On String Prioritization in Web-based User Interface Localization

Luis A. Leiva and Vicent Alabau

PRHLT Research Center, Universitat Politècnica de València
{luileito,valabau}@prhlt.upv.es

Abstract. We have noticed that most of the current challenges affecting user interface localization could be easily approached if string prioritization would be made possible. In this paper, we tackle these challenges through Nimrod, a web-based internationalization tool that prioritizes user interface strings using a number of discriminative features. As a practical application, we investigate different prioritization strategies for different string categories from Wordpress, a popular open-source content management system with a large message catalog. Further, we contribute with WPLoc, a carefully annotated dataset so that others can reproduce our experiments and build upon this work. Strings in the WPLoc dataset are labeled as relevant and non-relevant, where relevant strings are in turn categorized as critical, informative, or navigational. Using state-of-the-art classifiers, we are able to retrieve strings in these categories with competitive accuracy. Nimrod and the WPLoc dataset are both publicly available for download.

Keywords: Localization; L10n; Internationalization; i18n; Translation

1 Introduction

Today most applications are looking forward to being available in more than one language, mainly to reach a global audience, to gain competitive advantage, or just because of legal requirements. In general, there is an increasing and stringent need to adapt any type of software so that it can meet the cultural and linguistic needs of every customer. For instance, according to a recent survey [7], a few years ago web companies would have to translate content into 37 languages to reach 98% of Internet users, and now it takes 48 languages to reach the same amount of users.

Taking a different tack, the "translate frequently and fast" mantra is central to companies seeking to increase global market share. As discussed in the Drupal Translation project,[1] with high-quality translation enterprises can tailor web content to consumers around the world; but when it comes to the "frequently and fast" part of the equation, enterprises run into problems. In fact, one place where localization has always had big problems is within graphical user interfaces [10,

Paper published in Proc. WISE 2014. LNCS 8787.

15]. Later on we discuss the most important of these problems and show that they could be easily approached if string prioritization would be made possible. Consequently, we have developed a method to support this goal.

1.1 Research Goals and Contributions

This paper presents Nimrod, a particular instantiation of our method for PHP-based software. Nimrod is a standalone open-source internationalization tool, however we integrate it on Wordpress to illustrate its capabilities. By means of an intuitive administration panel, webmasters can gain control over different string sorting features. We first investigate the best combination of these features, according to a manually annotated dataset. Next, we approach string prioritization as an information retrieval task, where relevant messages have to be differentiated from the non-relevant ones on the basis of a featurized string representation. Further, we explore different state-of-the-art classifiers to retrieve different categories among the relevant messages. The results show that ours is a valuable new method to improve web-based software localization. Nimrod and our dataset are both publicly available for download.

This paper is organized as follows. Section 2 provides the research background and discusses related work. Section 3 describes our system implementation. Section 4 evaluates the system. Section 5 concludes this paper and provides opportunities for future work.

2 Research Background

When developing software for a global market, applications must follow a two-step process: first internationalization (i18n), then localization (L10n). Internationalization consists of decoupling translatable text out of the application source code, basically by wrapping each message or "resource string" with a translation-capable function. After internationalization, the user interface (UI) is ready to support the requirements of different locales, i.e., specific languages and countries of the target audience; in short, the linguistic preferences that the user wants to see in their UI.

Localization in turn comprises 2 sub-levels: first *language translation*, then *aesthetic adaptation*; being the former the core activity due to its importance and associated costs [6, 8, 13]. Indeed, most companies are well aware of both sub-levels but eventually focus on translation due to time and budget limitations [9, 12]. While aesthetic adaptation can improve the user experience [16], localized applications *must* speak the language of its users. What is more, nowadays that the Internet is pervasive, people use web browsers more than any other class of desktop software. Therefore, multilingual websites and web-based applications, much like any other type of software, are crucial to every player in the industry [12, 18, 19]. Thus, as businesses continue to globalize, localizing web-based UIs becomes more compelling. Last but not least, localization is a unique opportunity of preserving a language [13, 14].

2.1 Related Work

Previous work on how to prioritize UI localization strings is actually scarce. So far the closest attempts we have found are tools that focus on string extraction but little to none perform string prioritization. TranStrL [21] is an Eclipse plugin that takes the source code of a Java application and automatically produces a list of untranslated strings. This solves one of the current challenges UI localization (see Section 2). However, TranStrL is only available for the Java platform. Smartling[2] provides an Objective-C library that achieves the same effect by adding minimal modifications to the source code of the applications. Again, this is not applicable to web-based software. Finally, Globalyzer[3] and World Wide Navi[4] provide a suite of desktop tools to analyze, test, and fix internationalization issues in different programming languages. Unfortunately, none of these tools allows to prioritize localization strings. Neither do current web-based localization tools such as Launchpad,[5] Pootle,[6] Verbatim,[7] or Transifex;[8] not at least to cope with a proper prioritization method as discussed in the next section.

As a mechanism for string prioritization, the GNU gettext manual [10] encourages translators to "use and peruse [sic] the program like a user would do and then use the suite of `msg*` command line tools to translate most urgent messages first." This is a somewhat improved approach, but actually impractical for two reasons. First, by following this approach the roles of translators, end users, and programmers are tightly coupled, which is rather an exception than the norm. Second, string prioritization is done manually with command line tools, which is time consuming. Messages can be sorted according to translation status (untranslated messages first) and frequency (most frequent messages first), based on the analysis of calls to the `gettext()` function and the like, e.g., `t()` in Drupal or `translate()` in WordPress, two of the most popular web content management systems (CMS) today. However, we believe that a more informed method is necessary. Then, one could perform either manual or machine translation (or a combination of both) of the prioritized strings, completing thus the UI localization workflow.

2.2 User Interface Localization Challenges

From the previous discussion, it follows that string prioritization has been overlooked. In order to stress the importance of this topic, here we identify a series of key issues in software localization that are affected by the lack of string prioritization. We believe that understanding these will be useful to the research community and for others trying to build i18n/L10n tools.

Where do we start? Some applications have quite large message catalogs, e.g., most web CMS have well over 3000 messages, but an important number of them are seldom used on the UI. For instance, we have observed that in Wordpress this amounts to roughly half of the total messages. Usually, a translator sometimes has only a limited amount of time per week to spend on a package. Thus, it seems reasonable to start working on the strings that are most frequent.

The trouble with string sorting order. Of particular importance is the fact that strings in a message catalog are sorted according to the source code files instead of the UI views. Actually the order of the strings in the source code is quite apart from what it is shown on the UI. Moreover, the strings that appear on the main UI view are usually more visible than those that appear on less frequented UI views. Therefore, a method to prioritize strings on the basis of the UI view where they appear would be quite a feat for current localization technology. For instance, it would allow more software iterations and faster development cycles by allowing translators to focus on the context of a single UI view.

Lack of contextualization. Message catalogs lack of a proper localization context, which is a two-fold problem. On the one hand, there is not enough context due to the aforementioned sorting order according to source code files. On the other hand, there is no *visual* context available, because UI strings are decoupled from the source code. At best, translators can trust the comments developers may have left to them, such as `"# Translators: This message is related to..."`. Unfortunately, these comments are scarce overall. Thus, those strings with useful comments and/or pointers to the UI should be translated in the first place.

Application updates. Often a new software version comes with a new functionality and new messages attached to it. Moreover, some of the previous strings can be updated. Even if only a few words have changed in the original string, the translator may not see them with current localization tools; therefore she has to proofread the entire message. Consequently, new strings and string updates should be made more prominent to the translator, so that she can localize those messages earlier.

String obsolescence. Sometimes a software patch removes old widgets from the UI but their associated strings are still included in the message catalog. Then, when localizing the software into a new language there is no way to tell those strings apart and thus they would be unnecessarily translated. Therefore, being able to identify this "dead code" would allow translators to work only on what is really needed.

Supporting Agile Localization. On another line, some companies such as Adobe are releasing updated versions of their product multiple times per day [20], making it imperative for localization to catch up and keep improving its agility. In consequence, software localizers should be able to translate most important messages first, e.g., those that are most visible to the user or that require a special attention, like error messages.

3 System Overview

With the aforementioned localization challenges in mind, we have developed Nimrod, a PHP internationalization tool that prioritizes resource strings through *progressive filtering*. The tool is based on PHP's built-in gettext library. To begin with, it selects those strings that are exclusively required to build the UI. Next,

it takes into account a number of features to assign a different importance to each string, depending on the context of the string in the UI according to e.g. visibility, frequency, semantics, or the interaction received by the user. Meanwhile, the remainder strings are considered to be less imperative and thus are left untouched, as they appear in the original message catalog. In sum, our tool selectively picks the most relevant candidates among all strings available in the message catalog for early translation, by moving the relevant strings to the top of the catalog file.

At a lower level, our tool exposes two functions to the developer: `_gt()` and `_gx()`, where the only difference between the two is that the latter allows the developer to specify a particular gettext context. This is so because in gettext all strings are indexed according to source string (`msgid`) and context (`msgctxt`), if available. Both Nimrod functions augment the PHP `gettext()` function and its shorthand equivalent `_()` with a special code, so that whenever the function is invoked, the localized message is complemented with UI-based information where available, such as size or contrast of the UI element where each string belongs to. This UI information is compiled by the web browser using injected JavaScript code. Since Nimrod preserves the usual localization information (e.g., developer comments or name of source files), it is able to reproduce the original message catalog together with feature-rich information.

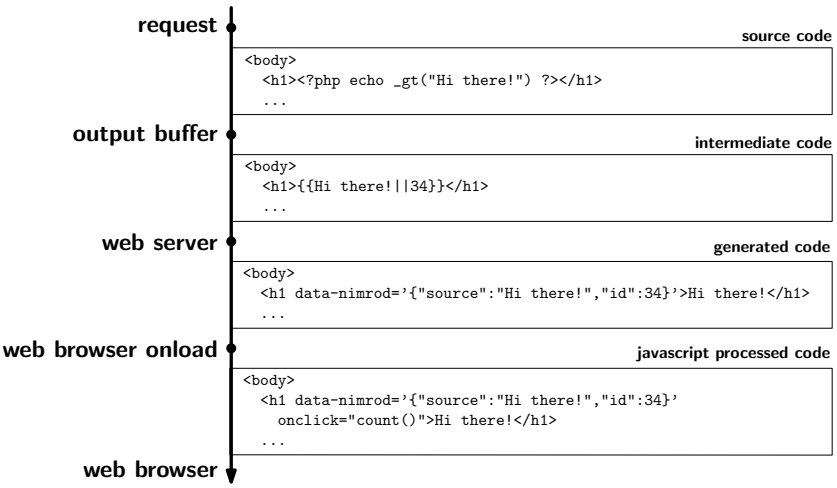


Fig. 1: Nimrod's processing pipeline.

This complementary information is logged in a dedicated database, in JSON files, by means of an asynchronous Ajax call invoked on page load. To achieve this, the HTML code is parsed before being sent to the web browser, by means of PHP's output buffering capabilities; see Figure 1. Without output buffering this would not be possible, as the page would be sent into pieces as PHP processes

each HTTP request and thus the document object model (DOM) would not be ready for manipulation. Then, the website administrator can access a control panel (Figure 2) and generate message catalogs on the fly, where strings can be sorted according to the following priority features (Section 3.1).

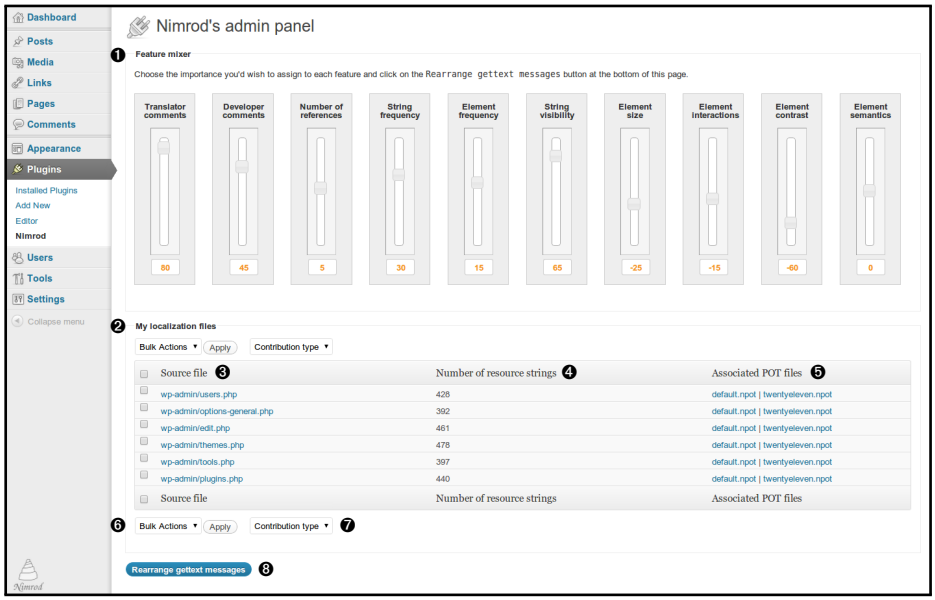


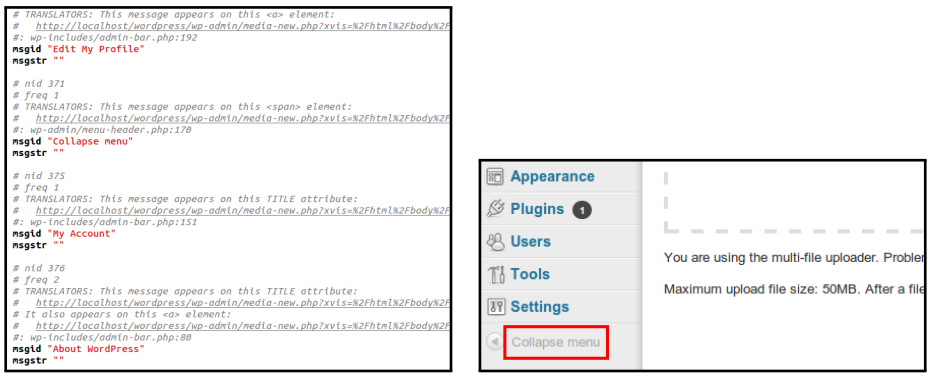
Fig. 2: Integration with Wordpress admin site. ❶ The importance degree of each feature is in $[-100, 100]$ %. Values are stored in a cookie. ❷ Localization files are shown in a dedicated table. ❸ Source files are available for each browsed URL. ❹ Number of resource strings, according to `gettext()` function calls. ❺ Each source file is associated at least with one gettext domain. ❻ Selected source files can be downloaded or inspected individually. ❼ Preferences can be considered coming from either the currently logged user or all admin users. ❽ This button creates one PO file per domain, taking into account all these preferences.

3.1 Prioritization Features

The following is a succinct description of the computed features.

Automatic comments [`int`] Number of autogenerated comments, aimed at giving element examples where each message appears (Figure 3).
Developer comments [`int`] Number of comments given by the programmer in the source code, directed at the translator.
Number of references [`int`] Number of source files where each message appears. Theoretically, the higher the more their importance.

String frequency [`int`] Number of times each message appears on the UI, as different elements may have the same string.
Element frequency [`int`] Number of UI elements where each message appears. Strings appearing in multiple elements may require a special attention.
String visibility [`bool`] Whether the string is shown on the UI or not. Page `TITLE` and most of the `BODY` elements are visible to the user.
Element size [`float`] The size (*height*) of an element may influence its importance. Typically, bigger elements are most visible and thus should be localized earlier.
Element contrast [`float`] Difference between background and foreground RGB colors. Elements with low contrast are less visually noticeable, so their priority may be less important.
Element semantics [`float`] A numerical weight for each type of DOM element, similar to what is done in information retrieval [3, p.2], to assign more importance to special elements like headers or links. This can also be used to reward (or penalize) good (or bad) HTML markups.



(a) (b)

Fig. 3: Messages are augmented with automatic descriptions of the UI elements (3a), together with URLs to highlight such UI elements in the original web page (3b).

In addition, the message catalog can be extended with behavioral information that suggests which strings are more relevant to the user; e.g., mouse or touch events. Although Nimrod allows these events to be captured, they are not considered for analysis at the moment, since they are too dependent on website usage and webmaster activity.

4 Evaluation

We aim to assess to what extent Nimrod can be effectively used to prioritize the localization of a real website by end users. To do so, first we study the combination of feature presets (Figure 2) that best represent a number of different string categories (Section 4.4). Then, we convey an experiment to perform a fine-grained categorization of strings based on their relevance (Section 4.5).

4.1 Experimental Setup

We developed `WP_Nimrod`, a plugin for Wordpress on top of Nimrod. We chose Wordpress for evaluation because it is one of the most popular web CMS, is open-source software (GPL) written in PHP, and is already internationalized with gettext. After a clean Wordpress installation on our web server, we just had to activate the plugin. Basically, the `WP_Nimrod` plugin traces the calls to Wordpress' `translate()` function and stores in a JSON file the strings of each requested web page, together with the features mentioned in the previous section.

4.2 Procedure

We manually browsed all pages at the Wordpress admin site, by clicking on all links of the rightmost navigation menu, including links in submenus; see Figure 4. In total, 37 admin pages were browsed. All translatable strings from these admin pages were automatically registered, removing duplicates, resulting in 1679 messages. These account for 47.5% of the messages in the latest version of the official Wordpress PO file (v3.8.1, 3533 messages), which reveal that an important number of strings are seldom used on the admin site. In the following, we briefly describe the dataset we gathered from the remaining strings, named "Wordpress Localization dataset", or WPLoc for short. The dataset will be made publicly available after publication.

4.3 Dataset

The WPLoc dataset comprises 1679 manually annotated messages from Wordpress admin site; see Table 1. These messages are distributed in a long tail: 1100 strings were logged only once, 390 strings appeared in more than 10 cases, and 49 strings appeared in more than 100 cases. Anecdotally, the most frequent message is `"Add New"`, appearing 540 times. It is worth pointing out that 93% of the strings appear in one single element, and that 70% of the messages have no visibility on the UI. This is the case of strings in drop-down menus, hidden lists, or help paragraphs that are revealed after an explicit user click/touch. Indeed, the most frequent HTML tags among all UI elements are `DIV`, `LABEL`, `OPTION`, and table cells. Of utmost importance is the fact that just 4 strings have human-generated comments, which reveals that Wordpress developers do not leave enough localization comments for translators.

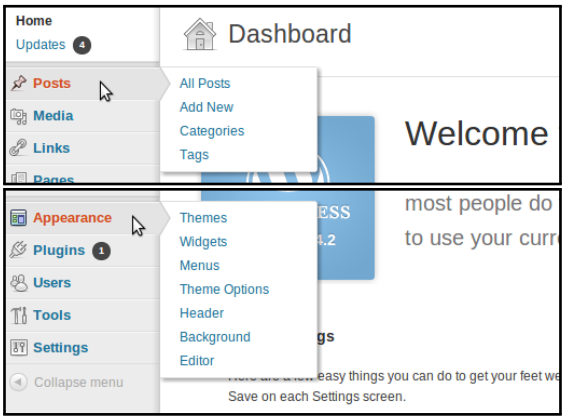


Fig. 4: Wordpress admin menu.

On the UI, different types of strings may appear, such as error messages, call-to-actions, labels, etc. Thus, each message in the WPLoc dataset was manually labeled according to the following categories:

1. **Relevant:** Messages that must be understood to make use of the UI.
2. **Non-relevant:** Less important strings, such as month names or cities.

Both categories are well balanced, with 841 relevant strings and 838 strings otherwise. In addition, a finer-grained categorization was achieved by dividing the relevant strings in 3 subclasses:

1.1. **Critical:** Most imperative messages, such as those related to website maintenance or error messages.
1.2. **Informative:** Those messages that facilitate the understanding of the basic functionality of the website, such as paragraphs.
1.3. **Navigational:** Messages that inform about browsing actions, such as menu items.

Table 1: Message categories used for evaluation.

Group Label	Importance	No. Strings
Critical	relevant	92
Informative	relevant	437
Navigational	relevant	312
Others	non-relevant	838
Total		1679

4.4 Feature Analysis for User-driven Prioritization

We investigated how the previous string categories are characterized by the prioritization features described in Section 3.1. This would give an intuition to webmasters on how they could take control over the panel shown in Figure 2 and sort the strings according to their wishes. A straightforward approach to finding such characterization consists in standardizing all sample features (to have zero-mean and unit-variance) and computing the group centroids. Without feature standardization, the features would be in different ranges and the importance of each feature would be biased toward features with high values. Table 2 summarizes these values for each message category, normalized to -100 and 100 as in Nimrod's control panel (Figure 2). This study provides a better idea on whether a given feature affects prioritization, either positively or negatively.

Table 2: Feature combinations, or "mixer preset" weights (see Figure 2), that prioritize different UI string categories. All weights are bounded to $[-100, 100]$ %, the sign indicating positive or negative influence.

Group	A.C	D.C	N.R	S.F	E.F	S.V	E.Si	E.C	E.Se
Informative	14.7	-14.3	4.4	-24.4	0.6	41.0	40.8	34.6	23.6
Navigational	-6.3	-14.3	8.8	43.2	15.5	12.2	18.6	57.2	22.7
Critical	-14.9	-14.3	-8.1	-13.8	-31.9	4.7	11.2	-8.2	-32.2
Other	-3.7	14.4	-4.7	-1.8	-2.6	-26.5	-29.4	-38.5	-17.2

Column names follow the same order given in Prioritization Features (Section 3.1); e.g., **A.C**: automatic comments, ..., **E.Se**: Element semantics.

Looking at Table 2, it can be observed that navigational messages show a high contrast, whereas critical messages are represented by a lower contrast. Although this may sound contradictory, it must be noted that in Wordpress most navigational elements have a white font over dark background, while critical messages often use a red font color over light red background, which would diminish its contrast. Thus, perhaps a more appropriate metric such as color saliency based on visual human perception would be a better option to measure element contrast.

On the other hand, it was found that string visibility and element size are more prominent in informative messages rather than in critical messages. This may be due to the fact that informative texts are often very descriptive and thus might have a higher size on the UI. However, critical messages are typically exceptions that are rarely shown, and thus they remain hidden most of the time. Something similar happens to string and element frequency: critical messages appear scarcely and often at the same places. Conversely, navigational messages appear in almost any page and often in different menu entries. In this regard, informative messages appear less frequently than its critical counterparts, probably because the former are more specific and descriptive. Additionally, developer

comments appear only in the "other" category, hence their weights are uniform across categories. Finally, the number of references was found to have little importance to discriminate among the analyzed categories.

4.5 String Retrieval for Fine-grained Prioritization

Given that we had defined a well-established ground truth, another experiment to analyze string prioritization possibilities consisted in performing a more detailed, automatic string retrieval task. As in the previous experiment, strings belong to the classes (and subclasses) of the WPLoc dataset, and the same set of features is used.

As a practical application, we decided to use several state-of-the-art classifiers. All classifiers were evaluated using the open-source Weka machine learning suite [11], which will allow others to easily reproduce our experiments. The following results present a subset of the algorithms that, to our knowledge, are representative of a broad range of techniques: Logistic Regression [4], Multilayer Perceptron [17], KStar [5], Random Forest [2], and Bagging [1] with REPTree.

The experimentation was formulated as a retrieval problem, where we search for the strings that match a given class. Thus, *precision* indicates, among all retrieved strings, the percentage of strings that belong to the given class, whereas *recall* indicates the percentage of strings of a given class that were effectively retrieved. Finally, *F-measure* is the harmonic mean of precision and recall, and it can be interpreted as a weighted accuracy. All experiments were run on the whole set of 1679 strings with a 10-fold cross-validation setup.

Table 3: Retrieval results of *relevant* messages, in percentage.

Classifier	Precision	Recall	F-measure
Logistic Regression	73.8	83.9	78.6
Multilayer Perceptron	75.0	**94.4**	83.6
KStar	**80.6**	88.0	84.1
Random Forest	79.7	87.4	83.4
Bagging (REPTree)	78.5	90.7	**84.2**

First of all, Table 3 shows results when retrieving those strings that were classified as relevant. On the one hand, *KStar* achieves the best precision at the expense of a worse recall. On the other hand, *Multilayer Perceptron* obtains the best recall at the expense of a lower precision. In terms of F-measure, *Bagging* and *KStar* performed better than their peers. Actually, what is best depends on the particular user needs. For instance, high precision with low recall involves less strings to be translated. Concretely, *KStar* retrieves 918 strings, *Multilayer Perceptron* 1058 and *Bagging* 971. Thus, if the goal is to reduce localization costs, then *KStar* is a reasonable option. Nevertheless, *Multilayer Perceptron* retrieves

most of the relevant strings, which is probably a more important technique for a more imperative scenario, i.e., to reduce localization time.

Table 4: Results for message retrieval of the different relevant subclasses, in percentages. Classifiers marked with (*) were expanded with a weighted bag of words of each resource string as additional features.

	Critical			Informative			Navigational		
Classifier	**Prec**	**Rec**	**F_1**	**Prec**	**Rec**	**F_1**	**Prec**	**Rec**	**F_1**
Logistic Regression	0.0	0.0	0.0	53.0	48.5	50.7	42.7	15.1	22.3
Multilayer Perceptron	69.2	9.8	17.1	51.3	70.9	59.6	49.4	39.7	44.0
KStar	66.7	17.4	27.6	51.3	**75.5**	61.1	*67.8*	37.8	48.6
Random Forest	47.4	*19.6*	27.7	*56.2*	67.7	61.4	63.1	*48.7*	*55.0*
Bagging (REPTree)	*76.2*	17.4	*28.3*	55.1	73.5	*62.9*	60.8	45.2	51.8
Random Forest*	58.2	**34.8**	**43.5**	62.2	65.4	63.8	64.2	**63.8**	**64.0**
Bagging (REPTree)*	**83.3**	27.7	41.0	**63.3**	73.0	**67.8**	**68.2**	57.1	62.1

Table 4 disgregates the results for the different subclasses of relevant messages. Overall, critical messages present very low recall scores in all of the tested classifiers. We suspect that this is caused by the low number of strings that fall under this category (5.5% in the WPLoc dataset). Thus, the class prior probability is low and few samples are used to train each classifier. On the other hand, we observed that informative messages are easier to retrieve with *KStar*, with a 75.5% of recall. However, its precision suggests that half of the retrieved strings would not be informative. Finally, navigational messages achieve higher precision scores, probably also because of the class prior imbalance.

Intuitively, critical messages should be easier to retrieve, since typically they would contain words related to errors and warnings, while navigational messages should be short and semantically related to UI actions. Thus, we decided to expand the feature set with a weighted bag of words of each resource string, where weights are string frequencies. The results are presented in Table 4 and marked with an asterisk (*). Significant improvements can be observed in all cases, which remarks the importance of the text for relevance identification. Concretely, a qualitative analysis of Bagging's decision trees showed that the word 'error' is used to discriminate critical messages, while 'updated' is present in informative messages and 'link' in navigational messages.

It is worth mentioning that the bag-of-words analysis may not extrapolate to *any* website. Indeed, different websites and CMS do have different UI strings, and thus a classifier should be trained for each website or CMS. Nevertheless, it is possible to use any of the other classifiers tested so far, as they provide competitive accuracy. In any case, this study puts forward the fact that different classifiers are feasible to improve website localization.

4.6 General Discussion

The needs of the localization industry change often over time. Today's development cycles are typically shorter, with quick turnaround times. This trend encourages software localization for a prompt revision.

Often it is necessary to localize just the essential parts of a UI, either because of competitive advantage or economical reasons. For instance, an advanced word processor may comprise an important number of menus and options, but actually only a few of these are used by regular users. Then, localizing just what is most important would allow to reach emerging markets or even introduce a new product sooner than the competence. Unfortunately, following the typical localization workflow, it is difficult to decide which elements should be localized earlier. Moreover, typically websites and web applications do change over time, and so it should be possible to perform localization in an incremental fashion. Therefore, a solution to prioritize UI elements for localization is necessary. Nimrod is our contribution to tackle this topic.

5 Conclusion and Future Work

Software localization is both costly and a slow process, partially affected by the lack of string prioritization. We have shown that it is possible to automate the prioritization of UI strings, so that web-based software can be quickly and frequently translated. Our method augments string information in a message catalog with UI features, such as widget size, visibility, or color, so as to selectively pick the most relevant strings in the first place.

For future work we will incorporate our evaluation findings in our string prioritization tool. Concretely, we will add preset configurations that would allow to retrieve critical, informative, and navigational messages earlier. We also plan to consider UI usage information for analysis, like mouse or touch events, in order to gain more knowledge on the best prioritization schema. We hope this work will be useful to researchers and companies interested in UI localization. Nimrod is open source and can be downloaded at `http://personales.upv.es/luileito/nimrod/`.

Notes

[1] `http://www.drupaltranslate.com`
[2] `http://smartling.com`
[3] `http://lingoport.com/globalyzer`
[4] `http://kokusaika.jp/en/product/wwnavi.html`
[5] `http://launchpad.net`
[6] `http://pootle.translatehouse.org`
[7] `http://www.verbatimsolutions.com`
[8] `http://www.transifex.com`

Acknowledgments

This work is supported by the 7th Framework Program of EU Commision under grants 287576 (CASMACAT) and 600707 (tranScriptorium). The motivation of

choosing "Nimrod" to name our tool is left as an additional exercise for the reader.

References

1. Breiman, L.: Bagging predictors. Machine Learning 24(2) (1996)
2. Breiman, L.: Random forests. Machine Learning 45(1) (2001)
3. Cascia, M.L., Sethi, S., Sclaroff, S.: Combining textual and visual cues for content-based image retrieval on the world wide web. In: IEEE Workshop on Content-Based Access of Image and Video Libraries (CBAIVL) (1998)
4. le Cessie, S., van Houwelingen, J.: Ridge estimators in logistic regression. Applied Statistics 41(1) (1992)
5. Cleary, J.G., Trigg, L.E.: K*: An instance-based learner using an entropic distance measure. In: 12th International Conference on Machine Learning (1995)
6. Collins, R.W.: Software localization for internet software: Issues and methods. IEEE Software 19(2) (2002)
7. DePalma, D.A., Hegde, V., Pielmeier, H., Stewart, R.G.: The language services market. An annual review of the translation, localization, and interpreting services industry. Available at `http://commonsenseadvisory.com` (2013)
8. Dunne, K.J. (ed.): Perspectives on Localization. John Benjamins Publishing Company (2006)
9. Esselink, B.: A Practical Guide to Localization. John Benjamins Publishing Company (2000)
10. Gettext: The GNU gettext manual. Available at `http://www.gnu.org/` (1995), version 0.18.2.
11. Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H.: The WEKA data mining software: An update. SIGKDD Explorations 11(1) (2009)
12. Hogan, J.M., Ho-Stuart, C., Pham, B.: Key challenges in software internationalisation. In: Workshop on Australasian Information Security, Data Mining and Web Intelligence, and Software Internationalisation (ACSW Frontiers) (2004)
13. Keniston, K.: Software localization: Notes on technology and culture. Working Paper #26, Massachusetts Institute of Technology (1997)
14. Leiva, L.A., Alabau, V.: An automatically generated interlanguage tailored to speakers of minority but culturally influenced languages. In: Proceedings of the SIGCHI conference on Human factors in computing systems (CHI) (2012)
15. Leiva, L.A., Alabau, V.: The impact of visual contextualization on UI localization. In: Proceedings of the SIGCHI conference on Human factors in computing systems (CHI) (2014)
16. Reinecke, K., Bernstein, A.: Improving performance, perceived usability, and aesthetics with culturally adaptive user interfaces. ACM Transactions on Computer-Human Interaction (TOCHI) 18(2), 8:1–8:29 (2011)
17. Rosenblatt, F.: The perceptron: a probabilistic model for information storage and organization in the brain. Psychological review 65(6) (1958)
18. Sun, H.: Building a culturally-competent corporate web site: an exploratory study of cultural markers in multilingual web design. In: Proceedings of the 19th Annual International Conference on Computer Documentation (SIGDOC) (2001)
19. Troyer, O.D., Casteleyn, S.: Designing localized web sites. In: Proceedings of Web Information Systems Engineering (WISE) (2004)

20. VanReusel, J.F.: Five golden rules to achieve agile localization. Available at http://blogs.adobe.com/globalization/ (2013)
21. Wang, X., Zhang, L., Xie, T., Mei, H., Sun, J.: TranStrL: An automatic need-to-translate string locator for software internationlization. In: Proceedings of IEEE 31st International Conference onSoftware Engineering (ICSE) (2009)