

Indic to English Transliterator Programmer Manual

Pravin Paratey

July 17, 2008

1 Overview

The Transliterator module has been implemented in python. This makes it portable between operating systems and the module can be used without any modifications.

Our demo system contains the modules as shown in Figure (1). The system has been built to be easily extensible and this chapter will go through all parts of the code and also demo how one can easily extend (and not modify) this system to add Bengali support.

2 Organization of code

The folder structure is as,

- **db** - Contains the dictionary and learner databases.
 - dict.txt - Contains a dictionary. This will become redundant once the Lucene dictionary is used.
 - learner(s).txt - These are multiple learner files (if unmerged) which are the results of multiple training sessions.
- **models** - Contains various models for transliteration. These models are described later on in this chapter.
- **processor** - Contains various post-processors.

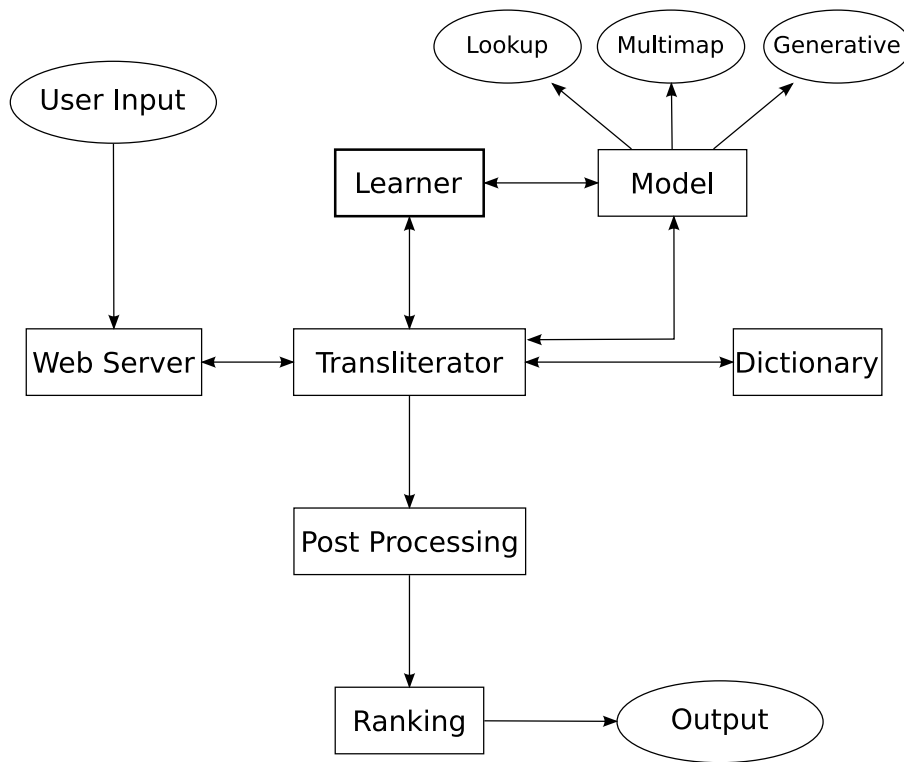


Figure 1: Block diagram for our transliterator implementation

- **scripts** - Contains helper scripts which include a Windows setup program generator, various test scripts for each approach and a script for manipulating parallel corpora.
- **SysTray** - Contains various files to display an interactive system tray icon in Windows.
- **WebRoot** - Contains HTML, CSS and Javascript files in addition to Images. They form part of the user interface of our demo.
- **Files**: DictionaryLookup, Learner, Main, MakeModel, MyError, Setup, SpellChecker, TestAll, TextAlignment, Transliterator, WebServer. These files are modules and are described in the next section.

3 Description of modules

3.1 Transliterator

Given a source string, the transliterator module is responsible for outputting the transliteration. The transliterator module needs to be initialized with a model. This module has two public functions:

- `init()` - Constructor.
- `transliterate(source-word)` - Transliterate function.

A typical example of using the transliterator is,

```
#!/usr/bin/env python

import Transliterator
from models import genmodel

model = genmodel.GenMap() # Using generative model
t = Transliterator(model) # Initialize transliterator with model
print t.transliterate(devanagari-string)
```

3.2 Model

A model is an abstract class which lets us implement transliteration models. Currently, the following models have been created:

- **Lookup** - The naive lookup model.
- **Multimap** - The multimap (one-to-many) lookup model.
- **GenModel** - A generative (statistical) model.

An example of using the multimap model is shown,

```
#!/usr/bin/env python

import Transliterator
from models import multimap
```

```
model = multimap.Multimap() # Using multimap model
t = Transliterator(model) # Initialize transliterator with model
print t.transliterate(devanagari-string)
```

3.3 Learner

This is the main module which illustrates our algorithm. This module contains the following public methods,

- `init()` - Constructor.
- `Load(filename)` - Loads the trained database from *filename*. Multiple calls to this function can be used to load multiple databases which append to each other. This can be used to save the results of training sessions incrementally or to merge data from training sessions from different machines.
- `Save(filename)` - Saves the overall results to *filename*. This is used to save the data to another location apart from the default.
- `SaveIncremental()` - Saves the results of the current training session. It only saves incremental data. The file names are auto-generated
- `Learn(source-word, target-word)` - Learns the relationship between the *source-word* and the *target-word*. The *source-word* and the *target-word* can be given in either the syllabified form or the non-syllabified form. One can also give either of those words as syllabified. The system learns most if both *source-word* and *target-word* are given in the syllabified form. Even otherwise, learning occurs.
- `Transliterate(word)` - This function returns the Devanagari transliteration for the given *word*. This same function for a different learning data set can return the transliteration for other languages too, for example, Bengali.

A typical usage of the Learner module is,

```
#!/usr/bin/env python
from Learner import Learner
```

```
l = Learner()
# Assume learner.txt already contained trained data
l.Load('learner.txt')
l.learn(word-in-lang-1, word-in-lang-2) # learn new words
# The following will print (transliteration, score) pairs
print l.transliterate(word)
```

3.4 DictionaryLookup

This module is responsible for maintaining a dictionary. When integrated with CLIA, the Lucene indexer will replace this module and will provide dictionary functions. The public methods of this module are,

- `init(file)` - Constructor. The *file* contains the words in the dictionary separated by line breaks.
- `Contains(word)` - Returns true if the *word* is found in the dictionary. False otherwise.
- `Add(word)` - Adds the *word* in the dictionary.

3.5 SpellChecker

This module does Edit-2 distance spell checking. It requires access to a dictionary. The exposed functions are,

- `init(dictionary-file)` - *dictionary-file* contains the words of the dictionary.
- `correct(word)` - Returns a list of possible correct spellings for the given *word*.

3.6 MakeModel

This module generates the *GenModel* which is a generative model which makes use of a statistical model. This module requires parallel corpora in the source and target languages. The functions exposed by this module are,

- `train(source-file, target-file)` - This is responsible for training the purely statistical model.

This module uses the `TextAlignment` module to perform alignment. The `TextAlignment` module requires parallel corpora in the source and target languages.

3.7 MyError

This is a custom error module for transliterator errors. This is meant for internal use only. One would use this module like,

```
#!/usr/bin/env python
from MyError import MyError
try:
    # Something
    raise MyError(102, "A custom error occured")
except MyError:
    # Do something on MyError
```

4 Helper modules

These modules have been implemented to show a nice demo. They do not have anything to do with the transliteration problem.

4.1 Webserver

This module implements a webserver. The server implements the `HTTP/1.1` specification. This server can be configured to allow connections from multiple machines and therefore have multiple users training our database. Or it can accept connections from only the client machine.

The folder `WebRoot` contains files which can be customized for demoing in a particular language.

4.2 SysTray

This module implements the windows system tray icon for easy operation. For non-windows systems, this tray icon is absent.

5 Learning by adjusting weights

In our demo system, if a transliteration is deemed incorrect, the user can enter the correct transliteration and the system will adjust its weights. The system also tries to auto-syllabify the input and presents it to the user who can correct it if it's incorrect. Thus the system learns.

6 Extending transliterator for other languages

In this chapter, we'll look at how to extend the transliterator for other Indian languages. We will illustrate this by doing it for Bengali.

6.1 Step 1 - Building a multimap

This step requires linguistic knowledge of Bengali. We create a one-to-many mapping from each Devanagari character to English character(s). This requires editing the `Learner.py` file and adding entries to the *base-table* data structure. Some entries in that table (for Bengali) are as follows:

```
class Learner:
    base_table = {
        u'\u0981': ['n', 'm'], # chandrabinu
        u'\u0982': ['n' 'm', 'un'], # anusvara
        ...
        u'\u0990': ['ai'],
        u'\u0993': ['o'],
        u'\u0994': ['au'],
        u'\u0995': ['k', 'ka'],
        u'\u0996': ['kha', 'kh'],
        ...
    }
```

Note that this effect can also be got **without editing** this file but by adding another file as a **partial class** and then overriding the *base-table*.

6.2 Step 2 - Training the system

For training, we require parallel corpora in Bengali and English. A possible resource can be <http://www.eci.gov.in/DevForum/Dictionary.asp>. Af-

ter downloading the names list for Bengali-English names, the script `make-parallel-corpora.py` (found in the scripts folder) has to be run to generate a corpus that our system can easily make use of.

To make the system better, a person with a working knowledge on Bengali can syllabify the corpus. The more the syllabified data, the more the accuracy of the system.

That done, use the following code for the system to learn and update its database:

```
#!/usr/bin/env python
from Learner import Learner
from os import sep
l = Learner()

f1 = codecs.open("bengali-syllabified.txt", "r", "utf-8")
for line in f1:
    words = line.split(',')
    try:
        l.Learn(words[1].strip(), words[0].strip())
    except MyError:
        print line.encode('utf-8')
f1.close()

l.Save("bengali.txt")
```

6.3 Step 3 - Changing UI

The last step is to change the User Interface for Bengali transliteration. All the files to be modified are present in the `WebRoot` folder and a person with HTML/Javascript experience can trivially change it to represent Bengali.

6.4 Step 4 - Done

Compile the transliterator by running `python setup.py py2exe` and we are done! Run the transliterator by double clicking on the exe and our application will now support Bengali! As we saw, it is really straightforward to extend this for other languages and no changes to core files were required.