Technical Specification

Project Name: Automatic Transliteration between English and Russian

Date Finished: 22-5-2017

Author: Kieron Drumm (13314446)

Supervisor: Prof. Qun Liu

Table of Contents

Abstract

Normally when one thinks about converting some text from one language to another, they instantly assume that the text is being translated. However, sometimes simply rewriting the text while maintaining its pronunciation can be extremely useful, especially to people like teachers or language learners. The aim of this project is to develop a system that can accurately transliterate proper nouns from English to Russian. By transliterate I mean, to write a letter or word from one language using the closest corresponding letters from another alphabet. In this case, the system is given a word written in the English Roman alphabet as input, and it outputs a word written in the Russian Cyrillic alphabet.

Motivation

Transliteration from one language to another can sometimes prove to be very useful as it can help someone to figure out the pronunciation of a word. The main motivation for this project was to create a system that could help language learners. Specifically Russian and English language learners. For example consider this scenario. A native Russian speaker learning English comes across a word with which they are not familiar. As a result, they are unsure as to how said word should be pronounced. With this system, they can simply enter in the English word they have come across, transliterate it into the Cyrillic alphabet with which they would be familiar and thus gain an insight into how that word should be pronounced. I believe that many scenarios such as this may occur, in which my application will prove to be very useful to language learners, especially beginner's level learners.

Automated translation and automated transliteration both rely heavily on machine learning nowadays. As I am greatly interested in both machine learning and languages, it seemed only logical to do a final year project on something based around either translation or transliteration. This interest proved to be another motivation for me when deciding on a project.

Research

Developing a graphical user interface with Tkinter

As this application is a desktop based application written in Python, after some research on graphical libraries, I found that Tkinter was the most used library by far. After making the decision to develop my graphical user interface with Tkinter, I spent some time learning about the basics of using Tkinter. This research was done by reading online blogs from python developers who had used Tkinter at some point, and by watching online tutorials explaining many of the basic concepts of Tkinter (widgets, frames, windows, etc…).

Training a transliteration system using Moses

As I had been learning about the Moses system in a college module about statistical machine translation, it seemed as if Moses would be a good choice with regards to training a transliteration system. In terms of research, much of my time was spent reading the Moses documentation on the official website in order to ascertain how to correctly install Moses on my own computer. I also spent a great deal of time researching how to train language and translation models using Moses.

Learning how to conduct data scraping

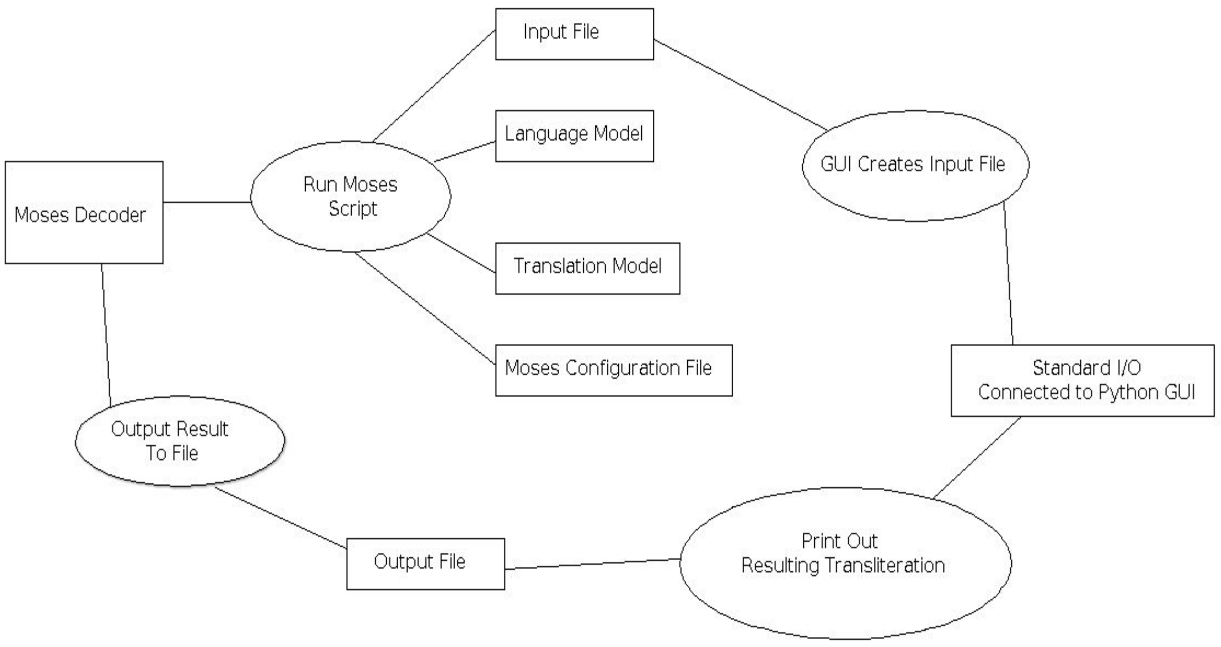
In order to correctly train a transliteration system, I was going to need large volumes of parallel training data with which to train my language and translation models. As such large volumes of data did not exist all in one place online, I decided that the best course of action would be to use the Scrapy framework in Python to scrape any data I could find from the internet. I would then consolidate this data into a single parallel corpus with which I could train my system. To accomplish this task I spent some time researching how to employ the Scrapy framework.

Creating a runnable executable

In order to make it easier for a user to run my system, I decided that the best course of action would be to create a runnable executable of my program that a user could simply double-click in order to run my application. I spent some time researching various tools for this task. In my search I came across tools such as cx_freeze, pyinstaller and py2exe, however I eventually decided on using the nuitka command line tool as I found it to be far simpler and easier to use.

Design

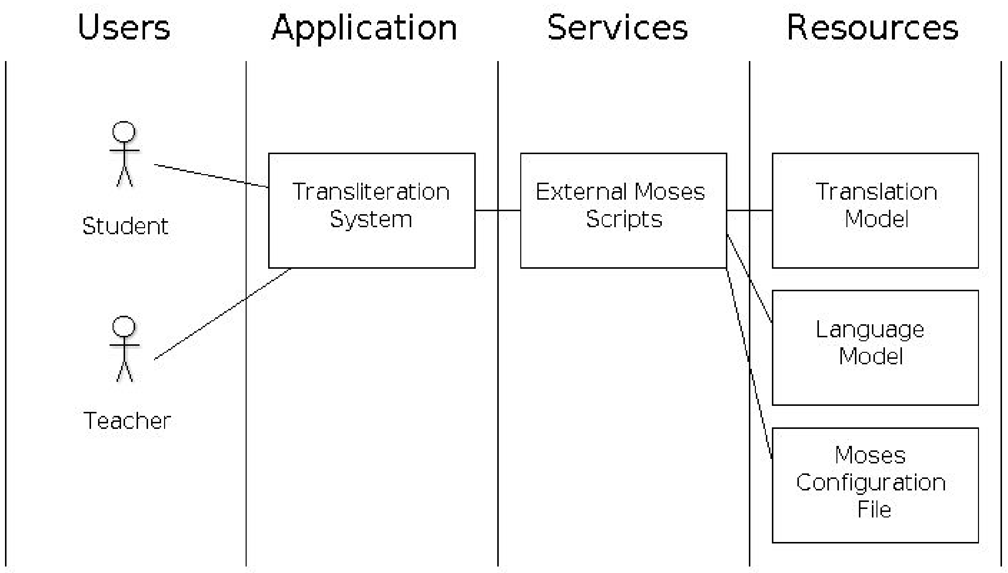
Data flow diagram



This data flow diagram describes all of the main functions of the system, along with how data flows through the system during each of these processes. The four main processes of the system are:

- GUI Creates Input File - This process involves the user entering inputting the message that they wish to transliterate into a textbox. Once they either press the enter key or press the ‘Transliterate’ button, the system generates an input file called ‘input.ro’ that will later be used to carry out the required transliteration.
- Run Moses Script - This process immediately follows the previous process and involves query the Moses system with the required files for a transliteration. These are the language and translation models, the aforementioned input file and a pre written Moses configuration file (moses.ini).
- Output Result To File - Once Moses has generated a suitable transliteration using the models supplied, the result is written into a file by one of my own scripts (user_interface.py).
- Print Out Resulting Transliteration - My program then reads in the results of the transliteration from the output file and outputs it to the user.

System architecture diagram



This system architecture diagram conceptually splits my system into four parts. The four conceptual components of my system are as follow:

- Users - The two primary target users for this application are teacher, more specifically those who teach either Russian or English; and students who are learning either English or Russian.
- Application - This system has one main component, the graphical user interface with which the user interacts in order to carry out transliterations.
- Services - This system uses some a script from the external system “Moses” in order to carry out transliterations. The main script that it uses is the “moses” script which acts as the decoder in the machine translation pipeline in partnership with models of my own creation.
- Resources - The main resources used by the external Moses system are files created by me, namely the translation model , language model, and the moses configuration file (moses.ini). Note that in the case of Moses.ini, it was not written from scratch but rather rewritten in places in order to tailor it to my own needs. For example, altering it to use a binarized language model rather than a standard ascii model.

Implementation

Graphical user interface

The user interface was implemented in Tkinter as a set of Frame objects connected together as children of a single Tk window object. I used frames to implement the interface as they have a similar function to Div's in HTML in that they can be used to position sections of an interface relative to other sections. Each Frame encapsulates a piece of the interface, for example, the main header, textbox area and button area. The TK window also contained a dropdown menu that contained entries linking to two other windows.

Initially, the user interface was comprised of three separate Tk object instances each representing a window. These were the main window in which all transliterations take place, the about window which contained information about the application such as the name and contact information for the developer and the current version number; and lastly the tutorial window which contained a set of instructions detailing how to use the application.

This implementation soon changed and the about window was replaced entirely with a small body of text at the bottom of the main window. This change was made in accordance with some feedback received during the user testing phase.

Transliteration system

In order to implement a transliteration system, I chose to apply the principles of statistical machine translation. To be more specific I chose to use the "noisy channel model" in order to implement my system, The noisy channel model is a framework that can be used to implement systems that contain some form of spell checking, speech recognition or machine translation. In my case, the overall pipeline for calculating the most likely transliteration of a word or words had three components. The language model, the translation model and the decoder. The translation model was used in order to generate a set of possible transliterations, the language model was then used in order to rate the fluency of each of these sentence, and lastly the decoder then searched through this set of potential transliterations and returned the most likely transliteration overall. The aforementioned pipeline can be represented as follows:

$$\widehat{e} = \mathrm{argmax}(e)P(e)P(f|e)$$

In the equation above P(f|e) represents the translation model, P(e) represents the language model and argmax(e) represents the decoder.

In order to train both a translation and language model, I used large amounts of parallel data (I had collected this data previously through or data scraping, and through creating my own parallel data with parallel_data_creator.py). I used this parallel data along with the Moses system in order to train both of my models. In terms of gram size I decided to train both a trigram language model and a trigram translation model. I decided upon using trigrams as they provided more context than bigrams while still being a small enough model that the overall efficiency of the system would not be affected. Whereas a 4-gram model may have been too large for the system to manage during runtime.

Integrating the transliteration system and the user interface

Once both the user interface and transliteration system had been implemented, it came time to integrate the two systems together in order to make the finished product, an interactive transliteration system for the standard computer user. In order to query my system to find a suitable transliteration for a given input I was going to have to use Moses decoder script. I would send in any required models and in turn would receive a transliteration that my system calculated as being the most suitable result. This result would then be displayed to the user via the user interface.

In order to run this external script, send in any required input, and receive an output that could then be displayed to the user, I decided to make use of Python's "subprocess" module. This is a module in Python that allows one to create a subprocess in which an external bash command can be run. Using subprocess I ran a simple script of my own creation that in turn ran Moses, supplying any required input (input sentence, models, configuration files). I was also able to store any output from running this command which made it possible to display the resulting transliteration to the user.

By binding the running of this external command to the clicking of a button in the user interface, I was able to effectively integrate the user interface and transliteration system.

Sample Code

```
def transliterate(self, event):
    if(self.english_roman_entry.get() != ''):
      self.current_progress.set('Transliterating...')
      self.master_window.update_idletasks()
      self.createInputFile()
      process = subprocess.Popen(['sh', './run_moses', 'input.ro', 'output.cy'], stdout =
subprocess.PIPE)
      command_output, errors = process.communicate()
      self.progress_label.configure(text = '')
      output_string = self.getOutput()
      self.russian_cyrillic_entry.delete(0, END)
      self.russian_cyrillic_entry.insert(0, output_string)
      self.current_progress.set('Transliteration Complete')
    else:
      self.current_progress.set('Must provide an English sentence')
def createInputFile(self):
    input_file = open('input.ro', 'w+')
    input_words = self.english_roman_entry.get().split(' ')
    for word in input_words:
      input_file.write(word.replace('', ' ').lower() + '\n')
    input_file.close()

def getOutput(self):
    output_file = open('output.cy', 'r+')
    output_words = output_file.readlines()
    output_string = ''
    for word in output_words:
      output_string += word.replace(' ', '').replace('\n', '') + ' '
    output_file.close()
    return output_string
```

This section of the code for my user interface is the piece code that I believe I am most proud of. This piece of code is present in the ‘user_interface.py’ file and it is executed when the user presses the ‘transliterate’ button. The transliterate() function first displays a message to the user indicating that the transliteration process has begun. It then runs an external shell script entitled ‘run_moses.sh’ which subsequently runs moses with the arguments ‘input.ro’ and ‘output.cy’. Lastly it will place the resulting transliteration in the second textbox and then display a message to the user indicating that the transliteration has complete. It should be noted that the input and output are placed into files in order to make them easier to deal with when calling Moses’ external script.

Problems Solved

Executing Bash commands from within my interface

One problem that I encountered during development involved finding a way to execute a Bash file from within a Python program, and to then have the Shell script send back any results to the Python program after executing. I found a way to do this by using the ‘subprocess’ module in Python. This module allows one to create new processes from a within a Python program, as well as handling any input or output from these processes. Using this module I was able to call a moses shell script from my user interface, and then display any important information resulting from the execution of that Shell script.

Gathering suitable training data

Sometime into my data scraping, I found that it was very difficult to find parallel Roman-Cyrillic data that was transliterated but not translated. After spending a great deal of time scouring the internet, I eventually decided that if I could not find sufficient training data, then I would create it instead. In order to ensure that the training data I created was satisfactory and usable for training my system, I used Python’s ‘transliterate’ module which contained a function that would allow me to provide English text as an input and receive correctly transliterated Cyrillic text as an output.

Processing the Cyrillic alphabet in Python

Another problem that I encounter during development occurred during the phase in which I was scraping, cleaning and tokenizing data in order to form a suitable parallel corpus for later training. This process involved me writing a few scripts to carry out these tasks, such as ‘remove_punctiation.py’, ‘tokenize_by_character.py’ and ‘parallel_data_creator.py’. However, I encountered some problems when trying to apply these changes to any files containing Cyrillic text as they were encoded differently to the Roman letters I was accustomed to working with. I managed to solve this problem by ensuring that I called the encode() function on any Cyrillic string being worked with in order to convert it to unicode. I also found that writing these script to be run by python3 rather than python solved some other encoding related runtime errors.

Optimising language and translation model load times

After I completed the first prototype of my system, I noticed that the load times for both my language and translation models were extremely long. For example, the phrase table that I was using took upwards of 22 seconds to load into memory before being available for Moses to use. After reading over the Moses documentation further, along with some online blogs from other Moses users, I was able to devise a partial solution to this problem. I discovered that it was possible to binarize a language model in order to make loading said model into memory much quicker. With the use of IRSTLM (IRST Language Modelling Toolkit) and it's 'compile-lm' script I was able to successfully binarize my language model. However at the moment I have unable to compress my phrase table into a compact phrase table. As a result doing so would most definitely be a task that I would hope to accomplish in the near future.

Making all training data suitable for Moses

After conducting some data scraping, and after creating some parallel training data myself with the use of Python's 'transliterate' module, I was faced with yet another problem. Converting that data into a suitable format with which Moses could work. To do so I wrote some scripts in Python to tokenize (tokenize.py) and remove any unnecessary punctuation (remove_punctuation.py). These scripts once run were able to successfully transform my data. I was then able to use that data to train my system with the help of Moses.

Results

Implementation Results

In terms of implementation, I feel that I have accomplished everything that I set out to accomplish in relation to my system. I have successfully developed a system that can transliterate proper nouns from the English Roman alphabet to the Russian Cyrillic alphabet.

Testing Results

Automated Testing

In order to ensure that my application was correctly tested, I carried out two rounds of tests using two different methods to measure the accuracy of my system. The first measure I used involved writing my own script called 'calculate_accuracy_py'. I used that script to automatically compare my own output with Python's 'transliteration' module's output with regards to the same set of testing data. My results for this method of testing were as follows. In the first round of testing, I used some training data that was within the some domain as the data with which I had trained my system. The resultant accuracy from this test was 0.9947. In the second round of testing, using data from a different domain the resultant accuracy was 0.8731.

The second measure that I used to estimate the accuracy of my system was the Bleu score. I calculated the Bleu score for my system using Moses' 'multi-bleu.perl' script. As with my first measure I carried out two rounds of testing, the first with data from the same domain as my training data, the second with data from a different domain. The results were as follows. For the first round of testing my system received a Bleu score of 0.9971 and for the second round of testing it received a Bleu score of 0.9802.

User Testing

After carrying out automated testing, I then proceeded to carry out some user testing. After filling out the necessary paperwork and receiving permission from the Dublin City University ethics committee, I enlisted the help of some of my classmates in order to carry out this testing. The focus of this testing was to identify any problems with the interface that made it more difficult or less intuitive to use. After the user testing phase, some of the changes made to the interface were as follows:

- The user was now about to initiate the transliteration process by pressing the enter key rather than just pressing the “Transliterate” button.
- The 'about' window was removed and replaced with a smaller piece of text in the main window that provided the same information to the user. Namely the name and contact information of the developer, and the current version number of this application.
- When the main window was closed, any child windows were now also closed whereas previously they had remained open and had to be closed manually.
- The tutorial window was resized in order to make to make it easier to read through the instructions provided in said window.
- A shortcut (ctrl-c) was to to clear both text boxes in the main window whereas before one could accomplish this by pressing the “Clear” button in the main window.
- A message was added to indicate that the transliteration process had ended whereas before a message was only displayed to show that the process had begun.

I believe that carrying out user testing proved to be extremely useful as it allowed to me to make changes that in turn created a user interface that was altogether more intuitive and more suitable to any potential users.

It should be noted that current video walkthrough does not contain these small changes as the user testing was carried out shortly after the creation of the video walkthrough and very close to the overall project deadline. As a result it seemed prudent to prioritise changing the actual demoable software rather than the video walkthrough.

Future Work

Shift to NMT (neural machine translation)

In the last few years the application of neural networks and their use in machine translation has become increasingly popular, in fact NMT (neural machine translation) is now seen as the cutting edge way of training a translation system. As I have both an interest and good fundamental knowledge regarding the inner workings of a neural network, I would be very interested in developing a brand new neural network based transliteration system in order to observe increases/decreases in speed and quality that could arise from a neural networking approach. Such information could later influence me to replace the statistical system I am currently using entirely.

Optimising phrase table load time

Due to time constraints I was unable to successfully create a compacted phrase table. As a future goal, I would be interested in using the CMPH (C Minimal Perfect Hashing Library) to create a compact phrase table in order to observe whether or not it would have any significant improvement on the phrase table's load time.

Running the system from a server with a website frontend

As there are many factors that must be taken into account when developing a desktop application such as the target operating system, the dependencies needed by any user who wishes to use the application, I would be interested in trying to put my system onto a server. I would then like to create a simple webpage to act as a frontend to the system. This idea had occurred to me during development but due to my having to run external shell scripts, there were a few security issues that I was unable to account for due to my not being an administrative user on the DCU servers.