This work was accomplished as part of the internship at the Information and Language Processing Research Lab, Kathmandu University, Dhulikhel, Kavre and Language Technology Kendra, Lalitpur, PatanDhoka, Nepal.

RESEARCH REPORT ON THE NEPALI SPELL CHECKER

A detailed report on the structure and functioning of the Nepali spell checker, including functioning of the Hunspell Spellcheck engine and extended support for Libre.

AkshayLabhKayastha
B. Tech. 2nd Year, NIT Warangal, Warangal, A.P., India

DilipYadav
B. Tech. 2nd Year, NIT Warangal, Warangal, A.P., India

Krishna Sarda
B. Tech. 2nd Year, NIT Warangal, Warangal, A.P., India

1, akshay.ksth@gmail.com

2,dilipptt@gmail.com
3,krishnasarda@gmail.com

Table of Contents

Acknowledgements

We would like to specially thank MrBal Krishna Bal of the Department of Computer Science and Engineering, Kathmandu University for providing his valuable mentorship for the project. In addition to that we thankLanguage Technology Kendra for providing the resources required for the project.

Also, we would like to thank Dr. BalaramPrasain of the Central Department of Linguistics, Tribhuvan University, for providing his invaluable help in the completion of this project. Furthermore we cannot forget Mr. Rhoit Man Amatya, researcher at the Information and Language Processing Research Lab,Department of Computer Science and Engineering, Kathmandu University, for guiding us through the tough times and for his contributions on the project without which we would have been nowhere.

Abstract

This report discusses the design and development issues of the Nepali Spell Checker application. The methods used for development of the application, as well as the implementation of the spellcheck feature have been discussed in the report. In addition to that, differences from the previous version of the application have also been highlighted.

Introduction

Since the advent of the Unicode encoding system[1]and the development of the Nepali Unicode keyboard layout, the task of typing in Nepali has been greatlysimplified. Being miles ahead of the ASCII based traditional keyboard layouts, in terms of ease of learning and accessibility, the Unicode layout has attracted a large number of the people into using it. Lives have been simplified and revolutionized. Along with this change, it is extremely important to incorporate efficient spellchecking modules to whatever software supports Unicode. While it is not possible to do so in Microsoft Word, which is by far the most used word processor today, we can certainly approach open source word processors such as Libre. Yet this is still inconvenient as Libre is not a cross platform word processor. Hence, a standalone cross platform spell checker for Nepali language is a necessity for today's users.

MadanPuraskarPustakalaya (MPP) has already released at least five different versions of the Nepali Spell Checker incorporated with the localized version of OpenOffice.org Writer application[2,3].The latest version of the Nepali Spell Checker came out with the Nepali localized OpenOffice.org Writer 2.4, which was released in May 2009. This version of the Spell Checker had the word coverage of 6 million Nepali words. The first version of Standalone Nepali Spell Checker that was also released on May 2009, freed users from the need to install OpenOffice .org Writer application as a prerequisite for using the Spell Checking utility for Nepali.

Just like the previous version, the current version of the Standalone Nepali Spell Checker also runs on the HunSpell[4] engine and the two resource files for Spell Checking – the .dict and .affix files, customized for Nepali language. The use of HunSpell engine in the standalone application gives robust performance, in the already feature rich and light weight text editor. Regrettably, this version of the Standalone Nepali Spell Checker works only on a UNIX platform.

---

[1] *http://unicode.org*
[2] B. K. Bal and P. Shrestha, "Nepali Spellchecker," PAN Localization Working Papers 2004-2007.
[3] B. K. Bal, B. Karki, and L. Khatiwada, "Nepali Spellchecker 1.1 and the Thesaurus, Research and Development," PAN Localization Working Papers 2004-2007.
[4] *http://en.wikipedia.org/wiki/Hunspell*

Previous Version – Shortcomings

The first Standalone Nepali Spell Checker was based on the .NET wrapper for the HunSpell Engine. The Graphical User Interface (GUI) for the application was created in Microsoft C#.NET 2008. The spell checking mechanism was flawless and produced efficient replacement suggestions. The context menu was easy to use. But there were still a few drawbacks in the application. They can be listed as follows:

FLAWS IN APPLICATION:

1. The application was unable to detect faulty spellings simultaneously as the user typed. A button was used to check for mistakes. This was felt as an inconvenience for the daily user.
2. The application had a bug, which limited its use to 32-bit architectures.

FLAWS IN RESOURCE FILES:

The entire spell checking engine is dependent upon the two resource files (.dic and .aff). Several words were seen to be absent in the dictionary (.dic) file. In addition to that the affix (.aff) file was reported to have some missing rules. These two issues were the primary drawbacks for effective spell checking.

Hence, a list of common issues was created. The **Issues** section involves all issues that have been taken care in this version of the Spell Checker.

The HunSpell Engine

BACKGROUND

Hunspell is a spell checker and morphological analyzer designed for languages with rich morphology and complex word compounding and character encoding, originally designed for the Hungarian language. Hunspell is based on MySpell[5] and is backward-compatible with MySpell dictionaries. While MySpell uses a single-byte character encoding, Hunspell can use Unicode UTF-8-encoded dictionaries.

This Spell Checker engine was initially developed for the Hungarian Spell Checker but now has the capabilities of processing theoretically all languages with Unicode support. The HunSpell framework comprises the HunSpell engine and two resource files – the .dict and the .affix files. Hunspell is open source and several programming languages have built modules that act as wrappers for its functionality. The python module wrapper is called pyhunspell. NHunspell is a C# interface library that uses Hunspell functions.

The Hunspell engine basically performs a lookup in the .dict file for a word and at the same time also works on forming derivational words out of the head words in the .dict file and the corresponding rules in the .affix file. If a matching word does not result out of the above process, it infers that the given word is typed incorrectly and hence it generates a list of possible suggestions of the word by using *Levenshtein*[6] Edit Distance Technique.

[5] *https://en.wikipedia.org/wiki/MySpell*
[6] *http://en.wikipedia.org/wiki/Levenshtein_distance*

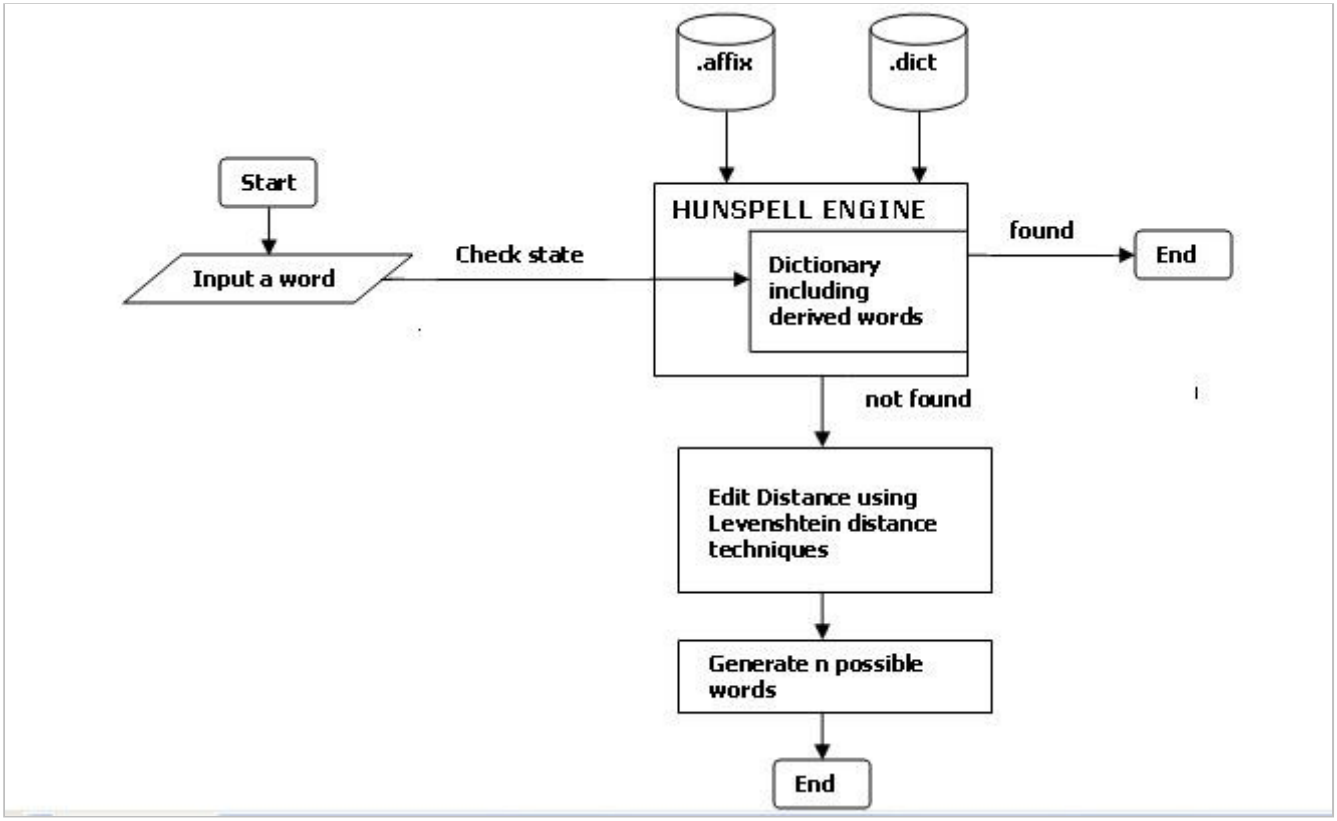

Figure 1 System Architecture of the Standalone Spell Checker

The Hunspell's code base comes from the OOoMySpell. It improves upon MySpell's functionalities in the following ways:

1. It has Unicode support (first 65535 Unicode character)
2. Morphological analysis can be done (in custom item and arrangement style)
3. Max. 65,535 affix classes and twofold affix stripping (for agglutinative languages, like Azeri, Basque,Estonian, Finnish, Hungarian, Nepali, Turkish, etc.)
4. It supports complex compounding (for example, Hungarian and German)
5. It supports language specific algorithms (for example, handling Azeri and Turkish dotted i, orGerman sharp s)
6. It can handle conditional affixes, circumfixes, fogemorphemes, forbidden words, pseudoroots andhomonyms.
7. It has been released under GPL/LGPL/MPL tri-license

STRUCTURE AND ORGANIZATION OF RESOURCE FILES

The dictionary files and affix files are extremely important in order to allow the proper functioning of the hunspell engine. The first file is a dictionary containing words for the language, and the second is an "affix" file that defines the meaning of special flags in the dictionary. The spell checking is done using the .aff file for the

language together with the .dic file. The .dic file is a list of words along with a group of characters which refer to the affixes found in the .aff file. This saves space because instead of including all forms of a word(for example, खा = खानु, खाने, खादै,etc.), the .dic file will include the word once and the references to the affixes in the .aff file allow the construction of all the other forms.

Both files thus have a certain format to be followed for the HunSpell to recognize them and effectively use them. A brief structure of both the files has been given below:

The dictionary file

A dictionary file (*.dic) contains a list of words, one per line. The first line of the dictionary (except personal dictionaries) contains the approximate word count (for optimal hash memory size). Each word may optionally be followed by a slash ("/") and one or more flags, which represent affixes or special attributes. Although default flag format is a single (usually alphabetic) character, the format for *'ne_NP.dic'* is numeric. A sample segment of *'ne_NP.dic'*is shown below:

In the above dictionary file, the first line has the number "3", which gives the optimal hash memory size andthe number of words, which are "अकमकाउनु", "अकर", and "अकरण". The first word does not have any flag. Theword "अकर" has one flag which is separated by a "/" back slash. The flag "15" points to a rule named "15" in theaffix file. Similarly in the next line, the word "अकरण" and its field are separated by a "/". The flag "18, 22, 15, 34"points to the rule name "18", "22", "15" and "34" in the affix file.

The affix file

```
3
अकमकाउनु
अकर /15
अकरण/18,22,15, 34
```

The affix file consists of the rules that will add affixes to the words which are present in the .dic file. Asmentioned earlier, the flag, which is usually a character, in the dictionary file points to these rules. To clarifythe concept of the affix file consider the following sample from our affix file,*'ne_Np.aff'*:

```
SET UTF-8

TRY ािीुूृेैोौंःँअाइईउऊऋएऐओऔकखगघङचछजझञटठडढणतथदधनपफबभमयरलवशषसहक्षत्रज्ञ

FLAG num

REP 24
REP ि ी
REP ी ि
SFX 1 Y 3
SFX 1 ् 0/17X ्
SFX 1 ुनु/17X,20X्
SFX 1 0 ेको/17X,19X [^ुइि ा]
```

An affix is either a prefix or a suffix attached to root words to make other words. For example खा ->खानु by dropping the "्" and adding a "नु" (the suffix), or राम्रो->राम्रो by simply adding "न" (the prefix).

An affix file (*.aff) may contain a lot of optional attributes. For example, the first line consisting of SETspecifies the character set used for both the dictionary file and the affix file (should be all uppercase). Theabove affix file example defines UTF-8 character encoding. TRY, in the second line, sets the changecharacters for suggestions. It is used in building suggestions for misspelled words; for example, themisspelled word "नामा" will have a suggestion "नामि" by substituting the "ा" with an "ि". The characters in thestring should be listed in order or according to the character frequency (highest to lowest). The suggestionsproduced using the 'TRY' option differs from the bad word with a single English letter or an apostrophe. Agood way to develop this string is to sort a simple character count of a word list. REP sets a replacementtable for multiple character corrections in suggestion mode. The first line that consists of REP informs thatthere are two entries for the REP option. With these REP definitions, Hunspell can suggest the right wordform, when the misspelled word contains 'ि' instead of 'ी' and vice versa, for example, if we

```
SFX 1 Y 3
SFX 1 ् 0/17X ्
SFX 1  ुनु/17X,20X ्
SFX 1 0 ेको/17X,19X [^ुइि ा]
```

write 'पनी' it cansuggest the right word 'पनि'. PFX and SFX defines prefix and suffix classes named with affix flags.The affix file is space delimited and case sensitive. So we can interpret the affix file's rule lines, as follows:

In the first line there are four fields, whose description are given in the table below:

Field	Name	Description
1	SFX	indicates that this is a suffix(PFX indicates a prefix)
2	1	this is a name for the suffix it represents which should be unique for every different suffix or prefix entry(or the name for the prefix when PFX is present)
3	Y	indicates it can be combined with prefixes(cross product)
4	4	indicates that sequence of 4 affix entries are needed to properly store the affix information

The remaining lines describe the unique information for the 4 affix entries that make up this affix. All thefields in the remaining line are the same; fields in the second line are described below:

Field	Name	Description
1	SFX	indicates that this is a suffix(PFX indicates a prefix)
2	1	this is a name for the suffix it represents which should be unique for every different suffix or prefix entry(or the name for the prefix when PFX is present)
3	◌्	the string of chars to strip off at the end before adding affix (a 0 here indicates the NULL string, and in case of PFX, the chars are strippedoff at the beginning of the word)
4	◌ेको	the string of affix characters to be added at the end of the word(a 0here indicates the NULL string,and in case of PFX, the chars areadded at the beginning of the word)
5	/17X,19X	Other rules that can be cascaded with the current affix rule are placed in this order. In this instance rule set 17 and 19 are allowed to be cascaded onto the current rule.
6	[^◌ुइि◌ा]◌्	the conditions which must be met before the affix can be applied,which represents a regular expression("." a dot means there is nocondition)

Field 6 might be confusing. Since this is a suffix, field 6 tells us that there are 2 conditions that must be met.The first condition is that the character next to the last

character in the word must *NOT* be any of thefollowing “ु”,”इ”,”ि” or”ा”. The second condition is that the last character of the word must end in "्".

The above format of the Dictionary and Affix files, although brief easily suffices basic upgrades for the Nepali Spell Checker. For additional *OPTION*s and *FLAG*s, please read through*“PAN Localization Guide to Localization of Open Source Software”.*

Issues

This section describes in detail all the noticeable issues perceived in the first version. The issues were discussed among by experts through several meetings[7]. Below is a list of all significant issues:

1. Several words have variations in their typing techniques. Let us take an example of the word ‘पक्का’. The second character is a half form of the consonant ‘क’. There are two ways to create the half form and connect it with the other consonant. The first one involves adding a ‘्’ after the full form of ‘क’, and then typing in another ‘क’. Doing this gives us ‘क्क’ (‘क’+‘्’+‘क’). Another way to do that is by adding a Zero Width Joiner (ZWJ) after the ‘्’. Hence ‘क’+‘्’+ZWJ+‘क’ is also ‘क्क’. The ZWJ in fact forces the consonant into its half form.
   The problem with this though is that the Spell Checker would only detect the former version as correct and the latter as incorrect.
2. Words in Nepali often consist of a number of added ‘*vibhaktis*’ such as ले, लाई, को, का, की, रो, रा, री, नो, ना, नीetc. The Spell Checker failed to recognize forms of words that had one or more than one such ‘*vibhaktis*’ added at the end of the word.
3. The spell checker also missed to recognize the variations in the writing style of several words. For example, अङ्क, अंक, मंशिर, मङ्सिर.
4. The ‘Add’ function in the Spell Checker worked only for the current session and would not work after the application was opened the next time. It hence worked more like an ‘Ignore’ function.
5. Errors in the affix file caused errors in several word structures that involved the Zero Width Joiner (ZWJ). Because of that words such as ‘पर्यो’ and ‘पुर्यउनु’ were shown to be correct instead of the words ‘पर्‌यो’ and ‘पुर्‌याउनु’. The ZWJ was

[7]*Meetings were held among members of the Language Technology Kendra (LTK) and SIL.*

found to be missing in the former word structure in between the character 'र्' and 'य'.

6. The Zero Width Non Joiner (ZWNJ) works just the opposite of ZWJ i.e. it blocks the formation of half forms in the consonant. Although the use of ZWNJ is very rare in Nepali, words such as 'श्रीमान्को' require it. Without the ZWNJ the word would look like 'श्रीमान्को' which is incorrect. The Spell Check although indicates the latter as correct instead of the former one.
7. The current dictionary files only contain a total of around 36000 words. It misses out several words. It was essential to add many more words to the dictionary to increase accuracy.
8. As mentioned before, the members of the meeting found it very important for the Spell Checker to work right after the completion of the word after hitting 'Space Bar'.

Development

The development of the application was finished in three phases.

PHASE I: UNDERSTANDING AND UPGRADING THE RESOURCE FILES

The primary objective of the new version of the Spell Checker was to be more efficient. This required properly understanding the working of both *.dic* and *.aff* files.

After reading through all documents related to the structure and formation of *.dic* and *.aff* files, a need arose to clearly map words in the *.dic* file to the rule sets in *.aff* file. Mapping the files would give us a strong insight on the target word type and functions of each word set. It was also noticed that the rules in the *.aff* file were numerically named, making it much easier for the mapping. A python script was written therefore to create separate files for each rule sets, with only the related words falling in each. A total of 50 rule sets are present in the *.aff* file. 50 separate text files for each rule set were created using the script, thus allowing us to figure out the target word group and the suffixes (or prefixes) that each rule added to the

root word. After several meetings with Dr. Prasain, it was seen that among the 50 rule set, only about 10 rules were recurrently used. Rule sets 15, 22, 18, 8, 43, 9 and 42 belong to this category each having at least more than 1000 words and with rule 15 being the most regularly used and rule 42 the least regularly used. Rule no 17 was the only prefix rule whereas Rule no 49 was the one that applied to numbers. The remaining rules had very small functionality and were seldom used.

One of the objectives required the upgradingof the current *.dic* file with more words. A total list of around 38000 words extracted from the *Brihat Sabdakosh*and provided for addition into the existing dictionary. A Python script was used to remove duplicates from this file, which already existed in the current dictionary. Hence a 'difference' dictionary file was created. With mapping data and considerable help from Dr. Prasain, the file was changed to match standard *.dic* file format by upgrading rules for each root word obtained. Merging the obtained file with the existing dictionary, a final dictionary file was obtained which contained approximately 52000 root words.

Fixing issues in the current affix file was the next target. This was done by carefully evaluating each one of the word types mentioned above in the **Issues** section. Each faulty words type and structure was evaluated and the previously obtained mapping was used to figure out the faulty rules in the *.aff* file.

The following methods were tried to resolve each issue in the **Issues** section:
(Note: The order below mirrors the issue list in the mentioned section)

1. Since this was an issue related to the root word itself. A new duplicate word was added into the *.dic* file for each such word with a 'क्क' present in its structure. The duplicate word was typed in the alternative typing format (including the ZWJ). Although this increased redundancy in the *.dic* file, it was felt to be very insignificant in terms of efficiency.

2. From the mapping obtained, it was found that the rules most used for *'vibhaktis'* was Rule no 15, 18, 19 and 22. The mentioned rules were fixed by compounding them to each other in a loop, in order to allow multiple *'vibhaktis'*in the root word. The drawback in this fix was that there was no way to ensure that a terminating *'vibhakti'* ('ले' and 'लाई') was added in the end.

3. Again like Issue no. 1, this concerned with the root words itself. Hence, multiple similar versions of words were added to the *.dic* file to achieve the fix.

4. It was not possible to fix this problem through the resource files. The fix has been mentioned later in **Phase II: Development of GUI**.

5. The fix for this issue was obtained by modifying all rules that concerned with adding suffixes starting with the letter 'य' such as 'यो' and 'याउनु'. A Zero Width Joiner (ZWJ) was added to the beginning of every such rule. Also a condition was set that made the rule valid only for root words ending with 'र्'.

6. This issue was rectified by adding a Zero Width Non Joiner (ZWNJ) to the end of the word 'श्रीमान्' in the *.dic* file.

7. Fix for this issue has been mentioned already in earlier parts of this section.

8. The fix has been mentioned later in **Phase II: Development of GUI**.

PHASE II: DEVELOPMENT OF GUI

Development of a new Graphic User Interface (GUI) was essential for this application as the previous one had various shortcomings such as inability to work on a 64-bit architecture and concurrent spellchecking. The first step of this phase was to study the existing source files of the previous application which was written in C# using Visual Studio 2008. The second step required selecting the right base language for developing the interface. Hence, Python was chosen as the base language and ***PyQt4*** was used to develop the GUI. All development work was done on a Linux system.

The target at first was to create a simple form of a rich text editor that supported Unicode and especially Nepali. ***PyQt4*** provided a nice support for doing all that, and hence creation of the central Text Box, Application Menus and additional Buttons was easily completed within a week. The process of creating the skeleton of the application can be summarized in three simple steps:

1. Using the ***QtDesigner*** application to form the overall structure, by easily adding menus, icons, toolbars, and widgets. ***QtDesigner*** is a WYSIWYG (What You See Is What You Get) application that allows creation of a *.ui* file.

2. Once the *.ui* is successfully created, ***pyuic4*** can be used to convert the *.ui* file into *.py* file. Doing so creates an importable class that stores everything required to create the Interface of the application. The structure of this *.py* file can be seen in the **Source files**.

3. With the converted *.py* file ready, all that is required is to import this python class into the main code file. The main file should also import the ***PyQt4*** module.

The***codecs*** module provided easy support for accessing Unicode encoded files and using them.

The next step was to figure out a way to access the HunSpell engine and use it in the main code. The previous version had used a .dll file in order to import functions from the HunSpell engine. Doing the same using Python was very difficult as the exact structure of the .dll file was unclear. Instead we used a python wrapper module called ***hunspell*** (or pyhunspell) which imported the HunSpell engines abilities. The use of this ***hunspell*** module was simple enough and efficiently used the resource files in the main code.

The next period of this phase involved correctly displaying incorrect words, while typing into the textbox. For this purpose a sub-class of the ***Qt*** module called

QtSyntaxHighlighter was overridden. ***QtSyntaxHighlighter*** class along with the ***hunspell*** module allowed checking for incorrectly spelled words and underlining them. For this purpose an external module called ***regex*** was also used as it provided an alternate for the ***re*** module[8] but with better support for Unicode.

A Custom Context Menu was also created using inbuilt features of the ***Qt*** module. Spelling suggestions for the selected word (if incorrect), and options to 'Add to Dictionary' or 'Ignore' the currently selected incorrect word, were added into the Custom Context Menu. The 'Add to Dictionary' feature was used without using the HunSpell engine and directly accessing the *.dic* file, adding the selected word into itand updating the file header. All other features used functions of the ***hunspell*** module.

This completed a major task of the application and delivered the target. All code used for the abovementioned purposes can be seen in the **Source files**.

After the completion of the primary objective of the GUI, a couple of other features were added to provide better functions for the Text Editor, such as ***Find and Replace*** feature and a ***GoTo Line*** feature. These were done by creating additional widgets in the GUI skeleton and then adding code for the respective feature.

One special feature added to the application is a ***BugReport*** feature, which allows users to report errors to a specific mail address[9], which can then be monitored and amended. The end product was a basic text editor with Nepali spell checking.

PHASE III: INTEGRATION INTO *LIBRE*

The second major objective of the project was to implement the same spell checking feature into Libre Office. Earlier *MadanPuraskarPustakalaya* (MPP) successfully integrated Nepali spell checking into earlier versions of OpenOffice.org. Since OpenOffice.org used the HunSpell engine, all that was required was adding the *.dic* and *.aff* files to the required directory.

OpenOffice.org though is now no longer used in popular Linux distributions. Most have switched to the more popular Libre Office. Luckily however after minor research it was found out that Libre also used the HunSpell engine to power its spell checking. Libre can integrate Nepali spell checking by adding an extension. These extensions are often called OpenOffice Extensions as they add OpenOffice features into Libre. The extensions are of the file format *.oxt*which on unzipping produces both the *.aff* and *.dic* files.

Hence, the upgraded resource files from the project were used to restructure the extension, which on use provided the same functionality as the Spell Checker in Libre.

[8] ***re*** is used to find certain patterns in a string of text, and easily iterate them.
[9] *nepalispellcheck@gmail.com*

Testing and Evaluation

After modifying the resource files with added wordsand rules, tests were carried out to measure the increase in efficiency of the application. A set of 10 Nepali texts were selected for the testing process. These text consisted minimal errors in spellings. These were then copied one by one into the spell checker while using the older *.aff*and *.dic* files. The total number of errors was counted by making a small modification to the main file that allowed it to print out the total words with incorrect spelling. The same process was then applied with the new set of *.aff*and *.dic* files. The outcome has been shown in the table below:

File No.	Total Word Count	Errors using old set	Errors using improved set	Difference in number of errors	Improvement in accuracy (in %)
1	629	123	90	33	**26.83**
2	1252	237	191	46	**19.41**
3	668	133	104	29	**21.80**
4	706	79	62	17	**21.52**
5	643	114	105	9	**7.89**
6	509	89	73	16	**17.98**
7	631	78	55	23	**29.49**
8	705	117	90	27	**23.08**
9	890	120	78	42	**35.00**
10	1238	231	100	131	**56.71**
Total	**7871**	**1321**	**948**	**373**	**259.71**
Average	**787.1**	**132.10**	**94.80**	**37.30**	**25.97**

The value for Improvement in accuracy was obtained using the following formula:

$$\textit{Improvement in Accuracy} = \frac{\textit{Difference in number of errors}}{\textit{Errors using old set}} \textit{ X } 100\%$$

Hence, from a sample of 10 text files with a total word count of 7871 words, it was found that on an average the new version was 25.97% more accurate than the older version. In simpler words, the newer version found made 25.97% less incorrect decisions than the older one.

Conclusion

Although several improvements were made to the resource files, it is yet not flawless. A perfect spell checker is an important tool for Nepali language processing. However improved the new resource files are there are yet many more words to be added to the dictionary and many more rules to be modified in the affix file. Only then will the efficiency of the spell checker close towards perfection.

The Spell Checker as well, can go through many more improvements. As all the tools and used for the creation of the Spell Checker are open source, developers are encouraged to reformat the code structure in order to increase efficiency of the software as well.

This is a small step towards localizing every aspect of our digital lives. Spell Checkers such as these will certainly encourage more users to start typing in Nepali. Maybe someday digital communication might no longer be awkward for native speakers and the people of our nation might start taking pride in our national language in a complete sense.