{image}, T_{input}, $P_{textbox}$
Menu Popup	P_{list}, P_{name}, $C_{table.list}$
Date Picker	$C_{text.prop}$, P_{table}, T_{input}, P_{name}, $C_{text.num}$, $C_{table.list}$

It can be also seen in Table 4 that P_{name} is predictive of Menu Popup and Date Picker widgets, but not the other two widgets, thereby, highlighting a difference in the way these types of widgets are implemented with reference to CSS; compared to Alert windows and Suggestion boxes, a higher percentage of Popup Menu and Date Picker widgets have at least one node in their DOM sub-trees storing the corresponding widget name as a class attribute. Also, as expected, P_{list} is a top discriminatory feature for Menu Popup, since almost all pop-up menus contain a list of selectable items.

It can be inferred from Table 4 that overall, the features related to the DOM subtree characteristics (C) are extremely useful for widget classification.. This claim is supported by two observations: (i) Feature Group 3 yielded the lowest performance among the feature groups as previously noted in Table 3 analysis, and (ii) 3 out of 4 classes in Table 4 have at least one top discriminatory feature belonging to this category, e.g., $C_{text.num}$ points to variations in textual composition of different widgets.

Figure 2 depicts the decision tree (considering the entire dataset and all the features) produced by the J48 decision tree algorithm supported by WEKA toolkit. It can be inferred from Figure 2 that the feature $C_{text.prop}$ is the most important feature for identifying Date Picker widget type. More specifically, the proportion of text nodes with only numbers, in all Date Picker widgets in the dataset was above 0.21. Only one other widget of a different type in the dataset had the value of $C_{text.prop}$ greater than 0.21 (The label '51/1' of Date Picker leaf node in Figure 2 indicates that out of 51 data points placed in that group, 1 of them is incorrectly classified).

Similar inferences can be made from the decision tree in Figure 2 with respect to other widget types. For example, it can be observed that the context feature T_{input} is critical for correctly identifying the Suggestion Box widget type. Recall that even the SVM classifier determined T_{input} to be the most discriminating feature for identifying the Suggestion Box widget type (Table 4).

Similarly, it can be seen that P_{list} is the most important feature required for the accurate classification of Menu Popup widget type. Also observe in Figure 2 that the feature P_{name} plays an important role in identifying those Menu Popup data points that are not covered by P_{list}. These observations are in accordance with the SVM results presented in Table 4 where P_{list} and P_{name} are the top two discriminating features for Menu Popup widget type. However, no such straightforward comparisons between SVM and J48 decision tree results can be made with respect to the Alert box widget type as it can be observed in Figure 2 that the Alert Box widget type relies on different combinations of a wide variety of features for their accurate identification.

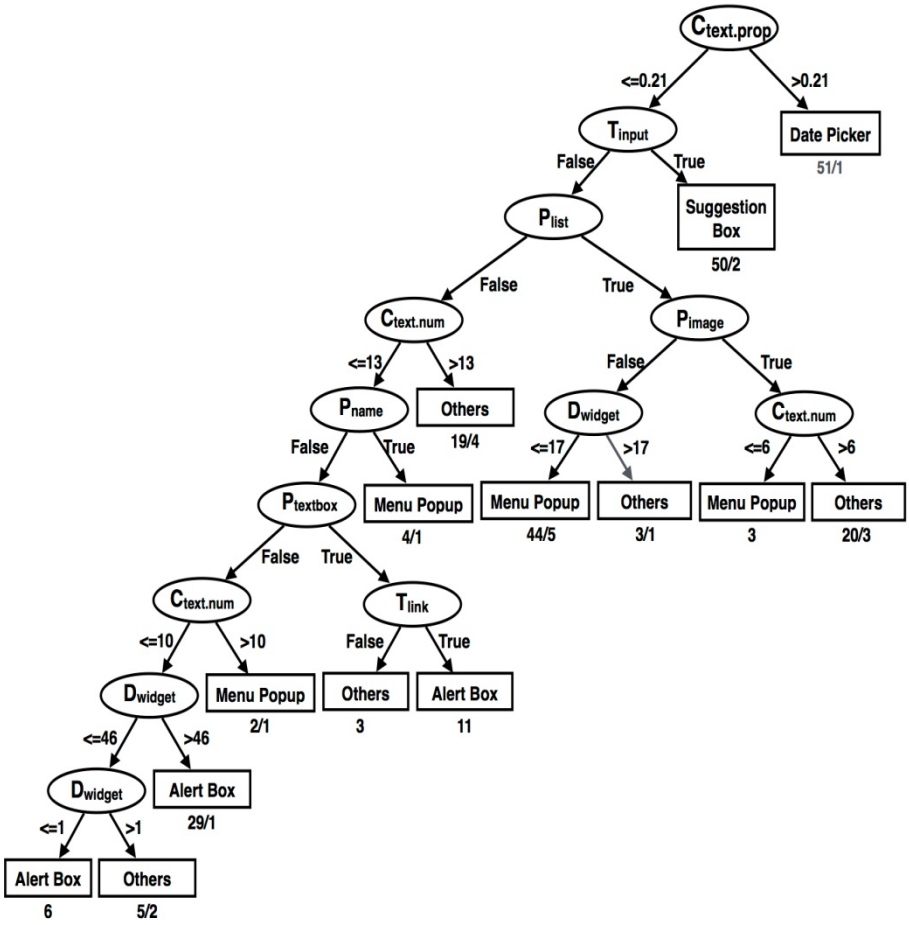


Fig. 2. Decision tree produced by the J48 algorithm on the entire dataset for feature group 5. Each leaf label indicates total classified data points followed by total incorrect classifications.

The importance of each group of features is apparent from the decision tree in Figure 2, where all features except P_{table} and P_{date} are used for classification. The absence of P_{table} in Figure 2 is a bit surprising since P_{table} was determined to be one of the top discriminatory features for Date Picker widget type. This observation adds weight to our earlier deduction that $C_{text.prop}$ is the single most important feature for identifying Date Picker widgets.

6 Conclusion and Future Work

In this paper, we have proposed and evaluated a scalable method for classification of dynamic web widgets using machine learning. The experiments on a corpus of 250 widgets showed that the decision tree learning was the most accurate machine learning technique for identifying and distinguishing among four popular types of

widgets: the popup menu, the HTML alert window, suggestion box, and data picker (Fig. 1).

To date, there exists no assistive technology capable of enabling consistent and usable interaction with dynamic content and dynamic widgets. As was discussed in Sections 2 and 3, the existing solutions are very limited and are inadequate given the continuing and rapid adoption of dynamic web technologies. The ability to access the same web applications that are available to sighted people will enable people with vision impairments to enjoy the latest assistive technology, making them more productive. The approach proposed in this paper is the first step towards making widgets accessible, requiring further research and development in this direction.

While this paper handled four popular types of dynamic widgets, there are many more types of dynamic widgets that are used more rarely, but are used nonetheless. Although the proposed approach is scalable for more types of widgets, an extensive dataset has to be first assembled in order to handle more types of widgets. New approaches to dynamic widget localization have to be explored and tested. A reliable method needs to be developed for identifying static widgets that are already in the web page as soon as it loads.

In parallel, automatically identified widgets need to be made accessible to screen readers. While injecting ARIA (Section 3.1) may provide general accessibility to all screen readers supporting ARIA, it is also possible to embed the method described in this paper into a screen reader. The latter approach will allow for a more powerful user interface that is not possible given the limited expressivity of ARIA. Longitudinal user studies will have to be conducted to evaluate the usability of the user interfaces and verify the accuracy of the widget identification approach in the wild.

Finally, the approach proposed in this paper can find use in other web-based application areas. For instance, widget identification can be useful in website crawling [22], website simplification [11], and other reverse engineering of web applications.

Acknowledgements. This work was developed under a grant from the Department of Education, NIDRR grant number H133S130028. However, contents do not represent the policy of the Department of Education, and you should not assume endorsement by the Federal Government. We are also grateful to our Accessibility Consultant, Glenn Dausch, for his insightful feedback on accessibility problems with dynamic widgets.

References

[1] Bezemer, C.-P., Mesbah, A., Deursen, A.V.: Automated security testing of web widget interactions. In: Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, pp. 81–90. ACM, Amsterdam (2009)

[2] Bolin, M., Webber, M., Rha, P., Wilson, T., Miller, R.C.: Automation and customization of rendered web pages. In: Proceedings of the 18th Annual ACM Symposium on User Interface Software and Technology, pp. 163–172. ACM, Seattle (2005)

[3] Borodin, Y., Bigham, J.P., Dausch, G., Ramakrishnan, I.V.: More than meets the eye: a survey of screen-reader browsing strategies. In: Proceedings of the 2010 International Cross Disciplinary Conference on Web Accessibility (W4A), pp. 1–10. ACM, Raleigh (2010)
[4] Borodin, Y., Bigham, J.P., Raman, R., Ramakrishnan, I.V.: What's new?: making web page updates accessible. In: Proceedings of the 10th International ACM SIGACCESS Conference on Computers and Accessibility. ACM, Halifax (2008)
[5] Borodin, Y., Sovyak, A., Dimitriyadi, A., Puzis, Y., Melnyk, V., Ahmed, F., Dausch, G., Ramakrishnan, I.V.: Universal and ubiquitous web access with Capti. In: Proceedings of the International Cross-Disciplinary Conference on Web Accessibility, pp. 1–2. ACM, Lyon (2012)
[6] Brown, A., Jay, C., Chen, A.Q., Harper, S.: The uptake of Web 2.0 technologies, and its impact on visually disabled users. Univers. Access Inf. Soc. 11(2), 185–199 (2012)
[7] Brown, A., Jay, C., Harper, S.: Audio access to calendars. In: Proceedings of the 2010 International Cross Disciplinary Conference on Web Accessibility (W4A), pp. 1–10. ACM, Raleigh (2010)
[8] Chen, A., Harper, S., Lunn, D., Brown, A.: Widget Identification: A High-Level Approach to Accessibility. World Wide Web 16(1), 73–89 (2013)
[9] Chen, C.L., Raman, T.V.: AxsJAX: a talking translation bot using Google IM: bringing Web-2.0 applications to life. In: Proceedings of the 2008 International Cross-Disciplinary Conference on Web Accessibility (W4A). ACM, Beijing (2008)
[10] DevExpress. DevExpress Widget Library (2014), https://www.devexpress.com (cited 2014)
[11] Dixon, M., Leventhal, D., Fogarty, J.: Content and hierarchy in pixel-based methods for reverse engineering interface structure. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pp. 969–978. ACM, Vancouver (2011)
[12] DOM. W3C Document Object Model (2004), http://www.w3.org/DOM/DOMTR (cited 2010)
[13] Google. Google Web Toolkit (2014), http://gwt-ext.com/demo/ (cited 2014)
[14] Hailpern, J., Guarino-Reid, L., Boardman, R., Annam, S.: Web 2.0: blind to an accessible new world. In: Proceedings of the 18th International Conference on World Wide Web, pp. 821–830. ACM, Madrid (2009)
[15] Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H.: The WEKA Data Mining Software: An Update. SIGKDD Explorations (2009)
[16] HTML5. Hyper-Text Markup Language v.5.0 (2010), http://dev.w3.org/html5/spec/ (cited 2010)
[17] JAWS. Screen reader from Freedom Scientific (2013), http://www.freedomscientific.com/products/fs/jaws-product-page.asp (cited 2013)
[18] Jay, C., Brown, A.J., Harper, S.: Internal evaluation of the SASWAT audio browser: method, results and experimental materials, The University of Manchester (2010)
[19] Joseph Scheuhammer, M.C.: WAI-ARIA 1.0 Authoring Practices (2013), http://www.w3.org/TR/wai-aria-practices/ (cited 2014)
[20] Linaje, M., Lozano-Tello, A., Perez-Toledano, M.A., Preciado, J.C., Rodriguez-Echeverria, R., Sanchez-Figueroa, F.: Providing RIA user interfaces with accessibility properties. Journal of Symbolic Computation 46(2), 207–217 (2011)
[21] Lourdes, M.: Toward an Equal Opportunity Web: Applications, Standards, and Tools that Increase Accessibility. In: Paloma, M., Belen, R., Ana, I. (eds.), pp. 18–26 (2011)

[22] Mesbah, A., Bozdag, E., Deursen, A.V.: Crawling AJAX by inferring user interface state changes. In: Proceedings of the 2008 8th International Conference on Web Engineering. IEEE Computer Society (2008)
[23] MiniwattsMarketingGroup. Internet Usage Statistics: The Internet Big Picture World Internet Users and Population Stats (2013), http://www.internetworldstats.com/stats.htm (cited 2013)
[24] NVDA. NonVisual Desktop Access (2013), http://www.nvda-project.org/ (cited 2013)
[25] Rayson, P., Wilson, A., Leech, G.: Grammatical word class variation within the British National Corpus sampler. Language and Computers 36(1), 295–306 (2001)
[26] SaToGo, Screen reader from Serotek (2010)
[27] SuperNova. Screen Reader from Dolphin (2013), http://www.yourdolphin.com/productdetail.asp?id=1 (cited 2013)
[28] Telerik. Telerik Widget Library, http://www.telerik.com (cited 2014)
[29] Thiessen, P., Chen, C.: Ajax live regions: chat as a case example. In: Proceedings of the 2007 International Cross-Disciplinary Conference on Web Accessibility (W4A), pp. 7–14. ACM, Banff (2007)
[30] VoiceOver, Screen reader from Apple (2010)
[31] W3C. Important Terms (2014), http://www.w3.org/TR/wai-aria/terms (cited 2014)
[32] WAI-ARIA. W3C Accessible Rich Internet Applications (2013), http://www.w3.org/TR/wai-aria (cited 2013)
[33] Window-Eyes, Screen Reader GW Micro (2010)